Kapitel 3

2D-Grundlagen

In diesem Kapitel geht es um die Frage, wie man zweidimensionale mathematische Elementarobjekte wie Punkte, Linien und Kreise im Computer repräsentieren und auf dem Bildschirm darstellen kann.

Da die zu beschreibenden Objekte dem Bereich der täglichen Erfahrung entstammen, liegt es nahe, die folgenden Betrachtungen alle im Vektorraum $\mathbb{R}^2$ mit euklidischer Norm durchzuführen.

Die Elemente des $\mathbb{R}^2$ werden *Vektoren* genannt. Sie geben eine Richtung und eine Länge an. Vektoren werden wir mit einem Kleinbuchstaben und einem Pfeil darüber kennzeichnen: $\vec{x}$.

Zum Rechnen mit Vektoren ist es notwendig, die *Koordinaten* des Vektors bzgl. eines bestimmten Koordinatensystems zu betrachten.

3.1 Koordinatensysteme

Das geläufigste Koordinatensystem für die oben genannte euklidische Ebene ($\mathbb{R}$) ist das *kartesische Koordinatensystem*. Seine beiden Koordinatenachsen stehen senkrecht aufeinander und schneiden sich im (willkürlich festgelegten) Ursprung O. Die beiden Einheitsvektoren $\vec{e}_x$ und $\vec{e}_y$ sind parallel zu den Achsen und führen, wenn sie von O abgetragen werden, zu den Punkten mit Abstand 1 von O. Als Spaltenvektoren geschrieben, haben sie folgendes Aussehen:

$$\vec{e}_x = \begin{pmatrix} 1 \\ 0 \end{pmatrix};\ \vec{e}_y = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

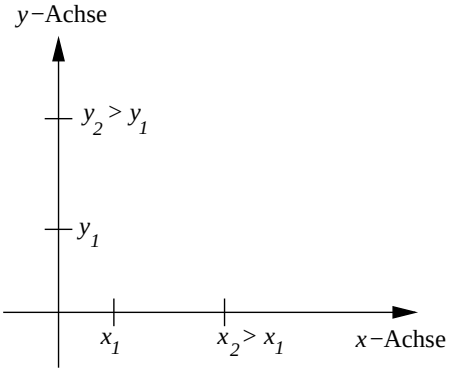
Aus Platzgründen wird stellenweise die Schreibweise als Zeilenvektor verwendet:

$$\vec{e}_x = (1\ \ 0)^T;\ \vec{e}_y = (0\ \ 1)^T$$

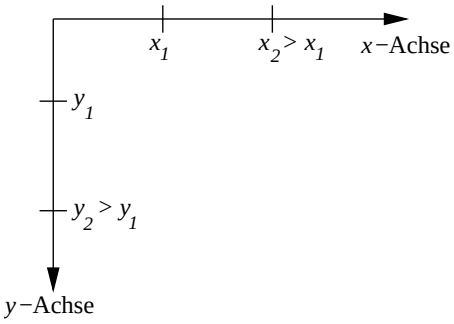
Die erste Koordinatenachse (x-Achse) wird immer von links nach rechts gezeichnet; d.h. die größeren Koordinatenwerte befinden sich weiter rechts.

An der Tafel bzw. in der Vorlesung wird für gewöhnlich die 2. Koordinatenachse (y-Achse) so gezeichnet, daß die größeren Werte weiter oben sind.

Auf dem Bildschirm hingegen befindet sich der Ursprung O oben links; d.h. die y-Achse liegt so, daß sich die größeren Koordinatenwerte weiter unten befinden. Insbesondere hat kein Bildschirmpunkt negative Koordinaten.



Kartesisches Koordinatensystem an der Tafel



Kartesisches Koordinatensystem am Bildschirm

Andere Koordinatensysteme und der Wechsel zwischen diesen werden später ausführlich behandelt.

3.2 Punkt

Mit Hilfe dieses Koordinatensystems läßt sich jeder Punkt P der euklidischen Ebene durch Angabe einer x- und einer y-Koordinate beschreiben:

$$P = (p_x, p_y)$$

Deutlich vom Punkt P zu unterscheiden ist der Vektor $\vec{p} = (p_x \;\; p_y)^T$, welcher von O abgetragen zu P führt. $\vec{p}$ kann als Linearkombination von $\vec{e}_x$ und $\vec{e}_y$ aufgefaßt werden:

$$\vec{p} = p_x \cdot \vec{e}_x + p_y \cdot \vec{e}_y$$

Die Koordinaten p_x und p_y sind Elemente von $\mathbb{R}$. Die Bildschirmpunkte hingegen sind ganzzahlig. Wir können einen bestimmten Bildschirmpunkt ”anschalten”, indem wir (bei dem Objekt, das den grafischen Kontext darstellt) die Methode

```
setPixel(int x, int y);
```

mit den entsprechenden ganzzahligen Koordinaten aufrufen.

Sei $P = (2.0, 2.0)$ gegeben. In diesem Fall tut

```
setPixel(2, 2);
```

genau das, was wir erwarten.

Wenn aber $P = (2.3, 3.7)$ gegeben ist, dann müssen die Koordinaten auf diejenigen ganzen Zahlen gerundet werden, die die gewünschten Koordinaten am besten repräsentieren:

```
x = 2.3; y = 3.7;
setPixel((int)(x+0.5), (int)(y+0.5));
```

3.3 Linie

- Gegeben sind Anfangspunkt $P_1 = (x_1, y_1)$ und Endpunkt $P_2 = (x_2, y_2)$ einer Linie l:
- Zu berechnen sind wieder die "anzuschaltenden" Pixel.

