Graphikprimitive

Um graphischen Output auf einem Gerät zu erzeugen, muss die verwendete Programmiersprache Befehle dafür zur Verfügung stellen. Die mit diesen Befehlen erzeugbaren einfachen Graphikbausteine nennt man auch Graphikprimitive. Solche Bausteine sind neben einfachen Zeichnungen auch Formatierungs-Anweisungen und Meta-Informationen. Die wichtigsten solchen Befehle sind:

In 2D:
- Punkte, Linien
- Polygone, Kreise, Ellipsen und andere Kurven, alles auch gefüllt
- Bitmap-Operationen
- Buchstaben und Zeichen

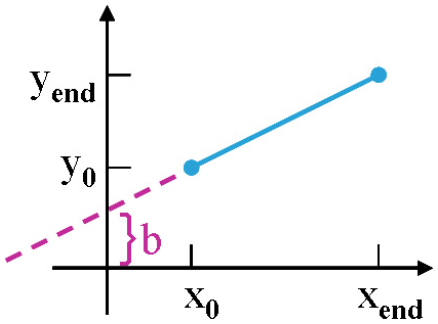
In 3D:
- Dreiecke und andere Polygone
- Freiformflächen

Darüber hinaus braucht man noch Befehle um die Eigenschaften der Primitive zu definieren, z.B. Farbe, Füllmuster, Textur, Materialeigenschaft, Transparenz. Diese Befehle bewirken meist, dass alle danach erzeugten Primitive die zuletzt definierten Eigenschaften annehmen.

Linienalgorithmen

Insbesondere das Zeichnen gerader Linien auf Rastergeräten ist eine wichtige Operation. Das Basisverfahren DDA (Digital Differential Analyzer) wurde von Bresenham durch geschickte Umformung so gestaltet, dass es nur mit Integer-Operationen auskommt und damit *schneller* und *leichter in Hardware implementierbar* wurde.

Notation: eine Linie wird in der Form y = mx + b angegeben,
wobei m den Anstieg der Linie beschreibt
und (0,b) der Schnittpunkt mit der y-Achse ist.



Aus den Endpunkten (x_0,y_0) und (x_{end},y_{end}) der Linie lassen sich m und b so berechnen:

$m = (y_{end} - y_0) / (x_{end} - x_0)$ $\quad b = y_0 - mx_0$

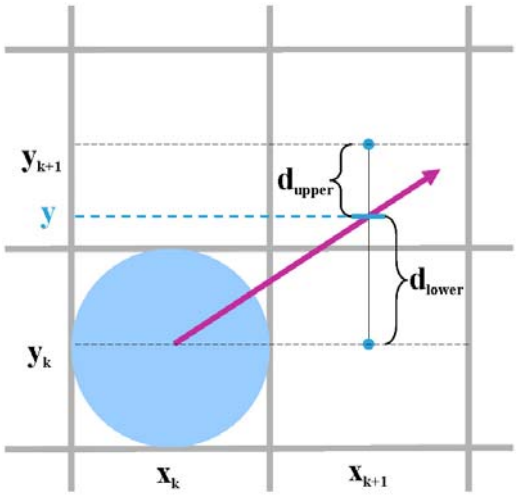
Der einfache ***DDA-Algorithmus*** für |m| < 1 zählt zu y_0 für jeden Schritt nach rechts (x+=1) den Wert m dazu und rundet das Ergebnis danach auf ganze Zahlen. Dadurch entsteht eine Linie, bei der für jeden x-Wert genau ein Pixel für die Linie erzeugt wird.

```
dx = xEnd – x0; dy = yEnd – y0;
m = dy / dx;

x = x0;  y = y0;
setPixel (round(x), round(y));

for (k = 0; k < dx; k++)
  { x += 1; y += m;
  setPixel (round(x), round(y)}
```

Für |m| > 1 werden x und y vertauscht, und das Verfahren wird in senkrechter Richtung durchgeführt. Auch der nachfolgende Bresenham-Algorithmus wird nur für 0 < |m| < 1 dargestellt, die anderen Richtungen funktionieren durch Spiegelung und Rotation um 90°.

Der ***Bresenham-Algorithmus*** erzeugt exakt dasselbe Ergebnis wie der einfache DDA, verwendet jedoch nur Integer-Arithmetik. Er ist dadurch schneller, leichter in Firm- oder Hardware zu implementieren, und überdies lässt er sich auch einfach für andere Kurven anpassen, z.B. Kreise, Ellipsen, Spline-Kurven usw.

