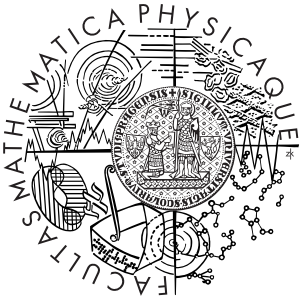
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Bohumír Zámečník

Advanced halftoning methods

Department of Software and Computer Science Education

Thesis supervisor: RNDr. Josef Pelikán
Study program: Computer Science, Programming

2009

I would like to thank hereby to my supervisor, RNDr. Josef Pelikán, for his valuable suggestions during the creation of the thesis and that he showed me the door to a very interesting area of digital halftoning. Also, I would like to thank to my parents for their support of my studies.

I declare that I have written this bachelor thesis on my own and entirely using the cited sources. I agree to lending of this thesis and its publishing.

In Prague, 3 August 2009 Bohumír Zámečník

Contents

Název práce: Pokročilé metody halftoningu
Autor: Bohumír Zámečník
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. Josef Pelikán
E-mail vedoucího: Josef.Pelikan@mff.cuni.cz

Abstrakt: Cílem této práce bylo navrhnout a implementovat modulární knihovnu pro metody digitálního halftoningu se zaměřením na fotorealistické metody využívající modrého šumu a fraktálních křivek vyplnujících plochu. V úvahu byly vzaty i umělecké metody. Z hlediska softwarového návrhu byl kladen důraz především na modularitu a rozšiřitelnost tak, aby bylo umožněno experimentování, a to i s novými algoritmy.

Knihovna spolu s grafickým rozhraním byla začleněna do prostředí populárního grafického editoru GIMP ve formě plug-inu.

Klíčová slova: halftoning, dithering, space-filling curves, blue-noise

Title: Advanced halftoning methods
Author: Bohumír Zámečník
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Josef Pelikán
Supervisor's e-mail address: Josef.Pelikan@mff.cuni.cz

Abstract: The goal of the thesis was to design and implement a modular library for digital halftoning methods. The aim were photorealistic methods utilizing space-filling curves and blue-noise, artistic methods were considered as well. From the point of view of software design, emphasis was on modularity and extensibility, in order to support experimenting, even with new algorithms.

The library along with a graphical user interface has been integrated as a plug-in with the framework of a popular image-processing application GIMP.

Keywords: halftoning, dithering, space-filling curves, blue-noise

Chapter 1

Introduction

1.1 What is halftoning?

Halftoning is a method of converting a continuous tone image into an image composed of only a few intensity tones (usually just two), which looks for the human eye similarly to the original image. It comes from the need of reproducing continuous tone images on devices capable of using only a few tones. In print industry greyscale images are usually printed using black ink dots on white paper, or color images are printed using a few basic color inks.

In general, halftoning performs tone quantization, but with respect to the properties of human visual system (HVS). The eye has a limited spatial resolution ability, so at certain looking distance, where dots seem small enough, they get blured, which results in perceiving an average tone [19]. There appear to be tones between physical device tones, hence the term *halftoning*.

As there are output devices with different properties, various halftoning methods exist to address those aspects. Besides printers halftoning was routinely done on computer displays with bit-depth limited to less than 8 bits per channel.

In addition to standard halftoning methods, where the primary goal is a precise image representation, there are other methods, considered artistic, where rather an interesting look is demanded, especially in details. Also the cool rough look of enlarged traditional newspaper screening dots stays in graphic designer's toolbox for years as an evergreen.

1.2 Project goals

The goals of this thesis were:

- to study various existing halftoning algorithms
- to design and implement a modular and extensible library for haltoning algorithms including some basic and advanced algorithms, in particular SFC clustering, some methods with good blue-noise properties and some artistic approaches

- to create a graphical user interface
- to integrate the project with an existing image editor

Potential target users benefitting from this project may include curious students of computer graphics exploring various halftoning algorithms or reserching new ones, or graphic designers wishing to use some cool graphic effects.

Current constraints are: input – digital greyscale images on square grid, output – digital black-and-white or greyscale images on square grid. In this project we limit us to bi-level output, but leave the door open to support multi-level halftoning in future.

The project was named *Halftone Laboratory* (or *HalftoneLab* in abbreviation) to suggest the ability to experiment with various methods.

1.3 Symbols and conventions in this document

In this paper we will work with greyscale raster images. An image element is refered to as a *pixel* and its value as *intensity* or *tone*. In theory the intensity could be continuous, usually normalized to the interval $[0.0, 1.0]$, however, computers are able to reproduce only finite number of possible values, so in practice pixel intensities are discrete. Eg. for 8-bit precision there are $256 = 2^8$ intensity levels, usually represented as integers in interval $[0, 255]$. Possible intensity values in a binary image lie just in the $\{0, 1\}$ set.

Input image: I, output image: O. Image pixel: $I(x, y)$ with intensity $\in [0.0, 1.0]$ where 0.0 is black and 1.0 is white.

Chapter 2

Halftoning methods

To be able to design a universal halftoning library it is necessary to acquaint oneself with the broad area of halftoning methods and related problems. A classification and a brief description of some selected traditional and some modern methods follows.

2.1 Historical background

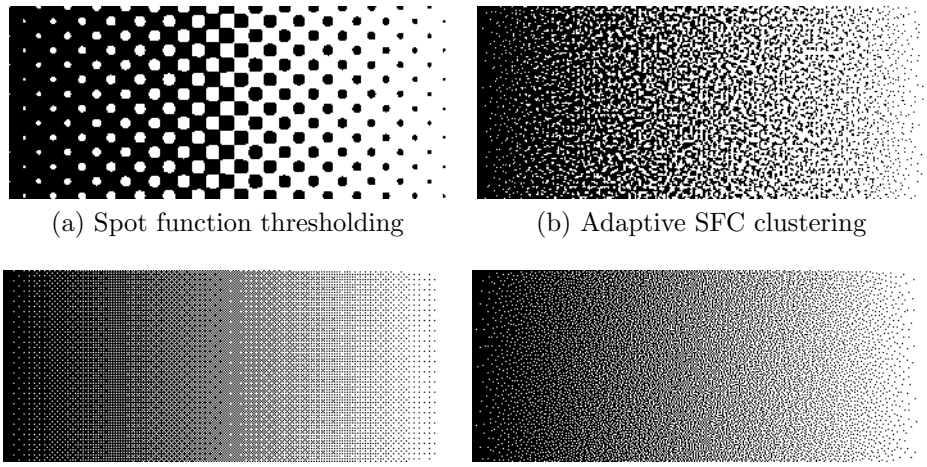
Since the early days of visual arts there were drawing techniques for displaying intermediate shades using only one colour tone, which could not be further diluted. An example is to use hatching technique in drawing, etching, engraving or in woodcut – to simulate continuous tones thin lines varying in thickness, spacing and angle are drawn. In another print technique, mezzotint, tonality is simulated by selective roughening and smoothing of the metallic print plate surface [40].

With the advent of photography came the need to print photographs via traditional press. In in second half of 18^{th} century screening was invented. A photograph was exposed to a metal photogravure plate through a fine silk screen, or later a ruled glass or film, which in turn produced small dots, varying in size, on a regular grid [19]. Removing the need to expensively etch every photograph by hand, it caused a revolution in printing.

2.2 Halftone methods classification

Halftone methods can be classified by various criteria [19], [12]. For some methods, however, certain criteria are not applicable. For each category examples are given.

AM vs. FM vs. hybrid methods In amplitude modulated (AM) halftone methods (fig. 2.1a) the intensity is simulated by varying dot size, while the distance between dots (more precisely between dot centers) is maintained constant. Conversely, frequency modulated (FM) halftone methods (fig. 2.1d) maintain dot size and instead vary distance between dots. Hybrid methods, as their name imply, vary both parameters (fig. 2.1b).



(a) Spot function thresholding (b) Adaptive SFC clustering

(c) Recursive tesselation ordered dither (d) Randomized error diffusion

Figure 2.1: Halftone methods classification

Clustered-dot vs. dispersed-dot methods Clustered dot methods (fig. 2.1a, 2.1b) connect pixels into compact groups, while dispersed dot (fig. 2.1c, 2.1d) methods try to spread pixels as uniformly as possible.

Periodic vs. aperiodic methods Periodic methods (fig. 2.1a, 2.1c) create repeated patterns, in contrast with aperiodic ones (fig. 2.1b, 2.1d). Periodic patterns may cause visual artifacts such as moiré, which degrade the overall look of the halftoned image.

Deterministic vs. stochastic methods Deterministic methods (fig. 2.1c) can be repeated anytime with the same and predictable results. Stochastic methods (fig. 2.3a) use some randomness and give slightly different results each time.

Point vs. area operations Halftoning methods may differ in how they operate on image pixels. Point operations (fig. 2.1c) treat each pixel separately and the result does not depend on the order of processing. Although such methods are simple and efficient, they may give suboptimal results. On the other hand, in area operations (fig. 2.1d) processing one pixel influences the other pixels. This kind of methods needs more memory and computational power, but can give far better results.

2.3 Output device considerations

Output devices work on different principles and differ slightly in how they reproduce digital images. This must be taken into consideration when choosing an appropriate type of halftoning method.

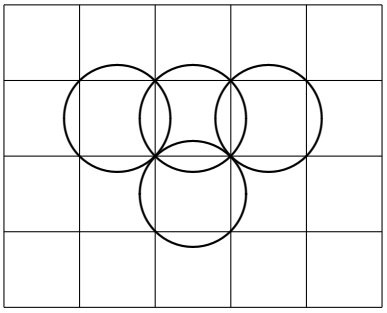


Figure 2.2: Dot gain

Dot gain Pixels in digital images are treated as tiny squares and so are reproduced by a majority of computer displays. However, printers are usually capable to print only small circular spots (fig. 2.2). In order to be able to print continuous black regions without any holes a spot must overlap the imaginary square. This is called *dot gain* [19]. As a result isolated black pixels get bigger and isolated white pixels may vanish completely. A separated spot contribute to the overall average image intensity more than a spot inside a cluster. Thus, dispersed-dot methods are more vulnerable to dot gain.