3.3.1 Parametrisierte Geradengleichung

Da P_1 und P_2 zwei Punkte auf einer Geraden sind, bietet es sich an, die ganze Linie als Teilstück einer Geraden aufzufassen und durch die Parametrisierung der Geraden mit einem Parameter r die Pixel zu bestimmen.

Gesucht ist der Vektor $\vec{v}$, der von P_1 nach P_2 führt.

Es gilt

$$\begin{aligned} & \vec{p_1} + \vec{v} &&= \vec{p_2} \\ \Leftrightarrow \quad & \vec{v} &&= \vec{p_2} - \vec{p_1} \\ \Leftrightarrow \quad & \begin{pmatrix} v_x \\ v_y \end{pmatrix} &&= \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} \end{aligned}$$

Die gesamte Gerade g ergibt sich, wenn man von O zu einem Punkt der Geraden geht (z.B. P_1) und von dort ein beliebiges Vielfaches des Richtungsvektors $\vec{v}$ abträgt (Punkt-Richtungsform):

$$g : \vec{u} = \vec{p_1} + r \cdot \vec{v}; r \in \mathbb{R}$$

Die gesuchte Linie l erhält man, wenn man r auf das Intervall $[0;1]$ beschränkt:

$$l : \vec{u} = \vec{p_1} + r \cdot \vec{v}; r \in [0;1]$$

Jetzt muß man nur noch entscheiden, in wievielen Schritten r das Intervall $[0;1]$ durchlaufen soll; d.h. wieviele Pixel man "anschalten" will, um die ganze Linie zu repräsentieren.

Eine von P_1 und P_2 unabhänigige Anzahl (z.B. 100 Schritte) würde bei kurzen Linien dazu führen, daß manche Pixel aufgrund der Rundung mehrfach gesetzt würden. Bei langen Linien hingegen wären die Pixel evtl. nicht benachbart.

Sinnvoll wäre es, soviele Pixel zu setzen, wie die Linie Einheiten lang ist. In der euklidischen Ebene ist die Länge d einer Strecke $\overline{P_1P_2}$ als Abstand von Anfangs- und Endpunkt definiert. Die Abstandsberechnung geschieht mit Hilfe der euklidischen Norm (vgl. Pythagoras):

$$d = \|\overline{P_1P_2}\| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Hier die `draw`-Methode der Klasse `line.VectorLine`. x_1, x_2, y_1 und y_2 sind `int`-Variablen, die bei der Instanziierung der Linie übergeben wurden. D.h. die Linie weiß, wo sie beginnt und wo sie aufhört und kann sich selber zeichnen.

```
public void draw(CGCanvas cgc) {
  int x, y, dx, dy;
  double r, step;

  dy = y2 - y1;                                  // Hoehenzuwachs
  dx = x2 - x1;                                  // Schrittweite

  step = 1.0 / Math.sqrt(dx*dx + dy*dy);         // Parameterschritt berechnen

  for(r=0.0; r < 1; r+=step) {                   // fuer jeden Parameterwert
    x = (int)(x1 + r*dx +0.5);                   // berechne neue x-Koordinate
    y = (int)(y1 + r*dy +0.5);                   // berechne neue y-Koordinate
    cgc.setPixel(x,y);                           // setze Pixel
  }
  cgc.setPixel(x2, y2);                          // letztes Pixel am Endpunkt
}
```

Diese Implementation hat den Nachteil, daß sehr viel Gleitkommaarithmetik durchgeführt werden muß. Gleitkommaarithmetik ist (in Java) im Gegensatz zu Integerarithmetik sehr zeitintensiv. Wir werden im Folgenden diesen Nachteil schrittweise beseitigen.

3.3.2 Geradengleichung als Funktion

Eigentlich ist es unnötig für jedes Pixel die neue x-Koordinate **und** die neue y-Koordinate auszurechnen, da man ja der Reihe nach alle Pixel der Linie setzen will. Wenn man also für jedes Pixel z.B. die x-Koordinate um 1 erhöht, braucht man nur noch die zugehörige y-Koordinate zu berechnen. Dazu muß die Geradengleichung aber in Form einer Funktion vorliegen:

$$y(x) = s \cdot x + c$$

Die Steigung s errechnet sich mit dem Steigungsdreieck wie folgt:

$$s = \frac{\text{Höhenzuwachs}}{\text{Schrittweite}} = \frac{y_2 - y_1}{x_2 - x_1}$$

das gilt sowohl für das Dreieck mit P_1 und P_2 als auch für das Dreieck mit P_1 und dem Punkt C, an dem die Gerade die y-Achse schneidet:

$$\begin{aligned} \frac{y_1 - c}{x_1 - 0} &= \frac{y_2 - y_1}{x_2 - x_1} \\ \Leftrightarrow c &= \frac{y_1 \cdot x_2 - y_2 \cdot x_1}{x_2 - x_1} \end{aligned}$$

einsetzen in die Geradengleichung ergibt:

$$y = \frac{y_2 - y_1}{x_2 - x_1} \cdot x + \frac{y_1 \cdot x_2 - y_2 \cdot x_1}{x_2 - x_1}$$

