COLORS IN LPL'S DRAWING TOOL

Tony Hürlimann
Tony.huerlimann@unifr.ch

Working Paper

April 2005

DEPARTMENT D'INFORMATIQUE, UNIVERSITE DE FRIBOURG
DEPARTEMENT FÜR INFORMATIK, UNIVERSITÄT FREIBURG
Department of Informatics, University of Fribourg
Site Regina Mundi, rue de Faucigny 2, CH-1700 Fribourg / Switzerland

Colors in LPL's Drawing Tool

Tony Hürlimann
Tony.huerlimann@unifr.ch

1 Introduction

Computer monitors *emit* color as RGB (red, green, blue) light. Although all colors of the visible spectrum can be produced by merging red, green and blue light, monitors are capable of displaying only a limited gamut (i.e., range) of the visible spectrum.

Whereas monitors emit light, inked paper *absorbs or reflects* specific wavelengths. Cyan, magenta and yellow pigments serve as filters, subtracting varying degrees of red, green and blue from white light to produce a selective gamut of spectral colors. Like monitors, printing inks also produce a color gamut that is only a subset of the visible spectrum, although the range is not the same for both. Consequently, the same art displayed on a computer monitor may not match to that printed in a publication. Also, because printing processes such as offset lithography use CMYK (cyan, magenta, yellow, black) inks, digital art must be created as CMYK color or must be converted from RGB color to enable use.

Colors on a screen are specified in the **RGB** space. RGB goes up to 24-bit (three 8-bit channels for red, green, and blue). RGB Color red, green, and blue are the primary colors of **light.** (Don't get confused with the normal primary colors of red, yellow and blue.)

RGB color is called *additive* because colors throughout the spectrum are created by adding varying intensities of red, green, and blue light to black (which on a computer screen equals no light). These intensities vary from 255 (full intensity) to 0. Each color channel has 256 variations (red, green and blue) and if you combine all of their possible combinations, you get 16,777,216 possible colors. Because the computer screen displays light as you add color in the RGB model, the more colors that are added, the more light is added, thus the lighter your resulting color.

There exists another specification for colors used in printing medias, the **CMYK**. The CMYK specification is a 32-bit model (one 8-bit channel for each of cyan, magenta, yellow and black). (The K in CMYK is for black in order not to confuse it with Blue.) CMYK Color cyan, magenta, and yellow are the secondary colors of RGB and are opposite to them. When RGB light strikes an object, the amount of cyan, magenta and yellow in the object's pigmentation affects how much light is reflected back (it's the reflected light that we see). Cyan absorbs red light, magenta absorbs green light, and yellow absorbs blue light. The degree of absorption depends on the amount of any of the CMYK colors that are present. That's why CMYK is considered subtractive, because the colors displayed by CMYK are the result of subtracting varying amounts of red, green, and blue light.

Figure 1 gives a view of the two color schemes.

In the modeling language LPL, one can specify the colors in the RGB space. That’s what the paper goes to explore now.

