Lena by Anna Donninger

MonaLena

A C++ Toolbox for Digital Dithering

Dr. Chrilly Donninger

Abstract:
This paper documents the **MonaLena** toolbox for Digital Dithering**.** The toolbox processes a color image in a 4 stage pipeline Color → Grayscale → Preprocessing → Dithering → Postprocessing. For each stage of the pipeline several algorithms are implemented which can be combined freely. One can create zillions of different halftone transformations from one color image. This paper serves also as a comprehensive introduction to the field. The Toolbox can be downloaded from https://github.com/DonChr/MonaLena

Introduction:
The Lena test image has been called the Mona-Lisa or the Madonna of the internet age. There is – in contrast to Mona-Lisa – no mystery about the woman on the image. It is the Swedish model Lena Forsen and the picture is from the centerfold of the November 1972 playboy. There is hence more to see on the full image. MonaLena is a yet another homage to Lena Forsen.
In recent years a discussion emerged about the political correctness of this image. The Bloomberg journalist Emily Chang called it "*tech's original sin*." The campaign „*Losing Lena*" demands: „*Why we need to remove one image and end techs original sin*". I did not know that it is so easy to dispel the evil under the sun.

The real and serious problem with Lena is: The picture is too perfect and too well known. It's like an Elvis Presley imitation. Even if the imitator is not good at all, it sounds reasonable, because our brain hears Elvis and not the imitator. Lena contains only a very restricted palette of colors. Red is dominating, green is missing and there are no clear edges which must be preserved.

The principal test image for MonaLena is Trini. It's a well made "*real life*" image. Trini is nice, but not as perfect as Lena. Especially the glasses and the green leaves build well defined edges. There are blue- and a variety of green shades present.

Color to Grayscale Conversion:
The first step in the MonaLena halftone pipeline is reading in the input and conversion from color to gray scale. MonaLena is a self-contained library with no external dependencies. It uses the `stb_image - v2.25 -` public domain image loader. These are just 2 C-Header files. The files are included in the distribution. The loader supports the *.jpg format.

There are several models for converting a color image to gray scale. The most popular is saturation. The gray scale intensity I is a weighted sum of the RGB values.

```
bool Saturate(const string fileName, double wRed, double wGreen, double wBlue);
```
is a general conversion method. The weights can be set to any value. The sum of the weights can differ from 1.0. The only restriction is: The gray scale I is stored as an int32_t. There must not be an over- and underflow. MonaLena handles gracefully values outside the [0,255] range. But at output the values are clamped to the [0,255] range. For my archery application ***Eagle-Eye*** the weights were set to 0.35 for each color channel. This improved the recognition rate of the hit-pattern. One could even think of negative weights for a channel.

```
bool SaturateGIMP(const string fileName, double scaleFac = 1.0);
```
Is a convenience function for Saturate. It sets the weights to the saturation values of GIMP (0.3,0.596,0.11). These weights are multiplied by the scale factor. With the default value the conversion is identical to GIMP.

```
bool SaturateQt(const string fileName, double scaleFac = 1.0);
```
This convenience function is the same as above. It uses the weights of the QImage class (0.34375,0.5,0.1625). Qt uses in fact the integer operations 11/32, 16/32, 5/32.


GIMP


Qt


R=0.35, G=0.35, B=0.35

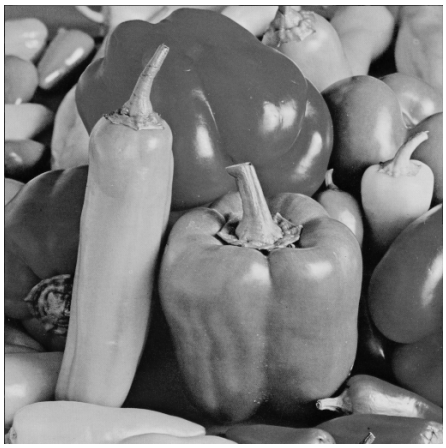

Helmholtz


Desaturate


Value

```
bool Helmholtz(const string fileName,double factor=0.149);
```
Helmholtz uses the same saturation weights than GIMP. But it corrects for the Helmholtz-Kohlrausch effect. Pure colors are perceived brighter than mixed colors. This is especially true for red. The difference can best seen at the red lips of Trini.


