Kapitel 5

Clipping

Ziel: Nur den Teil eines Objekts darstellen, der innerhalb eines Fensters sichtbar ist.

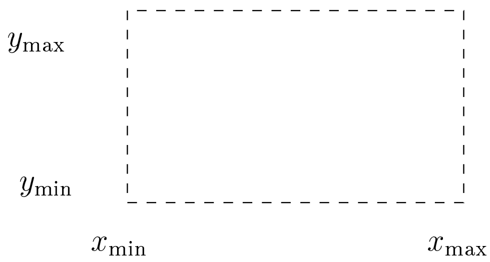


Abbildung 5.1: Clip-Fenster

5.1 Clipping von Linien

Zu einer Menge von Linien sind jeweils neue Anfangs- und Endpunkte zu bestimmen, welche komplett im Clip-Fenster liegen.

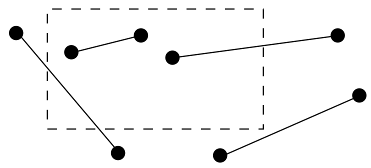


Abbildung 5.2: Ausgangslage beim Linien-Clipping

Idee von Cohen & Sutherland

Teile Ebene anhand des Clip-Fensters in 9 Bereiche ein, beschrieben durch 4-Bit-Bereichscode:

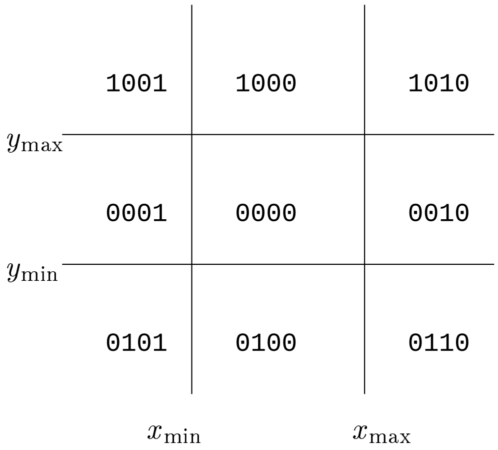


Abbildung 5.3: Bereichscodes nach Cohen & Sutherland

Bit 0:	links vom Fenster
Bit 1:	rechts vom Fenster
Bit 2:	unter dem Fenster
Bit 3:	über dem Fenster

Sei $\overline{P_1P_2}$ eine Linie. Dann gilt bei einer bitweisen Verknüpfung:

code (P_1) AND *code* $(P_2) \neq 0 \Rightarrow \overline{P_1P_2}$ komplett außerhalb (beide Punkte auf derselben Seite)

code (P_1) OR *code* $(P_2) = 0 \Rightarrow \overline{P_1P_2}$ komplett innerhalb (beide Punkte im Clip-Fenster)

In den anderen Fällen wird $\overline{P_1P_2}$ mit einer Fensterkante geschnitten und der Test mit der verkürzten Linie erneut ausgeführt.

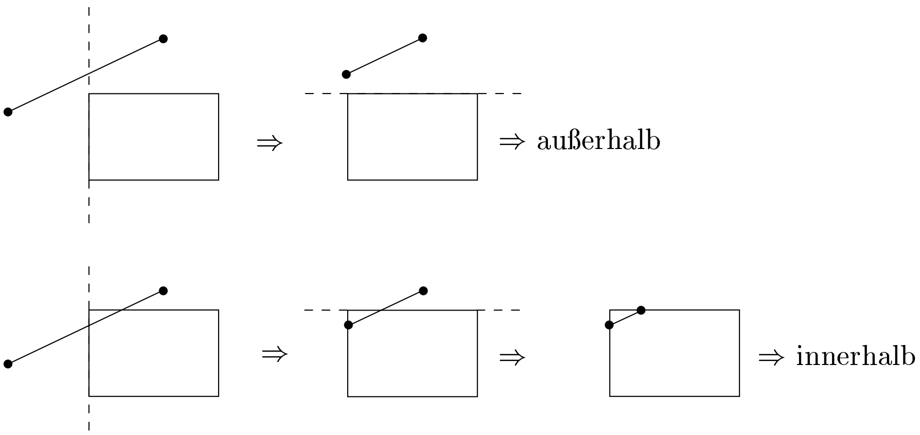


Abbildung 5.4: Möglichkeiten bei zwei außerhalb liegenden Punkten

```
/*************************************************************************************/
/*                                                                                   */
/*              Clippen von Linien an einem Fenster nach Cohen-Sutherland            */
/*                                                                                   */
/*************************************************************************************/

private static final byte EMPTY  = 0;
private static final byte LEFT   = 1;
private static final byte RIGHT  = 2;          // 4-Bit-Bereichscodes
private static final byte BOTTOM = 4;
private static final byte TOP    = 8;

private int xmin, xmax, ymin, ymax;            // Clip-Fensterraender

private byte region_code(                      // liefert den region-code
                          Point P )            // fuer den Punkt P
{
        byte c;

        c = EMPTY;
        if (P.x < xmin) c = LEFT; else
        if (P.x > xmax) c = RIGHT;
        if (P.y < ymin) c |= BOTTOM; else
        if (P.y > ymax) c |= TOP;
        return(c);
}


private void set_clip_window(         // setzt die Variablen xmin, ymin, xmax, ymax
                Point P,              // anhand des Ursprungs P des Clip-Fensters
                Point delta )         // und anhand seiner Breite/Hoehe delta
        {
                xmin = P.x; xmax = P.x + delta.x;
                ymin = P.y; ymax = P.y + delta.y;
        }


private boolean cohen_sutherland(     // liefert true,
                Point p1, Point p2,   // falls die Gerade p1-p2 sichtbar ist
                Point Q1, Point Q2)   // liefert ggf. sichtbaren Teil Q1-Q2 zurueck
{
        boolean finite_slope;         // true falls Gerade p1-p2 nicht senkrecht laeuft
        double slope = 0.0;           // Steigung der Geraden p1-p2
        byte C, C1, C2;               // 4-Bit-Bereichs-Code
        Point Q = new Point();        // zu berechnender Schnittpunkt mit Gerade

        Point P1 = new Point(p1);     // lokale
        Point P2 = new Point(p2);     // Variablen
        finite_slope = (P1.x != P2.x);
        if (finite_slope) slope = (double)(P2.y-P1.y)/(double)(P2.x-P1.x);
        C1 = region_code(P1);
        C2 = region_code(P2);
```

```
        while ((C1 != EMPTY) || (C2 != EMPTY))  { // mind. ein Endpunkt noch ausserhalb

            if ((C1&C2) != EMPTY) return(false);  // beide auf derselben Seite ausserhalb
            else
            {
                if (C1 == EMPTY) C = C2; else C = C1; // C ist ausserhalb. Berechne
                                                      // einen Schnittpunkt mit den
                                                      // verlaengerten Fensterkanten

                if ((C & LEFT) != EMPTY) {          // schneide mit linker Fenster-Kante
                        Q.x = xmin;
                        Q.y = (int)((Q.x-P1.x)*slope + P1.y);
                        } else

                if ((C & RIGHT) != EMPTY){         // schneide mit rechter Fenster-Kante
                        Q.x = xmax;
                        Q.y = (int)((Q.x-P1.x)*slope + P1.y);
                        } else

                if ((C & BOTTOM) != EMPTY) {       // schneide mit unterer Fenster-Kante
                        Q.y = ymin;
                        if (finite_slope)
                                Q.x = (int)((Q.y-P1.y)/slope + P1.x); else
                                Q.x = P1.x;
                        }else

                if ((C & TOP) != EMPTY) {          // schneide mit oberer Fenster-Kante
                        Q.y = ymax;
                        if (finite_slope)
                                Q.x = (int)((Q.y-P1.y)/slope + P1.x); else
                                Q.x = P1.x;
                        }

                if (C==C1) {
                        P1.x = Q.x; P1.y = Q.y; C1 = region_code (P1);
                } else {
                        P2.x = Q.x; P2.y = Q.y; C2 = region_code (P2);
                }
            }

        }
        // uebergib Anfang und Ende des sichtbaren Teils
        Q1.x=P1.x; Q1.y=P1.y;
        Q2.x=P2.x; Q2.y=P2.y;

        return(true);
}
```

5.2 Clipping von Polygonen

Für das Clippen von Polygonen reicht es nicht, jede beteiligte Kante zu clippen:

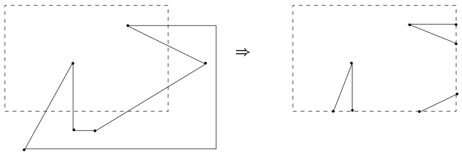


Abbildung 5.5: Zerfall eines Polygons bei reinem Linien-Clippen

Vielmehr müssen zusätzlich die Ein- und Austrittspunkte verbunden werden, um nach dem Clipping wieder ein Polygon zu erhalten:

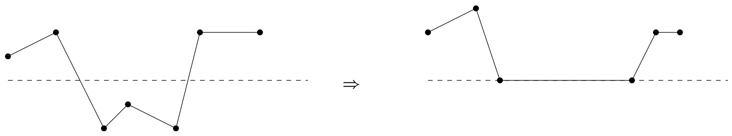


Abbildung 5.6: Verbinden der Ein- und Austrittspunkte

Obacht: Bzgl. der Ecken des Clip-Windows ist eine Spezialbehandlung erforderlich:

