Bit depth handling in CLF

by Nick Shaw, Antler Post

This is a write-up of my understanding of how bit depth is handled in CLF, based on testing of the reference Python implementation by HP Duiker, and subsequent email correspondence with HP, Alex Forsythe, Doug Walker and JD Vandenberg.

It also includes some general thoughts on CLF.

Overview

A CLF XML LUT consists of a ***Process List*** made up of a number of ***Process Nodes***. The types of ***Process Node*** are:

LUT1D
LUT3D
Matrix
Range
ASC CDL

The ***Process List*** itself has a number of global attributes, which are (as far as I am aware) purely informative. For example it may include:

```
<InputDescriptor>ACES (SMPTE ST 2065-1)</InputDescriptor>
```

But this is only a comment, and implementations are not expected to convert image data automatically to ACES2065-1 because of this. I would suggest that perhaps for ACES use the description of the various ACES spaces could be rigidly defined, and then implementations could pre-convert (or not) image data accordingly. For example LMTs are specified to be applied to ***ACES2065-1*** image data and return the same, but the working space for grading is normally ***ACEScct***. So if it were possible to designate a CLF as expecting ***ACEScct*** and returning the same, one could remove some redundant forward/reverse transforms from an LMT. This would remove the rounding errors which are inevitable when using a 1D LUT to perform a linear to log conversion over a wide range.

Each ***Process Node*** includes tags for ***inBitDepth*** and ***outBitDepth***, but as far as I am aware the ***Process List*** as a whole has no global ***inBitDepth*** and ***outBitDepth***, so if an implementation needs to pre-convert image data to make it suitable for CLF processing, it must read the ***inBitDepth*** of the first ***Process Node***. It must also read the ***outBitDepth*** of the last ***Process Node*** if it needs to convert the output of the CLF back to its own working bit depth.

The permitted values for bit depths are ***16f***, ***32f***, ***8i***, ***10i*** and ***12i***, corresponding to half-float, float32, and 8, 10 and 12 bit integer respectively.

The Python reference implementation performs only the processing in the CLF itself, and makes no assumptions about the incoming or outgoing data. Therefore a CLF where the first ***Process Node*** has an ***inBitDepth*** of ***10i*** and ***outBitDepth*** of ***12i*** for example, expects image data ranged 0-1023, and returns image data (depending of the array contents, of course) ranged 0-4095. However these are purely the ranges corresponding to 0-1 in normalised float. They do not define data types. Nor do they mandate clamping to the output range, unless explicitly set to do so by a flag in a ***Process Node***. Thus a table value of 1023 in a ***LUT1D*** with an outBitDepth of ***10i*** will return the float value 1023.0, not the int value 1023, or the float value 1.0. My misunderstanding regarding this was the root of my initial confusion on the subject.

The ***inBitDepth*** of each Process Node defines the way that incoming data should be interpreted. This means that it should match the ***outBitDepth*** of the preceding ***Process Node***. I am unclear whether this match is mandated by the spec (I cannot see that explicitly stated) and an error should be raised if there is a mismatch. Testing indicates that the Python reference implementation does not raise an error, and simply interprets the incoming data to a ***Process Node*** as being in the expected bit depth, even if this is not the same as the ***outBitDepth*** of the preceding ***Process Node***.

My understanding of the handling of bit depth in each type of ***Process Node*** is as follows:

LUT1D

If the ***inBitDepth*** is integer, the incoming data is normalised to 0-1 based on the bit depth n, before indexing into the array using the following pseudo-code:

```
index = input / (2^n - 1)
```

For float input ***index*** is the unmodified float input value.

The ***index*** may be then remapped by an ***IndexMap***, if present, listing the mappings between float values of the ***index*** and integer line numbers in the array. IndexMap support is not mandatory, and some implementations appear to simply ignore it.

The output value (or values for a 3x1D LUT) are then indexed and interpolated from the array in the usual way. The values are returned unmodified, as the array contents are assumed to be in a range appropriate for the ***outBitDepth***. Thus ***outBitDepth*** has no effect on processing for a ***LUT1D***.

A ***LUT1D Process Node*** may have a ***halfDomain*** tag, and in this case the float index value is converted to the 16 bit integer which codes that value in half-float, and the output is taken from that line in the array (which must have 65536 lines).

If the ***rawHalfs*** tag is set then the ***outBitDepth*** should be***16f***. I do not see anything in the spec which states what the handling should be if this is not the case. In line with the fact that output values of a ***LUT1D*** are not processed in any way based on ***outBitDepth***, the reference implementation simply returns the float value which is half-float coded by the integer in the array. The returned value is a float32, not a half-float.

LUT3D

Bit depth handling for a LUT3D is exactly the same as for a LUT1D, except that ***halfDomain*** and ***rawHalfs*** do not apply.

Matrix

A ***Matrix*** may be either 3x3 or 3x4, with the fourth column being offsets.

For a ***Matrix***, ***inBitDepth*** and ***outBitDepth*** are purely informative. If they do not match, the conversion is expected to be incorporated into the matrix coefficients. Using the example given by Doug Walker, a conversion from normalised float to 10-bit legal range integer (returned as un-quantised floats) could be performed with the following ***Matrix***:

```
<Matrix inBitDepth="16f" outBitDepth="10i" >
     <Array dim="3 4 3">
            876  0    0    64
            0    876  0    64
            0    0    876  64
     </Array>
</Matrix>
```

Range

With a ***Range Process Node***, the ***inBitDepth*** and ***outBitDepth*** are expected to be incorporated into the values of ***minInValue***, ***maxInValue***, ***minOutValue*** and ***maxOutValue***, so if all four are specified the bit depths are informative only. If any are not given, the default minimum and maximum values are 0.0 and 1.0 respectively for float input/output bit depths, and 0 and (2^n - 1) respectively for integer input/output bit depths.

