Aliasing Artifacts and Accidental Algorithmic Art

Craig S. Kaplan
School of Computer Science
University of Waterloo
csk@cgl.uwaterloo.ca

Abstract

While developing a program to render Voronoi diagrams, I accidentally produced a strange and surprising image. The unexpected behaviour turned out to be caused by a combination of reasons from signal processing and computer architecture. I describe the process that led to the pattern, explain its structure, and display many of the wonderful designs that can be produced from this and related techniques.

1. Introduction

This paper tells the story of a surprising image that I discovered by accident, and of the exploration that followed as I attempted to understand this image and produce more like it. The discovery begins with a program to draw Voronoi diagrams.

A Voronoi diagram is a subdivision of space induced by a set of generators. In its simplest form, the generators are points $\{G_i\}_{i=1}^n$ in the plane, and the Voronoi diagram associates every point in the plane with the generator to which it is closest (measured using ordinary Euclidean distance). If generator G_i is given colour c_i, the diagram can easily be visualized by colouring every point in the plane with the colour of its associated generator.

In previous work, I developed a simple renderer to explore ornamental applications of Voronoi diagrams [4]. Every pixel in the output image is interpreted as a point in the plane, and its distance to every generator is computed. The closest generator wins, and its colour is assigned to that pixel. There are other, more efficient algorithms for computing Voronoi diagrams, but this approach is simple and easily extendible to non-point generators. The algorithm is shown on the left in Figure 1. Note the minor optimization of comparing squared Euclidean distances directly, avoiding a costly square root.

One day, I accidentally ran this algorithm on a degenerate set of two generators: one black, one white, both located at the origin. We can specialize the original algorithm on these two generators, as shown on the right in Figure 1. The result seems trivial. No number is less than itself, and so we expect the program to produce a solid black image. But my renderer produced the surprising result shown in Figure 2(a). Not only does the image have both black and white pixels, it has them in an intricate pattern with a great deal of structure. The pattern appears to change in concentric rings around the centre of the image. Furthermore, the structure is very brittle: rendering the same region of the plane with sample points moved just a small amount changes the result completely, as shown in Figure 2(b).

The complexity of these patterns (which I will simply call "Voronoi patterns") is clearly not contained in the program. It must therefore be inside the computer's hardware. The rest of this paper tries to answer this riddle. First, we take a detour to explore a large family of well-known computer generated

```
for each pixel $(i, j)$ corresponding to
        point $(x, y)$ in the plane:
    $d = \infty$
    $c =$ *unknown*
    for each generator $G_i$ with colour $c_i$:
        if $\|(x, y) - G_i\|^2 < d$:
            $d = \|(x, y) - G_i\|^2$
            $c = c_i$

    set pixel $(i, j)$ to colour $c$
```

```
for each pixel $(i, j)$ corresponding to
        point $(x, y)$ in the plane:
    $d = x^2 + y^2$
    if $x^2 + y^2 < d$:
        set pixel $(i, j)$ to white
    else:
        set pixel $(i, j)$ to black
```

Figure 1 A simple algorithm for rendering Voronoi diagrams (left), and a version specialized on two generators both located at the origin (right).

interference patterns. We then return to the behaviour of the program above, and see why these interference patterns are lurking inside such a seemingly simple program.

2. Aliasing artifacts on the line and in the plane

Consider the function $f(x) = \sin(x^2)$. The graph of this function should display an increasingly compressed sine wave as we move away from the origin. Indeed, since we would say that $\sin(2\pi\alpha x)$ has frequency α for any $\alpha \neq 0$, we can describe $f(x) = \sin((x/2\pi)(2\pi x))$ as having a linearly increasing "instantaneous frequency" of $x/2\pi$ at every point x.

We might imagine plotting a graph of $f(x)$ by taking n evenly spaced samples with period d and connecting the samples via a polygonal path. In other words, we compute the points $(kd, f(kd))$ for each integer $k \in \{0, \ldots, n\}$ and draw line segments connecting adjacent points. Elementary signal processing tells us that this approach is bound to fail. If we sample below the *Nyquist limit* of twice the highest frequency in our signal, we will not be able to reproduce the signal accurately [3]. Since the frequency of f increases without bound, we know that eventually any fixed sampling rate will be inadequate.

