# Experiment with rendering views of 3D LUTs with OpenGL

## Description

Firstly, we've used Excel to examine 2D planes through the standard SLog3/SGamut3.Cine -> Rec.709 cube (extracted from Sony Raw Viewer). We've been able to appreciate the clipping and hue changes and approximately identify the Rec.709 black/red/green/blue/cyan/yellow/magenta/white points within the SGamut3 cube.

Next stop try rendering this Rec.709 hexahedron within the SGamut cube in OpenGL. OpenGL seems like the right tool for the job, and we can ultimately host OpenGL code in browser, Qt/QML, Qt/QtWidgets, etc.

Suggest aiming to render a wire-frame cube for SGamut and a solid body for Rec.709, allowing user to control view point.

Can experiment with colouring according to the input coordinates (SGamut space) or the output (Rec.709) values.

Suggest hosting with WebGL initially.

Useful tutorials:

[Mozilla Creating 3D objects using WebGL](https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Creating_3D_objects_using_WebGL)

[WebGL rendition of the famous NeHe tutorials](http://learningwebgl.com/blog/?page_id=1217)

There'll be lots of other useful 2D and 3D renderings we can usefully do in OpenGL. We can create new tracker issues for e.g. colour wheel, etc. using a shader.

## Updated by Callum McGregor

Investigated rendering the S-Gamut3-Cine to Rec.709 LUT as two nested RGB colour spaces using WebGL with help from the Mozilla tutorials (linked in previous post).

Initial experiments into rendering 2D squares were successful and progressed to applying a gradiented colour to the surface of the square (the same as one face of our RGB colour cube), animating the square, and making the square into a 3D cube with faces appropriately coloured to represent an RGB colour cube.

Vertices within the resulting Rec.709 colour space were discovered through examination of the LUT in Excel and used to generate a dodecahedron (as each face of the cube is constructed of 2 triangles), representing the resulting colour space. A wireframe of the original S-Gamut3-Cine colour space was modeled with coloured edges and the Rec.709 colour space nested inside: this allows one to see how colours are mapped between the two palettes and ultimately how distorted colours become when translated.

A need for user interaction of the 3D objects was identified; initial research uncovered a number of methods however implementation was complicated (code not included in [r2](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/2)). Further research found a JavaScript library called three.js which provides facilities for easier generation and interaction with 3D objects. Three also has the advantage of being able to be ported into QML 5.5 3D canvases.

To progress, we will generate our previous models using three.js and try to port over to a QML app.

([r2](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/2))

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

We will generate our previous models using three.js and try to port over to a QML app.

Some useful links:

Three.js is at <http://threejs.org/>

[Qt 5.5 announcement of Canvas3D](http://blog.qt.io/blog/2015/05/27/introducing-qt-canvas3d/)

[Porting Three.js code to Canvas3D](https://blog.qt.io/blog/2015/06/05/porting-three-js-code-to-canvas3d/)

And [Qt 5.5 actually released](https://blog.qt.io/blog/2015/07/01/qt-5-5-released/) just last week!

[Qt Canvas3D](http://doc.qt.io/qt-5/qtcanvas3d-index.html) documentation (some links in above posts were broken, I guess because they were written when Qt 5.5 was still in beta)

And the cool 3D image histogram demo in WebGL/Three.js that I shared this morning: <http://h3stogram.herokuapp.com/>

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

So current target for this tracker issue is to port the 'raw' WebGL rendering to Three.js (and learn Three.js idioms while we do so!).

Beyond that I wonder if we should make new issues for follow-up tasks.

E.g.

Host/port Three.js in/to QML Canvas3D

Remembering that we're really only showing the corners of the LUT at the moment, try some easy 'full' rendering of the 3D LUT such as:

A point (or sphere) cloud at each LUT co-ordinate in the colour of the LUT output at that point (and this obviously requires that we load the full LUT cube file into our 'app')

One line segment for each point in the LUT starting at LUT co-ordinate going to LUT output value to visualize the 'movement'

Some more renderings such as:

A tri-linearly interpolated coloured cube between adjacent points in the LUT - of course if these are opaque, we only see the outside of the whole cube, so we might experiment with controlled animations of the opacity, or camera moves through 3D to reveal the inside of the cube

Non-face-aligned 2D planes through the 3D space - such as black-white axis coming out of the scene - really this needs volumetric rendering which may be best accomplished some other way, but coiuld naively be done by sub-dividing each of the LUT point cubes into much smaller ones with interpolated values

Default camera positions such as making black-white axis aligned to z (come out of the screen) or y or x-y

I also want to render the CIE (L\*)a\*b\* colour plane using a GLSL shader, so I want to know how to plug this into Three.js framework. I'll add this as another tracker issue too.

