8. Color look up tables

In this chapter

- What is look-up table? What is (inverse) look-up operation?
- Preparing color look-up tables
- Building your own custumized object for (inverse) look-up operaration
- Built-in objects for look-up operations in core Java

8.1. What is a look-up table? Why do you need an inverse look-up operation?

When you want to draw anything on your screen, say a uniform gray square at the center of the monitor, you do this by creating an object with spatial attributes (its location and dimensions, etc.) and with a certain pixel value. The position and dimensions of the square are in units of number of pixels, which means that their actual size in centimeters will depend on your monitor. Similarly, the *luminance* of, i.e. the amount of light emitted by, the square will depend not only on the pixel value you assigned, but also on your monitor, its brightness and color settings, and on your video card. In a visual psychophysics experiment you naturally want to make sure that the luminance is under your control, just as you want to make sure the visual angle the stimulus subtends. If you can *assume* that your monitor's output is stable, i.e. the amount of light it emits does not change in time, you can measure its output once and then use that measurement as your reference. The first step is preparing a *look-up table*, a table which contains the luminance of your display corresponding to each pixel value, from 0 to 255. Second, upon deciding what luminance to display on your screen, you check your look-up table to find the pixel value whose luminance is closest to the luminance you desired to display. I refer to this second step as *inverse look-up operation*.

8.1.1. Preparing the look-up table

Computer monitors have three color channels (also called as *gun*, referring to the electron guns in conventional CRT monitors), R, G, and B. Therefore you have to prepare a look-up table for each channel. In principle one would have to measure and record the luminance for each pixel value at every location on the monitor. Not only that, one would also have to measure luminance for each channel with varying the other two channels. Moreover, one would also have to measure luminance at each location in relation to neighboring locations. That is a task almost impossible to perform. Fortunately there are some sound assumptions you can make to reduce the number of measurements. Those assumptions include

- Spatial independence, i.e. output of a pixel at one particular location on the screen doesn't depend on the values of other pixels,
- Channel constancy, i.e. the relative spectrum of a channel is independent of the pixel value,

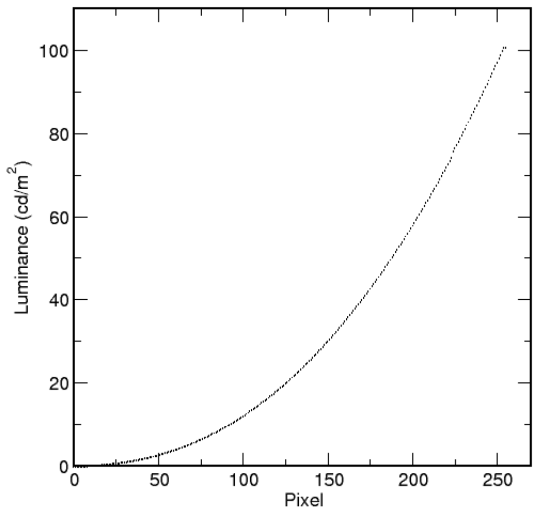

Figure 8.1.: A plot showing how monitor output typically depends on pixel values

- Channel independence, i.e. one channel's output doesn't affect the outputs of other two channels,
- Spatial homogeneity, i.e. the output of pixels do not depend on where they are on the screen.

These assumptions reasonably hold for most modern CRTs (See David H. Brainard, 1989, for more details).

Given that the above conditions are satisfied, you can prepare a look up table by simply presenting a uniform patch on your screen and measuring its luminance using a spectro-photometer for all possible pixel values and for all three color channels. Best is to place the patch at the center of the screen. Also remember that a CRT monitor has to *warm up* for at least half hour before it stabilizes. I had to wait up to 2 hours for some monitors until they finally warmed up completely and the luminance measurements stabilized. You can easily observe it: turn the monitor on and present a very bright patch at the center, (255,255,255). Start measuring the luminance at 5 minute intervals. You will notice that the measurement will fluctuate in the beginning and stabilize later. Only after that you should start the actual measurement. Then you simply measure the luminance for different pixel values for each color channel and store them somewhere (on a piece of paper or better in a file on your hard disk), ideally you make 256x3=768 measurements. It is advisable to repeat each measurement 3 times and to take the average and inspect the standard deviations. This is your look-up table. (Note: CRTs are most unstable at highest and lowest ends of their output range, they are more stable around middle values.) See Figure 8.1.

Although you normally have to prepare a separate look-up table for each channel (R,G,B) you can get away with only one table if the output of each channel is exact same. This is also not an uncommon property for modern computer monitors. Manufacturers pay attention to keeping the chromaticity of "gray" same throughout the entire pixel range (say D65). A fortunate consequence of this is that the contribution from all three channels vary in exact same way to maintain a constant gray, apart from its intensity.

A further reduction in number of measurements is possible. You can limit the number of measurements to every other 5th, 10th or even 20th pixel value, and then either fit a function to the data or use an interpolation algorithm to assign values to those pixel values you didn't measure. The functional relation between the pixel value and the luminance can be represented as

$$L(p) = f(p) \ \ p = 0, ..., 255$$

where L is the luminance value, p is the pixel value, for example a common form is

$$f(p) = b + g * p^{\gamma},$$

where b, g, and γ are free parameters determined by fitting a curve to the data at hand (b is usually referred to as *bias,* g as *gain.*) For interpolation, linear or cubic spline are usually sufficient. (Matlab users: use fminsearch() method for function fit, interp1() method for interpolation, or see Chapter XXX on numerical methods in this guide - *Coming soon!*). Note that in the functional form where there will always be some disagreement between $f(p)$ and the actual luminance presented on the screen. The function will even approximate the values you measured, and alter them. In case of interpolation those values you measured remain untouched, only the values you didn't measure are approximated. On the other hand the functional form may sometimes be advantageous because finding the inverse of a function may be easier than "inverting" a table. In the rest of this chapter, however, I will focus only on the tabular form as it is the more accurate. Whether functional fit or interpolation, you store the luminance values corresponding to entire range of pixel values at some safe place for later use. Later when you need, you export the data and store the table in an array, say pix2Lum - pixel to luminance table

$$L(p) = pix2Lum[p], \ \ p = 0, ..., 255.$$

(Maybe more on functional form later?)

8.2. Inverse operation: finding out which pixel gives the desired luminance

Suppose that the luminance value you want to show on the screen is Lum. You must first understand that you may not be able to present this exact value on the screen because of the limited number of pixel values. What you do is try and find the pixel whose luminance is closest to the desired value. Let's denote it by p^*

$$p^* = \arg\min_p (pix2Lum[p] - Lum), \ \ p = 0 \ldots 255.$$

This is the essence of what I call the inverse look-up operation. This operation would be costly if you repeat it every time for every pixel on the screen, searching the entire 0 to 255 range. However there are some ways to simplify it. I will show them step by step below.

First, let's normalize the luminance values to fit the range from 0 to 255 for each pixel

$$Lum(p) \rightarrow Lum(p)/MaxLum * 255,$$

where MaxLum is the maximum possible luminance. The resulting $pix2Lum[p]$ array is plotted in Figure 8.2.

