Kapitel 11

Fraktale

11.1 Selbstähnlichkeit

Viele in der Natur vorkommende Strukturen weisen eine starke Selbstähnlichkeit auf. Beispiele sind Gebirgsformationen, Meeresküsten oder Pflanzenblätter. Solche in sich wiederholende Muster, die beim Hereinzoomen immer wieder zutage treten, lassen sich ausnutzen, wenn man ein umfangreiches Gebilde durch ein kleines Erzeugendensystem darstellen möchte.

Die Produktion von Fraktal-Bildern erfolgt daher durch wiederholte Anwendung einer Transformation f auf einen bestimmten Teil des Bildes.

Abbildung 11.1: Farnblatt

11.2 Koch'sche Schneeflocke

Gegeben ein Polygon. Transformiere jede Polygonkante $\overline{PQ}$ zu einer Folge von Kanten $\overline{PR}$, $\overline{RS}$, $\overline{ST}$, $\overline{TQ}$, wie in Abbildung 11.2 gezeigt.

D.h., R und T dritteln die Kante $\overline{PQ}$, S ist eine 60°-Drehung des Knotens T um das Zentrum R gegen den Uhrzeigersinn. Das so erhaltene Polygon kann erneut transformiert werden.

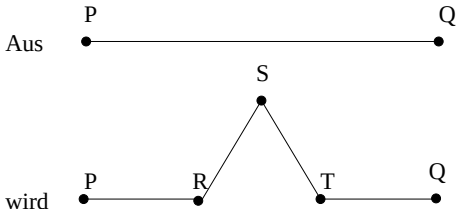
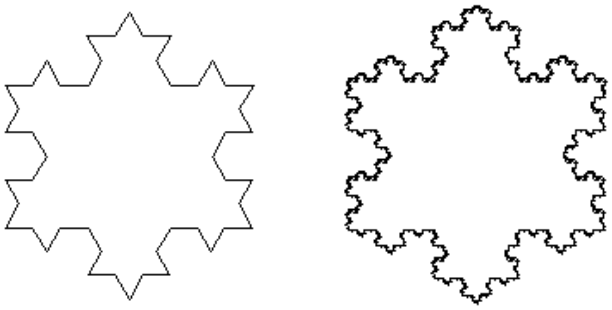


Abbildung 11.2: Iterationsvorschrift zur Koch'schen Schneeflocke



48 Polygonpunkte 3072 Polygonpunkte

Abbildung 11.3: Koch'sche Schneeflocke

11.3 Fraktale Dimension

Ein selbstähnliches Objekt hat *Dimension D*, falls es in N identische Kopien unterteilt werden kann, die jeweils skaliert sind mit dem Faktor $r = \frac{1}{N^{\frac{1}{D}}}$.

Beispiel: Linie hat Dimension 1, denn sie besteht aus N Stücken der Größe $\frac{1}{N}$.

Quadratische Fläche hat Dimension 2, denn sie besteht aus N Flächen der Größe $\frac{1}{N^{\frac{1}{2}}}$,

z.B. 9 Teilflächen mit Kantenlängen skaliert um $r = \frac{1}{9^{\frac{1}{2}}} = \frac{1}{3}$

Sind N und r bekannt, läßt sich D bestimmen:
$r = \frac{1}{N^{\frac{1}{D}}} \Rightarrow r \cdot N^{\frac{1}{D}} = 1 \Rightarrow N^{\frac{1}{D}} = \frac{1}{r} \Rightarrow \frac{1}{D} \cdot \log(N) = \log(\frac{1}{r})$
$\Rightarrow D = \frac{\log(N)}{\log(\frac{1}{r})}$

Beispiel: Die Koch'sche Schneeflocke hat Dimension $D = \frac{\log(4)}{\log(3)} = 1.2618$, denn jeder Kantenzug besteht aus $N = 4$ Kopien, jeweils skaliert um den Faktor $r = \frac{1}{3}$.