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

**File** [5.MEDICAL.cube](http://bprl-redmine.eu.sony.com/redmine/attachments/2418/5.MEDICAL.cube) added

FYI, you can access the standard Sony LUTs from RAW Viewer in its installation directory, e.g. C:\Program Files\Sony\RAW Viewer\Data\LookProfile\SGamut3Cine - and add new LUTs here if you want to try one in RAW Viewer (you don't seem to be able to just 'load' or 'import' a LUT file that's somewhere else?).

[5.MEDICAL.cube](http://bprl-redmine.eu.sony.com/redmine/attachments/download/2418) is the result of Pablo Garcia's playing with the heart surgery material, exported from Resolve.

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

Links that I have lying around in a gazillion open browser tabs that may later help us implement confidently colour space conversion e.g. between different RGB primaries, or L\*a\*b\* vs. Rec.709 RGB, etc.

[Cinematic Color](http://cinematiccolor.com/)

[Visualizing the XYZ Color Space](https://www.youtube.com/watch?v=x0-qoXOCOow)

Poynton on, for example, [How do I transform data from one set of RGB primaries to another?](http://www.poynton.com/notes/colour_and_gamma/ColorFAQ.html#RTFToC20)

[OpenCV cvtColor](http://docs.opencv.org/modules/imgproc/doc/miscellaneous_transformations.html#cvtcolor)

[EasyRGB](http://www.easyrgb.com/index.php?X=MATH&H=07#text7)

And a wonderful step-by-step page on [RGB Color Space Conversion](http://www.ryanjuckett.com/programming/rgb-color-space-conversion/).

## Updated by Gareth Sylvester-Bradley

Other ideas:

Render the *difference* between two LUTs

Tri-linearly interpolate points in the point cloud between the existing points in the LUT - though this seems to me like we're not making most efficient use of the GPU to do interpolation for us.

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

Moved away from representing the Rec.709 colour space as 12 faces (I refrain from calling it a solid dodecahedron as it has no interior) nested within the S-Gamut3.Cine unit cube to using a point cloud; however, as the input and output coordinates are both 0-1 but in different coordinate systems mapping all the output points only resulted in a cube of irregularly spaced points, not the smaller "dodecahedron" that was hoped for.

In an attempt to get my head around the subject I wrote a function that calculated the 8 vertices that we had previously found by hand. I achieved this by finding all the points in the output coordinates that were the same colour (as the LUT is a many-to-one mapping), finding the points in the input space that mapped to them and plotting the one closest to the center of the cube (the same process that we did by hand previously). While I believe the theory of this is solid, in practice the high precision (6 decimal places) and relatively low sampling rate of the LUT meant that few points were exactly the same; consequently apply this formula to all points in the LUT, not just the vertices, did not render the results that I had hoped for.

After this we decided to move to an alternative representation of the LUT while still using point clouds. I wrote four functions that displayed input or output coordinates coloured corresponding to the input or output space, which show:

**Input coords with input colours:** Nothing of great meaning, but quite pretty.

**Input coords with output colours:** How colours are transformed when moving to Rec.709. Note that extreme points around the edges are often the same colour as they are all mapped to the same colour within the output colour space.

**Output coords with input colours:** How colours from the input space are mapped onto the output space. Note how there is a high concentration of points and bright colour on the edges where extreme points have been clipped.

**Output coords with output colours:** Nothing much except for how the regular grid has been warped by the LUT's transformation.

Next we may look into styling points so it is clear which points within the input space are being represented in the output space (essentially what I was attempting to do at the beginning of this check-in), however the benefits of this are unclear. Other ways of visualising the LUT would be to render the input points coloured according to the difference in input and output colours: this would make colour shifts apparent.

([r10](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/10))

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

Added the ability to view the difference in colours between the input and output on the input (regularly spaced) point cloud. This allows one to see shifts in hue for each point with mid-grey being no change: note how the corner that is green in the input and yellow in the output is pinky-red when viewing the difference. These interesting results increase our desire to be able to view inside the point cloud: I will investigate using plane clipping to limit the number of points drawn and therefore "cut" through the point cloud.

Also used a .png file from Three's GitHub (link below) to display points in the point cloud as small discs rather than squares.

<https://github.com/mrdoob/three.js/blob/master/examples/textures/sprites/disc.png>

Next I will look at the possibility of using a collection of 3D Sphere geometries, stored inside a 3DObject to render the point cloud, rather than using Three's provided PointCloud.