Hier die `draw`-Methode aus der Klasse `line.StraightLine`:

```
public void draw(CGCanvas cgc) {
  int x, y;
  double s, c;

  s = (double)(y2 - y1) / (double)(x2 - x1);   // Steigung berechnen
  c = (double)(y1*x2 - y2*x1) /                // y-Achsenabschnitt
      (double)(x2 - x1);

  x = x1;                                      // Koordinaten retten
  y = y1;

  if(x < x2)  {                                // Linie links -> rechts
    while(x <= x2) {                           // fuer jede x-Koordinate
      y = (int)(s*x + c + 0.5);                // berechne y-Koordinate
      cgc.setPixel(x,y);                       // setze Pixel
      x++;                                     // naechste x-Koordinate
    }
  }
  else {                                       // Linie rechts -> links
    while(x >= x2) {                           // fuer jede x-Koordinate
      y = (int)(s*x + c + 0.5);                // berechne y-Koordinate
      cgc.setPixel(x,y);                       // setze Pixel
      x--;                                     // naechste x-Koordinate
    }
  }
}
```

Diese Version kommt mit ungefähr halb soviel Gleitkommaarithmetik aus, wie die letzte. Allerdings brauchen wir eine zusätzliche Fallunterscheidung, um zu klären, ob die Linie von links nach rechts verläuft oder andersherum. Ein weiterer schwerer Nachteil liegt in der Tatsache, daß die Pixel für steile Geraden nicht benachbart sind.

3.3.3 Bresenham-Algorithmus

Wir werden zunächst den Fall diskutieren, bei dem die Steigung in $[0;1]$ liegt und später verallgemeinern.

Wie oben erwähnt, weicht die gezeichnete Linie für gewöhnlich von der idealen Linie ab:

Die Größe dieser Abweichung läßt sich ausnutzen, um zu entscheiden, ob man für die nächste *x*-Koordinate die aktuelle *y*-Koordinate beibehalten kann oder ob man die *y*-Koordinate um 1 erhöhen muß. Diese Art der Implementation ist aus dem Jahr 1965 und stammt von Jack Bresenham. Der Algorithmus berechnet den jeweils nächsten *y*-Wert aus dem vorherigen und hält dabei den momentanen Fehler nach.

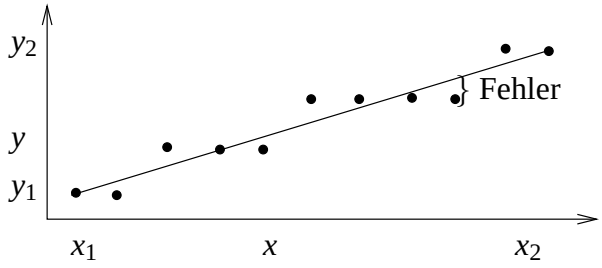


Abbildung 3.1: Fehler als Abweichung von der Ideal-Linie

```
public void drawBresenhamLine1(CGCanvas cgc) {
  int x, y, dx, dy;
  double s, error;

  dy    = y2 - y1;                                  // Hoehenzuwachs berechnen
  dx    = x2 - x1;                                  // Schrittweite

  x = x1;                                           // Koordinaten retten
  y = y1;

  error = 0.0;                                      // momentaner Fehler
  s     = (double) dy / (double) dx;                // Steigung

  while (x <= x2) {                                 // fuer jede x-Koordinate
    cgc.setPixel(x, y);                             // setze Pixel
    x++;                                            // naechste x-Koordinate
    error += s;                                     // Fehler aktualisieren
    if (error > 0.5) {                              // naechste Zeile erreicht?
      y++;                                          // neue y-Koordinate
      error-- ;                                     // Fehler anpassen
    }
  }
}
```

Man kann die verbleibende Gleitkommaarithmetik vermeiden, indem man zur Fehlerbestimmung und zur Entscheidung, ob die y-Koordinate angepasst werden muß, eine zweite (wesentlich steilere) Gerade verwendet. Die Steigung dieser neuen Geraden berechnet sich folgendermaßen:

$$s_{neu} = s_{alt} \cdot 2dx = \frac{dy}{dx} \cdot 2dx = 2dy$$

Für ein ganzzahliges s würde bereits die Multiplikation mit dx genügen. Da wir aber auch den Fahler ganzzahlig machen wollen, müssen wir zusätzlich mit 2 multiplizieren:

```
public void drawBresenhamLine2(CGCanvas cgc) {
  int x, y, dx, dy, error, delta;

  dy    = y2 - y1;                                  // Hoehenzuwachs berechnen
  dx    = x2 - x1;                                  // Schrittweite

  x = x1;                                           // Koordinaten retten
  y = y1;

  error = 0;                                        // momentaner Fehler
  delta = 2*dy;                                     // 'Steigung'

  while (x <= x2) {                                 // fuer jede x-Koordinate
    cgc.setPixel(x, y);                             // setze Pixel
    x++;                                            // naechste x-Koordinate
    error += delta;                                 // Fehler aktualisieren
    if (error > dx) {                               // naechste Zeile erreicht?
      y++;                                          // neue y-Koordinate
      error -= 2*dx;                                // Fehler anpassen
    }
  }
}
```

