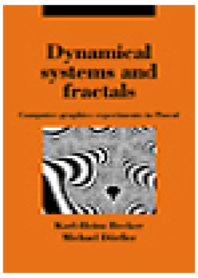


Dynamical Systems and Fractals

Computer Graphics Experiments with Pascal

Karl-Heinz Becker, Michael Dörfler, Translated by I. Stewart

Book DOI: http://dx.doi.org/10.1017/CBO9780511663031

Online ISBN: 9780511663031

Hardback ISBN: 9780521360258

Paperback ISBN: 9780521369107

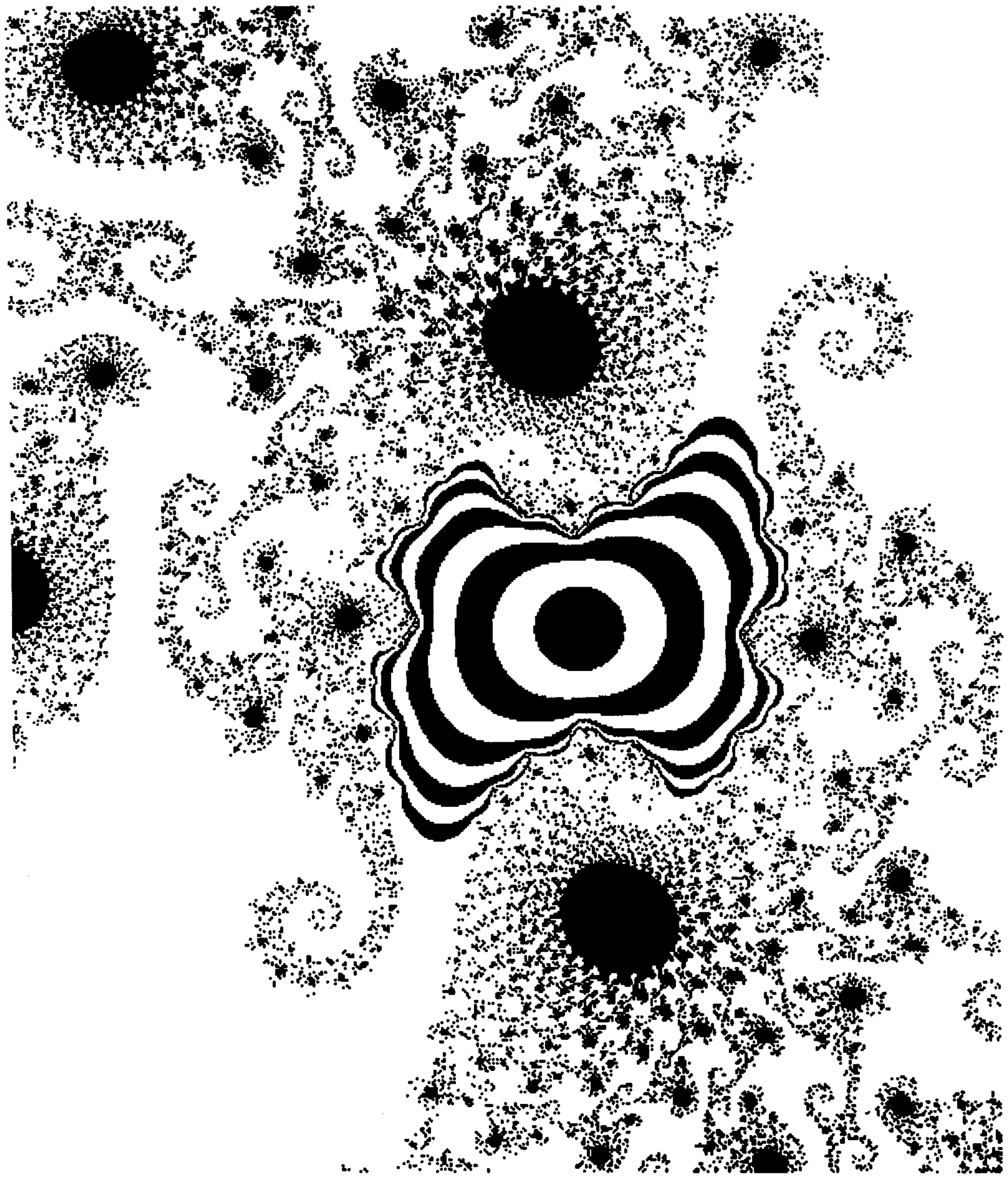
Chapter

7 - New Sights – new Insights pp. 179-202

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511663031.009

Cambridge University Press

7 New Sights – New Insights



Until now we have departed from the world of the ground-plan `Mapping` only in exceptional cases, but this chapter will show that the results of iterative calculations can be represented in other ways. The emphasis here should not be only on naked power: for the understanding of complicated relationships, different graphical methods of representation can also be used. If 'a picture is worth a thousand words', perhaps two pictures can make clear facts that cannot be expressed in words at all.

7.1 Up Hill and Down Dale

Among the most impressive achievements of computer graphics, which we encounter at every local or general election, are 3D pictures. Of course we all know that a video screen is flat, hence has only two dimensions. But by suitable choice of perspective, projection, motion, and other techniques, at least an impression of three-dimensionality can be created, as we know from cinema and television. The architectural and engineering professions employ Computer Aided Design (CAD) packages with 3D graphical input, which rapidly made an impact on television and newspapers. Although we certainly cannot compare our pictures with the products of major computer corporations of the 'Cray' class, at least we can give a few tips on how to generate pseudo-3D graphics, like those in §2.2.3.

The principle leans heavily on the mapping method of the previous chapter. The entire picture is thus divided into a series of parallel stripes. For each of them we work out the picture for a section of the 3D form. We join together the drawings of these sections, displaced upwards and to the side. We thus obtain, in a simple fashion, a 3D effect, but one without true perspective and without shadows. It also becomes apparent that there is no point in raising the iteration number too high. This of course helps to improve the computing time. To avoid searching through huge data sets, which must be checked to see which object is in front of another, we note the greatest height that occurs for each horizontal position on the screen. These points are if necessary drawn many times. For all computations concerned with iteration sequences, the iteration step should be the quantity that appears in the third direction. Two further quantities, usually the components of a complex number, form the basis of the drawing. In general we will denote these by x and y, and the third by z.

To begin with, we can generate a new pseudo-3D graphic for each picture in the previous chapters. In this type of representation Newton's method for an equation of third degree, which we know from Figure 4.3-5, generates Figure 7.1-1.

The central procedure bears the name `D3mapping`, which you will recognise because all new variables, procedures, etc. carry the prefix `D3` before their names.

The resulting picture is in a sense 'inclined' to the screen, and sticks out a bit on each side. To avoid cutting off the interesting parts of the picture, we can for instance be generous about the limits `Left` and `Right`. We have chosen a different possibility here,

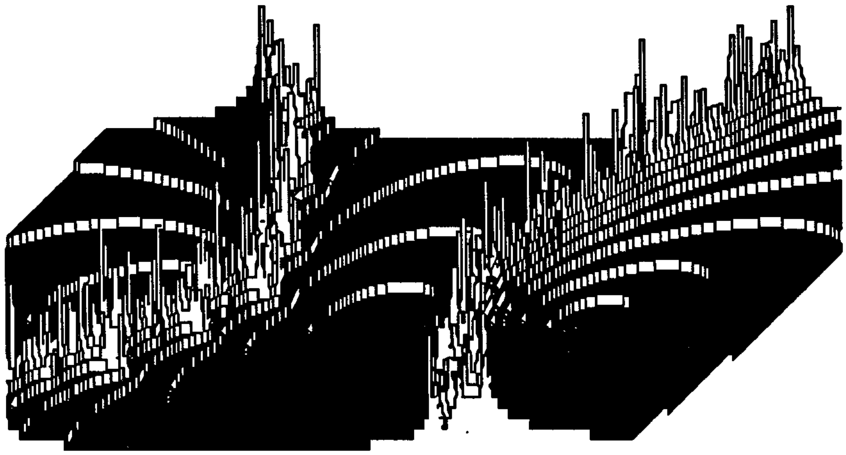


Figure 7.1-1 Boundaries between three attractors on the real axis.

in order to make this program as similar as possible to those already encountered. This gives rise to a somewhat modified denominator in the calculation of `deltaxPerPixel` and `deltayPerPixel`, and the upper limit of the `FOR`-loop.