Second, given the normalized table it is possible to determine p^* for integer (normalized) luminance values

$$p^* = \arg\min_p (pix2Lum[p] - Lum), \ \ \text{where } Lum \text{ is integer}$$

and store this in a new inverse look-up table, say lum2Pix[lum] - luminance-to-pixel table. You can just use this table for your inverse look up operation and this wouldn't be such a bad approximation. The result is plotted in Figure 8.3. But how accurate is this inverse look-up table? If you examine the tableGray255.txt file in this directory (it holds the normalized luminance values), you will notice that the spacing between

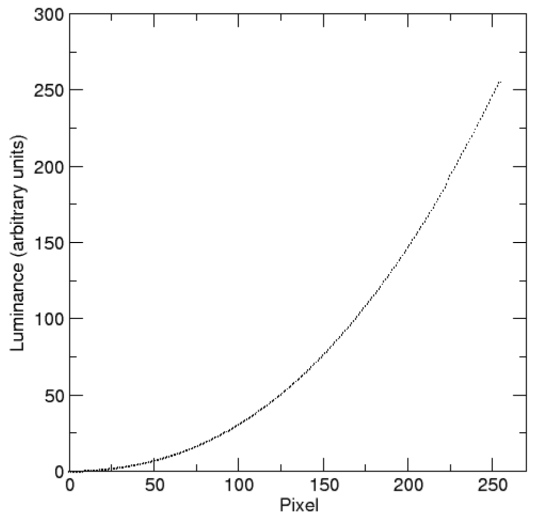

Figure 8.2.: Normalized look-up table. pix2Lum[p] array, p=0..255.

the luminance values are usually larger than 1, but what happens when this isn't the case? Consider the following situation: suppose that you want to display a value of 49.40 in relative luminance. Look at the tableGray255.dat file, the closest luminance value in the table is 49.52. But if you are using the inverse look up table (lum2Pix[l] array), you will have to show a relative luminance value closest to 49 (because 49.40 rounds to 49), which is 48.60, not 49.52! Of course it is more accurate to use 49.52, not 48.60. In case you want more accuracy, you still have to go back to the original equation and find the p^* whose luminance is closest to your desired, floating value, luminance.

Fortunately, even if you prefer to be more accurate, the pix2Lum[] array you created will considerably reduce your computational expenses. Again, suppose that you want to display a luminance of 49.40. Assuming monotonicity, i.e. luminance is a monotonically increasing function of pixel values (or decreasing for that matter), you can restrict the search to the neighborhood of 49.40, for example from $p_{start} = lum2Pix[48]$ to $p_{end} = lum2Pix[50]$. This will reduce the search range from 256 to only 4 in this example.

Let's go back and look at the implementation of each of the above steps. After normalizing the luminance values, we construct the crude inverse look-up table as follows, assuming monotonically increasing form here,

```
int DIM = 256;
//...
int lastPixel = 0;
for(int lum=0; lum<DIM; lum++){
  double diff = Double.MAX_VALUE;
  for(int j=lastPixel; j<DIM; j++){
    double diffTmp = abs(pix2Lum[j]-lum);
    if(diffTmp == 0){
      lum2Pix[lum] = j;
      lastPixel = j;
      break;
```

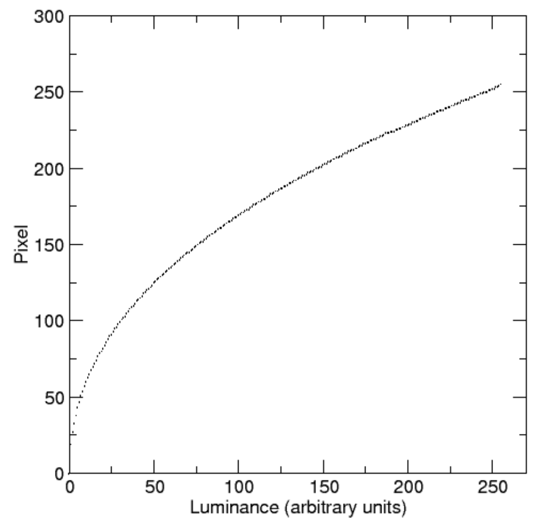

Figure 8.3.: Inverse look-up table. lum2Pix[l] array, l=0..255 (integer).

```
      }
      else if(diffTmp <= diff){
        lum2Pix[lum] = j;
        lastPixel = j;
        diff = diffTmp;
      }
      else
        break;
    }
  }
```

Similar algorithms can be implemented for monotonically decreasing tables (see next section). For tables neither monotonically increasing nor decreasing, you have to live without the simplifications. The more accurate inverse look-up operation can be done as follows, assuming monotonicity

```
    // lum is the desired luminance
    int intLum = (int)round(lum);
    int pixel = lum2Pix[intLum];

    if(lum == (double)intLum)
      return pixel;
    else {
      int pixelStart = lum2Pix[max(intLum-1,0)];
      int pixelEnd = lum2Pix[min(intLum+1,DIM-1)];
      double diff = Double.MAX_VALUE;
      for(int j=pixelStart; j<=pixelEnd; j++){
        double diffTmp = abs(pix2Lum[j] - lum);
        if(diffTmp == 0){
          pixel = j;
          break;
        }
        else if(diffTmp <= diff){
          pixel = j;
          diff = diffTmp;
        }
        else
          break;
      }
      return pixel;
    }
```

I will show the entire code below in the next section.

8.2.1. A class to perform inverse look-up operation

In this section I will build a class, CLUT8, for inverse look-up operations. CLUT8 has two constructors. You can construct a CLUT8 object either by providing a file name which holds the output luminance values for pixels from 255 to 0 in descending order, or you provide a double array which holds the output values for each pixel value. Both constructors invoke the setClut() method to prepare the look-up table, i.e. pix2Lum array, and the crude inverse look-up table, i.e. lum2Pix array. First, the constructor with the file name:

```
public CLUT8(String filename) throws IllegalArgumentException{
  InputStream stream = CLUT8.class.getResourceAsStream(filename);
  Scanner in = new Scanner(stream);
  maxLum = 0;
  double[] table = new double[DIM];
  for (int pix = DIM - 1; pix > -1; pix--) {
    try {
      table[pix] = in.nextDouble();
    } catch (NoSuchElementException e) {
      throw new IllegalArgumentException(
          "Error in CLUT8(String): wrong input file - file too short");
    }
  }
  if (in.hasNext())
    throw new IllegalArgumentException(
        "Error in CLUT8(String): suspicious input file - file too long");
  in.close();
  setClut(table);
}
```

The constructor makes sure that the file contains exactly 256 double valued numbers (see the tableGray.txt file provided in the same directory.) In case the number of elements is not equal to 256 (=DIM), the program throws an exception. The other constructor is similar, except the user provides the data in a double array

```
public CLUT8(double[] table) throws IllegalArgumentException{
    if (table.length != DIM)
      throw new IllegalArgumentException(
          "Error in CLUT8(double[]): incompatible data");
    else
      setClut(table);
  }
```

Both constructors invoke the setClut() method. setClut() method normalizes the look up table and prepares the inverse look up table. It utilizes different approaches depending on whether the table is monotonically increasing or decreasing, or not monotonic at all. In case it is not monotonic, it warns the user but not throws an exception.

```
  public void setClut(double[] table){

    if (table.length != DIM)
      throw new IllegalArgumentException(
          "Error in CLUT8.setClut(double[]): incompatible data");

    maxLum = 0;
    pix2Lum = table;
    double lastLum = pix2Lum[0];
    for (double lum : pix2Lum) {
      maxLum = max(maxLum, lum);
      if (monotonicIncrease && lastLum > lum)
        monotonicIncrease = false;
      else if(monotonicDecrease && lastLum < lum)
        monotonicDecrease = false;
      lastLum = lum;
    }

    if(!monotonicIncrease && !monotonicDecrease){
      System.err
      .println("Warning in CLUT8.setClut(double[]): "
          + "LUT is not monotonically increasing or decreasing, "
          + "speed may degrade");
    }

    for (int pix = 0; pix < DIM; pix++)
      pix2Lum[pix] *= 255 / maxLum;

    if(monotonicIncrease){
      int lastPixel = 0;
      for (int lum = 0; lum < DIM; lum++) {
        double diff = Double.MAX_VALUE;
        for (int j = lastPixel; j < DIM; j++) {
          double diffTmp = abs(pix2Lum[j] - lum);
          if (diffTmp == 0) {
```

```
        lum2Pix[lum] = j;
        lastPixel = j;
        break;
      }
      else if (diffTmp <= diff) {
        lum2Pix[lum] = j;
        lastPixel = j;
        diff = diffTmp;
      }
      else
        break;
    }
  }
}
else if(monotonicDecrease){
  int lastPixel = 0;
  for (int lum = DIM-1; lum >= 0; lum--) {
    double diff = Double.MAX_VALUE;
    for (int j = lastPixel; j < DIM; j++) {
      double diffTmp = abs(pix2Lum[j] - lum);
      if (diffTmp == 0) {
        lum2Pix[lum] = j;
        lastPixel = j;
        break;
      }
      else if (diffTmp <= diff) {
        lum2Pix[lum] = j;
        lastPixel = j;
        diff = diffTmp;
      }
      else
        break;
    }
  }
}
else{
  for (int lum = 0; lum < DIM; lum++) {
    double diff = Double.MAX_VALUE;
    for (int j = 0; j < DIM; j++) {
      double diffTmp = abs(pix2Lum[j] - lum);
      if (diffTmp == 0) {
        lum2Pix[lum] = j;
        break;
      }
      else if (diffTmp <= diff) {
        lum2Pix[lum] = j;
        diff = diffTmp;
      }
```

```
          }
        }
      }
    }
```

