From Stippling to Scribbling

Abdalla G. M. Ahmed
abdalla_gafar@hotmail.com

Abstract

We address the brightness/contrast problem in some line-based artistic halftoning methods including TSP Art, MST halftoning, and recursive division methods, which all work by connecting stipple points. We suggest a general solution, and we introduce three new line-based halftoning styles.

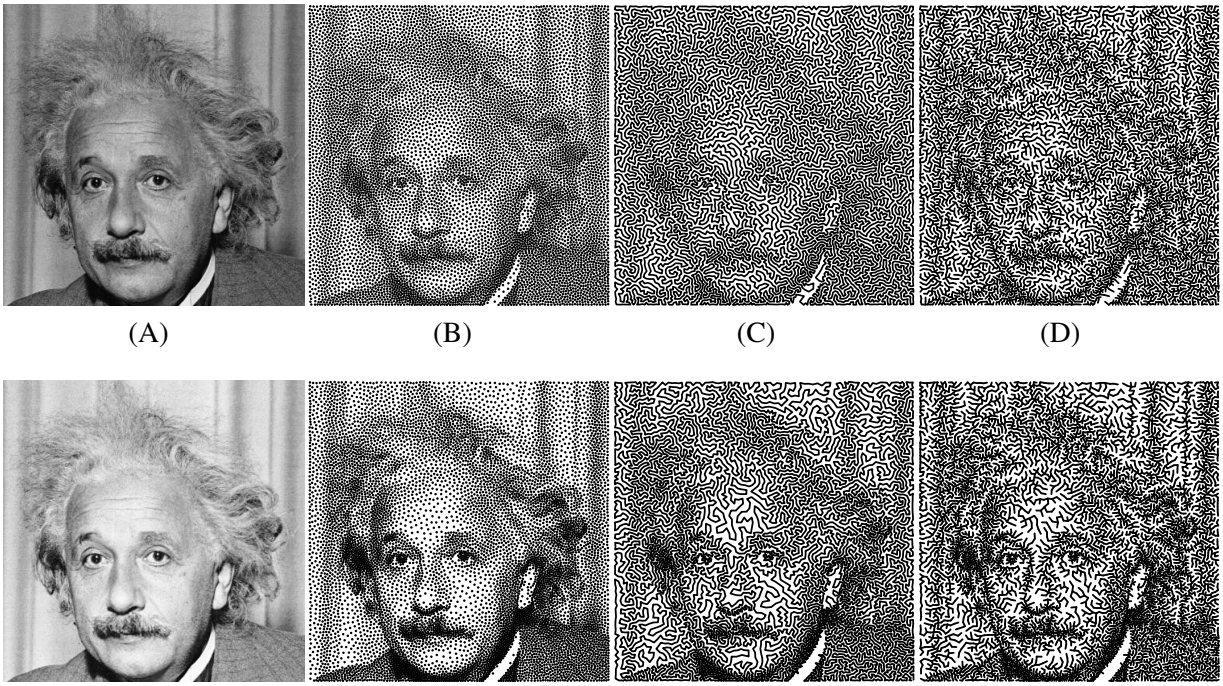


Figure 1 : *(A) A 256-tone image; rendered using (B) 10k CCCVT stipples, (C) a TSP tour through the stipples, and (D) a minimum spanning tree over the stipples; with loss of details and noticeable degradation of contrast in (C, D). (a) Shows the pre-processed image (as described in the paper), the resulting (b) stippling, and (c, d) line-based renderings, which match the original image better than (C, D). Note that same amount of ink in the reference image (A) is used in the two stipplings and line-based renderings in all figures.*

Introduction

Halftoning is the process of rendering a continuous tone image on a monochrome device. The objective is to make the rendered image look as close as possible to the input image when viewed from a distance. To achieve that goal a halftoning technique needs to adhere to Pnueli and Bruckstein paradigm: "The local density of black elements should be proportional to the local grayness of the original image" [11]. The word "density" refers to the proportion of area covered by black. 'Functional' halftoning (e.g. in printers) uses tiny dots of ink as "black elements", and they are evenly distributed in the covered area so that they

blend smoothly with the white background and give the illusion of a gray tone. If you are reading the printed version of this article then Figure 1(A) shows an example of functional halftoning. Besides functional halftoning there are also artistic halftoning methods which intentionally make the black elements visible in a closeup view, and blending with the background only occurs in a distant view. The objective is to show interesting details in a closeup, and/or reveal an image in a distant view. Black elements can be letters (ASCII art), Truchet tiles [4], solid circles [7, 13], lines [11, 5, 9, 2], or any other thing.