Für $0 < |m| < 1$ wird ausgehend von der bekannten Lage des Pixels in der Spalte x_k für x_{k+1} nicht der exakte y-Wert berechnet, sondern lediglich eine Entscheidung getroffen, ob y_k oder y_{k+1} näher zum exakten y-Wert liegen.



Aus $y = mx + b$ folgt für die Spalte rechts von x_k der exakte y-Wert

$$y = m\cdot(x_k + 1) + b$$

Der Abstand zu y_k ist $\mathbf{d_{lower}} = y - y_k = m(x_k + 1) + b - y_k$
der Abstand zu y_{k+1} ist $\mathbf{d_{upper}} = (y_k + 1) - y = y_k + 1 - m(x_k + 1) - b$

Wenn nun die Differenz $d_{lower} - d_{upper} = 2m\cdot(x_k + 1) - 2y_k + 2b - 1$ negativ ist, dann wird der untere Punkt (x_{k+1},y_k) gewählt, wenn sie positiv ist wird (x_{k+1},y_{k+1}) gewählt.

Setzt man für $m = \Delta y/\Delta x$ ein ($\Delta x = x_{end} - x_0$, $\Delta y = y_{end} - y_0$) , und multipliziert diese Differenz mit Δx so erhält man eine Entscheidungsvariable $p_k = \Delta x\cdot(d_{lower} - d_{upper}) = 2\Delta y\cdot x_k - 2\Delta x\cdot y_k + c$,
die *das gleiche Vorzeichen* wie $d_{lower} - d_{upper}$ hat, aber keine Division erfordert.

Nun kann man ganz leicht aus der Entscheidungsvariablen für x_k
$p_k = 2\Delta y\cdot x_k - 2\Delta x\cdot y_k + c$ die Entscheidungsvariable für x_{k+1} berechnen:

$$\mathbf{p_{k+1}} = 2\Delta y\cdot x_{k+1} - 2\Delta x\cdot y_{k+1} + c + p_k - 2\Delta y\cdot x_k + 2\Delta x\cdot y_k - c = \mathbf{p_k + 2\Delta y - 2\Delta x\cdot(y_{k+1} - y_k)}$$

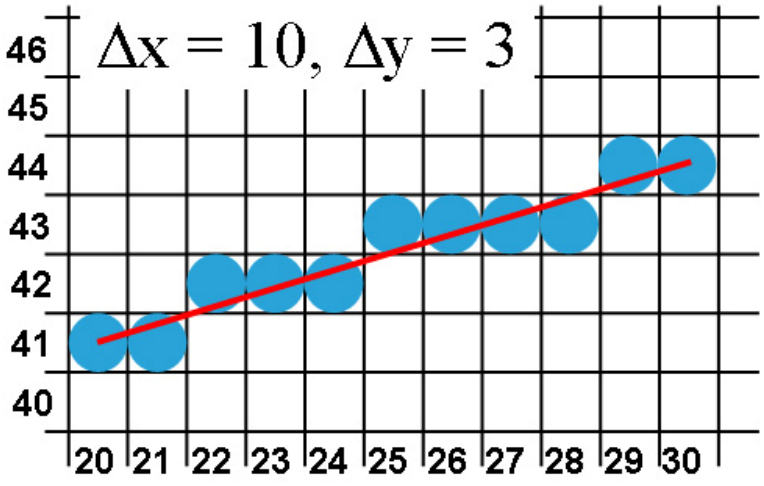
also lediglich durch Addition einer Zahl, die für alle Punkte der Linie konstant bleibt. $p_0 = 2\Delta y - \Delta x$

Damit sieht der Bresenham-Algorithmus etwa so aus:

1. store left line endpoint in (x_0,y_0)
2. plot pixel (x_0,y_0)
3. calculate constants Δx, Δy, $2\Delta y$, $2\Delta y - 2\Delta x$, and obtain $p_0 = 2\Delta y - \Delta x$
4. At each x_k along the line, perform test:
   if $p_k<0$
   then plot pixel (x_k+1,y_k); $p_{k+1} = p_k + 2\Delta y$
   else plot pixel (x_k+1,y_k+1); $p_{k+1} = p_k + 2\Delta y - 2\Delta x$
5. perform step 4 ($\Delta x - 1$) times.