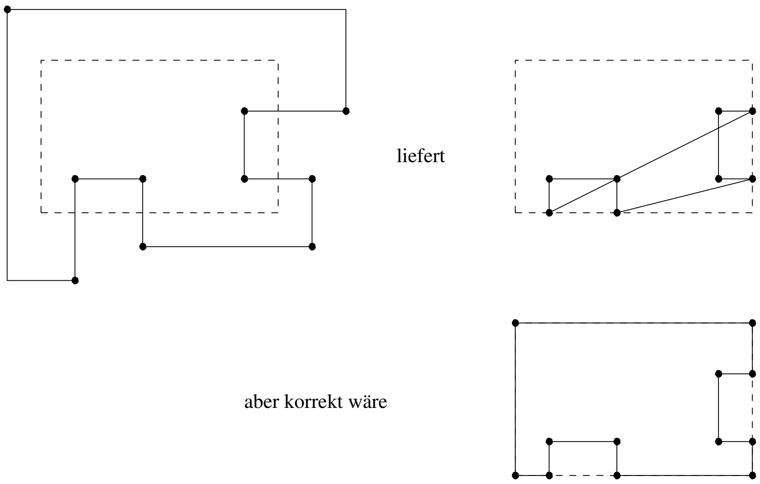


Abbildung 5.7: Spezialbehandlung an den Clipfensterecken

Der **Sutherland & Hodgman-Algorithmus** clippt an vier Fensterkanten nacheinander:

```
fuer jede Clipping-Gerade E tue:
  fuer jeden Polygonpunkt P tue:
    falls P sichtbar: uebernimm ihn
    falls Kante von P zu seinem Nachfolger E schneidet:
          uebernimm den Schnittpunkt
```

```
/*****************************************************************************************/
/*                                                                                       */
/*            Clippen von Polygonen an einem Fenster nach Sutherland-Hodgmann            */
/*                                                                                       */
/*****************************************************************************************/

private boolean On_Visible_Side(                    /* liefert true, falls        */
                        Point P,                    /* Punkt p auf der sichtbaren  */
                        int wert,                   /* Seite der Kante wert liegt  */
                        int fall)                   /* gemaess Fall fall           */
{
        switch (fall) {

                case 0: return (P.x >= wert);       /* linke  Fenster-Kante        */
                case 1: return (P.y >= wert);       /* untere Fenster-Kante        */
                case 2: return (P.x <= wert);       /* rechte Fenster-Kante        */
                case 3: return (P.y <= wert);       /* obere  Fenster-Kante        */
        }

        return false;
}


private boolean Intersection(                /* liefert true,                          */
                        Point P1, Point P2,  /* falls sich die Linie P1-P2             */
                        int wert,            /* mit Clipping-Kante wert schneidet      */
                        int fall,            /* gemaess Fall fall                      */
                        Point I)             /* liefert den Schnittpunkt I zurueck     */
{
        boolean P1_vis, P2_vis;              /* true, falls P1 bzw P2 sichtbar         */
        double slope;                        /* Steigung der Geraden P1-P2             */
                                             /* y = (x-P1.x)*slope + P1.y              */

        P1_vis = On_Visible_Side(P1,wert,fall);      /* P1 sichtbar ?                  */
        P2_vis = On_Visible_Side(P2,wert,fall);      /* P2 sichtbar ?                  */

        if ((P1_vis && P2_vis) || (!P1_vis && !P2_vis)) return (false);/*kein Schnittpkt*/
        else
        {
                slope = (double)(P2.y-P1.y) / (double) (P2.x-P1.x);   /* Steigung      */

                if (fall%2==1) {
                        I.x = (int)((wert-P1.y) / slope) + P1.x;
                        I.y = wert;
                } else
                {
                        I.x = wert;
                        I.y = (int)((wert-P1.x) * slope) + P1.y;
                }
                return(true);
        }
}
```

```
private int sutherland_hodgman(                  /* fuehrt ein Clipping durch          */
                        int num_points,          /* fuer num_points Punkte             */
                        Point[] points,          /* der Polygonliste Points            */
                        int wert,                /* bezueglich Clipping-Kante wert     */
                        int fall)                /* gemaess Fall fall                  */
                                                 /* liefert neue Anzahl Punkte         */
{
    int     i;
    int     old_ptr = 0;
    int     new_ptr = 0;
    Point   S, I;
    Point[] hpoints = new Point[MAX_POINTS];

    if (num_points > 2) {

        points[num_points] = points[0];

        while (old_ptr < num_points) {

            S = points[old_ptr++];
            if (On_Visible_Side(S,wert,fall)) hpoints[new_ptr++] = S;

            I = new Point();
            if (Intersection(S,points[old_ptr],wert,fall,I)) hpoints[new_ptr++] = I;
        }

        for (i=0; i<new_ptr; i++) points[i]=hpoints[i];
        return new_ptr;
}
```

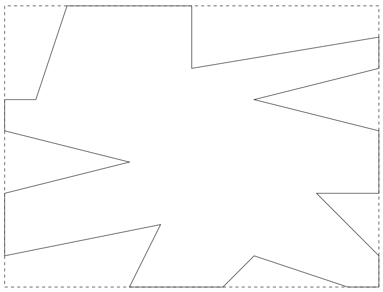


Abbildung 5.8: Vom Sutherland-Hodgman-Algorithmus geclipptes Polygon

5.3 Beispiel-Applet zu 2D-Operationen

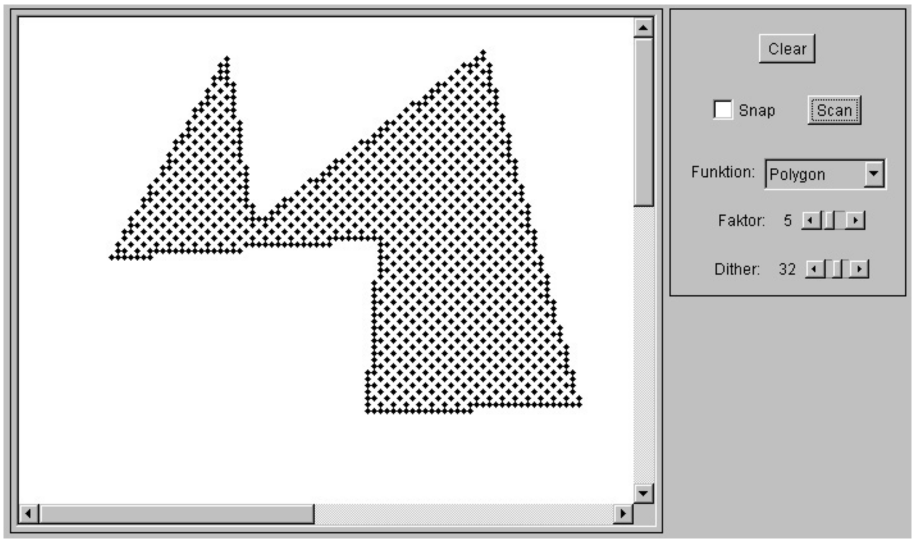


Abbildung 5.9: Screenshot vom 2D-Basic-Applet

Kapitel 6

2D-Transformationen

Transformationen eines Objekts werden durch Operationen auf den Definitionspunkten realisiert. Durch 2 Punkte definierte Rechtecke müssen zunächst zu Polygonen gemacht werden.

6.1 Translation

Gradlinige Verschiebung um einen Translationsvektor $T = (t_x, t_y)$, d.h. $P' := P + T$

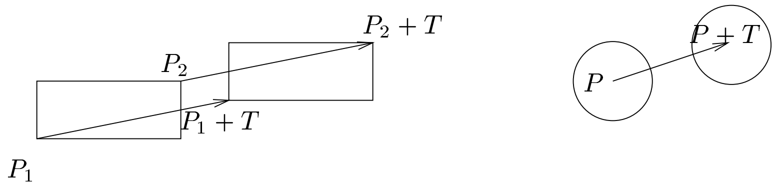


Abbildung 6.1: Translation von Rechtecken und Kreisen

6.2 Skalierung

Vergrößerung bzw. Verkleinerung bzgl. eines Fixpunktes. Zunächst liege der Fixpunkt im Ursprung $(0, 0)$.

$$(x', y') \quad := \quad (s_x \cdot x, s_y \cdot y)$$

$s_x = s_y \Rightarrow$ uniforme Skalierung; $s_x \neq s_y \Rightarrow$ Verzerrung

Abbildung 6.2 zeigt ein Beispiel für eine Skalierung bezüglich des Ursprungs mit den Faktoren $s_x = 4,\ s_y = 2$.

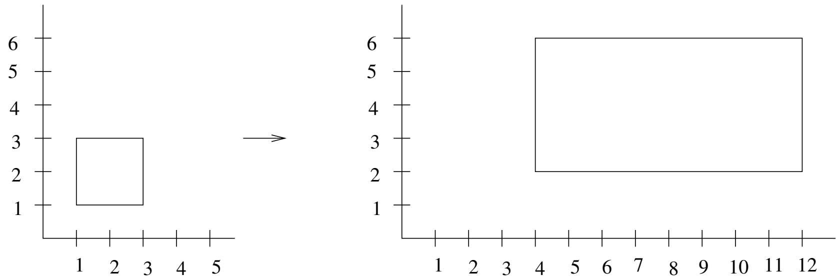


Abbildung 6.2: Skalierung mit $s_x = 4$ und $s_y = 2$

Bei Wahl eines beliebigen Fixpunktes (Z_x, Z_y) folgt für den Punkt P:

1. Translation um $(-Z_x, -Z_y)$ liefert P_1.
2. Skalierung mit (s_x, s_y) liefert P_2.
3. Translation um (Z_x, Z_y) liefert $P_3 = P'$.

Die neuen Koordiaten berechnen sich dann wie folgt:

$$\Rightarrow (x', y') := ((x - Z_x) \cdot s_x + Z_x, (y - Z_y) \cdot s_y + Z_y)$$

Vorteilhafter bei mehreren Objekten:

$$\begin{aligned} (x', y') &= (x \cdot s_x + d_x, y \cdot s_y + d_y) \\ & \quad \text{mit } d_x = Z_x \cdot (1 - s_x), d_y = Z_y \cdot (1 - s_y) \end{aligned}$$

Abbildung 6.3 zeigt ein Beispiel für eine Skalierung bezüglich des Punktes $Z = (1, 3)$ mit den Faktoren $s_x = 3$, $s_y = 2$.

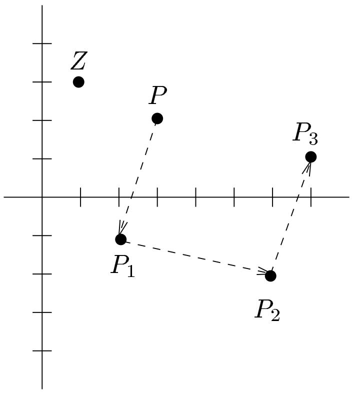
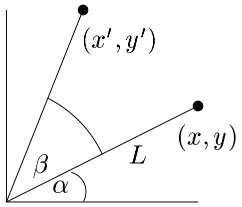


Abbildung 6.3: Skalierung bzgl. Punkt (1,3) mit Faktoren $s_x = 3$ und $s_y = 2$

6.3 Rotation

Drehung des Objekts bzgl. eines Fixpunktes um einen Winkel β. Der Fixpunkt liege im Ursprung.



$L = \sqrt{x^2 + y^2}$
$\sin(\alpha) = y/L$
$\cos(\alpha) = x/L$

Abbildung 6.4: Rotation um den Winkel β bzgl. des Ursprungs

$$
\begin{aligned}
\sin(\alpha+\beta) &= \cos(\beta)\cdot\sin(\alpha) + \sin(\beta)\cdot\cos(\alpha) \\
\cos(\alpha+\beta) &= \cos(\beta)\cdot\cos(\alpha) - \sin(\beta)\cdot\sin(\alpha) \\
\sin(\alpha+\beta) = y'/L &= \cos(\beta)\cdot\sin(\alpha) + \sin(\beta)\cdot\cos(\alpha) \\
\Rightarrow y' &= L\cdot\cos(\beta)\cdot\frac{y}{L} + L\cdot\sin(\beta)\cdot\frac{x}{L} \\
\Rightarrow y' &= y\cdot\cos(\beta) + x\cdot\sin(\beta) \\
\cos(\alpha+\beta) = x'/L &= \cos(\beta)\cdot\cos(\alpha) - \sin(\beta)\cdot\sin(\alpha) \\
\Rightarrow x' &= L\cdot\cos(\beta)\cdot\frac{x}{L} - \sin(\beta)\cdot\frac{y}{L}\cdot L \\
\Rightarrow x' &= x\cdot\cos(\beta) - y\cdot\sin(\beta)
\end{aligned}
$$

Positive Werte für den Drehwinkel β bewirken eine Rotation gegen den Uhrzeigersinn.
Bei Wahl eines beliebigen Rotationszentrums (R_x, R_y) folgt für den Punkt P

1. Translation um $(-R_x, -R_y)$ liefert P_1
2. Rotation bzgl. Ursprung um Winkel β liefert P_2
3. Translation um (R_x, R_y) liefert $P_3 = P'$

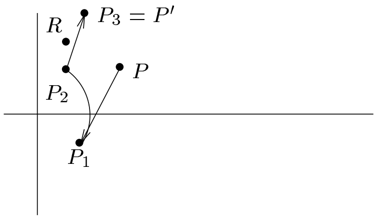


Abbildung 6.5: Rotation des Punktes P um 90 Grad bzgl Punkt R

```
/*************************************************************************************/
/*                                                                                   */
/*                       Skalieren und Rotieren eines Polygons                       */
/*                       Obacht: Da die Polygon-Koordinaten ganzzahlig sind,         */
/*                       verstaerken sich die Rundungsfehler nach jeder Transformation   */
/*                                                                                   */
/*************************************************************************************/

private static final double DEG_TO_RAD = Math.PI/180.0; // Umrechnung Grad in Bogenmass


private void skalier_polygon(           // skaliert ein Polygon
                        Point[] points, // gespeichert im Array points
                        int n,          // mit n Eckpunkten
                        double sx,      // in x-Richtung um den Faktor sx
                        double sy,      // in y-Richtung um den Faktor sy
                        Point  z)       // bezueglich des Punktes z
{
        int i;
        for (i=0; i< n; i++) {

                points[i].x = (int)((points[i].x - z.x ) * sx + z.x + 0.5);
                points[i].y = (int)((points[i].y - z.y ) * sy + z.y + 0.5);

        }
}


private void rotate_polygon(            // rotiert ein Polygon
                        Point[] points, // gespeichert im Array points
                        int   n,        // mit n Eckpunkten
                        int   g,        // um einen Winkel von g Grad
                        Point z)        // bezueglich des Punktes z
{
        int i;
double beta = g * DEG_TO_RAD;   // Winkel im Bogenmass
        Point P = new Point();

        for (i=0; i < n; i++) {

                P.x = (int)((points[i].x - z.x ) * Math.cos( beta ) -
                      (points[i].y - z.y ) * Math.sin( beta ) + z.x + 0.5);

                P.y = (int)((points[i].y - z.y ) * Math.cos( beta ) +
                      (points[i].x - z.x ) * Math.sin( beta ) + z.y + 0.5);

                points[i].x = P.x;
                points[i].y = P.y;

        }
}
```

6.4 Homogene Koordinaten

Häufig werden mehrere Transformationen hintereinander auf ein Objekt angewendet. Es entstehen Rundungsfehler, wenn nach jeder Einzel-Transformation die ganzzahligen Koordinaten bestimmt werden. Deshalb sollten mehrere Transformationen zu einer zusammengesetzt werden.

