16 Bits++

In this chapter

- What is Bits++?
- Preparing and loading a 14 bits look up table to Bits++
- A fake Bits++ class

Bits++ is a hardware device manufactured by Cambridge Research Systems, which allows to display 2^{14} different luminance values (14 bits) for each color channel as opposed to the 2^8 (8 bits) capacity of common graphics cards. The device is connected to the video output of the computer on one end and to a computer monitor on the other end. To display images of higher resolution you don't need to create new versions of those images. Instead you provide a table which establishes a relation from 8 bits conventional pixels to 2^{14} different "pixel" levels, which I will call *bits++ pixels*, or *b-pixels* for short. I will name this table as *bits++ look up table* or *b-look up table* for short.

16.1 Why do you need 14 bits?

Let's suppose that you want to present a sinusoidal grating whose luminance profile is shown in Figure 16.1a. Normally you create an image with the desired luminance variation

```
for (int j = 0; j < width; j++) {
  //....
  lum[j] = mean + amp * mean * cos(x * cpp * 2 * Math.PI);
  // ...
}
```

and then apply an inverse look up operation as described in Chapter 8 to determine the correct pixel values in order to display the desired luminance pattern on the screen. In the above pseudo code amp is amplitude, mean is mean luminance level, cpp is frequency in units of cycles per pixel size. The resulting pattern could look like the one shown in Figure 16.1b.

Now suppose that you want to be able to vary the amplitude of your grating in very fine steps, because you are interested in contrast discrimination thresholds. For this example let's look at Figure 16.2a. The two curves in the plot are very close to each other, one has amplitude of 0.04, the other 0.045. And in terms of luminance values even their peaks differ less than a unit (in units of arbitray luminance levels ranging from 0 to 255.) As we have seen in Chapter 8, it is not always possible to display the exact luminance value you desire. And definetely the small difference shown in 16.2a will get lost in translation to pixel values and you will end up displaying the exact same stimulus for the two theoretically different luminance patterns. But check the second plot where the luminance values are seperated by finer intervals, it is possible to capture the difference between the peaks of two curves now. Obviously finer steps are more useful!

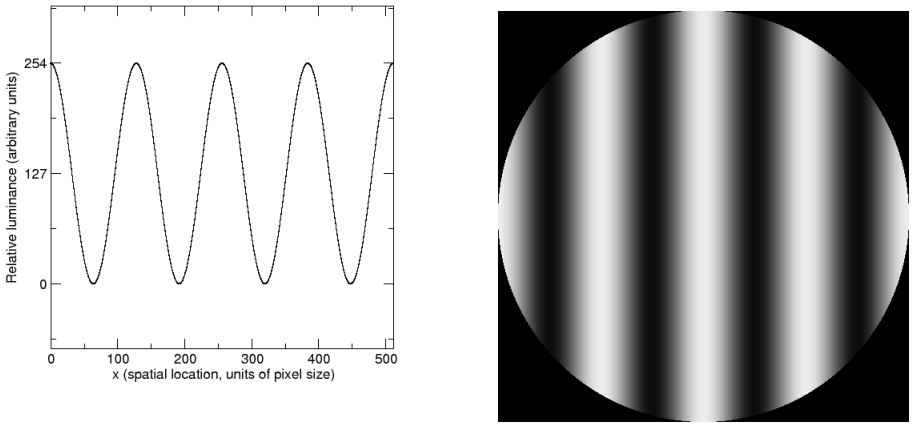

Figure 16.1: The luminance profile (left) along a horizontal line at the middle of a sample grating (right).

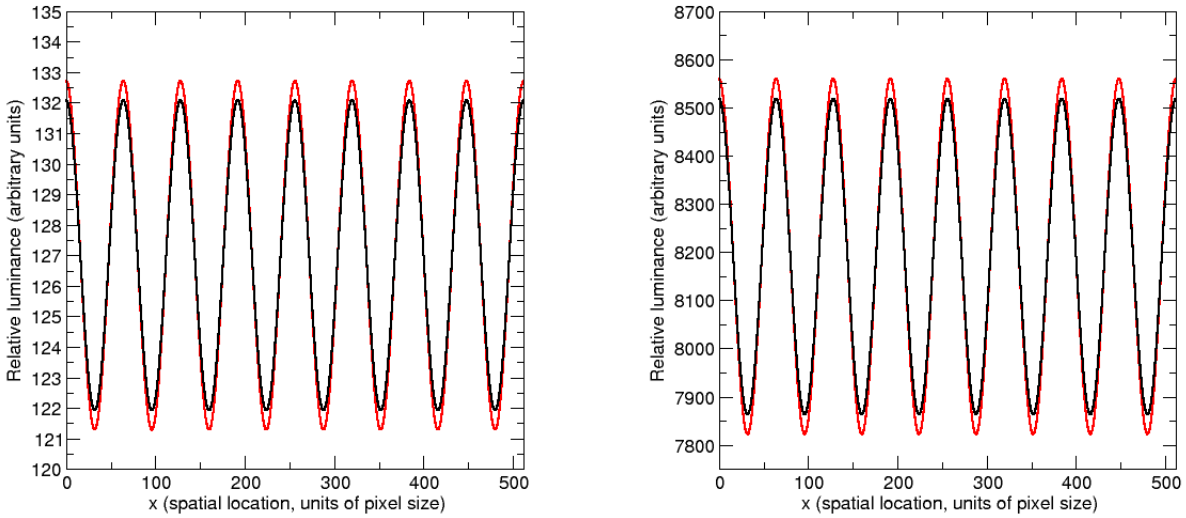

Figure 16.2: A finer amplitude difference. Black curve amplitude=0.04, red curve amplitude=0.045.