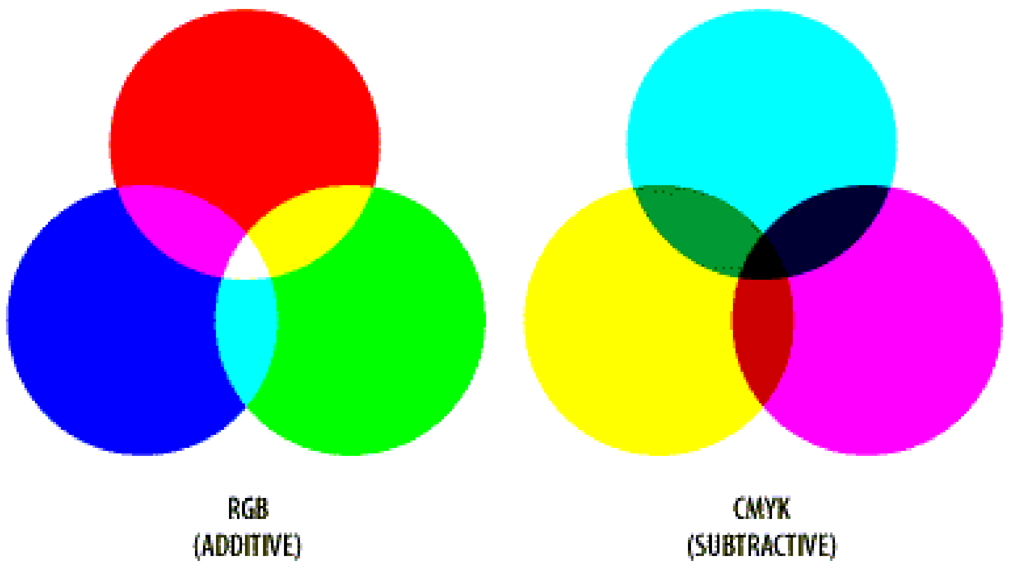


Figure 1: The RGB and CMYK Colors Mixture

2 The RGB Color Space

The Commission Internationale d'Eclairage (CIE) chart defines a standardized color space [Color Vision]. The CIE chart is a useful tool for specifying colors. It represents hue and saturation on a two dimensional chart. Fully saturated hues lie along the outside edge with desaturated colors toward the center of the chart. This CIE X-Y chart was later revised to better reflect human perception of color. While the CIE chart is a very useful tool, the color space it defines does not provide an intuitive model for our color vision system or the devices we use to reproduce color. The following diagram is an approximate representation of the CIE color chart.

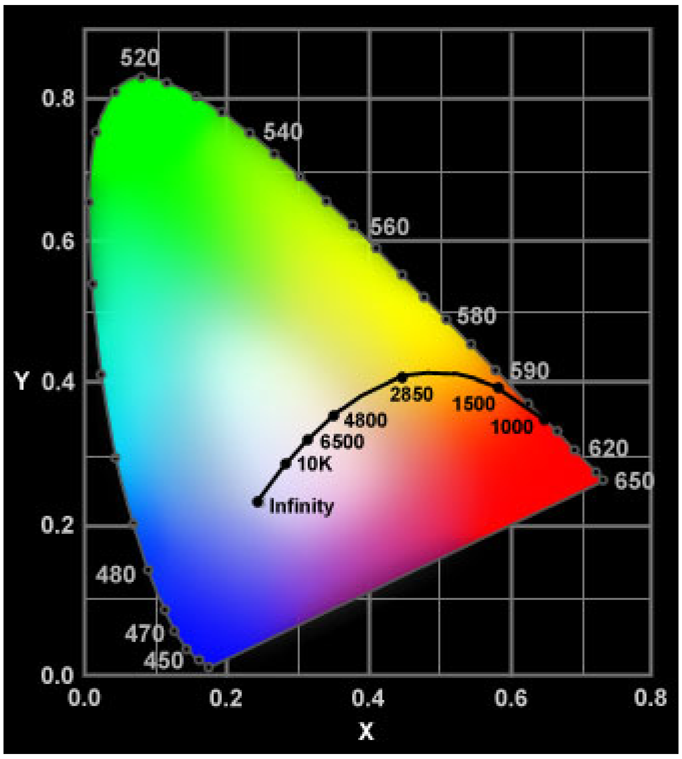


Figure 2: The CIE chart

Our vision system uses three different sensors to selectively detect the visual spectrum, the color space is best visualize as a three dimensional color cube. One corner represents zero excitation for all three sensors or the color we call *black*. There is a sensor vector along each of the three edges which leave this zero excitation corner. These vectors represent the extent of the stimulus for the **Rho**, **Gamma** and **Beta** sensors. This cube has *white* at the corner directly opposite *black*. It has a *primary color* (red, green or blue) in the corner opposite its *complimentary color* (cyan, magenta or yellow – the *secondary colors*). Figure 3 shows the visualization of the color space defined by the Rho, Gamma and Beta sensor stimulus vectors.

