11 Grafikprogrammierung

Die Umsetzung quantitativer Information in eine grafische Darstellung ist nicht erst seit der Erfindung des Computers üblich geworden.

Ein Bild sagt mehr als 1000 Worte – und erst recht mehr als 1000 Zahlen.

Die Anzeige und Bearbeitung von Bildern und der Schnitt von Urlaubsvideos ist längst eine selbstverständliche Aufgabe von Computern aller Art, auch des heimischen PCs. Keine Webseite verzichtet mehr auf Grafiken, Fotos oder gar kleine Animationen. Jeder moderne Browser kann solche Elemente darstellen. Textverarbeitungssysteme präsentieren das entstehende Dokument grafisch fast exakt so wie der Drucker es ausgeben wird. Auch die Bedienung von Computern erfolgt heute fast ausschließlich mithilfe grafischer Bediensysteme. Nicht zu vergessen sind auch Spiele, die aufwändige 3-dimensionale Szenen in Echtzeit berechnen und verblüffend realistisch auf dem Bildschirm darstellen.

Alle derartigen Anwendungen benötigen ein Grafiksystem, das in der Lage ist, Mausbewegungen, Fenster, Bilder und Filme sehr schnell auf den Monitor zu zaubern, sie zu verändern, vergrößern, verkleinern, verschieben, etc. Viele der aufgezählten Grafikoperationen sollen nicht die CPU belasten, sondern selbständig von einem separaten Grafiksystem ausgeführt werden.

Voraussetzung für die Bearbeitung von Grafiken und Bildern ist die Möglichkeit zur Grafikprogrammierung. In diesem Kapitel werden einige Grundlagen dazu besprochen. Weitere Informationen zum Thema Computergrafik kann man dem Buch von Foley, van Dam, Feiner und Hughes entnehmen.

11.1 Hardware

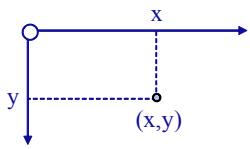
Standardgrafiksysteme sind in vielen Fällen bereits auf der Hauptplatine des Rechners integriert. Weitergehende grafische Fähigkeiten erwerben Rechner normalerweise durch gesonderte Grafikkarten. Eine heutige Grafikkarte hat eine eigene CPU, den so genannten *Grafik-Prozessor* (auch *GPU* genannt) und bis zu 8 GB eigenen Speicher, der auch als *VRAM* (Video-Ram) bezeichnet wird – neuerdings auch als *GDDR*: (Graphics Double Data Rate Ram). Die Ausgabe analoger Signale wird heute kaum noch unterstützt. Üblich ist die Ausgabe digitaler Signale im DVI- und HDMI-Format. Mit dem digitalen Signal kann man die heute üblichen Flachbildschirme direkt ansteuern.

Der Grafik-Prozessor verfügt u.a. über folgende elementaren Fähigkeiten:

- Bildpunkte lesen und schreiben,
- rechteckige Bildausschnitte verschieben,
- Linien und Dreiecke erzeugen,
- Texturen erzeugen,
- Unterstützung von 3D Grafikfunktionen.

11.1.1 Auflösungen

Für den Anwender ist zunächst die logische Sicht des Bildschirms von Interesse. In der oberen linken Ecke befindet sich der Ursprung eines Koordinatensystems:



Das Bild besteht aus einzelnen Punkten, die zusammen mit ihrer Färbung als Bildelemente *Pixel* (= Picture Elements) bezeichnet werden. Die Anzahl der Pixel ist abhängig von den Fähigkeiten der Grafikkarte. Gängige Formate sind:

x-maximal	y-maximal	Gesamtzahl der Pixel
1024	768	786 432
1280	1024	1 310 720
1600	1200	1 920 000
1920	1080	2 073 600
1920	1200	2 304 000
2880	1800	5 184 000

11.1.2 Farben

Wichtiger noch als die Zahl der Bildpunkte ist die Zahl der verschiedenen darstellbaren Farben. Aus technischen Gründen waren vor einiger Zeit meist nur 16 verschiedene Farbwerte möglich. Je nach Grafikkarte wurden 1, 2 oder 3 Byte pro Farbwert unterstützt. Heute sind es 3 Byte oder mehr pro Farbwert; das erlaubt bis zu 2^{24}=16777216 mögliche Farbwerte.

Der Speicherbedarf für eine Bilddarstellung derart vielen Farbwerten ist natürlich vergleichsweise hoch. Um z.B. ein Bild im Format 1920×1200 mit 3 Byte pro Farbwert zu speichern benötigt man 1920 ·1200· 3 B = 6912 MB.

