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

November 26, 2019

Summary: This document specifies 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

Table of Contents

Introduction

Look-Up Tables (LUTs) are a common implementation for transformations from one set of color values to another. 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. This can create workflow problems when a LUT being used on a production is not supported by one or more of the applications being used. Furthermore, many LUT formats are designed for a particular use case only and lack the quality, flexibility, and metadata needed to meet modern requirements.

The Common LUT Format (CLF) can communicate an arbitrary chain of color operators (also called processing nodes) which are sequentially processed to achieve an end result. The set of available operator types includes matrices, 1D LUTs, 3D LUTs, ASC-CDL, log and exponential shaper functions, and more. Even when 1D or 3D LUTs are not present, CLF can be used to encapsulate any supported color transforms as a text file conforming to the XML schema.

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."

The document defines what is a valid CLF file. Though it is not intended as a tutorial for users to create their own files, LUT creators will find it useful to understand the elements and attributes available for use in a CLF file. The document is also not intended to provide guidance to implementors on how to optimize their implementations, but does provide a few notes on the subject.

This document assumes the reader has knowledge of basic color transformation operators and XML.

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

ASC Color Decision List (ASC CDL) Transfer Functions and Interchange Syntax. ASC-CDL_Release1.2. Joshua Pines and David Reisner. 2009-05-04.

3 Specification

A file conforming to the Common LUT Format (a ”CLF”) is stored in a text file and adheres to 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 individual 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 operations 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.

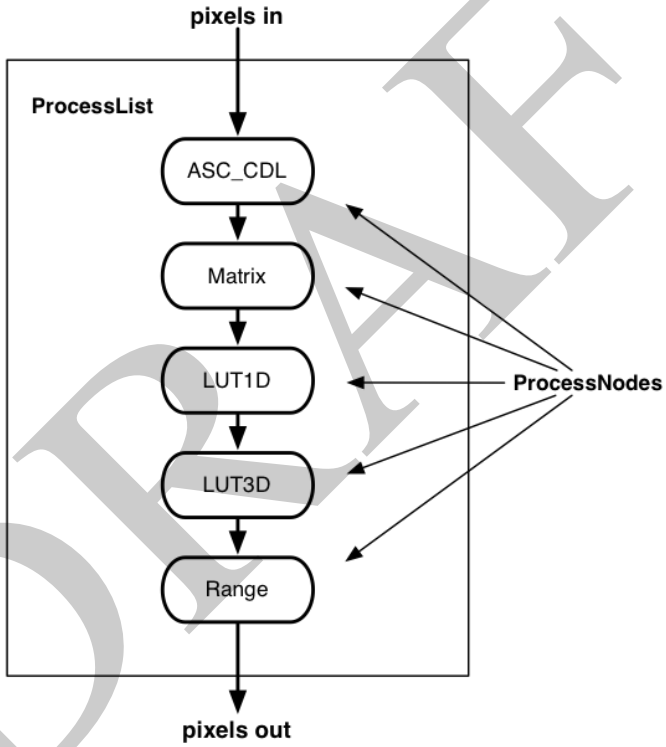

Figure 1 – Example of a Process List containing a sequence of multiple Process Nodes

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 beyond the scope of this document.

Each XML file shall be completely self-contained needing no external information or metadata. The full content of a color transform must be included in each file and a color transform may not be incorporated by reference to another XML LUT file. This restriction ensures that each LUT file can be an independent archival element.

Each `ProcessList` shall be given a unique ID for reference.

The data for LUTs shall be 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 4 for details).

For 3DLUTs, the indexes to the cube are assumed to have regular spacing across the range of input values. To accommodate irregular spacing, a `halfDomain` 1DLUT or Log node should be used as a shaper function prior to the 3DLUT.

For simplicity's sake, the CLF 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.

3.1 XML Structure

3.1.1 General

The XML files shall contain a single occurrence of the XML root element known as the `ProcessList`. The `ProcessList` element shall contain one or more elements known as `ProcessNode`s. The order and number of process nodes is determined by the designer of the XML file.

NOTE: A series of operations may be rolled into a single 3D LUT process node or be maintained as discrete process nodes. The latter is recommended for better clarity of the processing steps, though in practice, a transform engine might preprocess discrete Process Nodes into a single LUT for performance (See Section 5.3).

An example of the overall structure of a simple CLF 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>
```

3.1.2 Declaration

Transforms adhering to the Common LUT Format shall be written to text files using Extensible Markup Language (XML).

The file should contain the following XML declaration:

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

NOTE: The XML declaration indicates the version of XML and type of character encoding used.

3.1.3 XML version and encoding

The XML file shall include 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"?>
```