([r11](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/11))

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

(Old changes that I only just noticed)

Attempted to use a collection of Sphere geometries instead of a point cloud but due to performance issues this will have to be put on hold until more time can be devoted to optimising code, but it is possible as proved [here](http://www.html5rocks.com/en/tutorials/webgl/million_letters/).

([r14](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/14))

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

I have started looking into converting a CIE-Lab colour space into RGB (Rec.709). The aim of this task is to understand the main advantage of Lab: like our vision, Lab is not linearly spaced and therefore colour gradients in Lab appear linear to us, unlike those in RGB. By converting a part of the Lap space into RGB and displaying the colour gradient using OpenGL, this smooth gradient should be apparent.

The Lab colour space is explained well [here](http://www.colourphil.co.uk/lab_lch_colour_space.shtml) :

*... the vertical L\* axis represents Lightness, ranging from 0-100. The other (horizontal) axes are now represented by a\* and b\*. These are at right angles to each other and cross each other in the centre, which is neutral (grey, black or white). They are based on the principal that a colour cannot be both red and green, or blue and yellow.*

*The a\* axis is green at one extremity (represented by -a), and red at the other (+a).*

*The b\* axis has blue at one end (-b), and yellow (+b) at the other.*

*In theory there are no maximum values of a\* and b\*, but in practice they are usually numbered from -128 to +127 (256 levels).*

It is also notable that when converting from RGB colours with values of 0 to 255, some of these corresponding a and b values are not represented. As answered by [this question on Stack Overflow](http://stackoverflow.com/questions/19099063/what-are-the-ranges-of-coordinates-in-the-cielab-color-space), the practical limits of a and b can be calculated as:

*L in [0, 100]*

*A in [-86.185, 98,254]*

*B in [-107.863, 94.482]*

I have chosen to keep L constant at 50: it should be kept constant as changing it would add no change to the hue of colours, and I have chosen 50 as after playing with colours on this very useful [colour converter](http://www.workwithcolor.com/color-converter-01.htm), colours seem more true at L=50.

Using the formulae found on the [Lab Colour Space Wikipedia page](https://en.wikipedia.org/wiki/Lab_color_space) I have successfully transformed 256^2 CIE-Lab values into CIE-XYZ, then in Rec.709 Gamma using the same matrices and gamma functions as done in [~~#6150~~](http://bprl-redmine.eu.sony.com/redmine/issues/6150). My results differ slightly from those given by the previously linked colour converter, but I wonder if it's them that are wrong.

I will now research Fragment Shading in OpenGL, WebGL, or perhaps even Three.js if it allows.

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

Transferred the logic from the Excel conversion from CIE-LAB to RGB (Rec.709) in [r15](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/15) into OpenGLSL, written in browser [here](http://thijs.wolk.io/glsl/index.html#A). Copied into a .txt file for safe keeping in [r17](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/17).

This visualisation travels through the L\* axis through time, allowing us to see all parts of the LAB colour space in about 10 seconds. It is apparent from this visualisation that Lab is better than RGB at representing colour differences. Note also that the corners of the image appear to be solid colour because they are: as mentioned previously the a\* and b\* values are limited when using RGB and as we are graphing for values beyond these limits, no changes are seen beyond these points.

This visualisation is currently hosted online so next to do is use this fragment shader code in our own WebGL/Three.js program.

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

More clearly separated the LUT visualisers two functions: viewing one LUT and comparing two LUTs. Improved UI, debugged code, and refactored.

([r18](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/18))

# Construct 3D LUT for colour space transform programmatically

## Description

Follow-up to support [#6143](http://bprl-redmine.eu.sony.com/redmine/issues/6143).

Sony provides the S-Gamut3.Cine/S-Log3 to LC\_709 (Low Contrast Rec.709) cube file (used in Sony RAW Viewer and Da Vinci Resolve, etc.).

I'm interested how close we can get to constructing this mathematically/programmatically from known parameters.

It should be possible to calculate a 3x3 matrix transform based on:

S-Log3 definition

S-Gamut3 colour primaries and white point in CIE xy chromaticity space

Rec.709 colour primaries and white point in CIE xy chromaticity space

Rec.709 gamma function definition

I.e. Log to linear, RGB to CIE XYZ, CIE XYZ to RGB, linear to log.

See [RGB Color Space Conversion](http://www.ryanjuckett.com/programming/rgb-color-space-conversion/).

This will give a naive colour space transform.

