Linear Pixel Shuffling for Image Processing, an Introduction*

Peter G. Anderson

Computer Science Department
Rochester Institute of Technology
Rochester, New York 14623-0887
pga@cs.rit.edu

April 1993

Abstract

We present a method of ordering pixels (the elements of a rectangular matrix) based on an arithmetic progression with wrap-around (modular arithmetic). For appropriate choices of the progression's parameters based on a generalization of Fibonacci numbers and the golden mean, we achieve uniformly distributed collections of pixels formed by intervals of the pixel progression or "shuffle."

We illustrate this uniformity with a novel approach to progressive rendering of a synthetic image, and we note several opportunities for applications to other areas of image processing.

Introduction

The traditional raster-scan (row-order) ordering of an image's pixels can be inconvenient (waiting time for a computer graphics image to become evident) and can produce visually disturbing artifacts (television flicker; "worms" in half toned images using error diffusion[1]).

Alternative pixel orderings can improve this situation: interleaving scan lines removes television flicker; space-filling curves avoid error-diffusion's worms;[2, 3] pseudo-random pixel sequencing permits rapid graphics previewing.

Linear pixel shuffling is a *quasi-random*[4] ordering of pixels—a technique with many of the benefits of pseudo-random ordering, but described by an extremely simple rule based on an arithmetic progression. The ordering is designed so that when some fraction, a, where $0 \leq a \leq 1$, of the total number of pixels has been chosen, every sub-rectangle of size $w \times h$ contains nearly awh chosen pixels. That is, the chosen pixels are always smoothly distributed over the whole image.

The idea underlying these permutations is that the sequence of integers,

$$n_k = kA\%B, \text{ for } k = 0, \ldots, B-1$$

(% denotes the integer remaindering operation) is a permutation of the numbers $0, \ldots, B-1$ if and only if the greatest common divisor of A and B is 1 (in other words, *A and B are relatively prime*). The permutation is "most uniformly distributed" when the ratio, A/B, approximates the golden mean,[5] $\tau = (\sqrt{5}-1)/2$. (The best approximations are when A and B are two successive Fibonacci numbers.[6, 7, 8]) We have developed an analogous point, $\overline{\tau} \in R^2$, which allows us to construct uniformly distributed sequences of points in a rectangle, i.e., a permutation

*This article appears in *The Journal of Electronic Imaging*, April, 1993, pp. 147-154.

```
for every pixel (x,y)
    plot f(x,y) at (x,y);
```

Program 1. Generic rendering.

```
for( x = 0; x < A; x++ )
    for( y = 0; y < B; y++ )
        plot f(x,y) at (x,y);
```

Program 2. Specifying the scan lines.

of pixels.[9]

In the present note, we illustrate this method with a problem from computer graphics. We show how to use linear pixel shuffling to write graphics rendering programs to produce images that evolve from low to high resolution while always filling the entire screen. An image that may take several minutes to render is previewed in a few seconds.

There is no image degradation for using our pixel order, and only an insignificant time penalty.

The rendering process is continuous and smooth, and an acceptable image (in the sense of lossy compression) may be taken after any number of pixels have been rendered. This is in contrast to other progressive rendering schemes which typically give a sequence of approximate, low resolution images corresponding to a power of 2 or 4 pixels (averaged or subsampled).[10] If another pixel-count stopping point is used, the image's resolution will be obviously uneven—one side may have twice the resolution as the other.

Computer Graphics

Programs to create synthetic images (e.g., ray tracing, fractals) often have the general form shown in Program 1. Other general algorithms are commonly used, but they tend to be problem specific; our rendering algorithm is generic and problem independent.

Program 1's pixel ordering expressed in "`for every pixel (x,y)`" generally takes the form shown in Program 2, which specifies an image of size $A \times B$ pixels, which will draw the pixels in a specific order: first line $x = 0$, then line $x = 1$, an so on. Each line will be completely drawn, in order.