The float to 10-bit legal example from ***Matrix*** above could be implemented as the following ***Range Process Node***:

```
<Range inBitDepth="16f" outBitDepth="10i" style="noClamp">
     <minInValue>0.0</minInValue>
     <maxInValue>1.0</maxInValue>
     <minOutValue>64</minOutValue>
     <maxOutValue>940</maxOutValue>
</Range>
```

The in value lines could be omitted, as their values are the defaults for ***16f***.

The ***style="noClamp"*** setting allows values outside the 0-1 float range to be mapped to super-white and sub-black integer values. Again the 10-bit values are returned as un-quantised floats, not integers.

ASC CDL

ASC CDL is the only case where both ***inBitDepth*** and ***outBitDepth*** are always taken into account in the calculations. Because the values in an ***ASC CDL*** are intended to operate on normalised float data, the input data are normalised based on ***inBitDepth*** by dividing them by (2^n - 1) for integer input. For float ***inBitDepth*** the data are assumed to be already so normalised. Likewise for integer values of ***outBitDepth***, the float output of the CDL operation is multiplied by (2^n - 1).

The ***ASC CDL Process Node*** also includes a ***style*** tag which has the possible values ***Fwd***, ***Rev***, ***FwdNoClamp*** and ***RevNoClamp***. The first two are forward and reverse processing as per ASC CDL 1.2 spec, with clamping to the 0-1 range applied in normalised float, prior to bit depth multiplication. The ***NoClamp*** variants omit the clamping.

Conclusions and implications for CLF implementation in Colour Science for Python

Modern grading systems operate in normalised float, so when applying a CLF to image data it is necessary for them to convert it based on the ***inBitDepth*** value of the first ***Process Node***, multiplying the data by (2^n - 1) if integer data is expected by the CLF. Likewise the output of the CLF must be converted back to float if the ***outBitDepth*** of the last ***Process Node*** is integer.

This is similar to the way ***OCIOFileTransform*** or ***Vectorfield*** behave in Nuke when applying a .3dl LUT. The table values in the file may be integer, but Nuke treats them as mapping to and from the 0-1 range. You do not need to multiply your image data by 1023 before applying a 10-bit .3dl LUT to it, or divide by 1023 afterwards.

The question is whether an implementation of CLF in Colour should (optionally) perform the same transforms of the data, or follow the lead of the reference implementation, and leave that for the user to add as separate processing. The general approach in Colour has always been to expect reasonable knowledge from the users, and let them incorporate such conversions into their code separately. But perhaps the recently introduced ***from_range_1*** and ***to_domain_1*** functions might be used to allow automatic conversion. I am proposing this as a possibility, not necessarily my preferred approach.

If I were designing CLF myself from scratch, I think I would omit the integer options. To my mind they add unnecessary complexity and potential confusion. I was initially confused myself, expecting a 10i to 12i CLF to behave in the Python reference implementation like a 10-bit in, 12-bit out .3dl behaves in Nuke when sent a float value in the 0-1 range. Given that the calculations performed when building a LUT are normally done in floating point, and the array values stored in CLF are floating point, even if ranged 0.0-1023.0, for example, the integer options simply apply a scale factor. Although hardware LUT implementations may work in integer, the time to convert to that would seem to me to be at the point of upload to the hardware. Indeed, most upload utilities for LUT boxes expect LUTs with 0-1 float values, and hide the conversion to integer from the user.

I think my confusion stemmed from section 9.3 of the spec where it says:

> *Although it has been traditional practice in converting from integer to floats to normalize the top integer code to a value of 1.0 in floating point, there is an increasing need for calculations in high-dynamic range imaging systems where this assumption might be flawed, so the assumption in the ProcessList element is that integer values are normalized floats where for example of 1023 in integer is output from a node as 1.0.*

I find this slightly ambiguous, and it should perhaps be clarified in the next iteration of the spec.

I read (and I'm sure others may do the same) *"1023 in integer is output from a node as 1.0"* as meaning that if ***outBitDepth*** is ***"10i"*** and the value in a LUT table is 1023, this will be normalised **by the node**, and the output of the node will be 1.0. In fact, such normalisation needs to be done after the CLF process by the system applying the CLF based on the value of ***outBitDepth*** of the last Process Node in the Process List, in order to transform back to its own working bit depth (probably normalised float in a modern grading system).

The integer options in CLF as it stands, are in fact no more than named ranges. They do not modify the data type, and array values do not have to be integers. A CLF variant without the ***inBitDepth*** and ***outBitDepth*** tags would not be restricted to input and output in the 0-1 range, as conversions could be included as ***Range*** operations. Indeed, I would argue that doing so might make the processing in a CLF clearer when reading the XML.

I am also puzzled by the option to have ***16f*** and ***32f*** as options for bit depths. Since the reference implementation appears not to treat the two differently (***halfDomain*** and ***rawHalfs*** are a specific exception, but then the bit depth is implicit anyway) there seems no purpose. The spec specifically says *"transform authors should not assume that applications will necessarily clamp and quantize based on the settings of the outBitDepth attribute"*.

I mean no criticism of the reference implementation when I say that it is extremely slow. I am aware that like ***ctlrender*** that it was never intended to be an efficient implementation for production use. Since Colour is entirely vectorized using NumPy, whereas the reference implementation appears to simply iterate over pixels, I expect an implementation in Colour to be significantly faster. Some test code I have written which parses an ***IndexMap***, for example, using Numpy can do so in a fraction of a second. The reference implementation seems particularly slow at this, taking about 14 seconds for the same test 4096 entry ***IndexMap***.