The number `D3factor` expresses how strongly the figure is stretched in the vertical direction. The product `D3factor * maximalIteration` should amount to roughly a third of the screen size, that is, about 100 pixels.

The maximal coordinate of each vertical screen coordinate is stored in the array `D3max`. To enable this array to be passed to other procedures, a special type is defined for it. To begin with, the contents of this array are initialised with the value 0.

Program Fragment 7.1-1

```
PROCEDURE D3mapping;
  TYPE
    D3maxtype = ARRAY[0..Xscreen] OF integer;
  VAR
    D3max : D3maxtype;
    xRange, yRange, D3factor : integer;
    x, y, deltaxPerPixel, deltayPerPixel : real;

 (* here some local procedures are omitted *)

BEGIN
  D3factor := 100 DIV maximalIteration;
  FOR xRange := 0 TO xScreen DO
```

```
         D3max[xRange] := 0;
      deltaxPerPixel := (Right - Left) / (Xscreen - 100);
      deltayPerPixel := (Top - Bottom) / (Yscreen - 100);
      y := Bottom;
      FOR yRange := 0 to (Yscreen - 100) DO
      BEGIN
         x := Left;
         FOR xRange := 0 TO (Xscreen - 100) DO
         BEGIN
            dummy := D3ComputeAndTest (x, y, xRange, yRange);
            x := x + deltaxPerPixel;
         END;
         D3Draw (D3max);
         y := y + deltayPerPixel;
      END;
   END; (* D3mapping *)
```

As you see, two further procedures must be introduced: `D3Draw` and the functional procedure`D3ComputeAndTest`. The latter naturally has a lot in common with the functional procedure `ComputeAndTest`, which we have already met. Since the drawing is carried out in `D3Draw`, we must pass to it the coordinates of the currently computed point in (*x*,*y*)-space. Instead of deciding whether each individual point is to be drawn, we store the values corresponding to a given row, so that eventually we can draw the line in a single piece. `D3set` controls this. From the coordinates in the (*x*,*y*)-space (`column`, `row`) and the computed iteration number (`height`) we can work out the pseudo-3D coordinates. First the horizontal value cell is calculated, if it fits on the screen, and then the value content, which gives the vertical component. If the value is higher than the previously determined maximal value for this column of the screen, then this is inserted in its place. If it is less, then that means that it corresponds to a hidden point in the picture, and hence it is omitted.

Program Fragment 7.1-2

```
   FUNCTION D3ComputeAndTest (x, y : real; xRange, yRange :
                            integer) : boolean;
      VAR
         iterationNo : integer;
         xSq, ySq, distanceSq : real;
         finished: boolean;
      PROCEDURE startVariableInitialisation;
       (* as usual *) BEGIN END;
      PROCEDURE compute;
      (* as usual *) BEGIN END;
      PROCEDURE test
      (* as usual *) BEGIN END;

      PROCEDURE D3set (VAR D3max : D3maxType;
                  column, row, height : integer);
         VAR
            cell, content : integer;
      BEGIN
         cell := column + row - (Yscreen -100) DIV 2;
         IF (cell >= 0) AND (cell <= Xscreen) THEN
         BEGIN
            content := height * D3factor + row;
            IF content > D3max[cell] THEN
               D3max[cell] := content;
         END;
      END;  (* D3set *)

   BEGIN (* D3ComputeAndTest *)
      StartVariableInitialisation;
      D3ComputeAndTest := true;
      REPEAT
         compute;
         test;
      UNTIL (iterationNo = MaximalIteration) OR finished;
      D3set (D3max, xRange, yRange, iterationNo);
   END (* D3ComputeAndTest *)
```

Program Fragment 7.1-3

```
   PROCEDURE D3draw (D3max: D3maxType);
      VAR
         cell, coordinate : integer;
   BEGIN
      setPoint (0, D3max[0]);
      FOR cell := 0 TO xScreen DO
      BEGIN
         coordinate := D3max[cell];
         IF coordinate >0 THEN DrawLine (cell, coordinate);
      END;
   END;  (* D3draw *)
```

The visible parts are not drawn until an entire row has been worked out. At the beginning of each row we start at the left side of the screen (`SetPoint`) and then draw the section as a series of straight lines (`DrawLine`).

With a few small supplements we can retain the principle of this computation, while improving the pictures somewhat.

The method can be changed in its first stage. In §2.2.3 we drew only every second line, and we can do the same here. As a result the gradations are more easily distinguished. The variable `D3yStep` controls the step size. For a quick survey, for example, every tenth line suffices. The variable `D3xStep` runs in the oblique direction, so that the slopes merge and the steps do not seem quite so abrupt if this value is relatively large.

Figure 7.1-2 shows the Julia set for Newton's method applied to

$z^3-1 = 0.$

There every second row is drawn, so `D3xStep` has the value 2. Essentially, this is a section from Figure 5.1-5, to the lower right of the middle.

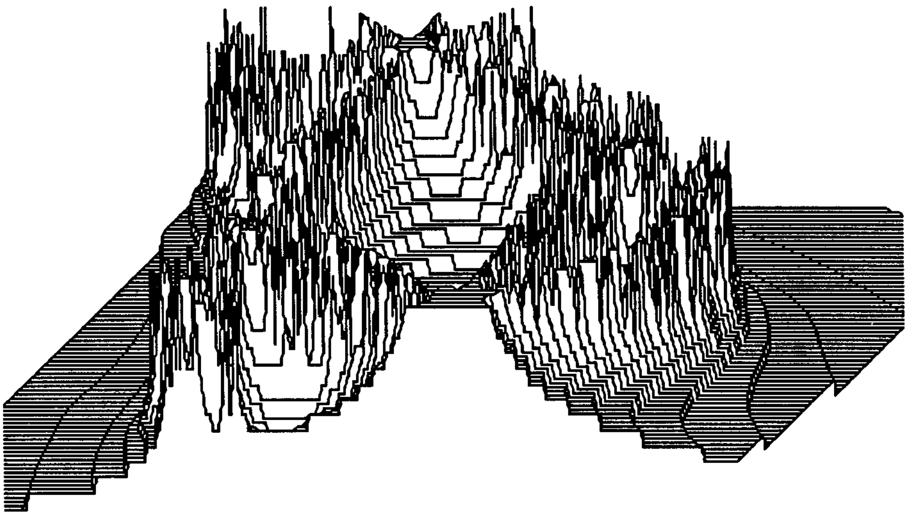


Figure 7.1-2 Julia set for Newton's method applied to $z^3-1 = 0$.

We have proceeded in fivefold steps in the next figure, Figure 7.1-3. It shows a Julia set that you already saw in its usual fashion in Figure 5.2-5.