For programmers interactively involved in program development, this ordering can be painfully slow, since the relatively uninteresting edge pixels are drawn first. Two possible solutions that can be used during this development phase are to draw a scaled down version (say, 100×100 pixels rather than 1000×1000) or to choose pixel coordinates using a pseudo-random number generator. But both of these approaches entail a separate image-previewing stage; rendering must be started over to create the final, desired image; there is no smooth scaling up from previewing to rendering the full image.

Line Shuffling

To introduce linear pixel shuffling and its underlying use of the Fibonacci numbers, the golden mean, and some generalizations, we first exhibit the "one dimensional linear pixel shuffling" as a modification to Program 2. Program 3 uses a shuffled order of the lines (the variable `x`). This one dimensional shuffling is the ultimate generalization of television's line interleaving, in which the odd-numbered lines are drawn, then the even-numbered lines are drawn. Program 3 is particularly simple, and it may be exactly what is needed for some applications.

The parameters `F1` and `F2` in Program 3 are two adjacent numbers in the familiar list of Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, ... The Fibonacci sequence is defined as follows:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \quad \text{for} \quad n \geq 2. \end{aligned}$$

Two adjacent Fibonacci numbers are always relatively prime: if an integer d divides both F_n and F_{n+1}, then d divides their difference, F_{n-1}; continuing this argument, d must divide F_1, which is 1. So, in Program 3, the line number `x` will never be repeated. `x` successively takes on the values: 0, 610, 233, 843, 466, 89, 699, 322, 932, 555, 178, 788, 411, 34, 644, 267, 877, 500, 123, 733, 356, 966, ...

```
F1 = 610; F2 = 987;
x = 0;

for( k = 0; k < F2; k++ )
{
    if( x < A )
    {
        /* render line x */
        for( y = 0; y < B; y++ )
            plot f(x,y) at (x,y);
    }
    /* determine next line */
    x = (x + F1) % F2;
 }
```

Program 3. Plotting lines in shuffled order.

We chose the parameters `F1 = 610` and `F2 = 987` since 987 is approximately the number of scan lines on a computer's graphics screen. (This is reminiscent of programmers' practices of choosing 1,024 whenever they want "a thousand.")

This line shuffling is optimum (i.e., maximally uniform) using a simple arithmetic progression. The distance between two rendered scan lines, at any time in this process, can only be one of three adjacent Fibonacci numbers. Each new scan line, line `x`, will go into one of the largest gaps (of size, say, F_k) and cut that gap into two smaller gaps, F_{k-1} and F_{k-2}, roughly a 60-to-40 division.

Illuminate the Screen

The program in Program 3 is acceptable on monochromatic screens, but it leaves color screens too dark, too long. The modification shown as Program 4 immediately illuminates the entire screen.

In Program 4, every new pixel's color value is spread to the left, up to but not overwriting the nearest correctly plotted pixel.

This technique works because the sequence of `x` values is an arithmetic progression reduced modulo `F2`. By maintaining the value of the program variable `wide` (the width of the artificially fat lines) to

```
F1 = 610; F2 = 987;
x = 0;
wide = F2;

for( k = 0; k < F2; k++ )
{
    if( x < wide ) wide = x;
    if( x < A )
    {
        /* render line x */
        for( y = 0; y < B; y++ )
        {
            plot a line
            from (x-wide+1,y)
            to (x,y)
            of color f(x,y);
        }
    }

    /* determine next line */
    x = (x + F1) % F2;
}
```

Program 4. Plotting fat lines.

be the minimum distance achieved between `x` and `0`, we assure that any previously chosen line to the left of `x` is at least `wide` pixels to the left of `x`.

In the following sections, we generalize this to two dimensions.

Shuffling Pixels

The line-shuffling Programs 3 and 4 are particularly simple to code, and they do not entail any significant run-time overhead (the remaindering operation can be implemented with a conditional subtraction). Line shuffling may be the best approach if there is some advantage to rendering an entire scan line all at once, as in some ray tracing procedures. But line shuffling is not the best way to spend computing effort to render the first, say, 10,000 pixels. So many pixels should be *all over the screen*, not confined to only 10 scan lines. Program 5 illustrates an alternative pixel order based on a generalization of Fibonacci numbers. (Mathematical justification for these numbers is given below).