In order to achieve something similar to the low-contrast LUT, we'll probably need at least to apply lift/gamma/gain to the input. I suggest we use [ASC-CDL](https://en.wikipedia.org/wiki/ASC_CDL) v1.2 formulation, offered by Sony Raw Viewer, which we've previously worked with for F65RAW, etc. in SMDK, see e.g. [MXFAcquisitionMetadata/ASCCDL.h](http://bprl-redmine.eu.sony.com/redmine/projects/smdk/repository/entry/trunk/smdk/Development/MXFAcquisitionMetadata/ASCCDL.h).

The LUT may be more involved than that, but this should be a good start.

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

Callum was asking how to convert between lift/gamma/gain terminology (and values!) and slope/offset/power. I've found several references that allowed me to piece it together, but the best was this description of a [colour corrector node on blender.org](http://wiki.blender.org/index.php/User:Xat/KeyerNode), which I repeat here:

*The formula for ASC CDL colour correction is:*

*out = (in \* slope + offset) ^ power*

*The formula for Lift/Gamma/Gain correction is:*

*out = (gain \* (in + lift \* (1 - in))) ^ (1 / gamma)*

*Converting to ASC CDL slope, offset and power value:*

*power = 1 / gamma*

*offset = lift \* gain*

*slope = (1 - lift) \* gain*

*Converting in the opposite direction:*

*gamma = 1 / power*

*gain = offset + slope*

*lift = offset / (offset + slope)*

Other references via the obvious search: [slope offset power lift gamma gain](https://www.google.co.uk/search?q=slope+offset+power+lift+gamma+gain)

[Lift/gamma/gain vs. ASC-CDL](http://www.francois-tarlier.com/blog/liftgammagain-vs-asc-cdl/)

[What's In A Name?](http://www.provideocoalition.com/whats-in-a-name/) by Alexis Van Hurkman who wrote the Colour Correction Handbook

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

We were also wondered how the ASC-CDL saturation value should be applied. I wasn't sure if it should be applied before or after the slope-offset-power function, nor what the channel weightings to get luminance should be.

For the former, some references clearly say the SATNode is applied after the SOPNode, others are... less clear.

For the latter, several references suggest that the ASC-CDL spec itself mandates the Rec.709 luma coefficients... which is a bit confusing since we think we're applying the CDL *before* the colour space transform from e.g. S-Gamut3.Cine into Rec.709. But this is e.g. offered as one explanation [why ARRI saturation differs from the ASC-CDL spec](http://video.stackexchange.com/questions/9866/how-does-arri-saturation-formula-differ-from-the-asc-cdl-spec).

On both fronts, the black-and-whitest explanation I found is in the [Autodesk Smoke online help on ASC\_CDL](http://help.autodesk.com/cloudhelp/2016/ENU/Smoke/files/GUID-8591FA2F-FC79-4C2A-8D46-64F6F43C17F9.htm).

Btw, Callum, for bonus points, is it possible to calculate the [luma coefficients](https://en.wikipedia.org/wiki/Rec._709#Luma_coefficients) (0.2126, 0.7152, 0.0722) from the Rec.709 RGB primaries and white point?

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

My research agrees with Gareth's and I have added two stages, ASC-CDL (SOP) and ASC-CDL (Sat), into the Excel sheet. I encountered #NUM! errors when calculating some output values for the ASC-CDL (SOP) when *(in \* slope + offset)* is negative and *power* is a non-integer, as these require complex number to solve. A solution is to clip *(in \* slope + offset)* at 0 and possibly 1 as well, however I am unsure if the latter will be of any benefit even though Gareth's previous source ([Autodesk Smoke online help on ASC\_CDL](http://help.autodesk.com/view/SMOKE/2016/ENU/?guid=GUID-8591FA2F-FC79-4C2A-8D46-64F6F43C17F9)) suggested so.

I have also found useful information into how CDL works and how it is used in this paper:

[ASC-CDL A Step Towards Look Management](http://help.autodesk.com/view/SMOKE/2016/ENU/?guid=GUID-8591FA2F-FC79-4C2A-8D46-64F6F43C17F9)

In particular it is useful to note that SOP values have the following ranges (where -/+ represents negative/positive infinity):

Slope : 0 -> + (1.0 has no effect on input)

Offset : - -> + (0.0 has no effect on input)

Power : 0 -> + (not-inclusive) (1.0 has no effect on input)

Lift : - -> + (0.0 has no effect)

Gamma : 0 -> + (1.0 has no effect)

Gain : - -> + (1.0 has no effect)

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

Confirmed in this [Introduction to Colour Decision List](http://www.niwa.nu/2013/05/color-decision-list-introduction/), saturation is applied after slope, offset, and power:

*Saturation is applied after Slope, Offset, and Power.*

[#5](http://bprl-redmine.eu.sony.com/redmine/issues/6150#note-5)

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

I have graphed the S-Gamut3.Cine S-Log3 input against Rec.709 output for all grey-scale values (i.e. all those where r=g=b for the input); graphing this for our generated Rec.709 Gamma, ditto clipped, and the actual LUT shows us the difference between each LUT independent of hue.