Stippling is a stylized halftoning technique which replaces the tiny dots of ink by large, clearly visible 'stipples'. See Figure 1(B). Stippling was commonly used for illustrations in archeology and biology books, as it can convey both tone and texture [7]. Stippling differs from functional halftoning in the size of dots, but they both share the same requirement that dots are distributed evenly and isotropically (no noticeable grid or pattern). Over the past 15 years a lot of research in computer graphics was devoted to computer-generated stippling (e.g. [7, 13, 3, 6]), and it culminated into weighted Centroidal Voronoi Tessellation (CVT) and eventually Capacity-Constrained Centroidal Voronoi Tessellation (CCCVT). In a nutshell, a Voronoi tessellation of a set of points associates with each point the area (Voronoi cell) closer to it than to any other point, darker regions carry higher weights in calculating cell areas, the capacity constraint enforces equal capacity (weighted area) to cells, and "centroidal" means that each point resides in the (weighted) center of its cell. From a halftoning point of view, CCCVT aggregates into each point all the ink in its cell [6]; it is a stippling technique which adheres faithfully to Pnueli and Bruckstein paradigm. See Figure 1(B).

Line-based halftoning is another category of artistic halftoning which uses lines as black elements. Many ideas for line based halftoning were proposed (e.g. [11]), but in this paper we are interested in methods which work by connecting stipple points [5, 10, 9, 2]. Upon using a point set which faithfully reproduce the tone (e.g. Figure 1(B)), these techniques fail to correctly reproduce the mid-tones (e.g. Figure 1(C, D)). In this paper we discuss this problem and propose a general solution. We start by reviewing the range of line-based methods which can use our suggested solution.

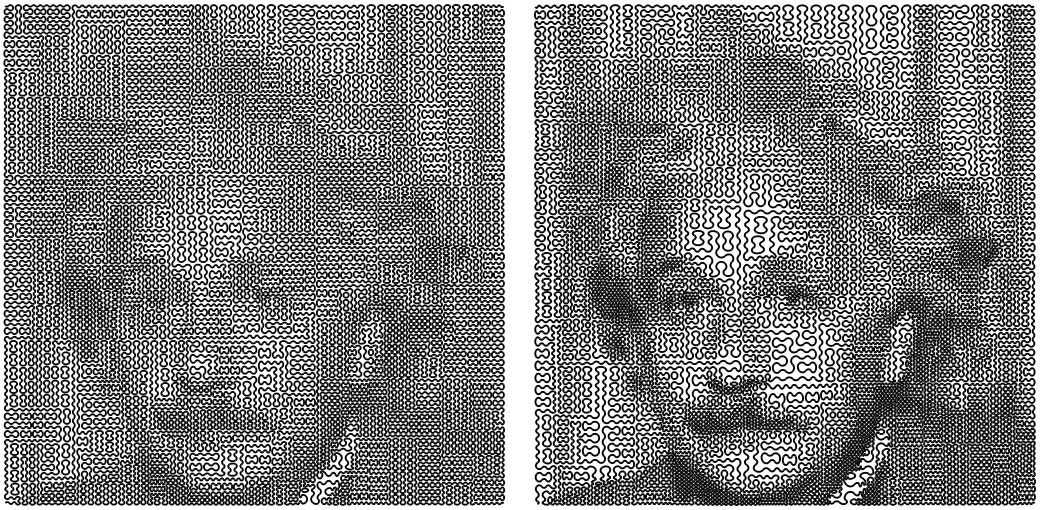
Connecting Stipple Points

TSP Art is a line-based artistic halftoning technique invented by Bosch and Herman [5] and greatly improved by Kaplan and Bosch [10]. It works by stroking the solution path of a traveling salesman problem (TSP) over points of a stippled rendering. TSP is the problem of minimizing the total traveled distance for a salesman based in one city who must visit each of a set of other cities exactly once before returning home. An optimum TSP tour is always a single non-self-intersecting loop. Figure 1(C, c) show example TSP renderings.