The parameters `G1`, `G2`, as well as `increment` come from a modified Fibonacci sequence: 0, 1, 1, 1, 2, 3, 4, 6, 9, 13, 19, 28, 41, 60, 88, 129, 189, 277, 406, 595, 872, 1278, 1873, 2745, 4023, 5896, ... The modified rule [1] is:

$$\begin{aligned} G_0 &= 0 \\ G_1 &= 1 \\ G_2 &= 1 \\ G_n &= G_{n-1} + G_{n-3} \quad \text{for} \quad n \geq 3. \end{aligned}$$

To use these numbers as in Program 5, the program parameters, `G1` and `G2`, are two adjacent numbers G_n and G_{n+1} which are relatively prime (they do not always satisfy this rule). Current computer graphics screens are often approximately 1000×1000, and the parameters used in Program 5 are especially suitable for this shuffling.

[1] The Fibonacci number F_n expresses how many pairs of rabbits a single pair will yield after n months, assuming that every pair produces another pair every month, if they are at least two months old. The modified sequence solves the same problem for *modified rabbits*—those that wait until they are three months old to reproduce.

Artificially Fat Pixels

We can illuminate a larger portion of screen early by artificially enlarging the pixels to squares. As with the variable `wide` in Program 4, we maintain a `radius` to avoid overwriting a previous, correctly plotted pixel. One way to do this is shown in Program 6:

It is not our intention to suggest programming practices involving explicit constants as in Program 6, but rather to simply sketch out and present an idea. The three constants come about as follows. When `k` is 27,201, the pixel currently being rendered (the `k`th pixel) is, for the first time, so close to `(0,0)` that a square ("fat pixel") of radius larger than 2 would overwrite the previous, correctly rendered color at `(0,0)`. The other values are derived similarly. This technique allows us to cover the screen with 27,200 overlapping 7×7 pixels, quickly allowing us to see a rough but full image.

The parameters used in Program 6 (which were computed explicitly) illustrate the quality of our two parameters, 1278 and 1873. We begin plotting an image using "fat pixels," squares of side 7. Until we have plotted 27,201 pixels (over 1% of the pixels), every 7×7 square of pixels (`radius` $= 3$) in the image contains at most one plotted pixel; until we have plotted 152,692 pixels (over 6% of the pixels), every 5×5 square of pixels in the image contains at most one plotted pixel; and, until we have plotted 265,519 pixels (over 11% of the pixels), every 3×3 square of pixels in the image contains at most one plotted pixel.

This is uniform distribution with a vengeance!

Mathematical Underpinnings

One dimension

Representation of real numbers by continued fractions

$$\alpha = \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \frac{1}{\cdots}}}}$$

where the a_k are positive integers, called the *partial quotients*, yields best rational approximations (for a

given range of denominators) for continued fractions that are truncated after a finite number of terms.[7, 8] When truncation occurs just prior to a large a_n, the approximation is particularly good.

The continued fraction for the golden mean is

$$\tau = \frac{\sqrt{5}-1}{2} = \frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{\cdots}}}} \quad (1)$$

All the partial quotients are 1, which is as small as possible. This implies that the golden mean is the hardest real number to approximate with rationals.[7, 8, 6] The best approximations are ratios of Fibonacci numbers:

$$\tau = \frac{\sqrt{5}-1}{2} \approx \frac{F_{n-1}}{F_n}$$

This implies that the set of m points in the unit interval

$$S_m = \{z_k : 0 \leq k < m\}$$

where

$$z_k = \frac{kF_{n-1}\%F_n}{F_n}$$

is particularly uniformly distributed.[5] Suppose that, on the contrary, two members of S_m were very close together, say $z_p - z_q$ is close to 0 or 1. If $p > q$, $z_p - z_q = z_{p-q}$, and $|F_n - (p-q)F_{n-1}\%F_n|$ is a small integer. But this can only happen when $p - q$ is a large Fibonacci number.[11, 12] That is, two member of S_m will only be close together in case m is a large number—the points are crowded only when there are very many of them.