In an attempt to quantify how accurate our generated LUT is I have totaled up the errors between our LUT and the one extracted from Sony RAW viewer into a single number. Through adjustments of the lift, gamma, gain in Sony RAW Viewer to I was able to reduce the error by 30% and then by a further 7% by tweaking the LGG values in Excel.

Viewing the graph while adjusting these values also shows our generated LUT's curve approaching the actual LUT's, however it is becoming clear that we will not be able to exactly match the two using only lift, gamma, and gain (and their equivalent SOP values).

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

Thinking about it a bit further, I think we can focus on the clipping in the blacks. S-Log3 black isn't at zero, it's at around 0.1 (I think it's 95.0/1023.0, see [TechnicalSummary\_for\_S-Gamut3Cine\_S-Gamut3\_S-Log3\_V1\_00.pdf](http://bprl-redmine.eu.sony.com/redmine/attachments/2406/TechnicalSummary_for_S-Gamut3Cine_S-Gamut3_S-Log3_V1_00.pdf)). Values below this are "super blacks". We need to apply a (negative?) lift (offset) to put black at black. On the other hand, as we've discovered, I think we can't hope to emulate the smooth clipping of the highlights demonstrated by the LUT... Having just spoken to Steve, the way he puts it is that's where the LUT steps away from the maths and into artistry. If we can match the bottom and the middle parts of the curve, we've done everything we can.

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

Looked into calculating the Rec.709 luma coefficients from the RGB primaries and white point. After hyperlink jumping through many Wikipedia pages I have found that the coefficients are based upon the [CIE colour matching functions](https://en.wikipedia.org/wiki/CIE_1931_color_space#Color_matching_functions) and borrows from the [definition of luminance](https://en.wikipedia.org/wiki/Luminance) and the [luminosity function](https://en.wikipedia.org/wiki/Luminosity_function).

I have come to the conclusion that these calculations, while possible, are too difficult for a small side task.

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

Final attempt to make our generated Rec.709 Gamma (Clipped) curve fit the actual LUT curve.

We identified that it is impossible to make our curve fit the target curve exactly as the target graph's gradient reverses at x=~16; the closest we can achieve is to make the lower portion of the two graphs match. Through experimenting with the SOP values it soon became clear that we could either preserve blacks and fit the belly of the curve well, or clip out blacks and fit the less gradiented, lower-mid part of the curve. The SOP values and corresponding LGG values (for input into Sony RAW Viewer and the like) for my best results are as follows (for all assume saturation=1.0):

Preserving blacks:

S: 0.925

O: 0.130

P: 1.350

L: 0.123

G: 0.741

G: 1.055

Preserving lower mid-tones:

S: 0.850

O:-0.075

P: 0.700

L:-0.097

G: 1.429

G: 0.775

In addition, below are the lift, gamma, gain values to my initial experiments into using CDL to match the two curves and the resulting magnitudes of error (again, assume sat=1.0):

No CDL:

L: 0.000

G: 1.000

G: 1.000

Error = 11703.064

Editing by eye in RAW Viewer:

L:-0.025

G: 1.670

G: 0.595

Error = 8185.478

Fine adjustments in Excel:

L:-0.065

G: 1.670

G: 0.585

Error = 7629.231

Further adjustments in Excel:

S: 0.660

O:-0.080

P: 0.550

Error = 7545.170

The latter result does a good job of preserving contrast in the mid-tones while also minimising the amount of clipping in the blacks and whites.

While it may seem that we failed to achieve our goal of recreating the S-Gamut3.Cine S-Log3 to Rec.709-LC Gamma LUT through mathematical and programmatic procedures, do note that inputting any of the above lift, gamma, gain values into Sony RAW Viewer or Da-Vinci Resolve produce colours that to the eye match those of the real LUT reasonably well.

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

Uploaded Excel document and .cube generated (before applying CDL).

([r15](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/15))

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

Status changed from *New* to *Closed*

Assignee deleted (*~~Callum McGregor~~*)

All done. Artifact is RGB\_Colour\_Space\_Conversion spreadsheet.

