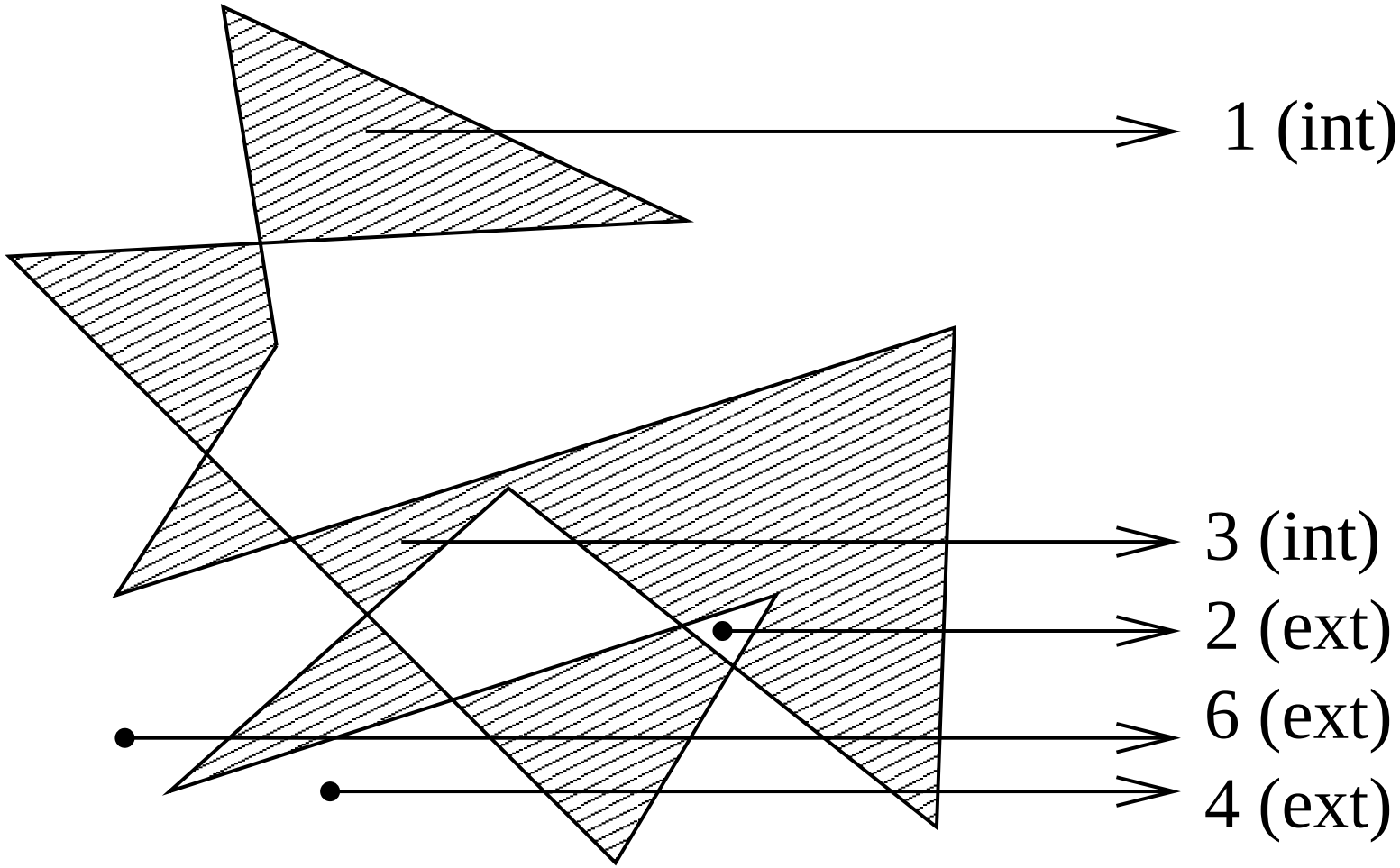
Füllen von Polygonen

Bestimmung des Inneren eines geschlossenen Polygonzuges (der sich eventuell mehrfach selbst schneidet):

Odd-Parity-Regel: Ein Punkt liegt genau dann im inneren des Polygons, wenn eine beliebige, von ihm in eine Richtung ausgehende unendliche Linie, die keine Ecke des Polygons berührt, eine ungerade Anzahl von Schnittpunkten mit dem Polygon aufweist.

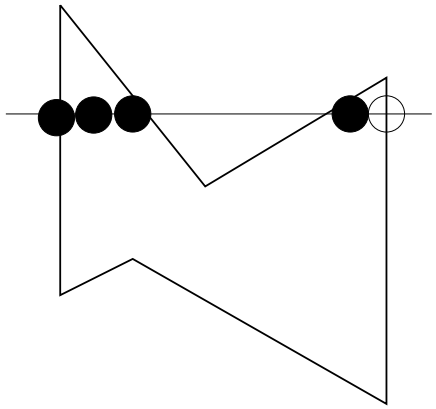
In Java 2D wird ein `Shape s` mittels `g2d.fill(s);` ausgefüllt.

Odd-Parity-Regel



Füllen von Polygonen

Soll ein Polygon mit einer Farbe ausgefüllt werden, so wird für jeden y-Wert eine Scan-Linie betrachtet und die zu zeichnenden Rasterpunkte auf dieser Geraden bestimmt.

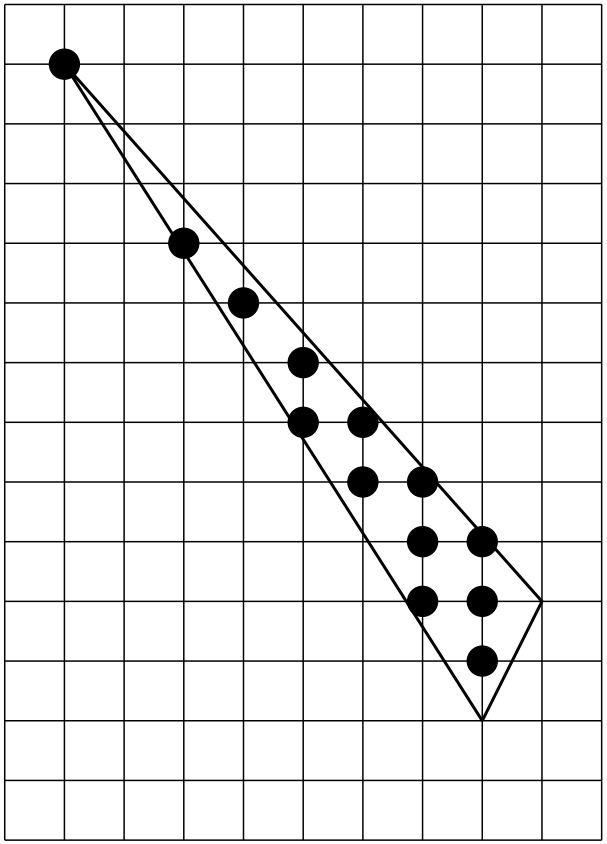


1. Bestimme für die Scan-Linie alle Schnittpunkte mit Polygonkanten (ohne Polygonecken!).
2. Sortiere die Schnittpunkte aufsteigend nach den x-Koordinaten.

Füllen von Polygonen

3. Wende die Odd-Parity-Regel an, um zu bestimmen, welche Punkte innerhalb des Polygons liegen.
   - Starte dabei am linken Rand des Rahmens (mit gerader Parität).
   - Springe zum ersten Schnittpunkt der Scan-Linie mit dem Polygon, (setze die Parität auf ungerade und) zeichne alle Pixel bis zum nächsten Schnittpunkt.
   - Springe bis zum nächsten Schnittpunkt und zeichne wieder alle Pixel bis zum folgenden Schnittpunkt usw.

