A Common File Format for Look-Up Tables

A project of

The Academy of Motion Picture Arts and Sciences[1]
Science and Technology Council

&

The American Society of Cinematographers
Technology Committee

Version:	**1.01**
Date:	May 11, 2008
Document Editor:	Jim Houston (jim.houston@mindspring.com)

[1] © 2008 Academy of Motion Picture Arts and Sciences (A.M.P.A.S.) All rights reserved.

A Common File Format for Look-Up Tables: Version 1.0

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.

1.0 Introduction

Color LUTs (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 also need to be 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 matrixes, 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 straight-forward 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.

Sections 2, 3, and 4 have the definitions, specifications, and conventions for the LUT format.

Section 5 provides the "object model" overview that is appropriate for implementers and XML readers.

Sections 6 and 7 give the detailed XML specification and element explanations. (Note: the full verifiable XML schema is attached as an appendix.)

Section 8 illustrates the forms of various XML LUT files.

Section 9 contains some notes for implementers.

The concluding section 10 discusses some possible future work.

2.0 Definitions

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], … , 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.

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.

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	1
…			…	
n-1	1		*n-1*	1000

the *out_result* resulting from an *input* can be calculated after finding the nearest *inValue*[i] < *input.* When *inValue[i] = input,* the result is evaluated directly.

$$out\_result = \frac{input - inValue[i]}{inValue[i+1] - inValue[i]} \;\; x \left(outValue[j+1] - outValue[j]\right) + outValue[j]$$

IndexMap definition

The mapping of input code values to indexes of the table is modifiable allowing changes to the spacing of the sampling function, reshaping of the function in the LUT or remapping the range of applicable input values. 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 clipped to the minimum and maximum output values provided in the LUT. 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@0 512@1 756@2 900@3 1023@4. This changes a table from the behavior of the left side to the new behavior on the right side. (note: the pixel output results of each of these transforms would be different.)

	Without IndexMap				With IndexMAP	
index	inValue	outValue		index	inValue	outValue
0	0	0		0	0	0
1	255	10		1	512	10
2	511	100		2	756	100
3	767	1000		3	900	1000
4	1023	1023		4	1023	1023

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 input value in the table. The transform algorithm then calculates the output value by interpolation between *outValue* entries in the table.

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.

3DLUT definition

In a 3DLUT, 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 the nearest eight table entries are identified. The 3-component output value is calculated by interpolating within the volume defined by the nearest 8 positions in the 3DLUT.[2]

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 output value 3-component 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].

[2] 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

3.0 Specification

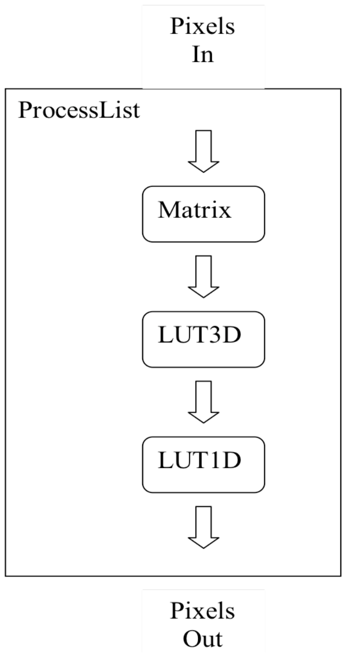
Color transformations (LUTs) are stored in text files that have a defined XML structure.

The top level element in the XML file defines a 'Process List' 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 node 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 process list 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 node must indicate for its input and output whether the pixel values are floats or integers, and must also specify the bit depth. When the input and the output of a LUT node do not match, it is required that the conversion is achieved within the LUT's output values. It is expected that the output of one node will match the input of the next node. Conversions between integer or floating point ranges should be made explicit either in the LUT data 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.

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 may be encapsulated in a larger XML structure which potentially could provide 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 <u>one</u> 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 XML Elements 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 may specify the index positions of the LUT relative to the input code value range and thus may change the spacing of samples relative to the input values. [see *`IndexMap`* for details]

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 4x4 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.

4.0 Conventions

Some characteristics of the XML format are constrained to improve interoperability and interchange between the different systems that use LUTs. Among these agreed upon conventions are:

Multiple *ProcessLists* in a file should be handled by a higher-level XML structure, which the user may define as the root node.

Matrices are either 3x3 or 4x4.

3DLUTs have the same dimension on all axes. A dimension greater than 128x128x128 should be avoided.

