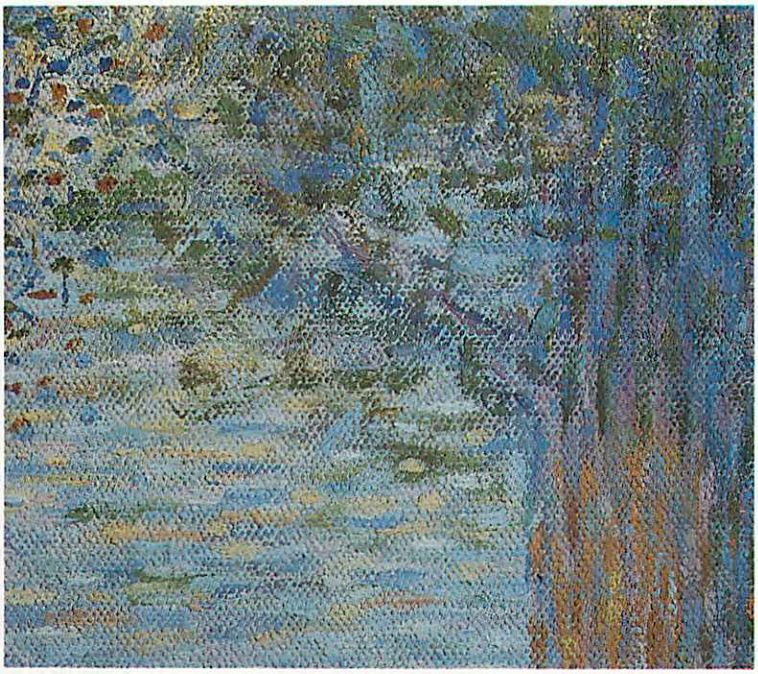
40,000 Points of Light

By Brian Hayes



Georges Seurat's Dimanche á la Grand Jatte *(detail)*

Georges Seurat's dotty paintings must have seemed weird and wonderful a hundred years ago. If he were painting them today, however, the world would likely find his pointillist principles rather humdrum. Viewers accustomed to seeing photographs reproduced by the halftone process, and who watch images projected through the shadow mask of a television screen, are no longer much surprised at the idea of composing a picture from dots of pure color.

Today it seems obvious that every possible image can be represented by an array of pixels. An interesting corollary is that a programming language for computer graphics really needs only one primitive concept: the point. Most graphics languages also provide a variety of higher-level constructs, such as lines, arcs and polygons, and perhaps even Bezier curves and cubic splines, but none of these facilities are strictly necessary. There is nothing one can draw with them that could not also be created by assembling a selection of individual pixels.

Here I shall describe a tiny programming language I call Seurat, which adopts a purely pointillist philosophy. Every image created in Seurat is defined as a locus of points on a plane. In the present implementation, the images are all black and white, but adding color would be a straightforward extension. I should state at the outset that Seurat is probably not of much practical use, at least for the current generation of graphics hardware. On the other hand, it allows certain graphic concepts to be expressed with particular ease and clarity. In those areas where the language works best, the programming statements needed to draw an object correspond closely to the mathematical statements that define it. And some of Seurat's failures—where expressing an idea turns out to be awkward, or where the finished image is not what it ought to be—are even more interesting than its successes.

What is a locus of points? Roughly speaking, it is the analogue in geometry of a set in logic or number theory. A locus comprises all those points that satisfy some specified condition or criterion. For example, if the criterion is that $x=y$, then the corresponding locus consists of all the points along a diagonal line. The locus of points where $y=0$ is made up of all the points on the x axis, and the locus where $x>0$ includes all the points in the half plane to the right of the y axis. The locus where $x=0$ and $y=0$ has just one member, namely the origin of the Cartesian plane. A locus can be discontinuous, or it can even be empty, as in the case of the locus defined by the simultaneous conditions $x>1$ and $x<1$.

My implementation of the Seurat programming language is built inside a Lisp system and employs Lisp syntax. Thus all expressions are fully parenthesized, and operations are given in prefix notation. The locus of points where $x=y$ is defined like this:

```
(locus diagonal (= x y))
```

not mean "burdensome for experts." As a user gains experience with a system, he/she should become faster at performing tasks, rather than being held back to the slowest possible level of interactions.