11.4 Lindenmayer-Systeme

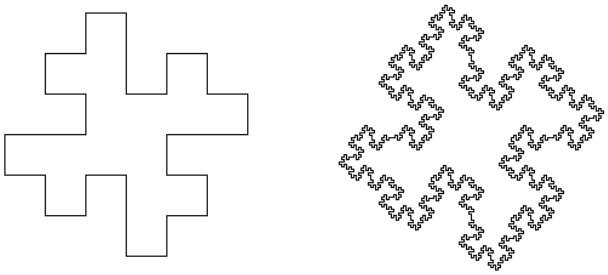
Eine nicht-grafische Beschreibung mancher Fraktale erfolgt mit Lindenmayer-Systemen, kurz *L*-Systems.
Ein Beispiel für ein L-System ist die quadratische Koch-Kurve:

$$\begin{aligned} &\text{Alphabet } \sum = \{r,u,l,d\} \text{ für right, up, left, down} \\ &\text{Regelmenge } f = \{ \quad r \Rightarrow r\,u\,r\,d\,d\,r\,u\,r, \\ &\qquad\qquad\quad u \Rightarrow u\,l\,u\,r\,r\,u\,l\,u, \\ &\qquad\qquad\quad l \Rightarrow l\,d\,l\,u\,u\,l\,d\,l, \\ &\qquad\qquad\quad d \Rightarrow d\,r\,d\,l\,l\,d\,r\,d\,\} \end{aligned}$$

Ausgehend von einem Startwort w wird in jedem Iterationsschritt auf jedes Zeichen von w eine Regel aus f angewandt.
Sei $w = r\,u$. Dann ist $f(w) = r\,u\,r\,d\,d\,r\,u\,r\,u\,l\,u\,r\,r\,u\,l\,u$.

Abbildung 11.4: Anwendung von 2 Regeln eines Lindenmayer-Systems

Ist nach n Iterationen das Wort $f^n(w)$ entstanden, so kann es, zusammen mit dem Parameter n (erforderlich für die Schrittweite), einer Ausgabeprozedur übergeben werden.
Die quadratische Koch-Kurve hat die Dimension $D = \frac{\log(8)}{\log(4)} = 1.5$, denn jeder Kantenzug besteht aus $N = 8$ Kopien, jeweils skaliert um den Faktor $r = \frac{1}{4}$.



32 Polygonpunkte 16384 Polygonpunkte

Abbildung 11.5: L-System

11.5 Baumstrukturen

Gegeben ein “Ast”, repräsentiert durch ein 5-seitiges Polygon:

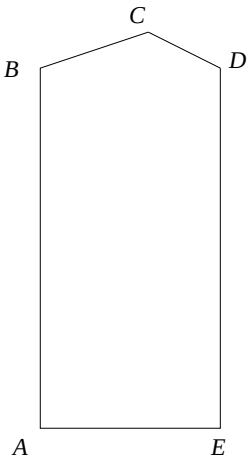
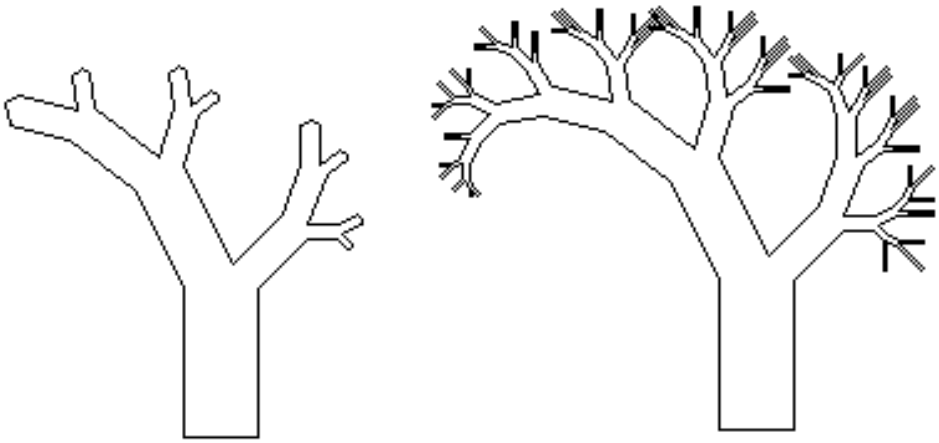


Abbildung 11.6: Ausgangspolygon für Baum

An den Kanten $\overline{BC}$ bzw. $\overline{CD}$ können nun weitere “Äste” als Polygone angesetzt werden, die bzgl. des gegebenen Polygons mit 0.75 bzw. 0.47 skaliert sind. Durch wiederholtes Anstückeln entsteht ein Baum mit immer feinerer Verästelung. Statt dieses deterministischen Verfahrens kann man z.B. die Lage des Punktes C von einem Zufallsparameter beeinflussen lassen.