Another way to describe τ uses matrix-eigenvector-eigenvalue formulation:

$$\begin{aligned} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \tau \\ 1 \end{pmatrix} &= \begin{pmatrix} 1 \\ \tau+1 \end{pmatrix} \\ &= (\tau+1)\begin{pmatrix} \tau \\ 1 \end{pmatrix} \end{aligned} \quad (2)$$

This formulation is equivalent[13] to τ's continued fraction representation in equation (1).

The matrix in equation (2) is related to the Fibonacci numbers by[12]

$$\begin{aligned} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix} &= \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} \\ &\approx F_{n+1}\begin{pmatrix} \tau \\ 1 \end{pmatrix} \end{aligned} \quad (3)$$

Two dimensions

An image of dimensions $A \times B$ can have all of its pixels represented in the sequence

$$\overline{p}_k = (kC\%A, kC\%B), \;\; k = 0, \ldots, AB-1$$

if and only if the numbers, A, B, and C, are pairwise relatively prime. That is, we require that $Z_A \times Z_B$ be a cyclic group with generator (C, C).[14]

In order to locate good values for A, B, and C, we find $\overline{\tau} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ where we have a 3×3 matrix-eigenvector-eigenvalue equation:

$$\begin{aligned} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ 1 \end{pmatrix} &= \begin{pmatrix} \beta \\ 1 \\ \alpha+1 \end{pmatrix} \\ &= (\alpha+1)\begin{pmatrix} \alpha \\ \beta \\ 1 \end{pmatrix} \end{aligned} \quad (4)$$

The system (4) is equivalent to the pair of equations

$$\begin{aligned} \alpha(\alpha+1)^2 &= 1 \\ \alpha(\alpha+1) &= \beta \end{aligned} \quad (5)$$

The defining quadratic equation for τ is

$$\tau(\tau+1) = 1 \quad (6)$$

Because of the close similarity between the equations (5) and (6) we call $\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ *the two-dimensional golden mean.*[9]

We can approximate the eigenvector by

$$\begin{aligned} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} &= \begin{pmatrix} G_n \\ G_{n+1} \\ G_{n+2} \end{pmatrix} \\ &\approx G_{n+2}\begin{pmatrix} \alpha \\ \beta \\ 1 \end{pmatrix} \end{aligned} \quad (7)$$

where the sequence G_n was introduced above as a "generalized Fibonacci sequence." Compare equations (7) and (3).

The multiples of (G_n, G_{n+1}) in the group $Z_{G_{n+2}} \times Z_{G_{n+2}}$ are evenly distributed, but those multiples do not exhaust the group, since $Z_{G_{n+1}} \times Z_{G_{n+1}}$ is not a cyclic group. So, we recommend using $(G_{n+1} - G_n, G_{n+1} - G_n)$ as a generator of the group $Z_{G_n} \times Z_{G_{n+1}}$, in the cases where G_n and G_{n+1} are relatively prime.

This, as well as a geometrical line of reasoning,[9] gives us the pixel shuffling rules to generalize Programs 5 and 6.

A Graphics Example

We illustrate our linear pixel shuffling technique (Program 6) to render $\mathcal{M}$, Mandelbrot's set.[15, 16] $\mathcal{M}$ is a subset of the complex plane defined as follows. For every complex number c, let $f_c(z) = z^2 + c$. Then,

$$\mathcal{M} = \{c : \limsup_{n \to \infty} |f_c^n(0)| < \infty\} \tag{8}$$

The superscript n in equation (8) denotes function composition.

To test whether a complex number, c, belongs to $\mathcal{M}$, initialize a complex variable z to 0, and iterate $z \leftarrow f_c(z)$. If some iterate exceeds 2 in magnitude, then $c \notin \mathcal{M}$. In Figures 1–3, if $|z| < 2$ for each of 170 iterations, we say $c \in \mathcal{M}$. (The higher we set the maximum allowable iterations, the better image we will generate, but the time to render the image will grow commensurately.)