Peppers


Peppers-GIMP


Peppers-Helmholtz 0.5

The difference between GIMP and Helmholtz for the peppers test image. The Helmholtz factor was set to 0.5 to emphasize the difference.

```
bool Desaturate(const string fileName);
```
The Desaturate method sets the gray scale I to (Max(R,G,B)+min(R,G,B))/2. This conversion is also supported by GIMP.

```
bool Value(const string fileName);
```
The Value method sets I to Max(R,G,B). This conversion is also supported by GIMP.

```
bool ColorChannel(const string fileName,int color = 0);
```
The gray scale is set to the given color channel. With Red==0, Green==1, Blue==2.

If the input file is in gray scale format the values are just copied.

Pre-Processing:
Before dithering the gray scale image can be filtered. The filter transforms the image to another gray scale representation. The effects will be demonstrated for the standard house test image. The effects can be best seen at the contours of the car, the sign on the front door, the door joints and the stones behind the rear.

```
bool Gauss77Filter();
```
The Gauss77 filter uses a separable kernel of 1,6,15,20,15,6,1. Separable kernels reduce the number of multiplications per pixel from m^2 to 2*m (m is the size of the kernel). The filter smooths/blurs the image. Dithering is essentially also a smoothing operation. One represents the gray scale I by the density of white and black pixels in an area. The human eye integrates the area to the corresponding gray scale. Gauss usually does not improve the halftone. But it can be used for post-processing and for measuring the quality of halftone representations (see below Optimal Thresholds).
`bool Gauss55Filter()` is the little brother with a kernel of 1,4,6,4,1. (not shown).

```
bool MedianFilter9();
```
Replaces each pixel with the median of its 3x3 area. This is like the Gaussian a smoothing operation. But the median preserves edges. It removes effectively salt&pepper noise.

```
bool LaplaceSharpen(double factor=-1.0);
```

Laplace is a high pass filter. It sharpens the edges of the image. The Laplace filter is added to the gray scale of the image. Bright edges will be brighter, dark darker. One can adjust the effect with factor. If factor has a positive sign the effect will be reversed (which is usually not the intention behind using the Laplace).

House

House-GIMP

House-GIMP-Gauss7

House-GIMP-Median

House-GIMP-Laplace

House-GIMP-MedLaplace

House-GIMP-KnuthEdge

House-GIMP-Logistic

House-GIMP-Rescale

```
bool Med5Laplace(double factor = -1.0);
```
Removes first salt&pepper noise with a median-5 filter (the pixel and its N,W,S,E neighbors) and applies then the Laplace filter. Although the result does not look good in most cases, the filter improves usually the final halftone image. The wrinkles in Trinis face are removed by the halftone blur.

Trini-Helmholtz

Trini-Helmholtz-MedLaplace

```
bool KnuthEdge(double factor = 0.8);
```
This method was proposed by Donald Knuth to enhance edges according the formula
$I=(I-factor*\bar{I})/(1-factor)$.
Where $\bar{I}$ is the mean over the 3x3 area around the pixel. The effect is similar to the Laplace filter.

```
bool Logistic(double scale=0.025);
```
Transforms the gray scale with a logistic function. The dark pixels are moved towards dark, the intensities in between are stretched and the bright pixel are moved towards white. A larger scale factor emphases this effect. For details see the documentation of the method in MLGray.h

```
bool Rescale(double offset = 25.5, double factor = 0.8);
```
Rescales the gray intensity by the linear transformation
I=offset+factor*I.