These large steps are useful, for example, if you just want to get a quick view of the anticipated picture.

In order to make the central shape of the Mandelbrot set, or a Julia set, stand out more clearly than before, it helps to draw the lines thicker - that is, to use several adjacent lines - at these places. Our Pascal version provides a facility for doing this very easily, using the procedure `penSize`. In other dialects you have to remember the beginning and

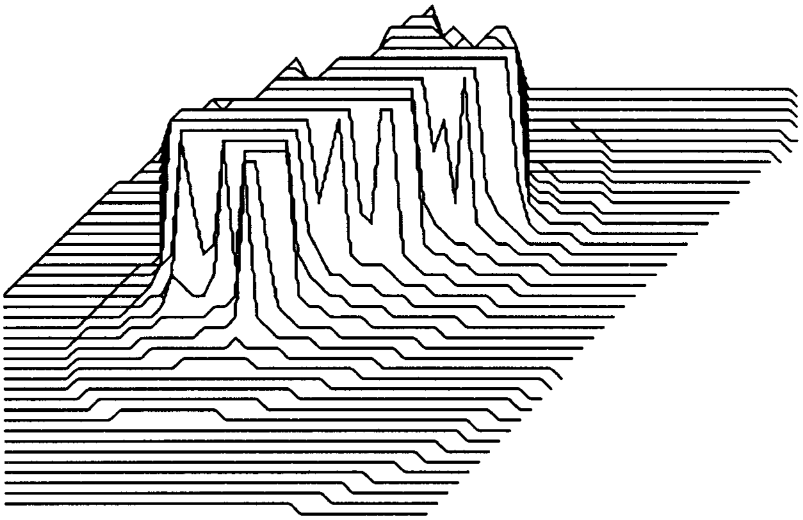


Figure 7.1-3 Julia set for $c = 0.5 + i * 0.5$.

end of this horizontal stretch and then draw a line one pixel displaced. Figure 7.10-4 shows a Mandelbrot set drawn in this manner.

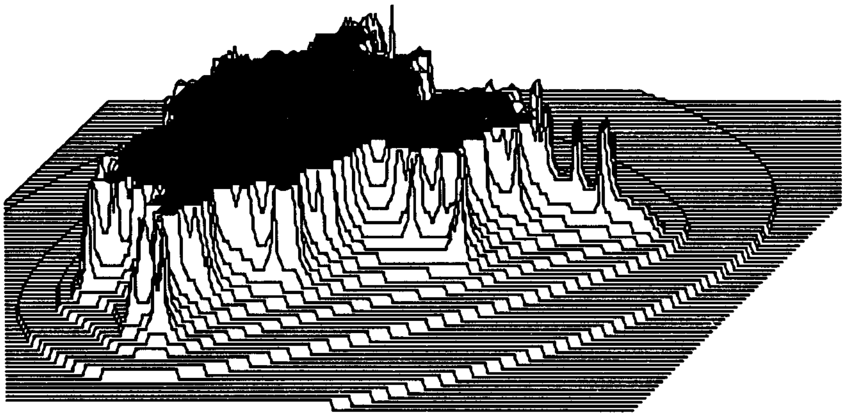


Figure 7.1-4 Gingerbread Man.

Sometimes in these pictures the slopes are interrupted by protruding peaks, and instead of mountains we find gentle valleys. That too can be handled. Instead of the iteration height we use the difference `MaximalIteration - iterationNo`. In Figure 7.1-5 we see a Julia set drawn in this manner:

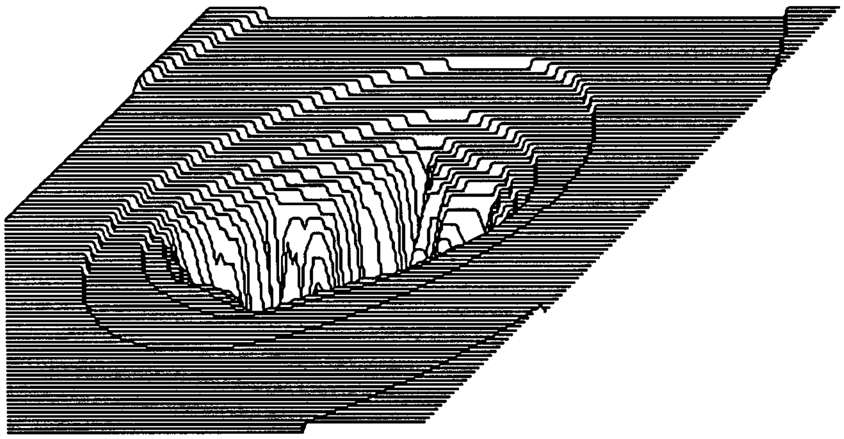


Figure 7.1-5 Julia set, top and bottom interchanged, $c = 0.745 + i * 0.113$.

Finally, as in Figure 7.1-6, we can use the reciprocal of the iteration depth in the 'third dimension', obtaining a convex imprint, but with steps of different heights.

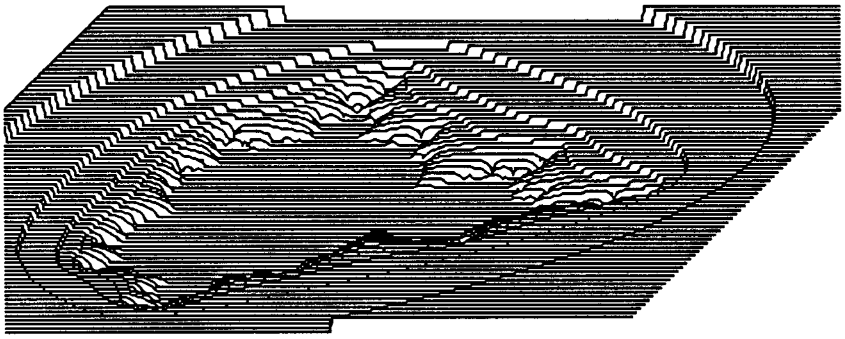


Figure 7.1-6 Gingerbread Man, inverse iteration height.

7.2 Invert It - It's Worth It!

In this section we bring to the fore something that hitherto has been at an infinite distance - namely, the attractor 'infinity'. It is clear that to do this we must forego what until now has been at the centre of things, the origin of the complex plane. The method whereby the two are interchanged is mathematically straightforward, if perhaps a little unfamiliar for complex numbers. To turn a number into another one, we invert it.

To each complex number z there corresponds another z', called its *inverse*, for which

$$z * z' = 1.$$

In other words, $z' = 1/z$, the reciprocal of z. The computational rules for complex numbers have already been set up in Chapter 4, so we can proceed at once to incorporate the idea into a Pascal program. The previous program is only changed a little. At the beginning of `ComputeAndTest` the appropriate complex parameter, namely z_0 for Julia sets and c for the Mandelbrot set, is inverted.

Program Fragment 7.2-1

```
  FUNCTION ComputeAndTest (Creal, Cimaginary : real)
                         : boolean;

   (* variables and local procedures as usual *)

      PROCEDURE invert (VAR x, y : real);
         VAR denominator : real;
      BEGIN
         denominator := sqr(x) + sqr(y);
         IF denominator = 0.0 THEN
            BEGIN
               x := 1.0E6; y := x;  {emergency solution}
            END ELSE BEGIN
               x := x / denominator;
               y := y / denominator;
            END;
         END; (* invert *)

   BEGIN
      invert (Creal, Cimaginary);
      startVariableInitialisation;
      REPEAT
         compute;
         test;
      UNTIL (iterationNo = MaximalIteration) OR fisnished;
      distinguish;
   END;  (* ComputeAndTest *)
```

With these changes we can recompute everything we have done so far. And look: the results are overwhelming. In Figure 7.2-1 you see what happens to the Gingerbread Man, when the underlying c-plane is inverted.