There are more causes of dot gain. Ink spreads due to its low viscosity and pressure from the printer cylinder making the spot bigger (*physical dot gain*). Also photons hitting the paper near the outer border of the spot may be blocked by it and the spot seems even bigger (*optical dot gain*).

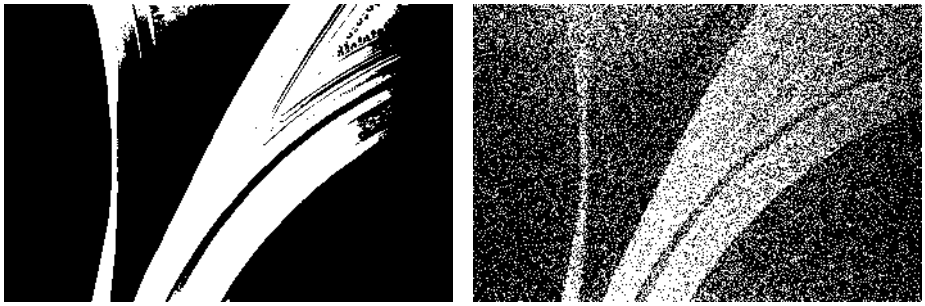
There exist several models of dot gain. It can be corrected by modifying the image tone curve prior halftoning, or its correcting could be incorporated into the halftone method itself. A dot gain correction curve can be empirically acquired for a particular printer or approximated using a model, eg. a gamma curve [37].

Isolated pixels There is a difference between the ability to reproduce isolated pixels on the target device. Computer displays are generally capable of that, as well as inkjet printers, which might have problems with isolated white pixels due to dot gain. On some high-resolution laser printers, however, isolated black pixels often disappear from paper due to ink smearing. Clustered-dot methods, though sacrificing some spatial resolution for reducing the number of isolated pixels, are suitable to solve this issue. [19]

2.4 Classic digital halftoning methods

When applicable each method's properties are expressed in terms of the classification above.

Thresholding Probably the simplest method of quantization a greyscale image to a bi-level one is *thresholding*. Every pixel with lower intensity than threshold t is rendered black and vice versa [19]. Mathematically expressed:



(a) Threshold at 50% grey (b) White noise dither

$$O(x,y) = \begin{cases} 1 & \text{if } I(x,y) \geq t \\ 0 & \text{otherwise} \end{cases} \qquad \text{or} \qquad O(x,y) = \text{trunc}(I(x,y) + 0.5)$$

Generally, threshold value $t = 0.5$ is used (resulting in a *uniform quantization*), but different values (mean or median intensity) can yield slightly better results, depending on the image histogram, but still far inferior to any of the following methods. In spite of that, the idea of varying the quantization threshold appears in many other halftoning methods.

Properties: aperiodic, deterministic, point-wise.

Random noise dither The problem with simple thresholding is that details in shadows and highlights get lost, and banding artifacts (also known as false contours) are introduced. One solution is to randomly perturb the threshold for each pixel, or in other words to add some random noise [35]:

$$t_p = 0.5 + a \cdot p, \qquad p \in U(-0.5, 0.5), a \in [0,1] \qquad \text{or}$$
$$t_p = t + a \cdot (1 - 2 \cdot |t - 0.5|) \cdot p, \qquad p \in U(-0.5, 0.5), a \in [0,1], t \in [0,1]$$

Explanation: t_p is the perturbed threshold, p the random perturbation with uniform distrubution U (and the noise is called white noise), and a is a parameter controling the amount of perturbation. If we allow threshold t to be variable the amount of perturbation will have to be limited, so that the perturbed threshold remains in the [0, 1] interval without clipping. The random perturbation can have a different distribution, eg. the Gauss distribution.

In general, the technique of varying quantization threshold for each pixel is called *dither* from the Old English word *didderen* meaning 'to shake'. Its usage is widespread in the area of digital signal processing, eg. in digital audio mastering [28].

Properties: dispersed-dot, aperiodic, stochastic, point-wise.

Matrix thresholding Another option is to vary the threshold in a deterministic and periodic manner (it is also called *ordered dither*). For that purpose matrices

repeatedly tiling the image plane are useful [19]. Let a $m \times n$ matrix T with elements $T(i,j) \in [0.0, 1.0]$ be the *(normalized) threshold matrix*, then the threshold value $t(x,y)$ for pixel $I(x,y)$ is obtained as follows:

$$t(x,y) = T(x \bmod m, y \bmod n)$$

Such a threshold matrix can represent at most $m \cdot n + 1$ different intensity tones. If the quantization is uniform, ie. the quantization thresholds divide the whole intensity range into equally sized steps, the threshold matrix can be represented in an *incremental* form as matrix T_i, containing all elements in set $\{1, 2, \ldots, mn\}$, for which holds: $T = (mn+1)^{-1} \cdot T_i$. This could be further generalized, as another matrix could contain $k \in \mathbb{N}$ copies of the original matrix (possibly arranged in a different order). Then the relation would be: $T = (k \cdot (mn+1))^{-1} \cdot T_i$.

Properties: deterministic[1], point-wise, others depend on specific matrix type.

Recursive tesselation A popular family of dispersed-dot threshold matrices is created by the *recursive tesselation* process and is refered to as *Bayer matrices* [3] (fig. 2.1c). The idea is to make dot patterns for each quantization step as homogenous as possible, so that with each following step new dots are inserted into the most 'void' places. A Bayer matrix B_m of magnitude m and size $2^m \times 2^n$ is recursively defined (in incremental form) as follows:

$$B_0 = \begin{pmatrix} 1 \end{pmatrix}, \quad B_1 = \begin{pmatrix} 1 & 3 \\ 4 & 2 \end{pmatrix},$$

$$B_{2 \cdot m} = \begin{pmatrix} 4 \cdot B_m & 4 \cdot B_m + 3 \cdot U_m \\ 4 \cdot B_m + 4 \cdot U_m & 4 \cdot B_m + 2 \cdot U_m \end{pmatrix},$$

where U_m is a unit matrix of size $m \times m$ [24].

Properties: FM, dispersed-dot, periodic, deterministic, point-wise.

Digital screening

Traditional analog screening can be classified as a clustered-dot AM method. The growth of screen dots can be digitally reproduced in form of incremental threshold matrices, where the individual increments tightly adjoin, thus making a cluster. Such matrices could be coded by hand or generated using a spot function [24].

Spot-functions *Spot functions* (fig. 2.1a) analytically define screening dot growth depending on pixel intensity as continuous periodic 2-D functions with values in range (0.0-1.0) [25]. They can be parametrized by screen angle and period (distance between dot centers).

[1] The process of generating a matrix could be stochastic, however.

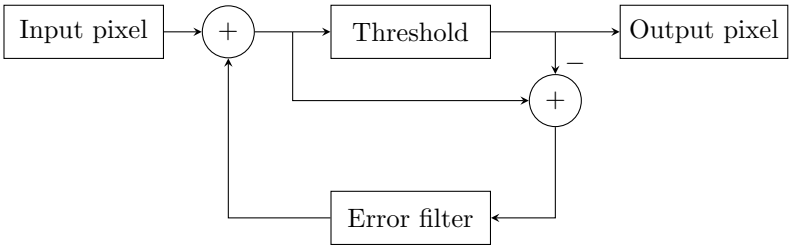


Figure 2.3: Error diffusion filter

$$\frac{1}{16}\cdot\begin{pmatrix} & \bullet & 7 \\ 3 & 5 & 1 \end{pmatrix}, \quad \frac{1}{48}\cdot\begin{pmatrix} & & \bullet & 7 & 5 \\ 3 & 5 & 7 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \end{pmatrix}$$

Figure 2.4: Example of error matrices: Floyd-Steiberg and Jarvis-Judice-Ninke. The bullet (•) denotes the source pixel position.

Patterning This simple method just replaces each input pixel with a pattern of several output pixels, thus multiplying the image dimensions [7]. An equal result can be obtained using more common methods in two steps: first rescale the input image with integer multipliers m_x, m_y and nearest-neighbour interpolation, then perform tileable matrix halftoning with a matrix of size $m_x \times m_y$.

Properties: AM, clustered-dot, periodic, deterministic, point-wise.

Error-diffusion

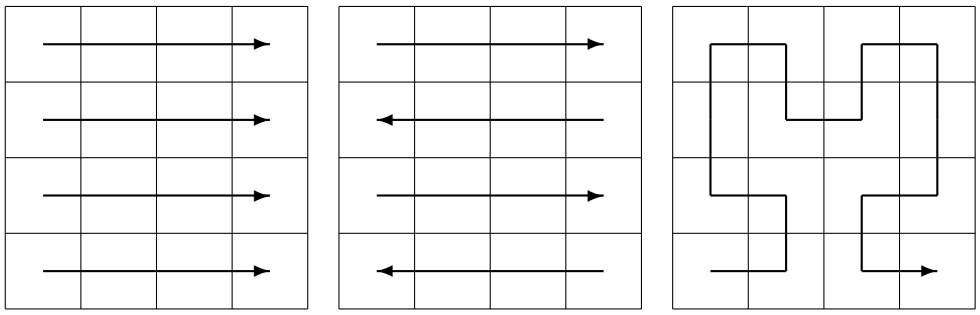
Every quantization process introduces a *quantization error* – a difference between the original and quantized value: $O(x, y) - I(x, y)$. The idea is to diffuse the error from already quantized pixels to their neighbourhood that will be processed later [8].

Most commonly used method is to utilize an *error distribution matrix*. The value and position of its elements indicate what portion of the error will be added to which pixel in the neighbourhood of the source pixel. In fact, this could be dually viewed as varying threshold for further pixels, while maintaining their value.