But you got a problem: you have a conventional image with pixel values ranging from 0 to 255. How can you get the 14 bit range from that 8 bit image? The answer lies in the bits++ look up table. You don't change the image but manipulate the b-look up table to get the finer scaled luminance pattern on the screen. Let's go back to the grating example. How can you present two slightly different gratings back to back (in time) on the screen? I will explain a method to do this: You first create a pixel value map of your grating, like in the pseudo code above using the entire 8 bit range (0-255). You store this map in an image. Suppose the equation governing the pixel values is

$$pix(x) = 128 + 127 * \cos(2\pi x/128),$$

now you have something like the one shown in Figure 16.3a below. In general one can write the pixel equation as follows

$$pix(x) = meanPix + pixAmplitude/2 * \cos(2\pi x * cpp).$$

This pixel map has a mean value of 128, and amplitude of 254, and ranges from 1 to 255 (0 is spared, which turns out to be useful as we'll see later). Now suppose that the desired relative luminance pattern, in arbitrary units ranging from 0 to $2^{14} - 1$ is

$$lum(x) = 8192 + 0.04 * 8192 * \cos(2\pi x/128).$$

Note that $8192 = 2^{14}/2 = 2^{13}$. The corresponding pattern is shown in Figure 16.3b below in black. The red curve in the same plot corresponds to the same equation above except with a slightly higher contrast (0.045). In general you can write the luminance pattern of a grating as follows

$$lum(x) = meanLum + contrast * meanLum * \cos(2\pi x * cpp)$$

where contrast is defined as

$$contrast = \frac{maxLum - minLum}{2 * meanLum} = \frac{lumAmplitude}{2 * meanLum}.$$

As you see, the problem is reduced to finding a suitable mapping from the 8 bit pixel map to 14 bit relative luminance values. By inspection one finds that the suitable mapping is

$$b{-}table(pix) = meanLum + slope * (pix - meanPix),$$

where slope is given by

$$slope = \frac{lumAmplitude}{pixAmplitude} = \frac{2 * contrast * meanLum}{pixAmplitude}.$$

For our example we get

$$b{-}table(pix) = 2^{13} + \frac{2^{15} * 10^{-2}}{2^7 - 1} * (pix - 2^7).$$

The validity of the above relation can be easily checked: at $x = 0$ we have $pix(0) = 255$ and we should have $lum(0) = 2^{13} + 2^{15} * 10^{-2}$. We find that $b{-}table(255) = 2^{13} + \frac{2^{15} * 10^{-2}}{2^7 - 1} * (255 - 2^7) = 2^{13} + 2^{15} * 10^{-2}$, which checks. Similarly at $x = 32$ we have $pix(32) = 128$ while the desire luminance is $lum(32) = 2^{13}$, and $b{-}table(128) = 2^{13}$, which again checks. You can use the above b-table form for any such grating problem. The mappings corresponding to the two desired luminance profiles are plotted in Figure 16.3c below.

One question which may come to the reader's mind is the following: Are we not loosing precision by storing the base grating profile in integer valued pixels as an image? The answer is yes, but unfortunately there is no other way around that. You will do your best to eliminate artifactual mappings by designing your b-table and base image carefully. The whole process is not 100% automatic.