Menu/Mouse Interfaces

With the advent of personal computers, we have all come to expect graphical user interfaces. They have become a way of life, literally transforming the world of computing, making it more accessible to a large number of users. With the proliferation of mouse-driven applications, language-based or command-line-oriented interfaces have fallen by the wayside. While this development is in some cases entirely appropriate, for a large number of applications, language-based systems should not be overlooked. Language is, after all, one of our most powerful tools. Why should we not use it as an element of our new visualization environment?

Rendering

Rendering puts the visual in visualization, and usually receives the most attention by software developers (partly, I suspect, because of our own computer-graphics backgrounds). There are available a wide variety of useful techniques. Simple two-dimensional line graphs are quite effective at conveying the results of an experiment and should not be overlooked. Other times, contour plots, height plots or more complex graphics, such as particle advection or volumetric visualization, are most appropriate.

A number of companies claim to offer scientific visualization systems when all they really give us are traditional rendering tools. While these tools may be useful to researchers, scientists often use them for tasks for which they are not well suited.

A scientific renderer should concentrate on producing images which clearly convey the maximum amount of information. A scientific renderer should avoid making dangerous assumptions about data, particularly during the interpolation steps that form an integral part of most rendering algorithms. At the least, assumptions and interpolations should be documented so that researchers can better understand the output from rendering systems and prevent a misleading interpretation of scientific data.

Very simple graphical techniques can be used for scientific visualization. The goal is not to produce a realistic image, but one that conveys the relevant information. Progressive techniques, which render images incrementally, can be useful because they allow a researcher to get part of the answer quickly and decide whether the rest of the data set needs to be processed.

Expandability

New techniques in scientific visualization are being explored every day. As new techniques are tried and found to be useful, one would like to be able to integrate these new tools with those that already exist. This expandability is absolutely necessary to support research.

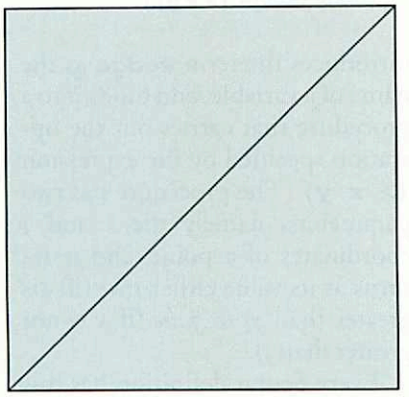
Expandability can be provided in many ways. Source-code availability allows the adaptation of software to the more specific needs of some applications. Other users may be satisfied with being able to call their own specific routines via "hooks" into the visualization software. Other systems provide complete languages for data retrieval and manipulation. Each of these options allows expansion of the visualization environment to areas the original designers may never have imagined.

Here **locus** is a keyword that introduces the definition of a locus, **diagonal** is a name by which this particular locus will be identified, and the expression **(= x y)** specifies the test that must be applied to a point to see if it is included in the locus.

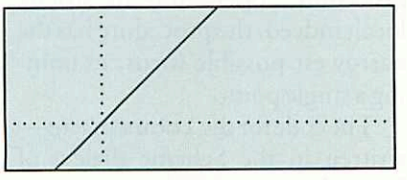
Defining a locus does not actually draw anything. To see a representation of the locus, it is necessary to issue another command:

```
(draw diagonal)
```

Here is the result of executing the **draw** command:



Of course the illustration does not exhibit the entire locus, which includes an infinite number of points along a line of infinite extent. By default, the **draw** command displays a region of 200 × 200 pixels centered on the origin, but other regions can be specified. Here is a different view of the same diagonal locus:



The graph was generated by the command:

```
(draw diagonal -50 -25 150 60)
```

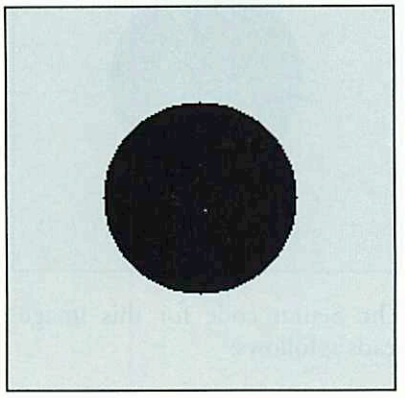
The numbers following the name of the locus are coordinates specifying a rectangular region to be displayed. For clarity, the *x* and *y* axes are also shown (as dotted lines); they too are defined as loci.