One can make an image brighter with a positive offset and factor == 1.0. The main purpose is according to D. Knuth to remove ugly artifacts in the transformed image. One has to play around with the parameters. According to my experience the other pre-processing methods are more effective.

Digital Dithering

The next stage in the pipeline is digital dithering. It transforms the 255 gray scale I to a halftone image. The pixels are either black (0) or white (255).


Trini-GIMP-Threshold:128


Trini-GIMP-ML-Threshold:128


Trini-GIMP-ML-Random

```
bool Threshold(int32_t threshold = 128);
```

Threshold is the most primitive dithering algorithm. If the intensity I >= threshold the pixel is mapped to white, below to black. As can be seen on the left picture thresholding destroys relevant structures of an image. Adjusting the threshold usually does not improve the result. It preserves one structure and removes instead another one. The halftone improves if one applies the MedLaplace beforehand. The lips, necklace and glasses and also the green leaves are preserved.

```
bool Random();
```

Creates at each pixel a random integer R in [0,255]. If I >= R the pixel is set to white, otherwise black. A bright pixel has a higher chance to be set to white, a dark to black. The result is in combination with MedLaplace not so bad.

I have not found this simple method in the literature. It's maybe my own "*invention*". The literature mentions a method where first noise is added to the pixel value and then thresholding is done. Random is a somewhat different idea. If the noise is in the range [-64,64] a value of 51 will always be black. Random is a different idea. An intensity of 51 has in Random() a chance of 20% to be set to white. This is from the integration point of view the correct density. Random adds too much noise to the picture. But it avoids also repeating patterns/texture which are in other methods a nuisance.


Trini-GIMP-ML-Bayer44


Trini-GIMP-ML-Bayer88


Trini-GIMP-ML-Bayer88Rnd

```
bool Bayer44();
```
Ordered Dither with a Bayer matrix is a classical dither method. The Bayer matrix contains the gray scale I in steps of 16. One accesses the entry of this matrix by x%4 and y%4 of the pixel coordinates. One performs then simple thresholding with the matrix-entry. If one has e.g. an area with a pixel value of 64, 4 entries of the matrix will be less equal than 64, 12 larger and hence the correct integrated value is generated. But a pixel value of 70 would imply the same pattern. The 4x4 matrix can only represent 16 shades of gray. The Bayer matrix is the optimal arrangement of the elements (for the details see the code in MLGray.y). The Bayer matrix preserves the structure of the image, but it generates a pronounced texture.

```
bool Bayer88();
```
Is exactly the same method, but it uses an 8x8 matrix. There are more gray shades available, but the texture effects are even more pronounced.

```
bool BayerRnd88(int32_t range=20);
```
Adds a uniform distributed number in [-range,range] to the pixel value and performs then the Bayer88 dither. This also seems to be my “invention”. The idea was to improve the texture effect. The result is not really better and but also not really worse than plain Bayer88().

Error-Diffusion

Trini-GIMP-FloydSteinberg:128

Trini-GIMP-Ostromoukhov:128

Trini-GIMP-Jarvis:128

Trini-GIMP-ML-FloydSteinberg:128

Trini-GIMP-ML-Ostromoukhov:128

Trini-GIMP-ML-Jarvis:128

The methods so far have one big advantage. They are simple and fast. This was historically an important point. But the methods of choice are nowadays based on error-diffusion. The error is the difference between the original pixel value I and the halftone. It's I-255 if the pixel is mapped to white and I if it is mapped to black. This error is diffused in the forward direction of the scan process to pixels in the neighborhood. There have been a number of methods proposed. The differ in the definition of neighborhood and the diffusion weights. The results look usually similar. The diffusion process creates like Bayer() or Random() the correct density. It can represent the full range of gray shades and the texture effects are much less visible.

```
bool FloydSteinberg(int32_t threshold = 128);
```
This is the ubiquitous dithering algorithm. It's the standard in GIMP. It diffuses the error according to
```
.   x   7
3   5   1
```
where x is the current pixel. The diffusion coefficients must be divided by 16. 7/16 of the error is added to the pixel to the East, 3/16 to SW, 5/16 to S and 1/16 to SE.
It is according my tests (see optimal Thresholds below) indeed the best algorithm. But the visual differences are in most cases minor. The gray scale conversion method, pre- and post-processing has a more pronounced influence. One should tune these steps and use Floyd-Steinberg for dithering.

