Cube LUT Specification
Version 1.0

Cube LUT Specification Version 1.0

Updated Information/Additional Third Party Code Information available at http://www.adobe.com/go/thirdparty.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Published September 2013

Contents

Introduction

This section is entirely informative and does not form an integral part of the specification.

The Cube format was originally developed by IRIDAS in 2003. This specification is in response to growing adoption and inquiries and is presented in an effort to insure broad interoperability. It captures previously undocumented details of the Cube format that were evident only in implementations made by IRIDAS and Adobe.

This specification is primarily intended for developers of applications that use look-up tables for color conversions. It includes detailed format requirements, three sample files, and a sample application.

Look-up tables (LUTs) are often used when converting an image from one color representation to another, for example, when converting between log and gamma encodings, changing the color space, applying a color correction, applying a look, changing the dynamic range, gamut mapping, exporting to an output device, or previewing how an image will be reproduced.

A look-up table is a sampled representation of a mathematical function, with the samples stored in table form. The table form provides a simple and universal representation of arbitrarily complex (or proprietary) functions, and doing a table look-up is often faster than calculating the original function.

Each sample in the table holds the output value(s) for a specific input coordinate. When an output value is requested for an input coordinate that falls between stored coordinates, the value is obtained by interpolating between adjacent samples.

A Cube file is a text file that defines a look-up table in the Cube format.

The Cube look-up tables store RGB values.

Advantages of the Cube format include:

- The Cube format can describe look-up tables for a wide range of purposes, from simple gamma adjustments for display output to complex HDR image processing.
- The format is well suited for professional digital cinema applications and for both normal range and High-Dynamic Range image processing.
- As Cube files are text files, they are easily edited or reviewed using a text editor.
- A Cube file can include three 1-dimensional tables or one 3-dimensional table.
- The tables can be in a wide range of sizes.
- Cube files are trivial to write and read.
- All values are human-readable as they are in decimal form, and can be of high precision.
- The input domain and output range are not limited to the range 0.0 to 1.0.

1 Scope

This document defines the *cube* file format for the storage and interchange of look-up tables.

2 Conventions

The keywords "shall" and "shall not" indicate requirements to be followed for compliance with this document.

The keywords "should" and "should not" indicate recommendations provided by this specification.

The keywords "may" and "need not" indicate options permitted, but not required, by this specification.

3 Normative References

The following documents contain provisions that, through reference in this text, constitute provisions of this specification.

ISO/IEC 10646 — Information technology -- Universal Coded Character Set (UCS)

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

<sp>

one or more characters, each character being a space character or a tab character

line of text

one or more characters, each character being a text character, or a tab character

newline character

single-byte character with value 0x0A (\n)

reader

application that supports (or claims to support) the Cube LUT format for color conversions

space character

single-byte character with value 0x20 (' ')

tab character

single-byte character with value 0x09 (\t)

text character

single-byte character in the BASIC LATIN block of UTF-8 (code points 0020 (space) to 007E (tilde)). See ISO 10646

5 Common Requirements

5.1 General

All cube files shall meet the requirements stated in section 5 (Common Requirements).

In addition, 1-dimensional cube files shall meet the requirements in section 6 (Cube File With One-Dimensional Tables).

In addition, 3-dimensional cube files shall meet the requirements in section 7 (Cube File With A Three-Dimensional Table).

5.2 File extension

The file name extension shall be ".cube".

5.3 Lines of text

The cube file shall comprise lines of text.

A line of text shall not be longer than 250 bytes. Lines of text do not contain newline characters.

One or more newline characters shall separate adjacent lines of text.

One or more newline characters should follow the last line of text.

Each line of text shall be a keyword with associated parameters, a comment, or table data.

NOTE: Some files use carriage return (\r, 0x0D, 13) as line separator, alone or with a newline character. These files are not valid cube files, as some readers do not support files with carriage return.

5.4 Numbers

Numbers shall use decimal base and decimal point.

Floating-point numbers may include a sign, shall include an integer part, a fractional part, or both, and may include an exponent part, comprising the character e or E followed by an integer.

No number shall exceed the range ±1e+37 (the range of a 32-bit IEEE float).

Unless otherwise stated, numerical values may be specified using integer or floating-point numbers.

5.5 Keyword Requirements

Keyword lines are formatted as a keyword followed by parameters.

Each keyword with its associated parameters shall appear as a separate line of text.

