# Kapitel 4

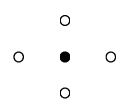
# 2-D Füllen

Zum Füllen eines Objekts mit einer einzigen Farbe oder mit einem Muster bieten sich zwei Ansätze an:

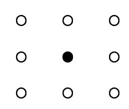
- universelle Verfahren, die die Zusammenhangseigenschaften der Pixel im Inneren der Objekte ausnützen.
- Scan-Line Verfahren, die eine geometrische Beschreibung der Begrenzungskurven voraussetzen.

## 4.1 Universelle Füll-Verfahren

Nachbarn zu • im 4-way-stepping:



Nachbarn zu • im 8-way-stepping:

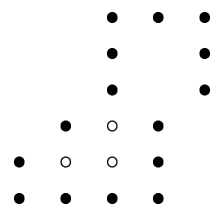


Ausgehend vom Startpunkt $(x, y)$ werden so lange die 4-way-stepping-Nachbarn gefärbt, bis die Umgrenzung erreicht ist.

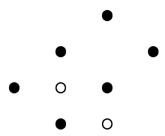
**Vorteil:** beliebige Umrandung möglich

**Nachteil:** hoher Rechen- und Speicherbedarf

**Obacht:** Gebiete, die nur durch 8-way-stepping erreicht werden können, werden beim Füllen mit 4-way-stepping "vergessen":



Wird hingegen die Nachbarschaft über 8-way-stepping definiert, so “läuft die Farbe aus”:



Benötigt werden

```
boolean get_pixel (            // liefert true, wenn Pixel p
                   Point p)    // die Vordergrundfarbe hat
```

und

```
boolean range_ok (             // liefert false, wenn Pixel p
                   Point p)    // ausserhalb des Bildbereichs
```

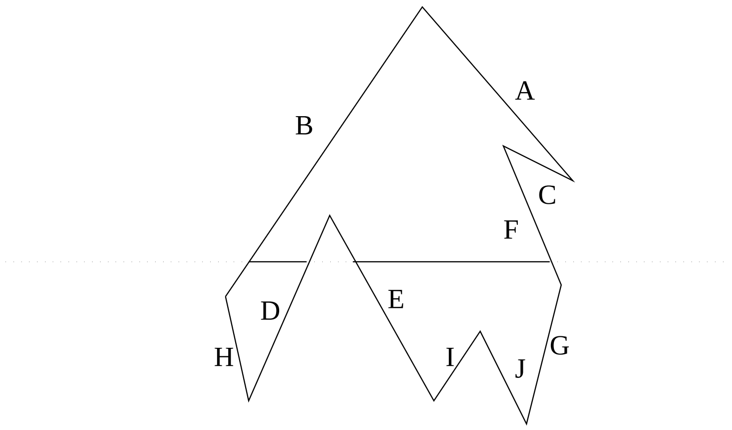
Der Nachteil der sehr ineffizienten Prozedur `boundary_fill` liegt darin, daß für jedes Pixel innerhalb der Begrenzungskurve(n) (mit Ausnahme der Randpixel des inneren Gebiets) der Algorithmus viermal aufgerufen wird. Dadurch werden die Pixel mehrfach auf dem Stapel abgelegt.

```
/****************************************************************************************/
/*                                                                                      */
/*                 Fuellen einer durch Vordergrundfarbe umrandeten Flaeche              */
/*                                                                                      */
/****************************************************************************************/

private void boundary_fill(            // setzt alle Nicht-Vordergrund-Pixel
                          Point p)     // beginnend bei Position p auf Vordergrund
{
        if ( range_ok(p) && !get_pixel(p) ) {     // falls p keine Vordergrundfarbe hat

                set_pixel(p);                     // setze Vordergrundfarbe

                boundary_fill( new Point(p.x+1, p.y   ));  // 4-way stepping
                boundary_fill( new Point(p.x,   p.y+1 ));
                boundary_fill( new Point(p.x-1, p.y   ));
                boundary_fill( new Point(p.x,   p.y-1 ));

        }
}



/****************************************************************************************/
/*                                                                                      */
/*                 Leeren einer durch Vordergrundfarbe definierten Flaeche              */
/*                                                                                      */
/****************************************************************************************/

private void boundary_empty(           // setzt alle Vordergrund-Pixel beginnend
                           Point p)    // bei Position p auf Hintergrundfarbe
{
        if ( range_ok(p) && get_pixel(p) ) {    // falls p Vordergrundfarbe hat

                del_pixel(p);                   // setze Hintergrundfarbe

                boundary_empty( new Point( p.x+1, p.y   ));  // 8-way-stepping
                boundary_empty( new Point( p.x+1, p.y+1 ));
                boundary_empty( new Point( p.x,   p.y+1 ));
                boundary_empty( new Point( p.x-1, p.y+1 ));
                boundary_empty( new Point( p.x-1, p.y   ));
                boundary_empty( new Point( p.x-1, p.y-1 ));
                boundary_empty( new Point( p.x,   p.y-1 ));
                boundary_empty( new Point( p.x+1, p.y-1 ));

        }
}
```