47 Polygonpunkte 3071 Polygonpunkte

Abbildung 11.7: Vom Baum-Algorithmus erzeugter Baum

11.6 Mandelbrot-Menge

Sei z eine komplexe Zahl mit Realteil $z.re$ und Imaginärteil $z.im$:

```
public class Complex
{
        double re;
        double im;

...
```

Das Quadrat einer komplexen Zahl z hat den Realteil $z.re^2 - z.im^2$ und den Imaginärteil $2 \cdot z.re \cdot z.im$.
Der Betrag einer komplexen Zahl c sei $|c| = \sqrt{c.re^2 + c.im^2}$.
Betrachte die Funktion $f : \mathbb{C} \to \mathbb{C}, f(z) = z^2 + c$ für festes c.

```
Complex f(Complex z)
{
        double tmp;

        tmp   = z.re;
        z.re  = z.re * z.re - z.im * z.im + c.re;
        z.im  = 2 * tmp * z.im + c.im;

        return z
}
```

Betrachte die Folge $z, f(z), f^2(z), f^3(z), \ldots$ beginnend bei $z = 0$ für festes c.

Für manche c konvergiert sie zu einem Fixpunkt, für manche c gerät sie in einen (beschränkten) Zyklus, für manche c verhält sie sich (beschränkt) chaotisch, für manche c strebt die Folge gegen Unendlich.

Die *Mandelbrotmenge* besteht aus solchen komplexen Zahlen c, die beim Startwert $z = 0$ zu einer beschränkten Folge führen. Um die Mandelbrotmenge grafisch darzustellen, definiert man

$$\text{farbe}(c) := \begin{cases} \text{schwarz}, & \text{falls die zu } c \text{ gehörende Folge bleibt beschränkt} \\ \text{weiß}, & \text{falls die zu } c \text{ gehörende Folge bleibt nicht beschränkt} \end{cases}$$

Ordne jedem Pixel $(p.x, p.y)$ des Bildschirms eine komplexe Zahl zu wie folgt. Die linke obere Ecke bezeichne die komplexe Zahl `start` (z.B. $-2.15, 2.15$). Die rechte obere Ecke bezeichne die komplexe Zahl `ende` (z.B. $0.85, 2.15$)
Dann ergibt sich bei 300 Pixeln pro Zeile eine Schrittweite von $(0.85 + 2.15)/300 = 0.01$.
Somit läßt sich ein Koordinatenpaar $(p.x, p.y)$ umrechnen durch den folgenden Konstruktor.

```
public Complex(Point p, Complex start, double schritt)
// bildet die komplexe Zahl zum Pixel p mit linker/oberer Ecke start
// und Schrittweite schritt
{
        this.re = start.re + schritt * (double)p.x;
        this.im = start.im - schritt * (double)p.y;
}
```

Sei $(p.x, p.y)$ das zur komplexen Zahl c gehörende Pixel. Dann färbe es mit farbe(c).

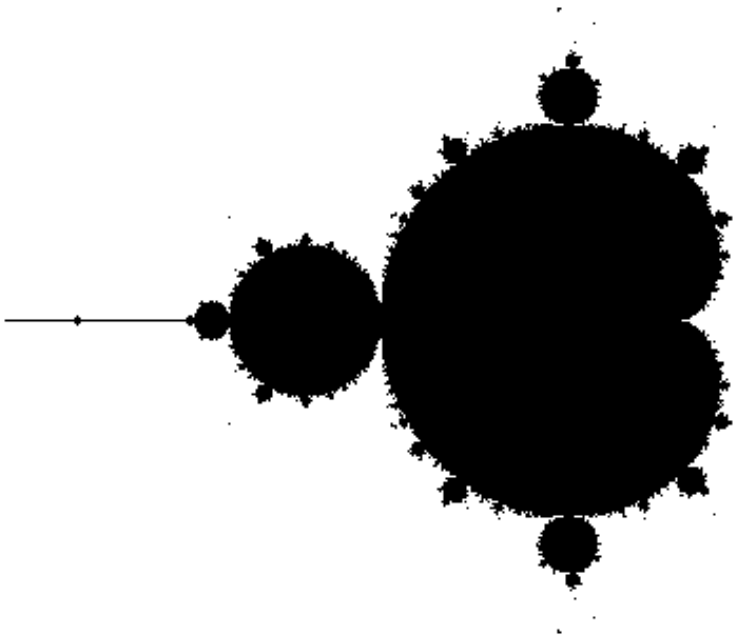


Abbildung 11.8: Mandelbrotmenge: $-2.2 \leq z.re \leq 0.6$, $-1.2 \leq z.im \leq 1.2$, 100 Iterationen

Implementation der Mandelbrot-Menge

Sobald während des Iterierens der Betrag von $z > 2 \Rightarrow$ Folge wächst mit Sicherheit über alle Grenzen (Satz von Fatou). Falls Betrag von z nach z.B. 100 Iterationen noch $< 2 \Rightarrow$ Folge bleibt vermutlich beschränkt.