Ein Punkt $P = (x, y)$ hat die *homogenen* Koordinaten $[x_h, y_h, w]$ mit $w \neq 0$ und

$$\begin{aligned} x_h &= x \cdot w \\ y_h &= y \cdot w \end{aligned}$$

Beispiel: Das homogene Koordinatentripel $[6, 8, 2]$ gehört zum Punkt $P = (3, 4)$. Derselbe Punkt hat die homogenen Koordiaten $[3, 4, 1]$.

Ein Richtungsvektor $R = (x, y)$ hat die homogenen Koordinaten $[x, y, 0]$.

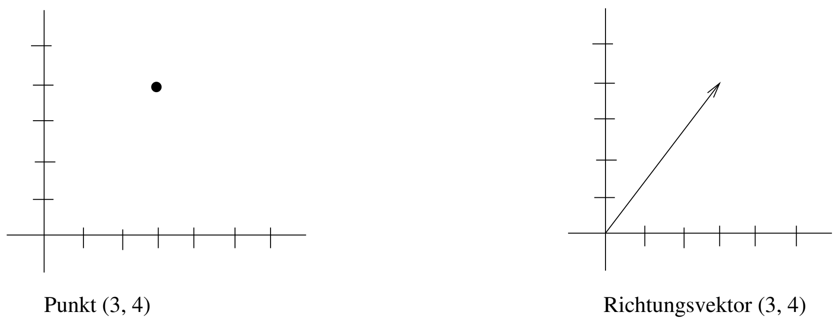


Abbildung 6.6: Punkt (3,4) und Richtungsvektor (3,4)

Die Transformationen Translation, Skalierung und Rotation werden nun als 3×3-Matrizen realisiert. Zusammengesetzte Transformationen ergeben sich durch Matrix-Multiplikation.

Translation

$$[x', y', 1] := [x, y, 1] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix} = [x + t_x, y + t_y, 1]$$

Skalierung

$$[x', y', 1] := [x, y, 1] \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x \cdot s_x, y \cdot s_y, 1]$$

Rotation

$$[x', y', 1] := [x, y, 1] \cdot \begin{bmatrix} \cos(\beta) & \sin(\beta) & 0 \\ -\sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x \cdot \cos(\beta) - y \cdot \sin(\beta), x \cdot \sin(\beta) + y \cdot \cos(\beta), 1]$$

Für die Auswertung einer zusammengesetzten Transformation

- gilt das Assoziativgesetz, d.h. $A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$,
- gilt im allgemeinen nicht das Kommutativgesetz, d.h. $A \cdot B \neq B \cdot A$.

Beispiel: Translation um $(-3,-1)$ mit anschließender Rotation um $90°$ ist verschieden von Rotation um $90°$ mit anschließender Translation um $(-3,-1)$.

Beispiel: Rotation bzgl. $(3,1)$ um $60°$

Matrix für Translation um $(-3,-1)$ lautet

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & -1 & 1 \end{bmatrix}$$

Matrix für Rotation um $60°$ lautet

$$B = \begin{bmatrix} 0.50 & 0.86 & 0.00 \\ -0.86 & 0.50 & 0.00 \\ 0.00 & 0.00 & 1.00 \end{bmatrix}$$

Matrix für Translation um $(3,1)$ lautet

$$C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 1 & 1 \end{bmatrix}$$

Matrix für gesamte Transformation lautet

$$D = A \cdot B \cdot C = \begin{bmatrix} 0.50 & 0.86 & 0 \\ -0.86 & 0.50 & 0 \\ 2.36 & -2.08 & 1 \end{bmatrix}$$

6.5 Allgemeine Transformationen

Weitere Transformationen, die sich durch Matrizen darstellen lassen, sind

- Spiegelung an einer beliebigen Geraden,
- Spiegelung an einem Punkt,
- Scherung.

Bei der Scherung in x bleiben die y-Werte konstant, und die x-Werte werden proportional zu den y-Werten horizontal verschoben, d.h. $x' = x + Sch_x \cdot y$.
Die Transformationsmatrix lautet:

$$\begin{bmatrix} 1 & 0 & 0 \\ Sch_x & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ bzw. } \begin{bmatrix} 1 & Sch_y & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

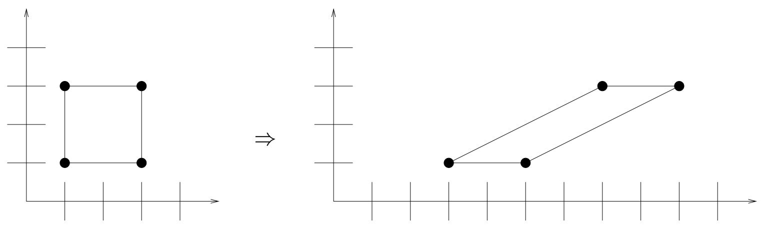


Abbildung 6.7: Scherung mit $Sch_x = 2$

6.6 Raster-Transformationen

Es werden nicht die Definitionspunkte, sondern die Pixel im Frame-Buffer modifiziert:

- Verschieben eines Fensterinhalts (Move),
- Kopieren eines Fensterinhalts (Copy),
- Vergrößern eines Fensterinhalts (Zoom).

Statt Pixelwerte zu setzen, können auch AND-, OR-, XOR-Verknüpfungen verwendet werden. Es gilt (x XOR y) XOR $y = x$. XOR ist daher geeignet, ein Objekt über den Schirm zu bewegen, ohne den Hintergrund zu zerstören.

```
zeichne Objekt mit XOR
repeat
   loesche Objekt mit XOR
   modifiziere Objekt_Koordinaten
   zeichne Objekt mit XOR
until Objekt am Ziel
```

Allerdings ändert das bewegte Objekt je nach Hintergrundfarbe sein Aussehen.

Kapitel 7

Kurven

Problem: Spezifiziere und zeichne eine beliebige "glatte" Kurve.

7.1 Algebraischer Ansatz

Gegeben n Stützpunkte.
Bestimme Polynom $(n-1)$-ten Grades

$$y = a_{n-1} \cdot x^{n-1} + a_{n-2} \cdot x^{n-2} + \ldots + a_1 \cdot x + a_0$$

welches durch alle Stützpunkte läuft.
Problem: Bei der Auswertung des Polynoms treten wegen der hohen Potenzen hohe Rechenzeiten und große Rundungsfehler auf. Außerdem oszilliert die Kurve.

7.2 Splines

Gegeben n Stützpunkte. Wähle $n-1$ Kurven kleiner Ordnung für den Verlauf innerhalb der $n-1$ Intervalle.

Kurven erster Ordnung (Geraden) ergeben nur den Linienzug.

Kurven zweiter Ordnung (Parabeln) sind an den Intervallgrenzen nur einmal differenzierbar.

Kurven dritter Ordnung mit stetigen ersten und zweiten Ableitungen an den Intervallgrenzen (kubische Splines) können zu einer Gesamtkurve mit weichen Übergängen an den Nahtstellen zusammengesetzt werden.

Gesucht: Approximation einer Kurve $f(t)$, die durch n vorgegebene Punkte laufen soll.

Bemerkung: f ist in parametrischer Form gegeben (d.h. $x = g(t), y = h(t)$), da die explizite Form $y = f(x)$ pro x-Wert nur einen y-Wert erlaubt.

Die n Stützpunkte seien $f(t_i)$, die resultierenden Intervalle seien $[t_i, t_{i+1}]$, die für das Intervall i zuständige Funktion sei $f_i(t)$, $1 \leq i \leq n-1$.

Jedes f_i hat die Form

$$\begin{aligned} f_i(t) &= f(t_i) + a_i \cdot (t - t_i) + b_i \cdot (t - t_i)^2 + c_i \cdot (t - t_i)^3 \\ & \quad t_i \leq t \leq t_{i+1},\ i = 1, \ldots, n-1 \end{aligned}$$

Es muß gelten

$$\begin{array}{rcllll} f_i(t_{i+1}) & = & f_{i+1}(t_{i+1}) & \text{für} & i = 1, \ldots, n-2 & \text{gleicher Wert} \\ f_i'(t_{i+1}) & = & f_{i+1}'(t_{i+1}) & \text{für} & i = 1, \ldots, n-2 & \text{gleiche Steigung} \\ f_i''(t_{i+1}) & = & f_{i+1}''(t_{i+1}) & \text{für} & i = 1, \ldots, n-2 & \text{gleiche Steigungsänderung} \end{array}$$

Nach Vorgabe von a_1 und a_{n-1} entsteht ein Gleichungssystem mit $n-2$ Gleichungen und $n-2$ Unbekannten $a_2, a_3, \ldots, a_{n-2}$.

Die Werte von a_1 und a_{n-1} können z.B. ermittelt werden durch Festlegung $f''(t_1) = f''(t_n) = 0$ (natürliche Spline-Interpolation).

Für die Folge t_i reicht jede monoton wachsende Folge. Geeignet ist z.B. die Folge der euklidischen Abstände zwischen den Stützpunkten. Statt den Abstand $d = \sqrt{dx^2 + dy^2}$ zu berechnen, verwendet man die Näherungsformel $\frac{1}{3} \cdot (dx + dy + 2 \cdot \max(dx, dy))$.