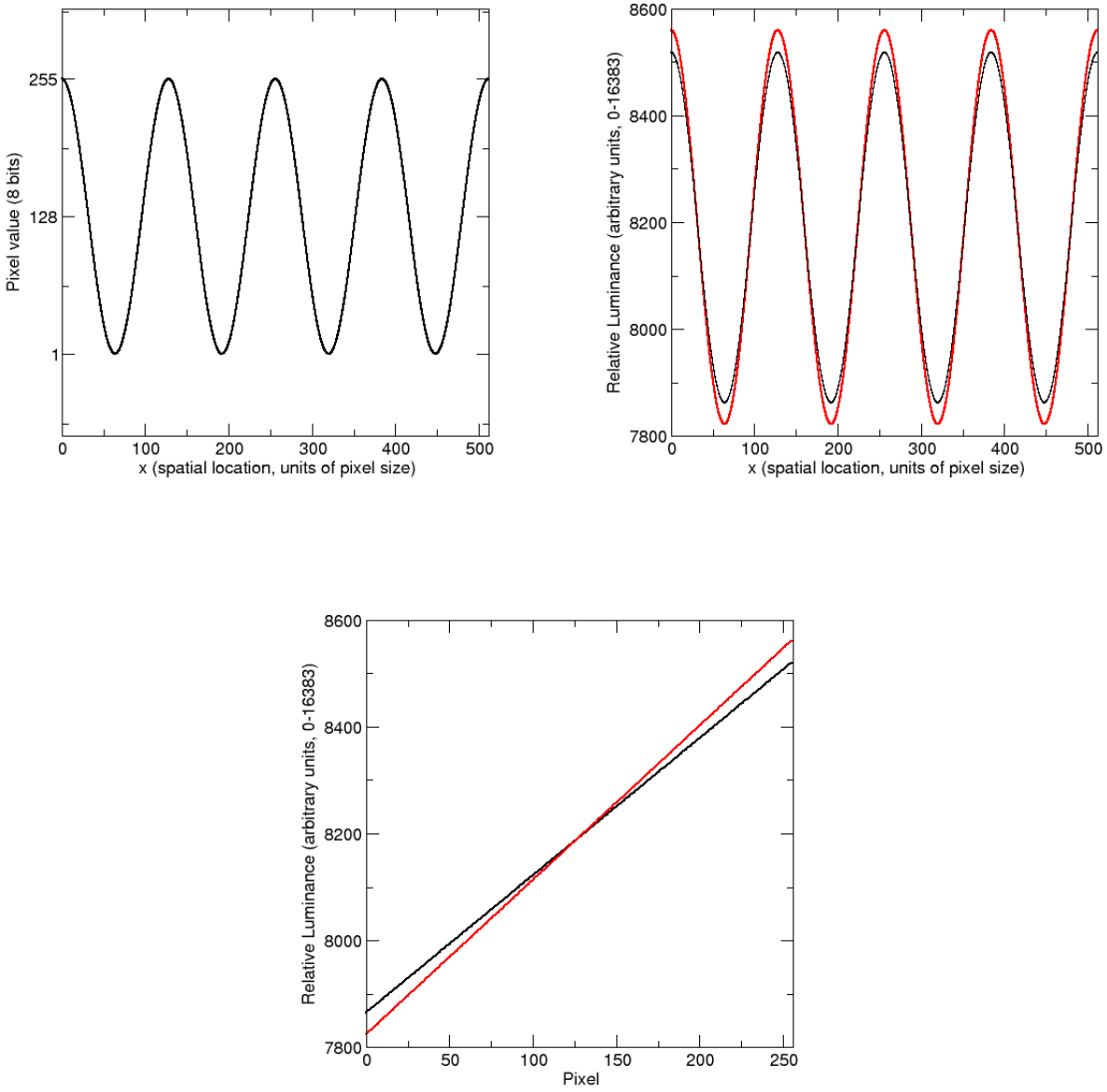

Figure 16.3: A) Grating base pixel profile, B) desired (slightly different) luminance patterns, C) the two slightly different b-tables which provide the mapping from (A) to corresponding (color coded) luminance pattern in (B).

From b-pixel to luminance and vice versa: 14 Bits look up table

We now have a finer scale, a 14 bits luminance map at hand, but what are the right b++ pixel values which give those luminance values? Just like we did in Chapter 8, we need an inverse look up operation, but this time in 14 bits. For this purpose I will use the same class from Chapter 8, only overload the constructor to allow the user specify number of bits. Not to break your former code, I keep the constructors in the original class and default to 8 bits, in other words you can use this class with your implementations which were using the older version. Here are the additions

```
//...
public class CLUT {

  private int DIM;
  //

  public CLUT(String filename){

    this(filename,8);
  }

  public CLUT(double[] table){

    this(table,8);
  }

  public CLUT(String filename, int bits) throws IllegalArgumentException{

    InputStream stream = CLUT.class.getResourceAsStream(filename);
    Scanner in = new Scanner(stream);

    DIM = (int)pow(2,bits);
    pix2Lum = new double[DIM];
    lum2Pix = new int[DIM];

    //...
  }
  public CLUT(double[] table, int bits) throws IllegalArgumentException{
    DIM = (int)pow(2,bits);
    pix2Lum = new double[DIM];
    lum2Pix = new int[DIM];
    //...
  }
  //...
}
```

Just as in Chapter 8 you can employ an inverse look up operation to find the b-pixel that provides the luminance closest to the luminance of your desire

```
CLUT lut = new CLUT("table14.txt");
```

```
for(int i=0; i<btable.length; i++)
  btable[i]=lut.lum2Pix(meanLum+slope*(i-meanPix));
```

A sample b-pixel versus luminance is plotted in Figure 16.4 below.

16.2 Communicating the bits++ look up table

You can dynamically load a b-table to Bits++ with every refresh of the screen. The way to load the table is to "draw" it on a single line and "display" it. It is sensible to put this line at the upper left corner of the screen. This way the lookup operation applies to the rest of the vertical scan. For Bits++ to understand that you are trying to upload a table, it must first be triggered by a specific combination of pixel values. This specific trigger consists of 36 values, or 12 pixels to draw. On the same line, right after those 12 pixels in the next 512 pixels, you provide the values of your new lookup table. Those 512 pixels contain 16 bits of information for each channel and for each of 256 conventional pixel values (note that Bits++ manufacturers require 16 bit mapping rather than 14 bits, this is done for future compatibility). First pixel holds the most significant 8 bit information for each channel for your conventional pixel of 0, second pixel holds the least significant 8 bits for each channel. Hence you have 2 pixels devoted to all three channels for each one of 256 levels (2*256=512). Figure 16.5 show the trigger and images which correspond to several b-look tables. Once it is triggered, Bits++ doesn't send that line to screen for display.

Note: Make sure that your video card has a linear ramp gamma. Because if your video card has anything but linear gamma it will destroy the values in trigger pixels and you will never be able to load your look up table.