Now the Mandelbrot set appears as a black region surrounding the rest of the complex plane, which has a drop-shaped form. The buds, which previously were on the

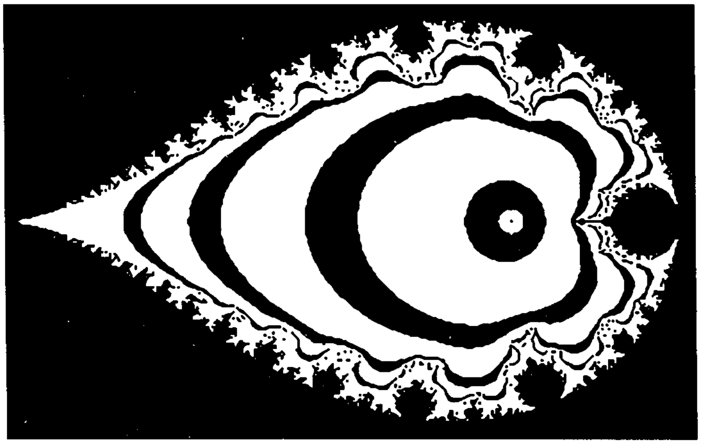


Figure 7.2-1 Inverted Mandelbrot set.

outside, are now on the inside. The first bud, as before, is still the largest, but it is quite comparable with the remainder. And the stripes, which got wider and more solid, the further away we got from the main body? If we do not pay attention, we may miss them completely. They are all collected together in a tiny region in the middle of the picture. The middle is where the point attractor ∞ appears. The mathematical inversion has thus turned everything inside out.

Let us now compare another Gingerbread Man with his inverse. It is the one for the third power.

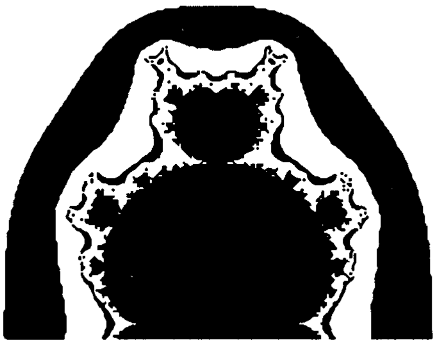


Figure 7.2-2 Gingerbread Man for the third power (top half; compare Figure 6.4-5).

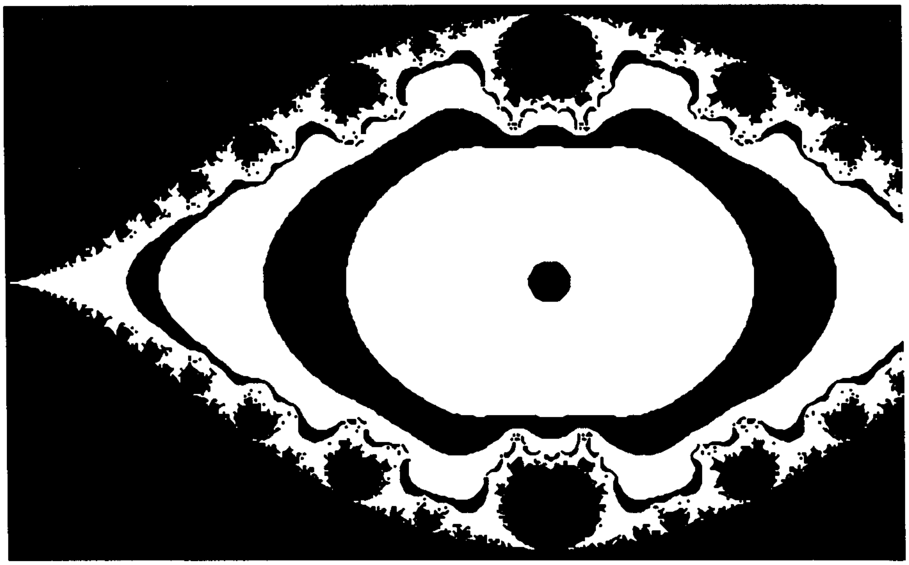


Figure 7.2-3 Gingerbread Man for the third power, inverted (cropped on right).

And what happens to the Julia sets? At first sight Figure 5.1-2 resembles its 'antipode' in Figure 7.2-4, but on further study the differences can be detected.

On the following pages are further pairs of normal and inverted Julia sets.

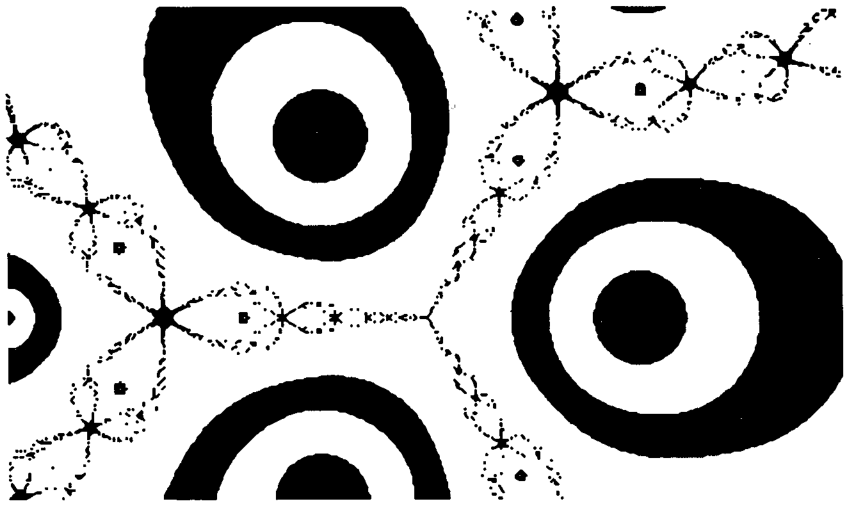


Figure 7.2-4 Inverted Julia set for Newton's method applied to $z^3-1 = 0$.

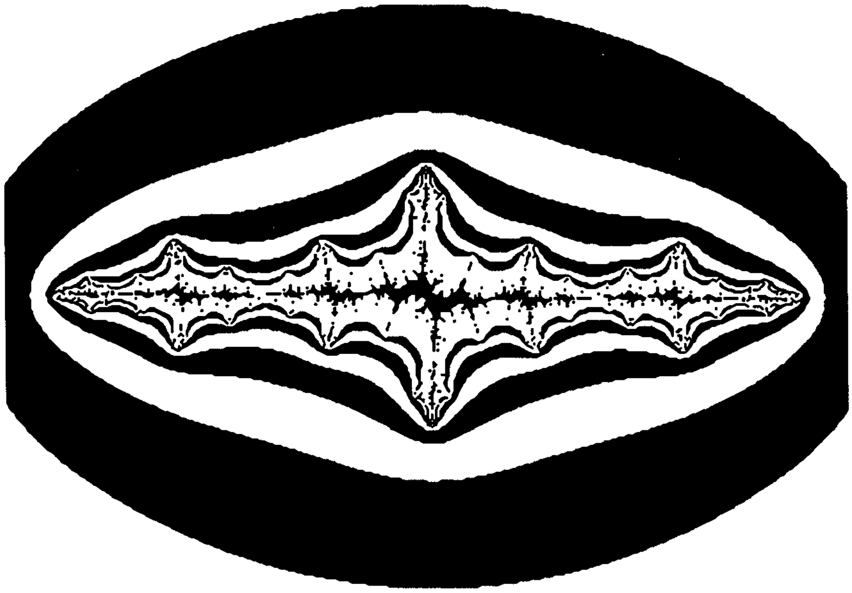


Figure 7.2-5 Julia set for $c = 1.39 - i * 0.02$.