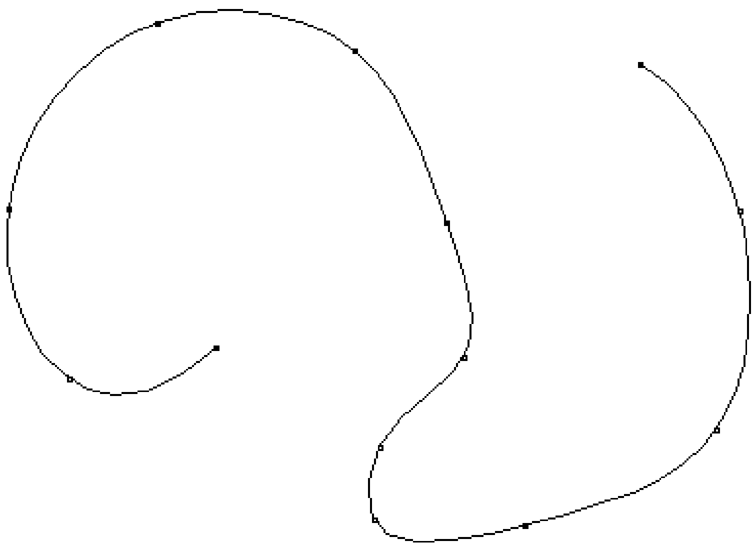


Abbildung 7.1: Vom Spline-Algorithmus erzeugte Kurveninterpolation

```
/*****************************************************************************/
/*              Spline-Interpolation durch Polynome 3. Grades               */
/*****************************************************************************/

private static double[] calcfx = new double[MAX_POINTS];       // x-Funktionswerte
private static double[] calcfy = new double[MAX_POINTS];       // y-Funktionswerte
private static double[] x = new double[MAX_POINTS];  // x-Koordinaten der Stuetzpunkte
private static double[] y = new double[MAX_POINTS];  // y-Koordinaten der Stuetzpunkte
private static double[] t = new double[MAX_POINTS];  // t-Werte der Stuetzpunkte

void besetze_arrays(                         // berechnet Intervallgrenzen
              int point_cnt,                 // Anzahl der Polygonpunkte
              Point punkt[])                 // Punktliste
{
      int i;
      double ax,ay,dd;

      for (i=0; i<point_cnt; i++) {          // besetze x- und y-Koordinaten
              x[i] = (double) punkt[i+1].x;
              y[i] = (double) punkt[i+1].y;}
      t[0] = 0.0;                            // besetze t-Werte mit dem
      for (i=1; i<point_cnt; i++) {          // Euklidischen Abstand der Stuetzpunke
              ax=Math.abs(x[i]-x[i-1]);
              ay=Math.abs(y[i]-y[i-1]);
              dd = ax + ay;
              if (ax > ay) dd = dd+2*ax;     // Naeherungsformel fuer Euklid:
                      else dd = dd+2*ay;     // Abstand = 1/3*(dx+dy+2*max{dx,dy})
              t[i] = t[i-1]+dd/3;
      }
}



void curve_fitting(                        // berechnet die Approximation der Kurve
      int point_cnt,                       // fuer point_cnt Stuetzpunkte
      Point punkt[],                       // uebergeben im Array punkt
      int spline_size)                     // fuer spline_size Interpolationswerte
{
      int i,n=0;
      besetze_arrays(point_cnt, punkt);              // erzeuge Intervallgrenzen und
                                                     // Gleitkommaversion von punkt
      cubic_spline(point_cnt, t, x, spline_size, calcfx);  // bestimme Interpol.werte
      cubic_spline(point_cnt, t, y, spline_size, calcfy);  // bestimme Interpol.werte
      for (i=0; i<spline_size; i++)                        // und uebertrage sie
        spline_punkt[i+1] = new Point((int)calcfx[i],(int)calcfy[i]);
}
```

```
void cubic_spline(                    // loest Gleichungssystem zur Bestimmung
                                      // der Koeffizienten a,b,c
              int n,                  // Anzahl der Stuetzpunkte
              double t[], double f[], // Stuetzpunkte in der Form (t[i],f[t[i]])
              int spline_size,        // Anzahl der Interpolationswerte
              double calcf[])         // Ausgabearray: Interpolationswerte
{
      int i,j;
      double a[] = new double[MAX_POINTS];
      double b[] = new double[MAX_POINTS];
      double c[] = new double[MAX_POINTS];
      double delta_t[] = new double[MAX_POINTS];
      double D[] = new double[MAX_POINTS];
      double m[] = new double[MAX_POINTS];
      double k[] = new double[MAX_POINTS];
      double bh,dh,e,h,wt,dt;
      for (i=1; i<n; i++) {
              delta_t[i] = t[i]-t[i-1];
              D[i] = (f[i]-f[i-1])/delta_t[i];
      }
      m[0] = delta_t[2];
      delta_t[0] = t[2]-t[0];
      h = delta_t[1];
      k[0] = ((h + 2*delta_t[0])*D[1]*delta_t[2]+h*h*D[2])/delta_t[0];
      for (i=1; i<(n-1); i++) {
              h = -delta_t[i+1]/m[i-1];
              k[i] = h*k[i-1]+3*(delta_t[i]*D[i+1]+delta_t[i+1]*D[i]);
              m[i] = h *delta_t[i-1] + 2* (delta_t[i] + delta_t[i+1]);
      }
      h = t[n-1]-t[n-3];
      dh = delta_t[n-1];
      k[n-1] = ((dh+h+h)*D[n-1]*delta_t[n-2] +  dh*dh*D[n-2])/h;
      h = -h/m[n-2];
      m[n-1] = delta_t[n-2];
      m[n-1] = h*delta_t[n-2] + m[n-1];
      a[n-1] = (h*k[n-2]+k[n-1])/m[n-1];
      for (i=n-2; i>=0; i--) a[i] = (k[i]-delta_t[i]*a[i+1])/m[i];
      for (i=1; i<n; i++) {
              dh = D[i]; bh = delta_t[i];
              e = a[i-1]+a[i]-dh-dh;
              b[i-1] = 2*(dh-a[i-1]-e)/bh;
              c[i-1] = 6*e/(bh*bh);
      }
      wt = 0; j=0;
      dt = t[n-1]/(double)(spline_size-1);
      for (i=0; i<spline_size; i++) {
              while ((t[j+1]<wt)&&(j<spline_size)) j++;
              h = wt - t[j];
              calcf[i] = f[j]+h*(a[j]+h*(b[j]+h*c[j]/3)/2);
              wt = wt+dt;
      }
      calcf[spline_size-1]= f[n-1];
}
```

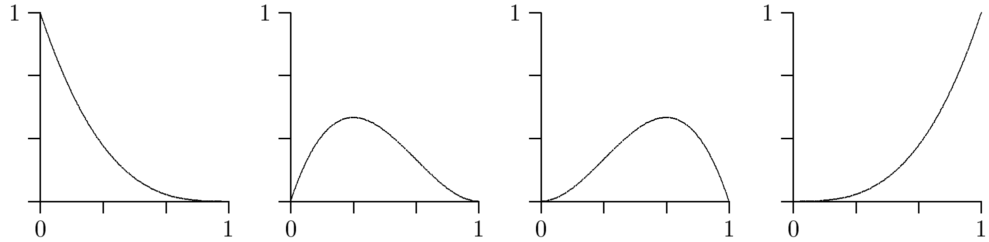
7.3 Bézier-Kurven

Spezifiziere die Kurve durch $n+1$ Stützpunkte $P_0, P_1, \ldots, P_n$. Die Kurve läuft nicht durch alle Stützpunkte, sondern wird von ihnen beeinflußt.

$$P(t) = \sum_{i=0}^{n} B_{i,n}(t) \cdot P_i, \quad 0 \leq t \leq 1$$

Der Punkt P_i wird gewichtet mit Hilfe eines Bernstein-Polynoms.

$$B_{i,n}(t) = \frac{n!}{i! \cdot (n-i)!} \cdot t^i \cdot (1-t)^{n-i}, \; i = 0, ..., n$$



$B_{0,3} = (1-t)^3$ $B_{1,3} = 3 \cdot t \cdot (1-t)^2$ $B_{2,3} = 3 \cdot t^2 \cdot (1-t)$ $B_{3,3} = t^3$

Abbildung 7.2: Verlauf der Bernsteinpolynome für n=3

Die wichtigsten Eigenschaften der Bernsteinpolynome:

- alle Bernsteinpolynome sind positiv auf $[0,1]$,
- für jedes feste t gilt $\sum_{i=0}^{n} B_{i,n}(t) = 1$,
- $B_{i,n}(t) = B_{n-i,n}(1-t)$,
- $\frac{dB_{i,n}(t)}{dt} = n \cdot (B_{i-1,n-1}(t) - B_{i,n-1}(t)), i > 0$,
- Maxima von $B_{i,n}(t)$ in $[0,1]$ an den Stellen $t = \frac{i}{n}, i = 0, \ldots, n$.

Da alle Bernsteinpolynome für $0 < t < 1$ ungleich Null sind, beeinflussen alle Stützpunkte in diesem Intervall den Kurvenverlauf.

Die Kurve beginnt im Stützpunkt P_0 tangential zur Geraden $\overline{P_0 P_1}$, endet im Stützpunkt P_n tangential zur Geraden $\overline{P_{n-1} P_n}$ und verläuft innerhalb der konvexen Hülle der Stützpunkte. Somit können mehrere Bézier-Kurven aneinandergesetzt werden durch Identifikation von Endpunkt und Anfangspunkt aufeinanderfolgender Bézierkurven. Einen stetig differenzierbaren Übergang erreicht man bei Kollinearität der Punkte $P_{n-1}, P_n = Q_0, Q_1$.

Berechnung der Bézier-Kurve nach de Casteljau

Um die aufwendige Auswertung der Bernsteinpolynome zu vermeiden, wurde von de Casteljau ein Iterationsverfahren vorgeschlagen, welches durch fortgesetztes Zerteilen von Linien den gewünschten Kurvenpunkt approximieren kann.

Es gilt:

$$P(t)_{0,n} = (1-t) \cdot P(t)_{0,n-1} + t \cdot P(t)_{1,n}$$

wobei die tiefgestellten Indizes angeben, welche Stützpunkte in die Berechnung einfließen. D.h. eine Bézier-Kurve vom Grad n läßt sich durch zwei Bézier-Kurven vom Grad $n-1$ definieren, indem für ein festes t die Punkte beider Kurven berechnet werden, und die Verbindungsstrecke im Verhältnis t geteilt wird.

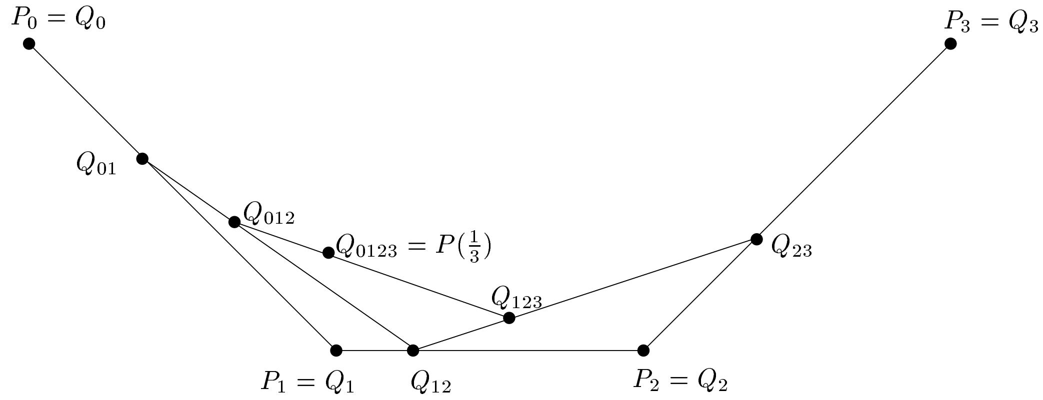


Abbildung 7.3: Approximation einer Bezier-Kurve nach de Castelau

Abbildung 7.3 zeigt die Berechnung eines Kurvenpunktes für $t = \frac{1}{3}$. Die Indizes geben an, welche Stützpunkte beteiligt sind. Abbildung **??** zeigt das Ergebnis dieser Iteration für zwei aneinanderliegende Bézierkurven bei Rekursionstiefe 3.

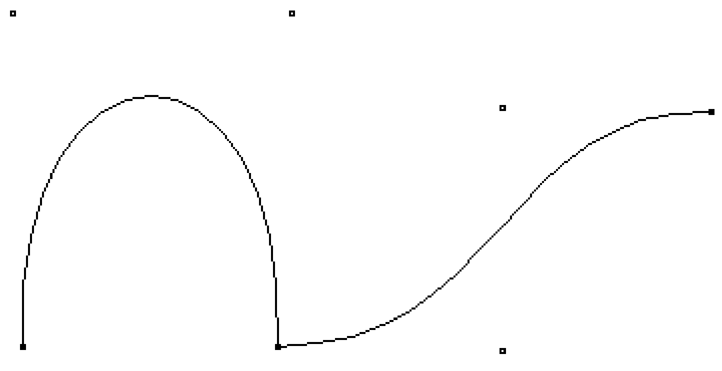


Abbildung 7.4: Vom De Casteljau-Algorithmus mit Rekursionstiefe 3 gezeichnete Bezierkurven

```
/*****************************************************************************/
/*                                                                           */
/*                 Zeichnen einer Bezierkurve 3. Grades nach De Casteljau    */
/*                                                                           */
/*****************************************************************************/

private void bezier(                             // Zeichne Bezierkurve 3. Grades
                    double px0, double py0,      // anhand des Stuetzpunktes p0
                    double px1, double py1,      // anhand des Stuetzpunktes p1
                    double px2, double py2,      // anhand des Stuetzpunktes p2
                    double px3, double py3,      // anhand des Stuetzpunktes p3
                    int depth)                   // aktuelle Rekursionstiefe
{
        double qx01,qy01,qx12,qy12,qx23,qy23,qx012,qy012,qx123,qy123,qx0123,qy0123;
                                                 // Hilfspunkte

        if (depth > iter)                        // Iterationstiefe erreicht
             set_line(new Point( (int)px0, (int)py0), new Point( (int)px3, (int)py3));
        else {
                depth++;

                qx01   = (px0+px1)/2;     qy01   = (py0+py1)/2;
                qx12   = (px1+px2)/2;     qy12   = (py1+py2)/2;
                qx23   = (px2+px3)/2;     qy23   = (py2+py3)/2;
                qx012  = (qx01+qx12)/2;   qy012  = (qy01+qy12)/2;
                qx123  = (qx12+qx23)/2;   qy123  = (qy12+qy23)/2;
                qx0123 = (qx012+qx123)/2; qy0123 = (qy012+qy123)/2;

                bezier (px0,py0,qx01,qy01,qx012,qy012,qx0123,qy0123,depth);
                bezier (qx0123,qy0123,qx123,qy123,qx23,qy23,px3,py3,depth);
        }
}
```

7.4 B-Splines

Es sollen nun nicht alle Stützpunkte Einfluß auf den gesamten Kurvenverlauf haben und der Grad der Polynome soll unabhängig von der Zahl der Stützpunkte sein.