When we sample below the Nyquist limit, we get *aliasing*: phantom low-frequency signals derived from the original high-frequency information. What's interesting here is that the aliases themselves exhibit a clear pattern. Figure 3 shows sampled plots of $f(x)$. The plots show that as the frequency of the function increases, the apparent frequency of the aliased signal oscillates up and down at a fixed rate. When the apparent frequency drops to zero, we get a kind of "nodal point", a region of relative calm.

We can use Fourier analysis to understand the behaviour of $f(x)$ when sampled at different rates. A complete analysis is beyond the scope of this paper, but it can be shown that when reconstructed from samples at an even spacing of d, the apparent frequency of $f(x)$ oscillates between 0 and $1/2d$. Furthermore, the nodal points occur precisely when x is a multiple of π/d.

A similar analysis applies to the function $F(x, y) = \sin(x^2 + y^2)$ in the plane. The frequency increases linearly along every ray leaving the origin. In theory, the graph of F should resemble ever tighter concentric rings. But when sampled, we encounter the same problem as in the one-dimensional case – the frequency of the function rapidly exceeds the sampling rate, and aliasing sets in.

In two dimensions, we encounter an additional phenomenon: the nodal points occur in a square lattice. They must line up this way because of the underlying square sampling grid. The same pattern of oscillating frequencies must be visible around every nodal point, which is only possible if they lie in a grid.

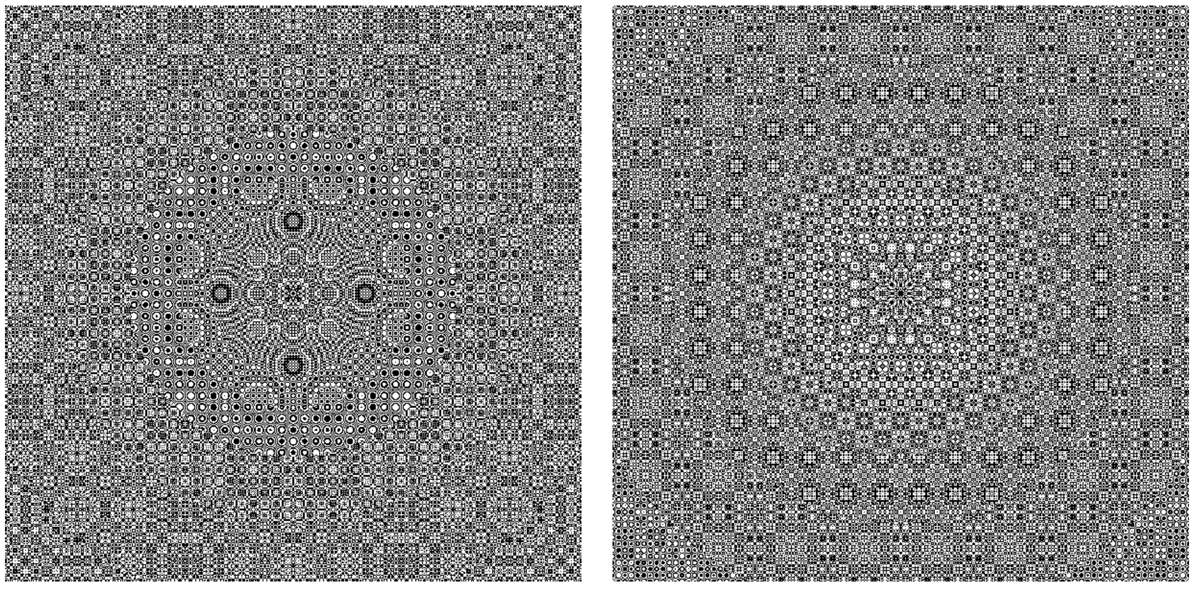

Figure 2 Two surprising images produced from a Voronoi diagram of a degenerate set of generators. Both images are renderings of the region $[-10, 10] \times [-10, 10]$ in the plane. They are rendered at resolutions of 600×600 in (a) and 601×601 in (b).