Um nochmals etwas Zeit zu sparen, vergleichen wir `error` mit 0 und verwenden die Abkürzung `schritt` für `-2*dx`:

```
public void drawBresenhamLine3(CGCanvas cgc) {
  int x, y, dx, dy, error, delta, schritt;

  dy    = y2 - y1;                                  // Hoehenzuwachs berechnen
  dx    = x2 - x1;                                  // Schrittweite

  x = x1;                                           // Koordinaten retten
  y = y1;

  error = -dx;                                      // momentaner Fehler
  delta = 2*dy;                                     // 'Steigung'
  schritt = -2*dx;                                  // Fehlerschrittweite

  while (x <= x2) {                                 // fuer jede x-Koordinate
    cgc.setPixel(x, y);                             // setze Pixel
    x++;                                            // naechste x-Koordinate
    error += delta;                                 // Fehler aktualisieren
    if (error > 0) {                                // naechste Zeile erreicht?
      y++;                                          // neue y-Koordinate
      error += schritt;                             // Fehler anpassen
    }
  }
}
```

Geraden in den anderen 7 Oktanten (Steigung $\notin [0;1]$) müssen durch Spiegelung und/oder Vertauschen von *x* und *y* auf den 1. Oktanten zurückgeführt werden:

```
public void draw(CGCanvas cgc) {
  int x, y, error, delta, schritt, dx, dy, inc_x, inc_y;

  x = x1;                                    // Koordinaten retten
  y = y1;

  dy = y2 - y1;                              // Hoehenzuwachs
  dx = x2 - x1;                              // Schrittweite

  if(dx > 0)                                 // Linie nach rechts?
    inc_x = 1;                               // x inkrementieren
  else                                       // Linie nach links
    inc_x = -1;                              // x dekrementieren

  if(dy > 0)                                 // Linie nach unten?
    inc_y = 1;                               // y inkrementieren
  else                                       // Linie nach oben
    inc_y = -1;                              // y dekrementieren

  if(Math.abs(dy) < Math.abs(dx)) {          // flach nach oben oder unten
    error = -Math.abs(dx);                   // Fehler bestimmen
    delta = 2*Math.abs(dy);                  // Delta bestimmen
    schritt = 2*error;                       // Schwelle bestimmen
    while(x != x2) {                         // Fuer jede x-Koordinate
      cgc.setPixel(x,y);                     // setze Pixel
      x += inc_x;                            // naechste x-Koordinate
      error = error + delta;                 // Fehler aktualisieren
      if (error > 0) {                       // neue Spalte erreicht?
        y += inc_y;                          // y-Koord. aktualisieren
        error += schritt;                    // Fehler aktualisieren
      }
    }
  }
  else {                                     // steil nach oben oder unten
    error = -Math.abs(dy);                   // Fehler bestimmen
    delta = 2*Math.abs(dx);                  // Delta bestimmen
    schritt = 2*error;                       // Schwelle bestimmen
    while(y != y2) {                         // fuer jede y-Koordinate
      cgc.setPixel(x,y);                     // setze Pixel
      y += inc_y;                            // naechste y-Koordinate
      error = error + delta;                 // Fehler aktualisieren
      if (error > 0) {                       // neue Zeile erreicht?
        x += inc_x;                          // x-Koord. aktualisieren
        error += schritt;                    // Fehler aktualisieren
      }
    }
  }
  cgc.setPixel(x2, y2);                      // letztes Pixel hier setzen,
}                                            // falls (x1==x2) & (y1==y2)
```

Bresenham-Algorithmus für Linien

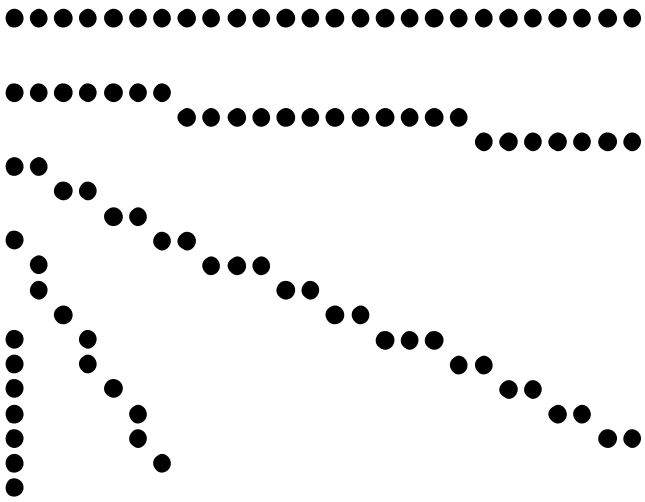


Abbildung 3.2: Vom Bresenham-Algorithmus erzeugte Linien

3.3.4 Antialiasing

Eine diagonale Linie verwendet dieselbe Anzahl von Pixeln wie eine horizontale Linie für eine bis zu $\sqrt{2}$mal so lange Strecke. Daher erscheinen horizontale und vertikale Linien kräftiger als diagonale. Dies kann durch Antialiasing-Techniken behoben werden.

Bei gleichbleibender Auflösung (d.h. Pixel pro Zeile/Spalte) kann bei Schirmen mit mehreren Grauwerten pro Pixel die Qualität der Linie gesteigert werden. Hierzu werden die Pixel proportional zur Überlappung mit der Ideallinie geschwärzt.

