Kapitel 4

2D-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 ausnutzen.
- Scan-Line-Verfahren, die eine geometrische Beschreibung der Begrenzungskurven voraussetzen.

4.1 Universelle Füll-Verfahren

Universelle Füllverfahren stützen sich auf die Nachbarschaft eines Pixels. Abbildung 4.1 zeigt zwei Varianten.

Nachbarn zu • im 4-way-stepping Nachbarn zu • im 8-way-stepping

Abbildung 4.1: Nachbarschaften bei universellen Füllverfahren

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”.

Bild 4.2 zeigt beide Effekte.

Unerreichtes Gebiet beim 4-way-stepping Farbauslauf beim 8-way-stepping

Abbildung 4.2: Probleme bei universellen Füllverfahren

Benötigt werden

```
public boolean getPixel(int x, int y)   // liefert true, wenn Pixel (x,y)
                                        // die Vordergrundfarbe hat
```

und

```
public boolean rangeOk(int x, int y)    // liefert true, wenn Pixel (x,y)
                                        // innerhalb des Bildbereichs liegt
```

```
/** fuellt eine durch Vordergrundfarbe umrandete Flaeche */
public void boundaryFill(int x, int y, CGCanvas cgc) {

  if (cgc.rangeOk(x,y) &&                       // falls Pixel im Bild
      !cgc.getPixel(x,y)) {                     // und bisher nicht gesetzt
    cgc.setPixel(x,y);                          // setze Vordergrundfarbe

    boundary_fill(x+1, y  , cgc);               // rekursive Aufrufe
    boundary_fill(x,   y+1, cgc);               // nach Schema des
    boundary_fill(x-1, y  , cgc);               // 4-Way-Stepping
    boundary_fill(x,   y-1, cgc);
  }
}
```

```
/** leert eine durch Vordergrundfarbe definierten Flaeche */

public void boundaryEmpty(int x, int y, CGCanvas cgc) { // Saatpixel (x,y)

  if(cgc.rangeOk(x, y) &&                          // falls Pixel im Bild und
     cgc.getPixel(x, y)) {                         // in Vordergrundfarbe

    cgc.del_pixel(x, y);                           // setze Hintergrundfarbe

    boundaryEmpty(x+1, y  , cgc);                  // loesche Nachbarpunkte
    boundaryEmpty(x+1, y+1, cgc);                  // nach Schema des
    boundaryEmpty(x,   y+1, cgc);                  // 8-Way-Stepping
    boundaryEmpty(x-1, y+1, cgc);
    boundaryEmpty(x-1, y  , cgc);
    boundaryEmpty(x-1, y-1, cgc);
    boundaryEmpty(x,   y-1, cgc);
    boundaryEmpty(x+1, y-1, cgc);
  }
}
```

Der Nachteil der sehr ineffizienten Methode `boundaryFill` 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.

Eine Beschleunigung des Verfahrens läßt sich dadurch erreichen, daß auf dem Stapel jeweils Repräsentanten für noch zu füllende Zeilen abgelegt werden, d.h. nach dem Einfärben einer kompletten horizontalen Linie werden von den unmittelbar angrenzenden Zeilen solche Punkte auf den Stapel gelegt, die noch nicht gefüllt worden sind und die unmittelbar links von einer Begrenzung liegen.