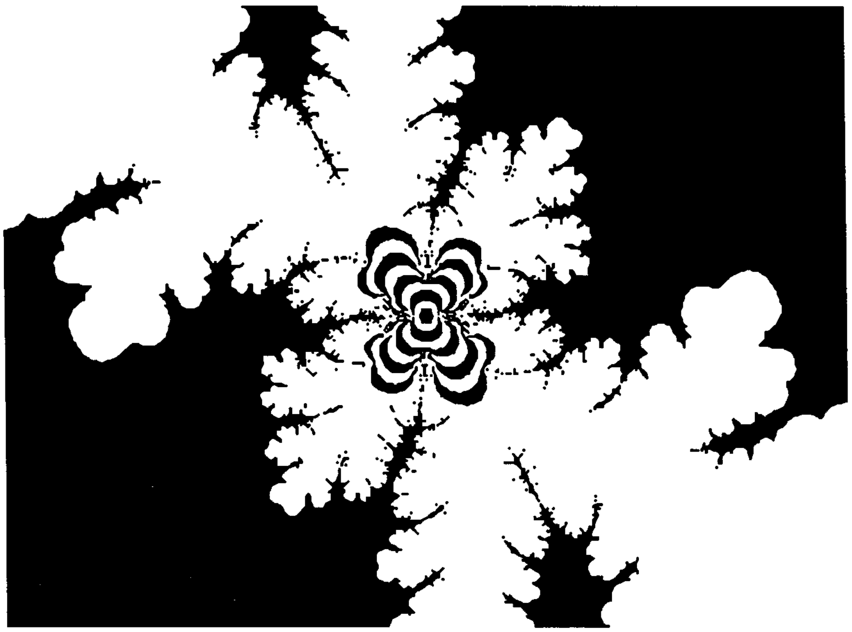


Figure 7.2-6 Inverted Julia set for $c = 1.39 - i * 0.02$.

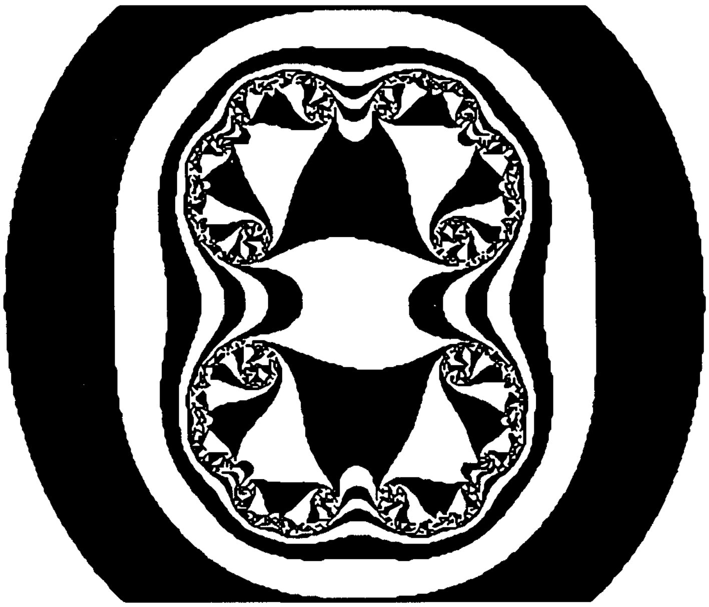


Figure 7.2-7 Julia set for $c = -0.35 - i * 0.004$.

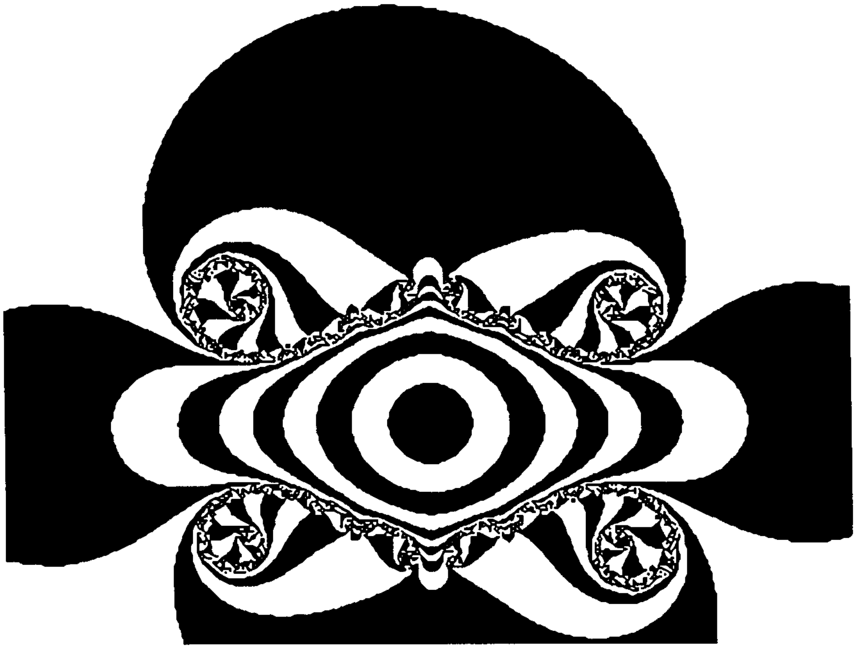


Figure 7.2-8 Inverted Julia set for $c = -0.35 - i * 0.004$.

7.3 The World Is Round

It is certainly attractive to compare corresponding pictures from the previous chapter with each other, and to observe the same structures represented in different ways and/or distorted. However, from a technical point of view a large part of the pictures is superfluous, in that it contains information which can already be obtained elsewhere. A more economical picture shows just the inside of the unit circle. Everything else can be seen in the inverted picture in the unit circle. It may amount to sacrilege on aesthetic grounds, but the pure information that lies in the complex plane can be contained in two circles of radius 1. The first contains all points (x,y) for which

$$x^2+y^2 \leq 1$$

and the second circle contains all the rest in inverted form:

$$x^2+y^2 \geq 1.$$

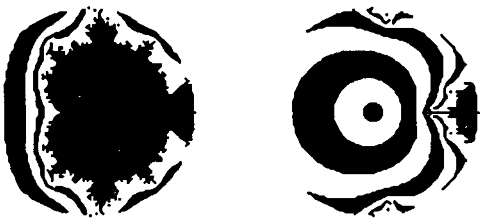


Figure 7.3-1 The entire complex plane in two unit circles.

In Figure 7.3-1 we see the Gingerbread Man in this form - of course, he has lost much of his charm. Imagine that these two circles are made of rubber: cut them out and glue the edges together back to back. Then all we need to do is blow up the picture like a balloon and the entire complex plane ends up on a sphere!

Mathematicians call this the *Riemann sphere*, in memory of the mathematician Bernhard Riemann (1826-66), who among other things made important discoveries in the area of complex numbers. His idea is explained in Figure 7.3-2.

The sphere and the plane meet at a point: the same point is the origin of the complex plane and the south pole S of the sphere. The north pole N acts as a centre of projection. A point P of the plane is to be mapped on the sphere. The connecting line NP cuts the sphere at R. The scales of the plane and the sphere are so adjusted that all points lying on the unit circle in the plane are mapped to the equator of the sphere. In the southern hemisphere we find the inner region, the neighbourhood of the origin. In the northern hemisphere we find everything that lies outside the unit circle, the 'neighbourhood of infinity'.