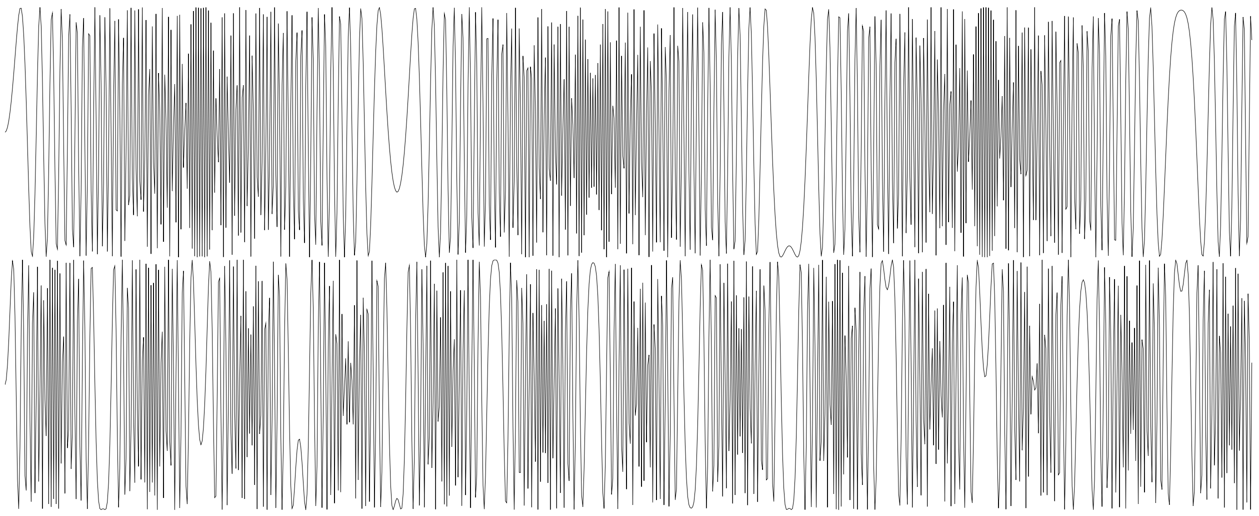

Figure 3 Sampled plots of the function $f(x) = \sin(x^2)$. The top plot shows the function from $x = 0$ to $x = 100$ with a distance between samples of 0.1. The bottom plot goes to $x = 200$ with distance 0.2.

Figure 4 shows several sample renderings of $F(x, y)$. In these images, each pixel is associated with a point (x, y) in the plane. The value of $F(x, y)$ is rescaled to $[0, 1]$ and mapped to a grayscale value for that pixel. Altering the sampling rate produces a continuum of widely varying images, of which I can only offer a few examples. As we zoom out from the origin, nodal points form. They have varying shades of gray in their centres. Eventually, the nodal points merge to become a low-frequency image of a single concentric ring at a much larger scale. The process then repeats.