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

Callum McGregor wrote:

*All done. Artifact is RGB\_Colour\_Space\_Conversion spreadsheet.*

You're British, it's [artefact](http://grammarist.com/spelling/artefact-artifact/). ;-)

# Implement CIE L\*a\*b\* visualization as Three.js ShaderMaterial

## Description

Follow-up to support [#6143](http://bprl-redmine.eu.sony.com/redmine/issues/6143) [note 10](http://bprl-redmine.eu.sony.com/redmine/issues/6143#note-10) and [note 11](http://bprl-redmine.eu.sony.com/redmine/issues/6143#note-11).

Callum has successfully created a 2D visualization of a\*b\* planes (with time used to explore the L dimension) as a fragment shader in GLSL within a WebGL environment.

We now want to convert this shader program into a [Three.js ShaderMaterial](http://threejs.org/docs/#Reference/Materials/ShaderMaterial) so that we can combine it with point cloud rendering.

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

So far we've been using gl\_FragCoord.xy (as a proportion of the window size) as the (a\*, b\*) coordinate.

We ultimately want our own vec2 or vec3 vertex attribute (i.e. "ab" or "lab") set up by our geometry/material, but we could start by using the Three.js [built-in uv attribute](http://threejs.org/docs/#Reference/Renderers.WebGL/WebGLProgram). Either way, we'll need a vertexShader to convert the vertex attribute to a varying variable for the fragmentShader like this: [three.js - shader material demo](http://mrdoob.github.io/three.js/examples/webgl_shader2.html).

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

Here's the example of setting up a custom vertex attribute ("customColor" in this example) and using it to create a varying vertexShader output/fragmentShader input ("vColor"): [webgl\_custom\_attributes\_lines source](https://threejsdoc.appspot.com/doc/three.js/examples.source/webgl_custom_attributes_lines.html.html) [(demo)](https://threejsdoc.appspot.com/doc/three.js/examples/webgl_custom_attributes_lines.html). There are actually several [examples](https://threejsdoc.appspot.com/doc/index.html#examples-PerspectiveCamera) of custom attributes in the documentation as well.

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

Status changed from *New* to *Closed*

Assignee deleted (*~~Callum McGregor~~*)

Successfully rendered a cube with faces coloured according to L\*a\*b\* using OpenGLSL shaders and Three.js' ShaderMaterial.

Now that this first hurdle has been overcome we can start to look at how we will use a combination of point clouds and surfaces to most effectively represent our LUTs.

([r20](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/20))

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

*Successfully rendered a cube with faces coloured according to L\*a\*b\* using OpenGLSL shaders and Three.js' ShaderMaterial.*

Good work!

*Now that this first hurdle has been overcome we can start to look at how we will use a combination of point clouds and surfaces to most effectively represent our LUTs.*

One of the other examples will really help here: [webgl\_custom\_attributes\_particles2](https://threejsdoc.appspot.com/doc/three.js/examples/webgl_custom_attributes_particles2.html) since it demonstrates how to use gl\_PointSize and a texture to render the point cloud (a.k.a. particle system), coloured according to a varying based on a vertex attribute.

My suggestion is that we actually construct a point cloud whose vertices have an attribute of the LUT output and are positioned (or have another custom attribute) according to the LUT input. The vertex shader could then use some uniforms set from the JavaScript to do the things we currently do outside in Three.js to decide which is used for position and colour. The vertex shader would modify the gl\_Position (one of the other examples does this) and the gl\_PointSize and the vertex colour (or at least a custom varying) and the fragment shader could render the point appropriately.

Open question if we take that approach: which coordinate system should the input/output be represented in? Each in its own (kind of as now), or both in a common space like CIE L\*a\*b\* or XYZ? Doing more calculation work in the GLSL, so on the GPU, makes sense as it'll be quicker than on the CPU. This will be happening every time the scene is rendered, whereas some of the calculations probably only need to be done once (e.g. point colour), but I suspect the shader is still the best place to do them.

# Visualize LUT in CIE L\*a\*b\* space

## Description

Follow-up to support [#6143](http://bprl-redmine.eu.sony.com/redmine/issues/6143) and support [~~#6155~~](http://bprl-redmine.eu.sony.com/redmine/issues/6155).

We have so far explored rendering 3D LUT visualizations in RGB space. We demonstrated these to Alan and Dave on Friday and while we can explain what we think they're showing, it's still not immediately obvious.

This was part of the reason for our work on constructing a naive 3D LUT based on the colour space primaries and log/gamma curves (support [~~#6150~~](http://bprl-redmine.eu.sony.com/redmine/issues/6150)), in case the differences between this 'mathematical' LUT and the provided 'creative' (LC\_709 and MEDICAL) LUTs made it easier to see the wood for the trees. Our conclusion was... it didn't help enough!