All keywords shall appear before any table data, and may otherwise appear in any order.

Each keyword shall appear at most once.

A line with a keyword may have leading and trailing <sp>.

5.6 Common Keywords

The following keywords with associated parameters are common to all Cube files:

TITLE <sp> "<text>"

DOMAIN_MIN <sp> rl <sp> gl <sp> bl

DOMAIN_MAX <sp> rh <sp> gh <sp> bh

where

<text> shall be any sequence of text characters, excluding the delimiter character " (0x22),

rl, gl, bl shall be three numbers, and shall specify the lower bounds of the input domains for the three components,

rh, gh, bh shall be three numbers, and shall specify the upper bounds of the input domains for the three components.

The file may include any combination of the common keywords.

If TITLE is omitted from the file, the title is undefined.

If DOMAIN_MIN is omitted from the file, the lower bounds shall be 0 0 0.

If DOMAIN_MAX is omitted from the file, the upper bound shall be 1 1 1.

The lower bounds shall be less than the upper bounds.

5.7 Table Data

Each line of table data shall be formatted as follows:

red <sp> green <sp> blue

where red, green, and blue shall be numbers, and shall be the output values for the Red, Green, and Blue components at the sample point corresponding to this line.

A line of table data may have leading and trailing <sp>.

NOTE: Table values are unconstrained (within the range of numbers), and can be non-monotonic.

5.8 Comments

The file may, at any location, include any number of comment lines.

Each comment line shall be formatted as follows:

<text>

where <text> is arbitrary text or tab characters.

The contents of these lines shall be ignored when parsing the table information, and can be used for comments.

NOTE: A comment line does not have leading <sp>.

6 Cube File With One-Dimensional Tables

6.1 The One-Dimensional Tables

The tables shall consist of three 1-dimensional tables, one each for the Red, Green, and Blue components.

Each table has one input channel and one output channel.

The number of sample points in each table shall be N, for N table rows and a total of 3 × N data values.

N shall be an integer in the range [2, 65536].

The first sample point of each table shall map to the lower bounds of the domain.

The last sample point of each table shall map to the upper bounds of the domain.

Intermediate values in the domain shall map proportionally to the intermediate sample points.

The table need not include a sample point for every possible input value. For input values not included in the table, readers will interpolate between nearby sample points, which are equidistantly spaced.

The values in the table shall be set so that linear interpolation will generate correct output values.

The table should have at most twice as many sample points as needed for desired accuracy after interpolation.

NOTE 1: Some readers deliver optimal performance when N is an integer power of 2.

NOTE 2: The three tables are independent of each other, except for having the same size (N) and being stored in parallel. They can have different domains and implement different transforms.

6.2 The One-Dimensional Cube File

The file shall include a line of text with the following keyword and parameter values:

LUT_1D_SIZE <sp> <nn>

where <nn> shall be the table size N.

The file shall include N lines of table data, each corresponding to a sample point.

The lines of table data shall be in ascending order from first to last sample point.

7 Cube File With A Three-Dimensional Table

7.1 The Three-Dimensional Table

The table shall be a 3-dimensional table. It has three input channels and three output channels.

The number of sample points in each dimension shall be N, for a total of N × N × N sample points, with each sample point having three output values.

N shall be an integer in the range [2, 256].

For each table dimension, the lower and upper bound of the input component's domain shall map to the first and last sample point in that dimension, and intermediate values in the domain shall map proportionally to the intermediate sample points.

The table need not include a sample point for every possible input value. For input values not included in the table, readers will interpolate between nearby sample points, which are equidistantly spaced.

The values in the table shall be set so that tetrahedral interpolation will generate correct output values.

The table should have at most twice as many table rows in each dimension as needed for desired accuracy after interpolation.

NOTE 1: Some readers deliver optimal performance when N is an integer power of 2.

NOTE 2: Some readers do not have enough memory (200 MBytes) to support N = 256.

7.2 The Three-Dimensional Cube File

The file shall include a line of text with the following keyword and parameter values:

LUT_3D_SIZE <sp><nn>

where <nn> shall be the table size N.

The file shall include N × N × N lines of table data, each corresponding to a sample point.

The lines of table data shall be in ascending index order, with the first component index (Red) changing most rapidly, and the last component index (Blue) changing least rapidly.