16.3 BitsPP class

In this section I will develop a sample class, BitsPP, with methods to control Bits++. It is a good idea to inherit all the useful methods of FullScreen class which do not need any modification, therefore BPP extends FullScreen

```
public class BitsPP extends FullScreen {
  //...
  public BitsPP(int screen_id) {
    super(screen_id);
    initClut();
  }
  //...
}
```

The constructors of BitsPP, apart from invoking FullScreen's constructor, invokes initClut() method which initialiazes the look up table line as a BufferedImage and places the trigger part in first 12 pixels

```
  private BufferedImage lutBI;
  private WritableRaster lutWR;
  public void initClut() {
    int[] unlock = { 36, 106, 133, 63, 136, 163, 8, 19,
        138, 211, 25, 46, 3, 115, 164, 112, 68, 9, 56, 41,
        49, 34, 159, 208, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
```

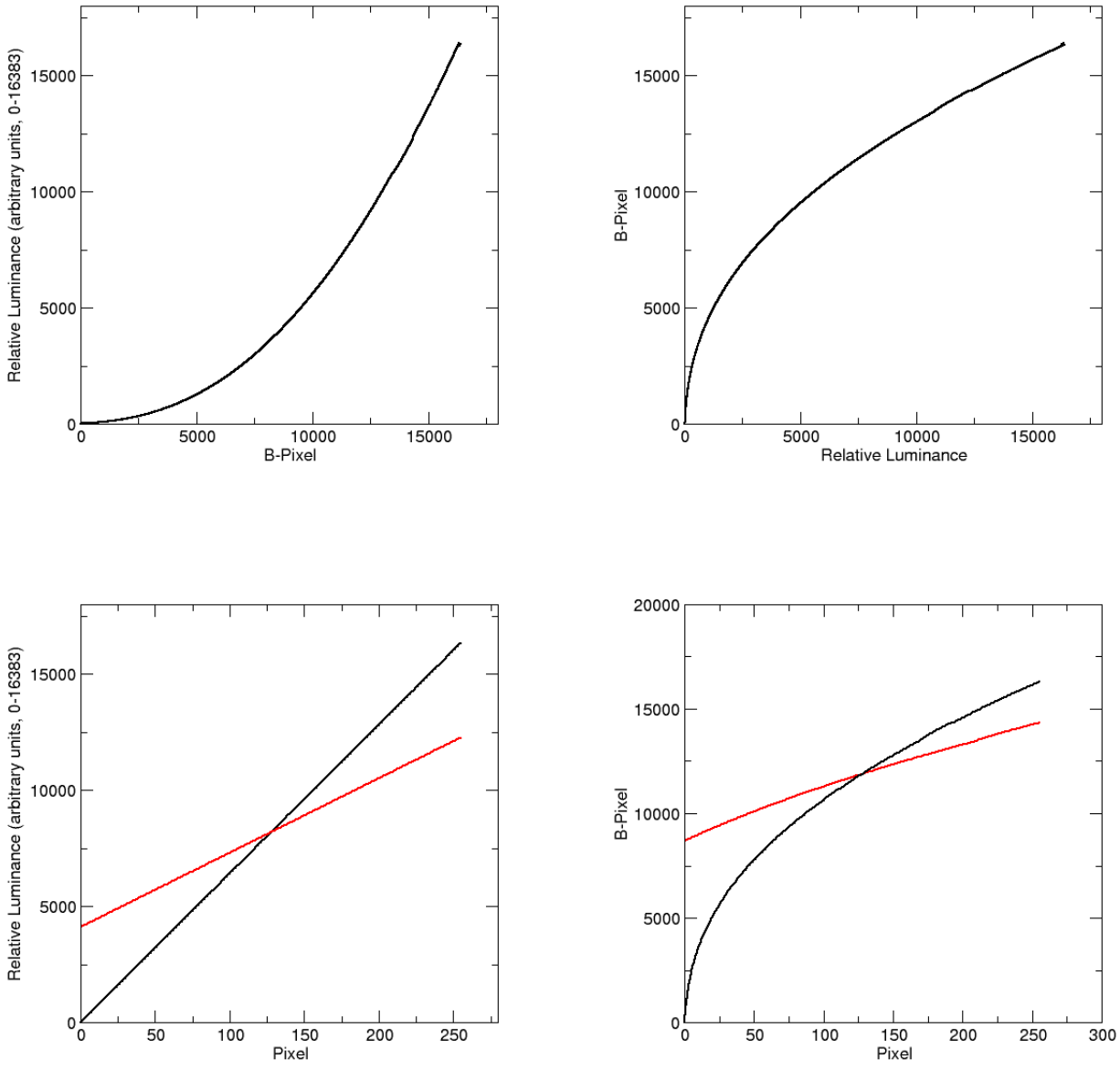

Figure 16.4: A) B-pixel vs. relative luminance, B) Inverse of (A), C) Desired relations between conventional pixels and the finer scale relative luminance, D) Bits++ look up table, b-table: relation between the conventional pixel and b-pixel to achive the desired pixel vs. luminance relation in (C) given the relation between b-pixels and luminance in (A) and (B).