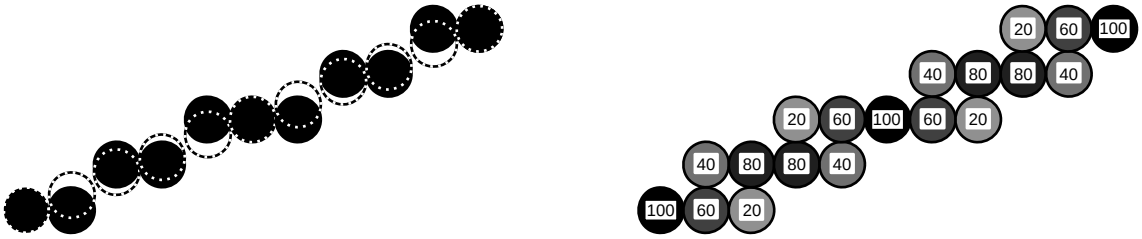


Abbildung 3.3: (a) Bresenham-Linie und Ideal-Linie (b) Resultierende Grauwerte

In Abbildung 3.3 liegt das zweite vom Bresenham-Algorithmus gesetzte Pixel unterhalb der Ideallinie und hat eine Überlappung mit der Ideallinie von 60 %. Daher wird das Bresenham-Pixel mit einem 60 % - Grauwert eingefärbt und das Pixel darüber mit einem 40 % - Grauwert. Das Auge integriert die verschiedenen Helligkeitswerte zu einer saubereren Linie als die reine Treppenform von schwarzen Pixeln.

3.4 Polygon

Ein Polygon wird spezifiziert durch eine Folge von Punkten, die jeweils durch Linien verbunden sind. Anfangs- und Endpunkt sind identisch.

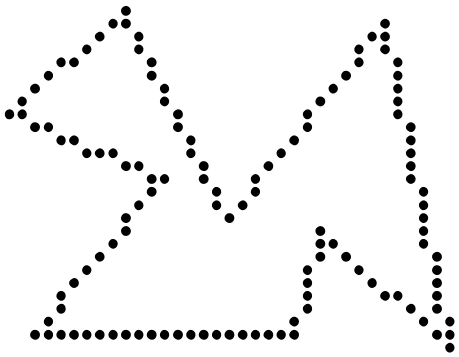
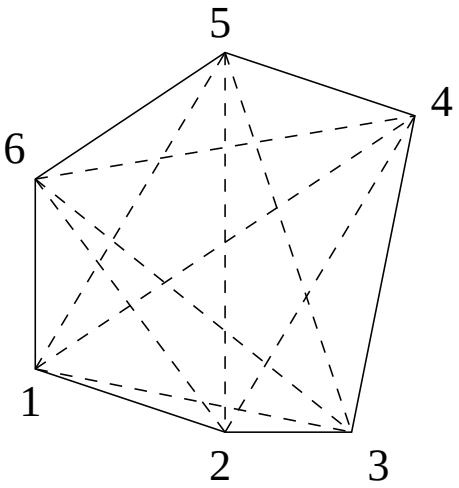


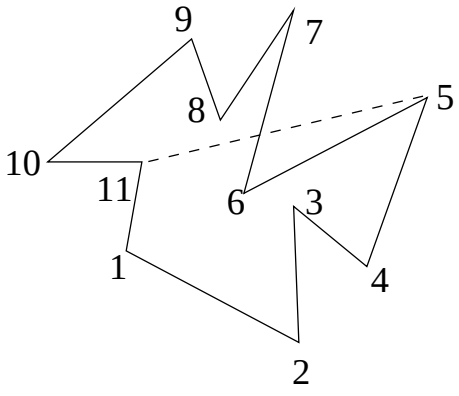
Abbildung 3.4: Polygon

3.4.1 Konvexität

Wir unterscheiden konvexe und konkave Polygone. Bei einem konvexen Polygon ist jeder Eckpunkt von jedem Eckpunkt aus ”sichtbar”. D.h. daß die Verbindungslinie zwischen zwei beliebigen Eckpunkten innerhalb des Polygons verläuft (bei benachbarten Eckpunkten ist diese Verbindungslinie natürlich identisch mit eine Polygonkante). Beim konkaven Polygon schneidet mindestens eine Verbindungslinie mindestens eine Polygonkante:



Konvexes Polygon; alle Verbindungen verlaufen innerhalb des Polygons oder auf dem Rand.



Konkaves Polygon; z.B. die Verbindung zwischen 5 und 11 schneidet die Polygonkanten.

Diese visuelle Bestimmung der Konvexität ist sehr aufwändig zu implementieren. Einfacher ist der Algorithmus von Paul Bourke, dessen Idee so veranschaulicht werden kann: ”Wenn man mit dem Fahrrad die Aussenkanten des Polygons entlangfährt und dabei nur nach links oder nur nach rechts lenken muß, ist das Polygon konvex. Wenn man zwischendurch mal die Lenkrichtung wechseln muß, dann ist es konkav. Dabei ist es egal, ob man im Uhrzeigersinn oder gegen den Uhrzeigersinn die Kanten abfährt.”

Um festzustellen, ob der nächste Punkt von der aktuellen Polygonkante aus gesehen links oder rechts liegt, benutzen wir die Geradengleichung in der Normalform, die sich aus der parametrisierten Geradengleichung ergibt, wenn man den Parameter r eliminiert:

$$\begin{aligned} g : \vec{u} &= \vec{p} + r \cdot \vec{v};\ r \in \mathbb{R} \\ \Rightarrow \begin{pmatrix} u_x \\ u_y \end{pmatrix} &= \begin{pmatrix} p_x \\ p_y \end{pmatrix} + r \cdot \begin{pmatrix} v_x \\ v_y \end{pmatrix} \\ \Rightarrow u_x &= p_x + r \cdot v_x \\ u_y &= p_y + r \cdot v_y \end{aligned}$$

Daraus ergibt sich die Funktion

$$F(u) = u_x \cdot v_y - u_y \cdot v_x + v_x \cdot p_y - v_y \cdot p_x$$

mit der Eigenschaft:

$$\begin{aligned} F(u) &= 0 \text{ für } u \text{ auf der Geraden.} \\ F(u) &< 0 \text{ für } u \text{ links der Geraden.} \\ F(u) &> 0 \text{ für } u \text{ rechts der Geraden.} \end{aligned}$$

”links” und ”rechts” sind dabei in der Richtung von $\vec{v}$ gemeint; d.h. die Seite wechselt, wenn man den Umlaufsinn des Polygons tauscht.