Spezifiziere die Kurve durch $n+1$ Stützpunkte $P_0, \ldots, P_n$ und einen Knotenvektor $T = (t_0, t_1, \ldots, t_{n+k}), t_j \leq t_{j+1}$.

$$P(t) = \sum_{i=0}^{n} N_{i,k}(t) \cdot P_i$$

Die Punkte P_i wirken sich nur auf maximal k Kurvenabschnitte aus und werden gewichtet durch die Basis des B-Splines, Polynome $N_{i,k}(t)$ vom Grad $k-1$ $(0 \leq i \leq n)$:

$$
\begin{aligned}
N_{i,1}(t) &= \begin{cases} 1 & \text{falls } t_i < t_{i+1} \;\; \textit{und} \;\; t_i \leq t \leq t_{i+1} \\ 0 & \text{sonst} \end{cases} \\
N_{i,k}(t) &= \frac{t - t_i}{t_{i+k-1} - t_i} \cdot N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} \cdot N_{i+1,k-1}(t), \;\; k > 1
\end{aligned}
$$

Bei Division durch Null wird der Quotient gleich 0 gesetzt.
Der Knotenvektor wird häufig wie folgt gewählt für $0 \leq j \leq n + k$:

$$
t_j = \begin{cases} 0 & \text{falls } j < k \\ j - k + 1 & \text{falls } k \leq j \leq n \\ n - k + 2 & \text{falls } j > n \end{cases}
$$

Hierdurch wird t im Intervall $[0, n - k + 2]$ definiert.

Beispiel: $k = 3$, $n = 4$ ergibt einen uniformen quadratischen B-Spline mit dem Knotenvektor $T = (0, 0, 0, 1, 2, 3, 3, 3)$. Die Stützpunkte P_0, P_1, P_2, P_3, P_4 haben nur lokal Einfluß auf den Kurvenverlauf, und zwar in den Intervallen $t \in [0, 1], [0, 2], [0, 3], [1, 3], [2, 3]$.

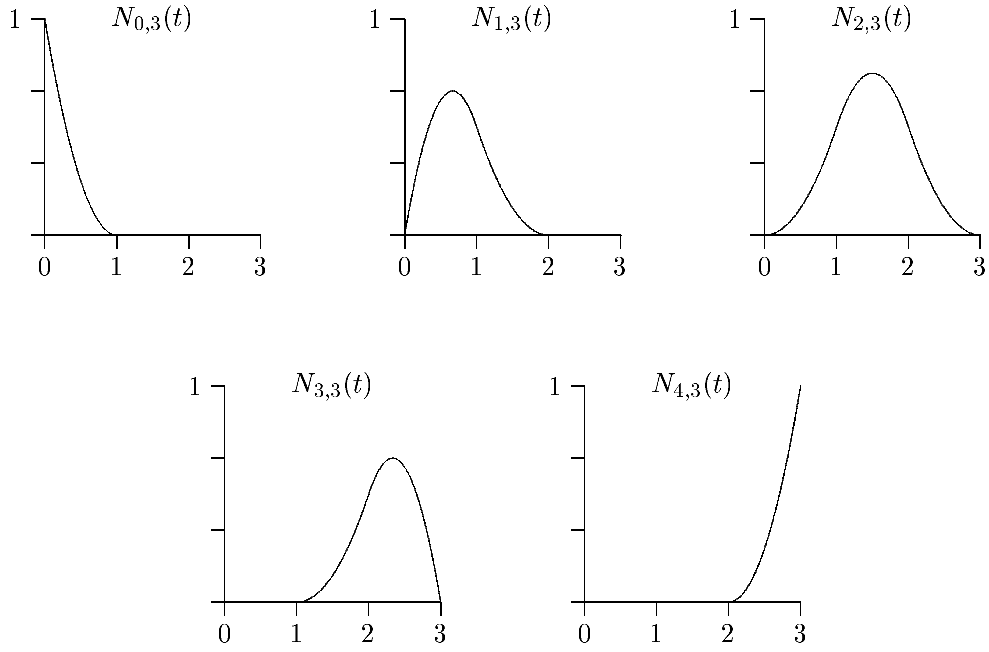


Abbildung 7.5: Verlauf der Gewichtungspolynome $N_{i,k}$ für $k = 3$

Sonderfall

Für $k = n + 1$ ergibt sich für den Knotenvektor der Sonderfall $T = (\underbrace{0, ..., 0}_{k \text{ mal}}, \underbrace{1, ..., 1}_{k \text{ mal}})$.

Die zugehörigen B-Splinefunktionen haben die schon bekannte Form

$$N_{i,k}(t) = \frac{(k-1)!}{i! \cdot (k-1-i)!} \cdot t^i \cdot (1-t)^{k-1-i}, \; i = 0, ..., k-1$$

Somit lassen sich also die B-Splinefunktionen als Verallgemeinerung der Bernsteinpolynome auffassen.

Wie bei den Bernsteinpolynomen gilt auch für die B-Splinefunktionen $N_{i,k}$, daß sie positiv sind, und $\sum_{i=0}^{n} N_{i,k} = 1$.

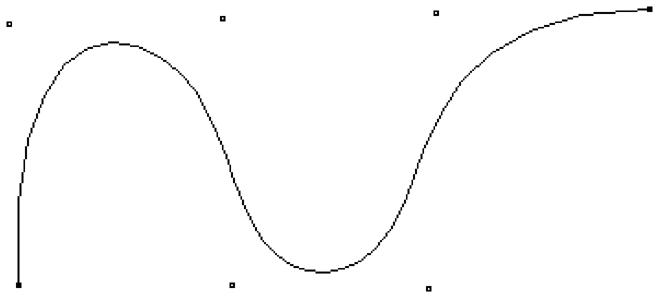


Abbildung 7.6: B-Spline mit $n = 6$, $k = 4$ und 35 Interpolationspunkten

7.5 NURBS

Die allgemeinste Form zum Zeichnen von Kurven wird realisiert durch die sogenannten *NURBS (nonuniform rational basis splines).*

Dabei bedeutet

nonuniform: Der Abstand zwischen den Werten des Kontrollvektors ist ungleichmäßig (und nicht *uniform* wie bei den *B-Splines*).

rational: Der Einfluß eines Kontrollpunktes wird über ein Gewicht gesteuert, welches durch den Quotienten (*ratio*) zweier Polynome gebildet wird.

NURBS sind (im Gegensatz zu allgemeinen *B-Splines*) in der Lage, Kreisstücke zu beschreiben und sind invariant bzgl. perspektivischer Projektion. Dies bedeutet, daß zur Projektion einer Kurve nicht alle Kurvenpunkte projeziert werden müssen, sondern es reicht, die Stützpunkte zu projezieren und dann zu einer Kurve zu verbinden.

7.6 Beispiel-Applet zu Splines

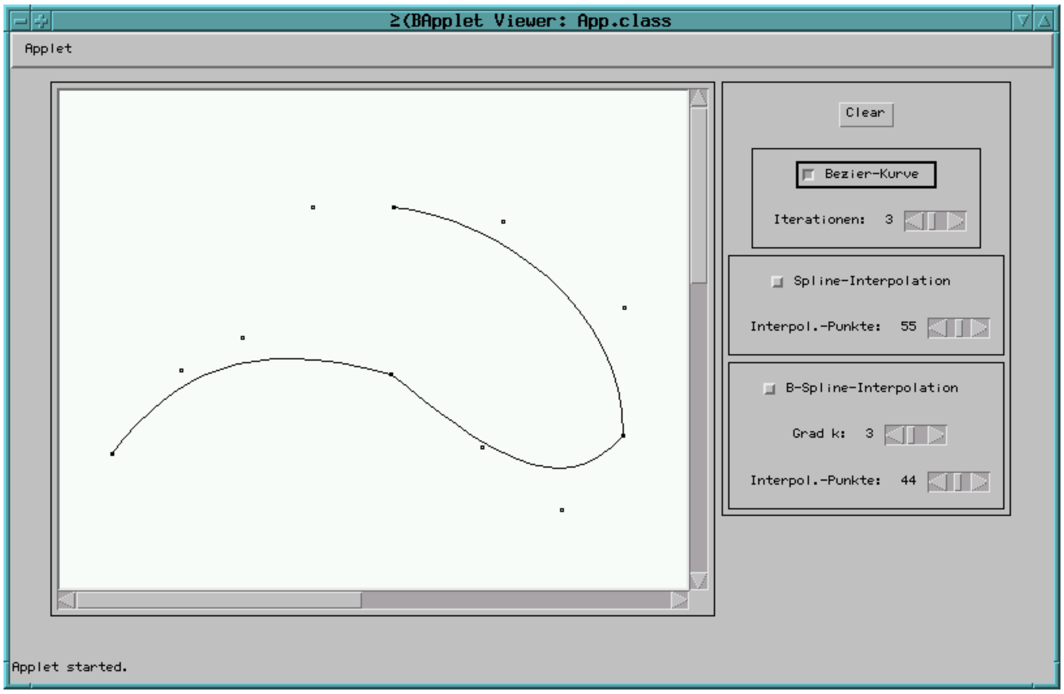


Abbildung 7.7: Screenshot vom Splines-Applet

7.7 Macromedia Flash

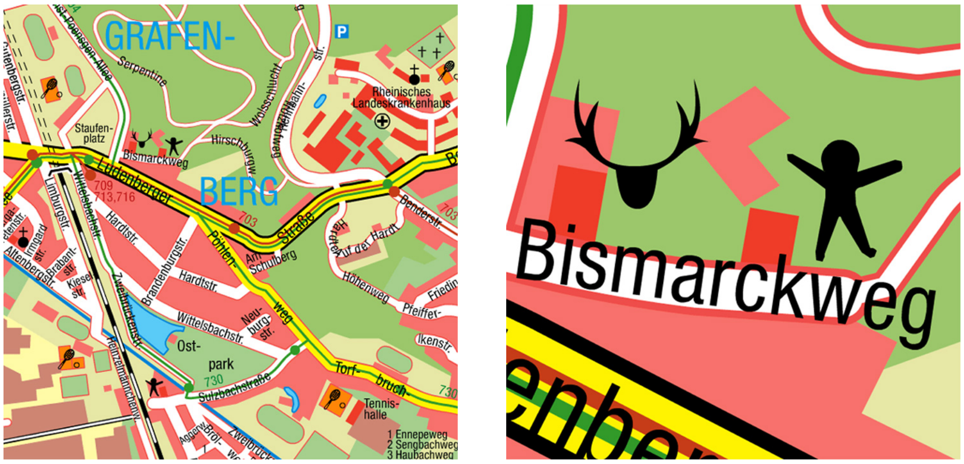
Macromedia Flash ist ein Werkzeug zum Editieren und Animieren von zweidimensionalen Vektorgrafiken. Durch sein kompaktes Speicherformat ist es besonders geeignet für plakative, animierte Grafiken im Internet. Zum Abspielen in einem Web-Browser ist das Macromedia-Flash-Plugin erforderlich.

Beispiel für Flash in der Kartographie vom Verwaltungsverlag:
`http://www.stadtplan.net/brd/nrw/duesseldorf/home.html` .