Um Bilder auf den alten Kathodenstrahlröhren flimmerfrei darzustellen war eine Bildwiederholfrequenz von 75/sec notwendig, so dass dies bei hohen Auflösungen zu einer extrem

hohen Datenrate führte. Dieses Problem hat sich bei den neueren Bildschirmen mit Flüssigkristallanzeigen weitgehend erledigt, da diese von Hause aus flimmerfrei sind. Für die Darstellung von 3D Bildern auf 3D Monitoren werden zwar 120 Hz benötigt, die Bildinformation wird jedoch auf zwei DVI-Leitungen mit je 60 Hz übertragen.

11.2 Rastergrafik und Vektorgrafik

Um ein Bild zu beschreiben, gibt es im Prinzip zwei naheliegende Möglichkeiten. Entweder stellt man es als zweidimensionale Tabelle von Pixeln dar, oder man erklärt, wie man das Bild auf Basis einfacher Zeichenroutinen erzeugen kann. Die beiden Grafiken in der folgenden Abbildung lassen sofort die Vor- und Nachteile erkennen. Der gezeigte Baum erscheint dermaßen unregelmäßig, dass nur eine vollständige Aufzählung aller Punkte `(x,y)` zusammen mit ihren jeweiligen Farbwerten, `f(x,y)` in Frage kommt. Diese Darstellung benötigt unabhängig von der dargestellten Szene immer gleich viel Speicherplatz. Eine solche Beschreibung einer Grafik durch Tabellierung aller Pixel nennt man *Rastergrafik*.

Das gleiche Verfahren könnten wir auch bei dem Smiley verwenden, doch ist hier diese Beschreibung offensichtlich unökonomisch. Wir haben eine gelb gefüllte Kreisfläche, mit zwei Ellipsen, einen großen und zwei kleinen Kreisbögen, die wir mit wenigen Koordinaten, Radien, Winkeln und Halbachsen spezifizieren können. In jedem Falle würden wir für die exakte Angabe der Grafik nur wenige kB an Informationen benötigen.

Nehmen wir an, dass uns einige Grafikprimitiva zur Verfügung stehen, wie zum Beispiel das Zeichnen von Punkten, Strecken, Ovalen, Kreissegmenten und Rechtecken, so könnten wir ein kurzes Programm formulieren, das den Smiley zeichnet. Wir hätten auf jeden Fall eine kompaktere Repräsentation gewonnen, als sie bei Pixelgrafik möglich wäre. Eine derartige Darstellung bezeichnet man als *Vektorgrafik*.

Abb. 11.1: Baum und Smiley

Eine Variante der Vektorgrafik, die wir uns später noch ansehen wollen, heißt *Turtlegrafik*. Dabei stellen wir uns vor, dass ein Zeichenstift die Linien der Abbildung nachzeichnet. Der Stift beschreibt einen Weg auf der Zeichenebene, den wir durch viele kleine Linienstücke approximieren. Insgesamt können wir den Vorgang des Zeichnens durch eine Folge dreier Befehle beschreiben: *vorwärts laufen*, *Laufrichtung verändern*, *Stift hoch/runter*. (Ganz analog haben wir in einem früheren Kapitel ein Zeichenprogramm geschrieben, mithilfe dessen wir mit der Maus freihandzeichnen konnten, indem wir die sukzessiven Positionen der Maus auf der Zeichenebene mit Linienstückchen zusammensetzten, siehe Abb. 8.21.)

Vektorgrafik eignet sich gut für technische Zeichnungen, Grafiken oder künstlich erzeugte Bilder wie z. B. Szenen für Computerspiele, aber weniger für natürliche Bilder wie etwa Photos.

11.2.1 Umrechnung in Rastergrafik

Auf der untersten Ebene ist Rastergrafik für viele Geräte die einfachste und flexibelste Schnittstelle. Dazu zählen Bildschirme und alle Arten von Druckern. Allerdings darf man nicht vergessen, dass die *Auflösung*, das heißt die Anzahl der gespeicherten Punkte pro Längeneinheit, bei dem gespeicherten Bild und bei dem durch Drucker oder Bildschirm darstellbaren Bild nicht notwendig übereinstimmen. Es muss also umgerechnet und ggf. noch interpoliert werden. Für einen gespeicherten Bildpunkt muss man den in der Auflösung des Bildschirms nächstliegenden Pixel finden. Eine Interpolation ist nötig, wenn die Bildinformation geringer ist, als die zur Darstellung vorhandenen Pixel. Der Wert eines Pixels zwischen zwei durch das gespeicherte Bild spezifizierten Punkten wird dann als Zwischenwert der vorhandenen Farbwerte angenommen.