Es muß nur für jedes Punktepaar die Geradengleichung aufgestellt und der darauffolgende Punkt eingesetzt werden. Sobald einmal ein Wert herauskommt, der im Vorzeichen von den bisherigen Werten abweicht, ist das Polygon als konkav identifiziert und es brauchen keine weiteren Punkte mehr getestet zu werden. Wenn sich für alle Punkte immer das gleiche Vorzeichen ergibt, dann ist das Polygon konvex.

3.4.2 Schwerpunkt

Der Schwerpunkt (Baryzentrum) S eines Polygons berechnet sich als *baryzentrische Kombination* aus den Eckpunkten P_i des Polygons, versehen mit Gewichten m_i:

$$S = \sum_{i=0}^{n-1} m_i \cdot P_i$$

Für ein Dreieck gilt z.B.:

$$S_D = \frac{1}{3} \cdot p_0 + \frac{1}{3} \cdot p_1 + \frac{1}{3} \cdot p_2$$

Und für ein Viereck gilt:

$$S_V = \frac{1}{4} \cdot p_0 + \frac{1}{4} \cdot p_1 + \frac{1}{4} \cdot p_2 + \frac{1}{4} \cdot p_3$$

Bei baryzentrischen Kombinationen von Punkten gilt immer:

$$\sum_{i=0}^{n-1} m_i = 1$$

Die m_i sind im Allgemeinen verschieden und repräsentieren in der Physik die Massen der beteiligten Massenpunkte P_i.

Baryzentrische Kombinationen sind die einzige Möglichkeit **Punkte** sinnvoll additiv miteinander zu verknüpfen.

3.5 Kreis

- Gegeben seien der Mittelpunkt (x,y) und der Radius r eines Kreises.
- Bestimme die “anzuschaltenden” Pixel für einen Kreis mit Radius r um den Punkt (x,y).

Betrachte zunächst den Verlauf eines Kreises um $(0,0)$ im 2. Oktanten (Abbildung 3.5).

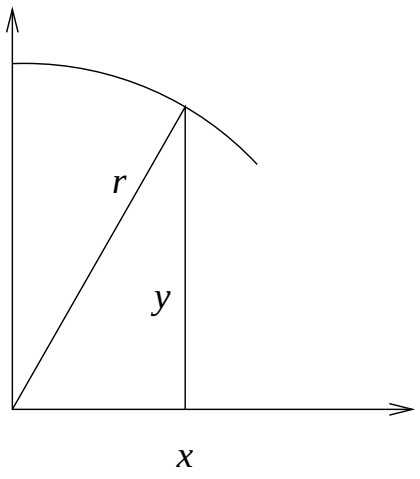


Abbildung 3.5: Verlauf des Kreises im 2. Oktanden

Jim Blinn nennt in ”A Trip Down the Graphics Pipeline” 15 verschiedene Arten dies Problem zu lösen. Wir wollen uns 2 etwas genauer ansehen.

3.5.1 Trigonometrische Funktionen

Die *x*- und *y*-Koordinaten ergeben sich sofort, wenn man die trigonometrischen Funktionen Sinus und Cosinus benutzt:

$$x = r \cdot \cos(\alpha); \qquad y = r \cdot \sin(\alpha) \;\; \alpha \in [0; 2\pi]$$

Als Zahl der Schritte für α bietet sich die Zahl der Längeneinheiten des Kreisumfanges an:

$$Winkelschritt = \frac{2\pi}{2\pi \cdot r} = \frac{1}{r}$$

Entsprechend sieht die `draw`-Methode der Klasse `TriCalcCircle.java` aus:

```
public void draw(CGCanvas cgc) {
  double step = 1.0 / (double)r;              // Soviele Winkelschritte
                                              // machen, wie der Umfang
                                              // Einheiten lang ist

  for(double winkel = 0.0; winkel < 2*Math.PI; winkel+=step) {
    cgc.setPixel((int)(x + r*Math.sin(winkel) + 0.5),
                 (int)(y + r*Math.cos(winkel) + 0.5));
  }
}
```

Der ständige Aufruf der trigonometrischen Funktionen kostet sehr viel Zeit. Eine Möglichkeit diesen Aufwand zu reduzieren, besteht darin, eine Tabelle von Sinus- und Cosinus-Werten anzulegen. Dies wird in der Klasse `TriTableCircle` getan:

```
package circle;

public class TriTableCircle extends Circle {
                                                  // Arrays fuer Sinus- und
                                                  // Cosinus-Werte.
  protected final static double[] sin = new double[360];
  protected final static double[] cos = new double[360];
  protected static boolean initialized = false;  // Flagge fuer 1. Instanz

  public TriTableCircle(int x, int y, int r) {
    super(x, y, r);
    if(!initialized) {                            // Falls dies 1. Instanz
      double step = Math.PI / 180;                // Arrays fuellen
      double winkel;

      for(int i=0; i<360; i++) {
        winkel = step * (double)i;
        sin[i] = Math.sin(winkel);
        cos[i] = Math.cos(winkel);
      }
      initialized = true;                         // Fuellung vermerken
    }
  }

  public void draw(CGCanvas cgc) {
    double step = Math.PI / 180;                  // Immer 360 Schritte

    for(int winkel = 0; winkel < 360; winkel++) {
      cgc.setPixel((int)(x + r*sin[winkel] + 0.5),
                   (int)(y + r*cos[winkel] + 0.5));
    }
  }
}
```

Durch die feste Zahl von 360 Schritten für jeden (noch so kleinen/großen) Kreis, geht ein Teil der gewonnenen Zeit wieder verloren bzw. entstehen Pixel, die nicht benachbart sind.