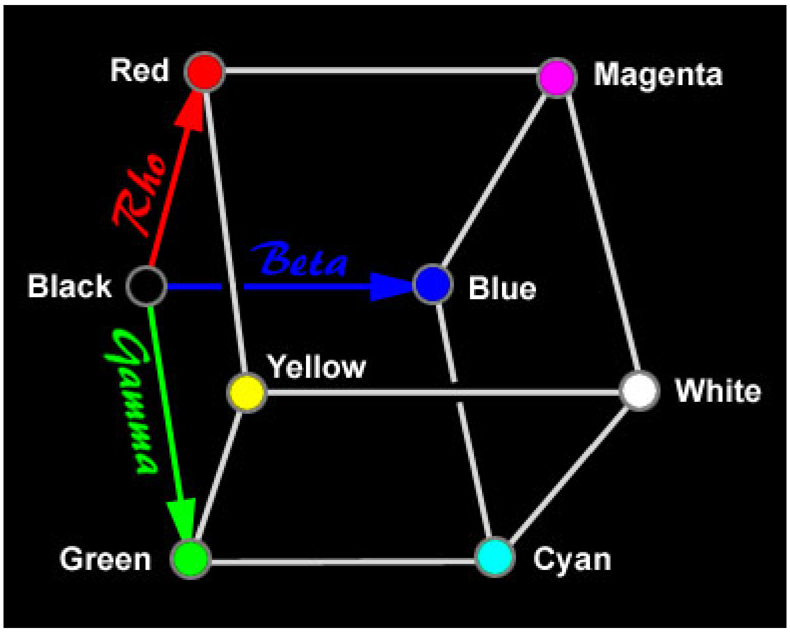
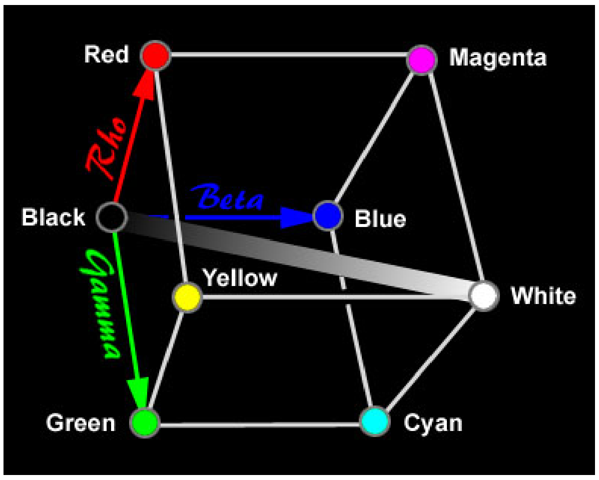


Figure 3: The RGB Color Cube

There is a line connecting the *black* and *white* corners of the cube. This is the line of neutral gradient (Figure 4). There are also lines connecting each of the primary colors (RGB) with their corresponding secondary colors (CMY). These are the lines of primary-secondary gradient. These are the lines along which we make color correction judgments for prints of color images.



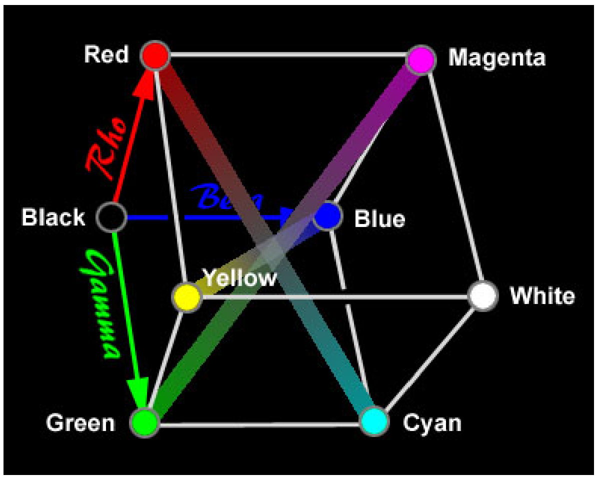


Figure 4: Neutral Gradient (White-Black) **Figure 5: The Primary/Secondary Gradients**

A triangular plane connecting each of the primary colors (RGB) can be build the "plane"-gradient between the three primary colors. There is also a triangular plane connecting each of the secondary colors (CMY). This plane crosses the neutral gradient line at a point closer to white than black and that the plane of the primaries crosses at a point closer to black than white. We generally expect secondary colors to reproduce lighter than primary colors in black and white images. In fact, a plane-gradient can be built between any three grid points. Notice also that all the fully saturated colors live on the surface of the cube.