Vortrag über Flash im Rahmen des Seminars *Film & Animation* von Tanja Schniederberend:
`http://www-lehre.inf.uos.de/fua/fua04/html/index.html`

Flash-Animation von Andreas Nie & Patrick Tschorn:
`http://www-lehre.inf.uos.de/mmii/flash/Fine_Menue_2.htm`

Trailer zum Universitätsjubiläum von Andreas Nie & Patrick Tschorn:
`http://www.uni-osnabrueck.de/jubilaeum/trailer/trailer.html`



Gesamtansicht des Stadtplans | Teilansicht des Stadtplans

Abbildung 7.8: Rasterfreies Zoomen durch Vektorgrafik

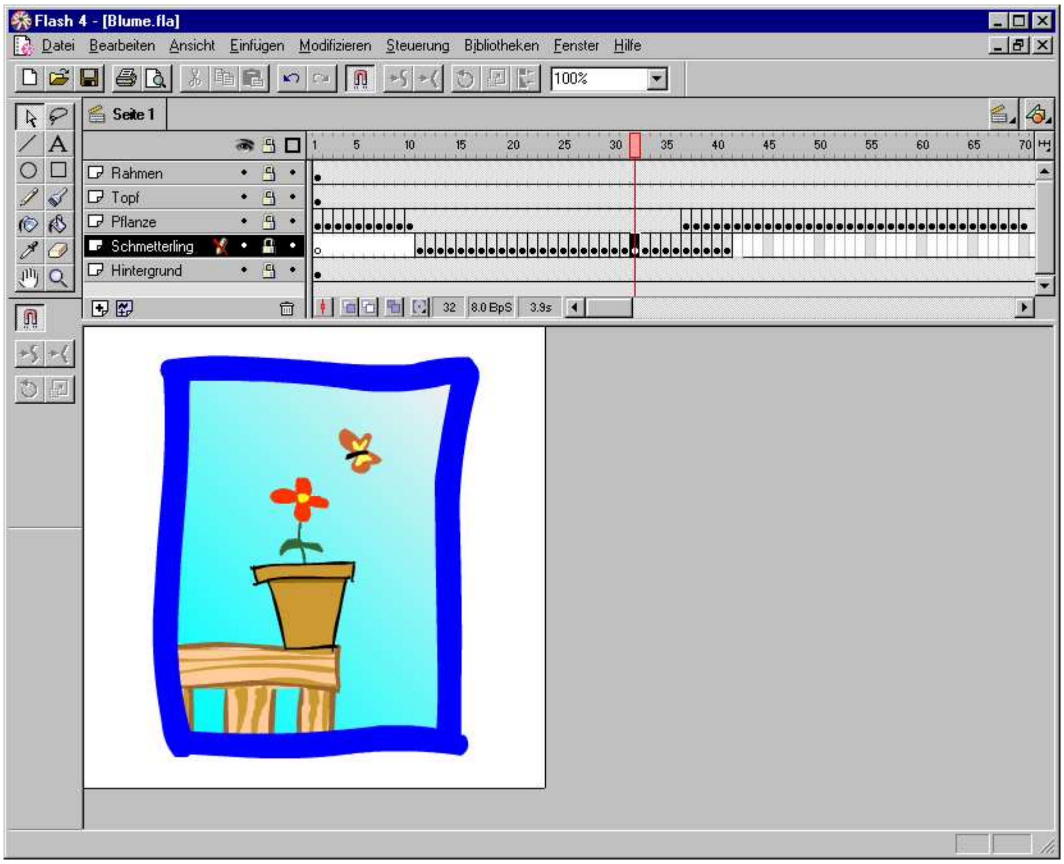


Abbildung 7.9: Screenshot vom Vektorgrafikwerkzeug Macromedia Flash

Kapitel 8

Fraktale

8.1 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 8.1: Farnblatt

8.2 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 8.2 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.

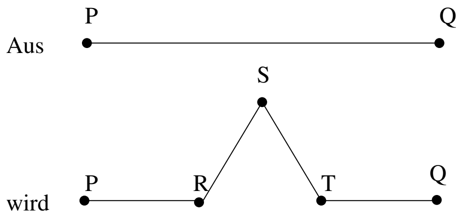
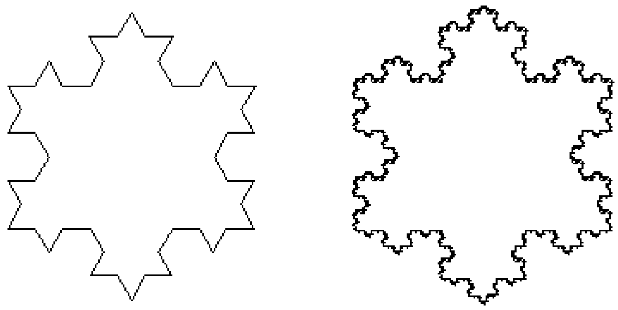


Abbildung 8.2: Iterationsvorschrift zur Koch'schen Schneeflocke



48 Polygonpunkte 3072 Polygonpunkte

Abbildung 8.3: Koch'sche Schneeflocke

8.3 Fraktale Dimension

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

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 $\dfrac{1}{N^{\frac{1}{2}}}$,

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

Sind N und r bekannt, läßt sich D bestimmen:

$r = \dfrac{1}{N^{\frac{1}{D}}} \Rightarrow r \cdot N^{\frac{1}{D}} = 1 \Rightarrow N^{\frac{1}{D}} = \dfrac{1}{r} \Rightarrow \dfrac{1}{D} \cdot \log(N) = \log(\dfrac{1}{r})$

$\Rightarrow D = \dfrac{\log(N)}{\log(\frac{1}{r})}$

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 $r = \frac{1}{3}$.

```
/*************************************************************************/
/*                                                                       */
/*                        Koch'sche Schneeflocke                         */
/*                                                                       */
/*************************************************************************/

private int koch_stern(         // wendet eine Koch-Iteration an auf
        Point poly_punkt[],     // das Polygon poly_punkt
        int poly_punkt_cnt)     // mit poly_point_cnt Punkten
                                // neue Anzahl Punkte wird zurueckgeliefert
{
        int i,j;
        double x0,x1,x2,x3,x4,y0,y1,y2,y3,y4;
        Point[] HPunkt = new Point[MAX_POINTS];
        j=0;

        for (i=0; i < poly_point_cnt; i++)
        {
                x0 = poly_punkt[i].x;
                y0 = poly_punkt[i].y;

                x4 = poly_punkt[(i+1)%(poly_point_cnt)].x;
                y4 = poly_punkt[(i+1)%(poly_point_cnt)].y;

                x1 = (x4-x0)/3+x0;
                y1 = (y4-y0)/3+y0;

                x3 = x4 - (x4-x0)/3;
                y3 = y4 - (y4-y0)/3;

                x2 = (int)(x1+ (x3-x1)*cos_60 - (y3-y1)*sin_60);
                y2 = (int)(y1+ (y3-y1)*cos_60 + (x3-x1)*sin_60);

                HPunkt[j]   = new Point();
                HPunkt[j+1]   = new Point();
                HPunkt[j+2]   = new Point();
                HPunkt[j+3]   = new Point();

                HPunkt[j]        = poly_punkt[i];
                HPunkt[j+1].x        = (int)x1;
                HPunkt[j+1].y        = (int)y1;
                HPunkt[j+2].x        = (int)x2;
                HPunkt[j+2].y        = (int)y2;
                HPunkt[j+3].x        = (int)x3;
                HPunkt[j+3].y        = (int)y3;
                j=j+4;
        }

        for (i=0; i < j; i++)                // neu berechnetes Polygon
                poly_punkt[i] = HPunkt[i];   // kopieren

        return j;                            // neue Anzahl Punkte zurueck
}
```

8.4 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:

$$\text{Alphabet } \sum = \{r, u, l, d\} \text{ für right, up, left, down}$$

$$\begin{array}{rccl} \text{Regelmenge } f = \{ & r & \Rightarrow & r\ u\ r\ d\ d\ r\ u\ r, \\ & u & \Rightarrow & u\ l\ u\ r\ r\ u\ l\ u, \\ & l & \Rightarrow & l\ d\ l\ u\ u\ l\ d\ l, \\ & d & \Rightarrow & d\ r\ d\ l\ l\ d\ r\ d\ \} \end{array}$$

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 8.4: 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 $r = \frac{1}{4}$.

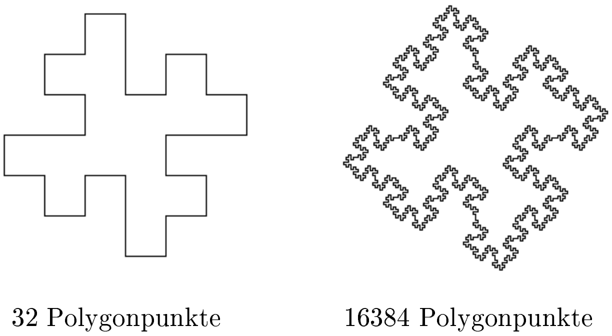


Abbildung 8.5: L-System

8.5 Baumstrukturen

Gegeben ein "Ast", repräsentiert durch ein 5-seitiges Polygon:

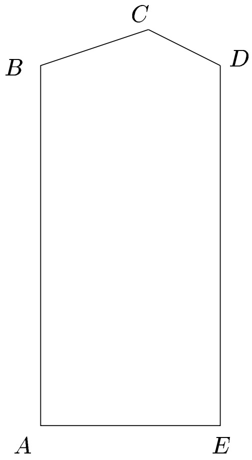
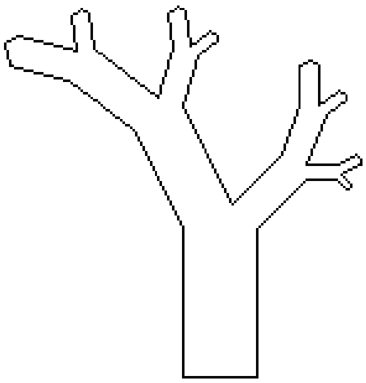
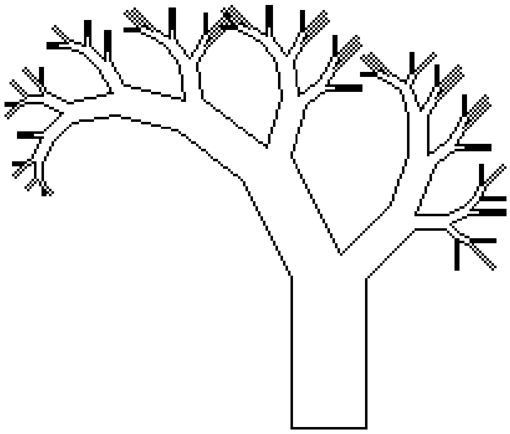


Abbildung 8.6: Ausgangspolygon für Baum

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 8.7: Vom Baum-Algorithmus erzeugter Baum

```
/****************************************************************************/
/*                                                                          */
/*                              Baum                                        */
/*                                                                          */
/****************************************************************************/

public void baum()             // wendet eine Iteration auf Ansatzstuecke an
        //Point ansatz[],      // alte und neue Folge von Ansatzstuecken
        //int   anzahl,        // Anzahl der Anzsatzstueckendpunkte
        //Point poly_punkt[],  // alte und neue Folge von Polygonecken
        //int   poly_point_cnt // Anzahl der Polygonecken
{
        int i,j,k,p;
        double a,b,l,si_de,co_de;
        Point P;
        Point Q;
        Point A;
        Point B;
        Point C;
        Point[] ast = new Point[3];
        Point[] Hpunkt = new Point[MAX_POINTS];
        Point[] Hansatz = new Point[MAX_POINTS];

        DoublePoint[] schema = new DoublePoint[3];

        ast[0] = new Point();
        ast[1] = new Point();
        ast[2] = new Point();

        schema[0] = new DoublePoint();
        schema[1] = new DoublePoint();
        schema[2] = new DoublePoint();

        schema[0].x = 0.0;                  // generiere Schema
        schema[0].y = 2.0;
        schema[1].x = 2.0/3.0;
        schema[1].y = 7.0/3.0;
        schema[2].x = 1.0;
        schema[2].y = 2.0;

        j=0;                                // Index fuer neue Ansatzstuecke
        k=0;                                // Index fuer neue Polygonpunkte
        for (p=0; p < anzahl; p=p+2)
        {
                P = ansatz[p];
                Q = ansatz[p+1];

                a     = P.y - Q.y;          // bestimme l, sinus, cosinus
                b     = Q.x - P.x;
                l     = Math.sqrt(a*a + b*b);
                si_de = a/l;
                co_de = b/l;
```

```
        for (i=0; i<3; i++)         // fuehre Skalierung und Rotation aus
        {
          ast[i].x = P.x + (int)(l*schema[i].x*co_de-l*schema[i].y*si_de);
          ast[i].y = P.y - (int)(l*schema[i].y*co_de+l*schema[i].x*si_de);
        }

        A = ast[0];                 // kreiere neue Ast-Punkte
        B = ast[1];
        C = ast[2];

        if (anzahl>2) del_line(P, Q); // loesche Ansatzstueck

        set_line(P,A);               // zeichne Figur P,A,B,C,Q
        set_line(A,B);
        set_line(B,C);
        set_line(C,Q);
                                     // speichere neue Ansatzstuecke
        Hansatz[j]          = new Point(A);
        Hansatz[j+1]        = new Point(B);
        Hansatz[j+2]        = new Point(B);
        Hansatz[j+3]        = new Point(C);
        j=j+4;
    }

    i=0;                                 // Index fuer altes Polygon
    k=0;                                 // Index fuer neues Polygon
    for (p=0; p < anzahl; p=p+2)         // Index fuer alte Ansatzstuecke
    {
        while  ((poly_punkt[i].x !=ansatz[p].x) ||
                (poly_punkt[i].y !=ansatz[p].y))
                              // uebernimm unveraenderte Polygonpunkte
            Hpunkt[k++] = poly_punkt[i++];

        Hpunkt[k]   = ansatz[p];      // uebernimm neue Polygonpunke
        Hpunkt[k+1] = Hansatz[2*p];
        Hpunkt[k+2] = Hansatz[2*p+1];
        Hpunkt[k+3] = Hansatz[2*p+3];
        Hpunkt[k+4] = ansatz[p+1];
        k+=4;
        i++;
    }

    while (i < poly_point_cnt) Hpunkt[k++]=poly_punkt[i++];

    for (i=0; i < j; i++)
        ansatz[i] = Hansatz[i];   // neu berechnete Ansatzstuecke kopieren
    anzahl=j;

    for (i=0; i < k; i++)
        poly_punkt[i] = Hpunkt[i];// neu berechnete Polygonpunkte kopieren
    poly_point_cnt=k;
}
```