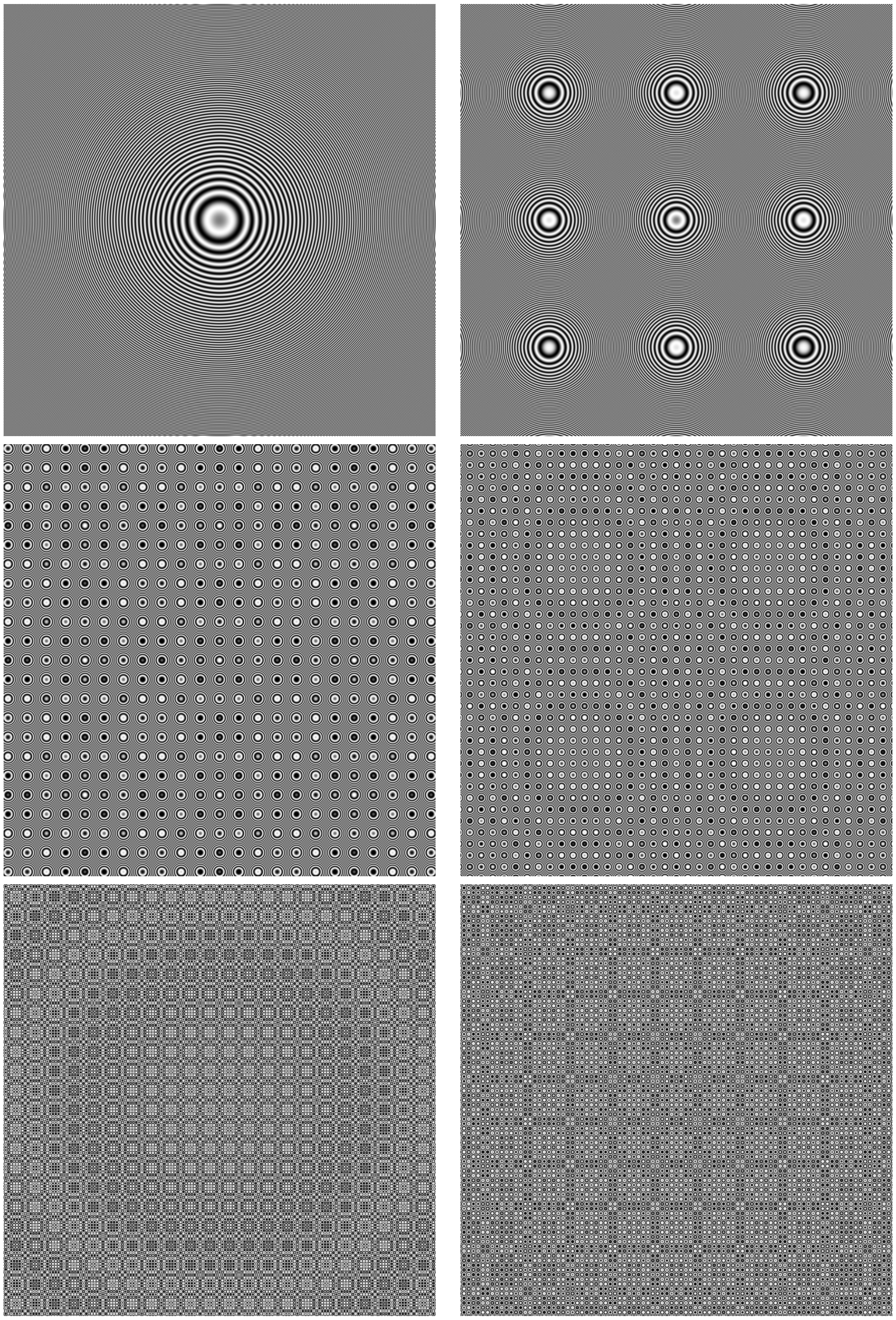

Figure 4 Example images formed by sampling the function $F(x, y) = \sin(x^2 + y^2)$ at various rates. In the top left image, aliases are just beginning to form.

Note that if $h(x)$ is any periodic function, then $h(x^2)$ will exhibit the same pattern of aliasing when sampled at a fixed rate, and $h(x^2 + y^2)$ will generate images very similar to the ones in Figure 4.

3. The bathtub algorithm and dot-matrix holograms

Patterns of this sort have been discovered and rediscovered many times in the history of computer graphics. The first mention in print seems to be by A.K. Dewdney nearly twenty years ago [1]. In his Computer Recreations column, he discusses John E. Connett's accidental discovery of the pattern while experimenting with a program originally intended to draw the Mandelbrot set. Connett truncates $x^2 + y^2$ to an integer, and assigns black or white to a pixel depending on whether the integer is even or odd. The periodic function in this case is simply $h(x) = \lfloor x \rfloor (\mathrm{mod}\ 2)$, and so Connett's patterns closely resemble those of the previous section. Dewdney named Connett's algorithm the CIRCLE^2 algorithm. It lives on under that name as a class of fractal in the popular software Fractint (though it is not, technically speaking, a fractal).

