Creating DCTL LUTs

45

Creating DCTL LUTs

This chapter describes how to create DCTL LUTs to perform your own custom mathematical transformations in DaVinci Resolve.

About DCTL

DCTL files are actually color transformation scripts that Resolve sees and applies just like any other LUT. Unlike other LUTs, which are 1D or 3D lookup tables of values that approximate image transformations using interpolation, DCTL files are actually comprised of computer code that directly transforms images using combinations of math functions that you devise. Additionally, DCTL files run natively on the GPU of your workstation, so they can be fast.

Anyone with the mathematical know-how can make and install a DCTL. Simply enter your transformation code, using a syntax that's similar to C (described in more detail below), into any text editor capable of saving a plain ASCII text file, and make sure its name ends with the ".dctl" (DaVinci Color Transform Language) file extension. Once that's done, move the file to the LUT directory of your workstation. Where that is depends on which OS you're using:

- On Mac OS X: Library/Application Support/Blackmagic Design/DaVinci Resolve/LUT/
- On Windows: C:\ProgramData\Blackmagic Design\DaVinci Resolve\Support\LUT
- On Linux: /home/resolve/LUT

When DaVinci Resolve starts up, assuming the syntax of your .dctl is correct, they appear in the Color page Node contextual menu within the DaVinci CTL submenu.

DCTL Syntax

Users need to put __DEVICE__ in front of each function they write. For example:

```
__DEVICE__ float2 DoSomething()
```

The main entry function (transform) should come after all other functions, with the following format argument:

```
__DEVICE__ float3 transform(float p_R, float p_G, float p_B)
```

The main entry function must also have a float3 return value.

For the following floating point math functions, please use the described syntax:

```
float _fabs(float)                          // Absolute Value
float _powf(float x, float y)               // Compute x to the power of y
float _logf(float)                          // Natural logarithm
float _log2f(float)                         // Base 2 logarithm
float _log10f(float)                        // Base 10 logarithm
float _exp2f(float)                         // Exponential base 2
float _expf(float)                          // Exponential base E
float _copysignf(float x, float y)          // Return x with sign changed to sign y
float _fmaxf(float x, float y)              // Return y if x < y
float _fminf(float x, float y)              // Return y if x > y
float _saturatef(float x, float minVal,
float maxVal)                               // Return min(max(x, minVal), maxVal)
float _sqrtf(float)                         // Square root
int   _ceil(float)                          // Round to integer toward + infinity
int   _floor(float)                         // Round to integer toward - infinity
```

```
float _fmod(float x, float y)          // Modulus. Returns x – y *
                                          trunc(x / y)
float _fremainder(float x, float y)    // Floating point remainder
int   _round(float x)                  // Integral value nearest to x rounding
float _hypotf(float x, float y)        // Square root of (x^2 + y^2)
float _atan2f(float x)                 // Arc tangent of (y / x)
float _sinf(float x)                   // Sine
float _cosf(float x)                   // Cosine
float _acosf(float x)                  // Arc cosine
float _asinf(float x)                  // Arc sine
float _fdivide(float x, float y)       // Return (x / y)
float _frecip(float x)                 // Return (1 / x)
```

The following functions support integer type:

```
min, max, abs, rotate
```

Other supported C Math functions include:

acosh, acospi, asinh, asinpi, atan, atanh, atanpi, atan2pi, cbrt, cosh, cospi, exp10, expm1, trunc, fdim, fma, lgamma, log1p, logb, rint, round, rsqrt, sincos, sinh, sinpi, tan, tanh, tanpi, tgamma

We support vector type float2, float3 and float4. The data fields are:

```
float x

float y

float z

float w
```

To generate a vector value, use make_floatN() where N = 2, 3 or 4.

Users can define their own structure using “typedef struct.” For example:

```
typedef struct
   {
       float c00, c01, c02;
       float c10, c11, c12;
   } Matrix;
```

To declare constant memory, use __CONSTANT__. For example:

```
__CONSTANT__ float NORM[] = {1.0f / 3.0f, 1.0f / 3.0f, 1.0f / 3.0f};
```

To pass the constant memory as a function argument, use the __CONSTANTREF__ qualifier, e.g.:

```
__DEVICE__ float DoSomething(__CONSTANTREF__ float* p_Params)
```

A float value must have the ‘f’ character at the end (e.g. 1.2f).

A Simple DCTL LUT Example

The following code shows an example of how to create a simple color gain transformation using the DCTL LUT syntax.

```
// Example to demonstrate simple color gain transformation

__DEVICE__ float3 transform(float p_R, float p_G, float p_B)

{

    const float r = p_R * 1.2f;

    const float g = p_G * 1.1f;

    const float b = p_B * 1.2f;

    return make_float3(r, g, b);

}
```

A Matrix DCTL LUT Example

The following code shows an example of creating a matrix transform using the DCTL LUT syntax.

```
// Example to demonstrate the usage of user defined matrix type to
transform RGB to YUV in Rec. 709


__CONSTANT__ float RGBToYUVMat[9] = { 0.2126f ,  0.7152f , 0.0722f,

                                     -0.09991f, -0.33609f, 0.436f,

                                      0.615f  , -0.55861f, -0.05639f };


__DEVICE__ float3 transform(int p_Width, int p_Height, int p_X, int
p_Y, float p_R, float p_G, float p_B)

{

    float3 result;


    result.x = RGBToYUVMat[0] * p_R + RGBToYUVMat[1] * p_G +
RGBToYUVMat[2] * p_B;

    result.y = RGBToYUVMat[3] * p_R + RGBToYUVMat[4] * p_G +
RGBToYUVMat[5] * p_B;

    result.z = RGBToYUVMat[6] * p_R + RGBToYUVMat[7] * p_G +
RGBToYUVMat[8] * p_B;


    return result;

}
```

A More Complex DCTL LUT Example

The following code shows an example of creating a mirror effect, illustrating how you can access pixels spatially.

```
// Example of spatial access for mirror effect


__DEVICE__ float3 transform(int p_Width, int p_Height, int p_X, int
p_Y, __TEXTURE__ p_TexR, __TEXTURE__ p_TexG, __TEXTURE__ p_TexB)

{

    const bool isMirror = (p_X < (p_Width / 2));

    const float r = (isMirror) ? _tex2D(p_TexR, p_X, p_Y) : _tex2D(p_
TexR, p_Width - 1 - p_X, p_Y);

    const float g = (isMirror) ? _tex2D(p_TexG, p_X, p_Y) : _tex2D(p_
TexG, p_Width - 1 - p_X, p_Y);

    const float b = (isMirror) ? _tex2D(p_TexB, p_X, p_Y) : _tex2D(p_
TexB, p_Width - 1 - p_X, p_Y);

    return make_float3(r, g, b);

}
```