8.6 Mandelbrot-Menge

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

```
public class Komplex
{
        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.

```
Komplex f(Komplex 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 wächst der Betrag beliebig, für manche c konvergiert er zu einem Fixpunkt.
Die *Mandelbrotmenge* besteht aus solchen komplexen Zahlen c, die beim Startwert $z = 0$ zur Konvergenz führen.
Um die Mandelbrotmenge grafisch darzustellen, definiert man

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

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 Komplex(Point p, Komplex 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).

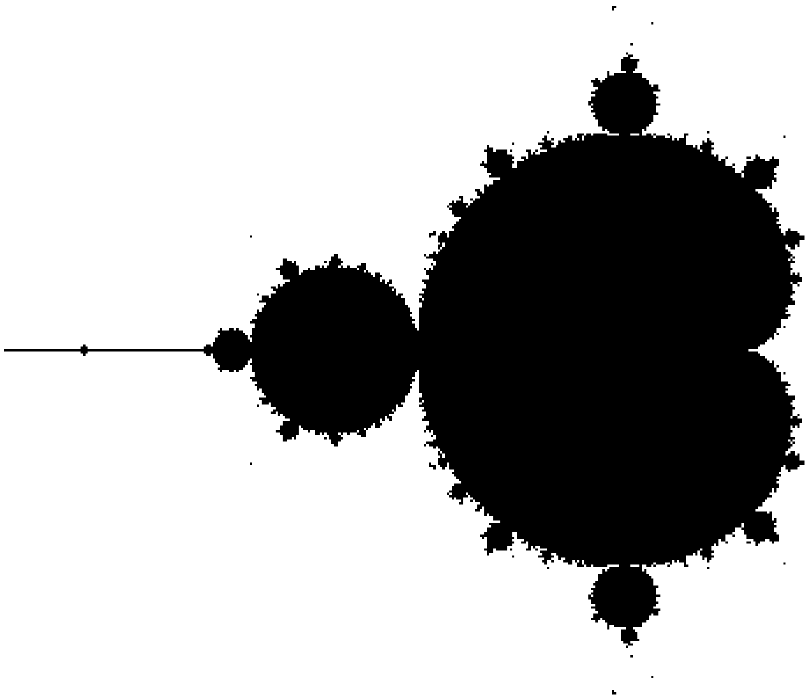


Abbildung 8.8: Mandelbrotmenge: $-2.2 \leq z.re \leq 0.6$, $-1.2 \leq z.im \leq 1.2$, 100 Iterationen

Implementation der Mandelbrot-Menge

Sobald während des Iterierens der Betrag von $z > 2 \Rightarrow$ Folge konvergiert definitiv nicht.
Falls Betrag von z nach z.B. 100 Iterationen noch $< 2 \Rightarrow$ Folge konvergiert vermutlich.
D.h., bei Erreichen einer vorgegebenen Iterationenzahl wird das Pixel, welches der komplexen Zahl c entspricht, schwarz gefärbt, da c vermutlich zur Konvergenz führt.

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

     for (p.x = 0; p.x < WIDTH;  p.x++)
     for (p.y = 0; p.y < HEIGHT; p.y++)
     {
          zaehler = 0;
          c       = new Komplex(p, start, schritt);
          z       = new Komplex(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 nicht zur Konvergenz geführt haben, weiter zu klassifizieren, setzt man

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

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])
```

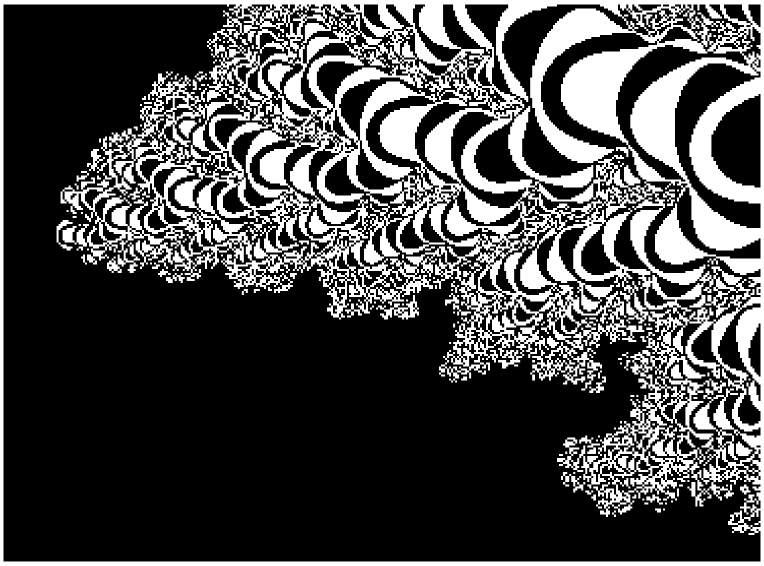


Abbildung 8.9: Mandelbrotmenge: $-1.2548 \le Re \le -1.2544$, $0.3816 \le Im \le 0.3822$, 100 Iterationen

```
/*****************************************************************************/
/*                           Mandelbrotmenge                                 */
/*****************************************************************************/
void  mandel(                    // berechnet und zeichnet Mandelbrotmenge ab
        Komplex start,           // linker oberer Ecke start der komplexen Flaeche
        double  schritt,         // mit Schrittweite schritt und
        int     max_iter,        // Iterationsanzahl max_iter (muss gerade sein)
        int     fall)            // gemaess Fall fall
                                 // Abbruch, wenn Quadrat des Betrags >= 4
                                 // oder wenn Iterationszahl erreicht ist
{
    Komplex c = new Komplex(0.0, 0.0);
    Komplex z = new Komplex(0.0, 0.0);
    Point   p = new Point();
    double  q_betrag, tmp;
    int     zaehler ;

    for (p.x = 0, c.re = start.re; p.x < WIDTH;  p.x++, c.re += schritt)
    for (p.y = 0, c.im = start.im; p.y < HEIGHT; p.y++, c.im += schritt)
    {
        z.re = 0.0;
        z.im = 0.0;
        zaehler = 0;
        q_betrag  = 0.0;

        while( (q_betrag < 4.0)  && (zaehler++ < max_iter) ) {
                tmp    = z.re;
                z.re   = z.re * z.re - z.im * z.im + c.re;
                z.im   = 2 * tmp * z.im + c.im;
                q_betrag = z.re*z.re + z.im*z.im;
        }

        switch (fall) {

        case 1: if (q_betrag(z) < 4.0) set_pixel(p); break;       // schwarz

        case 2: if (q_betrag(z) < 4.0) set_pixel(p); else         // schwarz
                if ((zaehler %2) != 0) set_pixel(p); break;       // mit modulo

        case 3: if(zaehler == max_iter+1)
                set_col_pixel(p,farbe[0]); else                   // Mandelbrotmenge
                set_col_pixel(p,farbe[zaehler%NUM_COL]); break;   // farbe modulo
        }
    }
}
```

8.7 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.

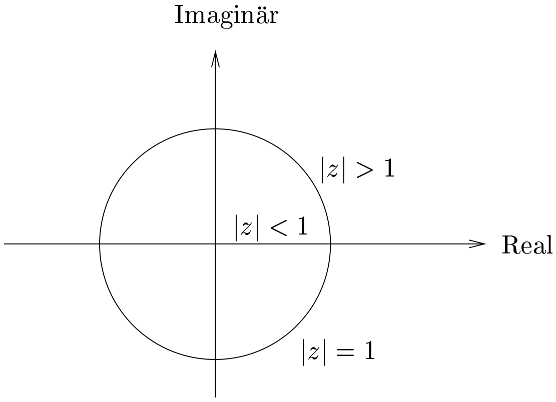


Abbildung 8.10: Julia-Kurve für $f(z) = z^2$

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

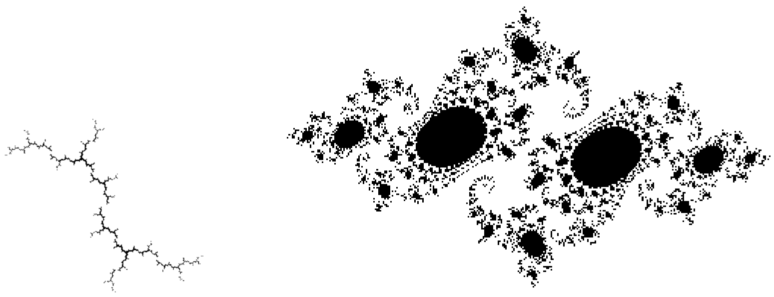


Abbildung 8.11: Vom Julia-Algorithmus erzeugte Bilder mit $c = i$ und $c = 0.746 + 0.112i$

Implementation der Julia-Menge:

1. Möglichkeit: Sei c gegeben. Jeder Wert z der komplexen Fläche wird überprüft, ob er als Startwert zur Konvergenz führt:

```
void  julia_voll(         // berechnet Julia-Menge als gefuelltes Gebiet
        Komplex c,        // fuer komplexes c,
        Komplex 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 ;
        Komplex 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 Komplex(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:

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

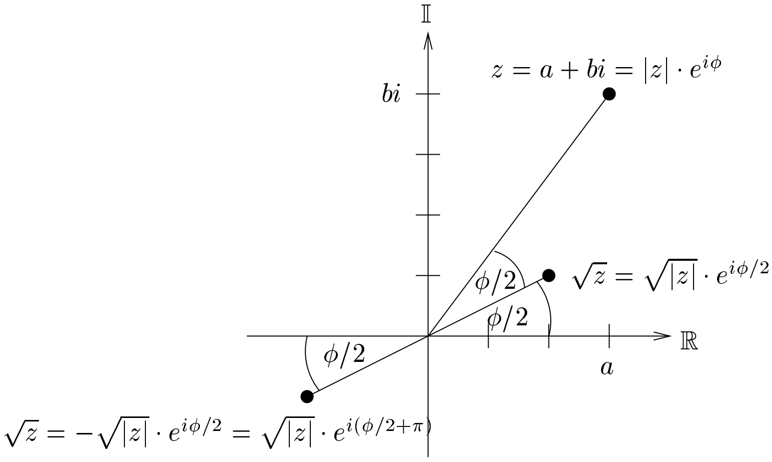


Abbildung 8.12: Zur Berechnung der Wurzel bei der Inverse Iteration Method

Realisiere f^{-1} durch

```
Komplex backward_random (Komplex z, Komplex 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
        Komplex c,         // fuer komplexes c
        Komplex start,     // linke obere Ecke der komplexen Flaeche
        double schritt,    // Schrittweite
        int anzahl)        // durch anzahl Rueckwaertsspruenge
{
        int k;
        Komplex z = new Komplex(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
        }

}
```