## 4.2 Scan-Line-Verfahren für Polygone

**Idee:** Bewege eine waagerechte Scan-Line schrittweise von oben nach unten über das Objekt, und berechne die Schnittpunkte der Scan-Line mit dem Objekt.

1. Sortiere alle Kanten nach ihrem größten $y$-Wert.
2. Bewege die Scan-Line vom größten $y$-Wert bis zum kleinsten $y$-Wert.
3. Für jede Position der Scan-Line
   - wird die Liste der aktiven Polygonkanten ermittelt,
   - werden die Schnittpunkte berechnet und nach $x$-Werten sortiert,
   - werden jene Scan-Line-Segmente, die im Inneren des Polygons liegen, angezeigt.

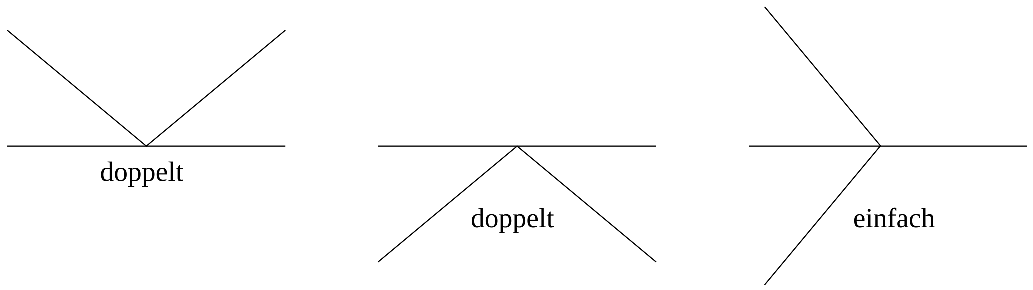



Sortierung der Kanten nach ihrem größten $y$-Wert ergibt: $ABCFDEGHIJ$
Für die momentane Scan-Linie gilt: aktive Kanten sind $BDEF$
Die sortierten $x$-Werte der Schnittpunkte $x_1, x_2, \ldots, x_n$ ergeben die zu zeichnenden Segmente $(x_1, x_2), (x_3, x_4), \ldots$
Die Sortierung der Kanten nach ihren größten $y$-Werten ermöglicht den einfachen Aufbau und die effiziente Aktualisierung einer Liste von aktiven Kanten. Eine Kante wird in diese Liste aufgenommen, wenn der Endpunkt mit dem größeren $y$-Wert von der Scan-Line überstrichen wird, und wird wieder entfernt, wenn die Scan-Line den anderen Endpunkt überstreicht.
Allgemein gilt, daß ein Punkt genau dann im Inneren des Polygons liegt, wenn ein von ihm ausgehender Halbstrahl die Polygonkanten ungeradzahlig oft schneidet.

### Sonderfälle

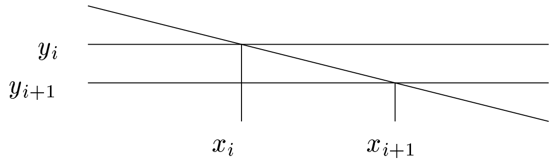
Horizontale Kanten werden nicht in die Kantenliste aufgenommen. Für sie wird eine Linie gezeichnet.
Trifft die Scan-Line auf einen Polygoneckpunkt, dessen Kanten beide oberhalb oder beide unterhalb liegen, so zählt der Schnittpunkt doppelt. Trifft die Scan-Line auf einen Polygoneckpunkt, dessen Kanten oberhalb und unterhalb liegen, so zählt der Schnittpunkt nur einfach.