The ability to work easily with an infinite locus is one of the pleasanter features of Seurat. Most other graphics notations can accommodate only line *segments*; indeed, a "line" is often defined by giving its endpoints, which is not very helpful when you want to talk about a line that has no end. In Seurat, the complete mathematical line exists, and it is clipped to finite bounds only when the time comes to paint it on a screen or a page of finite size.

Another simple locus is defined by the expression:

```
(locus disk
  (<= (sqrt (+ (* x x) (* y y)))
      50))
```

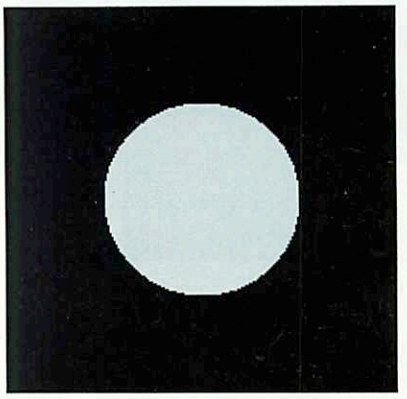
The locus consists of all points in the plane whose distance from the origin is less than 50; in other words, it is a disk with a radius of 50 pixels.



(We can gain a bit of efficiency by eliminating the square-root operation, which is relatively expensive. The criterion **(<= (+ (* x x) (* y y)) 2500)** defines the same locus.) The complementary locus—made up of all the points in the plane except those included in the **disk** locus—could be defined simply by replacing the **<=** operator with a **>** sign. The same end is achieved by the following definition:

```
(locus hole
    (not (disk x y)))
```

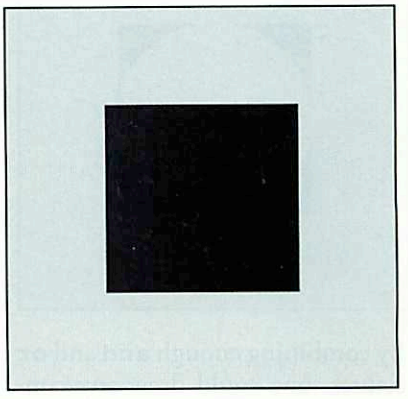
which makes explicit the complementary relation between the two loci: a point is a member of the **hole** locus if and only if it is not a member of the **disk** locus.



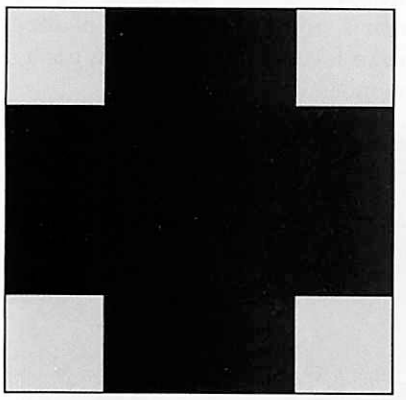
Clearly any locus can be inverted in this way. You need not know anything about the geometry of the original locus in order to write a definition for the inverse.

Creating a filled square is just a little more trouble than defining a circular disk:

```
(locus square
  (and (< (abs x) 50)
       (< (abs y) 50)))
```



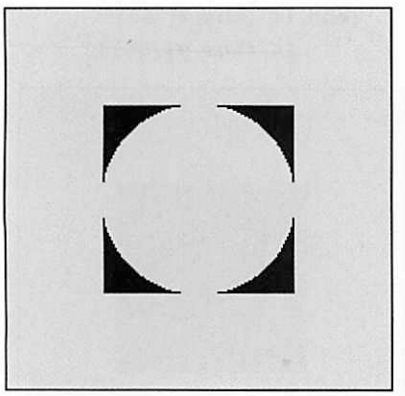
This time it is not the Pythagorean distance from the origin that must be less than 50 but the absolute value of the x and y coordinates. The constraints on the x and y values must be satisfied simultaneously, a requirement enforced by the **and** combining form. If an **or** form is substituted, the result is a different locus:



Program constructs such as **and** and **or** allow loci to be combined in a variety of ways. For example, the locus of all points that are members of **square** but are not members of **disk** is calculated by this expression:

```
(locus squarcle
   (and (square x y)
        (not (disk x y))))
```

The result looks like this:

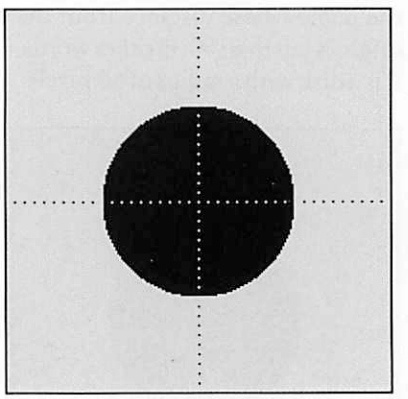


By combining enough **and** and **or** clauses, one could draw any conceivable image in Seurat. At worst, the definition of the image would simply be a giant **or** clause, containing a long list of **and** clauses, like these:

```
(or (and (= x 0) (= y 0))
    (and (= x 0) (= y 1))
    (and (= x 0) (= y 2)) ...)
```

In this way a Seurat program can control every pixel individually. In the standard 200×200 pixel neighborhood, a total of $2^{40,000}$ black-and-white images could be created.

The dotted-line axes that appear on one of the **diagonal** illustrations above were added to the image by means of an **or** clause. For a slightly different effect, we might try the exclusive-or relation, abbreviated **xor**. In the example below, the axes are black where the image is otherwise white, but on the black disk the axes are "dropped out," becoming lines made up of white dots.



The Seurat code for this image reads as follows:

```
(locus disk-with-axes
   (xor (axes x y)
        (disk x y)))
```

A pixel is turned on if it is a member of the **axes** locus or if it is a member of the **disk** locus, but not if it is a member of both loci.

The inner workings of the Seurat language are remarkably simple. The basic idea is to examine every pixel in the image area and ask whether or not it is a member of the locus being drawn; depending on the answer, the pixel is then set to either black or white. The apparatus for performing this survey of pixels consists of just two routines, which implement the **locus** and **draw** commands.

Locus is the keyword of a Lisp macro, which transforms a textual definition into an executable procedure. Thus the Seurat expression:

```
(locus wedge (> x y))
```

introduces the term **wedge** as the name of a variable, and binds it to a procedure that carries out the operation specified by the expression **(> x y)**. The procedure has two parameters, namely the x and y coordinates of a point, and it returns as its value either *true* (if x is greater than y) or *false* (if x is not greater than y).

Every Seurat definition has this same basic structure. The procedure created by the definition is invariably a predicate: a procedure whose returned value must be either *true* or *false*. The procedure's only duty is to determine whether or not a given point satisfies the relation set forth in the definition. Note that the procedure has no need to know anything about the overall geometry of the locus. The procedure's operations are strictly local; indeed, the procedure has the narrowest possible focus, examining a single point.

The code for the **locus** macro—written in the Scheme dialect of Lisp—is given in the listing on page 35. The same listing also includes a simplified version of the **draw** routine. (The simplification does not alter the basic structure of **draw**; it merely reduces clutter by

eliminating facilities for handling the optional extra arguments that specify a region to be displayed.)

The input to **draw** is a procedure—specifically a procedure of the kind created by the **locus** macro. A loop inside the **draw** routine steps through all the pixels in the displayed region and invokes the supplied procedure on each pair of coordinates. If the procedure returns a value of *true*, **draw** turns on the corresponding pixel; otherwise, the pixel remains off. Actually, the **draw** loop is a nested loop-within-a-loop: the outer loop progresses through rows of pixels, while the inner loop examines the pixels within each row. Both loops are defined by Scheme **do** forms.

In comparison with other graphics languages, what is most notable about the implementation of Seurat is what is missing from it. There are no algorithms for drawing lines or arcs or other geometric figures. All the knowledge of geometry is incorporated into the individual locus definitions.

