Raster Graphics

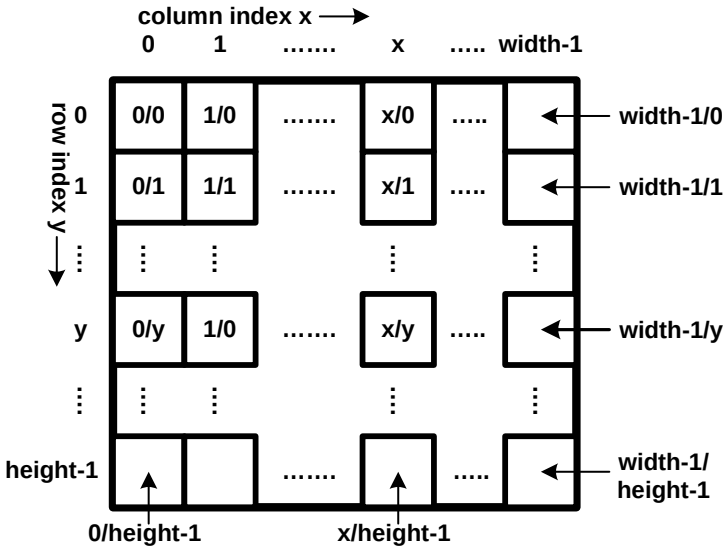
Warum Rastergraphik ?

Bis ca. 1980 verstand man unter Computergraphik nur und ausschließlich Vektorgraphik. Man war begeistert von den Ästhetik der schnellen Liniengraphiken der Vektordisplays, die viel eleganter waren als die dicken horizontalen Zeilen des Fernsehens und es erschien absurd, dass Fernsehen die eleganten Vektorbilder der Computer je ersetzen könnte. Man empfand nicht als Mangel, dass der Vektorbildschirm keine Flächen füllen konnte und dass es keine Mehrfarbigkeit gab.
Andererseits waren Fernsehgeräte billig und weiterverbreitet, Vektordisplays aber teuer und selten.
Die Computerindustrie musste folglich die Fernsehtechnik nutzen, wenn sie einen Massenmarkt erobern wollte. Sie musste eine linear adressierte, digitale Maschine (den Computer) mit einer flächendeckend zeilenadressierten, analogen Maschine (dem Fernseher) verbinden.
So entstand das Bauteil, das wir heute Graphikkarte nennen:
1) Schneller RAM zur Aufnahme der Rastermatrix (=digitaler Bildspeicher)
2) schnelle Adressierung der Matrix synchron zu den analogen H-Sync und V-Sync Signalen (=Videocontroller)
3) schneller DA-Wandler, der die aus der Matrix ausgelesenen Zahlen in ein analoges Helligkeitssignal verwandelt (=DAC).
Die so entstehende Rastergraphik sah abschreckend aus: Flächen mit treppigen Rändern und Bilder der realen Welt zusammengesetzt aus flickernden Klötzchen. Die Verschmelzung von Computer und Fernsehen war zunächst ein Misserfolg. Schnell wurde klar, dass brauchbare Rastergraphiken ca. die doppelte Orts- und Zeitauflösung eines Fernsehers benötigen. Aus diesem Zwang gingen zwei neue Produkte hervor:
der Computermonitor und die Graphikkarte.
Flächenfüllung, Farbfähigkeit und die Erhöhung der Orts- und Zeitauflösung haben der Rastergraphik zum Durchbruch verholfen, obwohl sie weder über dünne Linien noch über Kurven verfügt. Sie kann diese nur durch Treppen simulieren. Sie schleppt enorme Redundanz mit sich herum und ist schwer zu programmieren. Aus diesem Grund steckt hinter fast jeder Rastergraphik eine Vektorgraphik (Ausnahmen: Photo und Video).

Die Raster Matrix

ist eine rechteckige Anordnung ganzer Zahlen (vom Typ `Byte`, `UInt16`, `UInt32` oder `Color`) in `width` Spalten und `height` Zeilen.

```
Spaltenindex = column index: x, wobei 0 <= x < width
Zeilenindex  = row    index: y, wobei 0 <= y < height.
```



Beispiel: `Homunculus` =
heller Mensch auf dunklem Hintergrund
Spaltenzahl = `width = 9`
Zeilenzahl = `height = 10`

```
0 0 0 9 9 9 0 0 0
0 0 0 9 3 9 0 0 0    [4,1] = Mund
0 0 0 9 9 9 0 0 0
0 0 0 0 9 0 0 0 0    [4,3] = Hals
3 4 5 9 9 9 5 4 3    [0,4] und [8,4] = Hände
0 0 0 9 8 9 0 0 0    [4,5] = Nabel
0 0 0 9 9 9 0 0 0
0 0 0 6 0 6 0 0 0
0 0 0 5 0 5 0 0 0
0 0 0 5 0 5 0 0 0    [3,9] und [5,9] = Füße
```