Dadurch wird sichergestellt, daß die Paare $(x_1, x_2), (x_3, x_4), \ldots$ der sortierten $x$-Werte der Schnittpunkte die zu zeichnenden Segmente im Inneren korrekt darstellen.

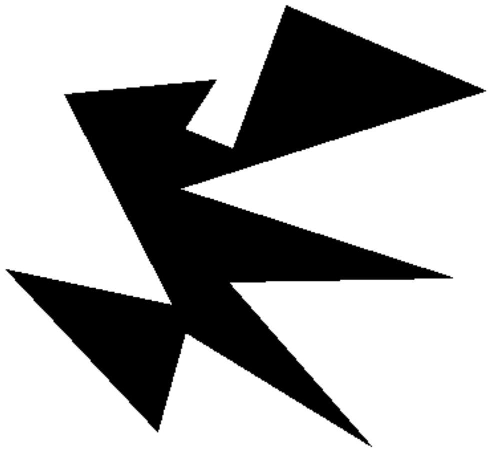



**Kohärenz-Eigenschaft**

Die Schnittpunkte für Scan-Line $y_{i+1}$ lassen sich mit Hilfe der Schnittpunkte von Scan-Line $y_i$ bestimmen:




Es gilt: Steigung der Geraden lautet $s = (y_i - y_{i+1})/(x_i - x_{i+1})$.
Wegen $y_i - y_{i+1} = 1$ ergibt sich $x_{i+1} = x_i - 1/s$.



Vom Scan-Line-Algorithmus gefülltes Polygon

In der folgenden Prozedur `scan_line_fill` wird als Datenstruktur für die Liste der Polygonkanten die Klasse `Edge` verwendet.

```
/** Klasse zur Implementation einer verzeigerten Kantenliste. */

public class Edge {

        /** groesste y-Koordinate der Kante. */
        public int      y_top;

        /** Schnittpunkt der Scan-Line mit der Kante. */
        public double   x_int;

        /** y-Ausdehnung der Kante. */
        public int      delta_y;

        /** inverse Steigung 1/s der Kante. */
        public double   delta_x;

        /** naechste Kante in der Kantenliste. */
        public Edge     next;

        /** Erzeugt ein Objekt vom Typ Kante mit den uebergebenen Parametern.
          * next ist die naechste Kante in der Liste.                       */

        public Edge(    int     y_top,
                        double  x_int,
                        int     delta_y,
                        double  delta_x,
                        Edge    next) {

                this.y_top    = y_top;
                this.x_int    = x_int;
                this.delta_y  = delta_y;
                this.delta_x  = delta_x;
                this.next     = next;
        }

        /** Erzeugt ein Objekt vom Typ Kante mit den uebergebenen Parametern. */

        public Edge(    int     y_top,
                        double  x_int,
                        int     delta_y,
                        double  delta_x) {

                this(y_top, x_int, delta_y, delta_x, null);
        }

        /** Erzeugt ein leeres Objekt vom Typ Kante. */

        public Edge() {

                this(0, 0.0, 0, 0.0, null);
        }
}
```

```
/*****************************************************************************************/
/*                                                                                       */
/*                     fuellt mit dem Scan-Line-Algorithmus das Innere eines Polygons   */
/*                                                                                       */
/*****************************************************************************************/

private        void Insert(                        /* fuegt in die Liste eine Kante ein */
        Edge edges,                                /* Beginn der Kantenliste            */
        Point P1, Point p2,                        /* einzufuegende Kante (P1,P2)       */
        int y_next)                                /* Behandlung von Sonderfaellen:     */
                                                   /* siehe Prozedur Next_y             */
{
        int max_y, dy;
        double x2, dx, max_x;

        Point P2 = new Point(p2);

        dx=(double)(P2.x-P1.x)/(double)(P2.y-P1.y);
        x2=(double) P2.x;

        if ((P2.y > P1.y) && (P2.y < y_next)) { P2.y--; x2=x2-dx; }     // Sonderfaelle
        if ((P2.y < P1.y) && (P2.y > y_next)) { P2.y++; x2=x2+dx; }

        dy=P2.y-P1.y;
        if (dy>0) { max_y = P2.y;
                    max_x = (double)x2;
                    dy++;
        } else    { max_y=P1.y;
                    max_x=(double)P1.x;
                    dy=1-dy;
        }

        Edge edge1 = edges;                                             // Hilfsobjekt

        while (edge1.next.y_top >= max_y) edge1 = edge1.next;

        Edge newedge = new Edge(max_y, max_x, dy, dx, edge1.next);      // einfuegen
                                                                        // sortiert nach
        edge1.next = newedge;                                           // max_y
}
```

```
private int Next_y(     /* liefert den y-Wert des naechsten Knoten laengs der Grenze   */
                        /* dessen y-Koordinate verschieden ist von P[k].y              */
        int k,                                         /* Index des Punktes            */
        Point[] Points,                                /* Liste von Punkten            */
        int n)                                         /* Anzahl der Punkte            */
{
        int compare_y, new_y;

        compare_y = Points[k].y;

        do {
                k = (k+1)%n;
                new_y = Points[k].y;
        } while (new_y == compare_y);

        return(new_y);
}

private int Edge_Sort(                           /* erzeugt nach y sortierte Kantenliste */
                                                 /* und liefert den kleinsten y-Wert     */
        int n,                                   /* Anzahl der Punkte                    */
        Point[] P,                               /* Punkteliste                          */
        Edge edges)                              /* Kantenliste                          */
{
        int bottom_y;
        Point P1;

        Edge edge1 = new Edge();

        edges.next=edge1;
        edge1.next=null;

        edges.y_top=Integer.MAX_VALUE;
        edge1.y_top=-Integer.MAX_VALUE;

        P1 = P[n-1];
        bottom_y = P1.y;

        for (int k=0; k<n; k++) {
                if (P1.y != P[k].y) Insert(edges,P1,P[k],Next_y(k,P,n));
                        else set_dither_line(P1,P[k].x);
                if (P[k].y < bottom_y) bottom_y = P[k].y;
                P1 = P[k];
        }
        return (bottom_y);
}
```

