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

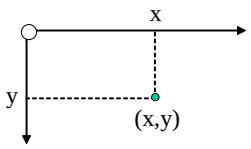
Standardgrafiksysteme sind in vielen Fällen bereits auf der Hauptplatine des Rechners integriert. Weitergehende grafische Fähigkeiten erwerben Rechner normalerweise durch gesonderte Grafikkarten. Eine heutige Grafikkarte hat eine eigene CPU, den so genannten *Grafik-Prozessor*, bis zu 256 MB eigenen Speicher, der auch als *VRAM* (Video-Ram) bezeichnet wird und jeweils eine Schnittstelle für die Ausgabe von analogen und digitalen Signalen. Das analoge Signal wird mit einem *Digital-Analog-Converter* (*DAC*) erzeugt. An diese Schnittstelle können konventionelle Bildschirme mit Kathodenstrahlröhre angeschlossen werden. Mit dem digitalen Signal kann man Flüssigkristallanzeigen direkt ansteuern.

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

- Bildpunkte lesen und schreiben,
- rechteckige Bildausschnitte verschieben,
- Linien und Rechtecke erzeugen,
- Ellipsen und Kreise 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
640	480	307 200
800	600	480 000
1024	768	786 432
1280	1024	1 228 800

Nur das erste dieser Formate hat weniger Bildpunkte als ein Standard-Fernsehbild mit ca. 400 000 Bildpunkten.

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. Heute werden je nach Grafikkarte folgende Farbzahlen unterstützt:

Bytes pro Pixel	Mögliche Farbwerte
1	256
2	65 536
3	16 777 216

Um Bilder darzustellen, genügen 256 Farbwerte nicht. Andererseits ist der Speicherbedarf für eine Bilddarstellung mit mehr als 256 Farben vergleichsweise hoch:

	Speicherbedarf in kB bei		
x/y-Format	1 B/Pixel	2 B/Pixel	3 B/Pixel
640 x 480	300	600	900
800 x 600	469	938	1406
1024 x 768	768	1536	2304
1280 x 1024	1311	2622	3933

Um Bilder auf einem gängigen Bildschirm mit Kathodenstrahlröhre flimmerfrei darzustellen, sind ca. 75 Bildwiederholungen pro Sekunde erforderlich. Dies führt bei hohen Auflösungen und mehreren Bytes pro Pixel offensichtlich zu Datenraten, die auch heute noch an der Grenze des Machbaren liegen.

Bildschirme mit Flüssigkristallanzeigen erfordern keine so hohen Bildwiederholfrequenzen, da die Bilder sowieso flimmerfrei sind. Außerdem ist eine wesentlich niedrigere Datenrate erforderlich, wenn diese Bildschirme direkt an eine digitale Schnittstelle angeschlossen werden. Derartige Bildschirme sind mittlerweile sehr weit verbreitet und ersetzen die älteren Röhrenbildschirme mehr und mehr.

Die Farbdarstellung beim Fernseher erfolgt nicht durch Übertragung von Farbwerten in Form von Bytes pro Bildelement, sondern durch *analoge Farbsignale*. Die Anzahl der auf diese Weise übertragbaren Farbwerte ist aber sehr hoch – vergleichbar Bildern mit 2-Byte-Farbinformationen pro Bildelement. *Dia-Scanner* bieten die Möglichkeit an, Dias in digitaler Form z.B. auf einer *CD-ROM* zu speichern. Dabei können hochauflösende Formate mit etwa 2500 x 3000 Bildelementen mit je 3-Byte-Farbinformationen pro Bildelement verwendet werden. Damit ergibt sich folgender Vergleich:

Farbfernsehbild (nicht *hochauflösend*)	Computerbild mittlerer Auflösung 768 x 1024 x 3 Byte	Farbdia in hoher Auflösung
ca. 800 kB	ca. 2,3 MB	ca. 20 MB

11.2 Grafikroutinen für Rastergrafik

Im letzten Abschnitt haben wir ausschließlich die Grundlagen der *Rastergrafik*, d.h. Grafik auf der Basis von diskreten Bildpunkten, die mit Farbinformationen versehen sind, diskutiert. Die Verwendung von Rastergrafik ist keineswegs selbstverständlich. Die ersten Grafikbildschirme arbeiteten nach dem Prinzip der *Vektorgrafik*. Dieses Prinzip wird auch heute noch von Plottern angewendet. Man hat einen Stift und gibt direkte Fahrbefehle zur Steuerung des Stiftes.

Für Bildschirme ist dieses Verfahren weitgehend unüblich geworden. Zur Definition von Bildern ist Vektorgrafik in vielen Fällen sogar besser geeignet als Rastergrafik. Wir werden in einem der folgenden Abschnitte *Turtle-Grafik* einführen und auf dieser Basis viele interessante Bilder definieren. Im Grunde ist Turtle-Grafik nichts anderes als *Vektorgrafik*. Vektorgrafik wird heute in vielen Fällen zur Speicherung und Definition von Bildern auf einer *höheren Ebene* wieder erfunden. Auf der untersten Ebene ist Rastergrafik für viele Geräte die einfachste und flexibelste Schnittstelle. Dazu zählen Bildschirme und alle Arten von Druckern – bis auf die Plotter.