The final ("99%") image took approximately fifteen minutes to render on a SUN SPARCstation SLC. However, the image available after only 1% of that time (see Figure 1) with the *fat shuffled pixels method* of Program 6 clearly indicates the image that will eventually be rendered. After a minute, 8% of the pixels have been computed, and a large amount of detail is visible (Figure 2). Then the user can decide to wait for the fully resolved image (Figure 3).

Image Processing Applications

In addition to the computer graphics "rapid rendering" application detailed above, there is a variety of image processing applications for linear pixel shuffling.

The applications in this section represent work in progress.[16]

Image data-bank browsing

When images are sent over limited bandwidths, the pixels or scan lines can be transmitted in the order suggested for graphics rendering. Users can decide quickly to wait for a full image or to skip to another.

Image compression

A compression technique for binary images suggests itself immediately from the "shuffled fat pixels" method for computer graphics. We give a brief sketch here.

We proceed as though we intend to *copy* an image locally. The copy will be built up in a manner similar to the development of the rendered image image in Figures 1–3.

First initialize the image copy to, say, all white. Then, we copy each pixel (in the order of Program 6) from the original image to the copy image as follows. If the copy image's current color at the pixel in question is the same as the original image's, do nothing; if the two colors are different, then paint a "fat pixel" in the copy image (as in Program 6), and record (i.e., store or transmit) that this change was done.

The recording need only provide the pixel's identification, `k`, or the increment, Δ`k`, since the last changed pixel.

An advantage of this compression algorithm is that one could truncate the sequence at any arbitrary point and have a lossy compression that may be of reasonable image quality; that is, a prefix of the representation is a representation for a lower resolution image. The type of resolution loss one would have is indicated by example in the image of Figure 2 as a lossy version of Figure 3.

Image morphology

The image morphology filter *opening*[17] is both important and time consuming. The *opening of the set S using the structuring element E* is given by

$$S \circ E = \bigcup_{\overline{v} \in Z^2} \{E + \overline{v} : E + \overline{v} \subset S\} \tag{9}$$

S and E are subsets of Z^2. S is usually the set of black pixels in an image, and E is a small set—such as a disk, square, or line—taking a role similar to that of a convolution kernel. The summation operator in (9) is defined by

$$E + \overline{v} = \{\overline{w} + \overline{v} : \overline{w} \in E\}$$

In practice, the set S is bounded, and $\overline{v}$ ranges over a rectangle of pixels. Therefore, we propose that

$$S \circ E \approx \bigcup_{k=0}^{n} \{E + \overline{p}_k : E + \overline{p}_k \subset S\} \tag{10}$$

In (10), $\overline{p}_k$ is our sequence of linearly shuffled pixels. The stopping point n can be determined by some rule so that the approximation is acceptable.

The *dilation of S by E*,

$$\begin{aligned} S \oplus E &= \{\overline{w} + \overline{v} : \overline{w} \in S, \overline{v} \in E\} \\ &= \bigcup_{\overline{v} \in Z^2} \{S + \overline{v} : \overline{v} \in E\} \end{aligned}$$

can be similarly rapidly approximated.

The erosion of S by E can be defined, using dilation,

$$S \ominus E = (S^c \oplus (-E))^c$$

where S^c is the complement of S and

$$-E = \{-\overline{v} : \overline{v} \in E\}$$

Consequently, erosion can be rapidly estimated. A wide variety of morphological filters can yield to this technique.

Monte Carlo integration

This application, although not specifically an image processing application, may provide a touchstone for the uniformity or equidistributivity properties of τ and $\overline{\tau} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$.

An integral can be estimated by sampling a function at a sequence of points, $x_1, x_2, x_3, \ldots$:

$$\int_S f(x)dx \approx \frac{1}{N} \sum_{k=1}^{N} f(x_k)$$

For functions of one variable and S the unit interval, let $x_k = \{k\tau\}$, where $\{a\} = a - [a]$, the fractional part of a, or "a modulo one." For functions of two variables and S the unit square, let $\overline{x}_k = (\{k\alpha\}, \{k\beta\})$.