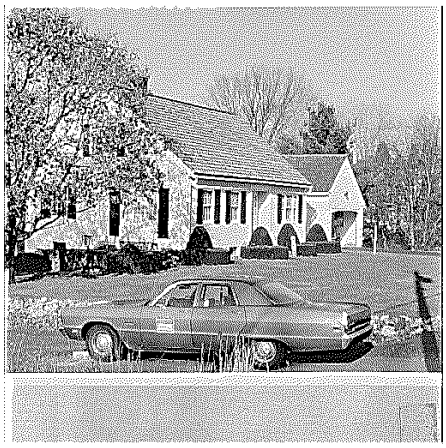
```
bool Ostromoukhov(int32_t threshold = 128);
```
The Ostromoukhov algorithm diffuses the error only to the E, SW and S neighbors. It uses for each gray-level different precalculated error diffusion coefficients. The coefficients were optimized according to the Fourier analysis of the error noise. This is from the academic point of view an interesting approach. From the practical point of view the result is about as good as Floyd-Steinberg and requires about the same number of operations.

```
bool Jarvis(int32_t threshold = 128);
```
This algorithm was published already before Floyd-Steinberg. The error is diffused to a larger area. The quality is according my measurements somewhat inferior to the standard algorithm. For details see the code in MLGray.cpp.

The first line problem:
If the first/top line of an image is either bright (e.g. sky) or dark error diffusion works initially like simple threshold. It takes some time till the diffusion mechanism reaches a stable state. To reduce this effect MLGray copies line 0 to a virtual line -1. The error diffusion process starts at line -1 and the errors of this virtual line (which is of course not drawn) are diffused to line 0. This simple procedure is not perfect, but it improves in some pictures the diffusion at the top considerable.
Note: I have not found a description of this effect in the literature. I assume that I have reinvented this wheel, too.

What is the best threshold?
So far the user has to enter the threshold for dithering by hand (or be happy with the default value 128). Ideally this threshold should be determined automatically. But what is the optimization criterion? A simple method is: Dithering works, because the human eye integrates over images areas. As a first crude approximation one can consider the eye as a Gaussian filter. The standard pipeline is to convert with GIMP to gray scale, sharpen the image with MedLaplance and then perform FloydSteinberg. In the left picture below no dithering is done, the image is just post-processed/blurred with a Gauss77 Filter. The picture in the middle is the standard halftone result. The right picture post-processes the

House-GIMP-ML-Gauss7



House-GIMP-ML-FloydSteinberg

House-GIMP-ML-FloydSteinberg-Gauss7

halftone image with the same Gauss77 filter as on the left. This step can be considered as the reconstruction of the halftone image. The result looks not too bad. A possible definition of an optimal dithering algorithm is: *The reconstructed image should look as similar as possible to the original one*.

For determining the optimal threshold MLGray calculates the L1-distance between the Gaussian of the original picture (on the left) with the Gaussian of the halftone (on the right). One could compare the original picture with the reconstructed, but comparing the two Gaussian gives slightly more appealing results. One could also use the L2-distance. The results are usually (almost) identical. The corresponding methods are

```
int OptFloydSteinberg(int from = 64, int to = 192);
int OptOstromoukhov(int from=64,int to=192);
int OptJarvis(int from = 64, int to = 192);
```

The algorithm searches first with a step size of 4 the best value in the range [from,to] and refines in the second step the search around [best-3,best+3] to find the final best threshold. A bisection or golden-ratio optimum finding algorithm would be faster, but this primitive approach seems to be safer. It makes less assumptions about the error-curve. The procedure is on a modern PC for moderate sized pictures fast enough. It should be noted that the best threshold according this simple criterion must not look best. As a rule of thumb try first 1 or 2 own thresholds and compare the result with the image produced by the optimizer.