The next thing to try is rendering the input and output coordinates/colours in a common colour space. Our plan is to do this in the three-dimensional CIE L\*a\*b\* space, by transforming the (implied) input points in the LUT, and the output points, into CIE L\*a\*b\* for visualization.

(In case it's not obvious, in this view, we expect the naive/mathematical LUT to show input and output points in the same place - apart from where there is clipping! On the other hand, we expect the creative LUTs to show something more interesting...)

We'll investigate this as a 3D visualization, and as 2D visualizations overlaid on the a\*b\* plane, and perhaps on other planes, e.g. L\*a\* or L\*b\*.

Development environment: WebGL, Three.js point cloud and ShaderMaterial.

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

Sorry, cross-posting from support [~~#6155~~](http://bprl-redmine.eu.sony.com/redmine/issues/6155); it should have been here in the first place:

*Now that this first hurdle has been overcome we can start to look at how we will use a combination of point clouds and surfaces to most effectively represent our LUTs.*

One of the other examples will really help here: [webgl\_custom\_attributes\_particles2](https://threejsdoc.appspot.com/doc/three.js/examples/webgl_custom_attributes_particles2.html) ([source](https://threejsdoc.appspot.com/doc/three.js/examples.source/webgl_custom_attributes_particles2.html.html)) since it demonstrates how to use gl\_PointSize and a texture to render the point cloud (a.k.a. particle system), coloured according to a varying based on a vertex attribute.

My suggestion is that we actually construct a point cloud whose vertices have an attribute of the LUT output and are positioned (or have another custom attribute) according to the LUT input. The vertex shader could then use some uniforms set from the JavaScript to do the things we currently do outside in Three.js to decide which is used for position and colour. The vertex shader would modify the gl\_Position (one of the other examples does this) and the gl\_PointSize and the vertex colour (or at least a custom varying) and the fragment shader could render the point appropriately.

Open question if we take that approach: which coordinate system should the input/output be represented in? Each in its own (kind of as now), or both in a common space like CIE L\*a\*b\* or XYZ? Doing more calculation work in the GLSL, so on the GPU, makes sense as it'll be quicker than on the CPU. This will be happening every time the scene is rendered, whereas some of the calculations probably only need to be done once (e.g. point colour), but I suspect the shader is still the best place to do them.

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

When and if we want it, rather than using textures for points, there's a good explanation of how the geometry shader fits between the vertex shader and the fragment shader here: <https://open.gl/geometry> (nice URL, huh?)

## Updated by [Gareth Sylvester-Bradley](http://bprl-redmine.eu.sony.com/redmine/users/4)

Cambridge In Colour tutorial [Visualizing Color Spaces](http://www.cambridgeincolour.com/tutorials/color-spaces.htm) is useful confirmation of what we're trying to do.

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

I have succeeded in displaying and comparing LUTs by translating input and output values into CIE-LAB space.

The previous LUT visualiser code has been modified and almost all of the coordinate and colour calculations moved into GLSL script, which is run on the GPU. This has the advantage of running quicker and having easy to use/read matrix operations and data types built into the language.

As before, a wireframe cube with edges coloured according to the colour space (in this case L\*a\*b\* in the range of L\*:0 to 100, a\*:-128 to 127, and b\*:-128 to 127), and a Point Cloud in which each vertex represents the position of the input or output values from the LUT and can be coloured according to the input or output values of the LUT. This gives the four main views seen in the previous LUT visualiser. ([r22](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/22))

Using L\*a\*b\* space allows the input and output values (in our case S-Gamut3.Cine and Rec.709) to be viewed and compared objectively. While the Rec.709 points were, as we expected, a small distorted cube within the L\*a\*b\* space, the S-Gamut3.Cine point cloud stretched far outside the bounds of our wireframe cube. Initially we were slow to trust it but after calculating the position of one of the most extreme points by hand it became clear that these points were accurate.

The fifth and newest visualisation added to this program was viewing the input coordinates, coloured according to output, flattened onto the plane where L\*=50 and placed in front of a flat square surface coloured by L\*a\*b\*. Because the coordinates are determined by input and the colour by output, the colour contrast between the points and the surface shows us how much they have been transformed. ([r23](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/23))([r24](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/24))

These visualisations can tell us a lot, but what is most useful is being able to compare two LUTs, for example the LC\_709 LUT we extracted from Sony RAW Viewer and our mathematically constructed LUT. By creating two separate scenes, cameras, and renderers in WebGL, I was able to have two adjacent HTML5 canvas' display the same visualation for two different LUTs. They will also both respond to the same input from the user, allowing both 3D objects to be moved together. ([r25](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/25))

One last minor thing that I achieved was fixing the bug where some objects would occasionally remain in the scene when it was cleared.