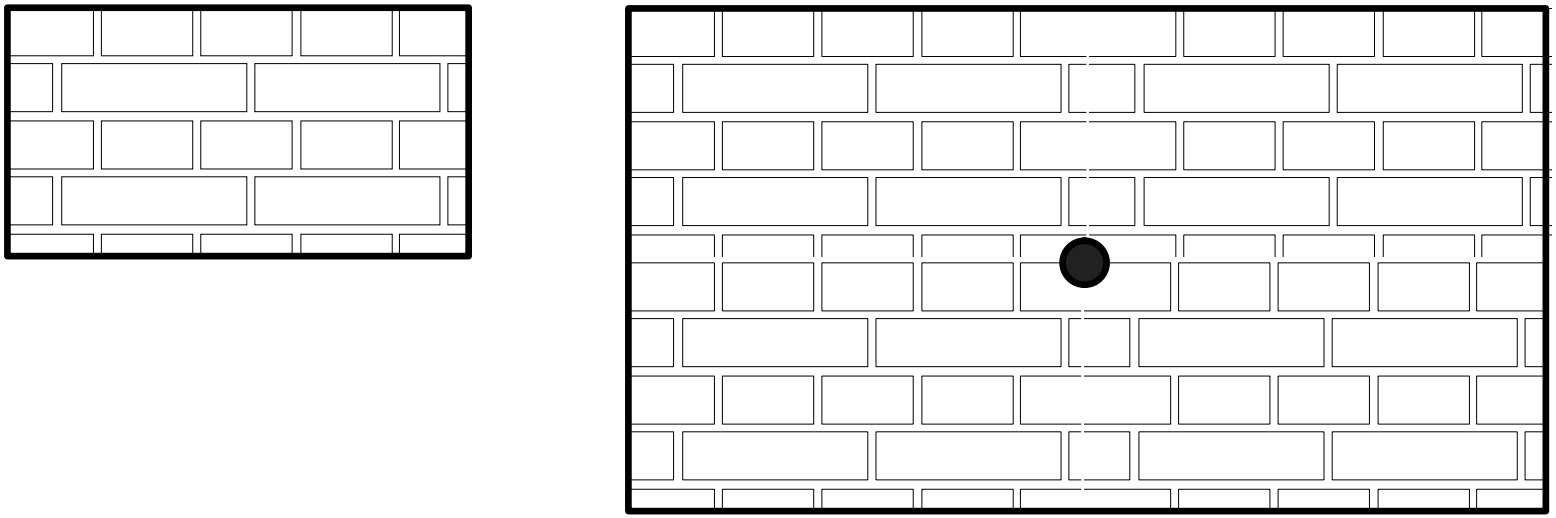
Aliasing-Effekt beim Füllen (dünner) Polygone:



Füllen von Kurvenzügen

- Das Füllen von Kreisen, Ellipsen, Kreis- oder Ellipsenbögen erfolgt nach demselben Prinzip.
- Dabei sollten wiederum die Symmetrien dieser geometrischen Figuren ausgenutzt werden.
- Das Füllen von beliebigen Kurvenzügen erfolgt ebenfalls nach demselben Prinzip.
- Werden beim Zeichnen der Ränder von gefüllten Flächen Antialiasing-Techniken angewendet, sind die Grenzen der Fläche geometrisch kaum noch eindeutig zu bestimmen.

Texturen



Einnmalige und mehrfache Verwendung einer Textur zum Füllen einer Fläche

Bei mehrfacher Verwendung muss ein Ankerpunkt festgelegt werden.

Buffered Images

Ein `BufferedImage` ist ein Bild im Speicher,

- auf das gezeichnet werden kann und
- das selbst auf ein `Image` gezeichnet werden kann.

```
BufferedImage bi =
    new BufferedImage(width,height,
            BufferedImage.TYPE_INT_RGB);

Graphics2D g2dbi = bi.createGraphics();
```

Buffered Images

```
g2dbi.draw(...);

g2dbi.fill(...)

g2d.drawImage(bi,xpos,ypos,null);
```

Mit `drawImage` kann auf das Bildschirmfenster oder ein `BufferedImage` gezeichnet werden.

Double Buffering

Prinzip des Double Buffering: Verwende

- ein `BufferedImage theBackground`, das den statischen (unveränderlichen) Hintergrund enthält
- ein `BufferedImage theImage`, auf das nach jedem Bewegungsschritt zuerst `theBackground` und danach das bzw. die sich bewegenden Objekte gezeichnet werden
- das eigentliche Graphics2D-Objekt (wie bisher `g2d`), auf das jeweils `theImage` gezeichnet wird.

Double Buffering

Fenster, in in das gezeichnet werden soll:

Attribute:

```
public BufferedImage bi;
public Graphics2D g2dbi;
```

Methoden:

```
public void paint(Graphics g)
{
  update(g);
}

public void update(Graphics g)
{
  g2d = (Graphics2D) g;
  g2d.drawImage(bi,0,0,null);
}
```

Double Buffering

Realisierung dieser Methoden in der Klasse `BufferedImageDrawer`. Erforderliche Schritte:

- Eigentliche Berechnungen in einer Klasse, die die Java-Klasse `TimerTask` erweitert.
- Die `run`-Methode dieser Klasse enthält das, was bisher in der FOR-Schleife in der `paint`-Methode stand.
- Es wird nicht mit dem `Graphics2D`-Objekt der `paint`-Methode des Fensters gezeichnet, sondern mit dem `Graphics2D`-Objekt des entsprechenden `BufferedImage bi`.

Double Buffering

- Anstatt den zu zeichnenden Bereich, d.h. das `BufferedImage bi`, jedes Mal weiß zu überschreiben, wird ein weiteres `BufferedImage` als Hintergrund verwendet, das jedes Mal auf `bi` gezeichnet wird.
- Am Ende der `run`-Methode muss die `repaint`-Methode des `BufferedImageDrawer` aufgerufen werden.

Double Buffering

Initialisierungen finden in der `main`-Methode und im Konstruktor der speziellen Klasse statt.

Die `run`-Methode, die die Bildsequenz berechnet und das Zeichnen veranlasst, wird mittels

```
Timer t = new Timer();
t.scheduleAtFixedRate(dbce,0,delay);
```

wiederholt aufgerufen.

(vgl. `DoubleBufferingClockExample.java`)

Laden von Bildern

Mittels

```
Image theImage;
theImage = new javax.swing.ImageIcon(
                "filename.jpg").getImage();
```

kann ein Bild geladen werden, das dann beispielsweise mittels

```
g2d.drawImage(theImage,x,y,null);
```

an der Position (x, y) gezeichnet werden kann.

Dabei ist `g2d` das Graphics2D-Objekt der eigentlichen Grafik oder eines `BufferedImages`.

Speichern von Bildern

- Erzeuge ein `BufferedImage theImage`.
- Erzeuge das entsprechende Graphics2D-Objekt:
  `Graphics2D g2dImage = theImage.createGraphics();`
- Zeichne das Bild (mit Hilfe von `g2dImage`).
- Erzeuge einen `FileOutputStream fos` für die Datei, in der das Bild gespeichert werden soll.
- Speichere das Bild unter Verwendung eines `JPEGImageEncoder`.