NOTE: This ordering is the opposite of the typical in-memory order of multi-dimensional tables. An equivalent C index would be r + N * g + N * N * b , where r, g, b are the Red, Green, and Blue indices in the range 0 to N – 1.

8 Application Requirements

The reader shall use interpolation for any input value that is within the domain and that does not map to a sample point. The reader should use linear interpolation for one-dimensional tables, and tetrahedral interpolation for three-dimensional tables.

When converting images with integer-encoded values, the reader shall scale the integer encoding range to the domain 0.0 to 1.0, and shall scale the output range 0.0 to 1.0 to the integer encoding range.

EXAMPLE: Scale 10-bit RGB from [0, 1023] to [0.0, 1.0], and then back to [0, 1023].

NOTE: Floating-point image values are usually in the proper scale, and can be processed without scaling.

Annex A. Sample Cube Files

A.1 Line Separators

The word processor has changed the line separator to \r (CR) in the included sample files. Convert these to \n (LF) before use. The sample application can be used for this conversion.

A.2 One-Dimensional Log To Lin Conversion

This example implements a log to linear conversion, here ACES Proxy 10 to ACES.

The example uses the default domain 0 1.

The input value 1.0 corresponds to the max integer input code, here 1023.

```
TITLE "ACES Proxy 10 to ACES"
LUT_1D_SIZE 32
0.0004883   0.0004883   0.0004883
0.0007714   0.0007714   0.0007714
0.001219   0.001219   0.001219
0.001926   0.001926   0.001926
0.003044   0.003044   0.003044
0.004809   0.004809   0.004809
0.007599   0.007599   0.007599
0.01201   0.01201   0.01201
0.01897   0.01897   0.01897
0.02998   0.02998   0.02998
0.04737   0.04737   0.04737
0.07484   0.07484   0.07484
0.1183   0.1183   0.1183
0.1869   0.1869   0.1869
0.2952   0.2952   0.2952
0.4665   0.4665   0.4665
0.7371   0.7371   0.7371
1.165   1.165   1.165
1.84   1.84   1.84
2.908   2.908   2.908
4.595   4.595   4.595
7.26   7.26   7.26
11.47   11.47   11.47
```

```
18.13   18.13   18.13
28.64   28.64   28.64
45.25   45.25   45.25
71.51   71.51   71.51
113   113   113
178.5   178.5   178.5
282.1   282.1   282.1
445.7   445.7   445.7
704.3   704.3   704.3
```

A.3 One-Dimensional Table With Mixed Domains

This example shows using unique domains for each channel.

The example function is y = x for x <= 1

```
TITLE "Demo"
LUT_1D_SIZE 3
DOMAIN_MIN 0 0 0
DOMAIN_MAX 1 2 3
0 0 0
# Comments can go anywhere
0.5 1 1.5
1 1 1
```

A.4 Three-Dimensional Table

The example function is {r, (3 * g + b)/4.0, b}.

The example uses the default domain 0 1.

Note that the first component (the red channel) changes fastest.

```
# Created Tue 19 Mar 2013
LUT_3D_SIZE 2
0 0 0
1 0 0
0 .75 0
1 .75 0
0 .25 1
1 .25 1
0 1 1
1 1 1
```

Annex B. Sample Application

B.1 Summary

This sample C++ application includes code to read, verify, and write Cube files.

B.2 Header file

```
//      CubeLUT.h

#ifndef CubeLUT_H
#define CubeLUT_H

#include <string>
#include <vector>
#include <fstream>

using namespace std;

class CubeLUT {

public:
      typedef  vector <float>        tableRow;
      typedef  vector <tableRow>     table1D;
      typedef  vector <table1D>      table2D;
      typedef  vector <table2D>      table3D;

      enum    LUTState {      OK = 0, NotInitialized = 1,
        ReadError = 10, WriteError, PrematureEndOfFile, LineError,
        UnknownOrRepeatedKeyword = 20, TitleMissingQuote, DomainBoundsReversed,
        LUTSizeOutOfRange, CouldNotParseTableData };

      LUTState status;
      string   title;
      tableRow domainMin;
      tableRow domainMax;
      table1D  LUT1D;
      table3D  LUT3D;

      CubeLUT   ( void )      { status = NotInitialized; };

      LUTState LoadCubeFile   ( ifstream & infile  );
      LUTState SaveCubeFile   ( ofstream & outfile );

private:
      string   ReadLine       ( ifstream & infile, char lineSeparator);
      tableRow ParseTableRow  ( const string & lineOfText );
};

#endif
//    end CubeLUT.h
```