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

It was discovered that our input coordinates are out by what appears to be a factor of 10. After a day of debugging it is unclear where the error is, however we believe that it is in our code and not in the generation of the LUT.

# Port WebGL/GLSL/Three.js to QML

## Description

Follow-up to support [#6143](http://bprl-redmine.eu.sony.com/redmine/issues/6143) [note 2](http://bprl-redmine.eu.sony.com/redmine/issues/6143#note-2) and support [#6156](http://bprl-redmine.eu.sony.com/redmine/issues/6156).

Now is the time to see if we can host the WebGL/GLSL/Three.js browser-based code we have within a Qt/QML app.

Development environment: Qt 5.5 in order to get access to Canvas 3D

Links:

[Three.js](http://threejs.org/)

Canvas3D [announcement](http://blog.qt.io/blog/2015/05/27/introducing-qt-canvas3d/) and [documentation](http://doc.qt.io/qt-5/qtcanvas3d-index.html)

[Porting Three.js code to Canvas3](https://blog.qt.io/blog/2015/06/05/porting-three-js-code-to-canvas3d/)D

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

The WebGL/Three.js LUT Visualiser code has been ported into a QML app and many features of the key functionality are working after some modification.

So far only one canvas is rendering a point cloud as I am not attempting to run before I can walk, but it can render all combinations of input/output coordinates and colours (with the exception of flattening on L\*=50). I have created a QML layout similar to the HTML however because the GLSL shader code was in the HTML I have had to extract these scripts and access them from a local python server using XMLHttpRequests: this may not be suitable long term but it works well for now. I have also implemented loading LUTs in the same way, however only the Sony RAW Viewer's LC\_709 LUT is currently usable.

The largest challenge so far has been making the controls work. OrbitControls is not supported in QML as it makes references to the HTML document. To overcome this problem I am in the process of modifying the OrbitControls code in my own OrbitControls2. I have chosen to do this as the code is a good starting point and is also compatible with touchscreen devices. So far one can rotate the camera around the point cloud however scrolling and panning does not work.

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

The alterations to OrbitControls have been successful and the visualisation can now be fully controlled in the QML app, as it was in the web page previously. I have included touch interactions, allowing it to work for touch screen devices; however this has not been tested yet as I have not deployed to a device. I have also left out keyboard input, as I don't believe this to be necessary.

Secondly, I now have two canvas' side by side rendering two different LUTs, controlled by the same input, just as was done in the web page. Currently I only have two LUTs saved in the resources directory (RAW Viewer's LC\_709 and our Generated\_Rec\_709\_Clipped) as I have not yet worked out how, or if, I can select files to display in the visualisation.

The visualisaions now render using the disc.png texture instead of the default square. For this, I believe the texture file must be saved locally, and loaded in along with the shaders and LUTs. A minor change that I had neglected until now was to use the size of the LUT to determine how many points to generate where it was hardwired to draw 33^3 points previously. I have also allowed point size to be changed in the JavaScript code (previously only set in the shader), and set the size to 1/lutSize: a size that would fill an evenly distributed cube and seems like a good, somewhat arbitrary, choice.

Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

Users may now select which of the available LUTs to display on each canvas, on the newly formatted UI.

Despite being improved, the UI leaves much to be desired and I have not yet got the hang of QML's methods of formatting. Apart from a few rough spots (drop down menus want to be populated from the JavaScript, and loading and storing of resources is messy) the next thing to do is deploy to a device and test if the touch screen interactions work.

## Updated by [Callum McGregor](http://bprl-redmine.eu.sony.com/redmine/users/158)

The out by a factor-of-10 error has been solved: the cube file being used by the program was out of date with the data from the Excel document. The files have been updated and committed in [r34](http://bprl-redmine.eu.sony.com/redmine/projects/mcip/repository/revisions/34).

Commented out code in f-shader-cloud.c clips output values outside the 0-1 range; this may be useful in future but for now it is not used.

The code currently loads resources (.cube files, shaders, etc.) from the localhost server. In order to deploy to a device, these files should be accessed locally.

Also, while not a job that could be done before I leave, redesigning the JavaScript to separate MVC and make it more OO would be a good use of time. [Here](https://github.com/tronlec/three.js) is a link to the Git repository for the fork of Three.js we are using in order to run in the QML Cavnas3D.