At almost exactly the same time, Brian Hayes published an article about what he called "dot-matrix holograms" [2]. He stumbled onto the aliasing patterns while writing a program to print contour maps of surfaces on a dot-matrix printer. It just happens that as a test surface he chose a paraboloid, expressed via the formula $z = x^2 + y^2$. Plotting a contour line at regular values of z is equivalent to mapping z through a periodic function.

Hayes called his approach the "bathtub algorithm", since it was if the surface being rendered were placed in a bathtub and rings allowed to form on it for a set of water heights. And because he was only able to view the generated images using his printer, he called them "dot-matrix holograms" due to the interference patterns that form.

Hayes points out that the lattice arrangement of nodal points (which he calls "bull's-eyes") follows from the underlying square sampling grid. He goes on to suggest that on a device with a hexagonal arrangement of pixels, the nodal points would repeat in a hexagonal lattice. Although we still don't have easy access to devices with hexagonal pixels, we can nevertheless draw a picture made from a honeycomb of tiny hexagons, and look at it using the high resolution available on today's printers. The results are striking, perhaps because of their novelty in our rectilinear world. Several examples that confirm Hayes's hypothesis are given in Figure 5.

These patterns also turn up several times in more-or-less disguised forms in Pickover's anthology of computer-generated patterns [5], and no doubt in many other places as well.

4. Voronoi patterns

Let us now return to the discovery that motivated this exploration: the degenerate Voronoi patterns of Section 1. Can we reproduce them explicitly, in a controlled way? Comparing the designs in Figures 2 and 4, it seems as if each of the concentric annuli produced in the Voronoi patterns contains a fragment of a function like those of Section 2. We should therefore look for some hint of a function of the form $h(x^2)$, where h is periodic.

I will note that when translated into a C program, the pseudocode in Section 1 does not always produce interesting patterns. The patterns appear only when the program is run on Intel x86 computers, and only under certain combinations of compiler switches. We must therefore suspect that the pattern is highly architecture dependent.

The answer to the riddle lies in the limited precision available for representing real numbers on