Each of the primary and secondary colors have their own paths from black to white. The RGB primaries move away from the black corner along three separate paths. Whenever a vector moves along a corner of the cube, it is changing in a single variable – in this case, the RGB primary itself. Once the RGB primaries reach their fully saturated corn of the cube, new vectors move diagonally across a cube side, toward the white corner. When a vector moves diagonally across a cube side, it is changing in two variables. To move from any one of the fully saturated primaries toward white, an equal amount of the other two primaries are added. For example to move from the red corner to the white corner, green and blue are added.

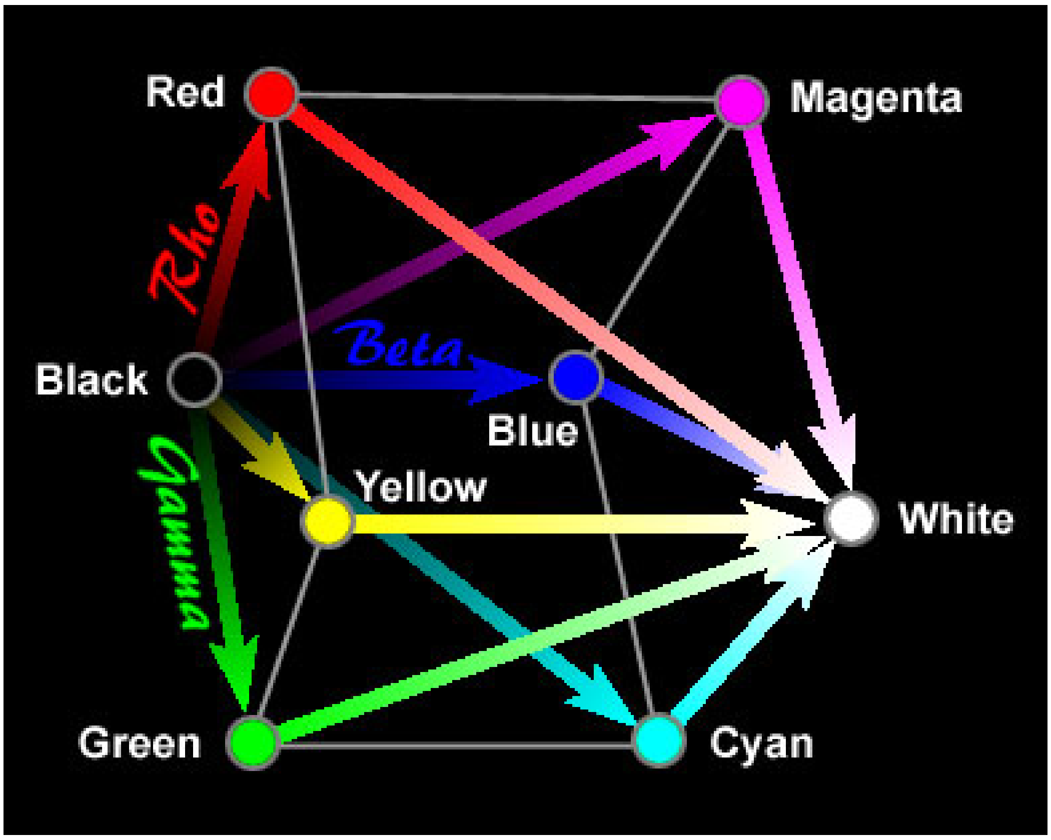


Figure 6: The Different Pathes from Black to White

There are *6 pathes* passing from black over a primary or secondary color corner to white. These 6 pathes give us interesting gradients of each of the primary and secondary color (Figure 7).