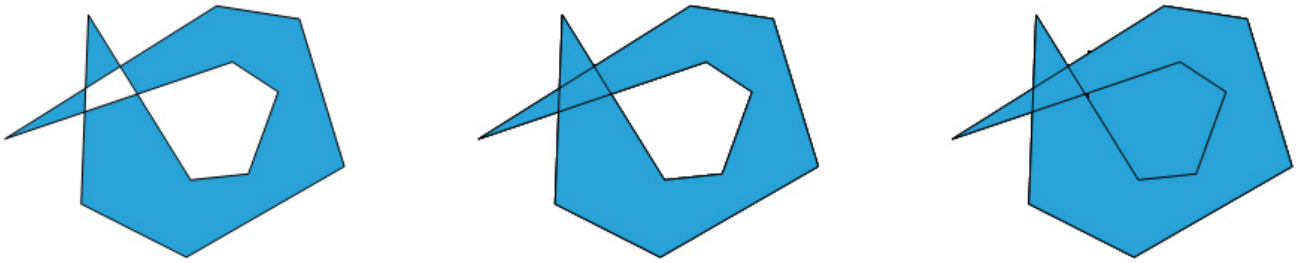
Beispiel:

k	p_k	(x_{k+1},y_{k+1})
		(20,41)
0	-4	(21,41)
1	2	(22,42)
2	-12	(23,42)
3	-6	(24,42)
4	0	(25,43)
5	-14	(26,43)
6	-8	(27,43)
7	-2	(28,43)
8	4	(29,44)
9	-10	(30,44)



Gefüllte Flächen

Bevor man mit dem Füllen von Flächen beginnt, muss man sich zuerst fragen, was denn zu füllen sei. Bei einer einfachen geschlossenen Kurve ist „innen“ leicht zu definieren, was aber bei komplizierteren Kurven?



Odd-Even-Rule: zieht man von einem Punkt aus einen beliebigen Halbstrahl, so ist der Punkt innerhalb, wenn die Zahl der Schnitte mit der Kurve ungerade ist, ansonsten ist der Punkt außerhalb (links).

Nonzero-Winding-Number-Rule: Punkte sind außerhalb, wenn sich auf einem beliebigen Halbstrahl gleich viele im Uhrzeigersinn und gegen den Uhrzeigersinn verlaufende Kurvenkanten befinden, ansonsten innerhalb (Mitte).

All-In-Rule: alles, was irgendwie umschlossen ist, ist innen. Wird selten verwendet, meist beim Pokern☺ (rechts).

Wenn man einmal entschieden hat, was innen ist, dann kann man entweder Zeile für Zeile füllen (*Scanline*-Algorithmen) oder von einem inneren Punkt ausgehend in alle Richtungen füllen (*Flood-Fill*-Algorithmen).

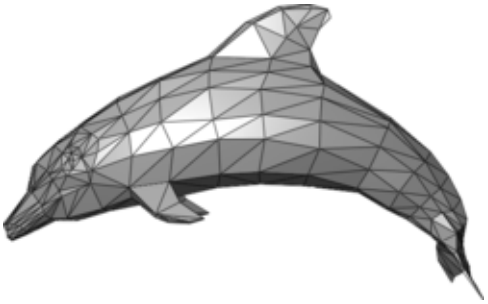
Ein Polygon heißt konvex wenn alle inneren Winkel kleiner als 180° sind, andernfalls konkav. Da konvexe Polygone viel weniger Sonderfälle erzeugen, sind viele Algorithmen für konvexe Polygone ausgelegt. Daher braucht man Methoden um konkave Polygone in mehrere konvexe Polygone zu zerteilen.

Buchstaben und Zeichen

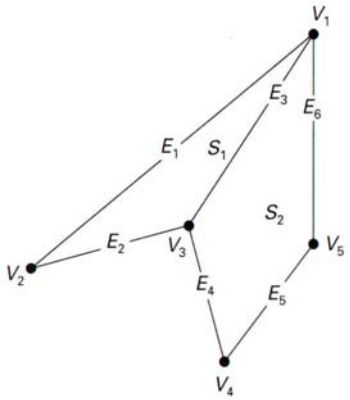
Die Definition von Schrift erfolgt über Fonts und Eigenschaften. Fonts mit Serifen (oben) eignen sich besser für Fließschrift, Fonts ohne Serifen für plakativen Text. Dazu werden Font-Eigenschaften definiert, wie gerade/schräg, normal/fett, unterstrichen usw. Die Repräsentation von Fonts erfolgt normalerweise durch die Definition der Umrisskurven der Buchstaben, für manche Anwendungen auch durch Pixelraster.