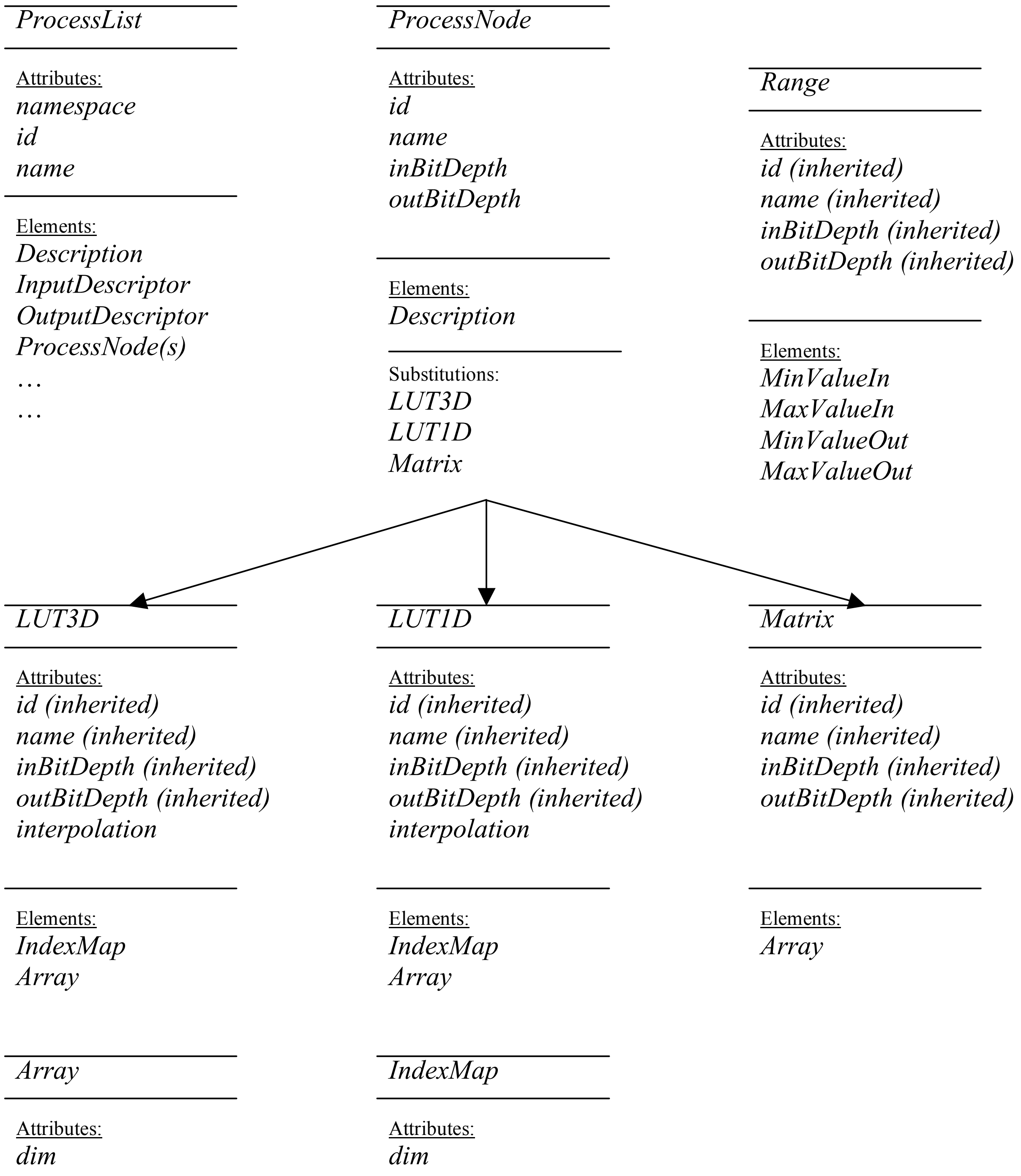
1D and 3x1D LUTs that are integer code lists should always include entries for the minimum and maximum representable code values. As a general principle, the LUT should make explicit the clipping behavior for values outside of the region of interest. It is recommended that integer LUTs should include input code value 0 as the first entry in the LUT.

Supported bit depth values are "`8i`", "`10i`","`12i`", ‶, "`16i`", "`16f`", "`32f`", and "`64f`".
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 and 64f luts should never be fully enumerated.

5.0 **Object Model**

The objects that a LUT designer and implementers should be aware of are:

Figure 2: **Object Model Diagram**



6.0 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 highly recommended although not a strict requirement 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      >
```

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.

IETF RFC 3066: IETF (Internet Engineering Task Force). RFC 3066: Tags for the Identification of Languages, ed. H. Alvestrand. 2001. (See http://www.ietf.org/rfc/rfc3066.txt.)

7.0 XML Elements

<u>`ProcessList`</u>

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

Attributes:

`id:` Unique identifier of the `ProcessList`. This attribute is required.

`name`: Text name of the `ProcessList` for display or selection from an application's user interface. This attribute is optional.

Elements:

`Description`: Comment field for an arbitrary string describing the function, usage, or notes about the `ProcessList`. A `ProcessList` can have one or more `Descriptions`.

`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`: At least one `ProcessNode` must be in the list.

<u>`ProcessNode`</u>

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 `ProcessNodes` can be defined as part of the standard to handle alternate forms of processing. All `ProcessNodes` inherit the attributes and elements below.

Attributes:

`id`: Unique identifier of the `ProcessNode`. This attribute is optional.

`name`: Text name of the `ProcessNode` for display or selection from an application's user interface. This attribute is optional.

`inBitDepth`: Number of bits and type of the input values to which the `ProcessNode` is applied. This attribute is required. The input values can be either integers or floats. Supported values are "`8i`", "`10i`", "`12i`", "`16i`", "`16f`", "`32f`", and "`64f`".

`outBitDepth`: Number of bits and type of the output values which the `ProcessNode` generates. This attribute is required. The output values can be either integers or floats, independent of the input values. Supported values are "`8i`", "`10i`","`12i`", ", "`16i`", "`16f`", "`32f`", and "`64f`".

Elements:

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

<u>`LUT1D`</u> (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 a 3by1D LUT is supplied, each color component is looked up in a separate 1D LUT of the same length. In both cases, the ordered entries of the LUT are provided in

`Array`. In a 3by1D LUT, by convention, the 1D LUT 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`: Name or description of the preferred calculation to interpolate values from the `LUT1D`. This attribute is optional. Systems that utilize LUTs may use different types of interpolation; therefore, this attribute is only 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`".

Elements:

`IndexMap`: Table that maps input values to index positions of the `LUT1D`'s `Array`. A `LUT1D` can have one or three `IndexMaps` for the 1D LUT and 3by1D LUT cases, respectively. In a 3by1D LUT for RGB, the first `IndexMap` corresponds to R, the second `IndexMap` to G, the third `IndexMap` to B.
(see `IndexMap` element below for more details)

`Array`: 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 R's 1D LUT, the 2nd column corresponds to G's 1D LUT, etc.

<u>`LUT3D`</u> (substitute for `ProcessNode`)

This element specifies a 3D LUT. In a `LUT3D`, the 3 color components of the input value are used to find the nearest indexed values along each axis of the 3D face-centered cube. The 3-component output value is calculated by interpolating within the volume defined by the nearest 8 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`: Name or description of the preferred calculation used to interpolate values in the 3DLUT. This attribute is optional. Systems that utilize LUTs may use different types of interpolation; therefore, this attribute is only 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`" of "`4pt tetrahedral`".

Elements:

`IndexMap`: Table that maps input values to index positions of the `LUT3D`'s `Array`. A `LUT3D` can have one or three `IndexMaps`. If there are three `IndexMaps`, they are in the RGB order. If only one `IndexMap` is present, it is applied to all three color components. (see `IndexMap` element below for more details)

`Array`: 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..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

`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.

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

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

$$R = (r\ a_{11})+(g\ a_{12})+(b\ a_{13}),\quad G=(r\ a_{21})+(g\ a_{22})+(b\ a_{23}),\quad B=(r\ a_{31})+(g\ a_{32})+(b\ a_{33})$$

A 4x4 matrix may also be used 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 4x4 matrix would be *(R, G, B, 1.0)* and the *1.0* term may be dropped.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \mathit{off}1 \\ a_{21} & a_{22} & a_{23} & \mathit{off}2 \\ a_{31} & a_{32} & a_{33} & \mathit{off}3 \\ 0 & 0 & 0 & 1.0 \end{bmatrix} \begin{bmatrix} r \\ g \\ b \\ 1.0 \end{bmatrix} = \begin{bmatrix} R \\ G \\ B \\ 1.0 \end{bmatrix}$$

Elements:

`Array`: Table that provides the coefficients of the transformation matrix. The matrix dimensions are either 3x3 or 4x4. 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} ..." for a 3x3 matrix.

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

`Range` (substitute for `ProcessNode`)

This node allows for range selection and rescaling. The InValue range is scaled to the OutValue range. The bitdepth element is used to determine the input and output bitdepths of this node. In this particular node, if the input and output bitdepths 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 bitdepth for the missing input range or missing output range.

Elements:

minInValue

maxInValue
minOutValue
maxOutValue

`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 to position 0 in the LUT.

Attributes:
`dim`: Field that specifies the number of items in the list

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>
```

A 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* calculated for 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	((*i/(n-1)*)*(*inMax-InMin*))+*inMin*	100
2	((*i/(n-1)*)*(*inMax-InMin*))+*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="16f"`), a LUT may have been constructed to use only the input values of *[0 .. 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.

It is not expected that an `IndexMap` would ever 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.

`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.

Attributes:

`dim`: Field that specifies the dimension of the LUT or the matrix and the number of components. This attribute is required. The data points for a *ProcessNode* are contained in an XML array element. Dim provides the dimensionality of the indexes, where:

4 entries has 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 has 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= 4 4 3 is a 4 by 4 matrix acting on 3-component values

2 entries has 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)

The dim attribute and the type of node should match.