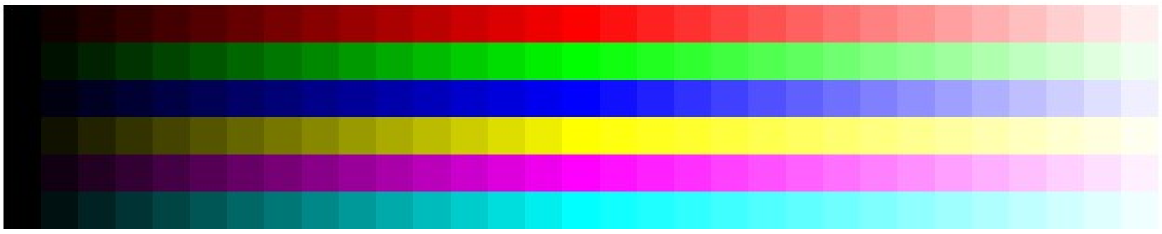


Figure 7: The 6 Gradients of the Primary and Seconary Colors

Of course, a multitude of pathes can be constructed. First there are the straight pathes from each corner to each other. Then there are pathes containing one single intermediate point within the cube. These pathes consists of two straight line segments. Our six gradients in Figure 7 are special cases of this kind of gradients. In the most general case, we may define any line or curve from any point inside the cube to any other point inside the cube to define a gradient.

3 Implementing the RGB Color

Discretisizing the edges along the color cube into 256 (2^8) units on each direction, allow us to define 256 colors on each edge along all three primary dimensions. Hence, the cube can define 256x256x256 (= 2^{24}) single colors. Moving along the *Black-Red* edge (the Rho sensor), the "red part" of the color is augmented from 0% to 100%. Moving along the Black-Green edge (the Gamma sensor), the "green part" of the color is augmented from 0% to 100%. Moving along the Black-Blue edge (the Beta sensor), the "blue part" is augmented from 0% to 100%. Hence, a triple (*r*,*g*,*b*) of integer number in the range of $0 \le r, g, b \le 255$ define a single point – or a single color – in the color cube. The triple (0,0,0) defines the black color, for example, and the (255,255,255) triple defines the white color (it has all three primary colors saturated). The six other corners are defines as:

(255,0,0)	red
(0,255,0)	green
(0,0,255)	blue
(255,255,0)	yellow
(255,0,255)	magenta
(0,255,255)	cyan

Specifying a RGB color, therefore, requires, 3 bytes, which can be implemented as a 4-byte integer in ordinary programming languages. Sometimes – for example in the Delphi environment – there exists a function that generates the integer out of the triple specification (*r*,*g*,*b*). It is:

```
C := RGB(r,g,b)
```

This function does nothing else than build an integer out of the three bytes *r*,*g*, and *b*. The same could be attained by shifting the corresponding 8 bits to the left:

```
C := r ; C := C shl 8 + g; C := C shl 8 + b;
```

where `shl` is a shift operator (in our case it shifts the bits 8 positions to the left). Another way to construct the color integer C is:

```
C := 255*255*r + 255*g + b;
```

although this is a relatively slow operation. The fourth byte is not use in this 4-byte integer specification.

4 Using the RGB Color in LPL

LPL as modeling language contains also a drawing tool. There are various drawing instructions that allow the modeler to generate a graphic. There is no need to install any graphic tool or a driver. Just calling the drawing instructions within an LPL model is all that the user needs to do. The modeler has a white space of 2000x2000 pixels on which he can draw. At the end of a run, the drawn space is saved as a JPG-file (normally called **`abc.jpg`**) or one can save it to a specific file using the function **`SaveJPG()`**. All the drawing function must be called with a preceded **`Draw`** followed be a dot an the function name. For example, to draw a line from (x,y) to (x1,y1) with the color c, one must write in an LPL model:

```
Draw.Line(x,y,x1,y1,c);
```

All functions are explained in the LPL Reference Manual [Hürlimann 2005]. Whenever a color is demanded – as here in the previous **`Line`** function the fifth argument c – one can use any expression returning a number. The expression can be a call to the function **`RGB(r,g,b)`** which returns the color number described in the previous section. Hence **`RGB(0,0,255)`** returns the number for the blue color, etc.