When the function's sample points are chosen using a "white noise" pseudo-random number generator, such as *drand48()*, the error for the integral approximation is $\mathcal{O}(N^{-1/2})$. Using equidistributed "quasi-random numbers," the error[18, 4, 19] is $\mathcal{O}(N^{-1} \log N)$.

Acknowledgements

I am very grateful for the activities of many of my graduate computer science students at RIT: for their masters projects, Anna Ting developed a SUN user interface, Steve Mongelli an MS/DOS interface, and Norm Wright and Randy Charlick developed algebraic ray tracing to investigate and demonstrate these ideas; Dave Kavanagh created a demonstration video on the Amiga; Alan Swires experimented with the halftone algorithm; and Marc Cannava showed how image morphology operators can exploit linear pixel shuffling.

References

[1] R. Ulichney, *Digital Halftoning*, The MIT Press, Cambridge, MA (1987).

[2] L. Velho and J. de Miranda Gomes, "Digital halftoning with space filling curves," *Computer Graphics*, Vol. 25, No. 4, July (1991), pp. 81-90.

[3] I. H. Witten and M. Neal, "Using Peano curves for bilevel display of continuous tone images," *IEEE CG&A*, May 1982, pp. 47-52.

[4] H. Niederreiter, "Quasi Monte Carlo Methods and Pseudo-Random Numbers," *Bulletin of the AMS,* Vol. 84, No. 6, Nov. (1978), pp. 957-1041.

[5] D. E. Knuth, *The Art of Computer Programming, Volume 3, Sorting and Searching,* Addison-Wesley, Reading, MA, page 511 (1975).

[6] J. W. S. Cassels, *An Introduction to Diophantine Approximation,* Cambridge Tracts in Mathematics and Mathematical Physics, No. 45, Cambridge University Press, Cambridge (1957).

[7] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers,* Fourth Edition, Oxford University Press, Oxford (1968).

[8] A. Khinchin, *Continued Fractions*, translated by B. V. Gnedenko, University of Chicago Press, Chicago, IL (1964).

[9] P. G. Anderson, "Multidimensional golden means," *Proc. 1992 Conf. on Fibonacci Numbers & Their Applications*, G. E. Bergum, *et al*, eds., Klewer Academic Publishers, Boston, MA (in press).

[10] F. S. Hill, Jr., S. Walker, Jr., and F. Gao, "Interactive image query system using progressive transmission," *SIGGRAPH Conference on Computer Graphics and Interactive Techniques*, pp. 323-330 (1983).

[11] P. G. Anderson, "A Fibonacci-Based Pseudo-Random Number Generator," *Proc. 1990 Conf. on Fibonacci Numbers & Their Applications.*, G. E. Bergum, A. N. Philippou, and A. F. Horodam (eds.), pp. 1-8. Klewer Academic Publishers, Boston, MA (1991).

[12] S. Vejda, *Fibonacci and Lucas Numbers and the Golden Section: Theory and Applications*, John Wiley and Sons, New York, NY (1989).

[13] S. Lang, *Introduction to Diophantine Approximations,* Addison-Wesley, Reading, MA (1966).

[14] G. Birkhoff and S. MacLane, *A Survey of Modern Algebra*, 4th edition, Macmillan Publishing Co., New York, NY (1977).

[15] H. Peitgen and D. Saupe (eds.), *The Science of Fractal Images*, Springer-Verlag, New York, NY (1988).

[16] P. G. Anderson, "Fast Rendering," *Computer Language,* vol. 10, no. 2, 40-48 (1993).

[17] E. R. Dougherty, *An Introduction to Morphological Image Processing*, SPIE Optical Engineering Press, Bellingham, WA (1992).

[18] U. Grenendar and W. Freiberger, *A Short Course in Computational Probability and Statistics,* Applied Mathematical Sciences 6, Springer Verlag, New York, NY (1971).