As an alternative to a TSP tour, Inoue and Urahama [9] suggested connecting stipple points with a minimum spanning tree (MST), which is far easier to generate than an optimum TSP tour. They used Prim algorithm [12], which can be stated in a few words: initialize the tree to an arbitrary point, then repeatedly connect the nearest non-connected point to the existing tree, until all points are connected. Figure 1(C, c) show example MST renderings.

Inspired by TSP and MST, Ahmed recently introduced a family of line-based rendering styles [2]. Rather than using an arbitrary underlying point set, his approach generates points by recursively dividing an image into rectangles of equal total amount of black, and placing one or more points inside each rectangle. He then exploits topological relations between rectangles to connect points in four distinct ways. Figure 2 shows example recursive division (RD) renderings.

Loss of contrast and blurring of mid-tone details is a common problem in all these line-based methods. This issue was noticed by Kaplan and Bosch [10] who offered an empirical solution for a specific underlying stippling technique. They also mentioned alternative workarounds, including boosting the contrast of the



(A) (B)

Figure 2: *Example recursive division rendering of (A) the original image, and (B) the pre-processed image of Figure 1.*

input images, and varying the thickness of the line segments according to the local densities of points. Although the latter solution gives some wiggle room for more faithful reproduction of tone, it is arguable if this should still be called a line-based rendering. Indeed, a truly line-based device, such as a pen plotter, an egg-bot, or an embroidery machine, can not vary line thickness. Inoue and Urahama [9] also considered the contrast problem, and they formulated a solution, but it too was geared towards one specific stippling technique. Ahmed [2] also mentioned the contrast problem, but did not suggest a solution.

We will subsequently formulate a general solution to the contrast problem, but we would first like to comment on tonal errors that can be propagated due to the underlying stippling technique. Recursive division methods use an exact approach for point set generation, but TSP and MST implementations typically use weighted CVT, which imposes some discrepancy in tone. As we mentioned in the introduction, CCCVT is more faithful in preserving tone, and we therefore recommend using it[1]. Another important source of image degradation is caused by thresholding errors. For example, both grid-based and ordered dithering approaches described in [10] quantize large 'cells' of the underlying image on the basis of tone, and place stipples accordingly, which introduces substantial quantization errors. Thresholding is a well-studied issue in computer graphics. One effective solution is error diffusion, using the famous Floyd-Steinberg algorithm [8].

Analysis and Treatment

Suppose that we take a stippling which faithfully reproduces the tone of an image, and we connect stipples to produce the line-based halftoning. In TSP Art each point is, by definition, connected to exactly two points; representing the city from which the salesman came and that to which he will go next. If we associate each point with the outgoing edge, then each point in the stippling is replaced by exactly one line segment in the line rendering. Similarly, in MST each point (except for one) can be associated with a single line segment; namely the one used to link it to the tree in the generating algorithm. Since both TSP and MST optimize for minimum edge lengths, a point is typically connected to one of its immediate neighbors; therefore the length of the line segment associated with each point is proportional to the average distance between points in the

[1] As of this writing, authors of [3] and [6] make their code freely available in their websites.

region. Since stipple points represent the black elements in Pnueli and Bruckstein paradigm, their density should be proportional to the local grayness of the original image. Conversely, the average distances between points is larger in light regions of the image than in dark regions. Upon connecting the stipple points using TSP or MST, the same amount of ink used for each point should be spread into the associated line segment; but this lead to a contradiction: we have to either use more ink where line segments are longer, or make the lines narrower as suggested by Kaplan and Bosch [10]. This explains why a stippling which faithfully reproduces the tones of an image generates a line-based rendering which fails to reproduce the tone correctly (assuming constant line thickness).