3.1.4 Comments

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    -->
```

3.1.5 Language

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.

3.1.6 White Space

Particularly when creating CLF files containing certain elements (such as `Array`, `LUT1D`, or `LUT3D`) it is desirable that single lines per entry are maintained so file contents can be scanned more easily by a human reader. There exist some difficulties with maintaining this behavior as XML has some non-specific methods for handling white-space. Especially if files are re-written from an XML parser, white space will not necessarily be maintained. To maintain line layout, XML style sheets may be used for reviewing and checking the CLF file's entries.

3.1.7 Newline Control Characters

Different end of line conventions, including <CR>, <LF>, and <CRLF>, are utilized between Mac, Unix, and PC systems. Different newline characters may result in the collapse of values into one long line of text. To maintain intended linebreaks, CLF specifies that the 'newline' string, i.e. the byte(s) to be interpreted as ending each line of text, shall be the single code value $10_{10} = 0A_{16}$ (ASCII 'Line Feed' character), also indicated <LF>.

NOTE: Parsers of CLF files may choose to interpret Microsoft's <CR><LF> or older-MacOS' <CR> newline conventions, but CLF files should only be generated with the <LF> encoding.

NOTE 2: <LF> is the newline convention native to all *nix operating systems (including Linux and modern macOS).

4 XML Elements

4.1 `Array`

The array element contains a table entries with a single line for each grouping of values. The element is used in the `LUT1D`, `LUT3D`, and `Matrix` `ProcessNode`s. The `dim` attribute specifies the dimensions of the array and, depending on context, defines the size of a matrix or the length of a LUT table. The specific formatting of the `dim` attribute and the type of node should match. The usages are summarized below but specific requirements for each application of `Array` are described when it appears as a child element for a particular `ProcessNode`.

Attributes:

`dim` (required)

Specifies the dimension of the LUT or the matrix and the number of color components. The `dim` attribute provides the dimensionality of the indexes, where:

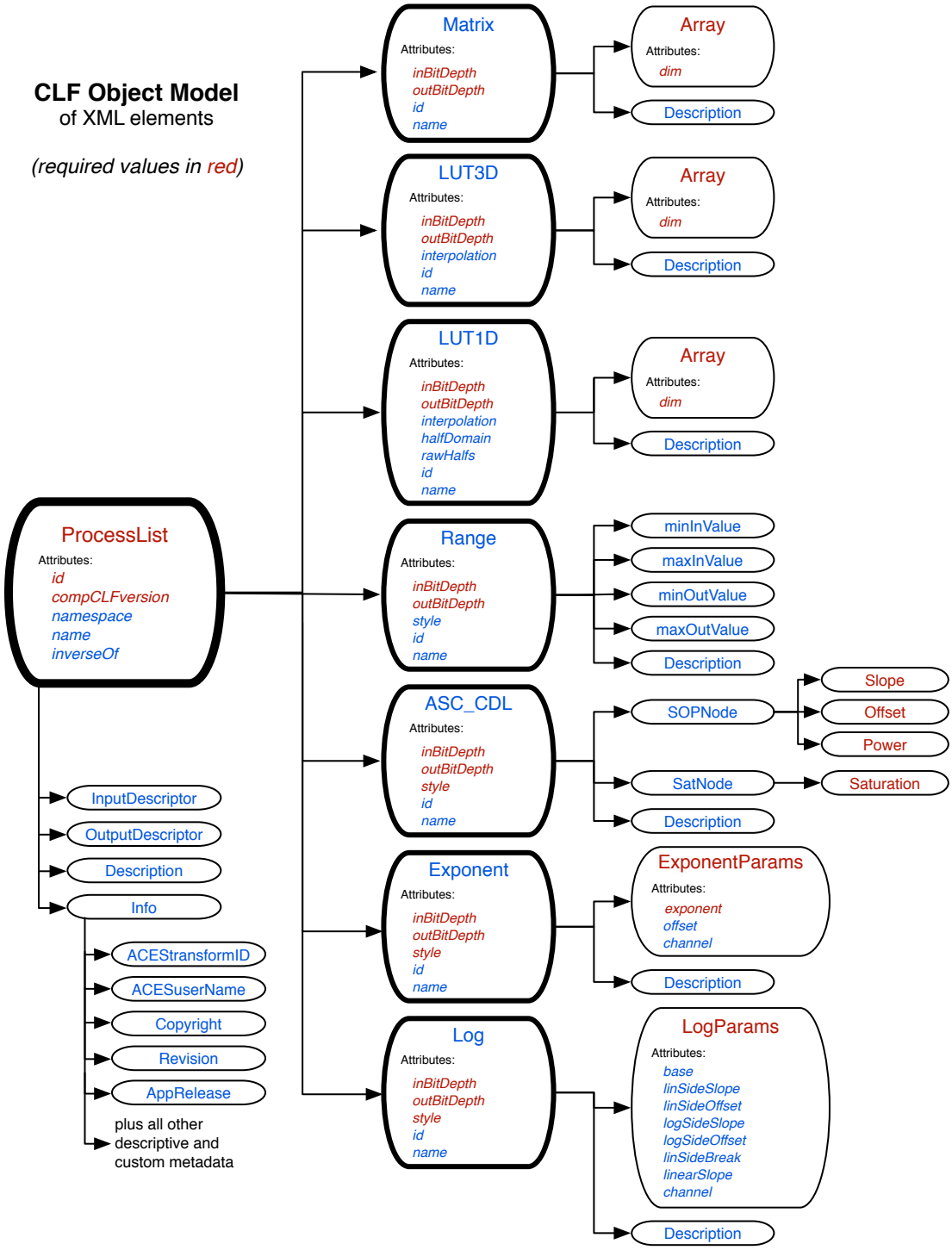

Figure 2 – Object Model of XML Elements

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)

4.2 `ProcessList`

Description:

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

NOTE: 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.

NOTE 2: A `Range` node as the very last node in the `ProcessList` allows specification of minimum and maximum output values and clamping if needed.

Attributes:

`id` (required)
: a string to serve as a unique identifier of the `ProcessList`
NOTE 3: This attribute may be used to contain the `ACEStransformID` which is mandatory for ACES transforms.

`name` (optional)
: a concise string used as a text name of the `ProcessList` for display or selection from an application's user interface

`compCLFversion` (required)
: a string indicating the minimum compatible CLF specification version required to read this file

`inverseOf` (optional)
: a string for linking to another `ProcessList` `id` (unique) which is the inverse of this one

Elements:

`Description` (required)
: a string for comments describing the function, usage, or any notes about the `ProcessList`. A `ProcessList` can have one or more `Description`s.

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

  `AppRelease` (optional)
  : a string used for indicating application software release level

  `Copyright` (optional)
  : a string containing a copyright notice for authorship of the CLF file

`Revision` (optional)
a string used to track the version of the LUT itself (e.g. an increased resolution from a previous version of the LUT)

`ACEStransformID` (optional)
a string containing 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` (optional)
a string containing the user-friendly name recommended for use in product user interfaces as described in Academy TB-2014-002.

`InputDescriptor` (optional)
an arbitrary string used to describe the intended source code values of the `ProcessList`.

`OutputDescriptor` (optional)
an arbitrary string used to describe the intended output target of the `ProcessList` (e.g. target display)

`ProcessNode` (required)
a generic XML element that in practice is substituted with a particular color operator. At least one `ProcessNode` must be in the list. The `ProcessNode` is described in Section 4.3.

4.3 `ProcessNode`

Description: A `ProcessNode` element represents a primary color transformation to be applied to the image data. At least one `ProcessNode` element shall be included in the `ProcessList`. The generic `ProcessNode` element contains attributes and elements that are common to and inherited by the specific sub-types of the `ProcessNode` element that can substitute for `ProcessNode`. All `ProcessNode` substitutes shall inherit the following attributes.

Attributes:

`id` (optional)
a unique identifier for the `ProcessNode`

`name` (optional)
a concise string defining a name for the `ProcessNode` that can be used by an application for display in a user interface

`inBitDepth` (required)
a string indicating the number of bits and data type of the expected input values to the `ProcessNode`. Input values can be either integers or floats.

`outBitDepth` (required)
a string indicating the number of bits and data type of the output values generated by the `ProcessNode`. Output values can be either integers or floats, independent of the input values.

The supported values for both `inBitDepth` and `outBitDepth` are the same:

- "`8i`": 8-bit unsigned integer
- "`10i`": 10-bit unsigned integer
- "`12i`": 12-bit unsigned integer
- "`16i`": 16-bit unsigned integer
- "`16f`": 16-bit floating point (half-float)

- "32f": 32-bit floating point (single precision)

Elements:

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

4.4 Substitutes for ProcessNode

4.4.1 General

The attributes and elements defined for ProcessNode are inherited by the substitutes for ProcessNode. This section defines the available substitutes for the generalized ProcessNode element.

NOTE: Additional substitutes for ProcessNode could be introduced in future versions of this specification.

4.4.2 LUT1D

Description:

The LUT1D element shall contain either a 1D LUT or a 3×1D LUT in the form of an Array.

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 shall be applied to all three color components.

A 3×1D LUT transform looks up each color component in a separate LUT1D of the same length. In a 3×1D LUT, by convention, the LUT1D for the first component goes in the first column of Array. The lookup operation may be altered by redefining the mapping of the input values to index positions of the LUT using an IndexMap.

Linear interpolation shall be used for LUT1D.

Elements:

Array (required)
an array of numeric values that are the output values of the 1D LUT. Array shall contain the table entries of a LUT in order from minimum value to maximum value.

For a 1D LUT, one value per entry is used for all color channels. For a 3×1D LUT, each line should contain 3 values, creating a table where each column defines a 1D LUT for each color component. For RGB, the first column shall correspond to R's 1D LUT, the second column shall correspond to G's 1D LUT, and the third column shall correspond to B's 1D LUT.

Attributes:

dim (required)
two integers that describe the dimensions of the array, representing the length and number of values per entry, respectively. The first value defines the length of the array and shall equal the number of entries actually present in the array. The second value indicates the number of values per entry and shall equal 1 for a 1D LUT or 3 for a 3×1D LUT.

2 entries have the length of the 1D LUT plus the number of components per entry.

e.g. dim = "1024 3" indicates a 1024 element 1D LUT with 3 component color (a 3×1D LUT)

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

NOTE: Array is formatted differently when it is contained in a LUT3D or Matrix element.

`IndexMap` (optional)

a table of 2 values that maps input values to index positions of the `LUT1D`'s `Array` element. An Index Map is used to define the input floating-point range that will be used by the LUT. The format of each item in the Index Map shall be $inValue@n$, where $inValue$ is an input code value and n is the index position.

For an array of integer bit-depth n, the element can be interpreted as $minValue@0$ and $maxValue@(2^n - 1)$.

For example, the index map below which assigns the $inValue = 64$ to position 0 in the LUT and $maxValue = 940$ to position 1023 (the last item) in the LUT.