LPL also implements the 7 following pathes gradients from white to black described earlier. The pathes are discretisized in a way to have 32 intermediary color steps. The pathes are with corresponding color number are:

Gradient from white to black	**attributed color numbers in LPL**
Path from white to black (straight line)	32 to 63
Path through red	64 to 91
Path through green	92 to 127
Path through blue	128 to 159
Path through yellow	160 to 192
Path through magenta	193 to 223
Path through cyan	224 to 255

The numbers 0 to 31 are reserved for a list of saturated colors. The following LPL model reveals the attributed numbers:

```
MODEL colorsInLPL;
  SET y:=/1:8/  "Eight color tables";
      x:=/1:32/ "32 darkness in gradients";
  INTEGER PARAMETER a{y,x} := 32*(y-1)+x-1;
  Draw.Ratio(20,20);
  FOR {y,x} DO Draw.Rect(x,y,x+1,y+1,a); END
  --- using function RGB
  FOR{i={1:256}} DO Draw.Rect((i-1)/8,10,i/8,12,RGB(0,0,i-1)); END
  FOR{i={1:256}} DO Draw.Rect((i-1)/8,13,i/8,15,RGB(i-1,i-1,255)); END
END
```

This model draws the graph in Figure 8. (The result is stored in the **`abc.jpg`** file by default.) The model first defines a two-dimenational table of integers of dimension 8x32.

	1	**2**	**3**	**4**	**5**	**…**	**30**	**31**	**32**
1	0	1	2	3	4	…	29	30	31
2	32	33	34	35	36	…	61	62	63
3	64	65	66	67	68	…	93	94	95
4	96	97	98	99	100	…	125	126	127
5	128	129	130	131	132	…	157	158	159
6	160	161	162	163	164	…	189	190	191
7	192	193	194	195	196	…	221	222	223
8	224	225	226	227	228	…	253	254	255

Hence, the table maps the integer 0 to 31 to the first line, the integer 32 to 63 to the second line, etc, up to the eighth line. The instruction

```
FOR {y,x} DO Draw.Rect(x,y,x+1,y+1,a); END
```

draws 8x32 rectangles in a grid with the corresponding color as shown in Figure 8.

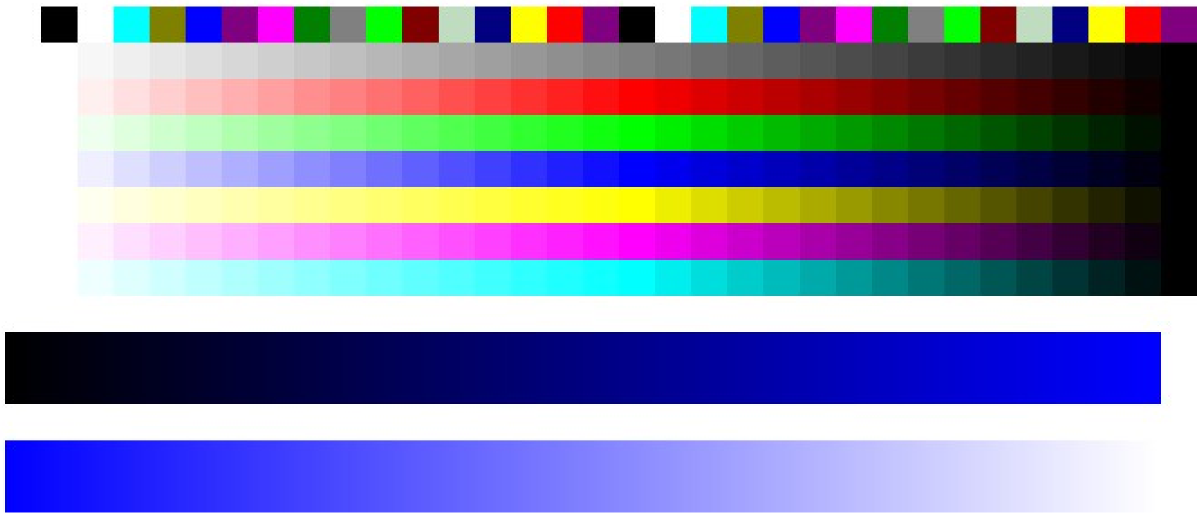


Figure 8: Attribute color numbers in LPL

The two other drawing instructions draw a finer gradient from black to blue, and from blue to white using the **`RGB(…)`** function for calculating the color number. Hence the instruction:

```
FOR{i={1:256}} DO Draw.Rect((i-1)/8,10,i/8,12,RGB(0,0,i-1)); END
```