To find a solution we proceed to inspect the relation between "density of black elements" in stippling and in associated line rendering. As we mentioned in the introduction, a faithful (capacity-constrained) stippling aggregates into each point the ink from its Voronoi cell. If ink is spread back evenly in the cell its density (amount per unit area) would be:

$$\rho_{\text{in}} = \frac{c}{a}. \tag{1}$$

Where a is the area of the cell, and c is the size of the stipple. A unit area of an image is a pixel, so ρ_{in} is essentially the average pixel value in the cell in the input image[2]. Similarly, in a line rendering ink is aggregated in the line segment associated with the stipple point, and if spread back in the cell its density would be:

$$\rho_{\text{out}} = \frac{w \times l}{a}. \tag{2}$$

where l is the length of the line segment, and w is the line width used for stroking, which we assume fixed. If cells look similar (a reasonable assumption for an even isotropic distribution of points) then the area of each cell should be proportional to the square of the distances between points, or:

$$l \propto \sqrt{a}\,. \tag{3}$$

Substituting (3) in (2) we find that:

$$\rho_{\text{out}} \propto \frac{1}{\sqrt{a}}. \tag{4}$$

Comparing (4) to (1), we see that:

$$\rho_{\text{out}} \propto \sqrt{\rho_{\text{in}}}. \tag{5}$$

This represents the general relationship in the local density of ink between stippling and the associated TSP or MST rendering. The implication of (5) is that in order for a region in a line rendering to look n-times dark than another region, it has to be n^2-times darker in the underlying stippling and input image. In other words, we need to square the darkness levels in the input image in order to maintain the appropriate proportions of darkness in the final line rendering. This leads us to the following simple pre-processing steps of input images:

1. Square pixel values,
2. scale them back to the standard range $[0, 255]$, making sure to apply error diffusion to reduce the potentially large quantization errors, and
3. use the resulting image to generate stipple points for line rendering[3].

[2]Typical grayscale pixel values run from 0 (black) to 255 (white), but in our discussion we will assume that 0 means white (no ink) and 255 means black (ink covers the whole pixel).

[3]There is a small tricky thing in our derivation: we are using the average of squared pixel values $\mathrm{E}\left(\rho^2\right)$, whereas we are supposed to use the square of average density $\left(\mathrm{E}\left(\rho\right)\right)^2$ (which is not easy because cell sizes are not known apriori). It can be shown that $\mathrm{E}\left(\rho^2\right) \geq \left(\mathrm{E}\left(\rho\right)\right)^2$, where equality holds for fixed ρ, and the difference is proportional to the variance in pixel values. Thus, regions with varying gray levels (edges and feature lines) tend to become a little bit darker; which is not bad, as supported by results.