For meaningful results the sum of all coefficients have to be 1.0, otherwise the image quality suffers. For sums in interval $(-1.0, 1.0)$ highlight and shadow details get lost, for sums in $(1.0, +\infty)$ resp. $(-\infty, -1.0)$ the quantization error diverges, resulting in progressive black resp. white clipping.

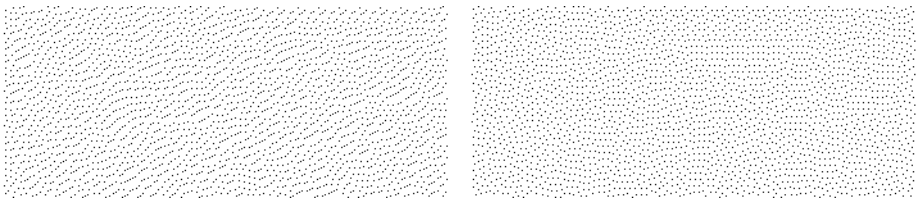
For a long time there has been an intensive research in looking for good error matrices. Most popularity gained a simple matrix by Floyd and Steinberg [8]. However, other successful matrices were discovered later by Jarvis-Judice-Ninke [15], Stucki [31], Sierra [30] or Burkes [6]. In general, larger matrices can distribute the error to wider area, but require more computational power.

Properties: FM, dispersed-dot, deterministic, neighbourhood.



(a) Scanline (b) Serpentine (c) Hilbert curve

Figure 2.5: Scanning orders



(a) Scanline scanning (b) Serpentine scanning

Figure 2.6: Scanning orders (Floyd-Steinberg error diffusion)

Serpentine scanning Some traditional error-diffusion methods may suffer from worm-like artifacts – clusters of pixels making small chains (fig. 2.6a). One solution to this issue is to change the order in which pixels are processed, from so-called scanline (fig. 2.5a) to serpentine scanning [32] (fig. 2.5b). Pixels on consecutive lines are not scanned in one direction, as usually, but the direction for even and odd lines differs, making a zig-zag movement. More advanced methods of image scanning are discussed in the next section.

Error buffer Error-diffusion filters perform a neighbourhood-wide operation on image pixels. As an error filter processes the pixels, it distributes (adds) a fractional part of the error from the pixel which has been just processed to the other pixels. Until those pixel are processed too, the accumulated error has to be stored somewhere. The original image is not the best place for two reasons:

1. It might not be the same as the output image (or be writable at all).
2. It might not be able to represent the error in such a precision.

Thus, a separate buffer to store an accumulated error for pixels waiting to be processed becomes useful.

2.5 Modern digital halftoning methods

Space-filling curves

Fractal 1-D curves that fill the entire 2-D plane, such as Hilbert curve or Peano curve, are called *space-filling curves* (SFC). They are constructed with a set of recursive rules as successive approximations[2] going towards a limit curve. Those self-similar approximations have interesting properties for image scanning. They visit each point exactly once on a continuous path which never intersects itself (fig. 2.5c). Moreover, Hilbert curve never keeps one direction for more than two steps thus effectively avoiding directional artifacts [37].

Error diffusion along a space-filling curve Riemersma [29] took the intuitive approach of thresholding along the Hilbert curve with error-diffusion, but fine-tuned the process of generating the error matrix (or more precisely error vector) coefficients. In his method error coefficients descrease exponentially depending on the distance from the source pixel.

Properties: FM, dispersed-dot, aperiodic, deterministic, neigbourhood.

Adaptive clustering along a space-filling curve A different approach was taken by Velho and Gomes [37]. The image is scanned along a space-filling curve, preferably the Hilbert curve, in blocks of consecutive pixels called cells. Average intensity of each cell is computed and a proportional number of cell pixels is turned black (fig. 2.1b). With varying cell size there is a trade-off between the ability to represent more grey levels (tonal resolution) and finer details (spatial resolution)[3] In addition, a per-cell error-diffusion greatly improves the resulting image quality.

An enhancement to this method is to control the size of each cell adaptively, according to the amount of local detail. This can be measured by a direction derivative in the SFC direction. One way to obtain it is from gradient at current pixel:

$$\frac{\partial\, I(x,y)}{\partial\, \vec{u}(x,y)} = \Big\langle \operatorname{grad} I(x,), \vec{u}(x,y) \Big\rangle \quad \text{where} \quad \operatorname{grad} I(x,y) = \left(\frac{\partial\, I(x,y)}{\partial\, x}, \frac{\partial\, I(x,y)}{\partial\, y} \right)$$

and where $\vec{u}(x,y)$ is a unit vector of SFC direction from previous to current pixel [38]. Gradient can be approximated using 2-D Laplace filter. Wong [41] also suggested that using 1-D Laplacian along the SFC direction gives comparable results with less computation efforts than 2-D Laplacian. Maximum allowed cell size for current pixel is computed as a function of the direction derivative. Velho and Gomes suggested an exponential mapping [38], which can be mathematically interpreted as:

$$c = 2^{(1-|\frac{\partial\, I(x,y)}{\partial\, \vec{u}(x,y)}|)\cdot \log_2 c_{max}}$$

where c is the new cell size limit and c_{max} is the global maximum cell size.

[2] Even those approximations are often refered to as space-filling curves.

[3] In larger areas with smooth gradients bigger cells representing more grey levels will prevent banding, while areas of high detail will benefit from smaller cells.

In order to better preserve the image details, each generated cluster of dots can be positioned inside the cell. Velho and Gomes suggested to move the cluster center to the darkest cell dot [38], while Wong and Hsu [41] move the cluster through the cell as a sliding window to find a place with minimum sum of intensities (maximum total grey level) and even break the cluster if needed.

Properties: hybrid AM&FM, clustered-dot, aperiodic, deterministic, neigbourhood.

Blue-noise and its metrics

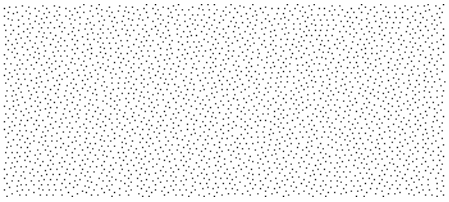
Since the earliest days halftoning methods have been compared according to how 'good' they look. People argeed that periodic or directional artifacts, or too much noise is not good. However, some exact metrics were needed for automatic measurement. To examine periodic patterns frequency-domain Fourier analysis proved to be useful. Ulichney [32] proposed several metrics based on the *power spectrum density estimation*. A sample image filled with a fixed grey level is transformed with 2-D Discrete Fourier Fransform (DFT), squared and normalized. Several resulting so-called periodograms are then averaged to get the power spectrum density estimation.

To get a one-dimensional metric of the quality of halftoned images, two statistics are derived. *Radially averaged power spectrum* (RAPS) shows the frequency spectrum of distances between dots averaged over all directions. For example, RAPS of a white noise dither exhibits a uniform spectrum similar to the spectrum of white light. In contrast, a spectrum of visually pleasant halftones lacks low frequencies, and similarly to the mainly high-frequency spectrum of blue light, the term *blue noise* was coined. The principle is that high-frequency quantization noise can be less percieved by human eye than low-frequency noise. Another metric called *anisotropy* measures the amount of directional artifacts present in a sample halftoned image.

Related to blue-noise is *green noise* with mainly mid-frequency content. While high-frequencies of blue noise correspond to dispersed-dot halftones, green noise is related to hybrid AM-FM halftones, where some clustering happens [18].

Error-diffusion approaches Several error diffusion techniques were suggested by Ulichney [33] as quite good blue-noise generators. One approach is to randomize error matrix coefficients on fixed positions, or even randomize their count. Another approach is to take a traditional error matrix and enhance the method by randomly perturbing the matrix coefficients and/or perturbing the quantization threshold. The amount of perturbation need not to be high at all to get rid of unwanted artifacts and at the same time not to make the image too noisy. These blue-noise metrics were employed by Ostromoukhov [26] to find optimal coefficients of an error matrix with given coefficient positions, for each intensity (fig. 2.7a). This approach, in cooperation with finding optimal threshold perturbation amounts for each intensity by Zhou and Fang [42], gives nice results.

Properties: FM, dispersed-dot, aperiodic, stochastic, neighbourhood-wise.



(a) Optimal weights by Ostromoukhov

Figure 2.7: Blue-noise pattern in highlights (95% grey)

Void-and-Cluster Void-and-Cluster by Ulichney [34] is a method of generating large thresholding matrices with good blue-noise properties. The advantage is that once a matrix is generated its further usage is very computationally inexpensive. The process of generating the matrix is divided in two phases. There is an incremental threshold matrix to be filled and a working binary matrix of the same size. At first, an inital seed of about 10% pixels in the working matrix is turned on. Their position can be chosen randomly or with another method, such as error-diffusion. Then clusters of pixels are broken to fill voids, finding the order of the thresholding matrix elements. The important thing is the module to find clusters and voids – Voronoi tesselation [1] or Gaussian convolution [34] are considered suitable for that purpose.

Properties: FM, dispersed-dot, periodic, deterministic, point-wise.

Blue noise mask Blue Noise Mask by Mitsa and Parker [22] is also a way to pre-generate large threshold matrices with good blue noise properties. However, the process of generation is different from Void-and-Cluster method. The mask is created iteratively – for each grey level a pattern of a uniform number of binary dots is turned on. Dots for each pattern are taken from the set of dots still turned off, acording to a metric, which is constructed roughly as follows: the previous dot pattern is trasformed via 2-D Fourier transform to frequency-domain, filtred with a blue noise filter, and transformed back. Then the difference between the original dot pattern and frequency-filtered dot pattern acts as a metric for choosing the dots for the next dot pattern. The final blue noise mask's elements indicate in which iteration the dot was turned on.

Properties: FM, dispersed-dot, periodic, deterministic, point-wise.