```
<IndexMap dim=2>64@0 940@1023</IndexMap>
```

Attributes:

`dim` (required)

an integer that indicates the number of items in the list. The only valid value is 2.

Attributes:

`interpolation` (optional)

a string indicating the preferred algorithm used to interpolate values in the `1DLUT`. This attribute is optional but is fixed to `"linear"` as the only allowed value.

NOTE 2: Support for additional interpolation types may be added in a future version of the specification.

`halfDomain` (optional)

If this attribute is present, its value must equal "`true`". When true, the input domain to the node is considered to be all possible 16-bit floating-point values, and there must be exactly 65536 entries in the `Array` element.

NOTE 3: For example, the unsigned integer 15360 has the same bit-pattern (0011110000000000) as the half-float value 1.0, so the 15360th entry (zero-indexed) in the `Array` element is the output value corresponding to an input value of 1.0.

`rawHalfs` (optional)

If this attribute is present, its value must equal "`true`". When true, the `rawHalfs` attribute indicates that the output array values in the form of unsigned 16-bit integers are interpreted as the equivalent bit pattern, half floating-point values.

NOTE 4: For example, to represent the value 1.0, one would use the integer 15360 in the `Array` element because it has the same bit-pattern. This allows the specification of exact half-float values without relying on conversion from decimal text strings.

Examples:

```
<LUT1D id="lut-23" name="4 Value Lut" inBitDepth="12i" outBitDepth="12i">
    <Description> 1D LUT - Turn 4 grey levels into 4 inverted codes </Description>
    <Array dim="4 1">
        3
        2
        1
        0
    </Array>
</LUT1D>
```

Example 1 – Example of a very simple `LUT1D`

4.4.3 LUT3D

Description:

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 preceding the LUT3D element with a LUT1D element.

Attributes:

interpolation (optional)

a string indicating the preferred algorithm used to interpolate values in the 3DLUT. This attribute is optional with a default of "trilinear" if the attribute is not present.

Supported values are:

- "trilinear": perform trilinear interpolation
- "tetrahedral": perform tetrahedral interpolation

NOTE: Interpolation methods are specified in Appendix C.

Elements:

Array (required)

an array of numeric values that are the output values of the 3D LUT. The Array shall contain the table entries for the LUT3D from the minimum to the maximum input values, with the third component index changing fastest.

Attributes:

dim (required)

four integers that describe the dimensions of the 3D LUT. The first three value define the dimensions of the array and if multiplied shall equal the number of entries actually present in the array. The fourth value indicates the number of components per entry.

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

NOTE 2: Array is formatted differently when it is contained in a LUT1D or Matrix element.

IndexMap (optional)

a table of 2 values that maps input values to index positions of the LUT3D's Array. This element is used to define the input floating-point range that will be used by the LUT.

The format of the two items in the list shall be $newMinValue@0$ and $newMaxValue@1$.

Examples:

```
<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 simple LUT3D

4.4.4 `Matrix`

Description:

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 row-order convention (Equation 4.1), which is equivalent in functionality to the equations in 4.2.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} r_1 \\ g_1 \\ b_1 \end{bmatrix} = \begin{bmatrix} r_2 \\ g_2 \\ b_2 \end{bmatrix} \tag{4.1}$$

$$\begin{aligned} r_2 &= (r_1 \cdot a_{11}) + (g_1 \cdot a_{12}) + (b_1 \cdot a_{13}) \\ g_2 &= (r_1 \cdot a_{21}) + (g_1 \cdot a_{22}) + (b_1 \cdot a_{23}) \\ b_2 &= (r_1 \cdot a_{31}) + (g_1 \cdot a_{32}) + (b_1 \cdot a_{33}) \end{aligned} \tag{4.2}$$

Matrices using an offset calculation will have one more column than rows. An offset matrix may be defined using a 3x4 array (4.3), wherein the fourth column is used to specify offset terms, k_1, k_2, k_3, that are added to the result of the normal matrix calculations (4.4).

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & k_1 \\ a_{21} & a_{22} & a_{23} & k_2 \\ a_{31} & a_{32} & a_{33} & k_3 \end{bmatrix} \begin{bmatrix} r_1 \\ g_1 \\ b_1 \\ 1.0 \end{bmatrix} = \begin{bmatrix} r_2 \\ g_2 \\ b_2 \end{bmatrix} \tag{4.3}$$

$$\begin{aligned} r_2 &= (r_1 \cdot a_{11}) + (g_1 \cdot a_{12}) + (b_1 \cdot a_{13}) + k_1 \\ g_2 &= (r_1 \cdot a_{21}) + (g_1 \cdot a_{22}) + (b_1 \cdot a_{23}) + k_2 \\ b_2 &= (r_1 \cdot a_{31}) + (g_1 \cdot a_{32}) + (b_1 \cdot a_{33}) + k_3 \end{aligned} \tag{4.4}$$

Elements:

`Array` (required)
: a table that provides the coefficients of the transformation matrix. The matrix dimensions are either 3-by-3 or 3-by-4. 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 3-by-3 matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{4.5}$$

Attributes:

`dim` (required)
: three integers that describe the dimensions of the matrix. The first two value define the number of

rows and the number of columns in the array, respectively. The third value indicates the number of components per entry.

3 entries have the dimensions of a matrix plus the number of components per entry..

e.g. `dim = "3 3 3"` indicates a 3-by-3 matrix acting on 3 component color

e.g. `dim = "3 4 3"` indicates a 3-by-4 matrix acting on 3 component color

NOTE: `Array` is formatted differently when it is contained in a `LUT1D` or `LUT3D` element.

Examples:

```
<Matrix id="lut-28" name="AP0 to AP1" inBitDepth="16f" outBitDepth="16f" >
    <Description>3x3 color space conversion from AP0 to AP1</Description>
    <Array dim="3 3 3">
         1.45143931614567     -0.236510746893740    -0.214928569251925
        -0.0765537733960204    1.17622969983357     -0.0996759264375522
         0.00831614842569772  -0.00603244979102103   0.997716301365324
    </Array>
</Matrix>
```

Example 3 – Example of a `Matrix` node with `dim="3 3 3"`

```
<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 4 – Example of a `Matrix` node

4.4.5 `Range`

Description:

The `Range` element maps the input domain to the output range by scaling and offsetting values. The `Range` element can also be used to clamp values.

Unless otherwise specified, the node's default behavior is to scale and offset without clamping. If clamping is required, the `style` attribute can be set to `"Clamp"`.

To achieve scale and/or offset of values, all of `minInValue`, `minOutValue`, `maxInValue`, and `maxOutValue` must be present. In this explicit case, the formula for `Range` shall be as in Equation 4.6. 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.

$$out = in \times scale + \texttt{minOutValue} - \texttt{minInValue} \times scale \tag{4.6}$$

If `style="Clamp"`, the output value from Eq 4.6 is furthur modified via Eq. 4.7.

$$out_{clamped} = \text{MIN}(\texttt{maxOutValue}, \text{MAX}(\texttt{minOutValue}, out)) \tag{4.7}$$

Where:

$$scale = \frac{(\texttt{maxOutValue} - \texttt{minOutValue})}{(\texttt{maxInValue} - \texttt{minInValue})}$$

MAX(a, b) returns a if $a > b$ and b if $b \geq a$

MIN(a, b) returns a if $a < b$ and b if $b \leq a$

The `Range` element can also be used in to clamp values. In such instances, no offset applied, and so the formula simplifies since only one pair of min or max values are required. If only the minimum value pair is provided, then the result shall be clamping at the low end, according to Equation 4.8.

$$out = \text{MAX}(\texttt{minOutValue}, in \times bitDepthScale) \tag{4.8}$$

If only the maximum values pairs are provided, the result shall be clamping at the high end, according to Equation 4.9.

$$out = \text{MIN}(\texttt{maxOutValue}, in \times bitDepthScale) \tag{4.9}$$

Where:

$$bitDepthScale = \frac{\text{SIZE}(\texttt{outBitDepth})}{\text{SIZE}(\texttt{inBitDepth})}$$

and where:

$$\text{SIZE}(a) = \begin{cases} 2^{bitDepth} - 1 & \text{when } a \in \{\texttt{"8i"}, \texttt{"10i"}, \texttt{"12i"}, \texttt{"16i"}\} \\ 1.0 & \text{when } a \in \{\texttt{"16f"}, \texttt{"32f"}\} \end{cases}$$

In both instances, values must be set such that `maxOutValue` $=$ `maxInValue` $\times\ bitDepthScale$.

NOTE: The bit depth scale factor intentionally uses $2^{bitDepth} - 1$ and not $2^{bitDepth}$. This means that the scale factor created for scaling between different bit depths is "non-integer" and is slightly different depending on the bit depths being scaled between. While intinct might be that this scale should be a clean bit-shift factor (i.e. 2x or 4x scale), testing with a few example values plugged into the formula will show that the resulting non-integer scale is the correct and intended behavior.

If none of `minInValue`, `minOutValue`, `maxInValue`, or `maxOutValue` are present, then the `Range` operator performs only bit-depth conversion.

Elements:

`minInValue` (optional)
The minimum input value. Required if `minOutValue` is present.

`maxInValue` (optional)
The maximum input value. Required if `maxOutValue` is present.

`minOutValue` (optional)
The minimum output value. Required if `minInValue` is present.

`maxOutValue` (optional)
The maximum output value. Required if `maxInValue` is present.

Attributes:

`style` (optional)
Describes the preferred handling of the scaling calculation of the `Range` node. If the style attribute is not present, clamping is performed.

`"noClamp"`
If present, scale and offset is applied without clamping, as in Eq. 4.6.
(i.e. values below `minOutValue` or above `maxOutValue` are preserved)

"Clamp"
If present, clamping is applied according to Eq. 4.7 upon the result of the scale and offset expressed in Eq. 4.6

Examples:

```
<Range inBitDepth="10i" outBitDepth="10i">
    <Description>10-bit full range to SMPTE range</Description>
    <minInValue>0</minInValue>
    <maxInValue>1023</minInValue>
    <minOutValue>64</minInValue>
    <maxOutValue>940</minInValue>
</Range>
```

Example 5 – Using "Range" for scaling 10-bit full range to 10-bit SMPTE (legal) range.

4.4.6 Log

Description:

The Log element contains parameters for processing pixels through a logarithmic or anti-logarithmic function. A couple of main formulations are supported. The most basic formula follows a pure logarithm or anti-logarithm of either base 2 or base 10. Another supported formula allows for a logarithmic function with a gain factor and offset. This formulation can be used to convert from linear to Cineon. Another style of log formula follows a piece-wise function consisting of a logarithmic function with a gain factor and offset and a linear segment. This style can be used to implement many common "camera-log" encodings.

NOTE: The equations for the Log node assume integer data are normalized to floating-point scaling. LogParams do not change based on the input and output bit-depths.

NOTE 2: On occasion it may be necessary to transform a logarithmic function specified in terms of traditional Cineon-style parameters to the parameters used by CLF. Guidance on how to do this is provided in Appendix D.

Attributes:

style (required)
specifies the form of the of log function to be applied

Supported values are:

- "log10": applies a base 10 logarithm according to

$$y = \log_{10}(x) \tag{4.10}$$

- "antiLog10": applies a base 10 anti-logarithm according to

$$x = 10^{y} \tag{4.11}$$

- "log2": applies a base 2 logarithm according to

$$y = \log_{2}(x) \tag{4.12}$$

- "antiLog2": applies a base 2 anti-logarithm according to

$$x = 2^{y} \tag{4.13}$$

- "linToLog": applies a logarithm according to

$$y = \mathit{logSideSlope} \times \log_{\mathit{base}}(\mathit{linSideSlope} \times x + \mathit{linSideOffset}) + \mathit{logSideOffset} \tag{4.14}$$

- `"logToLin"`: applies an anti-logarithm according to

$$x = \frac{\left(base^{\frac{y - logSideOffset}{logSideSlope}} - linSideOffset\right)}{linSideSlope} \tag{4.15}$$

- `"cameraLinToLog"`: applies a piecewise function with logarithmic and linear segments on linear values, converting them to non-linear values

$$y = \begin{cases} linearSlope \times x + linearOffset & \text{if } x \leq linSideBreak \\ logSideSlope \times \log_{base}(linSideSlope \times x + linSideOffset) + logSideOffset & \text{otherwise} \end{cases} \tag{4.16}$$

where:

linearSlope is calculated using Eq. 4.19
linearOffset is calculated using Eq. 4.20

- `"cameraLogToLin"`: applies a piecewise function with logarithmic and linear segments on non-linear values, converting them to linear values

$$x = \begin{cases} \dfrac{(y - linearOffset)}{linearSlope} & \text{if } y \leq logSideBreak \\ \dfrac{\left(base^{\frac{y - logSideOffset}{logSideSlope}} - linSideOffset\right)}{linSideSlope} & \text{otherwise} \end{cases} \tag{4.17}$$

where:

logSideBreak is calculated using Eq. 4.18
linearSlope is calculated using Eq. 4.19
linearOffset is calculated using Eq. 4.20

Elements:

`LogParams` (required - if `"style"` is not a basic logarithm)
contains the attributes that control the `"linToLog"`, `"logToLin"`, `"cameraLinToLog"`, or `"cameraLogToLin"` functions

This element is required if `style` is of type `"linToLog"`, `"logToLin"`, `"cameraLinToLog"`, or `"cameraLogToLin"`.

`"base"` (optional)
the base of the logarithmic function
Default is 10.

`"logSideSlope"` (optional)
"slope" (or gain) applied to the logarithmic segment of the function.
Default is 1.

`"logSideOffset"` (optional)
offset applied to the logarithmic segment of the function.
Default is 0.

`"linSideSlope"` (optional)
slope of the linear segment of the function.
Default is 1.

`"linSideOffset"` (optional)
offset applied to the linear segment of the function.
Default is 0.

`"linSideBreak"` (optional)
the break-point, defined in linear space at which the piecewise function transitions between the logarithmic and linear segments. This is required if `style="cameraLinToLog"`

`"linearSlope"` (optional)

the slope of the linear segment of the piecewise function. This attribute does not need to be provided unless the formula being implemented requires it. The default is to calculate using `linSideBreak` such that the linear portion is continuous in slope with the logarithmic portion of the curve, by using the derivative of the log portion of the curve at the break-point. This is described in the following note, using Equations 4.18-4.20.

`"linearOffset"` (optional)

the offset of the linear segment of the piecewise function. This attribute does not need to be provided unless the formula being implemented requires it. The default is to calculate it using `linSideBreak` such that the linear portion is continuous in slope with the logarithmic portion of the curve, using the derivative of the log portion of the curve at the break-point. This is described in the following note, using Equations 4.18-4.20.

NOTE 3: First, the value of the break-point on the log-axis is calculated using `linSideBreak` as input to the logarithmic segment of Eq. 4.16 (which is also equivalent to Eq. 4.14), as shown in Eq. 4.18.

$$logSideBreak = logSideSlope \times \log_{base}(linSideSlope \times \boldsymbol{linSideBreak} + linSideOffset) + logSideOffset \quad (4.18)$$

Then, again using the value of `linSideBreak`, but this time as input to the derivative of Eq. 4.14. This yields the instantaneous slope at the break-point, which becomes the value of *linearSlope*. The derivative is shown in Eq. 4.19:

$$linearSlope = logSideSlope \left(\frac{linSideSlope}{(linSideSlope \times \boldsymbol{linSideBreak} + linSideOffset) \times \ln(base)} \right) \quad (4.19)$$

Finally, the value of *linearOffset* can be solved for by rearranging the linear segment of Eq. 4.16 to get Equation 4.20, and using the values of *logSideBreak* (obtained from Eq. 4.18) and *linearSlope* (obtained from Eq. 4.19).

$$linearOffset = \boldsymbol{logSideBreak} - \boldsymbol{linearSlope} \times linSideBreak \quad (4.20)$$

`"channel"` (optional)

the color channel to which the exponential function is applied. Possible values are `"R"`, `"G"`, `"B"`.

If this attribute is not otherwise specified, the exponential function is applied to all color channels.

Examples:

```
<Log inBitDepth="16f" outBitDepth="16f" style="log10">
    <Description>Base 10 Logarithm</Description>
</Log>
```

Example 6 – Example `Log` node applying a base 10 logarithm.