(vgl. `ImageSavingExample.java`)

Speichern von Bildern

```
try
{
  FileOutputStream fos = new
        FileOutputStream("test.jpg");
  JPEGImageEncoder jie =
       JPEGCodec.createJPEGEncoder(fos);
  jie.encode(theImage);
}
catch (Exception e)
{
  System.out.println(e);
}
```

Texturen in Java 2D

Einmaliges Auftragen einer Textur `theImage` in Form eines JPEG-Bildes auf einen `Shape s`:

```
Shape clipShape = g2d.getClip();
g2d.setClip(s);
g2d.drawImage(theImage,50,50,null);
g2d.setClip(clipShape);
```

Texturen in Java 2D

Füllen eines `Shape s` mit einer Textur `theImage` in Form eines JPEG-Bildes:

- Das Muster muss in einem `BufferedImage buffImage` vorliegen.
- Erzeuge

```
TexturePaint tp =
new TexturePaint(buffImage,
  new Rectangle(0,0,
                buffImage.getWidth(),
                buffImage.getHeight()));
g2d.setPaint(tp);
g2d.fill(s);
```

Text

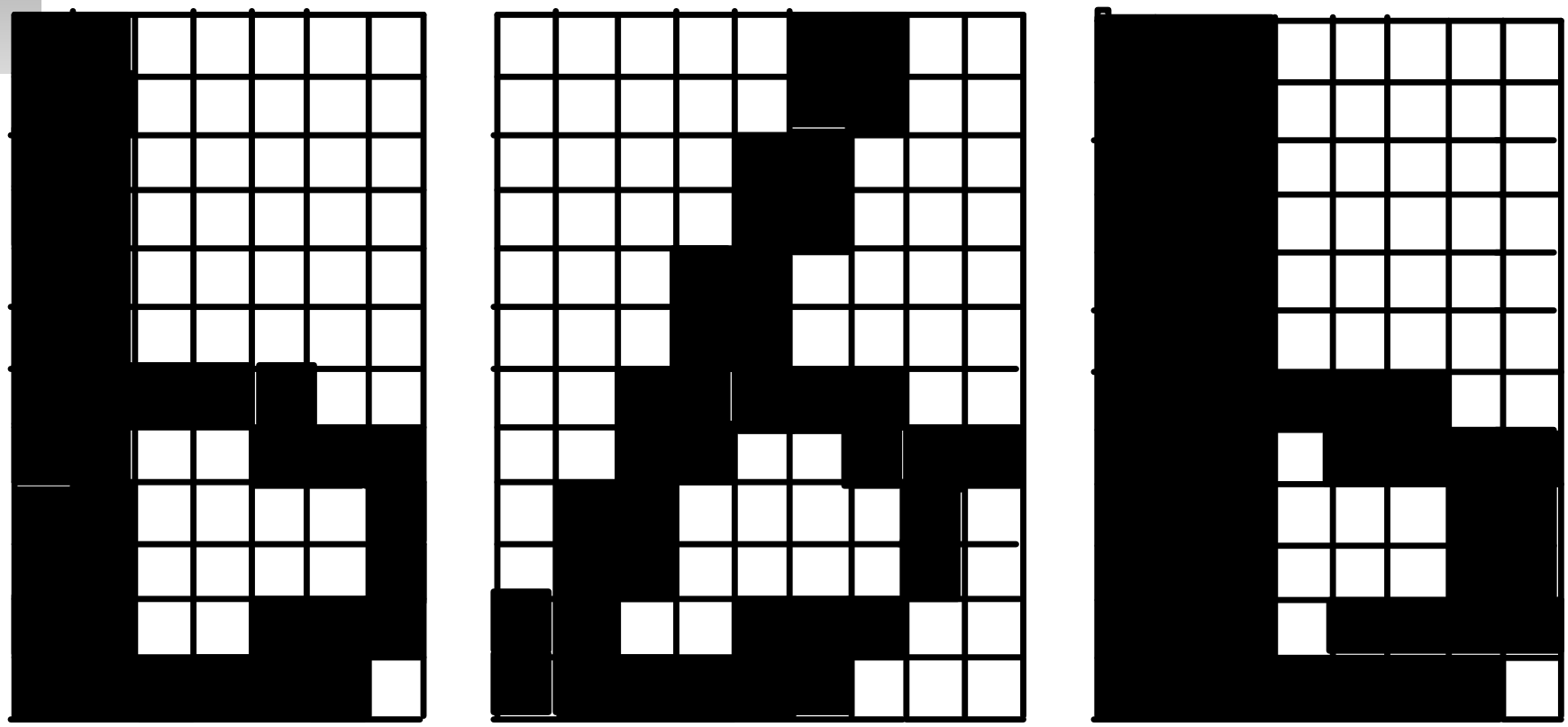
Zeichen werden üblicherweise in zwei Formen gespeichert

- als Bitmap (Pixelmatrix) oder
- als Umriss in Form eines Polygons oder einer Kurve.

Bei der Bitmapspeicherung muss für jedes Zeichen in jeder Größe und in jedem Stil (normal, bold face (fett), italic (kursiv), bold face italic) eine eigene Bitmap definiert werden.

Bei einer Polygon-/Kurvendarstellung können zumindest die unterschiedlichen Größen direkt berechnet und mittels Scan-Conversion dargestellt werden.

Einfache, nicht sehr schöne Techniken zur Erzeugung von Italic- und Bold-Face-Zeichen aus Bitmaps:



normal *kursiv* **fett**

Text in Java 2D

Zeichnen eines Textes:

```
g2d.drawString("text",posx,posy);
```

Auswahl des Fonts:

```
Font f = new Font("type",Font.STYLE,size);
```

verfügbare Fonts (für type):

```
Font[] fl =
GraphicsEnvironment.
  getLocalGraphicsEnvironment(
                        ).getAllFonts();
  for (int i=0; i<fl.length; i++)
  {
    System.out.println(fl[i].getName());
  }
```

Werte für `STYLE`

- `PLAIN` (normal)
- `ITALIC` (kursiv)
- `BOLD` (fett)
- `ITALIC | BOLD` (kursiv und fett)

`size` gibt die Größe des Zeichensatzes (in der Maßeinheit pt) an.

Nach dem Aufruf der Methode `g2d.setFont(f)` verwendet `drawString` den Font `f`.

Modifizierte Fonts

Anwendung einer Transformation `affTrans` auf den gesamten Font:

```
Font transformedFont =
                f.deriveFont(affTrans);
```

Transformation einzelner Buchstaben eines Strings `s`:

```
FontRenderContext frc =
                g2d.getFontRenderContext();
GlyphVector gv =
                f.createGlyphVector(frc,s);
```

Modifizierte Buchstaben

```
Point2D p = gv.getGlyphPosition(i);

Shape glyph = gv.getGlyphOutline(i);

Shape transGlyph =
        at.createTransformedShape(glyph);

g2d.fill(transfGlyph);
```

(vgl. `TextExample.java`)

Grauwert- und Farbdarstellung

Intensitätsstufen bei Grau- und Farbwerten:

Lassen sich Pixel zum Beispiel auf einem Bildschirm nicht nur setzen oder nicht setzen, sondern können mit einer wählbaren Intensität erscheinen, steht i.A. eine endliche Menge von Intensitätsstufen zur Verfügung.

Das menschliche Helligkeitsempfinden ist im wesentlichen relativ orientiert:

Eine 60-Watt-Lampe erscheint im Verhältnis zu einer 20-Watt-Lampe wesentlich heller als eine 100-Watt- zu einer 60-Watt-Lampe.

Grauwert- und Farbdarstellung

Die Intensitätsstufen bei einer Grauwert- oder Farbdarstellung sollten daher nicht linear, sondern logarithmisch skaliert sein.

Ausgehend von der geringsten Intensität (schwarz bei Grauwerten) I_0 sollten die tatsächlichen Helligkeitswerte in der Form

$$I_0 = I_0,\ I_1 = rI_0,\ I_2 = rI_1 = r^2 I_0, \ldots$$

bis zu einer maximalen Intensität $I_n = r^n I_0$ gewählt werden.

Grauwert- und Farbdarstellung

Der Mensch kann i.A. benachbarte Grauwerte nur unterscheiden, wenn $r > 1.01$.

Geht man von einer maximalen Intensität $I_n = 1$ und einer vom Ausgabegerät abhängigen minimalen Intensität I_0 aus, folgt aus

$$1.01^n I_0 \leq 1,$$

dass höchstens

$$n \leq \frac{\ln\left(\frac{1}{I_0}\right)}{\ln(1.01)}$$

Graustufen sinnvoll sind.