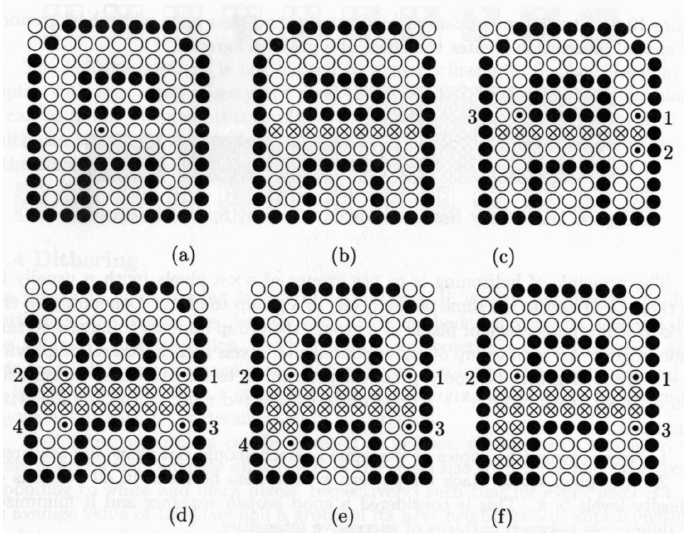


Abbildung 4.3: Beschleunigtes universelles Füllen (⊙: Saatpixel, ○: Pixel in Hintergundfarbe, ●: Randpixel, ⊗: vom Algorithmus gesetztes Pixel).

```
private void fillRowByRow(int x, int y,         // beginnend bei (x,y)
                          CGCanvas cgc) {       // fuelle eine durch die Vorder-
                                                // grundfarbe definierte Flaeche
                                                // linienweise

  int lg;                                       // linker Rand
  int rg;                                       // rechter Rand
  int px = x;                                   // Startpunkt

  while(!cgc.getPixel(x, y)) {                  // Solange Pixel ungesetzt
    cgc.setPixel(x, y);                         // Pixel setzen
    x--;                                        // naechstes Pixel links
  }
  lg = x+1;                                     // 1 zu weit gelaufen

  x = px + 1;                                   // da (px,y) schon gesetzt

  while(!cgc.getPixel(x, y)) {                  // Solange Pixel ungesetzt
    cgc.setPixel(x, y);                         // Pixel setzen
    x++;                                        // naechstes Pixel rechts
  }
  rg = x-1;                                     // 1 zu weit gelaufen

  for(x = rg; x >= lg; x--) {                   // von rechts nach links

    if(!cgc.getPixel(x, y-1)                    // falls Pixel drueber
                                                // nicht gesetzt:
      fillRowByRow(x, y-1, cgc);                // Repraesentant! Neuaufruf

    if(!cgc.getPixel(x, y+1))                   // falls Pixel drunter
                                                // nicht gesetzt:
      fillRowByRow(x, y+1, cgc);               // Repraesentant! Neuaufruf
  }
}
```

4.2 Scan-Line-Verfahren für Polygone

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

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.

Abbildung 4.4 zeigt eine Scanline beim Durchqueren eines Polygons. Die Sortierung der Kanten nach ihrem größten *y*-Wert ergibt die Folge $ABCDEFGHIJ$. Die zur Zeit aktiven Kanten sind $BEFD$. Die sortierten *x*-Werte der Schnittpunkte $x_1, x_2, \ldots, x_n$ ergeben die zu zeichnenden Segmente $(x_1, x_2), (x_3, x_4), \ldots$

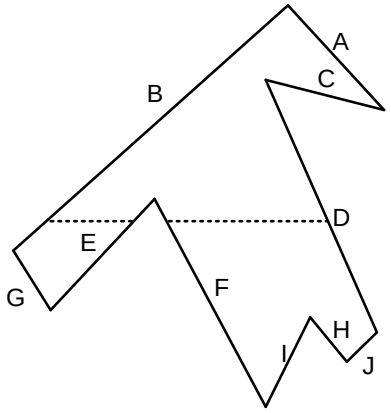


Abbildung 4.4: Polygon mit Scanline

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.

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 (siehe Abbildung 4.5).

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.

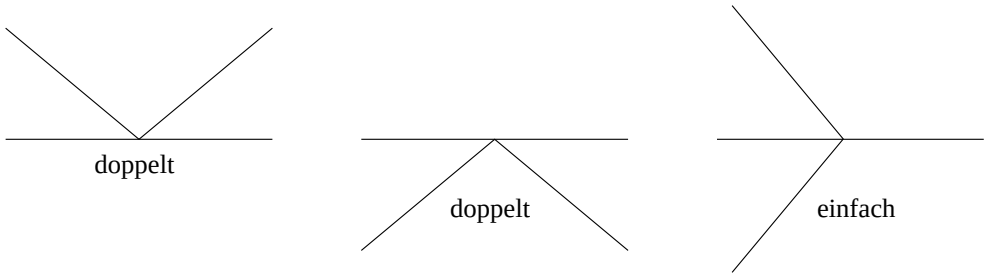


Abbildung 4.5: Fallunterscheidungen beim Berechnen der Schnittpunkte

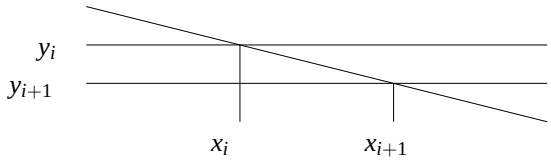


Abbildung 4.6: Fortschreiben der errechneten Schnittpunkte

Abbildung 4.6 zeigt, wie die Schnittpunkte für Scan-Line y_{i+1} sich mit Hilfe der Schnittpunkte von Scan-Line y_i bestimmen lassen.