```
<Log inBitDepth="32f" outBitDepth="32f" style="cameraLinToLog">
    <Description>Linear to DJI D-Log</Description>
    <LogParams base="10" logSideSlope="0.256663" logSideOffset="0.584555"
        linSideSlope="0.9892" linSideOffset="0.0108" linSideBreak="0.0078"
        linearOffset="0.0929" linearSlope="6.025"/>
</Log>
```

Example 7 – Example `Log` node applying the DJI D-Log formula.

4.4.7 `Exponent`

Description:

This node contains parameters for processing pixels through a power law function. Two main formulations are supported. The first follows a pure power law. The second is a piecewise function that follows a power function for larger values and has a linear segment that is followed for small and negative values. The latter formulation can be used to represent the Rec. 709, sRGB, and CIE L* equations.

Attributes:

`style` (required)

specifies the form of the exponential function to be applied.
Supported values are:

- `"basicFwd"`
- `"basicRev"`
- `"basicFwdMirror"`
- `"basicRevMirror"`
- `"basicFwdPassthru"`
- `"basicRevPassthru"`
- `"moncurveFwd"`
- `"moncurveRev"`

The formula to be applied for each style is included in Equations 4.21-4.28 for all of which:

$g =$ `exponent` $\quad k =$ `offset`

MAX(a, b) returns a if $a > b$ and b if $b \geq a$

`"basicFwd"`

applies a power law using the exponent value specified in the `ExponentParams` element. Values less than zero are clamped.

$$\text{basicFwd}(x) = [\text{MAX}(0, x)]^g \tag{4.21}$$

`"basicRev"`

applies power law using the exponent value specified in the `ExponentParams` element. Values less than zero are clamped.

$$\text{basicRev}(y) = [\text{MAX}(0, y)]^{1/g} \tag{4.22}$$

`"basicFwdMirror"`

applies a basic power law using the exponent value specified in the `ExponentParams` element for values greater than or equal to zero and mirrors the function for values less than zero (i.e. rotationally symmetric around the origin):

$$\text{basicFwdMirror}(x) = \begin{cases} x^g & \text{if } x \geq 0 \\ -\left[(-x)^g\right] & \text{otherwise} \end{cases} \tag{4.23}$$

`"basicRevMirror"`

applies a basic power law using the exponent value specified in the `ExponentParams` element for values greater than or equal to zero and mirrors the function for values less

than zero (i.e. rotationally symmetric around the origin):

$$\text{basicRevMirror}(y) = \begin{cases} y^{1/g} & \text{if } y \geq 0 \\ -\left[(-y)^{1/g}\right] & \text{otherwise} \end{cases} \tag{4.24}$$

`"basicFwdPassthru"`
applies a basic power law using the exponent value specified in the `ExponentParams` element for values greater than or equal to zero and passes values less than zero unchanged:

$$\text{basicFwdPassthru}(x) = \begin{cases} x^g & \text{if } x \geq 0 \\ x & \text{otherwise} \end{cases} \tag{4.25}$$

`"basicRevPassthru"`
applies a basic power law using the exponent value specified in the `ExponentParams` element for values greater than or equal to zero and and passes values less than zero unchanged:

$$\text{basicRevPassthru}(y) = \begin{cases} y^{1/g} & \text{if } y \geq 0 \\ y & \text{otherwise} \end{cases} \tag{4.26}$$

`"moncurveFwd"`
applies a power law function with a linear segment near the origin

$$\text{moncurveFwd}(x) = \begin{cases} \left(\frac{x+k}{1+k}\right)^g & \text{if } x \geq xBreak \\ x\,s & \text{otherwise} \end{cases} \tag{4.27}$$

Where:

$$xBreak = \frac{k}{g-1}$$

And for Eq. 4.27 and Eq. 4.28:

$$s = \left(\frac{g-1}{k}\right)\left(\frac{kg}{(g-1)(1+k)}\right)^g$$

`"moncurveRev"`
applies a power law function with a linear segment near the origin

$$\text{moncurveRev}(y) = \begin{cases} (1+k)\,y^{(1/g)} - k & \text{if } y \geq yBreak \\ \frac{y}{s} & \text{otherwise} \end{cases} \tag{4.28}$$

Where:

$$yBreak = \left(\frac{kg}{(g-1)(1+k)}\right)^g$$

NOTE: The above equations assume that the input and output bit-depths are floating-point. Integer values are normalized to the range [0.0, 1.0].

Elements:

`Description` (optional)
See Section 4.3

`ExponentParams` (required)
contains one or more attributes that provide the values to be used by the enclosing `Exponent` element.
If `style` is any of the ”basic” types, then only `exponent` is required.
If `style` is any of the ”moncurve” types, then `exponent` and `offset` are required.

Attributes:

`"exponent"` (optional)
the power to which the value is to be raised
Must be in the range 0.01 to 100.0, inclusive. Default is 1.0.

`"offset"` (optional)
the offset value to use
Must be in the range 0.0 to 0.9, inclusive.
If `offset` is used, the enclosing `Exponent` element’s style attribute must be set to one of the ”moncurve” types. Offset is not allowed when `style` is any of the ”basic” types.

NOTE 2: Zero is specified as a valid value for offset but if used for the calculation of $xBreak$ or $yBreak$ will create a divide-by-zero error. Implementors should protect against this case.

`"channel"` (optional)
the color channel to which the exponential function is applied. Possible values are `"R"`, `"G"`, `"B"`.
If this attribute is not otherwise specified, the exponential function is applied to all color channels.

Examples:

```
<Exponent inBitDepth="32f" outBitDepth="32f" style="basicFwd">
    <Description>Basic 2.4 Gamma</Description>
    <ExponentParams exponent="2.2" />
</Exponent>
```

Example 8 – Using `Exponent` node for applying a 2.2 gamma.

```
<Exponent inBitDepth="32f" outBitDepth="32f" style="moncurveRev">
    <Description>EOTF similar to (but not identical to) sRGB</Description>
    <ExponentParams exponent="2.4" offset="0.055" />
</Exponent>
```

Example 9 – Using `Exponent` node for applying an EOTF very similar to that found in IEC 61966-2-1:1999 (sRGB).

4.4.8 ASC_CDL

Description:

This node processes values according to the American Society of Cinematographers’ Color Decision List (ASC CDL) equations. Color correction using ASC CDL is an industry-wide method of recording and exchanging basic color correction adjustments via parameters that set particular color processing 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 style used.

NOTE: Equations 4.29-4.32 assume that *in* and *out* are scaled to normalized floating-point range. If the ASC_CDL node has inBitDepth or *outBitDepth* that are integer types, then the input or output 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.

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, and if present, must contain each of the following sub-elements:

Slope
three decimal values representing the R, G, and B slope values, which is similar to gain, but changes the slope of the transfer function without shifting the black level established by offset
The valid range is [0.0, 100.0]. The nominal value is 1.0 for all channels.

Offset
three decimal values representing the R, G, and B offset values, which raise or lower overall brightness of a color component by shifting the transfer function up or down while holding the slope constant
The valid range is [−100.0, 100.0]. The nominal value is 0.0 for all channels.

Power
three decimal values representing the R, G, and B power values, which change the intermediate shape of the transfer function
The valid range is (0.0, 10.0]. The nominal value is 1.0 for all channels.

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

Saturation
a single decimal value applied to all color channels
The valid range is [0.0 , 10.0]. The nominal value is 1.0.

NOTE 2: If either element is not specified, values should default to the nominal values for each element. If using the `"noClamp"` style, the result of defaulting to the nominal values is a no-op.

NOTE 3: 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:

$$out_{SOP} = \text{CLAMP}(in \times slope + offset)^{power} \tag{4.29}$$

$$\begin{aligned} luma &= 0.2126 \times in_R + 0.7152 \times in_G + 0.0722 \times in_B \\ out &= \text{CLAMP}(luma + saturation \times (out_{SOP} - luma)) \end{aligned} \tag{4.30}$$

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 $(input \times slope + offset) < 0$, then no power function is applied.

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

$$\begin{aligned} in_{clamp} &= \text{CLAMP}(in) \\ luma &= 0.2126 \times in_{clamp,R} + 0.7152 \times in_{clamp,G} + 0.0722 \times in_{clamp,B} \\ out_{SAT} &= luma + \frac{(in_{clamp} - luma)}{saturation} \end{aligned} \tag{4.31}$$

$$out = \text{CLAMP}\left(\frac{\text{CLAMP}(out_{SAT})^{\frac{1}{power}} - offset}{slope}\right) \tag{4.32}$$

Where:

CLAMP() clamps the argument to [0,1]

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

Examples:

```
<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 10 – Example of an `ASC_CDL` node

5 Implementation Notes

5.1 Bit Depth

5.1.1 Processing precision

All processing shall be performed using 32-bit floating-point values. The values of the `inBitDepth` and `outBitDepth` attributes shall not affect the quantization of color values.

NOTE: For some hardware devices, 32-bit float processing might not be possible. In such instances, processing should be performed at the highest precision available. Because CLF permits complex series of discrete operations, CLF LUT files are unlikely to run on hardware devices without some form of pre-processing. Any pre-processing to prepare a CLF for more limited hardware applications should adhere to the processing precision requirements.)

5.1.2 Input and output to a `ProcessList`

The bit-depth handling of input and output values shall be handled by the application or device implementing the specification. Specifically, an implementation shall scale input image data in order to to make it suitable for processing through a CLF based on the `inBitDepth` of the first `ProcessNode` in the `ProcessList`. Likewise, an implementation shall scale the output as determined by the `outBitDepth` tag of the final `ProcessNode` in the `ProcessList`.