Für die Darstellung auf Bildschirmen oder Druckern muss Vektorgrafik immer in Rastergrafik umgerechnet werden. Die *wahre Zeichnung* wird i.A. zwischen den Rasterpunkten hindurchführen. Geeignete nahe liegende Rasterpunkte müssen ausgewählt und als *Alias-Punkte* gemalt werden. Wir wollen dies am Beispiel der geraden Linie diskutieren. Routinen zum Zeichnen von Linien werden zwar von jedem Grafikpaket als elementare Funktionen zur Verfügung gestellt, trotzdem wollen wir den dafür verwendeten Algorithmus vorstellen, da er die Grundlagen der Darstellung von *gerasterten Zeichnungen* demonstriert.

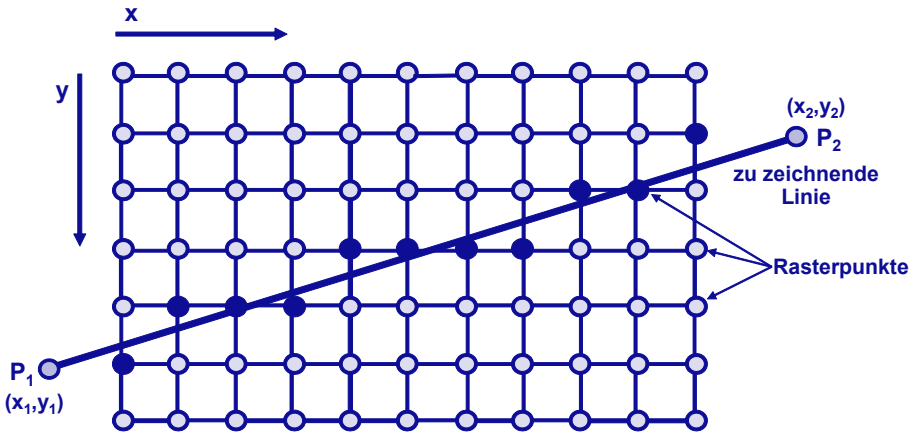


Abb. 11.2: Zeichnen einer gerasterten Linie

Das Bild zeigt eine Linie und die ihr nahe liegenden Rasterpunkte hervorgehoben. Die *gefüllten Punkte* wurden zur Darstellung ausgewählt. Pro senkrechte Rasterlinie wurde jeweils ein Punkt bestimmt – und zwar immer der der Geraden nächstliegende. Zu jedem `x`-Wert des Rasters gibt es einen echten Punkt `P(x,y)` auf der Linie, zu dem wir einen y-Wert des Rasters, y_R, bestimmen müssen. Wir können diesen einfach finden, indem wir y zur nächsten ganzen Zahl runden, in Java z.B. durch `yR = (int)(y+0.5);`

Eine Linie von `P(x1,y1)` zu `P(x2,y2)` hat eine Steigung `m=(y2-y1)/(x2-x1)` und kann in Java durch die folgende Schleife dargestellt werden. Dabei nehmen wir an, dass der Abstand zwischen Rasterpunkten jeweils 1 beträgt und *putPixel* einen Punkt zeichnet.

```
double  y = y1 + 0.5;
for (int x = x1; x <= x2; x++ ){
   putPixel(x, (int) y , Farbe);
   y += m;    }
```

Dieser Algorithmus ist sehr elementar, hat aber den offensichtlichen Nachteil der notwendigen Gleitpunktarithmetik.

11.2.2 Bresenham Algorithmus

J. E. Bresenham veröffentlichte 1965 einen Algorithmus zur optimierten Rasterdarstellung von Linien. Dieser kommt ganz ohne Gleitpunktarithmetik aus und verwendet nur Inkrementier-Operationen. Der Algorithmus wird in fast allen Hardware- bzw. Softwareimplementierungen zum Zeichnen von Linien verwendet. Da er diesen Algorithmus zur Verbesserung einer Plottersteuerung verwenden konnte, erhielt Bresenham sogar Patentrechte dafür.