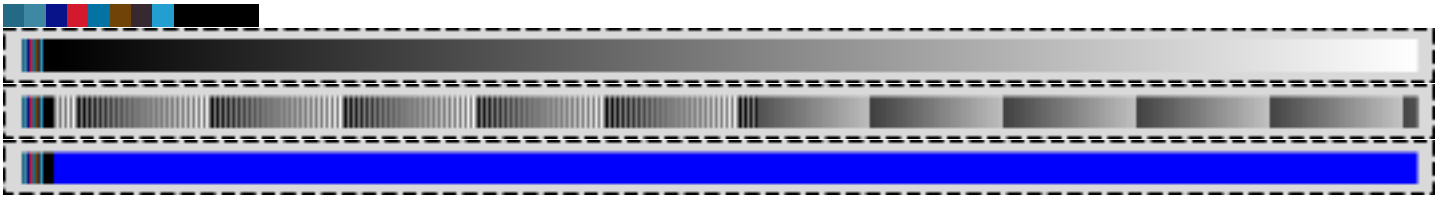

Figure 16.5: The look-up line uploaded to Bits++ device. On top, the trigger header is shown. Below, the slope=1 table for all three channels. The next image corresponds to the b-table given in the example equation in text (Eqn XXX). Bottom image is a chromatic version where red and green channels are all set to zero, blue channel set to maximum.

```
  int width = getWidth();
  lutBI = new BufferedImage(width, 1, BufferedImage.TYPE_INT_RGB);
  lutWR = (WritableRaster) lutBI.getRaster();
  try {
    lutWR.setPixels(0, 0, unlock.length / 3, 1, unlock);
  } catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("Error in BitsPP.initClut(): cannot initialize " +
        "clut row (is your screen wide enough (>524 pixels)?)");
    e.printStackTrace();
  }
  int[] table = new int[256];
  for (int i = 0; i < table.length; i++) {
    table[i] = i << 6;
  }
  setClut(table);
}
```

initClut() initializes the lutBI, a BufferedImage with 1 pixel height, and generates its WritableRaster lutWR. lutBI is going to be the bits++ look up table line. initCLut() first places the trigger sequence at the first 12 pixels of lutBI, then invokes setCLut() method with a dummy look up table, which is just a straight line. setCLut in turn prepares the rest of the lutBI

```
public void setClut(int clut[]) throws ArrayIndexOutOfBoundsException{
  int len = 256;
  if (clut.length != len)
    throw new ArrayIndexOutOfBoundsException(
      "Clut should have " + len + " elements");

  int[] row = new int[len * 2 * 3];
  int r;
  for (int i = 0; i < len; i++) {
    r = (clut[i]) << 2;
    row[i * 6] = r >> 8;
    row[i * 6 + 1] = row[i * 6];
    row[i * 6 + 2] = row[i * 6];
    row[i * 6 + 3] = r & 255;
    row[i * 6 + 4] = row[i * 6 + 3];
```

```
      row[i * 6 + 5] = row[i * 6 + 3];
    }
    lutWR.setPixels(12, 0, row.length / 3, 1, row);
  }
```

In the final version of the class I will include an overloaded version of this method to accept 3 lookup tables for each of R,G and B channels.

Once you have the b-table as an image all you have to do is to "display" it (remember Bits++ doesn't really display that line on the screen, as soon as it sees the trigger, the rest of the line is interpreted as b-table and is not sent to screen). It is sensible to put this line at the upper left corner of the screen. This way the lookup operation applies to the rest of the vertical scan. I will override the displayImage() method of FullScreen to automatically draw the b-table on the top line everytime it is invoked

```
public void displayImage(int x, int y, BufferedImage bi) {
    Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
    try {
      if(g!=null && bi!=null){
        g.drawImage(bi, x, y, null);
        g.drawImage(lutBI, 0, 0, null);
      }
    }
    finally {
      g.dispose();
    }
  }
```

The changes to the other two methods, displayText() and blankScreen(), follow the same logic. In order to leave the system in a nice condition (loaded with a linear b-table) after the program is terminated I will override the closeScreen() method

```
public void closeScreen() {
  initClut();
  blankScreen();
  updateScreen();
  try{
    Thread.sleep(100);
  }catch(InterruptedException e){}
  super.closeScreen();
}
```

That's all, and here is the complete listing of BitsPP.java followed by the new CLUT class

```
/*
 * Chapter A: BitsPP
 *
 * Provides Bits++ triggering mechanism and methods to load
 * Bits++ look up tables.
 *
 */
```

```
import java.awt.*;
import java.awt.image.*;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
public class BitsPP extends FullScreen {
  private BufferedImage lutBI;
  private WritableRaster lutWR;
  public BitsPP() {
    this(0);
  }
  public BitsPP(int screen_id) {
    super(screen_id);
    initClut();
  }
  public void initClut() {
    int[] unlock = { 36, 106, 133, 63, 136, 163, 8, 19,
        138, 211, 25, 46, 3, 115, 164, 112, 68, 9, 56, 41,
        49, 34, 159, 208, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    int width = getWidth();
    lutBI = new BufferedImage(width, 1, BufferedImage.TYPE_INT_RGB);
    lutWR = (WritableRaster) lutBI.getRaster();
    try {
      lutWR.setPixels(0, 0, unlock.length / 3, 1, unlock);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.err.println("Error in BitsPP.initClut(): cannot initialize " +
          "clut row (is your screen wide enough (>524 pixels)?)");
      e.printStackTrace();
    }
    int[] table = new int[256];
    for (int i = 0; i < table.length; i++) {
      table[i] = i << 6;
    }
    setClut(table);
  }
  public void setClut(int[] redClut, int[] greenClut, int[] blueClut)
      throws ArrayIndexOutOfBoundsException{
    int len = 256;
    if (redClut.length != len || greenClut.length != len
        || blueClut.length != len)
      throw new ArrayIndexOutOfBoundsException(
        "Clut should have " + len + " elements");

    int[] row = new int[len * 2 * 3];
    int r;
    int g;
    int b;
    for (int i = 0; i < len; i++) {
```

```
    r = redClut[i]   << 2;
    g = greenClut[i] << 2;
    b = blueClut[i]  << 2;
    row[i * 6] = r >> 8;
    row[i * 6 + 1] = g >> 8;
    row[i * 6 + 2] = b >> 8;
    row[i * 6 + 3] = r & 255;
    row[i * 6 + 4] = g & 255;
    row[i * 6 + 5] = b & 255;
  }
  lutWR.setPixels(12, 0, row.length / 3, 1, row);
}
public void setClut(int clut[]) throws ArrayIndexOutOfBoundsException{
  int len = 256;
  if (clut.length != len)
    throw new ArrayIndexOutOfBoundsException(
      "Clut should have " + len + " elements");

  int[] row = new int[len * 2 * 3];
  int r;
  for (int i = 0; i < len; i++) {
    r = (clut[i]) << 2;
    row[i * 6] = r >> 8;
    row[i * 6 + 1] = row[i * 6];
    row[i * 6 + 2] = row[i * 6];
    row[i * 6 + 3] = r & 255;
    row[i * 6 + 4] = row[i * 6 + 3];
    row[i * 6 + 5] = row[i * 6 + 3];
  }
  lutWR.setPixels(12, 0, row.length / 3, 1, row);
}
public void displayImage(int x, int y, BufferedImage bi) {
  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && bi!=null){
      g.drawImage(bi, x, y, null);
      g.drawImage(lutBI, 0, 0, null);
    }
  }
  finally {
    g.dispose();
  }
}
public void displayText(int x, int y, String text) {
  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && text!=null){
      g.setFont(getFont());
```

```
        g.setColor(getForeground());
        g.drawString(text, x, y);
        g.drawImage(lutBI, 0, 0, this);
      }
    }
    finally {
      g.dispose();
    }
  }

  public void blankScreen() {
    Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
    try {
      if(g!=null){
        g.setColor(getBackground());
        g.fillRect(0, 0, getWidth(), getHeight());
        g.drawImage(lutBI, 0, 0, this);
      }
    }
    finally {
      g.dispose();
    }
  }
  public void closeScreen() {
    initClut();
    blankScreen();
    updateScreen();
    try{
      Thread.sleep(100);
    }catch(InterruptedException e){}
    super.closeScreen();
  }
}
```