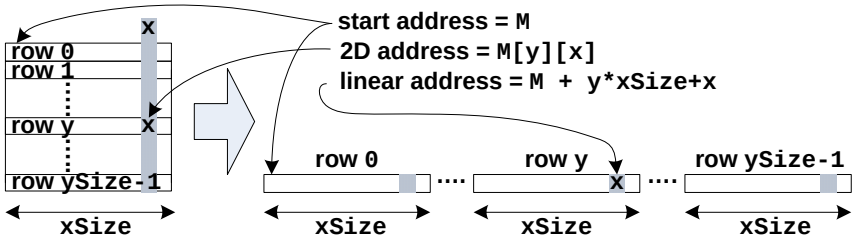
Beispiele für die Definition einer Bildmatrix `M` mit 32-Bit-Farb-Pixeln = ARGB-Pixeln:

```
C++  als Array : int M[height][width];
Java als Array : int[][] M = new int[height][width];
C#   als Array : Color[,] M = new Color[height, width];
C#   als Bitmap: Bitmap M = new Bitmap( width, height, PixelFormat.Format32bppArgb );
```

Verwirrend, aber wichtig: Alle Computersprachen verlangen, dass man bei 2-dimensionalen Raster-Arrays die `y`-Koordinate **vor** der `x`-Koordinate notiert. Begründung: Matrizen werden intern als lineare eindimensionale Arrays in den Speicher abgebildet. Die intuitiv richtige Reihenfolge: `0.` Zeile, `1.` Zeile usw. bis `(height-1)`te Zeile erreicht man nur, wenn der erste Index der Zeilenindex `y` ist. Benutzt man den Spaltenindex `x` zuerst, dann bildet der Rechner intern anders ab: `1`. Spalte, `2`. Spalte bis `(width-1)`te Spalte,
was jeder Intuition widerspricht.
Folge: Will man den Mund des Homunculus schwärzen, dann muss man programmieren:

```
C++  als Array : Homunculus[1][4] = 0;
Java als Array : Homunculus[1][4] = 0;
C#   als Array : Homunculus[1,4] = Color.Black;
C#   als Bitmap: Homunculus.SetPixel( 4, 1, Color.Black );
```

Lineare Adressierung:
Matrizen sind reine Sprachkonstrukte, denn Rechner kennen nur einen linearen Adressraum und speichern alle Matrizen linear ab. Die `[y][x]`-Matrix stellt man sich am besten vor als Schubladenkommode.
Im Hauptspeicher sind die Schubladen herausgezogen und nebeneinander gelegt.



Im Hauptspeicher: Zuerst Zeile `0`, dann Zeile `1` usw. bis Zeile `height-1`. Spricht man ein Pixel `M[y][x]` an, berechnet man in Wahrheit die Addresse `M + y * width + x`. Man bezahlt also den bequemen Zugriff `M[y][x]` mit einer Adressrechnung mit 2 Additionen und einer Multiplikation.

Folge: Bei Millionen Pixeln ist die Matrixadressierung langsam.
Besser: Pointerschreibweise für schnelle Operationen auf Bildern.
Beispiel: Langsamer Code für das löschen einer Matrix `M[height][width]`:

```
for ( y=0; y < height; y++ )
  for ( x=0; x < width; x++ )
    M[y][x] = 0;
```

Beispiel: Schneller Code für das löschen einer Matrix `M[height][width]`:

```
int* pointer = M;
for ( i=0; i < width*height; i++ ) *pointer++ = 0;
```

Beispiel: Sehr schneller Code für das löschen einer Matrix `M[height][width]`:

```
for ( int* pointer = M; pointer < M + width*height; ) *pointer++ = 0;
```

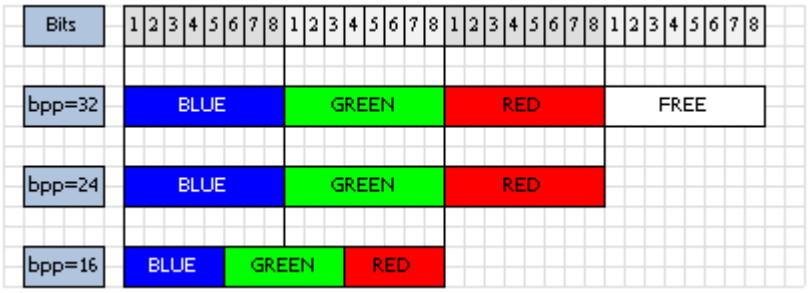
Pixel

= Abkürzung für Picture Element bezeichnet ein Element der Rastermatrix. Eine Matrix beherbergt immer nur Pixel eines Typs, aber es existieren sehr verschiedene Pixeltypen = Pixel-Formate. Beispiele:

1bppIndexed	1 bit per pixel with indexed color. Requires a LUT with 2 colors in it. For binary images.
4bppIndexed	4 bits per pixel, indexed. Requires a LUT with 3x16 palette entries.
8bppIndexed	8 bits per pixel, indexed. Requires a LUT with 3x256 palette entries.
16bppGrayScale	16 bits per pixel. The color information specifies 65536 shades of gray.
16bppRgb555	16 bits per pixel; 5 bits each are used for the red, green, and blue components. The remaining bit is not used.
24bppRgb	24 bits per pixel; 8 bits each are used for the red, green, and blue components.
32bppArgb	32 bits per pixel; 8 bits each are used for the alpha, red, green, and blue components.
32bppRgb	32 bits per pixel; 8 bits each are used for the red, green, and blue components. The remaining 8 bits are not used.
64bppArgb	64 bits per pixel; 16 bits each are used for the alpha, red, green, and blue components.

Falschfarbenbild = Überbegriff über die ersten 3 Formate 1bppIndexed, 4bppIndexed und 8bppIndexed.
Grauwertbild = 16bppGrayScale.
Echtfarbbild = True Color Image = Überbegriff über alle anderen Formate.

Die gebräuchlichsten Pixel-Formate sind:
1bppIndexed → für Binärbilder und s/w-Drucker, weil sehr platzsparend.
32bppRgb → für Farbphotos, weil es zur 32-Bit-Speicherarchitektur und 32-Bit-Adressierung der PCs passt.



3 Memory Layouts eines Pixels:

32bppRgb,

24bppRgb und

16bppRgb555

Bild von Thomas Schedl

Vergleiche Vektorgraphik ↔ Rastergraphik

Allgemeiner Vergleich

	Vektorgraphik	**Rastergraphik**
Hauptdatenstruktur	Polygon: `PointF[] p = new PointF[n];`	Matrix: `Bitmap bmp = new Bitmap( width, height, PixelFormat.Format32bppArgb );`
sonstige Datenstrukturen	Rechteck, Ellipse, Bézierkurve, Spline, Mesh	RLC, Crack Code, MPEG
Fileformate	WMF, PostScript, XAML, PDF, Flash, X	BMP, GIF, JPEG, MPEG, PNG, TIFF, AVI
Speicherplatzbedarf	gering: `n*sizeof(PointF)`	enorm: `width*height*sizeof(Color)`
Fähigkeit für Linien	sehr gut	bei CRTs nur waagrecht, bei Flat Panels waag- und senkrecht
Fähigkeit für Flächen	nur Schraffur	gut, aber mit treppigem Rand
Eignung für Schrift	zu dünn, aber gut für skalierbare Umrisse = ein Font für alle Größen = TrueType	gut, aber mit treppigem Rand, braucht für jede Größe je einen Font
Fähigkeit für Bilder der realen Welt	null, nur Umrisse	gut, siehe TV
Farbfähigkeit der Displays	fast immer einfarbig, maximal 2 Farben möglich	gut: fast immer RGB
Herstellung durch	immer einen Menschen	fast immer eine Maschine: a) Digitalbilder der realen Welt (Kamera) b) gerendert aus Vektorgraphik (Graphikkarte)
Mathematik	Es gelten alle Gesetze der analytischen Geometrie.	neue digitale Geometrie erforderlich
Flickern, Refresh	flickert nur, wenn Polygone zu viele Ecken haben	flickert unabhängig vom Bildinhalt
Verwendung für	Umriss-Zeichnungen: CAD, Comics	Bilder mit farbigen oder grauen Flächen
flüchtige Ausgabe	RA-CRT = Vektordisplay	Zeilen-CRT, Flat Panel Display = Rasterdisplay
dauerhafte Ausgabe	Plotter	Drucker

Vergleich der Operationen scroll, zoom, rot

scroll, zoom, rot von Polygon p0 → p1	scroll, zoom, rot von Bitmap bmp0 → bmp1
alle `x,y` sind reelle Zahlen	alle `x,y` sind ganze Zahlen
stufenlos	nur ganzzahlige Pixelschritte
immer hochpräzise	fast immer mit Rundungsfehlern
Vorwärtstransformation von `p0` nach `p1`: transformiere jeden Vertex aus `p0` nach `p1`	Rückwärtstransformation von `bmp1` nach `bmp0`: suche für jedes Zielpixel von `bmp1` ein Pixel aus `bmp0`
es gibt keinen Bildrand	großes Problem: Verluste durch Kappung am Bildrand
vollständig reversibel	Operationen fast nie rückgängig zu machen
Operationen sind kaskadierbar	Operationen müssen vom Originalbild `bmp0` ausgehen
`p0` darf durch `p1` überschrieben werden	`bmp0` wird meistens noch gebraucht, nicht durch `bmp1` überschreiben!