Simplicity is Seurat's one virtue. Every image is defined in the same way and rendered by the same drawing mechanism. But the price for this simplicity is paid in efficiency.

Consider the locus defined by the expression:

```
(locus origin
  (and (= x 0) (= y 0)))
```

When this locus is plotted in the standard 200×200 neighborhood, exactly one pixel is activated (namely the pixel at the origin); nevertheless, the **draw** routine must examine all 40,000 pixels in the region, invoking the **origin** procedure on each pixel in turn. As the size of the displayed region increases, the computational burden grows still larger. With a 1,000 × 1,000 pixel display, there are a million pixels to be checked, no matter how few or how many will be turned on. A more efficient algorithm would require an amount of computation proportional to the number of *active* pixels, rather than proportional to the total number of pixels in the drawing area.

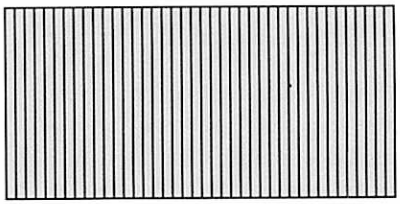
The cure for all such problems of speed and efficiency is well known: all we need is faster hardware and parallel processing. In this case the ideal solution would be a display system with one processor per pixel. The nested loops of the **draw** routine would then be eliminated altogether. A copy of the procedure implementing a locus would be supplied to each processor, and calculations for all the pixels would be carried out independently and simultaneously. (It is worth noting that this is one of those rare cases where adopting a highly parallel architecture simplifies a programming task rather than complicating it.)

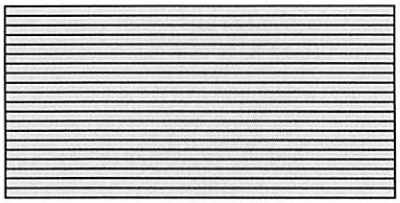
The idea of providing a processor for every pixel is not quite as outrageous as it might seem on first examination. Machines approaching the necessary scale of integration are being built already. Some models of the Connection Machine, for example, have 64,000 processors. Perhaps more to the point, an active-matrix liquid-crystal display can be regarded as a processor-per-pixel device, although the "processors" are extremely simple. (They consist of a single transistor.)

Several of the loci I have been exploring with Seurat exploit the **modulo** operation. They generate repetitive patterns that could potentially tile the entire plane. A simple example is defined by the expression:

```
(locus v-stripes
  (= (modulo x 5) 0))
```

The analogous wallpaper with horizontal stripes is created by substituting **y** for **x** in the formula. Here are the results.

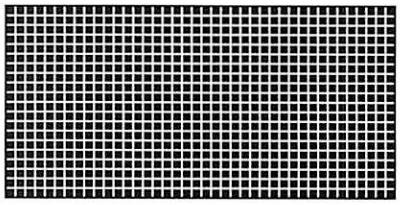




We can now invert and combine the two patterns, according to the formula:

```
(locus checks
 (and (not (v-stripes x y))
      (not (h-stripes x y))))
```

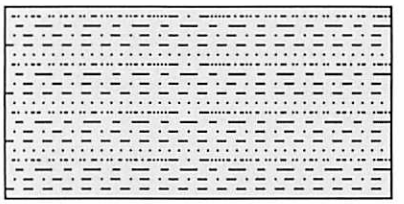
The result looks like this:



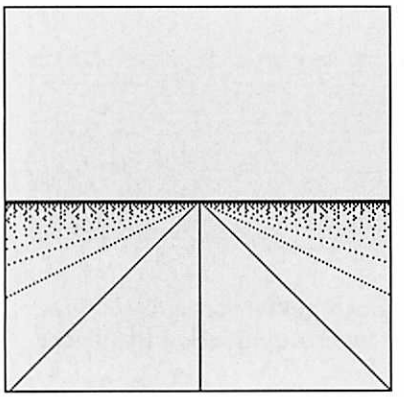
Another application of the **modulo** operation is in forming the dotted axes that appear in a few of the illustrations reproduced here. The code for the *x* axis is as follows:

```
(locus x-axis
  (and (= y 0)
       (= (modulo x 5) 0)))
```

Variations on the same technique give rise to more elaborate patterns of dots and dashes, as in these examples:



Here is still another pattern whose generating function relies on the **modulo** operator, although in a less-obvious way:



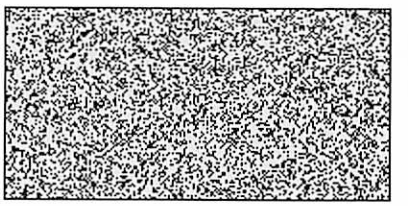
The Seurat definition that gives rise to this image is quite simple, and yet it is not easy to guess:

```
(locus orchard
 (or (= y 0)
     (=(modulo x (- y)) 0)))
```

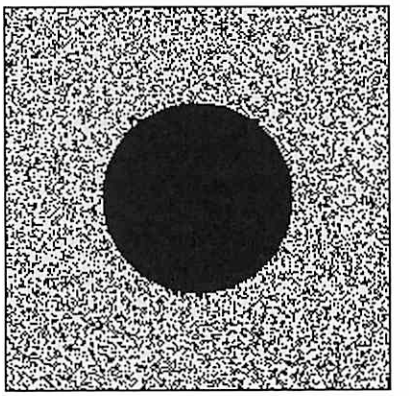
The locus consists of all those points on the plane where x is divisible by y. (Special provisions have to be made for the case of y=0, since an attempt to evaluate **(modulo x 0)** causes a divide-by-zero error. And the locus specifies **(- y)** rather than **y** alone so that Seurat will not turn the orchard upside-down.)

Another interesting class of loci are based on a pseudorandom function. Turning pixels on with a probability of one-third creates a pleasant mezzotint texture:

```
(locus mezzotint
    (= (random 3) 0))
```

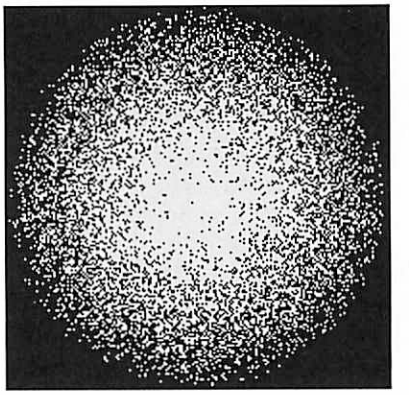


The texture can be applied to other figures by the usual methods of **and** and **or** combination:



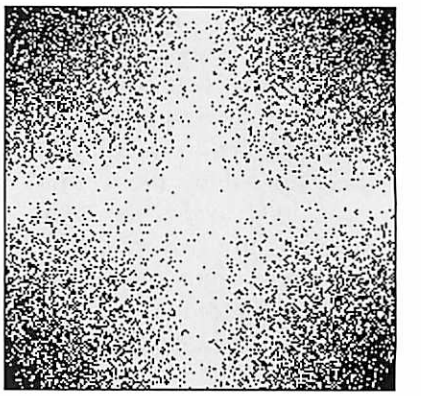
A density gradient forms when the probability of turning on a pixel varies over the visual field. Here the probability increases linearly with distance from the origin:

```
(locus circle-gradient
   (< (random 10000)
      (+ (* x x) (* y y))))
```



And in this case the density follows hyperbolic contours instead of circular ones:

```
(locus hyper-gradient
 (<(random 10000)(abs (*x y))))
```

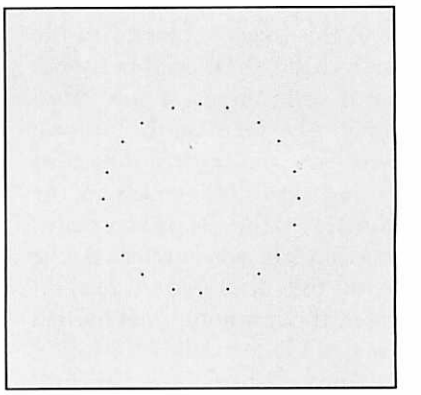


The images I have presented so far show those things that Seurat does well. But there are also areas where the language seems to make life harder rather than easier. The most fundamental problem is best revealed by means of a simple example.