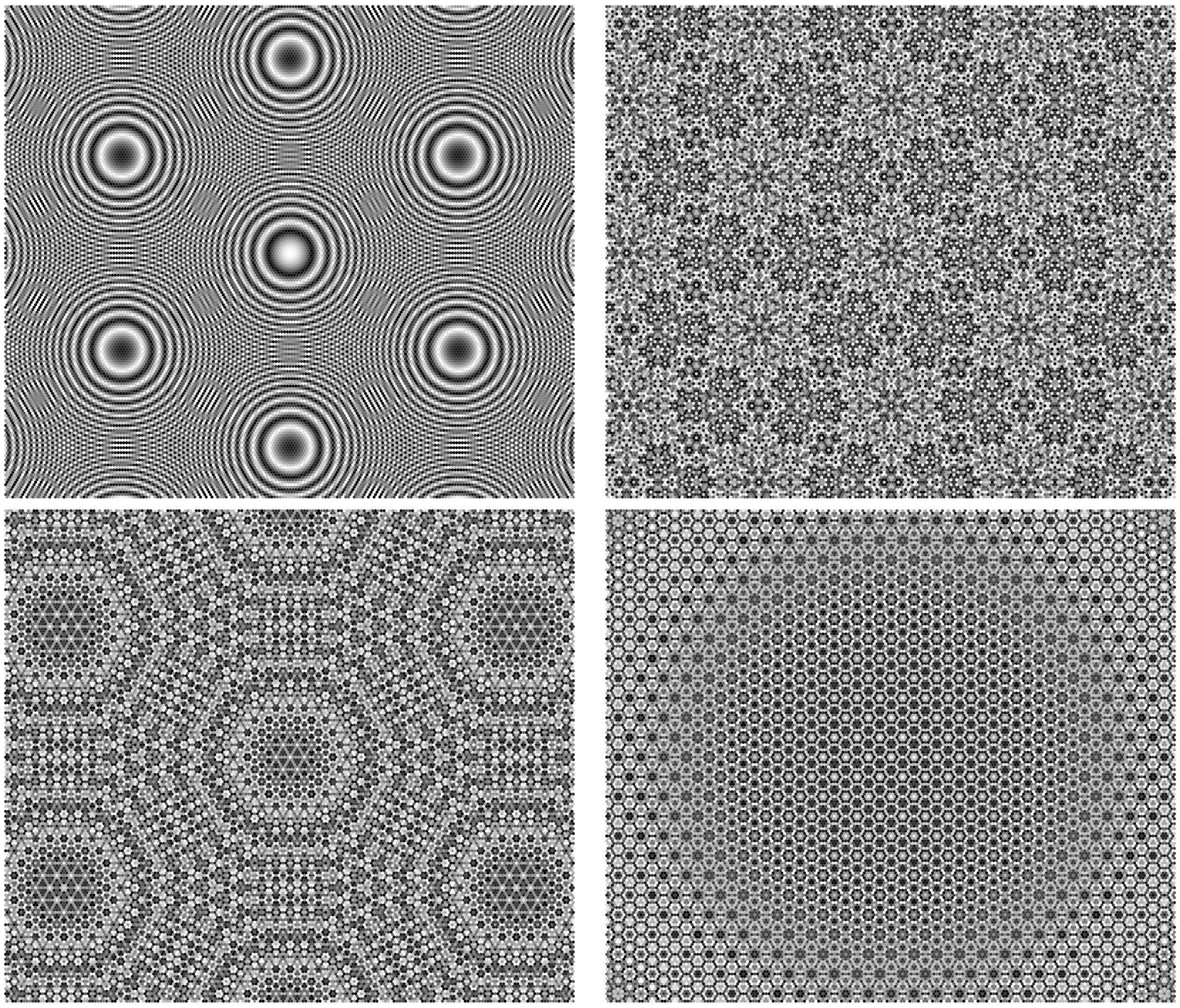

Figure 5 Example images formed by sampling the function $F(x, y) = \sin(x^2 + y^2)$ at various rates on a hexagonal grid.

computers. Computers use a floating-point representation, where a number is decomposed into a product of a power of two and a normalized fraction between $1/2$ and 1. On an Intel Pentium processor, 80 bits are used in the arithmetic unit while computing intermediate results, but only 64 bits are typically used to store results of computations in memory. Therefore, when d is compared to $x^2 + y^2$ in the pseudocode, the two values are not necessarily identical, because $x^2 + y^2$ had its precision reduced when it was stored in the variable d. Most importantly, when precision is reduced, numbers are rounded and not simply truncated. The comparison $x^2 + y^2 < d$ succeeds or fails depending on whether the low-order bits of d cause it to get rounded up or down, making it slightly larger or smaller than $x^2 + y^2$. And therein lies our periodic function. Numbers are rounded alternately up and down with unit frequency.

When $x^2 + y^2$ passes through a power of two, the position at which rounding occurs moves over by one bit in the representation of the numbers. This change in the rounding behaviour creates the annuli, each of which resembles a different sampling of the function $\sin(x^2 + y^2)$. Because the pattern shifts as the magnitude of the numbers shifts, the pattern has structure at every scale to within the limits of floating-point precision. These Voronoi patterns therefore have a more legitimate claim to being fractals than the CIRCLE^2

patterns they resemble, since the latter ultimately bottom out in an unaliased image.