8.0 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:

Figure 3: Example of a "3x1D LUT"

```
<?xml version="1.0" encoding="UTF-8"?>
<ProcessList xmlns="urn:NATAS:ASC:LUT:v0.95" id="ex1" name="example 1 transform">
        <Description> Turn 4 grey levels into 4 inverted codes using a 3by1D </Description>
        <LUT1D id="lut-23" name="4valueLut" inBitDepth="2i" outBitDepth="2i">
                <Description> 3by1D LUT </Description>
                <Array dim="4 1">
3
2
1
0
                </Array>
        </LUT1D>
</ProcessList>
```

The LUT1D node could be replaced with a 3D LUT (Figure 4) or a matrix (Figure 5).

Figure 4: Example of a "3D LUT"

```
<LUT3D id="lut-24" name="green look" interpolation="trilinear" inBitDepth="2i" outBitDepth="2f" >
        <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>
```

Figure 5: Example of a "Matrix"

```
<Matrix id="lut-25" name="colorspace conversion" inBitDepth="10i" outBitDepth="10i" >
        <Description> 4x4 Matrix </Description>
        <Array dim="4 4">
1.2     0.0     0.0     0.0
0.0     1.03    0.001   0.0
0.004   -0.007  1.004   0.0
0.002   -0.005  0.0     1
        </Array>
</Matrix>
```

“Shaper LUTs” require a bit more of an explanation. This is once again an illustration of the technique and not a real world example.

Figure 6: Example of a partially enumerated “Shaper LUT”

```
<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>
```

An `IndexMap` 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 less 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..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.

A full example of an XML file in Figure 7 shows three nodes in a `ProcessList`.

Figure 7: Full Example of an XML LUT file

```
<?xml version="1.0" encoding="UTF-8"?>
<ProcessList xmlns="urn:NATAS:ASC:LUT:v0.95" 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="2i" outBitDepth="2i">
		<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="2i" 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="2i">
		<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>
```

9.0 **Implementation notes**

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.

Although accuracy may mean that all of the intermediate calculations be done in floating point, an engine that calculates the transforms individually with rounded integer values will have a slightly different result than the same calculations done in full floating point. The specification does not mandate the details of the implementation's numerical accuracy, although a best effort is expected. The existence of a common LUT format cannot guarantee that the resulting images will look the same on all implementations.

When an IndexMap is used, the transform engine may need to pre-calculate the relevant input value for each integer index position into the LUT because the IndexMap is allowed to have lesser entries than the number of positions in the LUT.

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. It is up to the user to determine whether these approximate results are sufficient.

Floating point processing:

Conversions from float to integer must be done by rounding.

Conversions from integers to float should be done by: for bit depth n $\quad float\_val = intValue / (2^n - 1)$

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 values should be used instead. If floating point values are used by the application even when calculating with integer LUTs, it is the responsibility of the application to handle overflows and underflows correctly.

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. A "Range" node is provided to make explicit the choice of scaling that a LUT designer wants to provide in his transform.

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.

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 committee's 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.

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. Substitution of the correct end-of-line character may be needed to keep the files appearance in the desired form.

10 Conclusion:

The XML format provides an extensible and straight forward method for exchanging LUT and matrix data between users and facilities. New types of `ProcessNodes` can be defined in the future to handle other desired forms of color processing. Among possible additions are nodes that provide ASC-CDL processing or CTL ("Color Transformation Language") processing.

The next step in achieving wider use of this format is to convene an implementers group with the goal of providing a reference implementation. Access to a reader for the XML file should make integration into existing software much easier, and allow widespread use of this format. 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.

The Academy/ASC LUT XML 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.

Acknowledgements:

Thanks to Ana Benitez for creating and verifying the XML schema and helpful contributions to the written description of the XML elements. Thanks also to the other commenters and members of the committee: Richard Antley, Scot Barbour, Al Barton, Lars Borg, Curtis Clark ASC, Bob Currier, Steve Crouch, Simon Cuff, Ray Feeney, Alex Forsythe, 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, and Glenn Woodruff. Thanks also to the Academy of Motion Pictures Arts and Sciences Science and Technology Council and the American Society of Cinematographer's Technology committee for hosting the meetings for this project.

APPENDIX: XML Schema

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

<schema targetNamespace="urn:NATAS:ASC:LUT:v0.95"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:lut="urn:NATAS:ASC:LUT:v0.95"
    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:ProcessNode" minOccurs="1"
maxOccurs="unbounded"/>
        </sequence>
        <attribute name="id" type="anyURI" use="required"/>
        <attribute name="name" type="string" use="optional"/>
    </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"/>
            </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>
        </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>
```