Grauwert- und Farbdarstellung

Medium	I_0 (ca.)	max. Anz. Graustufen
Bildschirm	0.005-0.025	372-533
Zeitung	0.1	232
Foto	0.01	464
Dia	0.001	695

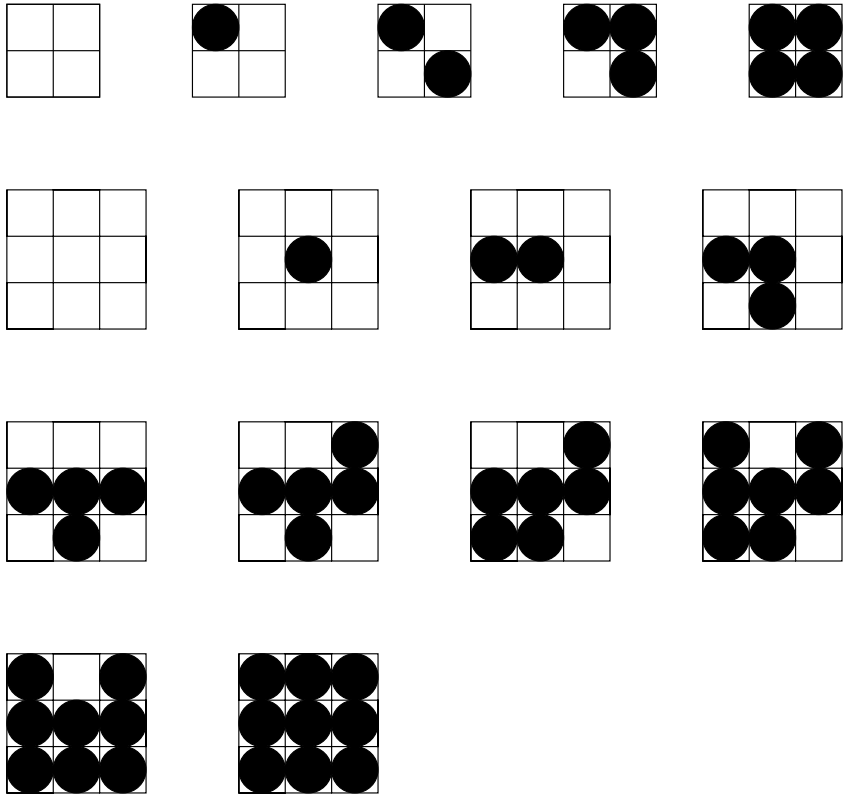
Soll ein Bild mit mehreren Intensitätsstufen auf einem Ausgabemedium dargestellt werden, das nur über binäre Pixelwerte verfügt, kann unter Verringerung der Auflösung eine Annäherung der Intensitätsstufen vorgenommen werden.

Halbtonverfahren

Fasst man 2×2-Pixel zu einem gröberen Pixel zusammen, lassen sich fünf Intensitätsstufen darstellen, bei 3×3-Pixeln 10, bei $n \times n$-Pixeln $n^2 + 1$.

- Die Vergröberung der Auflösung darf nur soweit vorgenommen werden, dass das Auge die Rasterung aus entsprechender Sichtentfernung nicht wahrnimmt.
- Die in dem $n \times n$ großen groben Pixel zu platzierenden Einzelpixel sollten möglichst benachbart gewählt werden und nicht auf einer Geraden liegen.

Halbtonverfahren



Halbtonverfahren

Repräsentation mittels Dither-Matrizen:

$$D_2 = \begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix}, \qquad D_3 = \begin{pmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{pmatrix}$$

Halbtonverfahren

Auch bei nicht-binären Intensitätsstufen lässt sich das Halbtonverfahren anwenden. Beispielsweise können bei einer Zusammenfassung von 2×2 Pixeln mit jeweils vier Intensitätsstufen insgesamt 13 Intensitätsstufen repräsentiert werden:

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix},$$

$$\begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}, \quad \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \quad \begin{pmatrix} 2 & 2 \\ 1 & 2 \end{pmatrix}, \quad \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}, \quad \begin{pmatrix} 3 & 2 \\ 2 & 2 \end{pmatrix},$$

$$\begin{pmatrix} 3 & 2 \\ 2 & 3 \end{pmatrix}, \quad \begin{pmatrix} 3 & 3 \\ 2 & 3 \end{pmatrix}, \quad \begin{pmatrix} 3 & 3 \\ 3 & 3 \end{pmatrix}$$

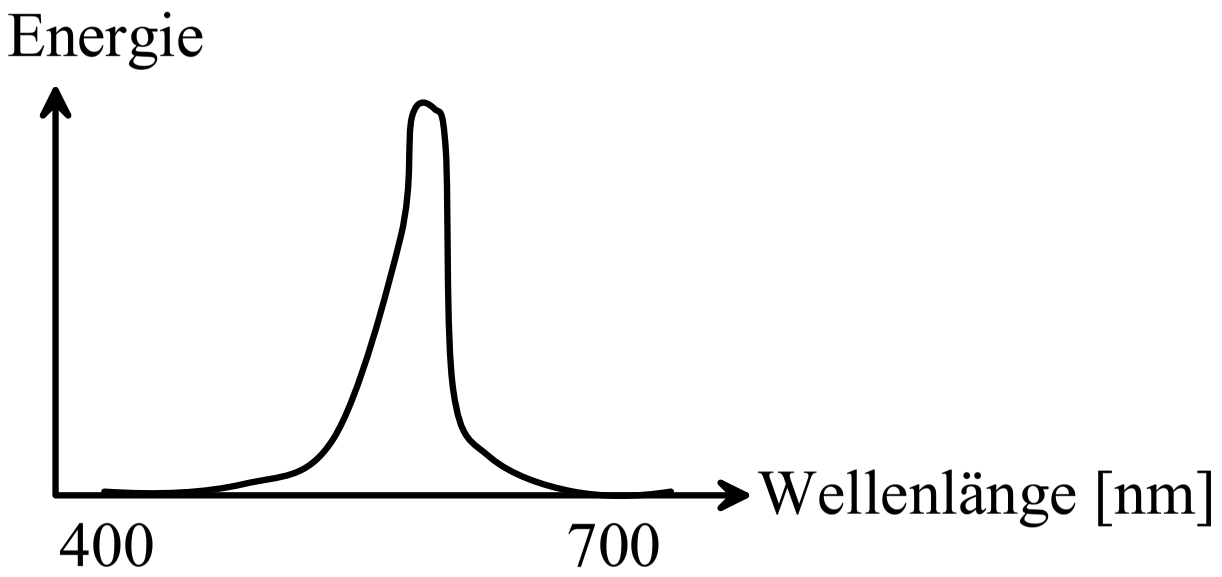
Farbmodelle

- theoretische Beschreibung einer Farbe: Intensitäten im Frequenzspektrum
- wesentliche Charakteristika einer Farbe:
  - Farbton (hue): dominante Wellenlänge
  - Sättigung/Farbreinheit (saturation): Breite/Streuung der Werte im Frequenzspektrum
  - Intensität (lightness/brightness): ähnlich wie bei Graustufen