NOTE: The `ProcessList` itself does not contain global `inBitDepth` and `outBitDepth` tags.

No clamping or quantization shall be performed based on the `inBitDepth` or `outBitDepth` tags, the bit-depth tags serve only to indicate to the processor how the data is expected to be scaled.

For example, a CLF where the first `ProcessNode` has an `inBitDepth` of `"10i"` and the last `ProcessNode` has an `outBitDepth` of `"12i"` expects image data to be scaled relative to 0-1023 and output image data scaled relative 0-4095.

NOTE 2: The output bit depth of `"12i"` does not require that the output values be quantized to an integer data type, just that they are scaled relative to 0-4095.

The assumption in the `ProcessList` element is that integer values become normalized floating-point values. The normalized value of 1.0 in corresponds to the maximum code value from an integer encoding, but normalized floating-point values may exceed the range 0-1. There is no clamping specified.

The `Range` node may be used one or more times in a `ProcessList` to provide explicit scaling or clamping, if desired.

5.1.3 Input and output to a `ProcessNode`

The input bit-depth of each subsequent node must be matched by the output bit-depth of the previous node in the Process List.

Conversion of values between floating-point and integer ranges shall be performed within a `ProcessNode` according to 5.1.4 in order to reconcile the input data as denoted by `inBitDepth` to the scaling expected by the node's operators. The `inBitDepth` and `outBitDepth` attributes of a `ProcessNode` serve as a guide to the implementation as to how to scale the data, if necessary, to be used in a node. Specifically, the formulas in the `Log`, `Exponent`, and `ASC_CDL` elements assume that data is normalized 0-1. `LUT1D`, `LUT3D` `Array`, `Matrix`, and `Range` have no presupposition of the scaling of data coming into those nodes.

For example, if a matrix is being copied into a CLF file from an implementation where all data was 10-bit, it might be more convenient to paste the matrix values into the node as integers in the range [0, 1023] then pre-normalizing them 0-1. This is easily allowed by setting the `inBitDepth` of the `Matrix` node to be `"10i"`, which will hint to the implementation that it must scale data by a factor 1023 before applying the matrix for a have have knowledge to scale implementValues can also be used outside this range, and those values will be used in intermediate computations unless explicitly clamped by a `Range` node or using a clamping style (such as in `ASC_CDL` node).

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.

5.1.4 Conversion between integer and normalized float scaling

The process of preparing float data for input to a ProcessNode with an integer inBitDepth or vice-versa, is accomplished via a simple scale factor based on the bit-depth, n.

Conversions from integers with bit-depth n to float should be performed according to Eq. 5.1.

$$\textit{floatValue} = \frac{\textit{intValue}}{(2^n - 1)} \tag{5.1}$$

Conversions from float to integer with bit-depth n should be performed according to Eq. 5.2.

$$\textit{intValue} = \text{ROUND}\left[\textit{floatValue} \times (2^n - 1)\right] \tag{5.2}$$

Where:
ROUND(x) rounds to the nearest integer by adding 0.5 and truncating the value after the decimal

5.2 Required vs Optional

The required or optional indicated in parentheses throughout this specification indicate the requirement for an element or attribute to be present for a valid CLF file. In the spirit of a LUT format to be used commonly across different software and hardware, none of the elements or attributes should be considered optional for implementors to support. All elements and attributes, if present, should be recognized and supported by an implementation.

If, due to hardware or software limitations, a particular element or attribute is not able to be supported, a warning should be issued to the user of a LUT that contains one of the offending elements. The focus shall be on the user and maintaining utmost compatibility with the specification so that LUTs can be interchanged seamlessly.

5.3 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.

5.4 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.

5.5 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 ensure 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`.

5.6 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).

5.7 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. To assure maximum compatibility and similar results across implementations, applications should support tetrahedral interpolation for 3DLUTs, if possible. Optimized matrix forms of trilinear and tetrahedral interpolation can be found in Appendix C.

5.8 Extensions

It is recommended that implementors of CLF file readers protect against unrecognized elements or attributes that are not defined in this specification. Unrecognized values should either raise an error or at least provide a warning message to the user to indicate that there is an operator present that is not recognized by the reader.

One or more `Description` elements in the `ProcessList` can and should be used for metadata that does not fit into a provided field in the `Info` element and/or is unlikely to be recognized by other applications.

6 Examples

Example CLF files can be found in the supplemental materials to this document.

A full example of an XML file (Example 11) 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 11 – Full example of an XML LUT file

Appendix A

(normative)

XML Schema

The XML schema is also provided as a text file in the supplemental files provided with this specification.

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

<xs:schema targetNamespace="urn:NATAS:AMPAS:LUT:v2.0"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:lut="urn:NATAS:AMPAS:LUT:v2.0"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

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

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

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

    <xs:complexType name="InfoType">
        <xs:sequence>
            <xs:element name="AppRelease" type="xs:string" minOccurs="0"
                maxOccurs="1"/>
            <xs:element name="Copyright" type="xs:string" minOccurs="0"
                maxOccurs="1"/>
            <xs:element name="Revision" type="xs:string" minOccurs="0"
                maxOccurs="1"/>
            <xs:element name="ACEStransformID" type="xs:string" minOccurs="0"
                maxOccurs="1"/>
            <xs:element name="ACESuserName" type="xs:string" minOccurs="0"
                maxOccurs="1"/>
            <xs:element ref="lut:CalibrationInfo" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

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

    <xs:complexType name="CalibrationInfoType">
        <xs:sequence>
            <xs:element name="DisplayDeviceSerialNum" type="xs:string"
                minOccurs="0" maxOccurs="1"/>
            <xs:element name="DisplayDeviceHostName" type="xs:string"
                minOccurs="0" maxOccurs="1"/>
```

```
            <xs:element name="OperatorName" type="xs:string" minOccurs="0"
                maxOccurs="1"/>
            <xs:element name="CalibrationDateTime" type="xs:string"
                minOccurs="0" maxOccurs="1"/>
            <xs:element name="MeasurementProbe" type="xs:string" minOccurs="0"
                maxOccurs="1"/>
            <xs:element name="CalibrationSoftwareName" type="xs:string"
                minOccurs="0" maxOccurs="1"/>
            <xs:element name="CalibrationSoftwareVersion" type="xs:string"
                minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

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

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

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

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

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

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

    <!--  ProcessNode: Exponent definition  -->
    <xs:element name="Exponent" type="lut:ExponentType" substitutionGroup="lut:ProcessNode"/>
```

```
<xs:complexType name="ExponentType">
    <xs:complexContent>
        <xs:extension base="lut:ProcessNodeType">
            <xs:sequence>
                <xs:element name="ExponentParamsType" type="lut:ExponentParamsType"
                    minOccurs="0" maxOccurs="1"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

<xs:complexType name="ExponentParamsType">
    <xs:attribute name="exponent" type="lut:exponentType" use="optional" default="1.0"/>
    <xs:attribute name="offset" type="lut:offsetType" use="optional" default="0.0"/>
    <xs:attribute name="channel" type="lut:channelType" use="optional"/>
</xs:complexType>

<xs:simpleType name="exponentType">
    <xs:restriction base="xs:decimal">
        <xs:minInclusive value="0.01"/>
        <xs:maxInclusive value="100.0"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="offsetType">
    <xs:restriction base="xs:decimal">
        <xs:minInclusive value="0.0"/>
        <xs:maxInclusive value="0.9"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="channelType">
    <xs:restriction base="xs:string">
        <xs:pattern value="[RGB]"/>
    </xs:restriction>
</xs:simpleType>

<!--  ProcessNode: Log definition  -->
<xs:element name="Log" type="lut:LogType" substitutionGroup="lut:ProcessNode"/>

<xs:complexType name="LogType">
    <xs:complexContent>
        <xs:extension base="lut:ProcessNodeType">
            <xs:sequence>
                <xs:element name="LogParamsType" type="lut:LogParamsType" minOccurs="0"
                    maxOccurs="1"/>
            </xs:sequence>
            <xs:attribute name="style" type="xs:string" use="required"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