Note that setClut() is a public method and the user can invoke it as well. This would be useful if you do a look up table animation (see Chapter XX on Bits++).

There in one method to obtain the outcome of inverse look up operation, it is lum2Pix() method. However this method is overloaded. If you invoke it with an integer value, it simply returns that element of lum2Pix[] array

```
    public int lum2Pix(int lum) {

      return lum2Pix[lum];
    }
```

if you invoke it with a double argument, then it performs the more accurate inverse look up operation as described above

```
    public int lum2Pix(double lum) {
      int intLum = (int) round(lum);
      int pixel = lum2Pix[intLum];
      if (lum == (double) intLum)
        return pixel;
      else {
        if(monotonicIncrease){
          int pixelStart = lum2Pix[max(intLum - 1, 0)];
          int pixelEnd = lum2Pix[min(intLum + 1, DIM - 1)];
          double diff = Double.MAX_VALUE;
          for (int j = pixelStart; j <= pixelEnd; j++) {
            double diffTmp = abs(pix2Lum[j] - lum);
            if (diffTmp == 0) {
              pixel = j;
              break;
            }
            else if (diffTmp <= diff) {
              pixel = j;
              diff = diffTmp;
            }
            else
              break;
          }
        }
        else if(monotonicDecrease){
          int pixelStart = lum2Pix[max(intLum + 1, 0)];
          int pixelEnd = lum2Pix[min(intLum - 1, DIM - 1)];
          double diff = Double.MAX_VALUE;
          for (int j = pixelStart; j <= pixelEnd; j++) {
            double diffTmp = abs(pix2Lum[j] - lum);
            if (diffTmp == 0) {
              pixel = j;
              break;
            }
            else if (diffTmp <= diff) {
              pixel = j;
              diff = diffTmp;
            }
            else
              break;
          }
        }
        else {
          double diff = Double.MAX_VALUE;
          for (int j = 0; j < DIM; j++) {
            double diffTmp = abs(pix2Lum[j] - lum);
            if (diffTmp == 0) {
              pixel = j;
              break;
            }
            else if (diffTmp <= diff) {
              pixel = j;
              diff = diffTmp;
            }
          }
        }
        return pixel;
      }
    }
```

Here are the other methods of the class. To inquire the maximum luminance in your look up table use getMaxLum() method

```
public double getMaxLum() {

  return maxLum;
}
```

To obtain the relative luminance of a pixel

```
  public double pix2Lum(int pixel){

    return pix2Lum[pixel];
  }
```

Here is the entire code of CLUT8 class

```
/*
 * Chapter 8: CLUT8.java
 *
 * Provides methods to perform (inverse) look up operations.
 *
 */
import static java.lang.Math.*;
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.event.KeyEvent;
import java.awt.image.BufferedImage;
import java.io.InputStream;
import java.util.NoSuchElementException;
import java.util.Scanner;
public class CLUT8 {
  private final static int DIM = 256;
  private double maxLum;
  private double[] pix2Lum = new double[DIM];
  private int[] lum2Pix = new int[DIM];
  private boolean monotonicIncrease = true;
  private boolean monotonicDecrease = true;
  public CLUT8(String filename) throws IllegalArgumentException{
    InputStream stream = CLUT8.class.getResourceAsStream(filename);
    Scanner in = new Scanner(stream);
    maxLum = 0;
    double[] table = new double[DIM];
    for (int pix = DIM - 1; pix > -1; pix--) {
      try {
        table[pix] = in.nextDouble();
      } catch (NoSuchElementException e) {
        throw new IllegalArgumentException(
            "Error in CLUT8(String): wrong input file - file too short");
      }
    }
    if (in.hasNext())
      throw new IllegalArgumentException(
          "Error in CLUT8(String): suspicious input file - file too long");
    in.close();
    setClut(table);
  }
  public CLUT8(double[] table) throws IllegalArgumentException{
    if (table.length != DIM)
      throw new IllegalArgumentException(
          "Error in CLUT8(double[]): incompatible data");
    else
      setClut(table);
  }

  public void setClut(double[] table){
```

```
if (table.length != DIM)
  throw new IllegalArgumentException(
      "Error in CLUT8.setClut(double[]): incompatible data");

maxLum = 0;
pix2Lum = table;
double lastLum = pix2Lum[0];
for (double lum : pix2Lum) {
  maxLum = max(maxLum, lum);
  if (monotonicIncrease && lastLum > lum)
    monotonicIncrease = false;
  else if(monotonicDecrease && lastLum < lum)
    monotonicDecrease = false;
  lastLum = lum;
}

if(!monotonicIncrease && !monotonicDecrease){
  System.err
  .println("Warning in CLUT8.setClut(double[]): "
      + "LUT is not monotonically increasing or decreasing, "
      + "speed may degrade");
}

for (int pix = 0; pix < DIM; pix++)
  pix2Lum[pix] *= 255 / maxLum;

if(monotonicIncrease){
  int lastPixel = 0;
  for (int lum = 0; lum < DIM; lum++) {
    double diff = Double.MAX_VALUE;
    for (int j = lastPixel; j < DIM; j++) {
      double diffTmp = abs(pix2Lum[j] - lum);
      if (diffTmp == 0) {
        lum2Pix[lum] = j;
        lastPixel = j;
        break;
      }
      else if (diffTmp <= diff) {
        lum2Pix[lum] = j;
        lastPixel = j;
        diff = diffTmp;
      }
      else
        break;
    }
  }
}
else if(monotonicDecrease){
    int lastPixel = 0;
    for (int lum = DIM-1; lum >= 0; lum--) {
      double diff = Double.MAX_VALUE;
      for (int j = lastPixel; j < DIM; j++) {
        double diffTmp = abs(pix2Lum[j] - lum);
        if (diffTmp == 0) {
          lum2Pix[lum] = j;
          lastPixel = j;
          break;
        }
        else if (diffTmp <= diff) {
          lum2Pix[lum] = j;
          lastPixel = j;
          diff = diffTmp;
        }
        else
          break;
      }
    }
  }
  else{
    for (int lum = 0; lum < DIM; lum++) {
      double diff = Double.MAX_VALUE;
      for (int j = 0; j < DIM; j++) {
        double diffTmp = abs(pix2Lum[j] - lum);
        if (diffTmp == 0) {
          lum2Pix[lum] = j;
          break;
        }
        else if (diffTmp <= diff) {
          lum2Pix[lum] = j;
          diff = diffTmp;
        }
      }
    }
  }
}

public double getMaxLum() {
  return maxLum;
}

public int lum2Pix(double lum) {
  int intLum = (int) round(lum);
  int pixel = lum2Pix[intLum];
  if (lum == (double) intLum)
    return pixel;
  else {
```

```
if(monotonicIncrease){
  int pixelStart = lum2Pix[max(intLum - 1, 0)];
  int pixelEnd = lum2Pix[min(intLum + 1, DIM - 1)];
  double diff = Double.MAX_VALUE;
  for (int j = pixelStart; j <= pixelEnd; j++) {
    double diffTmp = abs(pix2Lum[j] - lum);
    if (diffTmp == 0) {
      pixel = j;
      break;
    }
    else if (diffTmp <= diff) {
      pixel = j;
      diff = diffTmp;
    }
    else
      break;
  }
}
else if(monotonicDecrease){
  int pixelStart = lum2Pix[max(intLum + 1, 0)];
  int pixelEnd = lum2Pix[min(intLum - 1, DIM - 1)];
  double diff = Double.MAX_VALUE;
  for (int j = pixelStart; j <= pixelEnd; j++) {
    double diffTmp = abs(pix2Lum[j] - lum);
    if (diffTmp == 0) {
      pixel = j;
      break;
    }
    else if (diffTmp <= diff) {
      pixel = j;
      diff = diffTmp;
    }
    else
      break;
  }
}
else {
  double diff = Double.MAX_VALUE;
  for (int j = 0; j < DIM; j++) {
    double diffTmp = abs(pix2Lum[j] - lum);
    if (diffTmp == 0) {
      pixel = j;
      break;
    }
    else if (diffTmp <= diff) {
      pixel = j;
      diff = diffTmp;
    }
```

```
      }
    }
    return pixel;
  }
}

public int lum2Pix(int lum) {
  return lum2Pix[lum];
}

public double pix2Lum(int pixel) {
  return pix2Lum[pixel];
}

public static void main(String[] args) {

  FullScreen fs = new FullScreen();
  try {
    // Choose one of the two constructors
    //CLUT8 lut8 = new CLUT8("tableGray.txt");
    double[] table = new double[256];
    for (int i = 0; i < 256; i++)
      table[i] = 1.0 - 1.0 * pow((i / 255.0), 1.0);

    CLUT8 lut8 = new CLUT8(table);

    BufferedImage rectangle = new BufferedImage(fs.getWidth() / 2, fs
        .getHeight() / 2, BufferedImage.TYPE_BYTE_GRAY);
    double maxLum = lut8.getMaxLum();
    double lum = 127.5;
    int pix = lut8.lum2Pix(lum);
    fs.blankScreen();
    fs.displayText(20, 50, "Use arrow keys to change the luminance");
    fs.displayText(20, 100, "(Press any key to continue, press ESC to quit)");
    fs.updateScreen();
    while (true) {
      int res = fs.getKeyPressed(-1);
      if (res == KeyEvent.VK_UP)
        lum += 0.5;
      else if (res == KeyEvent.VK_DOWN)
        lum -= 0.5;
      else if (res == KeyEvent.VK_LEFT)
        lum -= 10;
      else if (res == KeyEvent.VK_RIGHT)
        lum += 10;
      lum = Math.max(0, lum);
      lum = Math.min(255, lum);
      pix = lut8.lum2Pix(lum);
```

```
            Graphics2D g = rectangle.createGraphics();
            g.setColor(new Color(pix, pix, pix));
            g.fillRect(0, 0, rectangle.getWidth(), rectangle.getHeight());
            g.dispose();
            fs.blankScreen();
            fs.displayImage(rectangle);
            fs.displayText(10, 50, "Desired relative Luminance was: " + lum);
            fs.displayText(10, 120, "Displayed relative Luminance is: "
                + lut8.pix2Lum(pix));
            fs.displayText(10, fs.getHeight() - 40, "(Actual luminance : "
                + lut8.pix2Lum(pix) * maxLum / 255.0 + " cd/m2)");
            fs.updateScreen();
          }
        }
        catch (IllegalArgumentException e){
          e.printStackTrace();
        }
        finally {
          fs.closeScreen();
        }
      }
    }
```