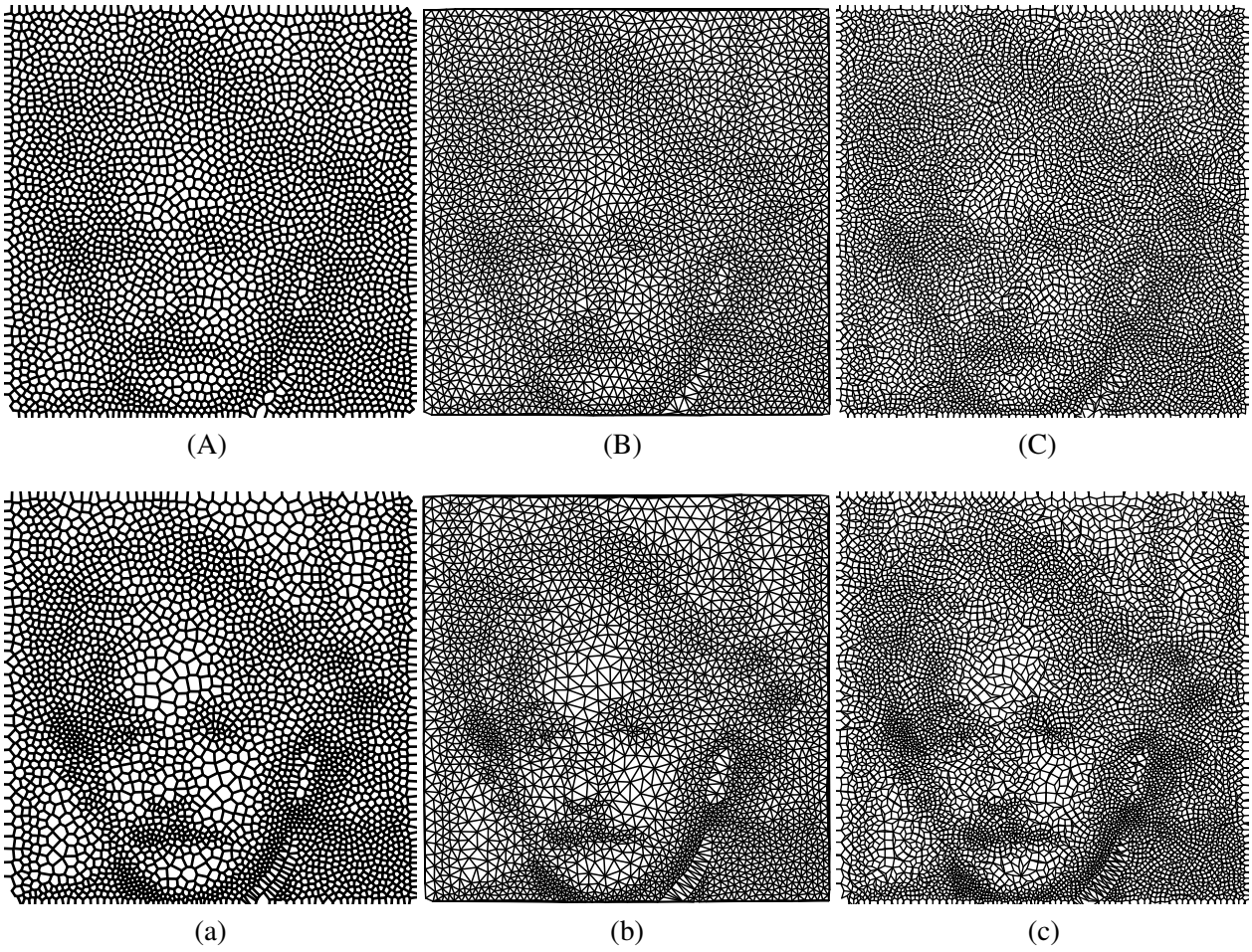


Figure 3 : *3k Stipplings of the the original and pre-processed image of Figure 1 converted into line renderings, in respective rows, by (A, a) stroking the Voronoi tessellation, (B, b) stroking the Delaunay triangulation, and (C, c) connecting points to vertices of Voronoi cells. The improvement of pre-processing is evident.*

Applying these steps leads to significant improvement and fairly faithful reproduction of tone in TSP and MST halftoning, as demonstrated in Figure 1.

Even though the discussion above is devoted to TSP and MST, a similar argument holds for RD methods; just that instead of a single line segment a whole set of straight or curved segments is associated with each stipple point. Figure 2 demonstrates the effect of pre-processing on RD rendering. Further, the suggested pre-processing should work well with any other line-based rendering which works by connecting stipple points in a consistent way. To demonstrate this we introduce three new rendering styles, as illustrated in Figure 3. The first style is obtained by stroking the Voronoi tessellation of stipple points, which gives the impression of pavement. In the second style we connect all points which have adjacent Voronoi cells. This is known as a Delaunay triangulation, and it gives the impression of a structure. In the third style we connect each point to vertices of its Voronoi cell; which give an impression of cubes. Given a set of stipple points, these diagrams are relatively easy to produce using, for example, CGAL library [1] (see appendix). Figure 4 and Figure 5 show four more examples of line renderings after pre-processing the input images.

Conclusion

We conclude this paper by highlighting the mathematical part of the story. In going between functional halftoning (spreading ink over areas), line-based halftoning, and stippling, we are actually switching between 2, 1, and 0 dimensions for spreading ink. In doing so, we would expect taking roots or raising to powers to take place, which is probably the intuition behind (5).

Acknowledgments. Thanks to Ahmed Fuad for his insightful discussions.

References

[1] CGAL, Computational Geometry Algorithms Library. `http://www.cgal.org`, as of Apr 24, 2015.

[2] Abdalla G. M. Ahmed. Modular line-based halftoning via recursive division. In *Proceedings of the Workshop on Non-Photorealistic Animation and Rendering*, NPAR '14, pages 41–48, New York, NY, USA, 2014. ACM.

[3] Michael Balzer, Thomas Schlömer, and Oliver Deussen. Capacity-constrained point distributions: A variant of lloyd's method. In *ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, pages 86:1–86:8, New York, NY, USA, 2009. ACM.

[4] Robert Bosch. Opt Art: Special Cases. In Reza Sarhangi and Carlo H. Séquin, editors, *Proceedings of Bridges 2011: Mathematics, Music, Art, Architecture, Culture*, pages 249–256, Phoenix, Arizona, 2011. Tessellations Publishing. Available online at `http://archive.bridgesmathart.org/2011/bridges2011-249.pdf`.