Why have we told you about that now? It leads naturally to new and dramatic graphical effects. Our programme for the next few pages will be to map the Gingerbread Man (playing the role of any picture in the complex plane) on to the Riemann sphere, to

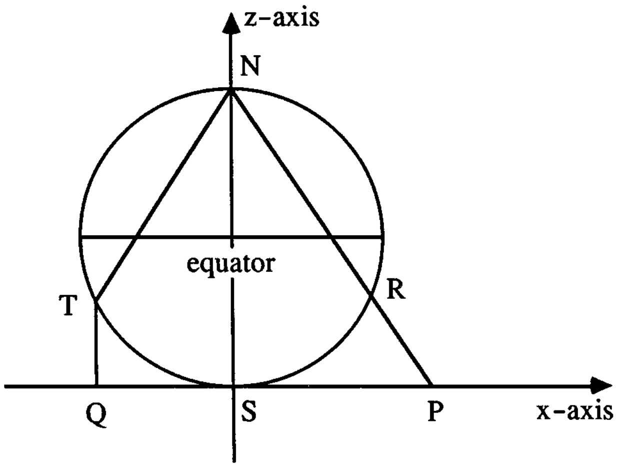


Figure 7.3-2 Projection from plane to sphere.

rotate this, and to draw the resulting picture. In order to make the resulting distortion clear, the drawing is performed by a simple orthogonal projection, which for example transforms point T to point Q in Figure 7.3-2.

For a clear and unified specification of the quantities occurring we adopt the following conventions. The final output is the complex plane, in which as before everything is computed and drawn. Each point is determined by the coordinate pair xRange, yRange (picture coordinates). The universal coordinates (for the Gingerbread Man these are c_{real} and $c_{\text{imaginary}}$) follow from these. They will be mapped on to a circle. From the two-dimensional coordinate pair a coordinate triple x_{sphere}, y_{sphere}, z_{sphere} is therefore formed.

The centre of the sphere lies above the origin of the complex plane. The x-axis and y-axis run along the corresponding axes of the plane, that is, $x_{\text{sphere}} \approx c_{\text{real}}$ and $y_{\text{sphere}} \approx c_{\text{imaginary}}$. The z_{sphere}-axis runs perpendicular to these two along the axis through the south and north poles.

We transfer all the points within the unit circle on to the southern hemisphere; all the others lie in the northern hemisphere. However, these are obscured, because we view the sphere from below. The points directly above the unit circle form the equator of the sphere. At the north pole, then, lies the point '∞'. It is in fact only one point! Two diametrically opposite points on the sphere (antipodes) are reciprocals of each other.

For mapping the complex plane we employ a second possibility as well as Riemann's; both are explained in Figure 7.3-2. There we see a section through the sphere and the plane. Suppose that a point P with coordinates (c_{real}, $c_{\text{imaginary}}$) is being mapped. The first possibility for achieving this is simply to transfer the x- and

y-coordinates, thus $x_{\text{sphere}} = c_{\text{real}}$ and $y_{\text{sphere}} = c_{\text{imaginary}}$. Then z is computed using the fact that

$$x^2+y^2+z^2 = 1,$$

so

$$z_{\text{sphere}} = \texttt{sqrt(1.0-sqr(}x_{\text{sphere}}\texttt{)-sqr(}y_{\text{sphere}}\texttt{))}.$$

This method we call *orthogonal projection.*

The second method is Riemann's. This time the north pole acts as a centre of projection.

To make different perspectives possible, we construct our figure on the Riemann sphere. Then we rotate it so that the point whose neighbourhood we wish to inspect falls at the south pole. Finally we project the resulting 3D-coordinates back into the complex plane.

On the next three pages we have again written out the procedure `Mapping`, including all local procedures. New global variables `Radius`, `Xcentre` and `Ycentre` are defined, which determine the position and size of the circle on the screen.

Because we want to use the mapping method (scan the rows and columns of the screen), we must of course work backwards through the procedure laid down above.

At the first step we test whether the point under investigation lies inside the circle. If so, we find its space coordinates using orthogonal projection. The x- and y-coordinates carry over unchanged. We compute the z-coordinate as described above. Because we are observing the sphere from below (south pole), this value is given a negative sign.

Next we rotate the entire sphere - on which each point lies - through appropriate angles (given by variables `width` and `length`). To prescribe this rotation in general we use matrix algebra, which cannot be explained further here. For further information see, for example, Newman and Sproull (1979).

From the rotated sphere we transform back into the complex plane using the Riemann method, and thus obtain the values x and y (for the Gingerbread Man these are interpreted as c_{real} and $c_{\text{imaginary}}$) for which the iteration is to be carried out.

The result of the iteration determines the colour of the original point on the screen.

Program Fragment 7.3-1

```
    PROCEDURE Mapping;
       TYPE
          matrix : ARRAY [1..3,1..3] OF real;
       VAR
          xRange, yRange : integer;
          x, y, deltaxPerPixel, deltayPerPixel : real;
          xAxisMatrix, yAxisMatrix: matrix;

       PROCEDURE ConstructxRotationMatrix (VAR m: matrix; alpha
                     : real);
```

```
BEGIN
   alpha := alpha * pi / 180.0; {radian measure}
   m[1,1] := 1.0; m[1,2] := 0.0; m[1,3] := 0.0;
   m[2,1] := 0.0; m[2,2] := cos(alpha); m[2,3] :=
                  sin(alpha);
   m[3,1] := 0.0; m[3,2] := -sin(alpha); m[3,3] :=
                  cos(alpha);
END; (* ConstructxRotationMatrix *)

PROCEDURE ConstructyRotationMatrix (VAR m: matrix; beta
                  : real);
BEGIN
   beta := beta * pi / 180.0; {radian measure}
   m[1,1] := cos(beta); m[1,2] := 0.0; m[1,3] :=
                  sin(beta);
   m[2,1] := 0.0; m[2,2] := 1.0; m[2,3] := 0.0;
   m[3,1] := -sin(beta); m[3,2] := 0.0; m[3,3] :=
                  cos(beta);
END; (* ConstructyRotationMatrix *)

PROCEDURE VectorMatrixMultiply (xIn, yIn, zIn : real; m
                  : matrix;
                  VAR xOut, yOut, zOut : real);
BEGIN
   xOut := m[1,1]*xIn + m[1,2]*yIn+m[1,3]*zIn;
   yOut := m[2,1]*xIn + m[2,2]*yIn+m[2,3]*zIn;
   zOut := m[3,1]*xIn + m[3,2]*yIn+m[3,3]*zIn;
END; (* VectorMatrixMultiply *)

FUNCTION ComputeAndTest (Creal, Cimaginary : real)
                  : boolean;
   VAR
      iterationNo : integer;
      finished : boolean;
      x, y, xSq, ySq : real;

PROCEDURE StartVariableInitialisation;
BEGIN
   finished := false;
   iterationNo := 0;
   x := 0.0;
```

```
      xSq := sqr(x);
      y := 0.0;
      ySq := sqr(y);
    END (* StartVariableInitialisation *)

    PROCEDURE Compute;
    BEGIN
      y := x*y;
      y := y+y-Cimaginary;
      xSq := sqr(x);
      ySq := sqr(y);
      iterationNo := iterationNo + 1;
    END;

    PROCEDURE test;
    BEGIN
      finish := ((xSq + ySq) > 100.0);
    END;

    PROCEDURE distinguish;
    BEGIN
      ComputeAndTest := (iterationNo = MaximalIteration) OR
         (iterationNo < Bound) AND (odd(iterationNo));
    END;

  BEGIN (* ComputeAndTest *)
    StartVariableInitialisation;
    REPEAT
      compute;
      test;
    UNTIL (iterationNo = MaximalIteration OR finished);
  END; (* ComputeAndTest *)

  FUNCTION  calculateXYok(VAR x, y : real;
                  xRange, yRange : integer) : boolean;
    VAR
      xSphere, ySphere, zSphere,
      xInter, yInter, zInter : real;
    BEGIN
      IF ((sqr(1.0 * (xRange - xCentre))+sqr(1.0*(yRange-
                  yCentre)))
           > sqr(1.0*Radius))
```

```
        THEN calculateXYok := false;
        ELSE BEGIN
           caculateXYok := true;
           xSphere := (xRange-Xcentre)/Radius;
           ySphere := (yRange-Ycentre)/Radius;
           zSphere := -sqrt(abs(1.0-sqr(xSphere)-
                         sqr(ySphere)));
           VectorMatrixMultiply
              (xSphere, ySphere, zSphere, yAxisMatrix,
                 xInter, yInter, zInter);
           VectorMatrixMultiply
              (xInter, yInter, zInter, xAxisMAtrix,
                 xSphere, ySphere, zSphere);
           IF zSphere = 1.0 THEN BEGIN
              x := 0.0;
              y := 0.0;
           END ELSE BEGIN
              x := xSphere/(1.0 - zSphere);
              y := ySphere/(1.0 - zSphere);
           END;
        END;
     END;  (* calculateXYok *)

  BEGIN
     ConstructxRotationMatrix (xAxisMatrix, width);
     ConstructyRotationMatrix (yAxisMatrix, length);
     FOR yRange := Ycentre-Radius TO Ycentre+Radius DO
        FOR xRange := Xcentre-Radius TO Xcentre+Radius DO
        BEGIN
           IF calculateXYok (x, y, xRange, yRange) THEN
              IF ComputeAndTest (x,y) THEN
                 SetPoint (xRange, yRange);
        END;
  END; (* Mapping *)
```