Note that in this class I included a main() method to test it. Including main() method like this one is a useful strategy to test your classes quickly. You can execute the LUT8 class as usual and inspect how the luminance values shown on your screen and the desired values differ.

8.3. Example: (inverse) Look-up operation on an image

One often stores stimuli in image files of standard formats, for example jpeg or png. In case you have such an image, whose pixel values are to be treated as luminance values, and want to apply an inverse look up operation, you can use the CLUT8 class. Note that in case of standard images you will always be using the integer valued relative luminances. Here is sample program to show how you can filter your images by an inverse look up operation

```
/*
 * chapter 8: CLUT8Test.java
 *
 * demonstrates how to apply look up operation on an image using CLUT8
 *
 */
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
public class CLUT8Test {
```

```
public static void main(String[] args){

  FullScreen fs = new FullScreen();

  try {
    BufferedImage bi = ImageIO.read(new File("fechner.png"));

    fs.displayImage((fs.getWidth()/2-bi.getWidth())/2,
        (fs.getHeight()-bi.getHeight())/2, bi);

    double[] table = new double[256];
    for(int i=0; i<256; i++)
      table[i]=255-i;

    CLUT8 lut8 = new CLUT8(table);

    bi = filter(lut8,bi);

    fs.displayImage((fs.getWidth()+bi.getWidth())/2,
        (fs.getHeight()-bi.getHeight())/2, bi);

    fs.displayText(15,fs.getHeight()-15,"(press any key to finish)");

    fs.updateScreen();
    fs.getKeyPressed(-1);

  } catch (IOException e) {
    e.printStackTrace();
  } finally {
    fs.closeScreen();
  }

}

public static BufferedImage filter(CLUT8 lut8, BufferedImage bi){

  BufferedImage result = new BufferedImage(bi.getWidth(), bi.getHeight(),
      BufferedImage.TYPE_BYTE_GRAY);
  Graphics2D g = result.createGraphics();
  g.drawImage(bi,0,0,null);
  g.dispose();
  if(bi.getType() != BufferedImage.TYPE_BYTE_GRAY){
    System.err.println("Image must be gray scale, cannot filter");
    return result;
  }
  else {
    int[] gResult = new int[result.getWidth() * result.getHeight()];
    int[] gBi = new int[bi.getWidth() * bi.getHeight()];
```

```
            bi.getRaster().getPixels(0,0,bi.getWidth() , bi.getHeight(), gBi);
            for (int j = 0; j<gBi.length; j++)
              gResult[j] = lut8.lum2Pix(gBi[j]);

            result.getRaster().setPixels(0, 0, result.getWidth() ,
                result.getHeight(), gResult);
            return result;
          }
        }
      }
```

This example demonstrates how you can use CLUT8 class to perform image filtering. However, see below the section on Standard Java utilities for look up operations. The utilities provided by Standard Java should be preferred for image look up operations.

8.4. Experiment: Cornsweet illusion

In this section I will show how to use CLUT8 in an actual experiment. In this experiment we want to measure the magnitude of the brightness illusion in Craik-O'Brien-Cornsweet stimulus. In the Crack-O'Brien-Cornsweet stimulus two flanking territories of equal luminance are perceived to have different luminances because of a contrast border between them. See Figure 8.4. In this experiment we want to psychophysically determine the magnitude of this illusion in a systematic way and find out how the illusion depends on the contrast at the border.