B.3 Sample Application

```
//      CubeLUT.cpp

#include "CubeLUT.h"
#include <iostream>
#include <sstream>

string                CubeLUT:: ReadLine  ( ifstream & infile, char lineSeparator )
{
      //       Skip empty lines and comments
      const char CommentMarker = '#';
      string   textLine("");
      while ( textLine.size() == 0 || textLine[0] == CommentMarker  ) {
        if ( infile.eof() )           { status = PrematureEndOfFile; break; }
        getline ( infile, textLine, lineSeparator );
        if ( infile.fail() )  { status = ReadError; break; }
      }
      return textLine;
}     //       ReadLine

vector <float> CubeLUT:: ParseTableRow ( const string & lineOfText )
{
      int N = 3;
      tableRow f ( N );
      istringstream line ( lineOfText );
      for (int i = 0; i < N; i++) {
        line >> f[i];
        if ( line.fail() ) { status = CouldNotParseTableData; break; };
      }
      return f;
}     //       ParseTableRow

CubeLUT:: LUTState   CubeLUT:: LoadCubeFile ( ifstream & infile  )
{
      //       Set defaults
      status = OK;
      title.clear();
      domainMin = tableRow ( 3, 0.0 );
      domainMax = tableRow ( 3, 1.0 );
      LUT1D.clear();
      LUT3D.clear();

      //       Read file data line by line
      const char NewlineCharacter = '\n';
      char lineSeparator = NewlineCharacter;

      //       sniff use of legacy lineSeparator
      const char CarriageReturnCharacter = '\r';
      for (int i = 0; i < 255; i++) {
        char inc = infile.get();
        if ( inc == NewlineCharacter ) break;
        if ( inc == CarriageReturnCharacter ) {
                if ( infile.get() == NewlineCharacter ) break;
                lineSeparator = CarriageReturnCharacter;
                clog << "INFO: This file uses non-compliant line separator \\r (0x0D)" << endl;
                break;
        }
        if ( i > 250 ) { status = LineError;  break; }
      }
      infile.seekg ( 0 );
      infile.clear();

      //       read keywords
      int      N, CntTitle, CntSize, CntMin, CntMax;
```

```
    // each keyword to occur zero or one time
    N = CntTitle = CntSize = CntMin = CntMax = 0;

    while ( status == OK ) {
      long linePos = infile.tellg();
      string lineOfText = ReadLine ( infile, lineSeparator );
      if ( ! status == OK ) break;

      //      Parse keywords and parameters
      istringstream line ( lineOfText );
      string keyword;
      line >> keyword;

      if ( "+" < keyword && keyword < ":" ) {
              //      lines of table data come after keywords
              //      restore stream pos to re-read line of data
              infile.seekg ( linePos );
              break;
      } else if ( keyword == "TITLE" && CntTitle++ == 0 ) {
              const char     QUOTE = '"';
              char startOfTitle;
              line >> startOfTitle;
              if ( startOfTitle != QUOTE )  { status = TitleMissingQuote; break; }
              getline ( line, title, QUOTE );               //      read to "
      } else if ( keyword == "DOMAIN_MIN" && CntMin++ == 0 ) {
              line >> domainMin[0] >> domainMin[1] >> domainMin[2];
      } else if ( keyword == "DOMAIN_MAX" && CntMax++ == 0 ) {
              line >> domainMax[0] >> domainMax[1] >> domainMax[2];
      } else if ( keyword == "LUT_1D_SIZE" && CntSize++ == 0 ) {
              line >> N;
              if ( N < 2 || N > 65536 ) { status = LUTSizeOutOfRange; break; }
              LUT1D = table1D ( N, tableRow ( 3 ) );
      } else if ( keyword == "LUT_3D_SIZE" && CntSize++ == 0 ) {
              line >> N;
              if ( N < 2 || N > 256 ) { status = LUTSizeOutOfRange; break; }
              LUT3D = table3D ( N, table2D ( N, table1D ( N, tableRow ( 3 ) ) ) );
      } else  { status = UnknownOrRepeatedKeyword; break; }

      if ( line.fail() ) { status = ReadError; break; }
    } //      read keywords

    if ( status == OK && CntSize == 0 ) status = LUTSizeOutOfRange;

    if ( status == OK && ( domainMin[0] >= domainMax[0] || domainMin[1] >= domainMax[1]