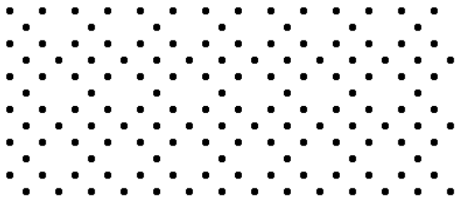
```
private Edge Update_List_Ptr(                /* aktualisiert den Zeiger auf die letzte */
                                             /* aktive Kante und gibt ihn zurueck      */
                                             /* wegen Dekrementieren der Scan-Line     */
                                             /* werden einige Kanten aktiv             */
        int scan,
        Edge l_act_edge)
{
        while (l_act_edge.next.y_top >= scan) l_act_edge=l_act_edge.next;

        return(l_act_edge);
}


private         Edge Sort_Intersections (      /* sortiert die aktive Kantenliste       */
        Edge edges,                            /* Beginn der Kantenliste                */
        Edge l_act_edge)                       /* Ende der Kantenliste                  */
                                               /* Liefert den ggf. modifizierten Zeiger */
                                               /* auf die letzte aktive Kante zurueck   */
{
        Edge edge1, edge2, edge3;

        edge2 = edges.next;

        do {
                edge1=edges;
                while (edge1.next.x_int < edge2.next.x_int)
                        edge1=edge1.next;
                if (edge1 != edge2) {               // tausche edge1.next und edge2.next
                        edge3          = edge2.next.next;
                        edge2.next.next = edge1.next;
                        edge1.next     = edge2.next;
                        edge2.next     = edge3;
                        if (edge1.next==l_act_edge) l_act_edge=edge2;
                } else edge2 = edge2.next;

        } while (edge2 != l_act_edge);

        return (l_act_edge);
}
```

```
private void Fill(                        /* generiert fuer je zwei Schnittpunkte   */
                                          /* aus der Kantenliste den Zeichne-Aufruf */
        Edge edges,                       /* Beginn der aktuellen Kantenliste       */
        Edge l_act_edge,                  /* Ende der aktuellen Kantenliste         */
        int scan)                         /* Scanline                               */
{
        Point Q = new Point();

        do {
                edges = edges.next;
                Q.x = (int) (edges.x_int+0.5);
                Q.y = scan;
                edges = edges.next;
                set_dither_line(Q,(int)(edges.x_int+0.5) );

        } while (edges != l_act_edge);
}

private        Edge Update_Edges(   /* aktual. die aktiven Kanten in der Kantenliste   */
        Edge edges,                 /* beginnend bei edges                             */
        Edge l_act_edge)            /* und endend bei l_act_edge                       */
                                    /* Kanten mit delta_y=0 werden entfernt. Der ggf.  */
                                    /* modifizierte Zeiger auf die letzte aktive Kante */
                                    /* wird zurueckgegeben                             */
{
        Edge prev_edge;

        prev_edge = edges;

        do {
                edges = prev_edge.next;
                if (edges.delta_y > 1) {
                        edges.delta_y--;
                        edges.x_int = edges.x_int - edges.delta_x;
                        prev_edge = edges;
                } else
                {
                        prev_edge.next = edges.next;
                        if (edges == l_act_edge) l_act_edge = prev_edge;
                        edges = null;        /* dispose edges */
                } /* if */
        } while (prev_edge != l_act_edge);
        return (l_act_edge);
}
```

```
private void scan_line_fill(                          /* Fuellt das Innere eines Polygons */
        int NumPoints,                                /* Anzahl der Punkte                */
        Point[] Points)                               /* Liste von Punkten                */
{
        Edge l_act_edge;
        int scan, bottom_y;

        Edge edges = new Edge();

        bottom_y = Edge_Sort(NumPoints, Points, edges);

        l_act_edge = edges.next;

        for (scan = edges.next.y_top; scan >= bottom_y; scan--) {
                l_act_edge = Update_List_Ptr(scan, l_act_edge);
                l_act_edge = Sort_Intersections(edges, l_act_edge);
                Fill (edges, l_act_edge, scan);
                l_act_edge = Update_Edges(edges, l_act_edge);
        }

        /* dispose dummies edges.next und edges */
        edges.next = null;
        edges = null;
}
```