To determine the magnitude of illusion we use two interval forced choice method (2IFC). The experiment is designed as follows: There are two intervals in which we present stimuli, temporally separated from each other. One of the stimuli is the Craik-O'Brein-Cornsweet stimulus, the other one is a "real" stimulus. The "real" stimulus is composed of two flanking territories with un-equal luminances and a contrast border at the center. Observer's task is to indicate the interval in which the perceived luminance difference was larger between the two f1anks. To make the experiment more controlled, we superimpose two square frames on the flanks, and ask the observer to make the judgment by comparing the luminance within the square frames. We use three fixed contrast levels: 0.3, 0.6, and 0.9. For each contrast level we seek to find the corresponding perceptually equivalent real stimulus. The difference between the flanks of that perceptually equivalent real stimulus will be assigned as the magnitude of the illusion. Every time the observer responds that the illusory stimulus has larger difference between its flanks, the program increases the contrast of the real stimulus for the next trial, otherwise the next trial has a lower real contrast. This is a 1 up 1 down staircase procedure. The number of trials for each staircase is fixed, then the data can easily be analyzed to determine the subjectively equivalent real stimulus for each contrast level. See Figure 8.5.

Let's start with the methods which create the illusory and real stimuli. The real stimulus is created as follows

```
    public BufferedImage prepReal(int polarity, double contrast) {
      // Luminances of Uniform flanks
      double meanLeft = meanLum * (1 + contrast / 2 * polarity);
      double meanRight = meanLum * (1 - contrast / 2 * polarity);
      int[] gStim = new int[stimWidth * stimHeight];
      // Convert the Luminance to Pixel using the LUT
      for (int j = 0; j < stimWidth / 2; j++)
        gStim[j] = lut8.lum2Pix(meanLeft);
```

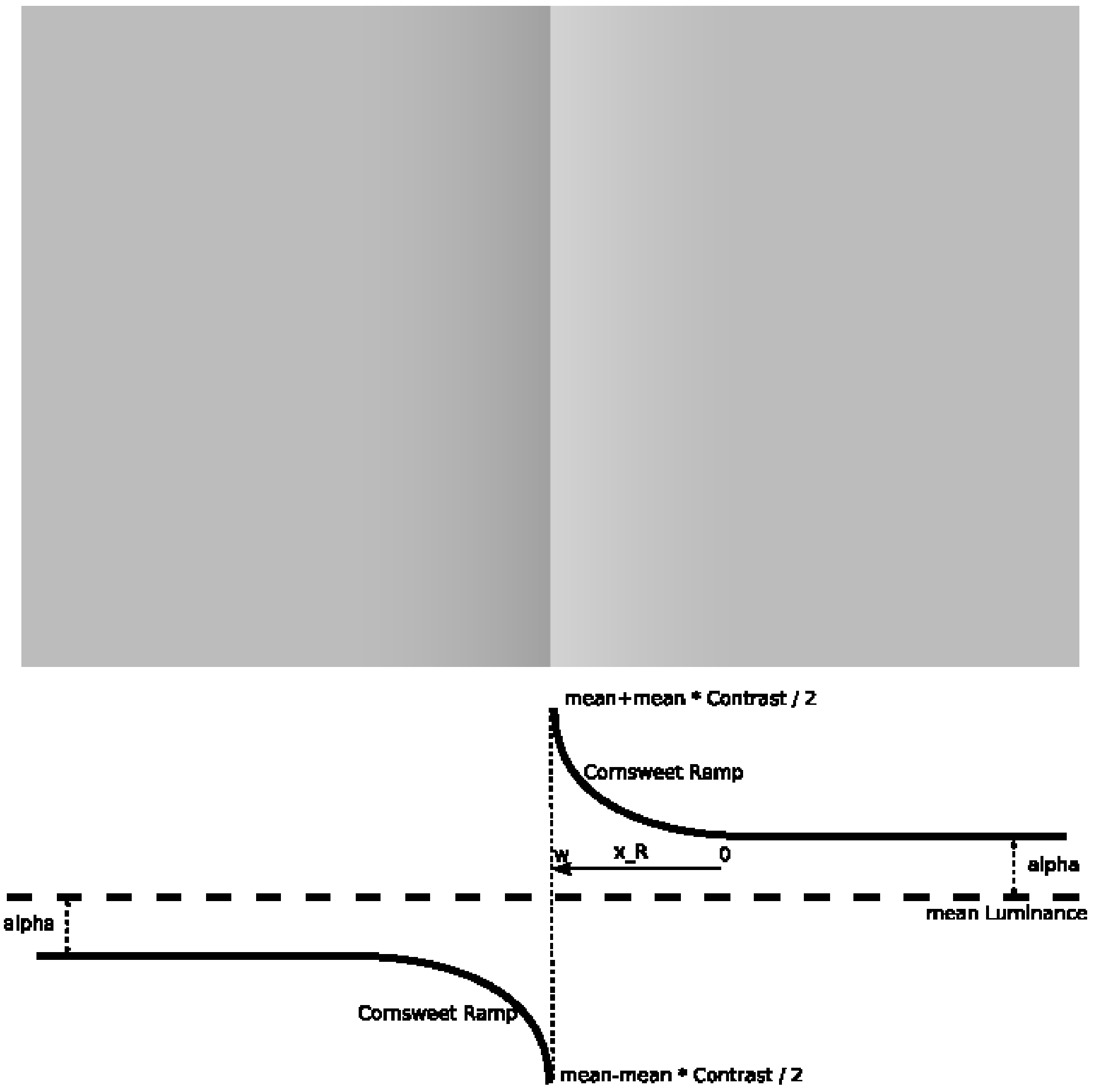

Figure 8.4.: Craik-O'Brein-Cornsweet illusion. Cornsweet luminance profile. The equation of the Right ramp is: $L_R = (mean + \alpha) + (mean * Contrast/2 - \alpha) * (\frac{x_R}{w})^{\xi}$. This gives us the flexibility to choose $Contrast$,α and ξ. $\alpha = 0$ in the original Cornsweet stimulus at the top.