Model-based methods

There exist halftoning methods which directly exploit mathematical models of the human visual system (HSV) [23] or specific printer technologies [27]. Upon those models some metrics are built to measure the quality of halftoned images. A typical layout of model-based algorithms is that the whole image is processed in multiple iterations – in each one only such changes are done that comply to the metric the most. Another common approach is to integrate a metric inside a feedback loop into a simpler algorithm, such as error diffusion.

Direct Binary Search Direct Binary Search [20] tries to minimize a metric of error between the original and the halftoned image. It starts with an initial image, halftoned by another method, and iteratively alters it, in order to diminish the error. A HVS model can be employed to construct a suitable error metric. Alterations include toggling a binary pixel value or swapping two neighbouring pixels. In principle, this method is more computationally intensive than other simpler methods, however, some efficient ways of implementation exist [4]. The profit is in very high quality halftone.

Properties: FM, dispersed-dot, aperiodic, deterministic, area-wide.

Other methods

Omnidirectional error diffusion A disadvanage of classical error diffusion is that the image pixels are scanned in lines and the error is distributed to unprocessed pixels in one global direction. In order to make the distribution more isotropic, two methods sharing a similar concept were developed: Dot diffusion [17] and omnidirectional diffusion via Linear pixel shuffling [2]. The key is to scan the image pixels isolatedly and diffuse the quantization error to their yet unprocesses neighbours. Dot diffusion uses some tileable matrices for scanning order definition, while Linear pixel shuffling is a pseudo-random technique.

Properties: FM, dispersed-dot, periodic, deterministic, area-wide.

Artistic methods

Apart from methods where the main goal is perfect rendition of the original image, there exist some artistic methods. In contrast, the primary concern there is an appealing look, especially in details. Artistic halftoning lies halfway between ordinary halftoning, and artistic image processing effects, such as digital engraving or hatching.

The most common approach is to make one or more tileable thresholding matrices which produce small ornaments, letters or textures as suggested by Hausner [13], or Veryovka and Buchanan [39]. Ostromoukhov [25] developed an interesting method of generating ornamental matrices by blending and rasterizing multiple vector shapes that fit together. Also the threshold matrix can be as big as the halftoned image to enable spatial transformations thereof.

Image-based thresholding Veryovka and Buchanan [39] use an ordinary image to form a tileable threshold matrix (fig. 2.8a). The trick is, however, to modify the local contrast, in order to evenly cover the whole intensity range. For that purpose a kind of *adaptive histogram equalization* algorithm is used. In addition, error-diffusion is applied.

Another related artistic approach can be seen in popular ASCII-art, where the smallest addressable output image elements are not single pixels but rather a subset of pixel blocks representing available text characters.

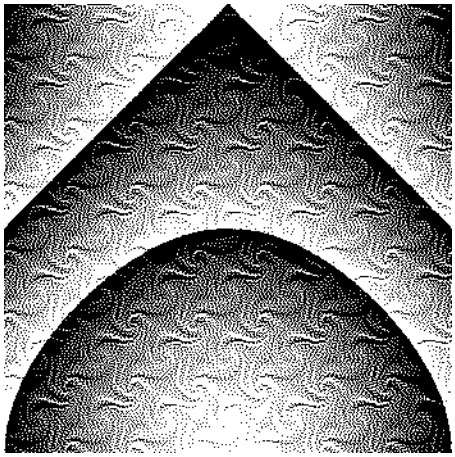


Figure 2.8: Image-based thresholding with Jarvis-Judice-Ninke error filter

In case graphic designers wish to use a clustered dot halftone with big dots as a stylish means for computer display as a target device, the dot edges look better when smoothed (antialiased). There are at least two ways how to achieve that. In supersampling the image is first upsampled by an integer factor $r > 1$ and a good interpolation method (eg. bicubic interpolation), then halftoned as usual, and in the end downsampled back by factor r^{-1}. The output image is no longer black-and-white, but the edges are smooth and smaller details appear.

A smoothing method proposed in this paper is to blur the halftoned image with Gaussian blur of radius smaller than screen dots period and then apply Levels image editor command to compress the histogram to a tight region around middle grey. The edges get even smoother without the expense for halftoning $r^2\times$ more pixels. Smallish dots are not perfectly preserved, though.

2.6 Related problems

A brief look follows just to be aware of what kind of problems related to halftoning also exist.

Black-and-white halftoning can be generalized to *multitone greyscale halftoning*, when the output device supports more grey levels [35], or even to *color halftoning*, when the quantization is generalized to multidimensional color space [14].

Getting an approximation of the original image from the halftoned one is a problem being solved by *inverse halftoning* [16]. When it is needed to render a halftoned image many times per second (eg. in an interactive game) *real-time halftoning* methods take place [9]. For further speeding-up the rendering on multiprocessing platforms *parallel halftoning* searches for ways how to limit dependence between area operations [21].

Chapter 3

Implementation

3.1 Initial design thoughts

The original assigment was to implement a library, with an optional graphical user interface (GUI), as a collection of basic and advanced halftoning algorithms. During the study of various existing halftoning methods I soon realized that they share similar structures and follow some common patterns. Thus, making the library modular and extensible turned out not only possible, but also highly desirable. This resonated with the instinctive intention to abstract and modularize what is possible.

The task was to extract parts of various algorithms into modules, analyze their function in the system, interaction with other modules and the data flow, and design a concrete object-oriented model. This changed the way how to implement the algoritms, as they would not be defined as monoliths, but rather as living structures of interconnected parts, some of them parametrized.

There were two opposite forces influencing the design: to be abstract enough in order to possibly support any method, but not to be abstract too much, so that it would be barely possible to implement it in context of existing frameworks. Which methods and what level of abstraction to support had to be decided. One of the limiting constraints was also the amount of time to finish the whole project. On the other hand, during the design process I always had to bear in mind to let the door open for different methods, not to make it very hard to intergrate them in future.

The tasks ordered by descending priority were:

- to make a working infrastructure
- to design and implement a reasonable amount of modules, which cover most of the halftone method types
- to enable storing of algorithm configurations and to prepare some presets
- to make a GUI for configuring the modules interactively

3.2 Programming language and environment

The first thing to decide was the programming environment. As a modular architecture was to be designed, using the object-oriented paradigm was essential. From my previous experience I prefered a higher-level programming language for such a task. The initial set of potential languages included C#, Java, C++ and C, ordered by a mix of preference and knowledge. C was soon considered too low-level and it lacked object-oriented features. C# were prefered over Java and C++ for its modern features, such as built-in functional programming paradigm, just to name one.

Targetting the project as a GIMP plug-in

As I started to think about the project, there were two possibilities how to use the library. Either to make a standalone application, or to integrate it in an existing framework (possibly in a form of an image editor plug-in). This is always a trade-off between independence and not reinventing wheel.

Plug-in variant meant being tied to an external program or library, but allowed saving a lot of work on creating a basic infrastructre, which has been implemented many times in other programs, and which is not the goal of this project. Only a mature image editor can also offer a great amount of advanced, ready-to-use features that might prove useful someday. However, the greatest asset of this variant was that it enables the project to integrate into user's workflow seamlessly. For those reasons I decided to incorporate the Halftone Laboratory project into an existing image editor as a plug-in with a door open to make a standalone interface in future.

The potential host application or library had to be extensible, or in other words provide a way to create plug-ins. Preferably, a reasonable programming language should have been supported. It was desirable for the host application to be widely used, cross-platform and free of charge. Several applications or libraries were considered.

Adobe Photoshop seemed to be perfect match at the first sight as it is widespread, but the application itself is quite expensive, and even its plug-in SDK appeared to be paid. Moreover, it is portable with a great difficulty.

Another candidate was GIMP image editor [10]. It is not as widely used, but it is stable, free and offers a well-documented plug-in API. Although the native API is provided only in C, C++, Scheme and Python, lots of supporting projects exist around, since GIMP is open-source. Among them plug-in API bindings for many other languages, such as GIMP# [11], a binding for .NET languages including C#. After a practical examination GIMP# was considered stable and capable enough to provide a solid ground for the Halftone Laboratory project.

Meanwhile, ImageMagick library was surveyed and soon rejected, as it is mainly suitable for batch processing, it provides less features and no interactive GUI. At the time, a better alternative had already been found in GIMP.

The project was developed in Microsoft Visual Studio C# 2008 using GIMP#, Microsoft .NET 3.5 and Gtk# libraries.

Resulting overall architecture

Finally, the project has been divided into three parts:

- the halftoning library itself
- GTK# graphical user interface
- GIMP# plug-in

The library contains the core code and is completely independent on the GUI and only little dependent on GIMP#. The GIMP#-related code is separated on a well-defined place. Also the GUI is completely independent on GIMP#. The plug-in part acts only as a simple thin layer for integrating the library and GUI with GIMP.

3.3 Modular halftoning library architecture and implementation notes

Life-cycle of an image filter The typical way how an image filter (such as a halftone algorithm) operates can be divided in two phases: During the *design-time* the filter parameters are being set (interactively or from a saved configuration). Then when prepared the filter can be started to process an image, which is further refered as *run-time*.

Module In this paper, a separate part of a halftone algorithm is refered to as a *module* if it satisfies certain properties: it is intended to be stored as a part of a configuration, it offers a clean interface allowing to change the implementation, it might optionally offer some parameters to be set, in particular, other modules can be plugged in as submodules.

Besides persistent parameters a module might contain some temporary parts, which are not stored and need to be always re-computed before run-time. Moreover, for its proper function a module might need some run-time information, such as image dimensions or scanning order. To meet all those demands all concerned modules are initialized just before run-time with an instance of `Image.ImageRunInfo` class. For more information on `Module` abstract base class usage see page 27. Also note that presented module names correspond with classes in the program.

Halftone methods

HalftoneMethod The basic abstraction of a halftone algorithm is the `HalftoneMethod`. It transforms a greyscale input image to a black-and-white output image of the same size. How the transformation is accomplished is left to its various implementations. Currently, there are two main halftone method types: `PointHalftoneMethod` represents point-wise methods where each pixel is read, processed and written separately, and `CellHalftoneMethod` where pixels are processed in continuous adjacent groups. More types could be added in future.