draws a small slice rectangle with the colors **`RGB(0,0,i-1)`** with *i={1…256}*, hence a gradient from black to blue.
The instruction

```
FOR{i={1:256}} DO Draw.Rect((i-1)/8,13,i/8,15,RGB(i-1,i-1,255)); END
```

draws a small slice rectangle with the colors **`RGB(i-1,i-1,255)`** with *i={1…256}*, hence a gradient from blue to white (recall that blue is **`RGB(0,0,255)`** and white is **`RGB(255,255,255)`**).

These gradients are very useful in modelling, for example to plot the "intensity", in our case the size, of a number, without complicated color calculations. Figure 9 displays a 100x100 matrix with randomly generated numbers between 32 and 63. The darkness of a point corresponds directly to the size of the number generated. The LPL code that generates Figure 9 is as follows:

```
SET i alias j := /1:100/;
INTEGER PARAMETER c{i,j} := TRUNC(RND(32,63));
FOR{i,j} DO Draw.Rect(i,j,i+1,j+1,c); END
```

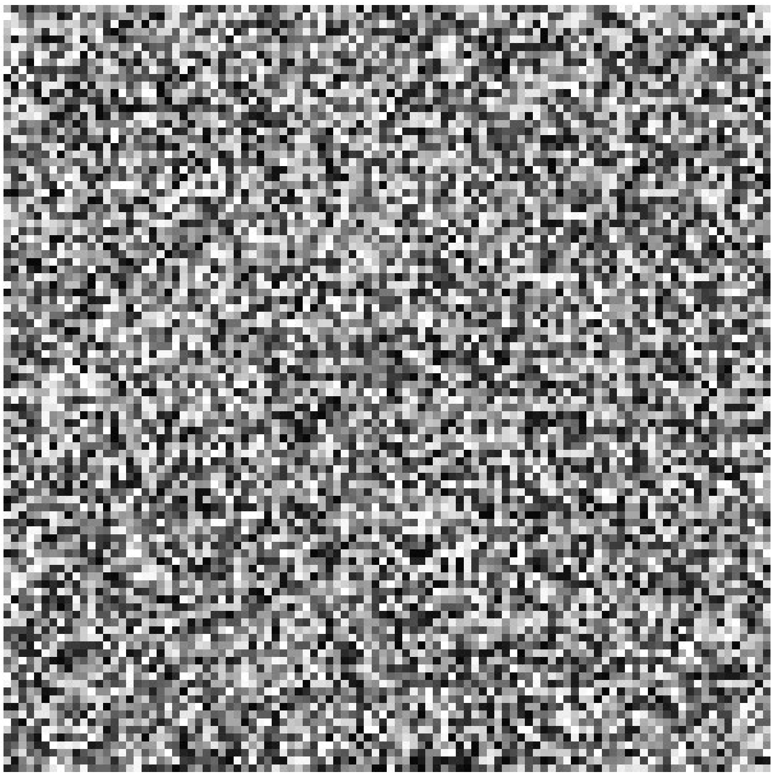


Figure 9: Randomly Generated Matrix

Some more interesting patterns are functional displays, where the matrix represents a pattern as in Figure 10. These pattern are generated by the two color tables:

```
INTEGER PARAMETER c{i,j} := (i^2+2*j)%32+64;
INTEGER PARAMETER c{i,j} := (sqrt(10*i*j))%32+128;
```