Post-Processing:

The left picture above is the result of Trini-Helmholtz-ML-FloydSteinberg. In the middle picture a post-processing step with Salt&Pepper is added. This is the favorite result of Trini. In the right picture the image is post-processed with a majority filter. The result looks similar to the simple threshold.
It is not the purpose of post-processing to make the image more similar to the original. It can be used to create artistic effects.

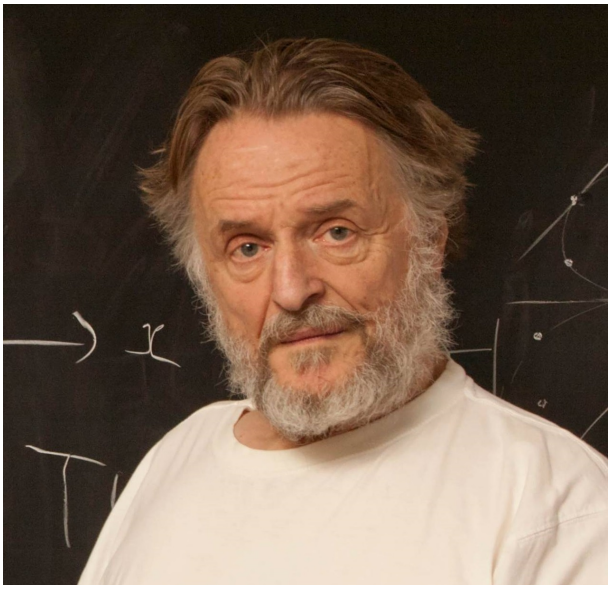
```
bool SaltPepper(int32_t threshold = 1);
```
“Salt” is a white pixel within a black area”. “Pepper” is a black pixel in a white area. This post-processing filter removes both. If the neighbors of a pixel all have the opposite color the pixel is flipped to the color of the neighbors. If the threshold is set to 2, the pixel is flipped, even if there is one neighbor with the same color. I got only reasonable results with the default setting.

```
bool HalftoneMajority();
```
Sets the halftone value to the majority in the 3x3 area. This is a special case of a median filter.

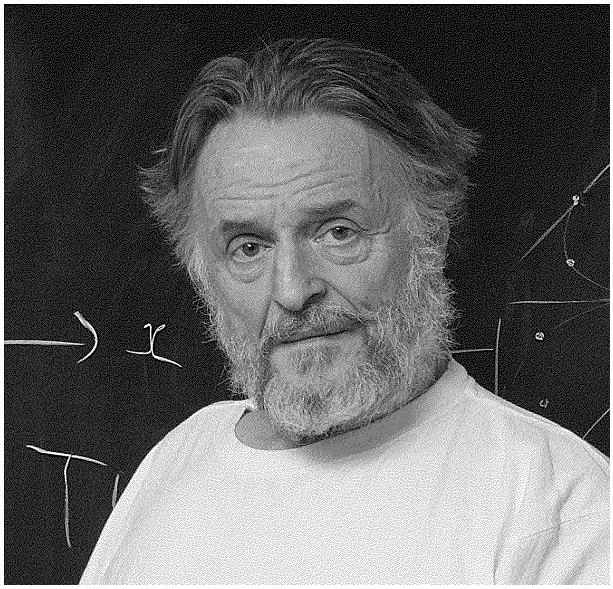
Game of Life:

```
bool GameOfLife(bool whiteAlive=true,int generations=1);
```
One of the prominent fatalities of the Covid-19 pandemic was the mathematician John Conway. The GameOfLife method post-processes a halftone image with the rules of Conway’s Game of Life.
A living cell survives if it has 2 or 3 living neighbors. A dead cell becomes alive if there are 3 living cells around. All other combinations result in a dead cell. If the parameter “whiteAlive” is set to true, the white pixels are considered alive. If whiteAlive is false, the black pixels are the living ones. The second parameter specifies the number of generations.