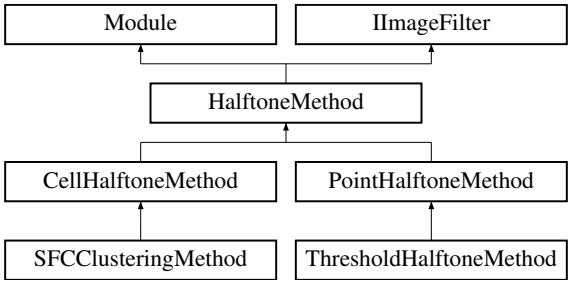


Figure 3.1: Halftone methods

HalftoneAlgorithm As some pre-processing and post-processing steps apart from the core halftoning can take place, it was decided to make a generalized wrapper over the `HalftoneMethod`. It is restricted neither to maintain the image size, nor to output only a black-and-white image. Common pre- and post-processing steps may include scaling, sharpening or smoothing.

For some halftoning methods scaling the image up is an essential part of the method (eg. for patterning), for others it is optional but appropriate (SFC clustering) One can imagine graphic designers wishing big smooth AM halftone dots as a graphic effect. Smoothing can be acomplished by supersampling (upscaling, halftoning and downscaling back to the original size) or by a custom smoothing filter.

For pre- and post-processing filters there is an abstract interface, so they can be implemented differently. Currently, host application (GIMP) facilities are utilized if available.

ThresholdHalftoneMethod Threshold halftone method is the base for a majority of classical halftoning methods. It simply processes the image along a `ScanningOrder` and quantizes each pixel separately with threshold supplied by a `ThresholdFilter`. Quantization error is optionally diffused by an `ErrorFilter`.

SFCClusteringMethod In this module is implemented SFC clustering, as described earlier on page 14. Cluster positioning and adaptive varying of the cell size can be enabled or disabled. Error-diffusion between cells is managed by a plugable `VectorErrorFilter`. To vary the cell size according to local detail the directional derivative along the SFC is approximated. It is done in a simple way – using the difference between current and last pixel:

$$\frac{\partial\, I(x,y)}{\partial\, \vec{u}(x,y)} \approx I(x,y) - I(x',y')$$

where $\vec{u}(x,y)$ is a unit vector of SFC direction from previous to current pixel and x' resp. y' are previous pixel coordinates. When knowing the derivative, the cell size is adjusted according to the mentioned mapping (see page 14) within the limits of minimum and maximum cell size (given as module parameters).

Threshold filters

ThresholdFilter A threshold filter computes the threshold for bi-level intensity quantization depending on current pixel coordinates and/or its intensity. Its `quantize(intensity, x, y)` function acts as a comparer, where the threshold is supplied by the `threshold()` function implemented in child classes. Intensities lower than threshold are turned to black, intensities equal or higher than threshold are turned to white.

Note: In future this could be generalized to perform multi-level quantization.

MatrixThresholdFilter Matrix threshold filter stores threshold values in a matrix repeatedly tiling the plane (ie. the image). `ThresholdMatrix` module is used for that purpose. Threshold matrix elements need to be scaled to the pixel intensity range, but they can be also defined as incremental (see page 10).

The threshold matrix implementation in fact holds two matrices: a definition one (which can be incremental or already normalized) and a temporary one which is always scaled. This is to enable users to repeatedly edit the threshold matrix as incremental without any loss of information and without a run-time performance penalty. A similar concept is applied to `ErrorMatrix` (see page 25). Actually, `ThresholdMatrix` and `ErrorMatrix` modules share code in their common base class, `Matrix`. In the `ThresholdMatrix.Samples` inner class there is a procedure for generating Bayer recursive tesselation matrices (see page 11).

DynamicMatrixThresholdFilter Dynamic matrix threshold filter can use different matrices for different intensities and randomly perturb threshold values in addition. The idea is to split the discrete pixel intensity range into several subranges, each with its own matrix and noise amplitude.

The definition-cache concept seen before is utilized as well. The definition is held by `DynamicMatrixTable` module, which is an easily editable sorted list of custom intensity range records (start intensity, matrix, noise amplitude). Just before runtime the list is converted to an array of records for faster record retrieval. The code is separated into a generic `DynamicMatrixTable` module as it is also used by the `DynamicMatrixErrorFilter` module (see page 25). The difference is just in the table records defined as inner classes.

Noise application can be enabled or disabled for the filter as whole.

SpotFunctionThresholdFilter Threshold values in Spot function threshold filter are computed directly using a spot function (see page 11). Technically, spot functions are defined in `SpotFunction` module as anonymous lambda functions. First a prototype lambda function is defined. It is then parametrized by screen angle and period creating the final lambda function, which takes coordinate parameters.

ImageThresholdFilter Spot functions could be used to generate threshold matrices to save computation. However, to make those matrices tile seamlessly not

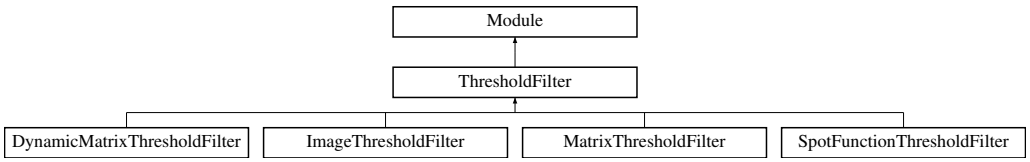


Figure 3.2: Threshold filters

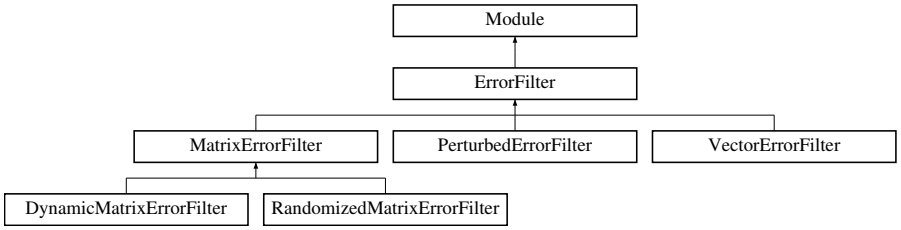


Figure 3.3: Error diffusion filters

every screen angle can be used [24]. To support arbitrary screen angles another approach has been utilized: instead of generating a small tileable matrix a matrix as big as the halftoned image is generated. It is stored a an `Image`[1] for efficiency. Although the memory demands are quite big, new possibilities come. One can easily modify the whole matrix using standard image processing operations – imagine applying one or more of the myriad of GIMP filters. This is when cooperation with a mature graphics library come in useful. Also, this way the screen plane can be transformed in a similar way to [25].

For defining how to generate an `Image` the `ImageThresholdFilter` uses an `ImageGenerator` module which contains an initial `SpotFunction` and a list of effects applied on top of it in form of `IImageFilter`s with lambda functions calling other GIMP plug-ins.

Error-diffusion filters

ErrorFilter Error-diffusion filters diffuse the quantization error to the neighbourhood of the current pixel. Although they differ in exact behaviour how the error is distributed, they generally use a kind of internal buffer which holds the current position. The run-time interace is very simple: `getError()` function gives the value of the accumulated error for current pixel, `setError(error, intensity)` diffuses the error and finally `moveNext()` advances to the next position.

Knowing current pixel intensity when setting the error is not needed in general, yet some error filters can utilize it (eg. `DynamicMatrixErrorFilter`). Polluting the interface with an additional parameter which is seldom used appeared to be less evil than making a separate interface for dynamic error filters and duplicating code elsewhere.

[1] See page 26.

MatrixErrorFilter Matrix error filter is an error-diffusion filter where the distribution of the quantization error to current pixel neighbourhood is controlled by a single matrix. `ErrorMatrix` coefficients indicate what portion of the error goes to which relative position (see fig. 2.4). In the `ErrorMatrix.Samples` inner class there is a lot of pre-defined matrices, as well as a procedure for generating Riemersma error vectors (see page 14).

DynamicMatrixErrorFilter Dynamic matrix error filter works in a similar way to `DynamicMatrixThresholdFilter`, except that no perturbation is added[2]. Which one of multiple matrices to use depends on pixel intensity.

RandomizedMatrixErrorFilter Randomized matrix error-diffusion filter can generate matrix coefficients randomly. It can use an existing matrix as a template for positions of individual coefficients and the source pixel. The other way is to randomize even the number of coefficients up to the the original matrix capacity.

PerturbedErrorFilter Perturbed error filter acts as a wrapper over a `MatrixErrorFilter`. It gets the original matrix and adds random perturbations to its coefficients according to [33]. The sum of all perturbations is ensured to be zero. The overall amount of perturbation can be controlled.

Coefficients in the original matrix are sorted according to their value and grouped in pairs. A random perturbation of varying magnitude is added to one coefficient and subtracted from the other. Perturbation magnitude depends on the lesser of the two coeficients. If there is an odd number of coefficients the last perturbation is divided into a group of three coefficients.

VectorErrorFilter Error filter with a one-dimensional error matrix and buffer. This can be useful eg. for image scanning order based on a space-filling curve.

ErrorBuffer The reasons for using a separate error buffer are described on page 13. Its behavior depends on the image scanning order, so there are several types of error buffers. In fact, the buffer does not use coordinates from a `ScanningOrder` module but performs its own scanning. `ErrorBuffer` is not a module because it needn't be stored, it is only a run-time helper structure being not intended to be a part of any configuration.

MatrixErrorBuffer A two-dimensional error buffer for matrix error filters. It is as wide as the image and as high as the error matrix. Currently, only scanline and serpentine scanning is supported. For majority of halftone methods is this enough, but probably for methods such as omni-directional error-diffusion a buffer of size equal to the image might be useful.