D.h., bei Erreichen einer vorgegebenen Iterationenzahl wird das Pixel, welches der komplexen Zahl c entspricht, schwarz gefärbt, da c vermutlich zu einer beschränkten Folge führt.

```
void mandel ()
{
     Point p;
     int zaehler;
     Complex c, z;

     for (p.x = 0; p.x < WIDTH;  p.x++)
     for (p.y = 0; p.y < HEIGHT; p.y++)
     {
          zaehler = 0;
          c       = new Complex(p, start, schritt);
          z       = new Complex(c);

          while ((betrag (z) < 2.0) && (zaehler++ < max_iter))

             z = f(z);

          if (betrag(z) < 2.0) set_pixel(p);
      }
}
```

Bemerkung: Bei Erhöhung der Iterationszahl können einige Pixel weiß werden, da sich herausgestellt hat, daß durch weitere Iterationen die Betragsgrenze 2 überschritten wird.
Um die Zahlen c, die zu einer unbeschränkten Folge geführt haben, weiter zu klassifizieren, setzt man

$$\text{farbe}(c) := \begin{cases} \text{schwarz}, & \text{falls Folge beschränkt bleibt} \\ \text{weiß}, & \text{falls Iterationszahl nach Überschreiten von Betrag 2 gerade} \\ \text{schwarz}, & \text{falls Iterationszahl nach Überschreiten von Betrag 2 ungerade} \end{cases}$$

Also ergibt sich:

```
if (betrag < 2.0)        set_pixel (p); else
if ((zaehler %  2) != 0) set_pixel (p);
```

Bei Farbbildschirmen gibt es folgende Möglichkeit, die Divergenzgeschwindigkeit der Folge für ein c zu veranschaulichen:

Es seien die Farben `farbe[0], ..., farbe[NUM_COL-1]` vorhanden

```
set_col_pixel (p, farbe [zaehler % NUM_COL])
```

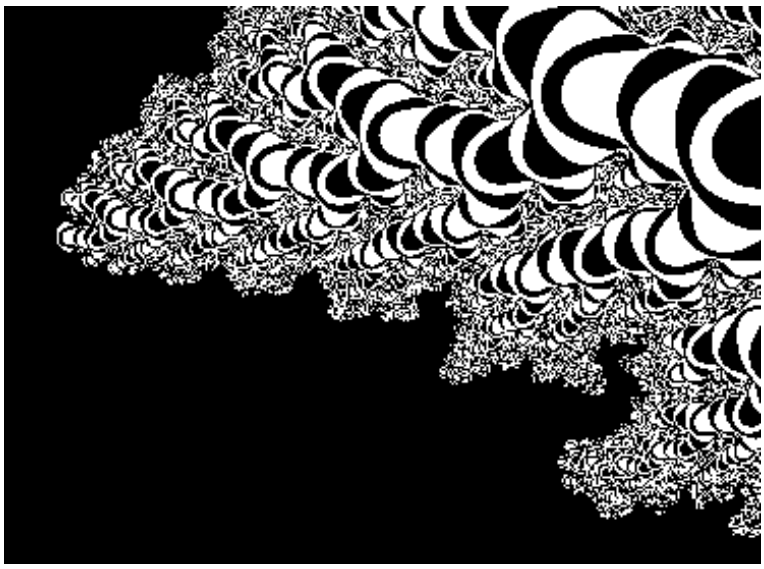


Abbildung 11.9: Mandelbrotmenge: $-1.2548 \leq Re \leq -1.2544$, $0.3816 \leq Im \leq 0.3822$, 100 Iterationen

11.7 Julia-Menge

Gegeben eine komplexe Funktion $f : \mathbb{C} \to \mathbb{C}$, z.B. $f(z) = z^2$. Wähle Startwert z_0. Betrachte die Folge $z_0, {z_0}^2, z_0^4, z_0^8, \ldots$. Es gilt:

a) für $|z_0| < 1$ werden die erzeugten Zahlen immer kleiner und konvergieren zum Nullpunkt,

b) für $|z_0| > 1$ werden die erzeugten Zahlen immer größer und laufen gegen unendlich,

c) für $|z_0| = 1$ bleiben die Zahlen auf dem Einheitskreis um den Ursprung, der die Gebiete a) und b) trennt.