and here is the new CLUT

```
/*
 * Chapter A: CLUT8.java
 *
 * Provides methods to perform (inverse) lookup 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 CLUT {
  private int DIM;
  private double maxLum;
  private double[] pix2Lum;
  private int[] lum2Pix;
  private boolean monotonicIncrease = true;
  private boolean monotonicDecrease = true;
  public CLUT(String filename){

    this(filename,8);
  }

  public CLUT(double[] table){

    this(table,8);
  }

  public CLUT(String filename, int bits) throws IllegalArgumentException{
    InputStream stream = CLUT.class.getResourceAsStream(filename);
    Scanner in = new Scanner(stream);

    DIM = (int)pow(2,bits);
    pix2Lum = new double[DIM];
    lum2Pix = new int[DIM];

    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 CLUT(double[] table, int bits) throws IllegalArgumentException{
    DIM = (int)pow(2,bits);
    pix2Lum = new double[DIM];
    lum2Pix = new int[DIM];
```

```
  if (table.length != DIM)
    throw new IllegalArgumentException(
        "Error in CLUT(double[], int): 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] *= (double)(DIM-1) / 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) {
          System.err.println(lum);
          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[min(intLum + 1, DIM-1)];
      int pixelEnd = lum2Pix[max(intLum - 1, 0)];
      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];
  }
}
```

16.4 A Fake BitsPP class

One often needs to work on the experimental code on a different computer rather than the one connected to Bits++. Of course it wouldn't be very productive to have two separate code for development and actual experiment. However, we can easily create a fake Bits++ class which has the exact same structure except it performs software lookup operation within the limits of a standard 8 bit video card. Here are the initClut() and setClut() methods of the fake Bits++ class

```
private BufferedImageOp op;
private LookupTable table;

public void initClut() {
  int[] convert = new int[256];
  for (int i = 0; i < 256; i++)
    convert[i] = i << 6;
  setClut(convert);
}

public void setClut(int clut[]) throws ArrayIndexOutOfBoundsException{
    int len = 256;
    if (clut.length != len)
      throw new ArrayIndexOutOfBoundsException(
          "Clut should have " + len + " elements");
    byte[] row = new byte[len];
    for (int i = 0; i < len; i++)
      row[i] = (byte) (clut[i] >> 6);
```

```
        table = new ByteLookupTable(0, row);
        op = new LookupOp(table, new RenderingHints(
            RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON));
    }
```

Since standard video cards have 8 bit resolution, in the fake mode we use only the most significant part of the table provided by the application. Note that you can omit the RenderingHints and pass null instead. This would improve the speed of the filtering but result in poorer quality image. As in the actual BitsPP class, I overload the setClut() method to accept 3 tables, one for each of R, G and B channels. In the displayImage() method we use the BufferedImageOP.filter() method to perform the software lookup operation

```
    public void displayImage(int x, int y, BufferedImage bi) {

        Graphics2D g = (Graphics2D) getBufferStrategy().getDrawGraphics();
        try {
          // check image type ...
          g.drawImage(op.filter(bi, null), (int) x, (int) y, null);
          getBufferStrategy().show();
        }
        finally {
          g.dispose();
        }
      }
```