Die Übertragung einer Zeichnung auf Rastergrafik erfordert eine Anpassung an das Rastermaß. 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. Das folgende Bild zeigt eine Linie und die nahe liegenden Rasterpunkte hervorgehoben:

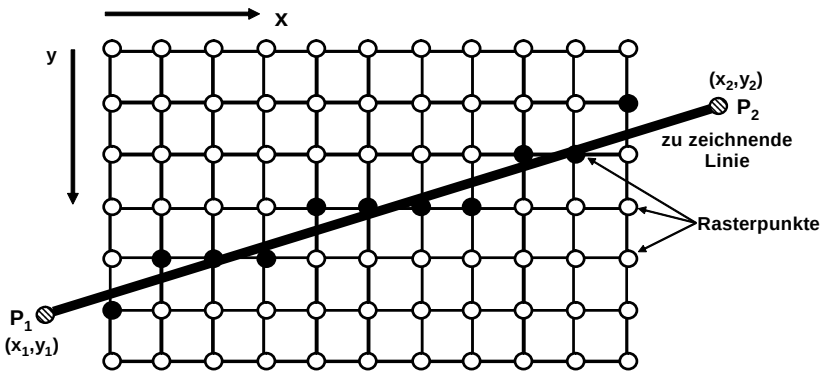


Abb. 11.1: *Zeichnen einer gerasterten Linie*

Die *gefüllten Punkte* wurden zur Darstellung ausgewählt. Pro senkrechte Rasterlinie wurde jeweils ein Punkt bestimmt – und zwar immer der der Geraden nächstliegende. Dafür können wir einen elementaren Algorithmus angeben:

```
void Linie(int x1, int y1, int x2, int y2, Color Farbe){
double dy = y2-y1;
double dx = x2-x1;
double  m = dy/dx;
double  y = y1 + 0.5;
for (int x = x1; x <= x2; x++ ){
  putPixel(x, (int) y , Farbe);
  y += m;
  }
}
```

Dabei nehmen wir an, dass der Abstand zwischen Rasterpunkten jeweils 1 beträgt und *putPixel* einen Punkt in der gewünschten Farbe setzt. Dieser Algorithmus ist sehr elementar, hat aber einen offensichtlichen Nachteil: Bei jedem Schritt ist eine aufwändige Gleitpunktarithmetik erforderlich. Dafür kann allerdings die Punktauswahl der *round*-Methode überlassen werden.

11.2.1 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.

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.

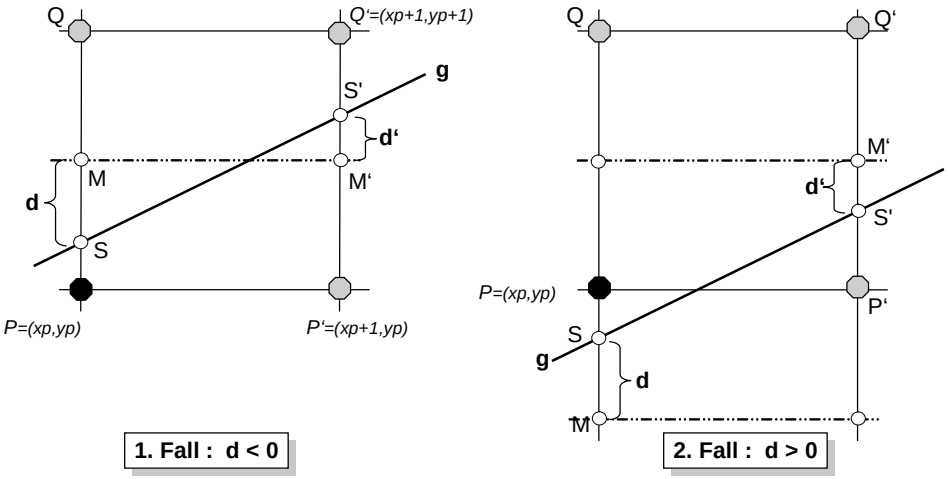


Abb. 11.2: *Mittelpunkt-Algorithmus von Bresenham*

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.

11.3 Einfache Programmierbeispiele

Zu neueren Betriebssystemen mit grafischer Benutzeroberfläche gehört normalerweise ein Paket mit mehr oder weniger elementaren Grafikroutinen. Bei Windows handelt es sich um das so genannte *GDI* (*graphic device interface*). Java bietet eine Grafikschnittstelle, die unabhängig vom jeweiligen Betriebssystem ist. Diese haben wir am Ende des dritten Kapitels vorgestellt. Erstaunlicherweise sind keine Methoden zum Malen einzelner Punkte (Pixel) und zum Setzen der Linienstärke vorgesehen. Seit Version 1.2 des JDK steht eine erweiterte