Die Menge c) wird *Julia-Menge* genannt. Sie ist invariant bzgl. der komplexen Funktion $f(z)$ und punktsymmetrisch zum Ursprung.

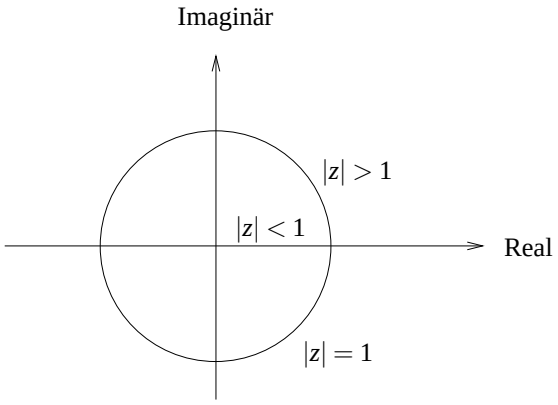


Abbildung 11.10: Julia-Kurve für $f(z) = z^2$

Es gilt: Julia-Menge für $f(z) = z^2 + c$ ist genau dann zusammenhängend, wenn c in der Mandelbrotmenge für $f(z)$ liegt.

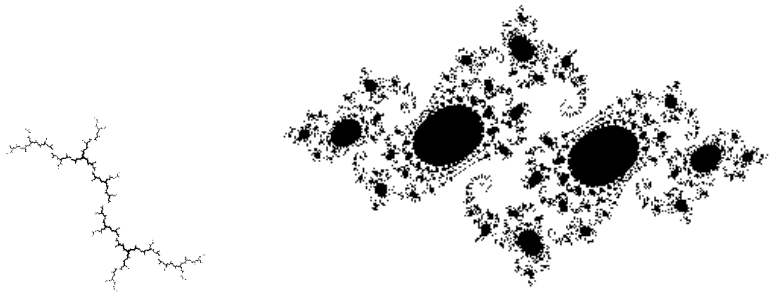


Abbildung 11.11: Vom Julia-Algorithmus erzeugte Bilder mit $c = i$ und $c = 0.746 + 0.112i$

Implementation der Julia-Menge:

1. Möglichkeit: Sei *c* gegeben. Jeder Wert *z* der komplexen Fläche wird überprüft, ob er als Startwert zu einer beschränkten Folge führt:

```
void  julia_voll(         // berechnet Julia-Menge als gefuelltes Gebiet
        Complex c,        // fuer komplexes c,
        Complex start,    // die linke obere Ecke start der komplexen Flaeche,
        double schritt,   // Schrittweite schritt
        int max_iter)     // mit vorgegebener Iterationszahl max_iter
{
        Point p = new Point();
        Point q = new Point();          // Symmetrischer Punkt
        int zaehler ;
        Complex z;

        for (p.x = 0; p.x <= WIDTH;     p.x++)
        for (p.y = 0; p.y <= HEIGHT/2;  p.y++)
        // Wegen Symmetrie nur bis zur Haelfte laufen
        {
                z = new Complex(p, start, schritt);

                q.x = WIDTH  - p.x;     // Symmetrie nutzen
                q.y = HEIGHT - p.y;

                zaehler = 0;
                while ((q_betrag(z) < 4.0) && (zaehler++ < max_iter))
                {
                        z = f(z,c);
                }

                if (q_betrag(z) < 4.0)
                {
                        set_pixel(p);
                        set_pixel(q);
                }
        }
}
```

Es gilt: Der Rand der berechneten Fläche stellt die Julia-Menge für *c* dar.