Once we know what is causing the Voronoi patterns, it becomes easy to reproduce the behaviour on any floating-point architecture. We must explicitly request types that force the rounding to occur correctly. The easiest approach is to use the C types `float` and `double`, which have 32 and 64 bits of precision. We compute $x^2 + y^2$ as a `double` and also convert the result into a `float`. We can compare the two as before. The result does not have quite as much precision as the comparison of 64 and 80 bit floating-point numbers, but the precision is more than adequate for rendering images. These patterns can also be rendered directly on the processors of modern graphics cards using a pixel-level "fragment program".

Figure 6 gives a few examples of Voronoi patterns. The number of possible images that can be produced this way is very large; an interactive program the best way to explore this space of designs.

We can also use the C library function `frexp` to produce a grayscale version of the pattern. This function returns the exponent and normalized fraction stored internally for the floating-point number. If we compute $x^2 + y^2$ as both a `float` and a `double`, then we can apply `frexp` to the difference between the two values and use the normalized fraction part as a grayscale value. Two examples produced this way appear in Figure 7. When the same region of the plane is rendered using the rounding method above and the `frexp` method, the resulting pictures are very similar, confirming that we have explained the mathematical structure of these Voronoi patterns.

5. Conclusions

The patterns described in this paper are based on a curious mix of the abstract mathematics of Fourier analysis and the practicalities of computer architecture. They are often fascinating and surprising, if somewhat idiosyncratic.

As a researcher in computer graphics, I found it difficult to analyze these patterns. I would claim that this is because we are trained in computer graphics that aliasing can only be regarded as a problem, as an artifact to be suppressed or eliminated. There are few opportunities to celebrate aliasing and study the fragile patterns it brings into being.

Acknowledgments

Thanks to Michael McCool for helpful discussions about signal processing.

References

[1] A.K. Dewdney. Wallpaper for the mind: computer images that are almost, but not quite, repetitive. *Scientific American*, 253:14–23, September 1986.

[2] Brian Hayes. On the bathtub algorithm for dot-matrix holograms. *Computer Language*, 3:21–32, October 1986.

[3] Robert J. Marks II. *Introduction to Shannon Sampling and Interpolation Theory*. Springer-Verlag, 1991.

[4] Craig S. Kaplan. Voronoi diagrams and ornamental design. In Nathaniel A. Friedman and Javier Barralo, editors, *Proceedings of ISAMA 99*, 1999.

[5] Clifford A. Pickover. *The Pattern Book: Fractals, Art, and Nature*. World Scientific, 1995.