8.8 Beispiel-Applet zu Fraktalen

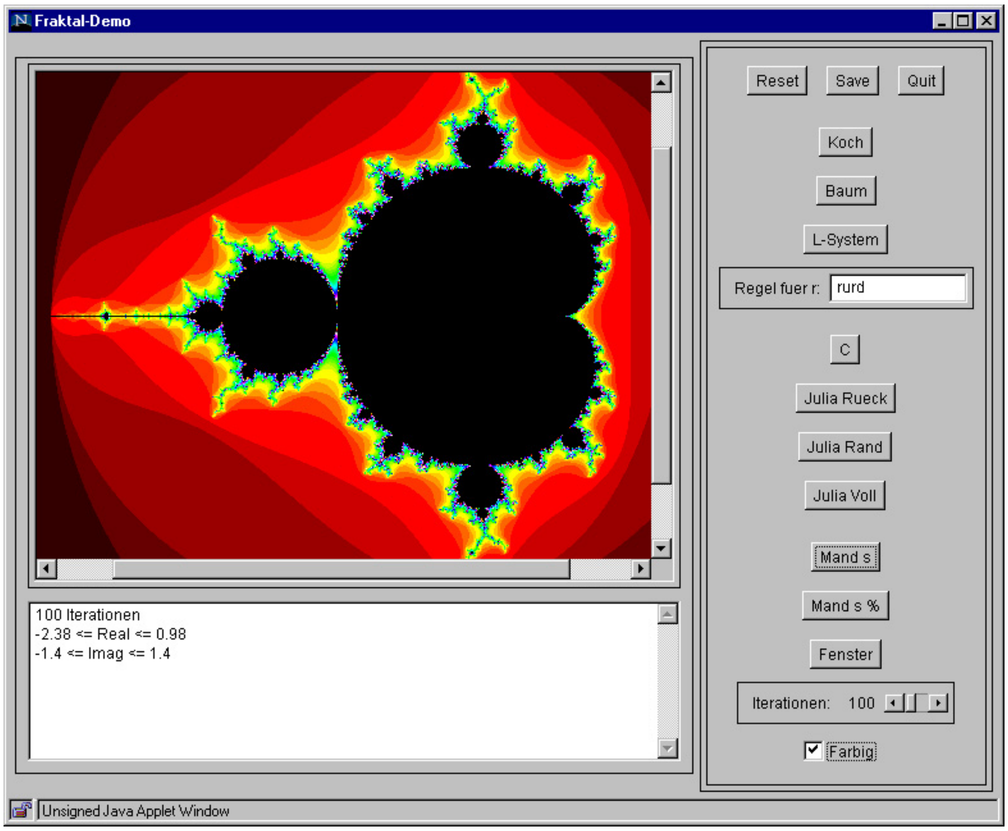


Abbildung 8.13: Screenshot vom Fraktale-Applet

8.9 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 p, wodurch ein Punkt $\overline{x}$ abgebildet wird auf $A\overline{x} + b$. Zum Beispiel erzeugen folgende 4 Regeln das Farnblatt in Abbildung 8.14 :

$$
\begin{array}{cccc}
 & A & b & p \\
w_1 & \begin{pmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{pmatrix} & \begin{pmatrix} 0.0 \\ 10.6 \end{pmatrix} & 85\ \% \\[2ex]
w_2 & \begin{pmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{pmatrix} & \begin{pmatrix} 0.0 \\ 10.6 \end{pmatrix} & 7\ \% \\[2ex]
w_3 & \begin{pmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{pmatrix} & \begin{pmatrix} 0.0 \\ 4.4 \end{pmatrix} & 7\ \% \\[2ex]
w_4 & \begin{pmatrix} 0.00 & 0.00 \\ 0.00 & 0.16 \end{pmatrix} & \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} & 1\ \%
\end{array}
$$

Abbildung 8.14: 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*.

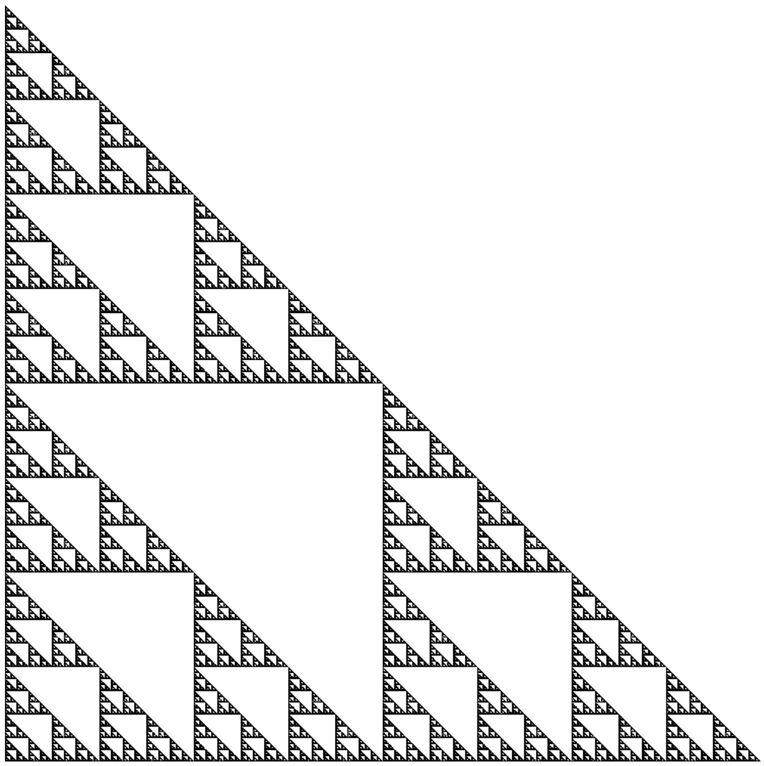


Abbildung 8.15: Sierpinsky-Dreieck

8.10 Beispiel-Applet zu Iterierten Funktionensystemen

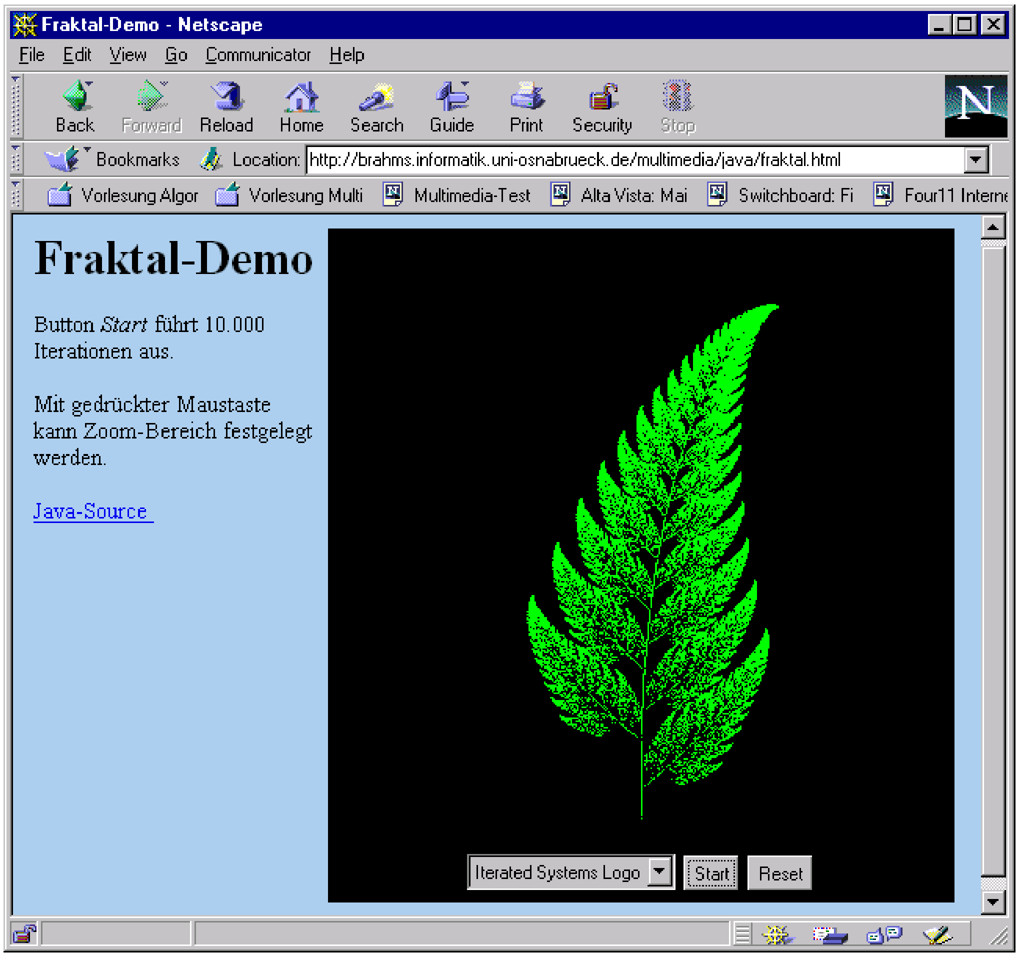


Abbildung 8.16: Screenshot vom Iterierte-Funktionen-Systeme-Applet

```
/*****************************  IFS.java * * ******************************************/
import java.util.Random;

public class IFS {    // Iterated Function System zur Erzeugung eines fraktalen Farns

  final static double [][][][] a = {                   // affine Abbildungen
     {{{ 0.85,  0.04}, {-0.04, 0.85}},                  // fuer Farn
      {{ 0.20, -0.26}, { 0.23, 0.22}},
      {{-0.15,  0.28}, { 0.26, 0.24}},
      {{ 0.00,  0.00}, { 0.00, 0.16}}},

     {{{0.5,  0.0}, {0.0,  0.5}},                      // fuer Sierpinski
      {{0.5,  0.0}, {0.0,  0.5}},
      {{0.5,  0.0}, {0.0,  0.5}}},
```

```
    {{{0.49, 0.01}, {0.0, 0.62}},                     // fuer Blatt
     {{0.27, 0.52}, {-0.40, 0.36}},
     {{0.18, -0.73}, {0.50, 0.26}},
     {{0.04, -0.01}, {0.50, 0.0}}}
  };

  final static double [][][] b = {                    // Translationsvektoren
        {{0.0, 10.6 },                                // fuer Farn
         {0.0, 10.6},
         {0.0,  4.4 },
         {0.0,  0.0 }},

        {{0.0, 0.0},                                  // fuer Sierpinski
         {50.0, 0.0},
         {0.0, 50.0}} ,

        {{25.0, 2.0},                                 // fuer Ahorn-Blatt
         { 0.0, 56.0},
         {88.0, 8.0},
         {52.0, 32.0}}
  };

  final static int [][] prob = {                      // Regelwahrscheinlichkeiten
        {85, 7, 7, 1},                                // fuer Farn
        {33, 33, 34},                                 // fuer Sierpinski
        {25, 25, 25, 25}                              // fuer Ahorn-Blatt
  };

  static Random r = new Random((long)42);

  public static int wuerfel(int sys) {                // wuerfelt eine Regel gemaess
    int k, sum;                                       // ihrer Wahrscheinlichkeit
    int w  = r.nextInt() % 100;
    w = w < 0 ? 1-w : 1+w;                            // 1 <= w <= 100
    sum = prob[sys][0]; k = 0;
    while (sum < w) sum +=prob[sys][++k];
    return k;
  }

  public static void transform ( Punkt p, int sys )   // transformiere Punkt p
  {
    int k;
    double x, y;

    k = wuerfel(sys);                                 // wuerfel eine Regel

    x = a[sys][k][0][0]*p.x +a[sys][k][0][1]*p.y + b[sys][k][0]; // bestimme
    y = a[sys][k][1][0]*p.x +a[sys][k][1][1]*p.y + b[sys][k][1]; // neuen Punkt
    p.x = x;
    p.y = y;
  }
}
```