2. Möglichkeit: *Inverse Iteration Method*:
Aus $f(z) = z^2 + c \Rightarrow f^{-1}(z) = \pm\sqrt{z-c}$

Sei $z = a + b \cdot i$ eine komplexe Zahl. Es gilt:

$$\sqrt{z} = \begin{cases} \pm\left(\sqrt{\frac{|z|+a}{2}} + i \cdot \sqrt{\frac{|z|-a}{2}}\right) & \text{falls } b \geq 0, \\ \pm\left(\sqrt{\frac{|z|+a}{2}} - i \cdot \sqrt{\frac{|z|-a}{2}}\right) & \text{falls } b < 0 \end{cases}$$

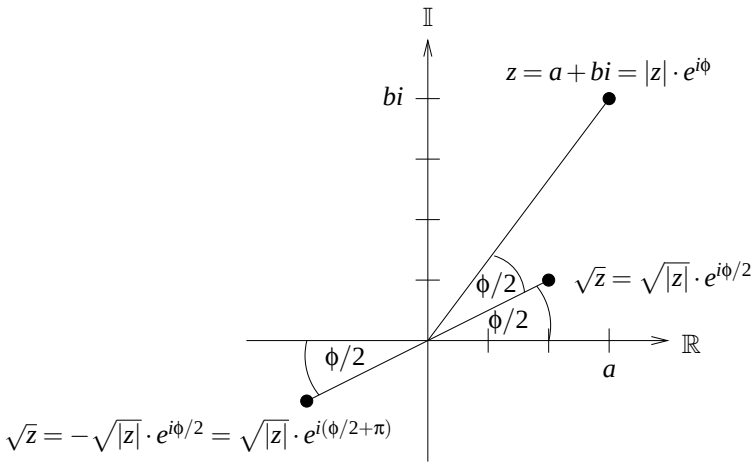


Abbildung 11.12: Zur Berechnung der Wurzel bei der Inverse Iteration Method

Realisiere f^{-1} durch

```
Complex backward_random (Complex z, Complex c)
{
     /* z = z - c */
     /* bestimme r mit r * r = z */
     /* wuerfel das Vorzeichen von r */
     return (r);
}
```

Die Wahl des Vorzeichens bei dieser inversen Iteration erfolgt zufällig.

Ausgehend von $z = 1 + 0i$ werden zunächst 50 Iterationen $z := f^{-1}(z)$ durchgeführt, um die berechneten Punkte nahe genug an die Julia-Kurve heranzuführen. Iteriere danach weiter $z := f^{-1}(z)$ und zeige jeweils die berechneten z:

```
void julia_rueck(          // berechnet Julia-Menge
        Complex c,         // fuer komplexes c
        Complex start,     // linke obere Ecke der komplexen Flaeche
        double schritt,    // Schrittweite
        int anzahl)        // durch anzahl Rueckwaertsspruenge
{
        int k;
        Complex z = new Complex(1.0, 0.0);   // Startposition

        for (k=0; k<50; k++)        // tastet sich an die Julia-Kurve heran
            z = backward_random(z,c);


        for (k=0; k < anzahl; k++) {        // bestimme anzahl Vorgaenger
            z = backward_random(z,c);
            set_pixel(z.getPoint(start, schritt)); // und zeichne sie
        }

}
```