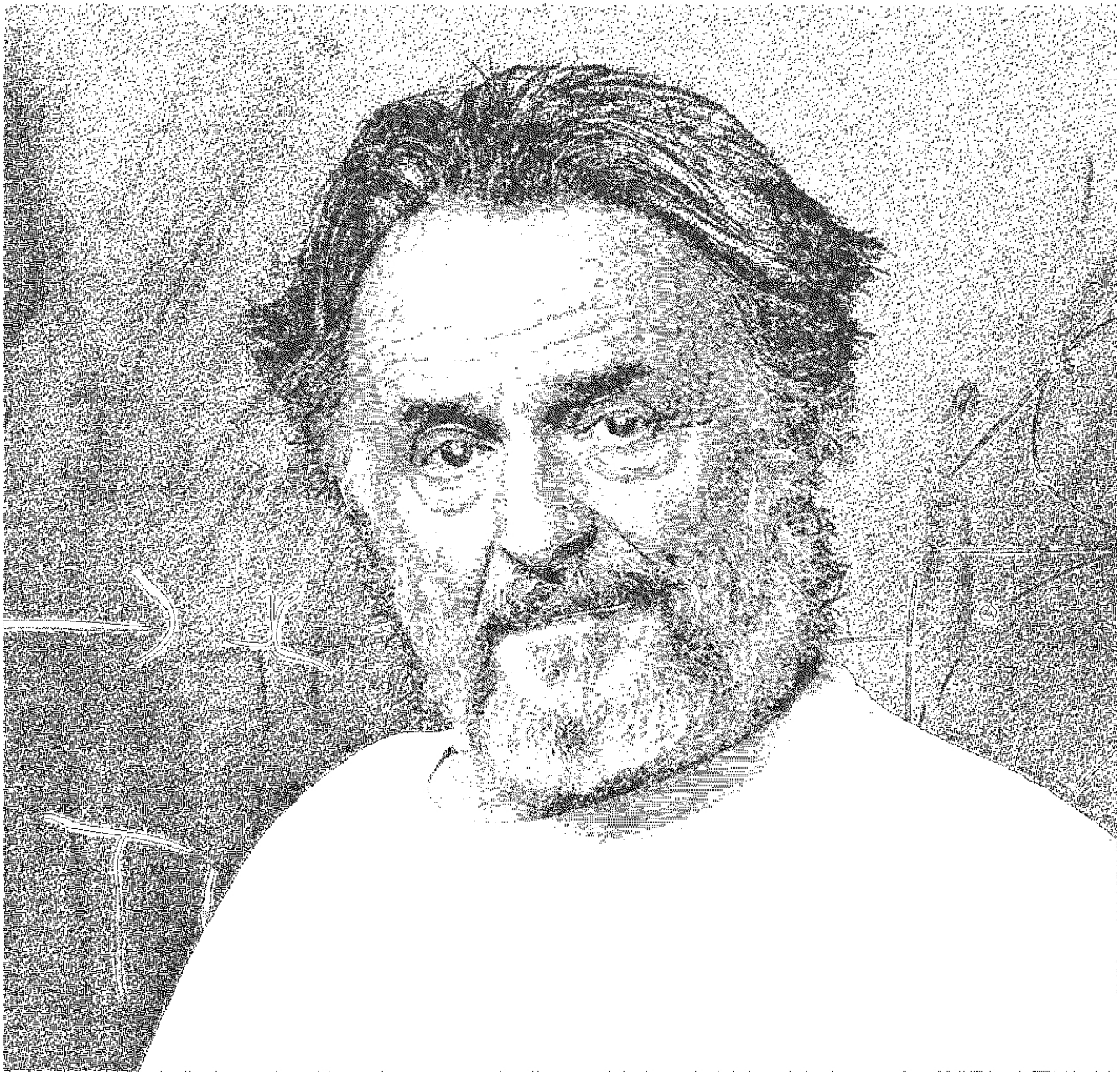
The dither pipeline can in this case be interpreted as a tool to create interesting start positions for the Game of Live. When I started the MoneLena project GameOfLive was not on the agenda. It came to my mind when I spoke with my friend Steffen Jakob about the death of John Conway. The method is a sort of obituary.