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

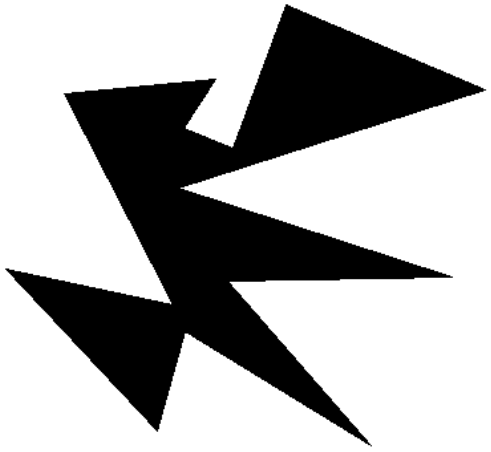


Abbildung 4.7: Vom Scanline-Algorithmus gefülltes Polygon

4.3 Dithering

Wenn einerseits die beiden ”Farben” Schwarz und Weiß zur Färbung der Objekte nicht ausreichen, andererseits der Bildschirm aber nur schwarze Pixel darstellen kann, wird wieder zu einem Trick gegriffen, bei dem das menschliche Auge betrogen wird.
Das Auge kann bei normalen Lichtverhältnissen Details unterscheiden, die mindestens eine Bogenminute ($1/60$ Grad) auseinander liegen. Wenn man in eine weiße Fläche eine Anzahl a von nicht benachbarten schwarzen Punkten hineinzeichnet und diese Fläche dann aus grösserer Entfernung betrachtet, dann ist das Auge nicht mehr in der Lage, die einzelnen Punkte voneinander zu unterscheiden. Die schwarzen und weißen Bereiche werden vom Auge integriert und dadurch entsteht der Eindruck einer grauen Fläche. Je größer a ist, desto dunkler wirkt die Fläche. Damit die Fläche gleichmäßig gefärbt erscheint, wird versucht die Punkte möglichst gleichmäßig zu verteilen. Eine Möglichkeit dies zu tun, ist das *Dithering*.

Sei n eine Zweierpotenz. Eine $n \times n$-Dithermatrix D 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 $D[i \mod n, j \mod n] < k$. Abbildung 4.8 zeigt das Füllmuster zum Schwellwert 7.

Die n^2 Zahlen können auf verschiedene Weisen auf die Matrix verteilt werden. Wir werden die Matrix nach dem Schema des *ordered dithering* füllen. Dabei wird die Matrix D_n rekursiv aus der Matrix D_{n-1} gebildet:

$$D_n = \begin{pmatrix} 4 \cdot D_{n-1} + 0 \cdot U_{n-1} & 4 \cdot D_{n-1} + 2 \cdot U_{n-1} \\ 4 \cdot D_{n-1} + 3 \cdot U_{n-1} & 4 \cdot D_{n-1} + 1 \cdot U_{n-1} \end{pmatrix}$$

Dabei bezeichnet U_n eine $n \times n$-Matrix, in der alle Elemente auf 1 gesetzt sind.

Es ergeben sich:

$$D_0 = (0); \quad D_1 = \begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix}; \quad D_2 = \begin{pmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{pmatrix}$$