                                      || domainMin[2] >= domainMax[2] ) )
      status = DomainBoundsReversed;

    //       read lines of table data
    if ( LUT1D.size() > 0 ) {
      N = LUT1D.size();
      for ( int i = 0; i < N && status == OK; i++ ) {
              LUT1D [i] = ParseTableRow ( ReadLine ( infile, lineSeparator ) );
      }
    } else {
      N = LUT3D.size();
      //      NOTE that r loops fastest
      for ( int b = 0; b < N && status == OK; b++ ) {
              for ( int g = 0; g < N && status == OK; g++ ) {
                      for ( int r = 0; r < N && status == OK; r++ ) {
                              LUT3D[r][g][b] = ParseTableRow
                              ( ReadLine ( infile, lineSeparator ) );

                      }
              }
      }
    } //      read 3D LUT
```

```
      return status;
}     //       LoadCubeFile



CubeLUT:: LUTState   CubeLUT:: SaveCubeFile ( ofstream & outfile )
{
      if ( ! status == OK ) return status;   // Write only good Cubes

      //       Write keywords
      const char SPACE = ' ';
      const char QUOTE = '"';

      if ( title.size() > 0 ) outfile << "TITLE" << SPACE << QUOTE << title << QUOTE << endl;
      outfile << "# Created by CubeLUT.cpp" << endl;
      outfile << "DOMAIN_MIN" << SPACE << domainMin[0] << SPACE << domainMin[1]
        << SPACE << domainMin[2] << endl;
      outfile << "DOMAIN_MAX" << SPACE << domainMax[0] << SPACE << domainMax[1]
        << SPACE << domainMax[2] << endl;

      //       Write LUT data
      if ( LUT1D.size() > 0 ) {
        int N = LUT1D.size();
        outfile << "LUT_1D_SIZE" << SPACE << N << endl;
        for ( int i = 0; i < N && outfile.good(); i++ ) {
                outfile << LUT1D[i] [0] << SPACE << LUT1D[i] [1] << SPACE << LUT1D[i] [2] << endl;
        }
      } else {
        int N = LUT3D.size();
        outfile << "LUT_3D_SIZE" << SPACE << N << endl;
        //      NOTE that r loops fastest
        for ( int b = 0; b < N && outfile.good(); b++ ) {
                for ( int g = 0; g < N && outfile.good(); g++ ) {
                        for ( int r = 0; r < N && outfile.good(); r++ ) {
                                outfile << LUT3D[r][g][b] [0] << SPACE << LUT3D[r][g][b] [1]
                                << SPACE << LUT3D[r][g][b] [2] << endl;
                        }
                }
        }
      } //      write 3D LUT

      outfile.flush();
      return ( outfile.good() ? OK : WriteError );
}     //       SaveCubeFile
```

```
int main (int argc, char * const argv[])
{
      CubeLUT theCube;
      enum { OK = 0, ErrorOpenInFile = 100, ErrorOpenOutFile };
      if ( argc < 2 || 3 < argc ) {
        cout << "Usage: " << argv[0] << " cubeFileIn [cubeFileOut]" << endl;
        return OK;
      }

      //       Load a Cube
      ifstream infile ( argv[1] );
      if ( !infile.good() ) {
        cout << "Could not open input file " << argv[1] << endl;
        return ErrorOpenInFile;
      }
      int ret = theCube.LoadCubeFile ( infile );
      infile.close();
      if ( ret != OK ) {
        cout << "Could not parse the cube info in the input file. Return code = "
        << ret << endl;
        return theCube.status;
      }

      //       Save a Cube
      if (argc > 2 )
      {
        ofstream outfile ( argv[2], fstream::trunc );
        if ( !outfile.good() ) {
                cout << "Could not open output file " << argv[1] << endl;
                return ErrorOpenOutFile;
        }
        int ret = theCube.SaveCubeFile ( outfile );
        outfile.close();
        if ( ret != OK  ) {
                cout << "Could not write the cube to the output file. Return code = "
                << ret << endl;
                return theCube.status;
        }
      }
      return OK;
}
```

Annex C. Bibliography

Selan, Jeremy (2004). "Using Lookup Tables to Accelerate Color Transformations" *GPU Gems 2*, Chapter 24. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter24.html