3.5.2 Bresenham-Algorithmus

Für einen Punkt (x,y) sei $F(x,y) = x^2 + y^2 - r^2$

Es gilt:

$$\begin{aligned} F(x,y) &= 0 \text{ für } (x,y) \text{ auf dem Kreis,} \\ F(x,y) &< 0 \text{ für } (x,y) \text{ innerhalb des Kreises.} \\ F(x,y) &> 0 \text{ für } (x,y) \text{ außerhalb des Kreises,} \end{aligned}$$

Verwende F als Entscheidungsvariable dafür, ob y erniedrigt werden muß, wenn x erhöht wird. F wird angewendet auf die Mitte M zwischen Zeile y und $y-1$ (siehe Abbildung 3.6).

$\Delta = F(x+1, y-\frac{1}{2})$

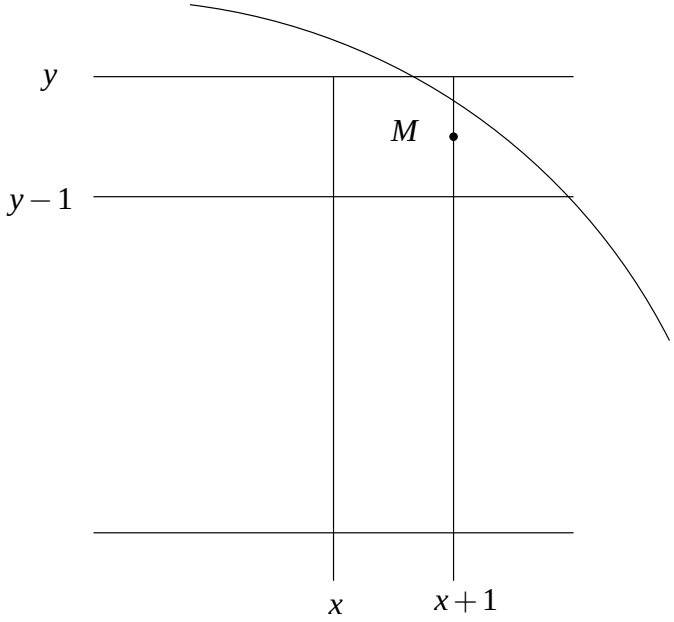


Abbildung 3.6: Testpunkt M für Entscheidungsvariable

Falls $\Delta < 0 \Rightarrow M$ liegt innerhalb $\Rightarrow (x+1, y)$ ist ideal
$\Delta \geq 0 \Rightarrow M$ liegt außerhalb $\Rightarrow (x+1, y-1)$ ist ideal

Idee: Entscheide anhand von Δ, ob y erniedrigt werden muß oder nicht, und rechne neues Δ aus.

$$
\begin{aligned}
\text{Sei “altes” } \Delta &= F(x+1, y-\tfrac{1}{2}) &&= (x+1)^2 + (y-\frac{1}{2})^2 - r^2 \text{ gegeben.} \\
\text{falls } \Delta &< 0 \Rightarrow \\
\text{“neues” } \Delta' &= F(x+2, y-\tfrac{1}{2}) &&= (x+2)^2 + (y-\frac{1}{2})^2 - r^2 = \Delta + 2x + 3 \\
\text{falls } \Delta &\geq 0 \Rightarrow \\
\text{“neues”} \Delta' &= F(x+2, y-\tfrac{3}{2}) &&= (x+2)^2 + (y-\frac{3}{2})^2 - r^2 = \Delta + 2x - 2y + 5 \\
\text{Startwert für } \Delta &= F(1, r-\tfrac{1}{2}) &&= 1^2 + (r-\frac{1}{2})^2 - r^2 = \frac{5}{4} - r
\end{aligned}
$$

Also ergibt sich:

```
public void drawBresenhamCircle1(CGCanvas cgc) {
  int xh, yh;
  double delta;

  xh = 0;                                        // Koordinaten retten
  yh = r;
  delta = 5.0/4.0 - r;

  while(yh >= xh) {                             // Fuer jede x-Koordinate
    cgc.setPixel(xh, yh);                       // Pixel im 2. Oktanden setzen

    if (delta < 0.0) {                          // Falls noch im Kreis
      delta+=2*xh + 3.0;                        // Abweichung aktualisieren
      xh++;                                     // naechste x-Koordinate
    }
    else {                                      // aus dem Kreis gelaufen
      delta+=2*xh - 2*yh + 5.0;                 // Abweichung aktualisieren
      xh++;                                     // naechste x-Koordinate
      yh--;                                     // naechste y-Koordinate
    }
  }
}
```

Substituiere `delta` durch `d := delta - 1/4`. Dann ergibt sich
als neue Startbedingung: `d := 5/4 - r - 1/4 = 1 - r`
als neue if-Bedingung: `if (d < 0)`
Da `d` nur ganzzahlige Werte annimmt, reicht der Vergleich mit 0.