However a software lookup operation is much slower than a hardware one, especially for larger images. Another problem is that the BufferedImageOP and the image we provide may not be compatible. If they are incompatible, the result of the operation would be unpredictable. To address these issues, I implement the displayImage() method to accept only two certain types of BufferedImages and to check the compatibility of the BufferedImage and the BufferedImageOP. The images accepted are of type TYPE_BYTE_GRAY or TYPE_3BYTE_BGR. In the examples below I will show how to create images of these types. As for the lookup tables, if the image to display is of TYPE_BYTE_GRAY, then the lookup table should be prepared with setClut(clut) method, if it is of TYPE_3BYTE_BGR, the lookup table should be prepared with setClut(redClut, greenClut, blueClut). In this sample fake version I will not override displayText(), blankScreen() and closeScreen() methods, you can easily improve those methods to fit your needs. The final listing of BitsPPFake.java follows

```
/*
 * Chapter A: BitsPPFake.java
 *
 * Provides Fake Bits++ methods for development
 *
 */
import java.awt.*;
import java.awt.image.*;
public class BitsPPFake extends FullScreen {
  private BufferedImageOp op;
  private LookupTable table;
  public BitsPPFake() {
```

```
  this(0);
}
public BitsPPFake(int screen_id) {
  super(screen_id);
  initClut();
}
public void initClut() {
  int[] convert = new int[256];
  for (int i = 0; i < 256; i++)
    convert[i] = i << 6;
  setClut(convert);
}
public void setClut(int[] redClut, int[] greenClut, int[] blueClut)
throws ArrayIndexOutOfBoundsException{
  int len = 256;
  if (redClut.length != len || greenClut.length != len
      || blueClut.length != len)
    throw new ArrayIndexOutOfBoundsException(
        "Clut should have " + len + " elements");
  byte[][] row = new byte[3][len];
  for (int i = 0; i < len; i++) {
    row[0][i] = (byte) (blueClut[i] >> 6);
    row[1][i] = (byte) (greenClut[i] >> 6);
    row[2][i] = (byte) (redClut[i] >> 6);
  }
  table = new ByteLookupTable(0, row);
  op = new LookupOp(table, new RenderingHints(
      RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON));
}
public void setClut(int clut[]) throws ArrayIndexOutOfBoundsException{
  int len = 256;
  if (clut.length != len)
    throw new ArrayIndexOutOfBoundsException(
        "Clut should have " + len + " elements");
  byte[] row = new byte[len];
  for (int i = 0; i < len; i++)
    row[i] = (byte) (clut[i] >> 6);
  table = new ByteLookupTable(0, row);
  op = new LookupOp(table, new RenderingHints(
      RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON));
}
public void displayImage(int x, int y, BufferedImage bi) {
  Graphics g =  getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && bi!=null){
      if (bi.getRaster().getNumBands() != table.getNumComponents()
```

```
            || (bi.getType() != BufferedImage.TYPE_3BYTE_BGR
                && bi.getType() != BufferedImage.TYPE_BYTE_GRAY)) {
          System.err.println(
          "Exception in BitsPPFake.displayImage(): Wrong image or table type");
          System.err.println(
          "use either a BufferedImage of TYPE_BYTE_GRAY with setClut(table)");
          System.err.println(
          "        or a TYPE_3BYTE_BGR with setClut(tableR, tableG, tableB)");
          closeScreen();
          System.exit(0);
        }
        else
          g.drawImage(op.filter(bi, null), x, y, null);
      }
    }
    finally {
      g.dispose();
    }
  }
}
```

16.5 Examples

I will now show how to dynamically load lookup tables to Bits++ and animate otherwise stationary images. In the first example we generate a sinusoidal grating and make it counter-phase flicker by inverting the lookup table. In the second example, we generate a sinusoidal grating and allow the observer change its contrast by pressing up and down arrow keys. Here is the listing of the first example

```
/*
 * chapter A: BitsPlusFlicker.java
 *
 * displays a flickering grating using bits++
 *
 */
import java.awt.image.BufferedImage;
import static java.lang.Math.*;
//Choose either fake or actual mode
//public class BitsPlusFlicker extends BitsPP implements Runnable {
public class BitsPlusFlicker extends BitsPPFake implements Runnable {

  private static final long flickDuration = 64;
  private static final int repeat = 100;

  BitsPlusFlicker(){

    super(0);
  }
```

```
BitsPlusFlicker(int i){

  super(i);
}
public static void main(String[] args) {
  int screen_id = 0;
  BitsPlusFlicker fs = new BitsPlusFlicker(screen_id);
  fs.setNBuffers(2);
  new Thread(fs).start();
}

public void run(){

  try {

    int dd = (int) pow(2, 14) - 1;
    float onePix = dd/255f;
    int[][] table = new int[2][256];
    for (int i = 1; i < table[0].length; i++){
      table[0][i] = (int)(i * onePix);
      table[1][i] = dd - table[0][i];
    }

    BufferedImage bi =
      Tools.aGrating(128, (double)1/128, (double)127/128, 513);
    blankScreen();
    displayImage(bi);
    updateScreen();

    Thread.sleep(200);

    long total = System.currentTimeMillis();

    for (int k = 0; k < repeat / table.length; k++) {

      for (int j=0; j < table.length; j++){
        long start = System.currentTimeMillis();
        setClut(table[j]);
        displayImage(bi);
        updateScreen();
        Thread.sleep(
            max(flickDuration - System.currentTimeMillis() + start, 0));
      }
    }

    System.err.println("rendered and showed " + repeat +
        " images in: " + (System.currentTimeMillis()-total) + " msec");
```

```
    } catch (PixelOutOfRangeException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      e.printStackTrace();
      Thread.currentThread().interrupt();
    }
    finally {
      closeScreen();
    }
  }
}
```

to choose either the fake or actual mode, just uncomment the mode you like to use and comment out the other one. This example works in both modes without any further change. You should also choose a screen device. If you have only one screen, leave screen_id = 0. If you have more than one screen, find out the order by playing with screen_id variable. Next, the program prepares two linear lookup tables with inverse slopes, and prepares the grating using Tools.aGrating() method. This image stays constant for the entire duration of the test. But in the loop below, we use the two tables prepared before. After 100 iteration, the program reports the total time spent on setting the lookup table and displaying the image.