Polygon-Listen

Dreidimensionale Objekte werden meist durch Polygonlisten repräsentiert (oft Dreiecke). Eine Menge von Polygonen, die die Oberfläche eines Objektes beschreibt, nennt man *Boundary-Representation* („B-Rep“). Datenstrukturen für B-Reps enthalten neben geometrischer Information auch Attribute (Eigenschaften). Die Geometrie besteht aus Punktlisten, Kantenlisten, Flächenlisten und muss auf Konsistenz und Vollständigkeit überprüft werden.



Das folgende Beispiel zeigt für eine ganz einfache Situation mit 2 Polygonen, wie die Punktliste (Vertex Table), Kantenliste (Edge Table) und die Flächenliste (Surface Table) sich gegenseitig referenzieren. Nur in der Punktliste ist die tatsächliche geometrische Information enthalten, die anderen Listen beschreiben lediglich die Topologie.



VERTEX TABLE

V_1:	x_1, y_1, z_1
V_2:	x_2, y_2, z_2
V_3:	x_3, y_3, z_3
V_4:	x_4, y_4, z_4
V_5:	x_5, y_5, z_5

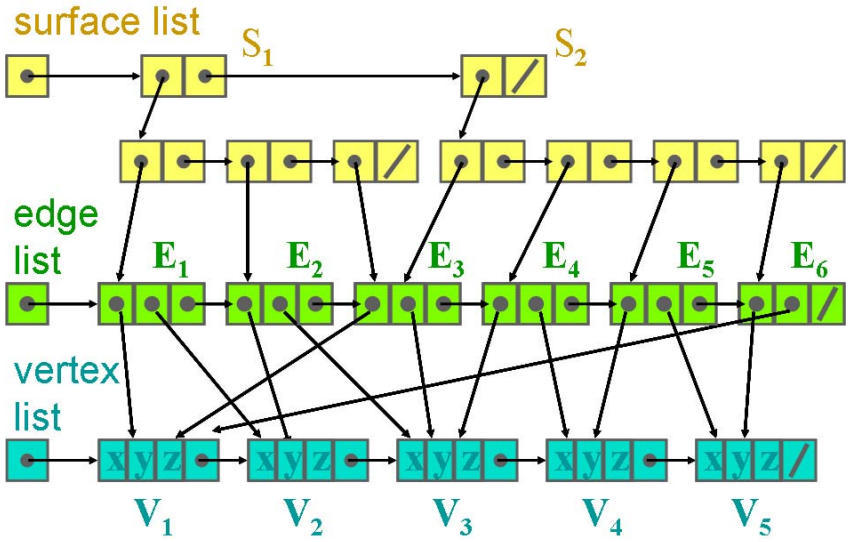
EDGE TABLE

E_1:	V_1, V_2
E_2:	V_2, V_3
E_3:	V_3, V_1
E_4:	V_3, V_4
E_5:	V_4, V_5
E_6:	V_5, V_1

POLYGON-SURFACE TABLE

S_1:	E_1, E_2, E_3
S_2:	E_3, E_4, E_5, E_6

Dieselbe Struktur lässt sich auch mit Zeigerlisten darstellen:



Die Repräsentation jeder einzelnen Polygonfläche inkludiert die Trägerebene Ax + By + Cz + D = 0 und die Eckpunkte V_1 bis V_n. Aus den Ebenenparametern A, B, C, D erhält man sofort den Normalvektor auf die Ebene (A, B, C). Als *Backface* bezeichnet man die Rückseite des Polygons, die also in das Objekt hinein schaut, als *Frontface* die Vorderseite, die also die Außenseite des Objektes mitformt. Mit „hinter dem Polygon“ meint man dann alle Punkte, die vom Backface aus sichtbar sind, „vor dem Polygon“ heißt, dass man von dort auf das Frontface sieht.

Wenn man ein rechtshändiges Koordinatensystem vorausschickt und die Eckpunkte jedes Polygons (von vorne betrachtet) im mathematisch positiven Sinn (also gegen den Uhrzeigersinn) anordnet, dann gilt für einen Punkt (x, y, z)

wenn Ax + By + Cz + D = 0 dann liegt der Punkt **auf** der Ebene
wenn Ax + By + Cz + D < 0 dann liegt der Punkt **hinter** der Ebene
wenn Ax + By + Cz + D > 0 dann liegt der Punkt **vor** der Ebene

Ebenso lässt sich aus drei aufeinanderfolgenden Eckpunkten V_1, V_2, V_3 ein nach außen gerichteter Normalvektor N durch $N = (V_2 - V_1) \times (V_3 - V_1)$ errechnen. Dieses Wissen werden wir später noch brauchen.