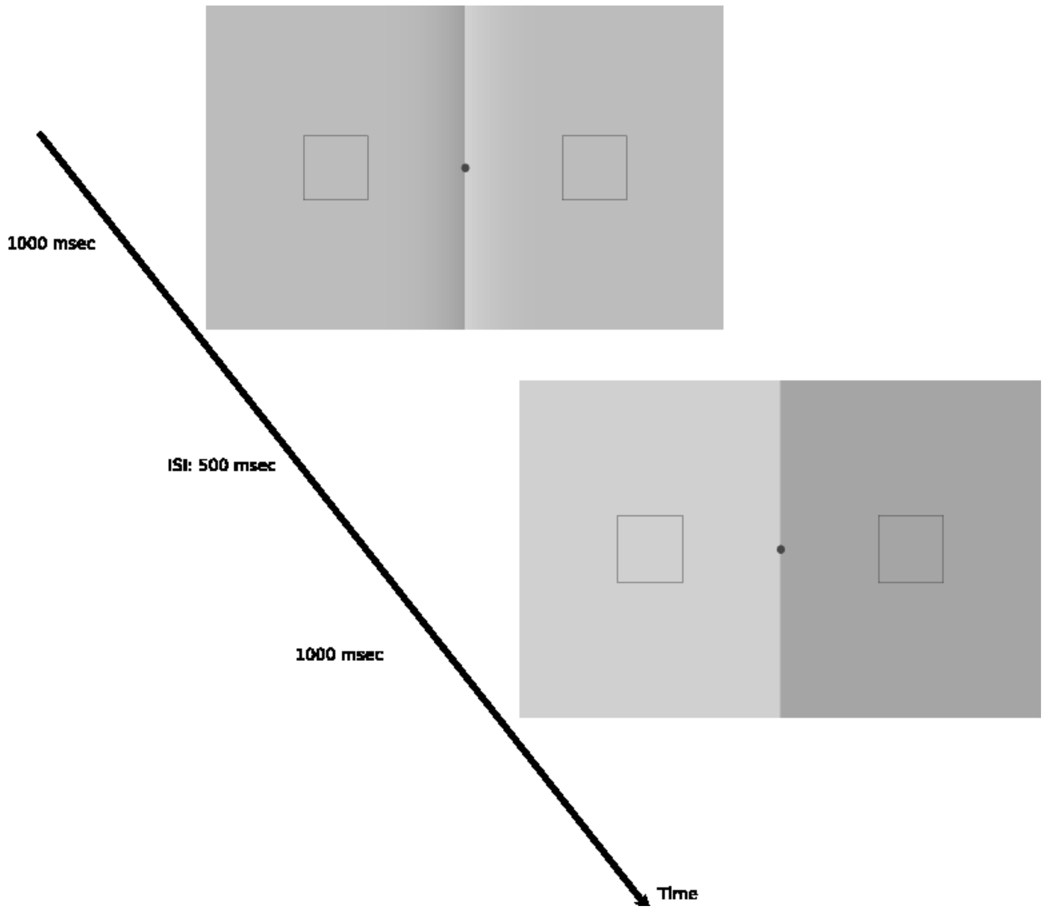

Figure 8.5.: Experimental design. Observer's task is to indicate the interval with a larger difference between the flanks. The thin frames are provided to help the observers with the task. Observers were required to fixate at the center. A staircase procedure is used with one up one down rule, this results in a subjective equality between illusory and actual luminance variation. During ISI the screen is blank except the fixation mark. In order to eliminate cognitive effects for the non-naive observers we include trials with positive $\alpha$where there are actual differences between the flanks (always in the direction of illusion, not in the nulling direction) of Cornsweet stimulus.

```
    for (int j = stimWidth / 2; j < stimWidth; j++)
      gStim[j] = lut8.lum2Pix(meanRight);
    for (int i = 1; i < stimHeight; i++)
      for (int j = 0; j < stimWidth; j++)
        gStim[i * stimWidth + j] = gStim[j];
    // write the result in an image
    BufferedImage stim = new BufferedImage(stimWidth, stimHeight,
        BufferedImage.TYPE_BYTE_GRAY);
    stim.getRaster().setPixels(0, 0, stimWidth, stimHeight, gStim);
    return stim;
  }
```

If polarity=1, the left flank is brighter, if polarity=-1 then the right flank is brighter. This method returns a BufferedImage that you can easily display on the screen using the displayImage() method of FullScreen class. And for the illusory stimulus

```
  public BufferedImage prepAsymCorn(int polarity, double alpha,
      double contrast) {
    // Luminances of Uniform part of flanks
    double meanLeft = meanLum + alpha * polarity;
    double meanRight = meanLum - alpha * polarity;
    // Convert the Luminance to Pixel using the LUT
    int[] gStim = new int[stimWidth * stimHeight];
    for (int j = 0; j < stimWidth / 2 - rampWidth; j++)
      gStim[j] = lut8.lum2Pix(meanLeft);
    for (int j = stimWidth / 2 - rampWidth; j < stimWidth; j++)
      gStim[j] = lut8.lum2Pix(meanRight);
    // Luminances of Cornsweet2AFC ramps
    double delta = contrast / 2 * meanLum;
    for (int j = 0; j < rampWidth; j++) {
      // Left ramp
      int k = j + stimWidth / 2 - rampWidth;
      double tmp = pow((double) j / (double) (rampWidth - 1), exponent)
          * (-alpha + delta) * polarity;
      gStim[k] = lut8.lum2Pix(meanLeft + tmp);
      // Right ramp
      k = stimWidth / 2 + rampWidth - j - 1;
      gStim[k] = lut8.lum2Pix(meanRight - tmp);
    }
    for (int i = 1; i < stimHeight; i++)
      for (int j = 0; j < stimWidth; j++)
        gStim[i * stimWidth + j] = gStim[j];
    // write the result in an image
    BufferedImage stim = new BufferedImage(stimWidth, stimHeight,
        BufferedImage.TYPE_BYTE_GRAY);
    stim.getRaster().setPixels(0, 0, stimWidth, stimHeight, gStim);
    return stim;
  }
```

Even though we assign some fixed contrast levels, the actual displayed contrast may not always be equal to those desired values, therefor we first check what actually the contrast and α values of the Craik-O'Brein-Cornsweet stimuli are

```
// compute the actual alpha, and contrastCorn values:
BufferedImage tmp;
for (int i = 0; i < contrastCorn.length; i++) {
  tmp = prepAsymCorn(1, 0, contrastCorn[i]);
  System.err.println("Requested contrastCorn = " + contrastCorn[i]);
  int[] px = new int[tmp.getWidth()];
  tmp.getRaster().getPixels(0, 0, tmp.getWidth() - 1, 1, px);
  contrastCorn[i] = (lut8.pix2Lum(px[tmp.getWidth() / 2 - 1]) - lut8
      .pix2Lum(px[tmp.getWidth() / 2]))
      / (lut8.pix2Lum(px[tmp.getWidth() / 2 - 1]) + lut8.pix2Lum(px[tmp
          .getWidth() / 2])) * 2;
  System.err.println("Received contrastCorn = " + contrastCorn[i]);
}
for (int i = 0; i < alphaCorn.length; i++) {
  tmp = prepAsymCorn(1, alphaCorn[i], 0);
  System.err.println("Requested alphaCorn = " + alphaCorn[i]);
  int[] px = new int[tmp.getWidth()];
  tmp.getRaster().getPixels(0, 0, tmp.getWidth() - 1, 1, px);
  alphaCorn[i] = (lut8.pix2Lum(px[1]) - lut8
      .pix2Lum(px[tmp.getWidth() - 2])) / 2;
  System.err.println("Received alphaCorn = " + alphaCorn[i]);
}
```

The trial order is taken care as follows: We create two HashMaps each holding the parameters of the illusory stimuli and the real stimuli respectively. In both HashMaps the keys are trial numbers, values are the parameters:

```
// trial initialization: corn holds the
// (KEY=)trial number - (VALUE=)cornsweet parameters (contrast,alpha)
HashMap<Integer, ArrayList<Double> > corn =
  new HashMap<Integer, ArrayList<Double> >();
// real holds the (KEY=)trial number - (VALUE=)real contrast
HashMap<Integer, Double> real = new HashMap<Integer, Double>();
// nTrial holds the entire trial number in a List
List<Integer> nTrial = new LinkedList<Integer>();
int t = 0;
for (int j = 0; j < contrastCorn.length; j++) {
  for (int k = 0; k < alphaCorn.length; k++) {
    ArrayList<Double> tmp = new ArrayList<Double>();
    tmp.add(contrastCorn[j]);
    tmp.add(alphaCorn[k]);
    corn.put(t, tmp);
    real.put(t, initContrast[k][j]);
    nTrial.add(t);
```

```
      t++;
    }
  }
```

I initialize the contrast of real stimuli deterministically, rather than randomly. It is done as follows, I create two staircases for each contrast value, one starting from very high initial real contrast values, one from very low. Of course first few trials of them will be trivial for the observer, but this strategy usually results in a data set which is easier to fit a psychometric function. Note that we also create a List for trial numbers. The next piece of code below shows how you randomize the order of trials and then extract the corresponding parameters

```
Collections.shuffle(nTrial);
Iterator<Integer> itTrial = nTrial.listIterator();
while (itTrial.hasNext()) {
  int it = itTrial.next();
  // extract the corresponding cornsweet parameters for this trial
  ArrayList<Double> tmp = corn.get(it);
  double contrast = tmp.get(0);
  double alpha = tmp.get(1);
  // extract the corresponding real contrast for this trial
  double contrastReal = real.get(it);
  // Code to present the stimuli with above parameters ....
}
```

Then we get the observer's response and decide what real contrast we should show in the next iteration for that trial number: if observer responded “illusory stimulus had larger difference between flanks” we set cornIsBigger=true and set the real contrast for the next trial higher than the current one. We do the opposite if the observer responds the other way

```
    resp = getKeyTyped(-1);
    if ((resp.equals(resp1) && order == -1)
        || (resp.equals(resp2) && order == 1)) {
      cornIsBigger = true;
      real.put(it, min(MAXCONTRAST, contrastReal + deltaContrastReal));
      break;
    }
    else if ((resp.equals(resp1) && order == 1)
        || (resp.equals(resp2) && order == -1)) {
      cornIsBigger = false;
      real.put(it, max(0.0, contrastReal - deltaContrastReal));
      break;
    }
    else
      continue;
  }
```