I've placed the aGrating() method in a different class because we will need it once agaoin for the next example below, here is the Tools class

```
/*
 * Chapter A: Tools.java
 *
 * Provides the method to prepare sinusoidal grating
 *
 */
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
public class Tools {
  public static BufferedImage aGrating(int mean, double cpp,
      double contrast, int size) throws PixelOutOfRangeException {
    int[] gPixel = new int[size * size];
    int size2 = size * size / 4;
    for (int j = 0; j < size; j++) {
      int x = (j - size / 2);
      int val = (int) (contrast * mean * Math.cos(x * cpp * 2 * Math.PI));
      x = x * x;
      int iLow = (int) (size / 2 - Math.sqrt(size2 - x));
      int iHigh = (int) (size / 2 + Math.sqrt(size2 - x));
      for (int i = iLow; i <= iHigh; i++) {
        int k = i * size + j;
        if ((gPixel[k] = mean + val) > 255)
          throw new PixelOutOfRangeException(
            "Exception in setGrating: calculated pixel value exceeds 255");
      }
```

```
    }
    BufferedImage bi = new BufferedImage(size, size,
        BufferedImage.TYPE_BYTE_GRAY);
    bi.getRaster().setPixels(0, 0, size, size, gPixel);
    return bi;
  }
  public static BufferedImage aGratingBGR(int mean, double cpp,
      double contrast, int size) throws PixelOutOfRangeException {
    BufferedImage bi = aGrating(mean, cpp, contrast, size);
    BufferedImage bic = new BufferedImage(size, size,
        BufferedImage.TYPE_3BYTE_BGR);
    Graphics2D g2 = (Graphics2D) bic.getGraphics();
    g2.drawImage(bi, 0, 0, null);
    return bic;
  }
}
class PixelOutOfRangeException extends Exception {
  PixelOutOfRangeException(String s) {
    super(s);
  }
}
```

This class has methods to create TYPE_BYTE_GRAY or TYPE_3BYTE_BGR BufferedImages. Note how one can convert an image to a different format. Using a TYPE_BYTE_GRAY is advantageous especially in the fake mode, because the otherwise slower software lookup operation is fast enough with a single band. The methods throw PixelOutOfRangeExceptions in case the pixel level exceeds 255.

The next example is interactive: user can change the contrast of the grating by pressing the up and down arrow keys

```
/*
 * chapter A: BitsPlusAdjustContrast.java
 *
 * displays a varying contrast grating using bits++
 *
 */
import java.awt.image.BufferedImage;
import java.awt.event.KeyEvent;
import static java.lang.Math.*;
//choose either fake or actual mode
public class BitsPlusAdjustContrast extends BitsPPFake implements Runnable{
//public class BitsPlusAdjustContrast extends BitsPP implements Runnable{
  public BitsPlusAdjustContrast(){

    super(0);
  }

  public BitsPlusAdjustContrast(int i){
```

```
  super(i);
}

public static void main(String[] args) {
  int screen_id = 0;
  BitsPlusAdjustContrast fs = new BitsPlusAdjustContrast(screen_id);
  fs.setNBuffers(2);
  new Thread(fs).start();
}

public void run(){

  try {

    int dd = (int) Math.pow(2, 14) - 1;
    float onePix = dd / 255f;
    BufferedImage bi =
      Tools.aGrating(128, (double)1/128, (double)127/128, 513);
    int[] table = new int[256];
    for (int i = 0; i < table.length; i++)
      table[i] = (int)(i * onePix);

    setClut(table);

    displayText(50, 100,
        "Press up/down arrow keys to increase/decrease contrast");
    displayText(50, 200, "Press q to quit");
    displayText(50, 300, "Now Press any key to start");
    updateScreen();

    getKeyPressed(-1);

    blankScreen();
    displayImage(bi);
    updateScreen();
    Integer res = null;

    int meanPix = 128;
    int meanLum = (int)pow(2,13);
    float maxSlope = (float)(meanLum/meanPix);
    float slope = maxSlope;
    while (true) {

      res = getKeyPressed(-1);

      if (res == KeyEvent.VK_UP)
        slope = Math.min(maxSlope, slope + .05f * maxSlope);
      else if (res == KeyEvent.VK_DOWN)
```

```
          slope = Math.max(-maxSlope, slope - .05f * maxSlope);
        else if (res == KeyEvent.VK_Q)
          break;

        // leave 0 always black
        for (int i = 1; i < table.length; i++)
          table[i] = (int)(meanLum + slope * (i - meanPix));

        setClut(table);
        displayImage(bi);
        updateScreen();
      }
      blankScreen();
    } catch (PixelOutOfRangeException e) {}
    finally {
      closeScreen();
    }
  }
}
```

The logic of this example is similar to the previous one. We create a stationary grating, then manipulate the contrast by altering the lookup table. Here we use a different scheme: we have a linear table, whose slope can vary between 0 and 1. When the user presses up arrow the slope increases, with down arrow press it decreases. The table is pivoted at the mean value of the grating, so that the mean is always fixed.