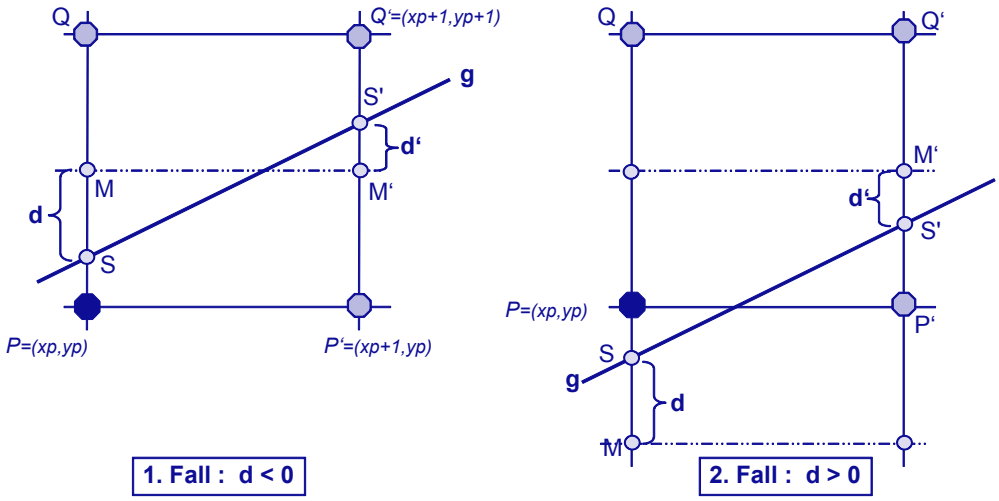


Abb. 11.3: Mittelpunkt-Algorithmus von Bresenham

Die zu zeichnende Gerade *g* sei durch die Punkte $P1 = (x1, y1)$ und $P2 = (x2, y2)$ gegeben. Wir können voraussetzen, dass ihre Steigung $m = (y2 - y1)/(x2 - x1)$ zwischen 0 und 1 liegt, denn alle anderen Fälle lassen sich durch naheliegende Symmetrien auf diesen Fall reduzieren. Wir nehmen an, ein Punkt P mit den Koordinaten (xp, yp) sei gerade gemalt worden. Die x-Koordinate des nächsten Punktes in *x*-Richtung steht schon fest: $xp+1$. Gesucht wird der zugehörige *y*-Wert. Dieser kann entweder $yp+1$ oder yp sein, je nachdem, ob die Gerade *g* die senkrechte Linie $x = xp+1$ über oder unter dem Mittelpunkt *M'* der Strecke *P'Q'* schneidet.

Ist d' die Differenz der y-Koordinate des Schnittpunktes S' von der des Mittelpunktes M', dann entspricht dies den Fällen $d' > 0$ bzw $d' < 0$. Der Fall $d' = 0$ kann beliebig entschieden werden. Die erste Idee von Bresenham ist es nun, den Differenzwert d' aus dem vorigen Differenzwert d zu berechnen. Dabei treten zwei Fälle auf, die man leicht aus der obigen Figur abliest:

- War $d < 0$, so gilt offensichtlich $d' - d = m = (y2 - y1)/(x2 - x1)$ und daher $d' = m + d$.
- War $d > 0$, so gilt stattdessen $1 - d + d' = m = (y2 - y1)/(x2 - x1)$ und daher $d' = m + d - 1$.

Bresenham wollte in seinem Algorithmus Operationen mit Gleitpunktzahlen sowie Divisionen vermeiden, da diese auf damaliger Plotter-Hardware sehr aufwändig waren. Würde man die obigen Formeln in ein Programm umsetzen, müssten m, d und d' mit Gleitpunktgenauigkeit berechnet werden. Der zweite Trick von Bresenham ist, d und d' mit dem konstanten Faktor $(x2 - x1) \times 2$ zu multiplizieren und statt ihrer mit den Werten $D = d \times (x2 - x1) \times 2$ bzw. $D' = d' \times (x2 - x1) \times 2$ zu rechnen, denn diese sind genau dann positiv oder negativ, wenn das auch für d bzw. d' zutrifft, können aber mit Ganzzahlarithmetik berechnet werden. Man erhält also:

$$D < 0 \Rightarrow D' := D + (y2 - y1) \times 2$$
$$D > 0 \Rightarrow D' := D + (y2 - y1) \times 2 - (x2 - x1) \times 2$$

Damit ergibt sich sofort folgender optimierter Linienalgorithmus:

```
void OptLinie(int x1, int y1, int x2, int y2, Color Farbe){
    int dy = (y2-y1);
    int dx = (x2-x1);
    int incKN = dy*2;
    int incGN = (dy-dx)*2;
    int D = incGN-dx;
    int y = y1;
    putPixel(g, x1, y1, Farbe);
    for (int x=x1+1; x <= x2; x++ ){
        if (D <= 0) D += incKN;
        else { D += incGN; y++; }
        putPixel(g, x, y, Farbe);
  }
}
```

Die scheinbar unmotivierte Erweiterung mit dem Faktor 2 war notwendig, um mit einem ganzzahligen Initialwert von D beginnen zu können. Die beiden Multiplikationen mit 2 können natürlich durch Additionen oder Shift-Operationen ersetzt werden.

11.2.3 Quadtrees und Octrees

Betrachtet man das Bild des Baumes aus Abb. 11.1, so fallen links und rechts unten die relativ großen weißen Bereiche auf. Es erscheint verschwenderisch für jeden Punkt in diesem Bereich den immer gleichen Pixelwert explizit zu speichern. Man könnte zum Beispiel versuchen,