[5] Robert Bosch and Adrianne Herman. Continuous Line Drawings via the Traveling Salesman Problem. *Operations Research Letters*, 32(4):302 – 303, 2004.

[6] Fernando de Goes, Katherine Breeden, Victor Ostromoukhov, and Mathieu Desbrun. Blue noise through optimal transport. *ACM Trans. Graph.*, 31(6):171:1–171:11, November 2012.

[7] Oliver Deussen, Stefan Hiller, Cornelius Van Overveld, and Thomas Strothotte. Floating points: A method for computing stipple drawings. *Computer Graphics Forum*, 19(3):41–50, 2000.

[8] R. W. Floyd and L. Steinberg. An Adaptive Algorithm for Spatial Grey Scale. *Proceedings of the Society of Information Display*, 17:75–77, 1976.

[9] Kohei Inoue and Kiichi Urahama. Chaos and graphics: Halftoning with minimum spanning trees and its application to maze-like images. *Comput. Graph.*, 33(5):638–647, October 2009.

[10] Craig S. Kaplan and Robert Bosch. TSP Art. In Reza Sarhangi and Robert V. Moody, editors, *Renaissance Banff: Mathematics, Music, Art, Culture*, pages 301–308, Banff, Alberta, 2005. Canadian Mathematical Society. Available online at `http://archive.bridgesmathart.org/2005/bridges2005-301.html`.

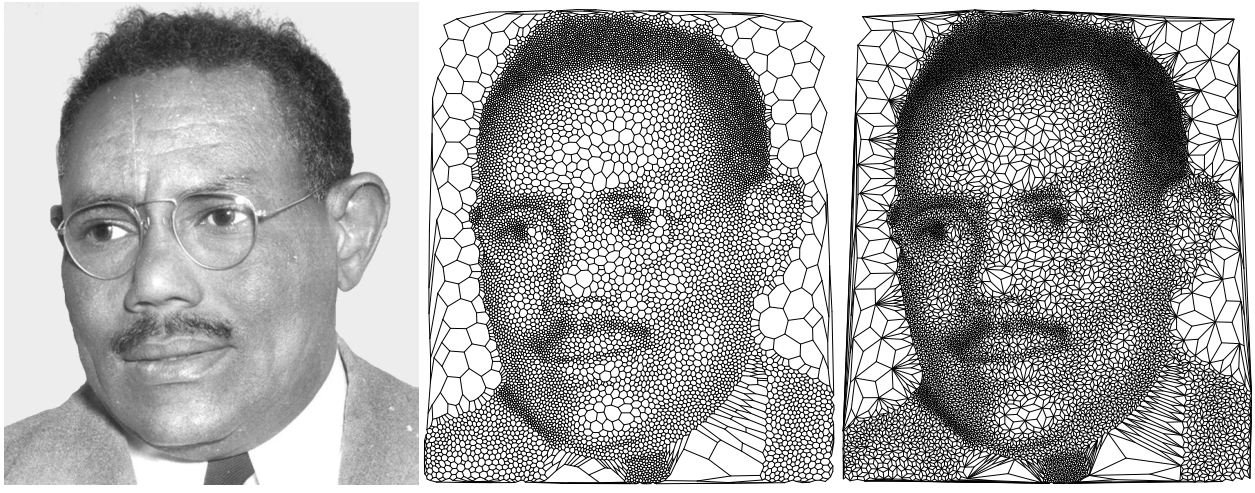
[11] Yachin Pnueli and Alfred M. Bruckstein. Gridless Halftoning: A Reincarnation of the Old Method. *Graphical Models and Image Processing*, 58(1):38 – 64, 1996.

[12] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[13] Adrian Secord. Weighted voronoi stippling. In *Proceedings of the 2Nd International Symposium on Non-photorealistic Animation and Rendering*, NPAR '02, pages 37–43, New York, NY, USA, 2002. ACM.