<xs:complexType name="LogParamsType">
    <xs:attribute name="base" type="xs:integer" use="optional" default="10"/>
    <xs:attribute name="logSideSlope" type="xs:decimal" use="optional" default="1.0"/>
    <xs:attribute name="logSideOffset" type="xs:decimal" use="optional" default="0.0"/>
    <xs:attribute name="linSideSlope" type="xs:decimal" use="optional" default="1.0"/>
    <xs:attribute name="linSideOffset" type="xs:decimal" use="optional" default="0.0"/>
    <xs:attribute name="linSideBreak" type="xs:decimal" use="optional" default="0.0"/>
    <xs:attribute name="linearSlope" type="xs:decimal" use="optional"/>
    <xs:attribute name="linearOffset" type="xs:decimal" use="optional"/>
    <xs:attribute name="channel" type="xs:string" use="optional"/>
</xs:complexType>

<!--  ProcessNode: ASC-CDL definition  -->
<xs:element name="ASC_CDL" type="lut:ASC_CDLType" substitutionGroup="lut:ProcessNode"/>
```

```
<xs:complexType name="ASC_CDLType">
    <xs:complexContent>
        <xs:extension base="lut:ProcessNodeType">
            <xs:sequence>
                <xs:element name="SOPNodeType" type="lut:SOPNodeType" minOccurs="0"
                    maxOccurs="1"/>
                <xs:element name="SatNodeType" type="lut:SatNodeType" minOccurs="0"
                    maxOccurs="1"/>
            </xs:sequence>
            <xs:attribute name="style" type="xs:string" use="required"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

<xs:complexType name="SOPNodeType">
    <xs:sequence>
        <xs:element name="Slope" type="lut:floatListType" minOccurs="0" maxOccurs="1"
            default="1.0 1.0 1.0"/>
        <xs:element name="Offset" type="lut:floatListType" minOccurs="0" maxOccurs="1"
            default="0.0 0.0 0.0"/>
        <xs:element name="Power" type="lut:floatListType" minOccurs="0" maxOccurs="1"
            default="1.0 1.0 1.0"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="SatNodeType">
    <xs:sequence>
        <xs:element name="Saturation" type="xs:float" minOccurs="0" maxOccurs="1"
            default="1.0"/>
    </xs:sequence>
</xs:complexType>

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

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

<!--  ProcessNode: Matrix definition  -->
<xs:element name="Matrix" type="lut:MatrixType"
    substitutionGroup="lut:ProcessNode"/>

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

```
    </xs:complexType>

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

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

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

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

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

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

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

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

</xs:schema>
```

Appendix B

(informative)

Changes between v2.0 and v3.0

- Add `Log` ProcessNode
- Add `Exponent` ProcessNode
- Revise formulas for defining use of `Range` ProcessNode to clamp at the low or high end.
- `IndexMap`s with dim > 2 no longer allowed. Use a `halfDomain` LUT to achieve reshaping of input to a LUT.
- Appendix explaining complex `IndexMap` removed.
- `CalibrationInfo` element removed from the `Info` element of `ProcessList`. Use one or more of the more generic `Description` elements to communicate any relevant details about a CLF's intended usage.
- Move `ACEStransform` elements to `Info` element of `ProcessList` in main spec
- Update schema to correct errors and add new elements

Appendix C

(normative)

Interpolation

When an input value falls between sampled positions in a LUT, the output value must be calculated as a proportion of the distance along some function that connects the nearest surrounding values in the LUT. There are many different types of interpolation possible, but only three types of interpolation are specified for use with the Common LUT Format (CLF).

The first type – linear interpolation – is specified for use with the LUT1D Process Node. The other two – trilinear and tetrahedral interpolation – are specified for use with the LUT3D Process Node.

C.1 Linear Interpolation

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
⋮	⋮	⋮	⋮
$n-1$	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.

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

C.2 Trilinear Interpolation

Trilinear interpolation implements linear interpolation in three-dimensions by successively interpolating each direction.

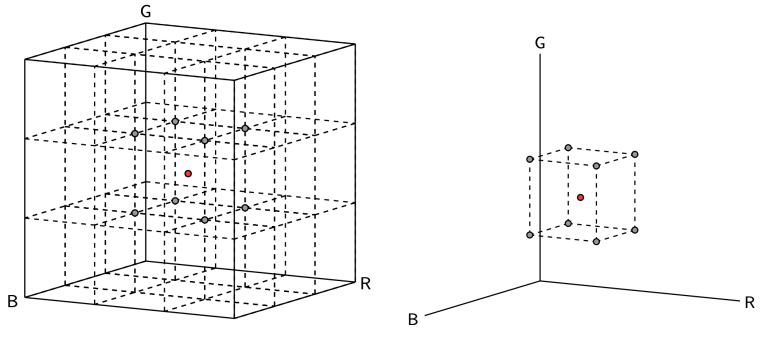

Figure 3 – Illustration of a sampled point located within a basic 3D LUT mesh grid (left) and the same point but with only the vertices surrounding the sampled point (right).

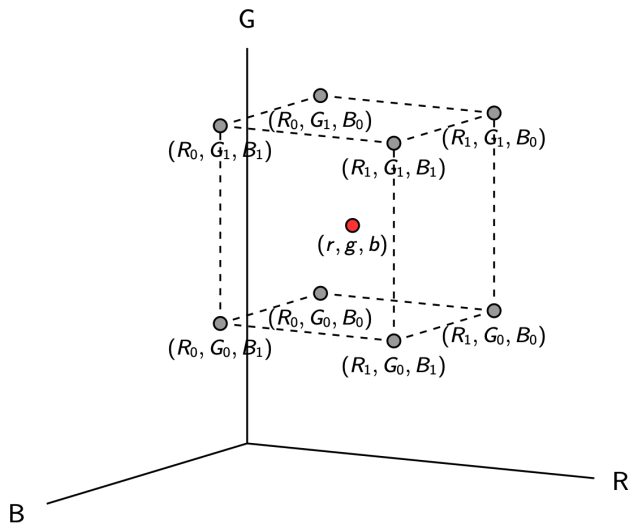

Figure 4 – Labeling the mesh points surrounding the sampled point (r, g, b).

NOTE: The convention used for notation is uppercase variables for mesh points and lowercase variables for points on the grid.

Consider a sampled point as depicted in Figure 4. Let $V(r, g, b)$ represent the value at the point with co-ordinate (r, g, b). The distance between each node per color coordinate shows the proportion of each mesh point's color coordinate values that contribute to the sampled point.

$$\Delta_r = \frac{r - R_0}{R_1 - R_0} \qquad \Delta_g = \frac{g - G_0}{G_1 - G_0} \qquad \Delta_b = \frac{b - B_0}{B_1 - B_0} \tag{C.1}$$

The general expression for trilinear interpolation can be expressed as:

$$V(r, g, b) = c_0 + c_1\Delta_b + c_2\Delta_r + c_3\Delta_g + c_4\Delta_b\Delta_r + c_5\Delta_r\Delta_g + c_6\Delta_g\Delta_b + c_7\Delta_r\Delta_g\Delta_b \tag{C.2}$$

where:

$$\begin{aligned}
c_0 &= V(R_0, G_0, B_0) \\
c_1 &= V(R_0, G_0, B_1) - V(R_0, G_0, B_0) \\
c_2 &= V(1_0, G_0, B_0) - V(R_0, G_0, B_0) \\
c_3 &= V(R_0, G_1, B_0) - V(R_0, G_0, B_0) \\
c_4 &= V(R_1, G_1, B_1) - V(R_1, G_0, B_0) - V(R_0, G_0, B_1) + V(R_0, G_0, B_0) \\
c_5 &= V(R_1, G_1, B_0) - V(R_0, G_1, B_0) - V(R_1, G_0, B_0) + V(R_0, G_0, B_0) \\
c_6 &= V(R_0, G_1, B_1) - V(R_1, G_1, B_0) - V(R_0, G_0, B_1) + V(R_0, G_0, B_0) \\
c_7 &= V(R_1, G_1, B_1) - V(R_1, G_1, B_0) - V(R_0, G_1, B_1) - V(R_1, G_0, B_1) \\
&\quad + V(R_0, G_0, B_1) + V(R_0, G_1, B_0) + V(R_1, G_0, B_0) - V(R_0, G_0, B_0)
\end{aligned}$$

Expressed in matrix form:

$$\mathbf{C} = [c_0 \;\; c_1 \;\; c_2 \;\; c_3 \;\; c_4 \;\; c_5 \;\; c_6 \;\; c_7]^T \tag{C.3}$$

$$\mathbf{\Delta} = [1 \quad \Delta_b \quad \Delta_r \quad \Delta_g \quad \Delta_b\Delta_r \quad \Delta_r\Delta_g \quad \Delta_g\Delta_b \quad \Delta_r\Delta_g\Delta_b]^T \tag{C.4}$$

$$V(r, g, b) = \mathbf{C}^T\mathbf{\Delta} \tag{C.5}$$

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 \\ 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 \\ -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} V(R_0, G_0, B_0) \\ V(R_0, G_1, B_0) \\ V(R_1, G_0, B_0) \\ V(R_1, G_1, B_0) \\ V(R_0, G_0, B_1) \\ V(R_0, G_1, B_1) \\ V(R_1, G_0, B_1) \\ V(R_1, G_1, B_1) \end{bmatrix} \tag{C.6}$$

The expression in Equation C.6 can be written as: $\mathbf{C} = \mathbf{AV}$.