In the end of the trial, before reporting the result we compute the actual contrast level of the real stimulus, because it may be different than the desired value

```
            // check the actual displayed contrastReal and report the result
            int tmp2 = (-order + 1) / 2;
            int[] px = new int[stimulus[tmp2].getWidth()];
            stimulus[tmp2].getRaster().getPixels(0, 0,
                stimulus[tmp2].getWidth() - 1, 1, px);
            double contrastRealActual = abs(lut8.pix2Lum(px[0])
                - lut8.pix2Lum(px[px.length - 2]))
                / (lut8.pix2Lum(px[0]) + lut8.pix2Lum(px[px.length - 2])) * 2;
            output.printf("%f %f %f %f %s \n", exponent, contrast, alpha,
                contrastRealActual, cornIsBigger);
            output.flush();
```

Here is the entire code

```
/*
 * chapter 8: Cornsweet2IFC.java
 *
 * An actual experiment demonstrating how to use CLUT8 class
 *
 */
import static java.lang.Math.*;
import java.awt.*;
import java.awt.geom.Ellipse2D;
import java.awt.image.BufferedImage;
import java.awt.image.ColorModel;
import java.io.*;
import java.util.*;
import java.util.List;
public class Cornsweet2IFC extends FullScreen implements Runnable {
  // How many pixels is oneDegree? This will depend on screen resolution,
  // monitor dimensions and viewing distance
  static final int oneDegree  = 51;
  static final int stimWidth  = 16 * oneDegree;
  static final int stimHeight = 10 * oneDegree;
  static final int rampWidth  = 3 * oneDegree;
  static final int frameWidth = 2 * oneDegree;
  // "desired" Luminance (0-255) and Contrast values
  static final double meanLum = 127.0;
  static final double[] contrastCorn = { .6, .6, .9, .9 };
  static final double exponent = 2.75;
  static final double[] alphaCorn = { 0.0, 6.35 };
  static final double[][] initContrastReal = { { .1, .5, .1, .5 },
    { .2, .7, .2, .7 } };
  static final double deltaContrastReal = .04;
  static final double MAXCONTRAST = 2.0;
  // time parameters for the 2IFC
  static final long INTERVAL1 = 1000;
  static final long INTERVAL2 = 1000;
  static final long ISI = 500;
  static final int NTRIALS = 15;
  static final int fixSize = oneDegree / 4;
  static final Color fixationMarkColor = new Color(255,70,70);
  static final Shape fixShape = new Ellipse2D.Double(0,0,fixSize,fixSize);
  BufferedImage fixation;
  int fixX;
  int fixY;

  BufferedImage frame;
  int[] frameX = new int[2];
  int frameY;
  // Response Buttons:
  static final String resp1 = "1";
  static final String resp2 = "2";

  CLUT8 lut8;
  public Cornsweet2IFC(int i) {
    super(i);

    lut8 = new CLUT8("tableGray.txt");
    // Fixation mark
    fixX = (getWidth() - fixSize) / 2;
    fixY = (getHeight() - fixSize) / 2;
    fixation = new BufferedImage(fixSize,fixSize,
        BufferedImage.TRANSLUCENT);
    Graphics2D fsG = (Graphics2D) fixation.getGraphics();
    fsG.setPaint(fixationMarkColor);
    fsG.fill(fixShape);
    fsG.dispose();
    // rectangular wire frames
    frame = new BufferedImage(frameWidth, frameWidth,
        BufferedImage.TRANSLUCENT);
    fsG = (Graphics2D) frame.getGraphics();
    fsG.setPaint(new Color(70, 70, 70));
    fsG.setStroke(new BasicStroke(.5f));
    fsG.drawRect(0, 0, frameWidth - 1, frameWidth - 1);
    fsG.dispose();
    frameX[0] = (getWidth() - stimWidth) / 2 + (int) (3 * oneDegree);
    frameX[1] = getWidth() - frameX[0] - frame.getWidth();
    frameY = (getHeight() - frame.getHeight()) / 2;
  }
  public static void main(String[] args) {
    Cornsweet2IFC cafc = new Cornsweet2IFC(0);
    cafc.setNBuffers(2);
    Thread exp = new Thread(cafc);
    exp.start();
  }
```

```
public void run() {
  try {
    { // compute the actual alpha, and contrastCorn values
      //(as opposed to "desired"):
      BufferedImage tmp;
      for (int i = 0; i < contrastCorn.length; i++) {
        tmp = prepAsymCorn(1, 0, contrastCorn[i]);
        System.err.println("Requested contrastCorn = " + contrastCorn[i]);
        int[] px = new int[tmp.getWidth()];
        tmp.getRaster().getPixels(0, 0, tmp.getWidth() - 1, 1, px);
        contrastCorn[i] = (lut8.pix2Lum(px[tmp.getWidth() / 2 - 1]) - lut8
            .pix2Lum(px[tmp.getWidth() / 2]))
            / (lut8.pix2Lum(px[tmp.getWidth() / 2 - 1]) + lut8.pix2Lum(px[tmp
                .getWidth() / 2])) * 2;
        System.err.println("Received contrastCorn = " + contrastCorn[i]);
      }
      for (int i = 0; i < alphaCorn.length; i++) {
        tmp = prepAsymCorn(1, alphaCorn[i], 0);
        System.err.println("Requested alphaCorn = " + alphaCorn[i]);
        int[] px = new int[tmp.getWidth()];
        tmp.getRaster().getPixels(0, 0, tmp.getWidth() - 1, 1, px);
        alphaCorn[i] = (lut8.pix2Lum(px[1]) - lut8
            .pix2Lum(px[tmp.getWidth() - 2])) / 2;
        System.err.println("Received alphaCorn = " + alphaCorn[i]);
      }
    }
    // trial initialization: corn holds the
    // (KEY=)trial number - (VALUE=)cornsweet parameters (contrast,alpha)
    HashMap<Integer, ArrayList<Double> > corn =
      new HashMap<Integer, ArrayList<Double> >();
    // real holds the (KEY=)trial number - (VALUE=)real contrast
    HashMap<Integer, Double> real = new HashMap<Integer, Double>();
    // nTrial holds the entire trial number in a List
    List<Integer> nTrial = new LinkedList<Integer>();
    int t = 0;
    for (int j = 0; j < contrastCorn.length; j++) {
      for (int k = 0; k < alphaCorn.length; k++) {
        ArrayList<Double> tmp = new ArrayList<Double>();
        tmp.add(contrastCorn[j]);
        tmp.add(alphaCorn[k]);
        corn.put(t, tmp);
        real.put(t, initContrastReal[k][j]);
        nTrial.add(t);
        t++;
      }
    }
    hideCursor();
    blankScreen();
```

```
displayText(10, 50, "Indicate the interval with");
displayText(10, 100, "a larger difference between flanks");
displayText(10, 150, "Press 1 for first, 3 for second");
displayText(10, 250, "Now press anykey to start");
updateScreen();
getKeyPressed(-1);
blankScreen();

PrintWriter output = new PrintWriter(new File("data.txt"));

BufferedImage[] stimulus = new BufferedImage[2];
for (int i = 0; i < NTRIALS; i++) {
  // initialize the trial:
  Collections.shuffle(nTrial);
  Iterator<Integer> itTrial = nTrial.listIterator();
  while (itTrial.hasNext()) {
    int it = itTrial.next();
    // extract the corresponding cornsweet parameters for this trial
    ArrayList<Double> tmp = corn.get(it);
    double contrast = tmp.get(0);
    double alpha = tmp.get(1);
    // extract the corresponding real contrast for this trial
    double contrastReal = real.get(it);
    // order = -1 : first cornsweet; second real
    int order = 1 - 2 * (int) round(random());
    // polarity = 1: Left is +ive ramp,
    // polarity = -1 : Right is +ive
    int polarity = 1 - 2 * (int) round(random());
    stimulus[(-order + 1) / 2] = prepReal(polarity, contrastReal);
    // different polarity for the cornsweet
    polarity = 1 - 2 * (int) round(random());
    stimulus[(order + 1) / 2] = prepAsymCorn(polarity, alpha, contrast);
    // First Interval Stimulus
    displayImage(stimulus[0]);
    displayImage(frameX[0], frameY, frame);
    displayImage(frameX[1], frameY, frame);
    displayImage(fixX,fixY,fixation);
    updateScreen();
    Thread.sleep(INTERVAL1);
    // ISI
    blankScreen();
    updateScreen();
    Thread.sleep(ISI);
    // Second Interval Stimulus
    displayImage(stimulus[1]);
    displayImage(frameX[0], frameY, frame);
    displayImage(frameX[1], frameY, frame);
    displayImage(fixX,fixY,fixation);
```

```
      updateScreen();
      flushKeyTyped();
      Thread.sleep(INTERVAL2);
      blankScreen();
      displayImage(fixX,fixY,fixation);
      updateScreen();
      boolean cornIsBigger = false;
      String resp = getKeyTyped();
      while (true) {
        resp = getKeyTyped(-1);
        if ((resp.equals(resp1) && order == -1)
            || (resp.equals(resp2) && order == 1)) {
          cornIsBigger = true;
          real.put(it, min(MAXCONTRAST, contrastReal + deltaContrastReal));
          break;
        }
        else if ((resp.equals(resp1) && order == 1)
            || (resp.equals(resp2) && order == -1)) {
          cornIsBigger = false;
          real.put(it, max(0.0, contrastReal - deltaContrastReal));
          break;
        }
        else
          continue;
      }
      // check the actual displayed contrastReal and report the result
      int tmp2 = (-order + 1) / 2;
      int[] px = new int[stimulus[tmp2].getWidth()];
      stimulus[tmp2].getRaster().getPixels(0, 0,
          stimulus[tmp2].getWidth() - 1, 1, px);
      double contrastRealActual = abs(lut8.pix2Lum(px[0])
          - lut8.pix2Lum(px[px.length - 2]))
          / (lut8.pix2Lum(px[0]) + lut8.pix2Lum(px[px.length - 2])) * 2;
      output.printf("%f %f %f %f %s \n", exponent, contrast, alpha,
          contrastRealActual, cornIsBigger);
      output.flush();
      // extra sleep between trials - helps observer to refocus
      Thread.sleep(2*ISI);
    }
  }
  output.close();
} catch (InterruptedException e) {
  e.printStackTrace();
} catch (FileNotFoundException e) {
  e.printStackTrace();
}
finally {
  closeScreen();
```

```
    }
  }
  public BufferedImage prepReal(int polarity, double contrast) {
    // Luminances of Uniform flanks
    double meanLeft = meanLum * (1 + contrast / 2 * polarity);
    double meanRight = meanLum * (1 - contrast / 2 * polarity);
    int[] gStim = new int[stimWidth * stimHeight];
    // Convert the Luminance to Pixel using the LUT
    for (int j = 0; j < stimWidth / 2; j++)
      gStim[j] = lut8.lum2Pix(meanLeft);
    for (int j = stimWidth / 2; j < stimWidth; j++)
      gStim[j] = lut8.lum2Pix(meanRight);
    for (int i = 1; i < stimHeight; i++)
      for (int j = 0; j < stimWidth; j++)
        gStim[i * stimWidth + j] = gStim[j];
    // write the result in an image
    BufferedImage stim = new BufferedImage(stimWidth, stimHeight,
        BufferedImage.TYPE_BYTE_GRAY);
    stim.getRaster().setPixels(0, 0, stimWidth, stimHeight, gStim);
    return stim;
  }
  public BufferedImage prepAsymCorn(int polarity, double alpha,
      double contrast) {
    // Luminances of Uniform part of flanks
    double meanLeft = meanLum + alpha * polarity;
    double meanRight = meanLum - alpha * polarity;
    // Convert the Luminance to Pixel using the LUT
    int[] gStim = new int[stimWidth * stimHeight];
    for (int j = 0; j < stimWidth / 2 - rampWidth; j++)
      gStim[j] = lut8.lum2Pix(meanLeft);
    for (int j = stimWidth / 2 - rampWidth; j < stimWidth; j++)
      gStim[j] = lut8.lum2Pix(meanRight);
    // Luminances of Cornsweet2IFC ramps
    double delta = contrast / 2 * meanLum;
    for (int j = 0; j < rampWidth; j++) {
      // Left ramp
      int k = j + stimWidth / 2 - rampWidth;
      double tmp = pow((double) j / (double) (rampWidth - 1), exponent)
          * (-alpha + delta) * polarity;
      gStim[k] = lut8.lum2Pix(meanLeft + tmp);
      // Right ramp
      k = stimWidth / 2 + rampWidth - j - 1;
      gStim[k] = lut8.lum2Pix(meanRight - tmp);
    }
    for (int i = 1; i < stimHeight; i++)
      for (int j = 0; j < stimWidth; j++)
        gStim[i * stimWidth + j] = gStim[j];
    // write the result in an image
    BufferedImage stim = new BufferedImage(stimWidth, stimHeight,
        BufferedImage.TYPE_BYTE_GRAY);
    stim.getRaster().setPixels(0, 0, stimWidth, stimHeight, gStim);
    return stim;
  }
}
```