The locus **disk** is made up of all the points that are either on or inside a circle, and the locus **hole** consists of all those points outside the same circle. The definition of **disk** employs the operator **<=**, and the definition of **hole** relies on **>**. It seems straightforward to draw the circle itself by constructing a similar expression with the operator **=**, as follows:

```
(locus circle
   (= (+ (* x x) (* y y))2500))
```

Unfortunately, the result of drawing this locus is not the circular form one might hope or expect to see:

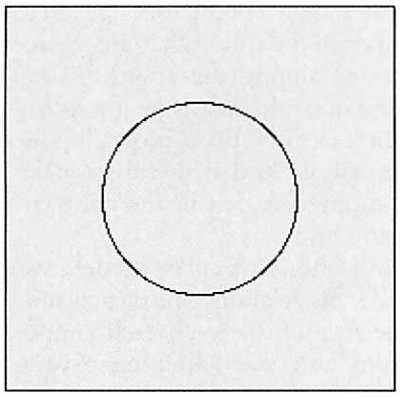


What has gone wrong here? Why does Seurat draw only a ring of bullet holes instead of a full circle? The problem turns out to be a kind of aliasing. I have been using the terms "point" and "pixel" as if they were interchangeable, but in fact there is an important distinction between these concepts. Points on the plane are infinitely dense, but pixels exist only at discrete intervals. There are infinitely many points on the circle, but very few of them happen to lie exactly at the coordinates of a pixel.

One approach to solving this problem is to adopt a more liberal definition of equality. In Scheme this is easily arranged; we can define an operator meaning "approximately equal to" as follows:

```
(define ≈
 (lambda (a b tolerance)
  (<= (abs (- a b)) tolerance)))
```

A predicate constructed with ≈ will return *true* if the arguments **a** and **b** differ by no more than **tolerance**. By choosing an appropriate value for **tolerance** we can now draw a fairly smooth circle:

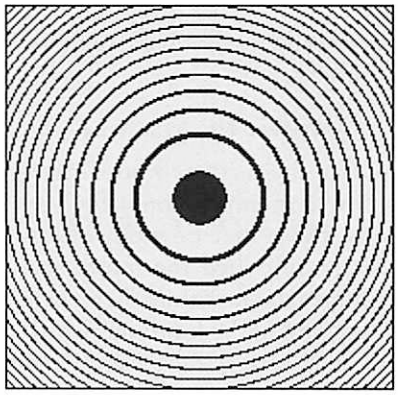


This one was done with **tolerance** set to 50. Thus the marked pixels include all those for which the square of the distance from the origin lies anywhere between 2,450 and 2,550.

But artificially thickening the lines in a picture can introduce problems of another kind. The locus **rings** ought to produce a series of concentric circles:

```
(locus rings
  (≈ (modulo (+ (* x x) (* y y))
             1000) 0 200))
```

When the locus is drawn, however, the black rings become progressively thicker toward the origin, and indeed the central annulus has filled in entirely:



This is not the result intended. The situation worsens as we attempt to represent more rings and finer lines. At this point the image is dominated by a false or aliased pattern; most of the real information has been lost. (On the other hand, the aliased pattern may well be more interesting than the real one.)

There are further remedies we might yet try. For example, we could create fat lines, as in the two illustrations above, and then shrink them again with one of the many thinning algorithms developed for optical pattern recognition. Another idea is to have nearby pixels communicate; thus all the pixels in a neighborhood could get together and decide which one is closest to a line or curve, and only that closest pixel would be activated. Such schemes are surely possible. On the other hand, if they were built into the Seurat interpreter, the language would no longer be so remarkably simple. The better strategy might be to introduce lines, circles and other geometric figures in their own right. Or perhaps the problem should be left for a later language to resolve: Klee or Kandinsky might have something to contribute, or maybe Miro.

```
(macro locus
        (lambda (definition)
              (list  define (cadr definition)
                     lambda (x y) , cadr definition)))))

(define draw
    (lambda (locus-procedure)
       (let ((xmin -100) (ymin -100) (xmax 100) (ymax 100)
             (do ((y ymin (+ y 1)))
                 ((> = y max))
                  (do ((x xmin (+ x 1)))
                    ((> = x max))
                      (if (locus-procedure x y)
                        (draw-point x y )))))))
```