Weitere Verbesserung:
Ersetze `2xh + 3` durch `dx` mit Initialwert `dx = 3;`
Ersetze `2xh - 2yh + 5` durch `dxy` mit Initialwert `dxy = -2*r + 5`
Es ergibt sich:

```
public void drawBresenhamCircle2(CGCanvas cgc) {
  int xh, yh, d, dx, dxy;

  xh = 0;                                    // Koordinaten retten
  yh = r;
  d = 1 - r;                                 // Startbedingung einstellen
  dx = 3;
  dxy = -2*r + 5;

  while(yh >= xh) {                          // Fuer jede x-Koordinate
    cgc.setPixel(xh, yh);                    // Pixel im 2. Oktanden setzen

    if (d < 0) {                             // Falls noch im Kreis
      d += dx; dx += 2; dxy += 2; xh++;      // Entscheidungsvariablen setzen
    }
    else {
      d +=dxy; dx += 2; dxy += 4; xh++; yh--;// Entscheidungsvariablen setzen
    }
  }
}
```

Um den ganzen Kreis zu zeichnen, wird die Symmetrie zum Mittelpunkt ausgenutzt.

Die Anzahl der erzeugten Punkte des Bresenham-Algorithmus für den vollen Kreis beträgt $4 \cdot \sqrt{2} \cdot r$ Punkte. Verglichen mit dem Kreisumfang von $2 \cdot \pi \cdot r$ liegt dieser Wert um 10% zu tief.

```
public void draw(CGCanvas cgc) {
  int xh, yh, d, dx, dxy;

  xh = 0;                                      // Koordinaten retten
  yh = r;
  d = 1-r;
  dx = 3;
  dxy = -2*r + 5;

  while(yh >= xh) {                          // Fuer jede x-Koordinate
    cgc.setPixel(x+xh, y+yh);                // alle 8 Oktanden werden
    cgc.setPixel(x+yh, y+xh);                // gleichzeitig gesetzt
    cgc.setPixel(x+yh, y-xh);
    cgc.setPixel(x+xh, y-yh);
    cgc.setPixel(x-xh, y-yh);
    cgc.setPixel(x-yh, y-xh);
    cgc.setPixel(x-yh, y+xh);
    cgc.setPixel(x-xh, y+yh);

    if (d < 0) {                                 // Falls noch im Kreis
      d+=dx; dx+=2; dxy+=2; xh++;                // passend aktualisieren
    }
    else {                                       // Aus dem Kreis gelaufen
      d+=dxy; dx+=2; dxy+=4; xh++; yh--;         // passend aktualisieren
    }
  }
}
```

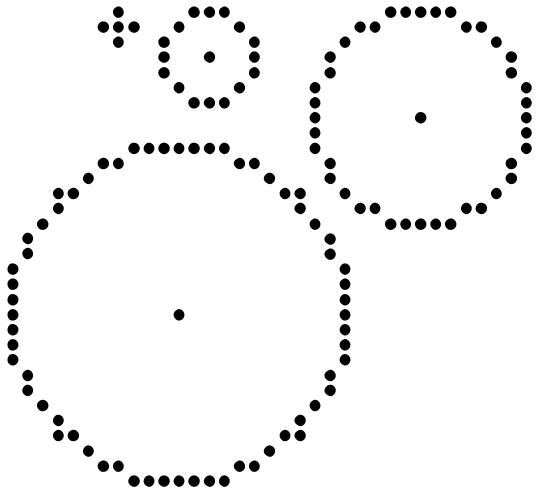


Abbildung 3.7: Vom Bresenham-Algorithmus erzeugte Kreise

3.6 Ellipse

die beim Kreis angewendeten Techniken können mit einigen Änderungen auf die Ellipse übertragen werden. Eine Ellipse ist definiert als die Menge aller Punkte P, deren Abstandssumme zu zwei gegebenen Punkten P_1 und P_2 insgesamt $2 \cdot a$ beträgt.

Abbildung 3.8 zeigt die Beziehungen innerhalb einer Ellipse. Es gilt

$e = \sqrt{a^2 - b^2}, \quad \frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$

$x = a \cdot \cos(r), \;\; y = a \cdot \sin(r), \;\; 0 \leq r \leq 2 \cdot \pi$

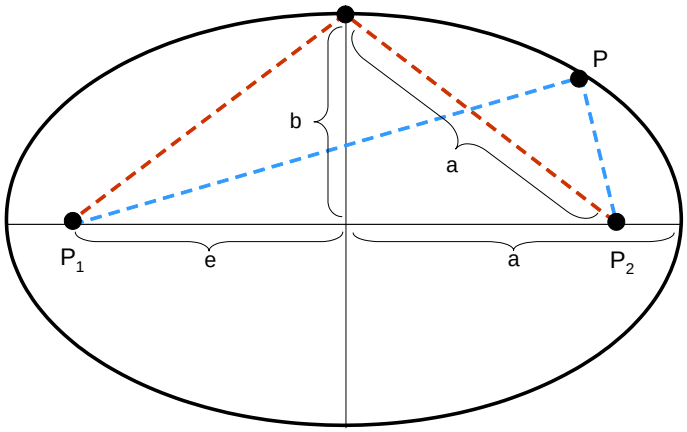


Abbildung 3.8: Beziehungen in einer Ellipse