11.8 Java-Applet zu Fraktalen

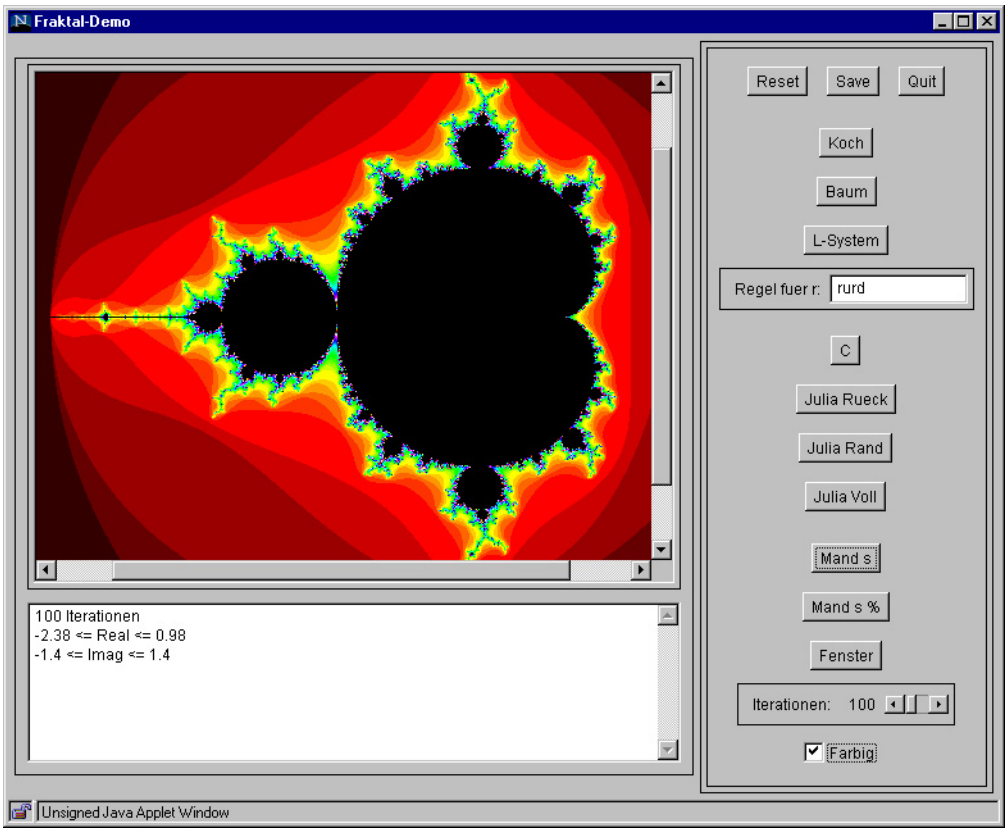


Abbildung 11.13: Screenshot vom Fraktale-Applet

11.9 Iterierte Funktionensysteme

Iterierte Funktionssysteme sind in der Lage, mit wenigen Regeln komplexe, natürlich aussehende Bilder zu erzeugen. Hierbei wird eine Folge von Punkten im $\mathbf{R}^2$ durch fortgesetzte Anwendung von affinen Transformationen durchlaufen. Jede Transformation ist definiert durch eine 2×2- Deformationsmatrix A, einen Translationsvektor b und eine Anwendungswahrscheinlichkeit w, wodurch ein Punkt $\bar{x}$ abgebildet wird auf $A\bar{x} + b$. Zum Beispiel erzeugen folgende 4 Regeln das Farnblatt in Abbildung 11.14 :

	A	b	w
r_1	$\begin{pmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{pmatrix}$	$\begin{pmatrix} 0.0 \\ 1.6 \end{pmatrix}$	85 %
r_2	$\begin{pmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{pmatrix}$	$\begin{pmatrix} 0.0 \\ 1.6 \end{pmatrix}$	7 %
r_3	$\begin{pmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{pmatrix}$	$\begin{pmatrix} 0.0 \\ 0.44 \end{pmatrix}$	7 %
r_4	$\begin{pmatrix} 0.00 & 0.00 \\ 0.00 & 0.16 \end{pmatrix}$	$\begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix}$	1 %

Abbildung 11.14: Vom Iterierten-Funktionen-System erzeugter Farn

Eine alternative Sichtweise zur Definition eines fraktalen Bildes verlangt die Selbstähnlichkeit von Teilen des Bildes mit dem Ganzen.

D_1	
D_2	D_3

Beispielsweise sei ein Rechteck R gesucht, so daß sich sein Inhalt, jeweils auf ein Viertel verkleinert, im linken oberen, linken unteren und rechten unteren Quadranten wiederfindet. Beginnend mit einem beliebigen Rechteckinhalt werden die 3 Quadranten so lange als skalierte Versionen des Gesamtrechtecks ersetzt, bis wir nahe am Fixpunkt dieser Iterationen angelangt sind. Das Ergebnis ist das sogenannte *Sierpinsky-Dreieck*.