(A) (B) (C)

Figure 4: *RD renderings (edges style) to illustrate how the ratio of darkest to lightest values can be used to control contrast. Here the tones (pixel values) in (A) were mapped to (B) [1, 3] and (C) [1, 8] before squaring; consequently the largest rectangles can be up to (B) 9, or (C) 64 times the smalest rectangles. Both renderings use 14,000 rectangle. Photo of Remaz Merghani, used with permission.*



(A) (B) (C)

Figure 5: *Two more line rendering styles applied to the image in (A) (after pre-processing): (B) Similar to Figure 3(A, a), but uses centroids of Delaunay triangles instead of their circumcenters. Since centroids are always inside triangles, only three edges meet at each vertex, leading to more regularly shaped cells, and the overall rendering looks more organic. (C) Similar to Figure 3(C, c) but, again, we use centroids of Delaunay triangles instead of their circumcenters, which leads to a more even distribution of angles. The underlying stipplings use (B) 10,000 and (C) 5,000 points.*

A Example code

Below is an example C++ program which receives a list of stipple points from stdin and prints to stdout (eps format) a line rendering in one of the styles of Figure 3. The code uses the CGAL library [1].

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point;
typedef CGAL::Delaunay_triangulation_2<K> DT;

const char *USAGE_MESSAGE = "Usage: %s <style> <xleft> <ybottom> <xright> <ytop>\n"
"Styles:\n"
"0: Delaunay triangulation (default)\n"
"1: Voronoi Diagram\n"
"2: Voronoi cubes\n"
"3: Cubes with more even angles, less even edge length\n";

int main(int argc, char **argv) {
    if (argc < 6) {
        fprintf(stderr, USAGE_MESSAGE, argv[0]); exit(1);
    }
    int style = atoi(argv[1]);
    int xleft = atoi(argv[2]);
    int ybottom = atoi(argv[3]);
    int xright = atoi(argv[4]);
    int ytop = atoi(argv[5]);

    std::istream_iterator<Point> begin(std::cin);
    std::istream_iterator<Point> end;
    DT dt;
    dt.insert(begin, end);
    fprintf(stderr, "Read %ld points\n", dt.number_of_vertices());
    printf("%%!PS-Adobe EPSF-3.0\n"
        "%%%%BoundingBox: %d %d %d %d\n"
        "1 setlinecap 1 setlinejoin 0 setlinewidth\n"
        "/e {moveto lineto stroke} def\n",
        xleft, ybottom, xright, ytop
    );

    switch (style) {
        case 0: {
            DT::Finite_edges_iterator eit = dt.finite_edges_begin();
            for ( ; eit != dt.finite_edges_end(); eit++) {
                std::cout << dt.segment(eit) << " e\n";
            }
            break;
        }
        case 1: {
            DT::Finite_faces_iterator fit = dt.finite_faces_begin();
            for ( ; fit != dt.finite_faces_end(); fit++) {
                Point c = dt.circumcenter(fit);
                for (int i = 0; i < 3; i++) {
                    if (dt.is_infinite(fit->neighbor(i))) continue;
                    Point c1 = dt.circumcenter(fit->neighbor(i));
                    if (
                        (c.x() < c1.x()) ||
                        ((c.x() == c1.x()) && (c.y() < c1.y()))
                    ) std::cout << c << " " << c1 << " e\n";
                }
            }
            break;
        }
        case 2: {
            DT::Finite_faces_iterator fit = dt.finite_faces_begin();
            for ( ; fit != dt.finite_faces_end(); fit++) {
                Point c = dt.circumcenter(fit);
                std::cout << c << " " << fit->vertex(0)->point() << " e\n";
                std::cout << c << " " << fit->vertex(1)->point() << " e\n";
                std::cout << c << " " << fit->vertex(2)->point() << " e\n";
            }
            break;
        }
        case 3: {
            DT::Finite_faces_iterator fit = dt.finite_faces_begin();
            for ( ; fit != dt.finite_faces_end(); fit++) {
                Point p0 = fit->vertex(0)->point();
                Point p1 = fit->vertex(1)->point();
                Point p2 = fit->vertex(2)->point();
                double x = (p0.x() + p1.x() + p2.x()) / 3;
                double y = (p0.y() + p1.y() + p2.y()) / 3;
                Point c = {x, y};
                std::cout << c << " " << p0 << " e\n";
                std::cout << c << " " << p1 << " e\n";
                std::cout << c << " " << p2 << " e\n";
            }
            break;
        }
        default:
            fprintf(stderr, "Undefined style %d\n", style);
            exit(1);
    }
    std::cout << "showpage\n";
    return 0;
}
```