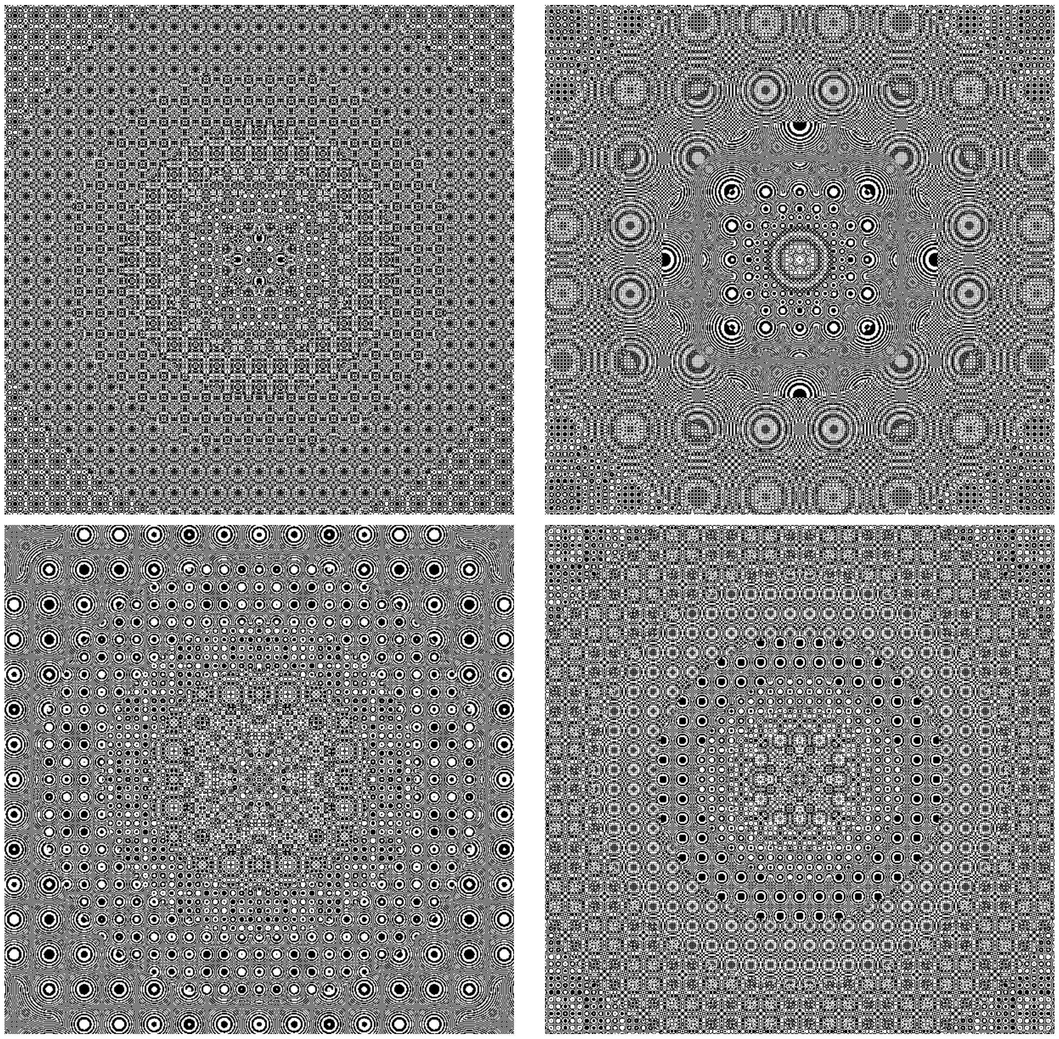

Figure 6 More examples of Voronoi patterns rendered using the simple pseudocode given in Section 1.

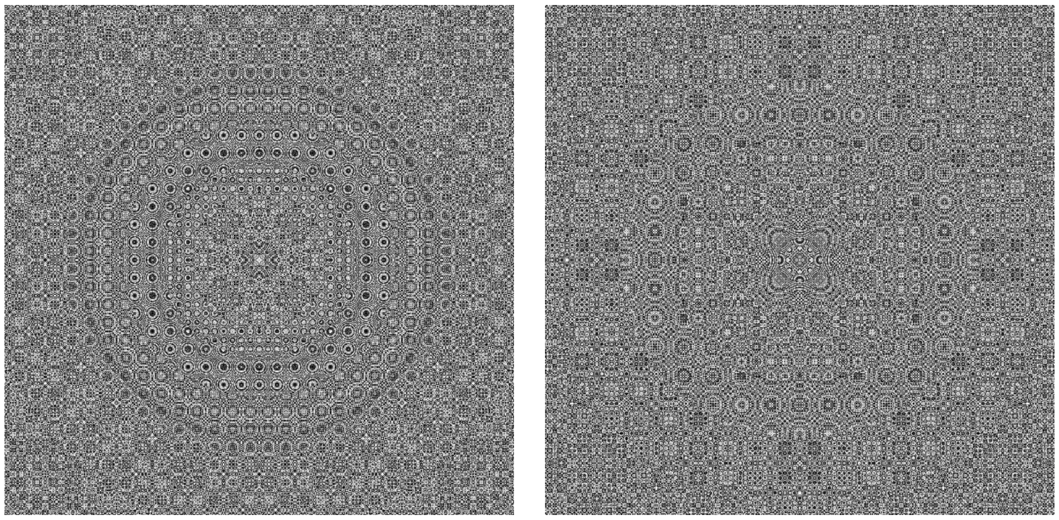

Figure 7 Examples of grayscale Voronoi patterns rendered using the C library function `frexp`, as explained in Section 4.