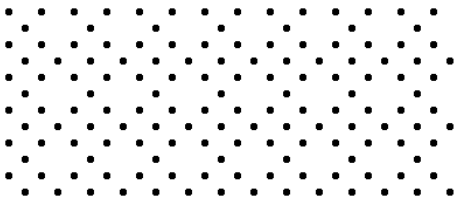


Abbildung 4.8: Zum Schwellwert 7 gehörender Grauwert

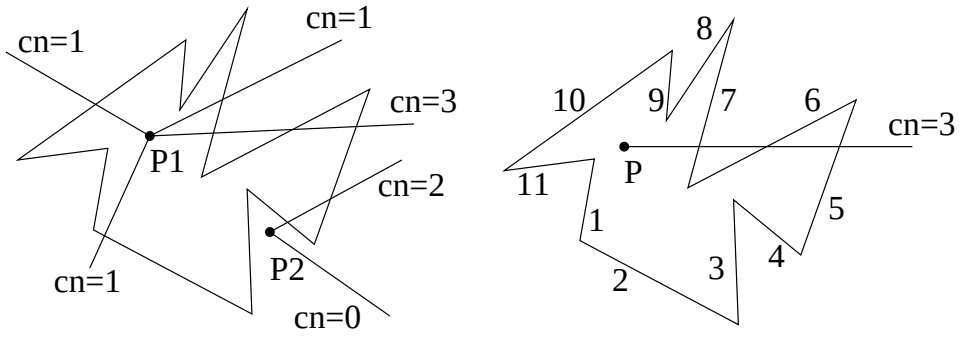
```
public class DitherMatrix extends NxNMatrix{

  /**
   * Gibt eine DitherMatrix zurueck, die mit Ditherwerten gefuellt ist.
   * @param n die Matrix ist (2 hoch n)x(2 hoch n) Elemente gross und
   * enthaelt die (2 hoch 2n) Ditherwerte von 0 bis (2 hoch 2n) - 1.
   */
  public DitherMatrix(int n) {
    super((int)Math.pow(2,n), 0.0);              // mit Nullen fuellen

    if(n != 0) {                                 // bei n==0 ist man fertig
      DitherMatrix dd = new DitherMatrix(n-1);   // 1/4 grosse DitherMatrix
                                                 // besorgen und die
      fill(0, 0, 0, dd);                         // Quadranten fuellen: ul
      fill(1, 1, 1, dd);                         // lr
      fill(1, 0, 2, dd);                         // ur
      fill(0, 1, 3, dd);                         // ll
    }
  }

  private void fill(int s, int z, int add, DitherMatrix dd) {
    int size = dd.size();                        // Ausmasse besorgen
    for(int i=0; i < size; i++) {                // passenden Quadranten
      for(int j=0; j < size; j++) {              // fuellen
        val[size*z+i][size*s+j]=4*dd.val[i][j] + add;
      }
    }
  }

  /**
   * Liefert zurueck, ob die Abbildung des uebergebenen Punktes in die
   * Matrix auf ein Matrixelement trifft, das kleiner als die uebergebene
   * Schwelle ist.
   * @param x, y  der zu testende Punkt
   * @param schwelle der Schwellwert
   */
  public boolean kleinerSchwelle(int x, int y, int schwelle) {
    if((x < 0) || (y < 0)) {                     // Wert(e) sinnvoll?
      return false;                              // Falls nicht: Abbruch
    }
    else {                                       // Sonst: Werte mit Modulo
      return (val[x%n][y%n] < schwelle);         // in Matrix abbilden und
    }                                            // Ergebnis zurueck
  }
}
```

4.4 Punkt in Polygon

Wenn der Benutzer mit der Maus auf dem Bildschirm in ein Polygon klickt, damit es anschließend gefüllt wird, so muß zunächst festgestellt werden, in welches Polygon geklickt wurde, damit einer der oben beschriebenen Algorithmen mit dem Füllen beginnen kann.

Nach dem *Jordanschen Kurvensatz* zerlegt jede einfache geschlossene Kurve die 2D-Ebene in genau zwei durch die Kurve verbundene Gebiete: Ein beschränktes Inneres und ein unbeschränktes Äußeres. Deshalb bedeutet eine Kreuzung mit der Kurve einen Wechsel von einem Gebiet in das andere.

Da ein einfaches Polygon eine Kurve im o.g. Sinne ist, gilt entsprechend, daß ein Punkt genau dann im Inneren des Polygons liegt, wenn ein von ihm in eine beliebige Richtung ausgehender Strahl ungeradzahlig viele Polygonkanten kreuzt, also die Kreuzungszahl (oder *Crossing Number*) ungerade ist (Abbildung 4.9 a)). Diese Idee ist auch als *Paritäts-* oder *Gerade-Ungerade-Test* bekannt.



a) Polygon mit zwei Testpunkten. P1 liegt im Polygon (cn ungerade); P2 nicht (cn gerade).

b) Zur effizienteren Berechnung wird der Teststrahl waagerecht nach rechts geschossen.

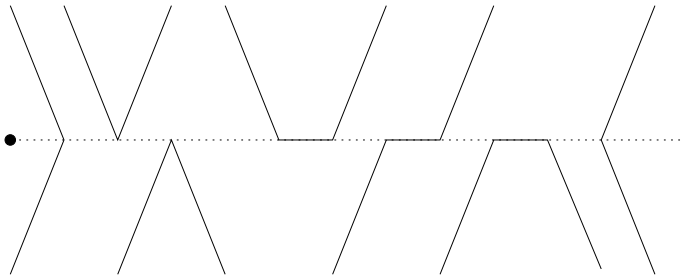
Abbildung 4.9: Ermittlung der Kreuzungszahl (cn) mit einem Teststrahl

Es muß für jede Polygonkante ermittelt werden, ob der Strahl sie schneidet. Wenn ja, wird die Kreuzungszahl um 1 erhöht, sonst nicht.