[2]There are some additional constraints for error perturbations, see the `PerturbedErrorFilter`.

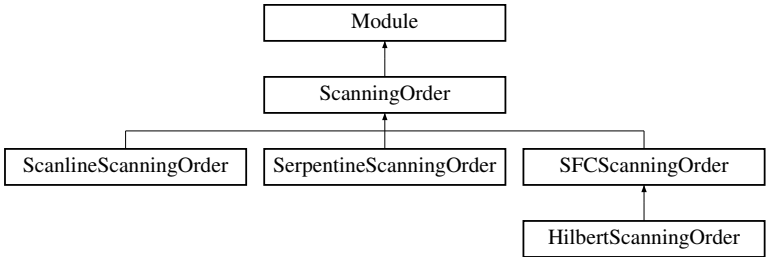


Figure 3.4: Image scanning orders

VectorErrorBuffer A one-dimensional error buffer for vector error filters. It is implemented as a simple cyclic buffer. Compared to using `MatrixErrorBuffer` as a special case it has a slightly different semantics, a simpler interface and should work more efficiently.

Image and its scanning

Image In order to be independent on a concrete image processing library an abstract image interface had to be created. Pixel-wise iteration along a scanning order and direct access to pixels were needed. Currently, there is only one implemenation of the `Image` abstract base class, `GSImage`, built upon GIMP# facilities, but it is easy to implement the `Image` on top of other back-ends.

IImageFilter Image filter interface is a general interface for modules that modify the `Image` in some way. `HalftoneAlgorithm`, `HalftoneMethod` and a lot of `ImageGenerator` effects implement it.

ScanningOrder This family of modules give an order in which image pixels are processed. Each image pixel have to be processed exactly once. It follows the Iterator design pattern, for each iteration it gives unique coordinates. Several implementations exist:

`ScanlineScanningOrder` follows a natural scanning order – pixels are passed line by line, each in the same direction, while `SerpentineScanningOrder` scans in zig-zag way, even and odd lines in the opposite directions. `SFCScanningOrder` is an abstract base for scanning orders based on space-filling curves. Hilbert curve is generated in `HilbertScanningOrder` module via a simple recursive algorithm [5].

Hilbert curve approximations are only defined for square images of size $2^n \times 2^n, n \in \mathbb{N}$. For images of different sizes (generally of rectangular shape) the least such a square which covers the image entirely must be found. Then skipping all coordinates outside the image rectangle is enough to meet the condition of passing each image pixel exactly once.

Configuration and presets – module serialization

As setting all the modules by hand can be quite complex, there is a great need to be able to save and load module configuration. The term *module configuration* represents the structure of how module objects are connected together and the values of their parameters. One use is to offer users some predefined configurations (presets) of various halftone algorithms, as well as configurations of separate modules (eg. commonly used error matrices). The other equally important usage is to enable users to store their own configurations and load them again. A following task was to persist the configuration to a permanent storage.

Low-level serialization and persistence infrastructure The first idea was to make a text configuration file with a custom format. The nice thing is this would be human readable, platform independent and inexpensive for storage. But a great drawback was that it would be hard to design and implement correctly, it wouldn't be type-safe, and worst it would draw a great deal of programming efforts away from the main subject of the project.

After a modest further research .NET's own serialization framework was found to perfectly suit the project's needs. Without reinventing wheel one can take living objects (even interconnected) and serialize them to several formats (XML, binary) that can be in turn saved to a file. XML would be perfect as it is human readable, but unfortunately due to some technical problems it didn't work well with hierarchies of abstract classes (that are heavily utilized throughout the whole project). Binary formatter worked well without any problem and so it was employed in the project.

High-level configuration manager On top of this serialization a configuration manager is built. Its purpose is to hold stored module objects and provide an interface to operate with them. One should be able to store a module into the manager, load it again, list all stored modules or search among them and delete a module. Each configuration should have a name and optional description – to be more user-friendly as well as to simply be able to address that configuration.

After some wild design thoughts the simplest solution appeared to convenient. All modules derive from the `Module` abstract base class and bear their names and descriptions inside. `ConfigManager` internally holds a list of such modules. It is then possible to query for all modules of a given type or address just a single module of given type and name. The whole list is held in memory and persisted to a file when needed. Saving the list can be performed automatically on each modification, or manually, eg. after inserting a bunch of modules in a batch. Modules are deep-copied when inserted into the `ConfigManager`.

ModuleAttribute It is useful to describe each module with a name and optional description. To statically attach such an information to a class .NET attributes (built-in or custom as in this case) are intended. Then in the GUI those texts can be retrieved via .NET reflection and shown to help users.

3.4 Graphical user interface

Finally, a GUI has been implemented. There were two design constraints: The library should be independent on the GUI and the GUI should be independent on the GIMP plug-in.

The general process of configuring a module via a GUI is as follows:

1. A widget for configuring a particular module type is created with a module object supplied (existing one or a default one).
2. Widget controls are set according to existing module contents.
3. The user makes some changes.
4. When configuring is finished the configured module is returned.

In case the widget is a modal dialog with an event fired on its closing, all the module changes can be applied at once. If this is not available the widget must provide its own event to signalize every change.

Naturally, almost every module needs its own specific widget. The issue was how to assign a widget type to a module type without touching the module code. Probably the simplest solution was to make a registry, `ModuleRegistry`, of those associations (currently coded by hand), which can be then queried. Another, probably stronger, reason for such a registry was to solve the following problem.

In the library there are several hierarchies of modules with a common abstract base class or with a more complicated structure. When a module incorporates a submodule of such an abstract type T, it is natural to provide a `ComboBox` widget with all suitable submodule types listed. A suitable submodule means a subclass of T (including T itself) which is instantiable (and then non-abstract). Those relations are detected via reflection.

So the `ModuleRegistry`, acessible via the Singleton design pattern, stores the list of available modules along with their submodules, dialogs (if there are any), and `ModuleAttribute` descriptions.

Another problem to solve was how to edit a matrix of numbers. There were three possible approaches: simply make a table of `SpinButton`s, use a `TreeView` with a `ListStore` model, or use a `GtkSheet` widget from the GtkExtra library[3]. The need was to be able to address each matrix element directly. A table of `SpinButton`s was the simplest solution, but had a severe performance issue – redrawing a table of even about 16×16 elements was quite slow, for bigger matrices it would be practically unusable. `TreeView` variant seemed to be alright, however, it turned out it is not possible to address each cell, provided there is a variable number of columns. `GtkSheet` would be perfect (addressable and fast), but currently no binding for .NET exists. As a result, a table of `SpinButton`s was used.

[3] `http://gtkextra.sourceforge.net/`

Chapter 4

Conclusion

Summary In the first chapter we have familiarized ourselves with digital halftoning and set the goals of the thesis. The next chapter described several classic halftoning algorithms, based on the priciple of thresholding and error diffusion, and classified them by various criteria. Several modern methods were introduced as well. Also, issues with output devices, to be solved by the halftoning algorithms, were discussed. The architecture of the Halftone Laboratory program and implementation details were presented in the third chapter.

Fulfilment of the goals In the presented work I have managed to build not only a collection of several algorithms, but an extensible framework being able to compose algorithms by assembling modules together. As many algorithms share similar structures, the code need not to be duplicated, it can rather be reused. To make the halftoning library easy to use a configurable a GUI was created and integrated into GIMP, a popular image editor, as a plug-in. Thus, even non-programmers may enjoy experimenting with various halftone methods.

Among other methods an SFC clustering method, as outlined by Velho and Gomes [38], as well as a blue-noise approach proposed by Ulichney in [33], have been implemented. Also two artistic approaches can be performed using HalftoneLab program[1].

Portability The program utilizes Gimp# library as its image handling back-end, but other libraries can be used without much effort, as there is an image abstraction layer. Although it was not tested thoroughly, there should be no explicit barriers against portability to different platforms. Thanks to the program is written in the .NET framework, potentially, it could be easily deployed on machines with either Microsoft .NET or Mono, a portable .NET framework implementation, installed – without recompilation. As for GIMP and GIMP#, they already run on many platforms.

[1] Image-based thresholding by Veryovka [39] and spot-function generated threshold image transformed in spatial domain [25] (see page 41 in User's guide)

Performance In choosing the technology to use, there was a trade-off between the ability to do a rapid higher-level object oriented development and the speed. I followed an old advice first to get working results and then measure and tune.

Although I managed to get a reasonable speed, it is not as high as native GIMP processing. Although some parts of the program were optimized only a little, observation showed that a performance bottleneck might lie already in GIMP# library. Even just reading and writing the image without any change was far slower than processing done in a native GIMP procedure written in C. So it might be useful to measure the performance with a different back-end library and try to tune the GIMP# library itself.

However, if the HalftoneLab should have been implemented in pure C, the development would take many times longer and there would certainly arise many memory management bugs.

Other implementations Halftoning algorithms with several decades of history were implemented uncountable times. They can be found in printer drivers, display drivers, graphic editors and on many other places. As their main focus is the utmost performance in production environment, they are implemented as fine-tuned monoliths. However, I haven't managed to find any single halftoning library modular to such an extent as HalftoneLab. Thus, I consider this project to be an original approach.

Further work During the creation of the HalftoneLab project I came across a lot of ideas how to improve it. Also, many interesting halftone methods were found, yet there was no more time to implement them in the scope of the bachelor thesis. Those future directions are outlined the the first appendix.

Bibliography

[1] Ancin, H., Bhattacharjya, A. K., and Shu, J. S. Improving void-and-cluster for better halftone uniformity. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* (Jan. 1998), G. B. Beretta and R. Eschbach, Eds., vol. 3300, pp. 321–329.

[2] Arney, J. S., Anderson, P. G., and Gunawan, S. Error diffusion and edge enhancement: Raster versus omni-directional processing. *Journal of Imaging Science & Technology 46*, 4 (2002), 359–364.