In the first step of `Mapping` we set up the two rotation matrices. These are arrays of numbers which will be useful in each computation. In this way we avoid using the rather lengthy computations of sine and cosine unneccessarily often.

The main work again takes place in the two `FOR` loops. You may have realised

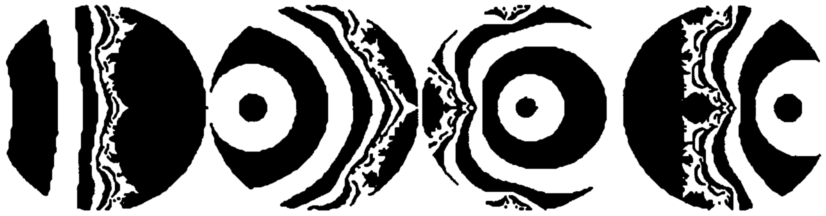


Figure 7.3-3 Examples of Mandelbrot sets on the Riemann sphere.

that we no longer need to scan the entire screen: it is sufficient to investigate the region in which the circle appears. We therefore terminate the computation if we discover from `calculateXYok` that the point is outside this region. However, if we find a point inside the circle on the screen, `calculateXYok` computes the coordinates in the steps descibed above. The final `ComputeAndTest` differs only in small ways from the version already laid down.

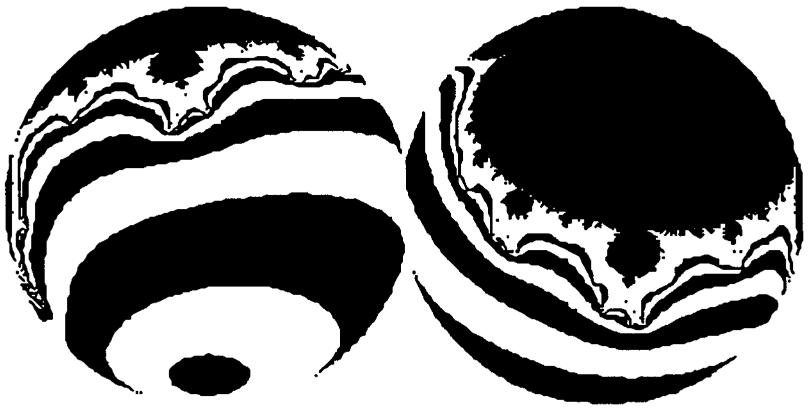


Figure 7.3-4 Gingerbread Man rotated 60°, front and back.

Thus the important new ingredient in this chapter for representation on the Riemann sphere is the functional procedure `calculateXYok`. First the screen coordinates `xRange` and `yRange`, together with the variables that determine the circle, are tested, to see whether we are inside it. Do not worry about the rather curious constructions such as `sqr(1.0*Radius)`. The variable `Radius` is an integer, and if for example we square 200, we may exceed the appropriate range of values for this type, which in many implementations of Pascal is limited to 32767. By multiplication by 1.0, `Radius` is implicitly converted to a real number, for which this limit does not hold.

The variables `xSphere` and `ySphere`, with values between -1 and +1, are computed from the screen coordinates, and from them the negative `zSphere`.

For the rotation we treat the three variables that define a point as a vector, which is multiplied by the previously computed matrix. The result is again a vector and contains the coordinates of the point after rotation. Intermediate values are stored in `xInter`, `yInter`, and `zInter`. In the next picture you once more see our favourite figure with the parameters width = 60, length = 0 (respectively 180).

7.4 Inside Story

As we have learned from our calculations, the most interesting thing about a fractal figure is undoubtedly the boundary. Its endless variety has occupied us for a great many pages. The approximation of this boundary by contour lines has led to a graphical construction of the neighbourhood of the boundary, which is often very attractive. But the inside of the basin of attraction has until now remained entirely black. That can be changed too!

In the interior the information that we can lay hands on is just the two values x and y; that is, the components of the complex number z. These are real numbers between -10 and +10. What can we do with those? First one might imagine taking one of the numbers, and using the `TRUNC` function to cut off everything after the decimal point, and then drawing the points for which the result is an odd number. To do this we can use the Pascal function `ODD`. Unfortunately the result is very disappointing, and it does not help to take $x+y$ or multiples of this. However, we have not tried everything that is possible here, and we recommend that you experiment for yourself. We have obtained the best results using a method which, to end with, we now describe. (It should be said at once that it is intended purely for graphical effect. No deeper meaning lies behind it.)

You must...

- Take the number `distanceSq`, which has already been found during the iteration.
- Construct its logarithm.
- Multiply by a constant `insideFactor`. This number should lie between 1 and 20.
- Cut off the part after the decimal point.
- If the resulting number is odd, colour the corresponding point on the screen.

All this takes place in the procedure distinguish below. The Pascal print-out has become so long that it spreads across three lines.