8.5. Look up operation in Standard Java Libraries

Java Standard library contains classes to perform look up operations. Here I will show how to perform a look up operation using those classes. The example is the same as the one above in which I showed how to filter an image using CLUT8. You first create a ByteLookupTable or a ShortLookupTable by providing an array which tells the relation between displayed pixel values and the image pixel values, i.e. an inverse look up table. Then you construct a LookupOp with this LookupTable. Finally you invoke the filter() method of the LookupOp class to apply the look up operation on your image. Here is the code

```
/*
 * chapter 8: JavaCoreLUTTest.java
 *
 * Demonstrates how to use tools of Standard Java library for look up operation
 */
import java.awt.image.*;
import java.io.*;
import javax.imageio.ImageIO;
public class JavaCoreLUTTest {

  public static void main(String[] args){

    FullScreen fs = new FullScreen();

    try {
      BufferedImage bi = ImageIO.read(new File("fechner.png"));

      fs.displayImage((fs.getWidth()/2-bi.getWidth())/2,
          (fs.getHeight()-bi.getHeight())/2, bi);

      byte[] table = new byte[256];
      for(int i=0; i<256; i++)
        table[i]=(byte)(255-i);

      LookupTable lTable = new ByteLookupTable(0, table);
      BufferedImageOp op = new LookupOp(lTable,null);
      op.filter(bi,bi);

      fs.displayImage((fs.getWidth()+bi.getWidth())/2,
          (fs.getHeight()-bi.getHeight())/2, bi);
      fs.displayText(15,fs.getHeight()-15,"(press any key to finish)");

      fs.updateScreen();
      fs.getKeyPressed(-1);

    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      fs.closeScreen();
    }
  }
}
```

8.6. Summary

If you need a more accurate inverse look up operation, you need to implement it yourself as I showed above. But if you are only concerned with filtering images of standard format you should better use the tools provided by the standard Java library. Whatever route you take, though, you must first prepare a look up table by carefully measuring the luminance output of your monitor.

8.6.1. Using CLUT8

- Construct a CLUT8 object using one of the two constructors
- etc.... coming more soon

8.6.2. Using Standard Java

- Construct a LookupTable object
- etc.... coming more soon