Dazu wird vom Testpunkt *P* der Strahl waagerecht nach rechts geschossen (Abbildung 4.9 b).

Um den Aufwand für die Schnittpunktberechnung auf diejenigen Kanten zu reduzieren, die den Strahl wirklich schneiden, werden zunächst einige Tests bezüglich der Koordinaten durchgeführt. Wenn Anfangs- und Endpunkt der aktuellen Polygonkante beide oberhalb oder beide unterhalb der Strahls liegen, kann die Kante den Strahl nicht schneiden (z.B. Kanten 1, 2, 3, 4, 8, 9, 11 in Abbildung 4.9 b)). Wenn ein Punkt oberhalb und der andere unterhalb liegt, kann ein Schnittpunkt vorliegen und die *x*-Koordinaten werden untersucht. Sind beide Punkte rechts von *P*, schneidet der Strahl die Kante irgendwo (diese Information ist ausreichend) und die Kreuzungszahl wird erhöht (Kanten 7, 6, 5). Sind beide links von *P* gibt es keinen Schnittpunkt. Ist einer links und einer rechts von *P* muß berechnet werden, wo die Kante den *y*-Wert von *P* erreicht; d.h. ob der Schnittpunkt rechts von *P* liegt (z.B. Kante 7). Wenn ja, wird die Kreuzungszahl erhöht.

Ein Problem stellen die bereits beim ScanLine-Verfahren erwähnten Sonderfälle dar, bei denen der Strahl genau einen Polygonpunkt trifft.



Ein Sonderfall liegt immer dann vor, wenn der Strahl einen oder mehrere Polygonpunkte trifft.

Das Problem wird gelöst, indem so getan wird als läge der Polygonpunkt infinitesimal üeber dem Strahl. Bei der Implementation wird dies dadurch erreicht, daß Polygonpunkte mit einem y-Wert $\geq p_y$ als über dem Strahl liegend interpretiert werden. Auf diese Weise wird gleichzeitig die Frage geklärt, zu welchem Polygon ein Pixel gehört, wenn zwei Polygone eine (oder mehrere) identische Kanten haben.

Damit ergibt sich folgender Algorithmus:

```
public boolean contains(int x, int y) {
  boolean inside = false;

  int x1 = xpoints[npoints-1];
  int y1 = ypoints[npoints-1];
  int x2 = xpoints[0];
  int y2 = ypoints[0];

  boolean startUeber = y1 >= y? true : false;
  for(int i = 1; i<npoints ; i++) {
    boolean endUeber = y2 >= y? true : false;
    if(startUeber != endUeber) {
      if((double)(y*(x2 - x1) -y1*x2 +y2*x1)/(double)(y2-y1) >= x) {
        inside = !inside;
      }
    }
    startUeber = endUeber;
    y1 = y2;
    x1 = x2;
    x2 = xpoints[i];
    y2 = ypoints[i];
  }

  return inside;
}
```

Die Division zur Berechnung des Schnittpunktes kann vermieden werden, wenn man die Steigung der Polygonkante mit der Steigung der Geraden durch P und den Endpunkt der Kante vergleicht und dabei berücksichtigt, ob der Endpunkt eine größere oder eine kleinere y-Koordinate hat.

```
public boolean contains(int x, int y) {
  boolean inside = false;

  int x1 = xpoints[npoints-1];
  int y1 = ypoints[npoints-1];
  int x2 = xpoints[0];
  int y2 = ypoints[0];

  boolean startUeber = y1 >= y? true : false;
  for(int i = 1; i<npoints ; i++) {
    boolean endUeber = y2 >= y? true : false;
    if(startUeber != endUeber) {
      if((y2 - y)*(x2 - x1) <= (y2 - y1)*(x2 - x)) {
        if(endUeber) {
          inside = !inside;
        }
      }
      else {
        if(!endUeber) {
          inside = !inside;
        }
      }
    }

    startUeber = endUeber;
    y1 = y2;
    x1 = x2;
    x2 = xpoints[i];
    y2 = ypoints[i];
  }
  return inside;
}
```