Program Fragment 7.4-1

```
PROCEDURE distinguish
BEGIN
   ComputeAndTest :=
      (iterationNo = MaximalIteration) AND
      ODD(TRUNC(insideFactor * ABS( ln (distanceSq))));
END;
```

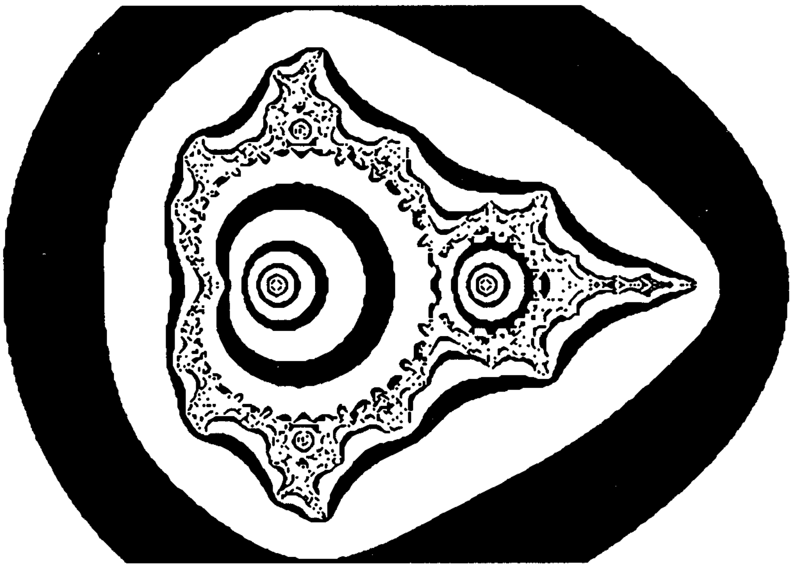


Figure 7.4-1 Gingerbread Man with interior structure (`insideFactor` = 2).

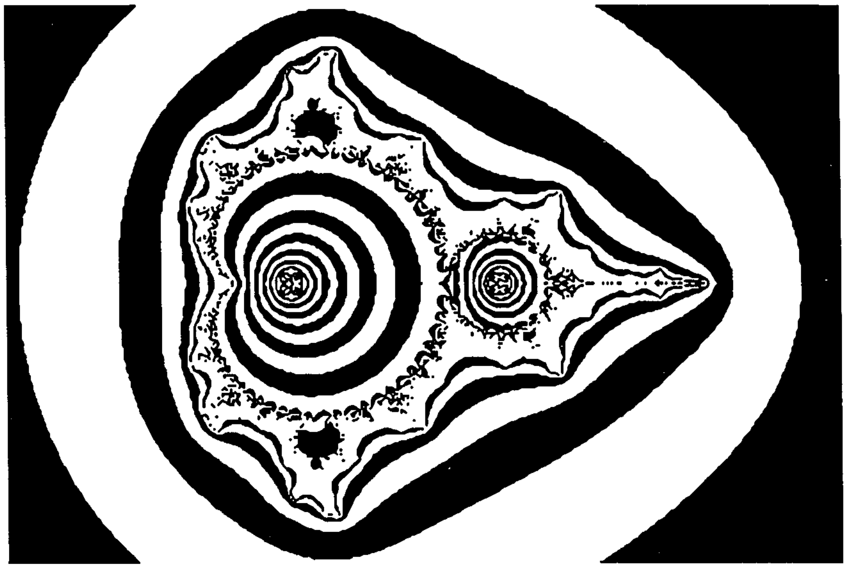


Figure 7.4-2 Gingerbread Man with interior structure (`insideFactor` = 10).

The effect of this computation is to divide the interior of the basin of attraction into several parts. Their thickness and size can be changed by altering the value of `insideFactor`.

Can you still see the Gingerbread Man in Figures 7.4-1 and 7.4-2?

Julia sets, too, provided they contain large black regions, can be improved by this method. First we show an example without contour lines, then one with.

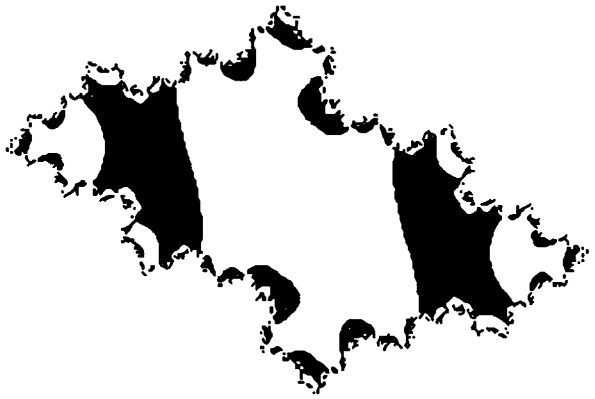


Figure 7.4-3 Julia set for c = 0.5 + i*0.5 with interior structure.

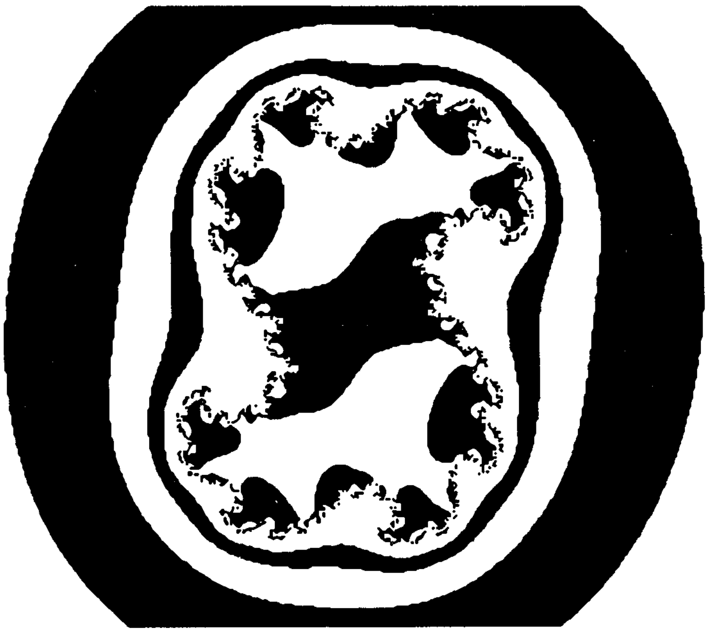


Figure 7.4-4 Julia set for c = -0.35 + i*0.15 with interior structure.

Computer Graphics Experiments and Exercises for Chapter 7

Exercise 7-1

Modify your programs to implement three-dimensional representation. Try out all the different methods. Parameters such as total height, or step-size in the two directions, let you generate many different pictures.

You get especially rugged contours if you allow drawing only in the horizontal and vertical directions. Then you must replace the `move-to` command, which generates slanting lines as well, by an appropriate instruction.

Exercise 7-2

If you wish to view the object from a different direction, you can interchange parameters, so that `Left` > `Right` and/or `Bottom` > `Top`. Alternatively or in addition you can displace the individual layers to the left instead of to the right.

Exercise 7-3

Add the inversion procedure to all programs that draw things in the complex plane. In particular you will obtain fantastic results for filigreed Julia sets.

Exercise 7-4

Further distortions occur if before or after the inversion you add a complex constant to the c- or z_0-value. In this way other regions of the complex plane can be moved to the middle of the picture.

Exercise 7-5

Transfer everything so far computed and drawn on to the Riemann sphere. Observe the objects from different directions.

Exercise 7-6

Magnified sections of the sphere are also interesting. Not in the middle - everything stays much the same there. But a view of the edge of the planet (perhaps with another behind it) can be very dramatic.

Exercise 7-7

Perhaps it has occurred to you to draw lines of longitude and latitude? Or to try other cartographic projections?

Exercise 7.8

The entire complex plane is given a very direct interpretation in the Riemann sphere. New questions arise which, as far as we know, are still open. For example: how big is the area of the Gingerbread Man on the Riemann sphere?

Exercise 7-9

Combine the method for colouring the interior with all previously encountered pictures!