[19] S. K. Zaremba, ed., *Applications of Number Theory to Numerical Analysis*, Academic Press, New York, NY (1972).

```
G1 = 1278; G2 = 1873;
increment = G2 - G1;

x = 0;
y = 0;

for( k = 0; k < G1 * G2; k++ )
{
    if( ( x < A ) && ( y < B ) )
        plot f(x,y) at (x,y);

    /* determine next pixel */
    x = ( x + increment ) % G1;
    y = ( y + increment ) % G2;
}
```

Program 5. Pixel shuffling.

```
G1 = 1278; G2 = 1873;
increment = G2 - G1;

x = 0;
y = 0;
radius = 3;

for( k = 0; k < G1 * G2; k++ )
{
    if( ( x < A ) && ( y < B ) )
    {
        plot a square
        of side 2 * radius + 1
        of color f(x,y)
        centered at (x,y);
    }

    /* determine next pixel */
    x = ( x + increment ) % G1;
    y = ( y + increment ) % G2;

    /* update size of fat pixel */
    if( k == 27201 ) radius = 2;
    if( k == 152692 ) radius = 1;
    if( k == 265519 ) radius = 0;
}
```

Program 6. Fat shuffled pixels.

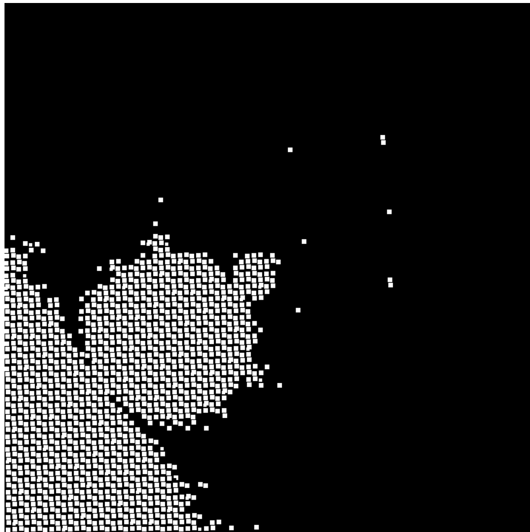

Figure 1: Mandelbrot's set 1% rendered.

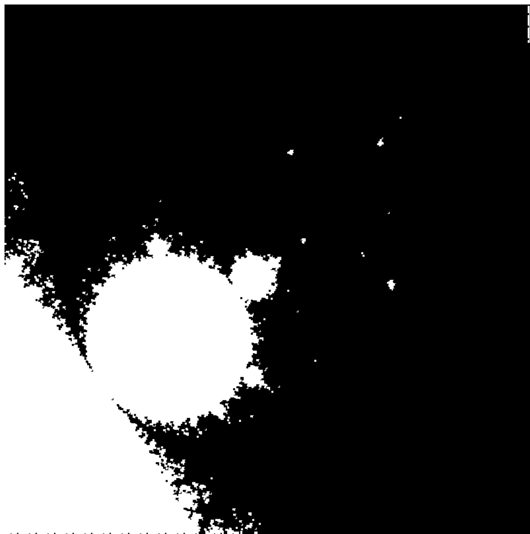

Figure 2: Mandelbrot's set 8% rendered. The white bar in the upper right of this picture indicates the width of the 8% strip that would show if the image were rendered in the usual row by row pixel order.

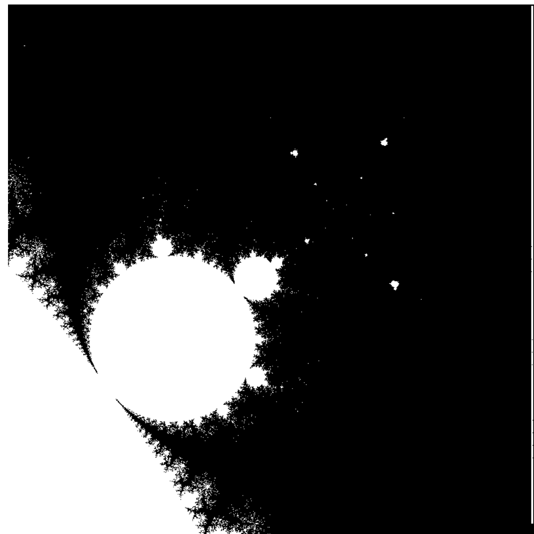

Figure 3: Mandelbrot's set 99% rendered.