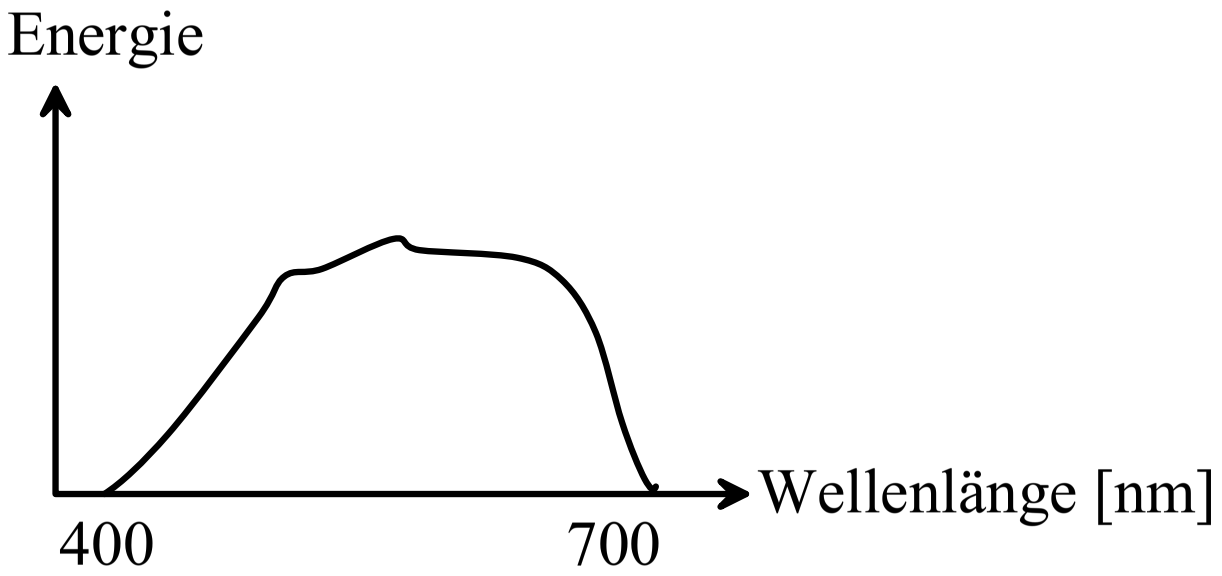
Farbmodelle

sichtbarer Bereich: ca. 400nm (violett) bis 700nm (rot)

hohe Sättigung bei geringer Streuung der Frequenzen:



geringe Sättigung bei großer Streuung der Frequenzen:



Additive/subtraktive Farbmischung

- additive Überlagerung von Licht:
  - schwarzer (dunkler) Hintergrund
  - rot+grün+blau ergibt weiß
  - Beispiel: Monitor
- subtraktive Überlagerung von Farben:
  - weißer (heller) Hintergrund
  - rot+grün+blau ergibt dunkelbraun/schwarz
  - Beispiel: Drucker

Das RGB-Modell

Für Monitore wird üblicherweise das **RGB-Modell** verwendet.

Jede Farbe wird additiv aus den drei Grundfarben Rot, Grün und Blau zusammengesetzt:

$$\text{colour} = r \cdot R + g \cdot G + b \cdot B$$

mit $r, g, b \in [0, 1]$.

Ist 0 die Minimal- und 1 die Maximalintensität, entspricht der Intensitätsvektor $(0, 0, 0)$ der Farbe Schwarz, $(1, 1, 1)$ Weiß, (x, x, x) einem Grauwert, $(1, 0, 0)$ Rot, $(0, 1, 0)$ Grün, $(0, 0, 1)$ Blau.

Das RGB-Modell

Bei Computermonitoren verwendet man üblicherweise für jede der drei Farben 256 Intensitätsstufen zwischen 0 und 255.
In Java 2D kann eine Farbe daher mit dem Konstruktor

- `new Color(float r, float g, float b)` (`r,g,b`$\in [0,1]$) oder
- `new Color(int r, int g, int b)` (`r,g,b`$\in \{0,1,\ldots,255\}$)

angegeben werden.

Nicht jede Farbe lässt sich mit dem RGB-Modell darstellen.

Das CIEXYZ-Modell

Daher wurde von der *Commission Internationale de l'Éclairage* (CIE) ein Model mit drei künstlichen Farben X, Y und Z eingeführt, mit dem alle Farben additiv darstellbar sind:

$$\text{colour} = x \cdot X + y \cdot Y + z \cdot Z$$

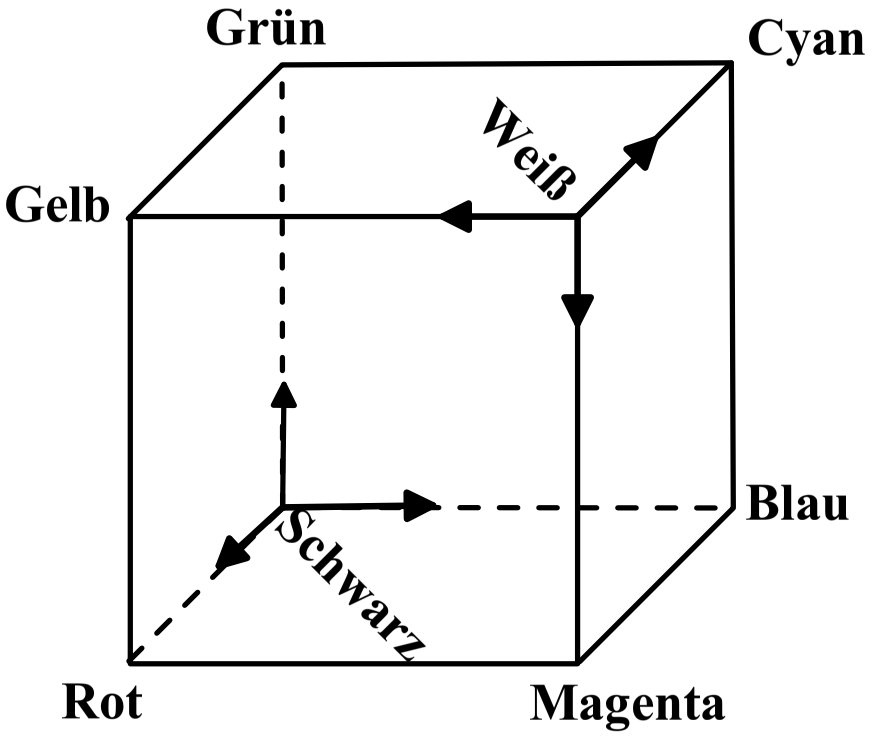
mit $x, y, z \in [0, 1]$.

Aufgrund der künstlichen Grundfarben ist das CIEXYZ-Modell sehr unhandlich und wird kaum verwendet.

Das CMY-Modell

Das subtraktive CMY-Modell ist das komplementäre Modell zum RGB-Modell und wird für Drucker und Plotter verwendet.

Die Grundfarben sind Cyan, Magenta und Gelb.



$$\begin{pmatrix} C \\ M \\ Y \end{pmatrix} = 1 - \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Das CMYK-Modell

Das CMYK-Modell benutzt als vierte Farbe Schwarz (K: key).

Anwendung: Vierfarbendruck bei Druckpressen zur besseren Darstellung von Schwarz

CMY $\rightarrow$ CMYK

$$\begin{aligned} K &:= \min\{C, M, Y\} \\ C &:= C - K \\ M &:= M - K \\ Y &:= Y - K \end{aligned}$$

$\rightarrow$ Mindestens einer der vier Werte C, Y, M, K ist 0.

Das YIQ-Modell

Beim YIQ-Modell werden nicht drei Primärfarben wie beim RGB- oder beim CMY-Modell verwendet, sondern die Kenngrößen

Luminanz (Helligkeit) Y und

Chrominanz (Farbart) I, Q

festgelegt.

Verwendung in der amerikanischen NTSC-Fernsehnorm.

Farbe $\rightarrow$ Schwarz/Weiß: Verwende nur den Y-Wert

Das YIQ-Modell

Es gilt: $$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Die Vorteile des YIQ-Modells bestehen

- bei der Umrechnung von Farb- in Grauwertintensität, die allein durch den Y-Wert gegeben ist, und
- bei der Darstellung auf verschiedenen Monitoren. RGB-Werte können auf verschiedenen Monitoren unterschiedlich aussehen. Beim YIQ-Modell muss nur der Y-Wert angepasst werden.