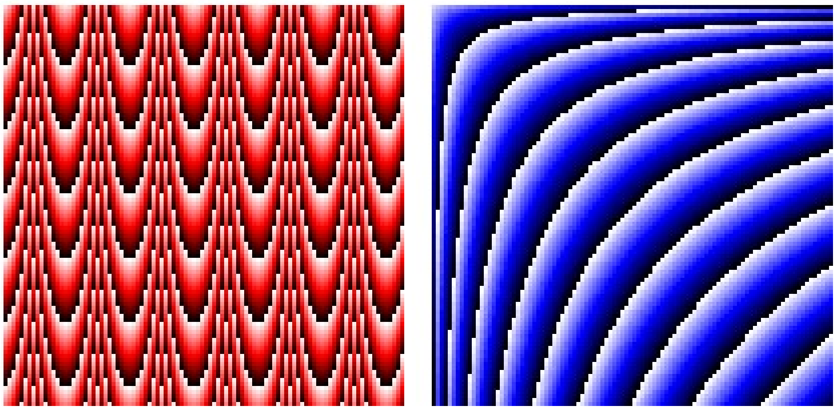


Figure 10: Some Patterns

5 Implementation of gradient pathes

In this section, an implementation of the gradient pathes is given. Straight line gradients are easy to implement. Suppose, one wants a gradient path from black to red, for example, then one only has to map the colors RGB(0,0,0) to RGB(255,0,0) to consecutive integers. This can easily be done by just defining the set RGB(i,0,0) with 0 ≤ i ≤ 255. The color step width can also be chosen, for example, to get a gradient path with 32 consecutive colors only needs to define the set RGB(8*i,0,0) with 0 ≤ i ≤ 31.

The path gradient of a surface diagonal in the RGB color space (for example from red to white) must be implemented by successively adding the two colors green and blue to the already saturated red, hence to define the set as RGB(255,i,i) with 0 ≤ i ≤ 255.

The inner diagonal (straight line from black to white) can be obtained by adding all three colors simultaneously, that is by defining the set RGB(i,i,i) with 0 ≤ i ≤ 255.

The get the combined gradient from black to red to white (for example), one must combine two straight line gradients as follows: RGB(i,0,0) with 0 ≤ i ≤ 127 and RGB(255,i,i) with 128 ≤ i ≤ 255. There is no way to combine these two line in smooth function *f(i)*, since the path is non-differentiable.

A possible implementation in Pascal of the mapping function is as follows:

```
function MyRGBColor(r,g,b,dark:integer):integer;
// r,g,b [0..255] , dark [0..31]
var i,c:integer;
begin
  if dark<16 then
    c:=RGB(round(r/15*dark),round(g/15*dark),round(b/15*dark))
  else begin
    dark:=dark-15;
    c:=RGB(r+round((255-r)/16*dark),g+round((255-g)/16*dark),
           b+round((255-b)/16*dark)); end;
  MyRGBColor:=c;
end;
```

This function can be used to map all kind of path gradient from white to black going through an intermediate point (r,g,b). For example:

`MyRGBColor(127,127,127,63-i)  with  32 ≤ i ≤ 63`

maps the integer i into the gray gradient from white to black (straight line from RGB(255,255,255) to RGB(0,0,0)). Or

`MyRGBColor(255,0,0,95-i)  with  64 ≤ i ≤ 95`

maps the integer i into the red gradient from white to black (two straight lines from RGB(255,255,255) to RGB(255,0,0) to RGB(0,0,0)). Or

`MyRGBColor(0,255,255,255-i)  with  224 ≤ i ≤ 255`

maps the integer i into the cyan gradient from white to black (two straight lines from RGB(255,255,255) to RGB(0,255,255) to RGB(0,0,0)).

Any intermediate point (r,g,b) cound be given to get a gradient from RGB(255,255,255) to RGB(r,g,b) to RGB(0,0,0)) in a range of i = [31…0] by just calling the function MyRGBColor(r,g,b,i).

6 References

[Color Vision], http://www.photo.net/photo/edscott/vis00020.htm.

Hürlimann T., [2005], Reference Manual of LPL, Version 4.71, www.virtual-optima.com.