Codevergleich von Scroll (Verschiebungen: float dx, float dy)

Vektor-Scroll von Polygon p0 → p1

```
for alle 0 <= i < n
{ p1[i].x = p0[i].x + dx;
  p1[i].y = p0[i].y + dy;
}
```

Raster-Scroll von Bitmap bmp0 → bmp1

```
int idx = Convert.ToInt32( dx ); //rounding
int idy = Convert.ToInt32( dy ); //rounding
for ( int y1=0; y1 < bmp1.Height; y1++ )
{ int y0 = y1 - idy; //backward
  if ( y0 < 0 || y0 >= bmp0.Height ) continue; //outside
  for ( int x1=0; x1 < bmp1.Width; x1++ )
  { int x0 = x1 - idx; //backward
    if ( x0 < 0 || x0 >= bmp0.Width ) continue; //outside
    Color color = bmp0.GetPixel( x0, y0 );
    bmp1.SetPixel( x1, y1, color );
  }
}
```

Codevergleich von Zoom (Vergrößerungen/Verkleinerungen: float zoomx, float zoomy)
Steckungs-/Stauchungszentrum im Ursprung des Koordinatensystems

Vektor-Zoom von Polygon p0 → p1

```
//Streckungszentrum (0/0)
for alle 0 <= i < n
{ p1[i].x = p0[i].x * zoomx;
  p1[i].y = p0[i].y * zoomy;
}
```

Raster-Zoom von Bitmap bmp0 → bmp1

```
for ( int y1=0; y1 < bmp1.Height; y1++ )
{ int y0 = Convert.ToInt32( y1 / zoomy ); //backward
  if ( y0 < 0 || y0 >= bmp0.Height ) continue; //outside
  for ( int x1=0; x1 < bmp1.Width; x1++ )
  { int x0 = Convert.ToInt32( x1 / zoomx ); //backward
    if ( x0 < 0 || x0 >= bmp0.Width ) continue; //outside
    Color color = bmp0.GetPixel( x0, y0 );
    bmp1.SetPixel( x1, y1, color );
  }
}
```

Codevergleich von Rotation (Drehung um α Grad um den Ursprung im Uhrzeigersinn)

Vektor-Rotation um α von Polygon p0 → p1

```
//Drehzentrum (0/0)
double arcus  = alpha * 2 * Math.PI / 360;
float sinus   = (float)Math.Sin( arcus );
float cosinus = (float)Math.Cos( arcus );
für alle Ecken 0 <= i < n
{ p1[i].x = p0[i].x * cosinus
          - p0[i].y * sinus;
  p1[i].y = p0[i].x * sinus
          + p0[i].y * cosinus;
}
```

Raster-Rotation um α von Bitmap bmp0 → bmp1

```
double arcus  = alpha * 2 * Math.PI / 360;
float sinus   = (float)Math.Sin( arcus );
float cosinus = (float)Math.Cos( arcus );
for ( int y1=0; y1 < bmp1.Height; y1++ )
{ float y1_sinus   = y1 * sinus;
  float y1_cosinus = y1 * cosinus;
  for ( int x1=0; x1 < bmp1.Width; x1++ )
  { int x0 = Convert.ToInt32( x1 * cosinus + y1_sinus );
    if ( x0 < 0 || x0 >= bmp0.Width ) continue;
    int y0 = Convert.ToInt32( -x1 * sinus + y1_cosinus );
    if ( y0 < 0 || y0 >= bmp0.Height ) continue;
    Color color = bmp0.GetPixel( x0, y0 );
    bmp1.SetPixel( x1, y1, color );
  }
}
```

Anmerkung zur Raster-Rotation:
.NET besitzt in der `Graphics`-Klasse eine sehr elegante Variante der Methode `DrawImage(...)`. Diese Variante akzeptiert als Parameter drei Eckpunkte `p[0]`, `p[1]` und `p[2]` welche ein Parallelogramm definieren. `DrawImage(...)` vollzieht eine Drehung und Scherung des Rasterbildes so, dass die Eckpunkte des Bildes in die Ecken des Parallelogramms zu liegen kommen. Das bedeutet, dass man nur das Dreieck `PointF[] p = new PointF[3];` per Vektorrotation dreht, das gedrehte Dreieck an `DrawImage(myBitmap, p)` übergibt und die gesamte Rasterdrehung automatisch folgt.
Siehe: **http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/...asp**.
Sie finden eine Drehanimation, wo die Ecken des Dreiecks entlang der Fensterränder wandern unter:
../../C_IPCis/C1_Bitmap/CIPCisBitmap_d.htm#a9.