## 4.3 Dithering

**Def.:** Eine $n \times n$-Dithermatrix $M$ ist mit den $n^2$ Zahlen zwischen 0 und $n^2 - 1$ besetzt. Zum Färben einer Fläche mit Grauwert $k, 0 \leq k \leq n^2$ werden alle Pixel $(i, j)$ gesetzt mit $M[i \bmod n, j \bmod n] < k$.



Zum Schwellwert 7 gehörender Grauwert

```
/************************************************************************************/
/*                                                                                  */
/*                    Definition einer Dithermatrix mit Routinen dafuer             */
/*                                                                                  */
/************************************************************************************/

private static final int DITHER_DIM  = 8;       // 8x8-Dithermatrix

private static final int[][] dit_mat =
{       { 0,32, 8,40, 2,34,10,42},              // 64-elementige Dithermatrix:
        {48,16,56,24,50,18,58,26},              // Bei Grauwert k werden alle
        {12,44, 4,36,14,46, 6,38},              // Pixel i,j auf schwarz
        {60,28,52,20,62,30,54,22},              // gesetzt, fuer die gilt:
        { 3,35,11,43, 1,33, 9,41},              // dit_mat[i%8,j%8] < k
        {51,19,59,27,49,17,57,25},
        {15,47, 7,39,13,45, 5,37},
        {63,31,55,23,61,29,53,21},
};

boolean kleiner_schwelle(                       // testet die Dither-Matrix
                          Point p )             // bezueglich des Punktes
{
        int x = p.x; int y = p.y;

        if (x<0) x=DITHER_DIM - (Math.abs(x)%DITHER_DIM);  // falls x negativ
        if (y<0) y=DITHER_DIM - (Math.abs(y)%DITHER_DIM);  // falls y negativ

        return (dit_mat[x%DITHER_DIM][y%DITHER_DIM] < schwelle);
}

int grauwert(                                   // liefert den Grauwert
              int i, int j )                    // an Position i, j

{       return(dit_mat[i][j]); }
```