[3] Bayer, B. E. An optimum method for two-level rendition of continuous tone pictures. In *IEEE International Conference on Communications, Conference Records* (New York, NY, USA, 1973), IEEE, pp. 26–11–26–15.

[4] Bhatt, S., Harlim, J., Lepak, J., Ronkese, R., and Sabino, J. Direct binary search with adaptive search and swap. `http://www.ima.umn.edu/2004-2005/MM8.1-10.05/activities/Wu-Chai/halftone.pdf`, 2005.

[5] Breinholt, G., and Schierz, C. Algorithm 781: generating hilbert's space-filling curve by recursion. *ACM Trans. Math. Softw. 24*, 2 (1998), 184–189.

[6] Burkes, D. Presentation of the burkes error filter for use in preparing continuous-tone images for presentation on bi-level devices. LIB 15 (Publications), CIS Graphics Support Forum, 1988.

[7] Crocker, L. Digital halftoning. `http://www.efg2.com/Lab/Library/ImageProcessing/DHALF.TXT`.

[8] Floyd, R. W., and Steinberg, L. An adaptive algorithm for spatial grayscale. In *Proceedings of the Society for Information Display 17 (2)* (1976), pp. 75–77.

[9] Freudenberg, B., Masuch, M., and Strothotte, T. Real-time halftoning: a primitive for non-photorealistic shading. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering* (2002), Eurographics Association, pp. 227–232.

[10] Gimp. `http://gimp.org/`.

[11] Gimp#. `http://gimp-sharp.sourceforge.net/`.

[12] Gooran, S. Digital halftoning (course material). http://staffwww.itn.liu.se/~sasgo/TNM011/Digital_Halftoning.pdf.

[13] Hausner, A. Versatile decorative halftoning. *journal of graphics, gpu, and game tools 13*, 2 (2008), 1–12.

[14] Heckbert, P. S. Color image quantization for frame buffer display. *ACM Computer Graphics (ACM SIGGRAPH '82 Proceedings) 16*, 3 (1982), 297–307.

[15] Jarvis, J., Judice, C., and Ninke, W. A survey of techniques for the display of continuous tone pictures on bilevel displays. *CGIP 5*, 1 (March 1976), 13–40.

[16] Kite, T., Damera Venkata, N., Evans, B., and Bovik, A. A fast, high-quality inverse halftoning algorithm for error diffused halftones. *Ieee Transactions on Image Processing 9*, 9 (September 2000), 1583–1592.

[17] Knuth, D. E. Digital halftones by dot diffusion. *ACM Trans. Graph. 6*, 4 (1987), 245–273.

[18] Lau, D., Ulichney, R., and Arce, G. Blue and green noise halftoning models. *Signal Processing Magazine, IEEE*, 4 (Jul 2003), 28–38.

[19] Lau, D. L., and Arce, G. *Modern Digital Halftoning*. CRC Press, 2001.

[20] Lieberman, D. J., and Allebach, J. P. Efficient model based halftoning using direct binary search. In *ICIP '97: Proceedings of the 1997 International Conference on Image Processing (ICIP '97) 3-Volume Set-Volume 1* (Washington, DC, USA, 1997), IEEE Computer Society, p. 775.

[21] Metaxas, P. T. Parallel digital halftoning by error-diffusion. In *PCK50: Proceedings of the Paris C. Kanellakis memorial workshop on Principles of computing & knowledge* (2003), ACM, pp. 35–41.

[22] Mitsa, T., and Parker, K. J. Digital halftoning using a blue noise mask. In *SPIE Proceedings, Image Processing Algorithms and Techniques II* (1991), vol. 1452.

[23] Mulligan, J. B., and Ahumada, A. J. Principled halftoning based on human vision models. In *Human Vision, Visual Processing, and Digital Display III* (1992), vol. 1666, SPIE, pp. 109–121.

[24] Ostromoukhov, V. *Reproduction couleur par trames irrégulieres et semi-régulieres, Ph.D. Thesis No.1330*. PhD thesis, EPFL, Lausanne, Switzerland, 1995.

[25] Ostromoukhov, V. Artistic halftoning: Between technology and art. In *Proceedings of Color Imaging: Device-Independent Color, Color Hardcopy, and Graphic Arts V* (2000), R. Eschbach and G. G. Marcu, Eds., vol. 3963, SPIE, pp. 489–509.

[26] Ostromoukhov, V. A simple and efficient error-diffusion algorithm. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM, pp. 567–572.

[27] Pappas, T. N., Allebach, J. P., and Neuhoff, D. L. Model-based digital halftoning. *IEEE Signal Processing Magazine 20*, 4 (July 2003), 14–27.

[28] Pohlmann, K. *Principles of Digital Audio.* McGraw-Hill Professional, 2005.

[29] Riemersma, T. A balanced dithering technique. *C/C++ Users J. 16*, 12 (1998), 51–58.

[30] Sierra, F. LIB 17 (Developer's Den) CIS Graphics Support Forum.

[31] Stucki, P. Mecca - a multiple error correcting computation algorithm for bi-level image hard copy reproduction. research report rz1060. Tech. rep., IBM Research Laboratory, Zurich, Switzerland, 1981.

[32] Ulichney, R. A. *Digital Halftoning.* MIT Press, Cambridge, 1987.

[33] Ulichney, R. A. Dithering with blue noise. *Proceedings of IEEE 76*, 1 (Jan. 1988), 56–79.

[34] Ulichney, R. A. Void-and-cluster method for dither array generation. *Human Vision, Visual Processing, and Digital Display IV 1913* (1993), 332–343.

[35] Ulichney, R. A. A review of halftoning techniques. *SPIE 3963* (2000), 378–391.

[36] Vanderhaeghe, D., and Ostromoukhov, V. Polyomino-based digital halftoning. In *IADIS International Conference on Computer Graphics and Visualization 2008* (jul 2008).

[37] Velho, L., and Gomes, J. d. M. Digital halftoning with space filling curves. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques* (1991), ACM, pp. 81–90.

[38] Velho, L., and Gomes, J. d. M. Stochastic screening dithering with adaptive clustering. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (1995), ACM, pp. 273–276.

[39] Veryovka, O., and Buchanan, J. W. Halftoning with image-based dither screens. In *Proceedings of Graphics Interface '99* (June 1999), Morgan Kaufmann, pp. 167–174.

[40] Wikipedia. Mezzotint — wikipedia, the free encyclopedia, 2009. `http://en.wikipedia.org/w/index.php?title=Mezzotint&oldid=274588002`.

[41] Wong, T.-t., and Hsu, S.-c. Halftoning with selective precipitation and adaptive clustering. In *Graphics Gems V* (1995), pp. 302–313.

[42] Zhou, B., and Fang, X. Improving mid-tone quality of variable-coefficient error diffusion using threshold modulation. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), ACM, pp. 437–444.

Appendix A

Future work

A.1 What should be implemented next?

As a large number of different halftoning methods exist, it was not possible to implement everything at once. There is also a room for further refinement of the library and the GUI. In this chapter is suggested what can be improved.

Improving current methods, GUI, and infrastructure

- The module configuration panel should be integrated to every meaningful module dialog or panel to enable storing presets for particular modules, not only for the top-level `HalftoneAlgorithm`. This could greatly improve the usefulness of the GUI for the user.
- Every module should be profiled and optimized. For sure a performace boost can be achieved even with modularity and readability preserved.
- XML serialization formatter should be used instead of binary one for configuration storage to make it more transparent and future-proof. At first must be solved some technical issues with serialization of abstract hierarchies.
- Spot functions are currently evaluated for each pixel. This allows for arbitrary screen angles, but it is not quite efficient. Better would be to generate tileable threshold matrices with spot functions and using discrete rotations as described in [24].
- Dot gain correction curves are currently approximated by the gamma curve. It would be better to represent them via GIMP Curves command or via a table of custom values.
- Spot functions and `ImageGenerator` effects deserve a better GUI which would enable users to edit them directly. Custom spot functions defined in form of lambda functions could be compiled in place. Probably a better approach would be to take advantage of an expression parser[1].

[1] For example Fast Lightweight Expression Evaluator – `http://www.codeplex.com/Flee`.

- For further experimentation it would be interesting to try some perturbation noise generators with distributions different than uniform (white noise), eg. with Gaussian distribution. In case a random value is needed for processing every pixel, pre-generating the noise to an `Image` with an external noise generating GIMP plug-in (just like a `ImageGenerator` does) might be more efficient than generating the values in place.

New halftone methods Many other halftone methods should be implemented. Among them preferably some model-based ones, such as Direct Binary Search [20], or some novel approaches like Polyomino halftoning [36]. Omnidirectional error diffusion and also further methods of generating threshold matrices should be examined, such as Void-and-cluster method [34], or artistic matrices from vector images [25].

Halftone quality measurement For objective measurement of halftone quality, some existing metrics should be implemented: blue-noise metrics, such as radially averaged power spectrum, anisotropy and Fourier spectrum, and HVS-model metrics of quantization error.

Further abstraction The library should be still more abstracted to support other intensity representations (at least 16-bit integer and normalized double), multi-level quantization, or even multi-channel (color) halftoning. Imagine a blue-noise halftoning method performed during the conversion of a 16-bit photograph with some smooth gradients to 8-bit resolution.

Portability and different back-end libraries The library should be made completely independent on Gimp# and different image processing libraries should be supported in the image abstraction layer. Deployment on Mono on unix systems should be tested. Also a unix package with makefile-driven building should be considered to enable standard package distribution.

Different user interfaces Also, the user interface should be made independent on GIMP. Three possible interfaces should be considered:

- A special GUI for studying halftone methods with integrated quality metrics.
- A console-based UI for batch processing.
- A simplified user-friendly GUI to halftone images into big multiple-page posters with PDF output. There is a popular application Rasterbator[2] which does exactly this job but is limited just to simple circular screening dots. The potential result of co-operation of the two projects could be very promising.