Das HSV-Modell

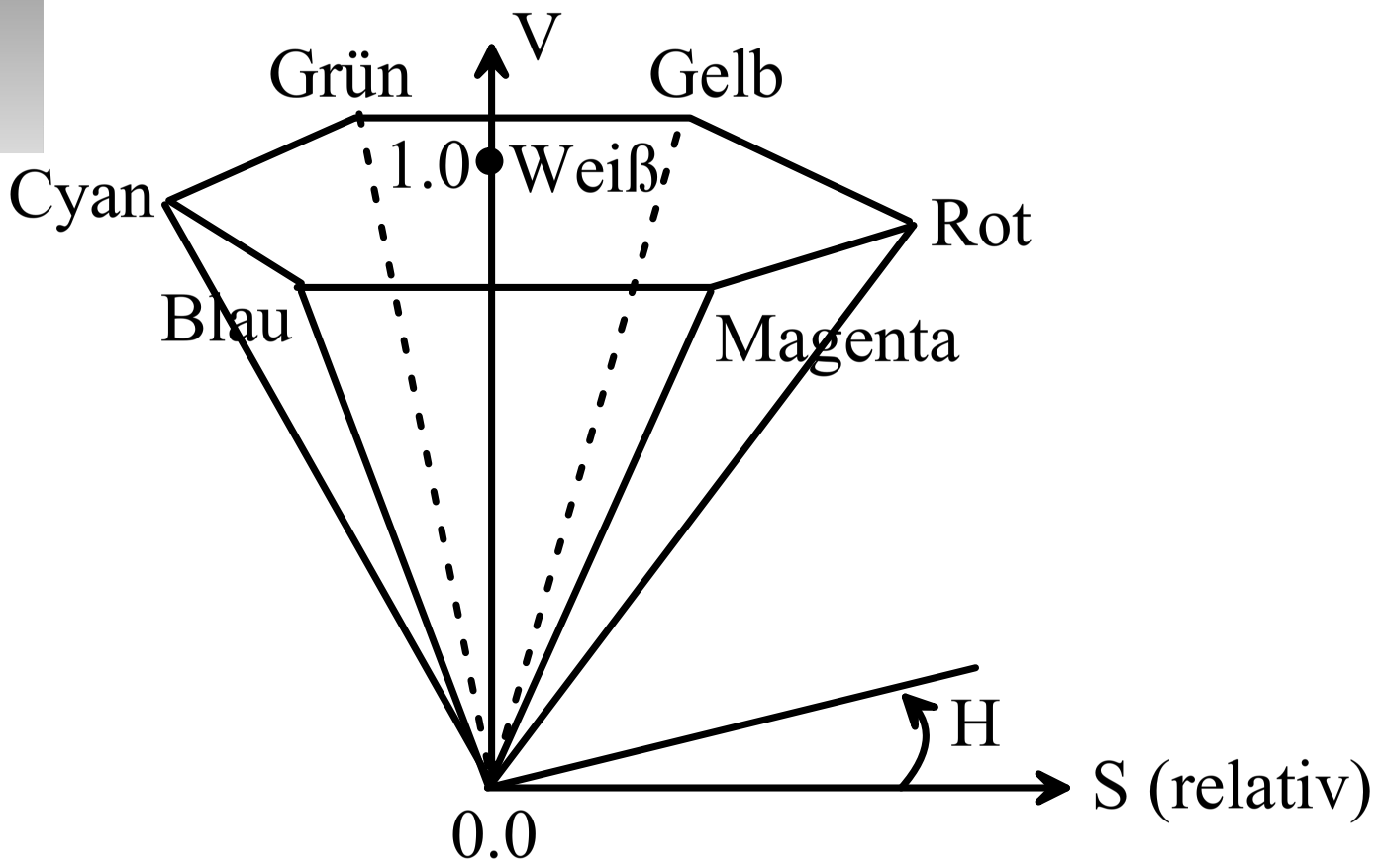
Parameter:

- Farbton (hue)
- Sättigung (saturation)
- Helligkeit/Intensität (value)

Modell in Form einer auf dem Kopf stehenden Pyramide:

- Pyramidenspitze: Schwarz
- Der Farbton H wird durch den Winkel spezifiert.
- Sättigung (S): 0 in der Mitte, 1 auf der Außenfläche
- Die Helligkeit V wird auf der mittleren Achse aufgetragen.

Das HSV-Modell



Das HLS-Modell

Ähnliches Prinzip wie das HSV-Modell

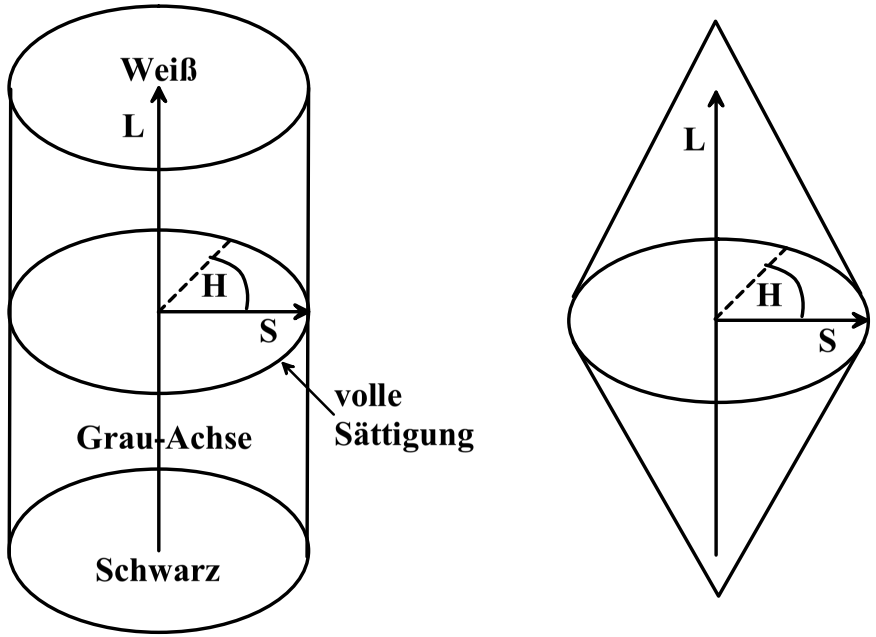
hue: Farbton: Winkel zwischen $0°$ und $360°$

lightness: Helligkeit: Wert zwischen 0 und 1

saturation: Sättigung: Abstand einer Farbe zum Mittelpunkt (Grauwert) zwischen 0 und 1

Grundlage: Farbkreis mit Rot ($0°$), Gelb ($60°$), Grün ($120°$), Blau ($240°$) und Purpur ($300°$)

Das HLS-Modell



Eine Variante des HLS-Systems verwendet anstatt des Zylinders einen Doppelkegel, da auf der oberen und unteren Fläche Sättigung keinen Sinn macht.

Umrechnung von RGB nach HLS

```
max = Max(R,G,B);
min = Min(R,G,B);
L = (max+min)/2;
if (min==max)
{
  S = 0;  //H beliebig
  return;
}
mm = max-min;
if (L<=0.5) S = mm/(max+min); else S = mm/(2-max-min);
r = (R-min)/mm; g = (G-min)/mm; b = (B-min)/mm;
if       (R==max) H = g-b;
else {if (G==max) H = 2+b-r;
else               H = 4+r-g;}
if (H<0) H = H+6;
H = H*60;
```

Wahrnehmungsorientierte Modelle

Das HSV- und das HLS-Modell werden auch als **wahrnehmungsorientierte Farbmodelle** bezeichnet, da sie der intuitiven Farbwahrnehmung relativ nahe kommen.

Die Einstellung einer Farbe oder die Interpretation einer Wertekombination ist bei Farbmodellen wie RGB oder CMY sehr schwierig, zumindest wenn alle drei Farbanteile zusammenwirken.

weiteres wahrnehmungsorientiertes Farbmodell: CNS

Das CNS-Modell

wie HSV und HLS Verwendung von Farbton, Sättigung und Helligkeit

umgangssprachliche Angabe der Farbe:

Farbe: purple, red, orange, brown, yellow, green, blue (jeweils mit weiterer Unterteilung, etwa: yellowish green, green-yellow, greenish yellow)

Helligkeit: very dark, dark, medium, light, very light

Sättigung: greyish, moderate, strong, vivid

Schränkt die Darstellung auf 560 Kombinationen ein. Dafür sehr intuitiv.