Trilinear interpolation shall be done according to $V(r, g, b) = \mathbf{C}^T\mathbf{\Delta} = \mathbf{V}^T\mathbf{A}^T\mathbf{\Delta}$.

NOTE 2: The term $\mathbf{V}^T\mathbf{A}^T$ does not depend on the variable (r, g, b) and thus can be computed in advance for optimization. Each sub-cube can have the values of the vector $\mathbf{C}$ already stored in memory. Therefore the algorithm can be summarized as:

1. Find the sub-cube containing the point (r, g, b)
2. Select the vector $\mathbf{C}$ corresponding to that sub-cube
3. Compute Δ_r, Δ_g, Δ_b
4. Return $V(r, g, b) = \mathbf{C}^T\mathbf{\Delta}$

C.3 Tetrahedral Interpolation

Tetrahedral interpolation subdivides the cubelet defined by the vertices surrounding a sampled point into six tetrahedra by segmenting along the main (and usually neutral) diagonal (Figure 5).

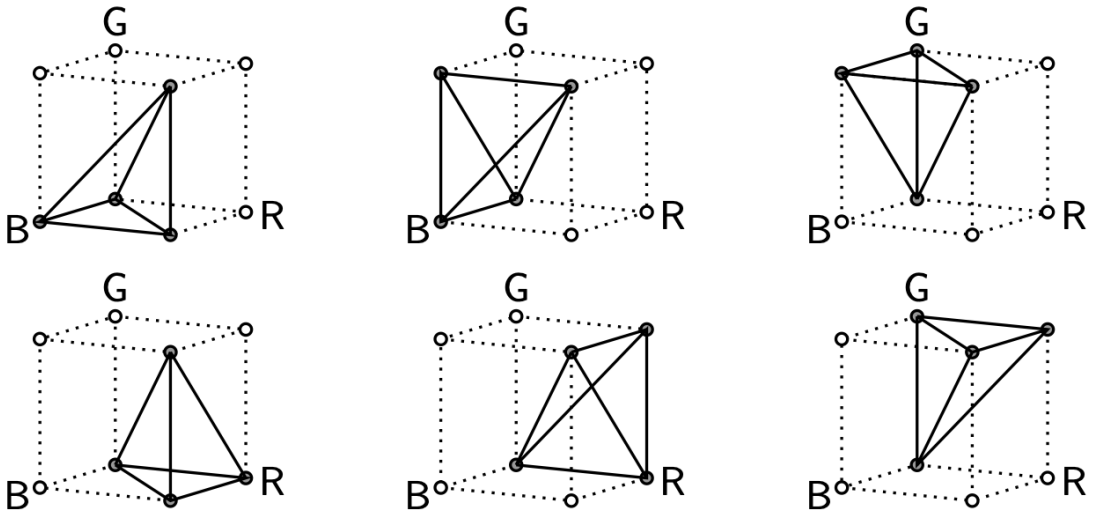

Figure 5 – Illustration of the six subdivided tetrahedra.

To find the tetrahedron containing the point (r, g, b):

- if $\Delta_b > \Delta_r > \Delta_g$, then use the first tetrahedron, $t1$
- if $\Delta_b > \Delta_g > \Delta_r$, then use the first tetrahedron, $t2$
- if $\Delta_g > \Delta_b > \Delta_r$, then use the first tetrahedron, $t3$
- if $\Delta_r > \Delta_b > \Delta_g$, then use the first tetrahedron, $t4$
- if $\Delta_r > \Delta_g > \Delta_b$, then use the first tetrahedron, $t5$
- else, use the sixth tetrahedron, $t6$

The matrix notation is:

$$\mathbf{V} = \begin{bmatrix} V(R_0, G_0, B_0) \\ V(R_0, G_1, B_0) \\ V(R_1, G_0, B_0) \\ V(R_1, G_1, B_0) \\ V(R_0, G_0, B_1) \\ V(R_0, G_1, B_1) \\ V(R_1, G_0, B_1) \\ V(R_1, G_1, B_1) \end{bmatrix} \tag{C.7}$$

$$\mathbf{\Delta_t} = [1 \;\; \Delta_b \;\; \Delta_r \;\; \Delta_g]^T \tag{C.8}$$

$$\mathbf{T}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \quad \mathbf{T}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \end{bmatrix}$$

$$\mathbf{T}_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{T}_4 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \tag{C.9}$$

$$\mathbf{T}_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{T}_6 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Trilinear interpolation shall be done according to:

$$V(r,g,b)_{t1} = \mathbf{\Delta}_t^T \mathbf{T}_1 \mathbf{V} \tag{C.10}$$
$$V(r,g,b)_{t2} = \mathbf{\Delta}_t^T \mathbf{T}_2 \mathbf{V} \tag{C.11}$$
$$V(r,g,b)_{t3} = \mathbf{\Delta}_t^T \mathbf{T}_3 \mathbf{V} \tag{C.12}$$
$$V(r,g,b)_{t4} = \mathbf{\Delta}_t^T \mathbf{T}_4 \mathbf{V} \tag{C.13}$$
$$V(r,g,b)_{t5} = \mathbf{\Delta}_t^T \mathbf{T}_5 \mathbf{V} \tag{C.14}$$
$$V(r,g,b)_{t6} = \mathbf{\Delta}_t^T \mathbf{T}_6 \mathbf{V} \tag{C.15}$$

NOTE: The vectors $\mathbf{T}_i\mathbf{V}$ for $i = 1, 2, 3, 4, 5, 6$ does not depend on the variable (r, g, b) and thus can be computed in advance for optimization.

Appendix D

(normative)

Cineon-style Log Parameters

When using a `Log` node, it might be desirable to conform an existing logarithmic function that uses Cineon style parameters to the parameters used by CLF. A translation from Cineon-style parameters to those used by CLF's `LogParams` element is quite straightforward using the following steps.

Traditionally, *refWhite* and *refBlack* are provided as 10-bit quantities, and if they indeed are, first normalize them to floating point by dividing by 1023.

$$refWhite = \frac{refWhite_{10i}}{1023.0} \tag{D.1}$$

$$refBlack = \frac{refBlack_{10i}}{1023.0} \tag{D.2}$$

Where:
subscript *10i* indicates a 10-bit quantity

The density range is assumed to be:

$$range = 0.002 \times 1023.0 \tag{D.3}$$

Then solve the following quantities:

$$multFactor = \frac{range}{gamma} \tag{D.4}$$

$$gain = \frac{highlight - shadow}{1.0 - 10^{(MIN(multFactor \times (refBlack - refWhite), -0.0001)}} \tag{D.5}$$

$$offset = gain - (highlight - shadow) \tag{D.6}$$

$$\tag{D.7}$$

Where:
$MIN(x, y)$ returns x if $x < y$, otherwise returns y

The parameters for the `LogParams` element are then:

$$\texttt{base} = 10.0 \tag{D.8}$$

$$\texttt{logSlope} = \frac{1}{multFactor} \tag{D.9}$$

$$\texttt{logOffset} = refWhite \tag{D.10}$$

$$\texttt{linSlope} = \frac{1}{gain} \tag{D.11}$$

$$\texttt{linOffset} = \frac{offset - shadow}{gain} \tag{D.12}$$