[2] `http://homokaasu.org/rasterbator/`

Appendix B

CD contents

The enclosed CD is organized as follows:

- `doc/`
  - `doxygen/` – API documentation
  - `README` – User's guide with installation instructions
  - `LICENSE` – License information
- `install/`
  - `gimp-2.4.6-i686-setup.exe` – GIMP installer
  - `gimp-sharp-setup-0.14-gimp-2.4.exe` – GIMP# installer
  - `halftonelab-setup.exe` – HalftoneLab installer
- `src/`
  - `halftonelab/` – HalftoneLab source code
  - `gimp-sharp/` – GIMP# source code
- `thesis/`
  - `latex/` – LaTeX sources of the thesis
  - `pdf/` – PDF output

Appendix C

User's guide

C.1 System requirements

Halftone Laboratory should run everywhere where Gimp# is able to run. Currently, it was tested sucessfully on Windows XP SP2 with .NET 3.5, GIMP 2.4.6, GIMP# 0.14 and GTK# 2.10.

- Operating system: Windows XP, Linux
- .NET 3.5 / Mono 2.4
- GIMP $\geq$ 2.4
- GIMP# $\geq$ 0.13

C.2 Installation

First GIMP and GIMP# must be installed, run `gimp-2.4.6-i686-setup.exe` and then `gimp-sharp-setup-0.14-gimp-2.4.exe`. After that you can install HalftoneLab with `halftonelab-setup.exe`. All installers act as wizards, so the installation is straightforward.

HalftoneLab stores its configuration in `halftonelab.cfg` file, located in the Application Data directory. On Windows platform it can be `Application Data` in the user directory, on other systems `.config` in the home directory, but it depends on specific operating system and its configuration. If the configuration file does not exist, it is automatically created and filled with some example configuration.

C.3 Usage

The basic graphical user interface for HalftoneLab project is available in form of a GIMP plug-in. So at first launch the GIMP application and open an image. Currently, HalftoneLab is able only to process GIMP images in greyscale mode. To convert an image to that mode click in the menu on Image → Mode → Greyscale.

Then you can start the Halftone Laboratory plug-in with Filters → Distorts → Halftone Laboratory. A main dialog opens. It consist of several panels:

- Configuration saving and loading
- Pre-processing
- Halftone method
- Post-processing

There you can configure the desired halftone algorithm. Hitting the OK button automatically saves the current configuration and starts the algorithm to process the image. The Cancel button quits the dialog. Last configuration is automatically saved, so subseqently hitting the Filters → Repeat "Halftone Laboratory" starts processing with the last configuration and without showing the dialog.

Configuration saving and loading This panel enables you to manage algorithm configurations – save current one along with a name and description to a persistent storage and load it again. Select a name from the combobox to load that configuration. There are two special configurations: *_DEFAULT* reverts all modules to their default settings, *_LAST* is the last configuration automatically saved at the end of the previous session.

To save the current configuration hit the Save button on the configuration panel. A dialog prompting for configuration name and description appears. A name is mandatory and cannot be "_DEFAULT" or "_LAST" or the configuration is not saved, a description is optional, but can help to quickly see what the algorithm does. The configurtion selected in the combobox can be removed hitting the Delete button.

Pre-processing

In the pre-preprocessing panel you can configure how the image will be altered before halftoning itself takes place. Modules are ordered as they will be executed. Each module can be enabled or disabled.

Resize With this module you can scale the image by a given factor using selected interpolation method – *Nearest neighbour*, *Bilinear*, *Bicubic*, *Lanczos*.

Sharpen Sharpening prior to halftoning can help preserving image details as most halftoning methods slightly blur the image. The amount of sharpening can be set.

Dot gain Dot gain (see page 9) caused by the behaviour of some printing processes can be corrected here. Simple gamma correction acts as a rough approximation of dot gain correction curves.

Post-processing

Post-processing panel embraces modules executed after the halftoning. It acts similarly to the pre-processing panel.

Resize The post-processing resize module acts the same way as its pre-processing counterpart. It is typically used to down-sample the image. You can perform *supersampling* technique with the two resize modules – in the pre-processing one set the desired upsample factor and in the other module enable SUPERSAMPLING checkbutton. The downsample factor will be computed automatically. Bicubic interpolation is a recommended method for supersampling.

Smoothen If you want to smoothen the halftoned image without the computation overhead of supersamping you can use the `Smoothen` module. First it Gaussian-blurs the image with given radius and then it applies Levels GIMP command. Thus, the rough black and white contours get perfectly smooth. However, the cost is bluring some fine details.

Halftone method

This is the most important control panel in the plug-in. There you can set the type of halftoning method to be used and its details. The panel itself follows a concept of submodule selector, which appears on many places inside specific module configuration dialogs. It consists of three elements: In a combobox there are submodule types, one of which can be selected. On hitting the EDIT button a configuration dialog for particular submodule type opens (provided there is anything to configure). Sometimes, the submodule is optional and it is possible that no type is set. To achieve this there is a NULL checkbutton. If it gets checked the module is unset and its internal settings get deleted.

Thresholding halftone method In thresholding methods the image is processed pixel-wise along a scanning order given by a `ScanningOrder` module. Each pixel is quantized using threshold computed by a `ThresholdFilter` module and the quantization error is optionally diffused by a `ErrorFilter` module. Each submodule can be configured via a submodule configuration panel. As for `ErrorFilter`, there is a USE ERROR FILTER? checkbutton to enable or disable the module without removing its configuration.

SFC clustering halftone method SFC clustering method, described on page 14, works differently. The main parameter, MAXIMUM CELL SIZE, controls the coarseness of the halftone (or the spatial vs. tonal resolution trade-off) – lower values give finer detais, but higher values enable representing more tones. There are checkbuttons for controlling the use of adaptive clustering[1] and cluster position-

[1] varying cell sizes according to local detail amount

ing techniques. Setting Minimum cell size value is meaningful only when using adaptive clustering.

The type of space-filling curve to scan along can be selected via `Scanning order` submodule panel. Currently only Hilbert SFC is supported. An optional `VectorErrorFilter` can be used (and is recommended).

Matrix threshold filter Matrix threshold filter enables you to edit a single thresholding matrix. The matrix dimensions can be changed using the Resize button. Matrix coefficients can be in incremental form (see page 10) or already normalized (scaled to 0-255 range). This can be set via the Coefficients already scaled? checkbutton.

Dynamic matrix threshold filter Dynamic matrix threshold filter allows a different matrix for each pixel intensity or a range of intensities as well as perturbing their threshold coefficients with random noise. There is a table of records consisting of a threshold matrix, starting intensity of that range[2] and noise amplitude. You can add a new record hitting the New button, which opens a separate dialog. Selected record can be edited using the Edit button in the same dialog or deleted with the Delete button. To delete all records at once hit the Clear all button. Applying perturbation noise can be enabled or disabled for the whole table with the Noise enabled? checkbutton.

Spot function threshold filter A spot function analytically defines threshold values for digital screening (see page 11). There are several presets for most common spot functions: Euclid dot[3], Euclid dot with random perturbation, Square dot, Line, Triangle dot. For each preset two parameters can be set: Screen angle (in radians) – the angle of screen rotation, Screen line distance (in pixels) – distance between adjacent screen elements.

Image threshold filter Image threshold filter behaves much like the Spot function threshold filter, except that spot functions are not evaluated at run-time but rather a big threshold matrix is pre-generated – into a GIMP image. The advantage is that the image can be then distorted with GIMP image filters. The possibilities are tremendous. A few examples are provided in the form of presets. The initial spot function can be configured the same way as in Spot function threshold filter. Currently, the effects applied cannot be controled, only a description is shown. One of the presets use GIMP Patterns to tile the plane, histogram is then equalized – this with an error filter enabled give approximately the Veryovka-Buchanan method [39] . To select a pattern, go to Dialogs → Pattern in Image window menu and select the desired pattern.

[2] the range spans up to the next record

[3] with growth: circle → square at intensity 0.5 → circle

Matrix error filter Matrix error filter performs error-diffusion with a single error matrix. The way of its editing is similar to the Matrix threshold filter dialog. Dimensions of the matrix can be changed using the Resize button.

Error matrix coefficients are represented as fractions – numerators are in the table and a common denominator is in the Divisor spinbutton. As the sum of all error matrix coefficients must be equal to 1.0, they must be scaled using the their sum. However, in case you want to experiment, you can choose a different divisor. Just enable the Use a custom divisor? checkbutton and fill the Divisor spinbutton.

The relative position of source pixel in the matrix can be controlled by the Source offset X (the Y offset is always the same – the first row).

Dynamic matrix error filter This is a concept similar to Dynamic matrix threshold filter (and the controls are almost identical) – there can be different error matrices for different pixel intensities or intensity ranges, however, there is no noise added.

Randomized matrix error filter Error matrix coefficients can be generated randomly for each pixel. There is a 'template' matrix which defines the dimensions of the final matrix and where the coefficients would be generated. A constraint is that error can be distributed only to places after the source pixel (following the scanning order). There are two possible modes controlled by the Randomize coefficient count? checkbox: If it is enabled, coefficients will be generated exactly in place of all non-zero coefficients in the template matrix, maintaining a constant number of them, otherwise the number and position of coefficients will be varied up to the available capacity of the matrix.

Perturbed matrix error filter For perturbing error matrix coefficients with random noise a separate filter is used. Inside it has a `MatrixErrorFilter` child filter as a submodule controlled by a standard submodule panel. In addition, the amount of noise can be controlled by the Perturbation amplidute scale.

Vector error filter For 1-D scanning, such as along an space-filling curve, a vector error filter is suitable. Its interface is similar to that of Matrix error filter, except that it can be resized only in one dimension (Length).

Scanning order For general scanning order you can select among *Scanline*, *Serpentine* and *Hilbert* scanning, for SFC scanning only *Hilbert* is available. Scanline processes pixels line by line in the same direction, serpentine in a zig-zag direction. Hilbert scans along an approximation of the Hilbert space-filling curve.