Java 2D: Farbmodelle

Java 2D unterstützt die folgenden Farbmodelle (`ColorSpace`-Typen):

CIEXYZ: `TYPE_XYZ`

RGB: `TYPE_RGB`

Grauwertbilder: `TYPE_GRAY`

HSV: `TYPE_HSV`

CMY: `TYPE_CMY`

CMYK: `TYPE_CMYK`

Java 2D: RGB-Farbhandhabung

Mittels `new Color(...)` lassen sich RGB-Farben erzeugen.

Argumente des Konstruktors sind:

- Drei `float`-Werte zwischen 0 und 1, die jeweils die Anteile von Rot, Grün bzw. Blau spezifizieren.
- Drei `int`-Werte zwischen 0 und 255, die jeweils den (Integer-)Wert von Rot, Grün bzw. Blau spezifizieren.
- Ein Integerwert, der binär interpretiert wird. Der R-, G- und B-Wert werden als Byte (Zahl zwischen 0 und 255) angegeben. Diese drei Byte ergeben den Integerwert.

Java 2D: RGB-Farbhandhabung

Mit `g2d.setPaint(col)` wird die Zeichenfarbe auf die Farbe `col` gesetzt.

Mit

```
GradientPaint gradPaint =
    new GradientPaint(x0,y0,colour0,
                      x1,y1,colour1, repeat);
```

lassen sich Farbverläufe erzeugen.

- Der Farbverlauf wird durch den Vektor von `(x0,y0)` nach `(x1,y1)` gekennzeichnet.
- Im Punkt `(x0,y0)` wird die Farbe `colour0`, im Punkt `(x1,y1)` die Farbe `colour1` gezeichnet.

Java 2D: RGB-Farbhandhabung

- Entlang dieses Vektors werden die RGB-Werte der Farbe durch entsprechende Konvexkombinationen von `colour0` und `colour1` berechnet. (Dies gilt auch für Vektoren parallel zu dem definierten Vektor.)
- `repeat` ist ein Wert vom Typ `boolean`.
  - Bei `true` wird der Farbverlauf zyklisch (in abwechselnder Richtung) wiederholt.
  - Bei `false` findet keine Wiederholung statt. Vor `(x0,y0)` wird die Farbe `colour0` nach `(x1,y1)` die Farbe `colour2` verwendet.

Verwendung: `g2d.setPaint(gradPaint);`

Farbinterpolation

`GradientPaint` interpoliert Farbwerte mittels Konvexkombinationen:

$$(r, g, b) = (1 - \alpha) \cdot (r_0, g_0, b_0) + \alpha \cdot (r_1, g_1, b_1) \qquad (\alpha \in [0, 1])$$

Farbinterpolation bei zusammengesetzten Texturen: Interpoliere an den Rändern, z.B. mit einer Filtermatrix

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

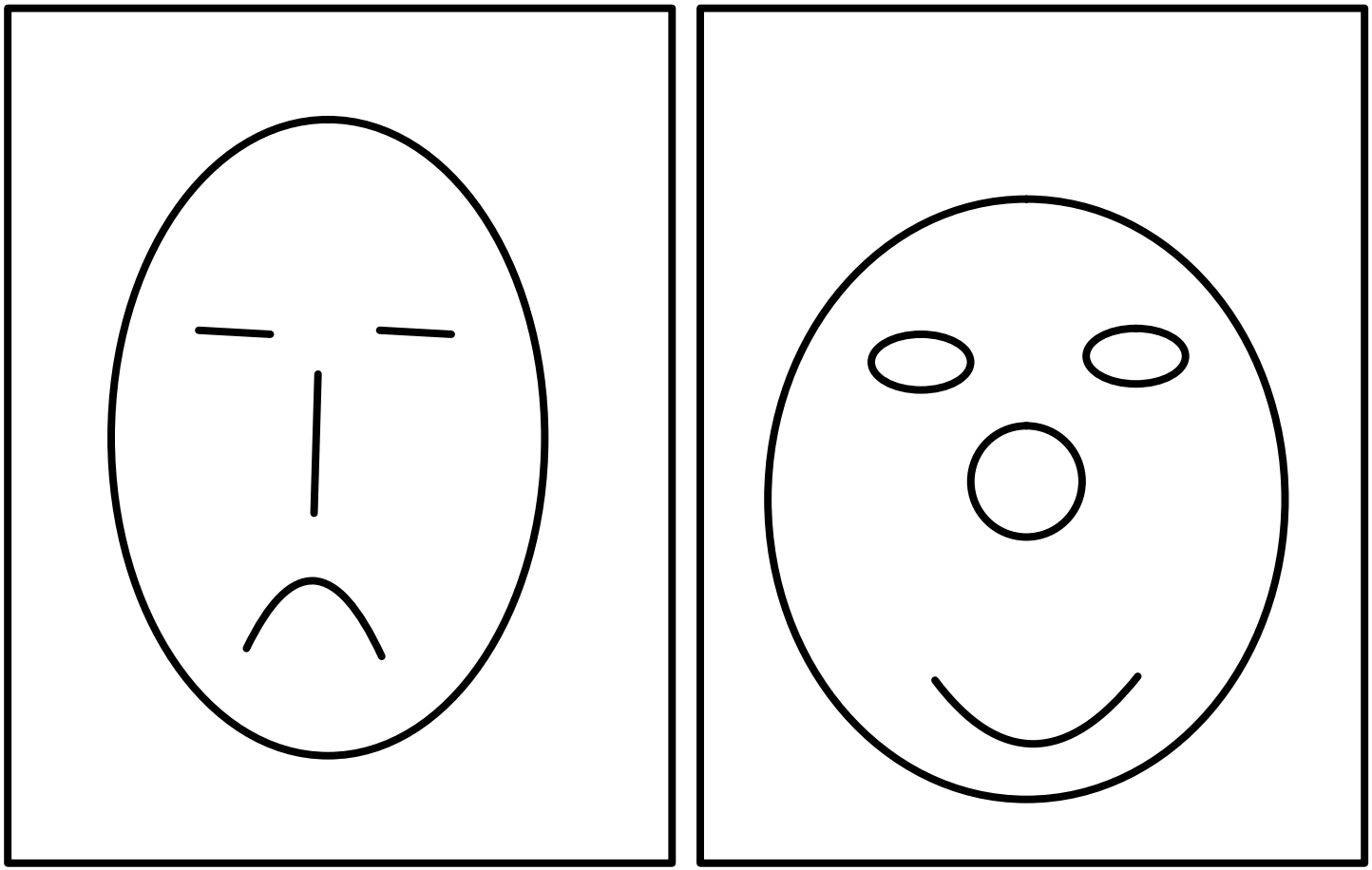
Farbinterpolation

interpolierte Farbe eines Pixels p_0:

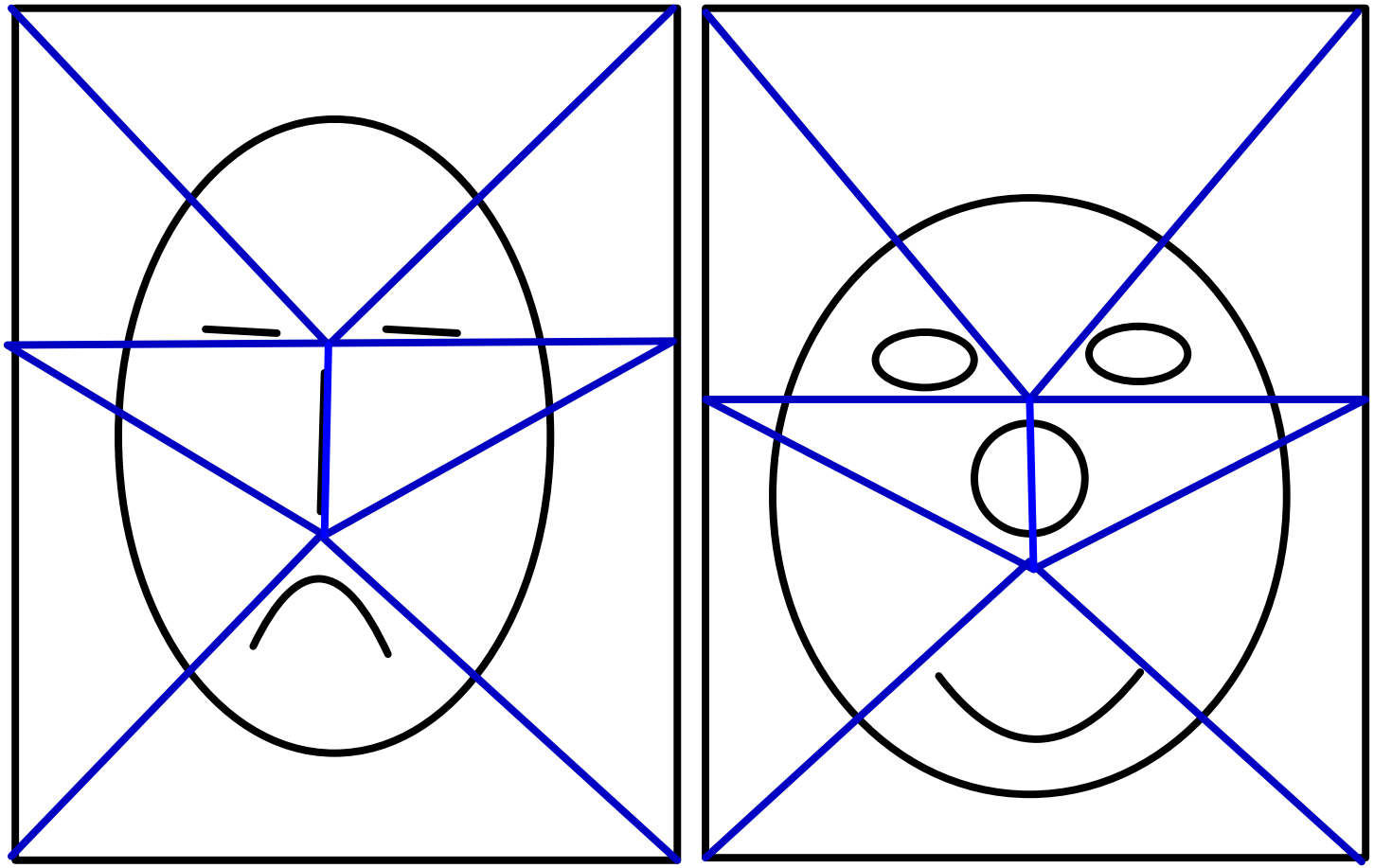
$$(r,g,b) \;=\; 0.2\cdot(r_{p_0},g_{p_0},b_{p_0}) + \sum_{p\text{: Nachbarpixel von } p_0} 0.1\cdot(r_p,g_p,b_p)$$

- statt 3×3-Filtermatrix auch größere Matrizen denkbar
- dynamische Filtermatrix: Am Rand des Bildes wird die obige Filtermatrix verwendet.
  Mit der Entfernung vom Rand gilt: Wert im Zentrum der Matrix $\rightarrow 1$, alle anderen Werte $\rightarrow 0$

Struktur- und Farbinterpolation



Struktur- und Farbinterpolation



Struktur- und Farbinterpolation

- Aus den Eckpunkten der Dreiecke werden schrittweise Konvexkombinationen gebildet, um die Triangulationen für die Zwischenbilder zu berechnen.
- Jedes Pixel $\mathbf{q}$ besitzt für jedes Dreieck eine eindeutige Darstellung durch die Eckpunkte:

$$\mathbf{q} = \alpha_1 \cdot \mathbf{p}_1 + \alpha_2 \cdot \mathbf{p}_2 + \alpha_3 \cdot \mathbf{p}_3$$

mit

$$\alpha_1 + \alpha_2 + \alpha_3 = 1.$$

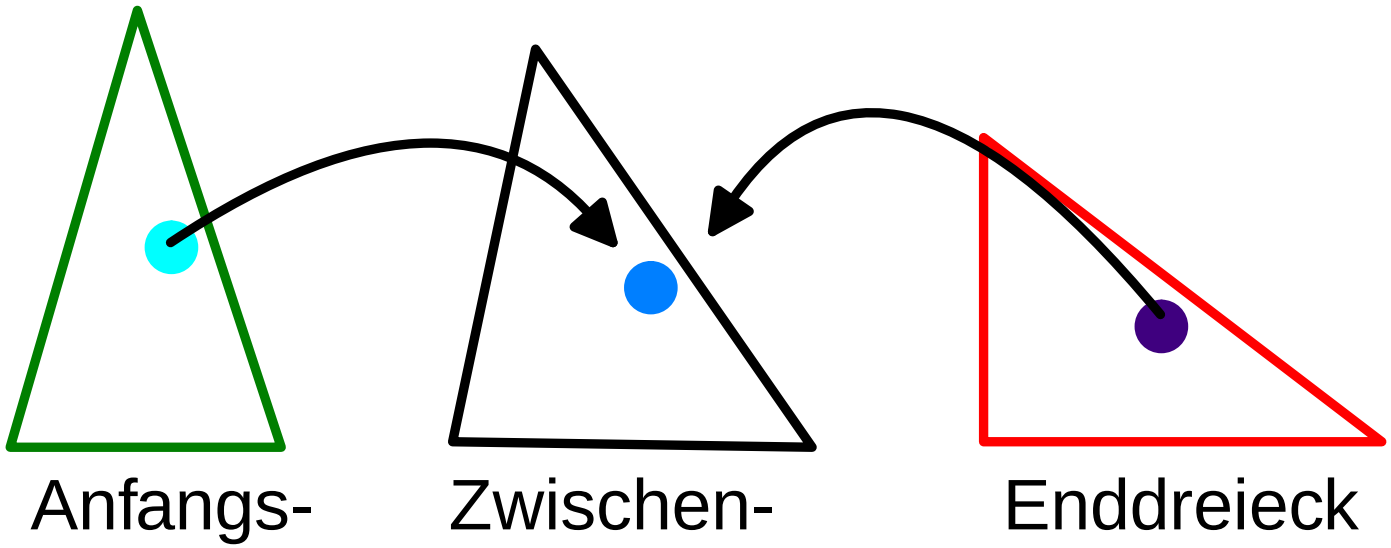
Struktur- und Farbinterpolation

- $\mathbf{q}$ liegt genau dann im Dreieck $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$, wenn $\mathbf{q}$ als Konvexkombination der Eckpunkte darstellbar ist, d.h. wenn zusätzlich gilt:

$$\alpha_1, \alpha_2, \alpha_3 \geq 0$$

Struktur- und Farbinterpolation

- In den jeweiligen Dreiecken des Zwischenschrittbildes wird der Grauwert bzw. die Farbe eines Pixels durch entsprechende konvexe Interpolation der Grau-/Farbwerte der beiden korrespondierenden Pixel im Anfangs- und Endbild berechnet.



Farbinterpolation mit Java 2D

Bestimmung der Farbe eines Pixels (x, y) in einem `BufferedImage bi`:

```
int rgbValue = bi.getRGB(x,y);
Color pixelColour = new Color(rgbValue);
```

Berechnung der RGB-Anteile:

```
  int red   = pixelColour.getRed();
  int green = pixelColour.getGreen();
  int blue  = pixelColour.getBlue();
```

Farbinterpolation mit Java 2D

Berechnung einer interpolierten Farbe mit den RGB-Anteilen (`rMix, gMix, bMix`)

Färbung des Pixels (x, y) in dieser Farbe im `BufferedImage mixedBi`:

```
Color mixedColour =
                new Color(rMix, gMix, bMix);
mixedBi.setRGB(x,y,pixelColour.getRGB());
```

(vgl. `MorphingCandS.java` und `TriangulatedImage.java`)