John Conway



Conway-Helmholtz-ML-Floyd-Steinberg



Conway Game Of Life, Black-Alive, after 1. Generation.

The result is with - whiteAlive==false, generations==1 - quite interesting. But the picture is fading away after a few generations. There are no stable patterns present.
The original pictures has 1275x1223 pixel. One gets better results if one scales the original image first to 2550x2446 pixels (not shown).

The Specification of the MonaLena Pipeline:

One specifies the MonaLena pipeline with a *.csv file. Each line contains one pipeline operation. The operations are independent from each other. The result does not depend on the order of the lines. Currently it is not possible to feed the result of one pipeline in the next one. A workaround is to use the result file in the next batch as input.

```
#input-image,grayconverter,preprocess,halftone,postprocess,output-image
Trini,GIMP,MedLaplace,OptFloydSteinberg,,Trini_GIMP_ML_OptFloydSteinberg
```

The first line starting with '`#`' is a comment. The first column is the name of the input file without the "*.jpg" extension. It is assumed that the image is stored in the ./image subdirectory. The command reads in "./image/Trini.jpg". The 2nd column is the color to gray conversion. It is in this case the SaturateGIMP method with the standard parameter. The 3rd column is the pre-processing filter. It is MedLaplace(). The 4th column is the dither algorithm. It is Floyd-Steinberg with automatic optimization of the threshold value. The 5th column is the post-processing filter. The entry is empty, post-processing is skipped. The final 6th column is the name of the result file. Again without the *.jpg file extension. The result is stored as an RGB image to the ./result subdirectory. The full filename is therefore `./result/Trini_GIMP_ML_OptFloydSteinberg.jpg`.
The name is arbitrary. One could have named it `Trini_Likes_It`. Usually the *.csv contains several lines. One creates dozens of pictures in one batch. A consistent naming convention is essential to identify the images.
Most methods have parameters. The example above uses for all operations the default values. If one wants to perform Floyd-Steinberg with a threshold of 130 the 4th column would be: `FloydSteinberg:130`. The method saturate has 3 parameters. One calls saturate with the command `Saturate:0.30:0.70:0.10`. This specifies the call to `Saturate(0.30,0,70,10);`

```
Trini,Helmholtz,MedLaplace,FloydSteinberg:194,GameOfLife:0:1,Trini_Helm_ML_FloydSteinberg_GoL01
```

This is the command for generating the top-left Game of Life image. Floyd-Steinberg is called with a threshold of 194 and the post-processing method is GameOfLife(0,1). Black is considered alive, the game is run for 1 generation.

Command	MLGray::Method
`ColorChannel:channel`	`bool ColorChannel(const string fileName, int color = 0);`
`GIMP:scaleFac`	`bool SaturateGIMP(const string fileName, double scaleFac = 1.0);`
`Qt:scaleFac`	`bool SaturateQt(const string fileName, double scaleFac = 1.0);`
`Helmholtz:factor`	`bool Helmholtz(const string fileName, double factor=0.149);`
`Desaturate`	`bool Desaturate(const string fileName);`
`Value`	`bool Value(const string fileName);`
`Saturate:wRed:wGreen:wBlue`	`bool Saturate(const string fileName, double wRed, double wGreen, double wBlue);`

Table-1: 2nd column. Gray scale conversion commands.