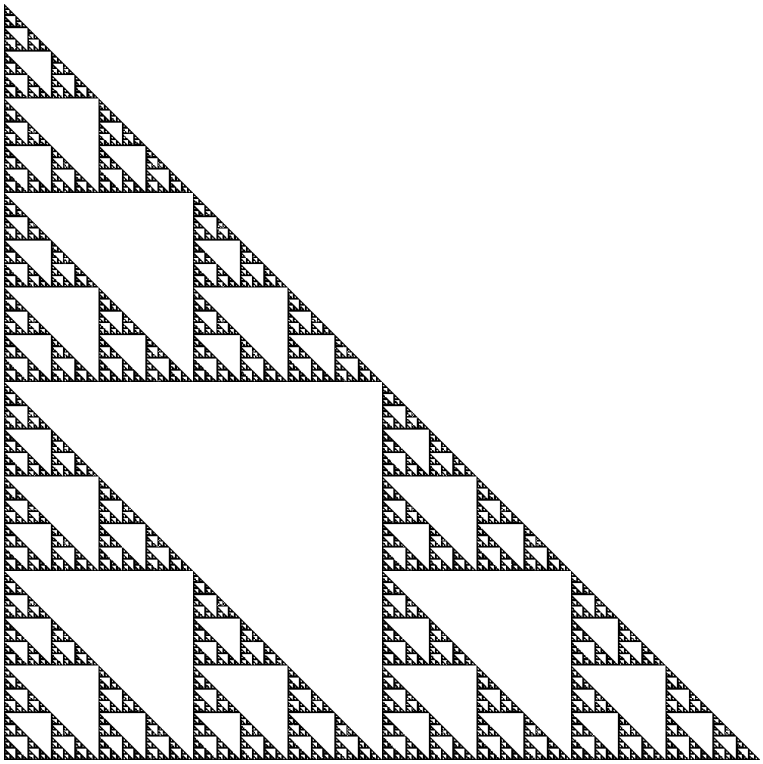


Abbildung 11.15: Sierpinsky-Dreieck

11.10 Java-Applet zu Iterierten Funktionensystemen

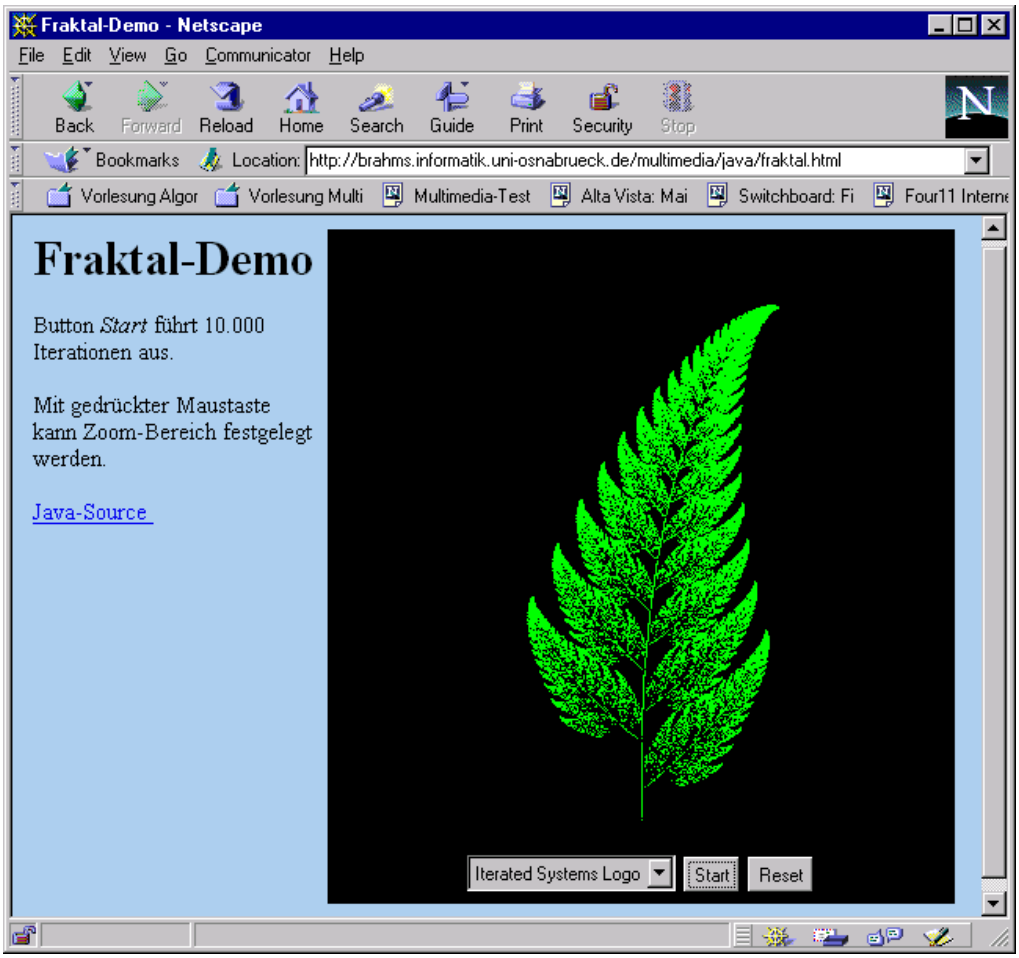


Abbildung 11.16: Screenshot vom Iterierte-Funktionen-Systeme-Applet