Note: The parameter fileName is generated from the 1th column as described above. Parameters with default values can be omitted. The parameters must have the same type as in the method call.

Command	MLGray::Method
Gauss5	bool Gauss55Filter();
Gauss7	bool Gauss77Filter();
Laplace:factor	bool LaplaceSharpen(double factor=-1.0);
MedLaplace:factor	bool Med5Laplace(double factor = -1.0);
Edge:factor	bool KnuthEdge(double factor = 0.8);
Logistic:scale	bool Logistic(double scale=0.025);
Rescale:offset:factor	bool Rescale(double offset = 25.5, double factor = 0.8);
Median	bool MedianFilter9();

Table-2: 3rd column. Pre-Processing commands.

Command	MLGray::Method
FloydSteinberg:threshold	bool FloydSteinberg(int32_t threshold = 128);
Jarvis:threshold	bool Jarvis(int32_t threshold = 128);
Ostromoukhov:threshold	bool Ostromoukhov(int32_t threshold = 128);
OptFloydSteinberg:from:to	int OptFloydSteinberg(int from = 64, int to = 192);
OptJarvis:from:to	int OptJarvis(int from = 64, int to = 192);
OptOstromoukhov:from:to	int OptOstromoukhov(int from=64,int to=192);
Bayer44	bool Bayer44();
Bayer88	bool Bayer88();
BayerRnd88	bool BayerRnd88();
Random	bool Random();
Threshold:threshold	bool Threshold(int32_t threshold = 128);

Table-3: 4th column. Dither commands.

Command	MLGray::Method
SaltPepper:threshold	bool SaltPepper(int32_t threshold = 1);
Gauss5	bool Gauss55Filter();
Gauss7	bool Gauss77Filter();
Majority	bool Majority();
GameOfLife:life:generations	bool GameOfLife(bool whiteAlive=true,int generations=1);

Table-4: 5th column. Post-Processing commands.

The values in the 1st,2nd and 6th column are mandatory. The entries in the 3rd, 4th and 5th column are optional.
For GameOfLife the command parameter must be an integer. 0 is interpreted as false, all other values are true.

Installation:

Download or clone the code from https://github.com/DonChr/MonaLena.
A project file for Visual-Studio-2019 is included. The code should compile and run out of the box.
For other IDEs and compilers one has to configure the system. The code has no external dependencies and the structure of MLGray is rather simple. The configuration should be relative straightforward.
The presented example images (and a few more) are in the ./image subdirectory.

If you want to process the commands in “trini.csv” start with

```
MonaLena.exe trini
```

It is assumed that “trini.csv” is in the same directory as the exe and the images are stored in ./image.
The result is stored in ./result. This subdirectory must exist. Visual Studio places in the defaut project setting in x64/Release or x64/Debug. Either copy the exe to the MonaLena root directory or change the Visual settings accordingly.
Personally I start directly from Visual-Studio. The command file can be set under Projects/Properties/Debugging/Command Arguments.

Acknowledgments:
To Anna Donninger for providing the Lena-Illustration.
To Trini Wohlfart for the picture and commenting on the output.
To Steffen Jakob for discussion and proof-reading.

Used Literature:
Rafael C.Gonzalez, Richard E. Woods: Digital Image Processing, Fourth Edition, Pearson
Raja Bala, Reiner Eschbach: Spatial Color-to-Grayscale Transform Preserving Chrominance Edge Information.
Victor Ostromoukhov: A Simple and Efficient Error-Diffusion Algorithm.
Donald E. Knuth: Digital Halftones by Dot Diffusion
FiveKo: Gaussian Blur – Noise Reduction Filter
https://fiveko.com/tutorials/image-processing/gaussian-blur-filter/
Visgraf: Ordered Dithering: https://www.visgraf.impa.br/Courses/ip00/proj/Dithering1/