

5. Rasterisierung

EVC_Skriptum_CG, p.19

Was ist Rasterisierung?

- Prozess der **Umwandlung von Vektordaten** (z.B. Linien, Kurven) in **Pixel** zur Anzeige auf **bildschirmbasierten Ausgabegeräten**.
- Damit Grafiken angezeigt werden können, braucht es in der **Programmiersprache** entsprechende Befehle → sogenannte **Grafikprimitive**.

Grafikprimitive – die Bausteine der Darstellung

In 2D:

- **Punkte und Linien**
- **Polygone** (z.B. Dreiecke, Rechtecke)
- **Kreise, Ellipsen, Kurven** – auch in gefüllter Form
- **Bitmap-Operationen** (z.B. Bilder einfügen, verschieben)
- **Text** – Buchstaben, Zeichen, Zahlen

In 3D:

- **Dreiecke und andere Polygone** – Grundelemente für 3D-Modelle
- **Freiformflächen** – z.B. Bézier-Flächen, Splines

Zusätzliche Befehle zur Eigenschaftsdefinition

- Neben dem „Was zeichnen?“ braucht man auch „Wie?“
- Beispiele für **Eigenschaftsdefinitionen**:
 - **Farbe**
 - **Füllmuster**
 - **Textur** (Bild, das über Fläche gelegt wird)
 - **Materialeigenschaften** (für Beleuchtung, Glanz etc.)
 - **Transparenz**

ⓘ Merke:

Diese Einstellungen gelten **global**, d.h. sie beeinflussen **alle danach gezeichneten Objekte**, bis sie erneut geändert werden.

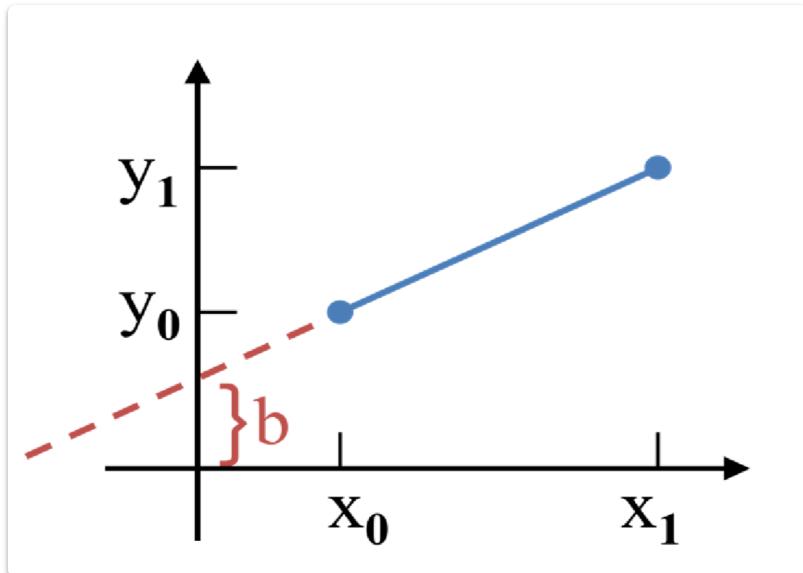
Linienalgorithmen

Linien werden in der Form $y = m * x + b$ angegeben wobei m den Anstieg beschreibt und $(0, b)$ den Schnittpunkt der y Achse.

Aus Endpunkten (x_0, y_0) und (x_1, y_1) kann man sich m und b berechnen:

$$m = \frac{(y_1 - y_0)}{(x_1 - x_0)}$$

$$b = y_0 - m * x_0$$



DDA-Verfahren

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.

```

1 dx = x1 - x0; dy = y1 - y0;
2 m = dy / dx;
3
4 x = x0; y = y0;
5 setPixel (round(x), round(y));
6
7 for (k = 0; k < dx; k++) {
8     x += 1; y += m;
9     setPixel (round(x), round(y))
10 }
```

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 erhält man durch Spiegelung und durch Rotation um 90° .

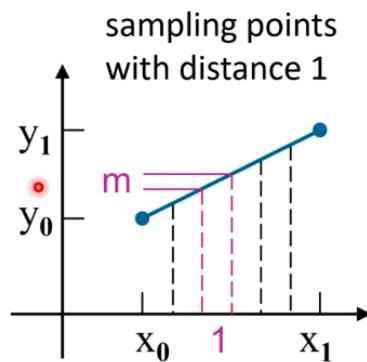
DDA Line-Drawing Algorithm

line equation: $y = m \cdot x + b$

$$\delta y = m \cdot \delta x \quad \text{for } |m| < 1$$

$$\left(\delta x = \frac{\delta y}{m} \quad \text{for } |m| > 1 \right)$$

DDA (digital differential analyzer):



```
for  $\delta x=1$ ,  $|m|<1$  :  $y_{k+1} = y_k + m$ 
```

DDA – Based on Taylor Series Expansion

$$T_{f(x;a)} = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \frac{f'''(a)}{6}(x-a)^3 + \dots$$

$$f(x+1) = g(f(x), f'(x), f''(x), \dots) \quad (x+1)^n = \sum_{k=0}^n \binom{n}{k} x^k$$

$$f(x) = x^2, \quad f'(x) = 2x, \quad f''(x) = 2, \quad f'''(x) = 0$$

$$f(x+1) = (x+1)^2 = x^2 + 2x + 1 = f(x) + f'(x) + 1$$

$$f'(x+1) = 2x + 2 = f'(x) + 2$$



Bresenham-Verfahren

Man schaut ob man näher zum oberen oder unteren Pixel ist und setzt das dann so

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.

Bei Linien lässt sich der nächste Punkt so berechnen:

$$y = m * (x_k + 1) + b$$

(lässt sich ganz einfach aus Linearen Funktionen erschließen)

Hier werden dann nicht die genauen y Werte berechnet sondern lediglich die Entscheidung getroffen, ob y_k oder y_{k+1} näher zum exakten y -Wert liegt.

Abstand zu y_k ist:

$$d_{lower} = y - y_k = m * (x_k + 1) + b - y_k$$

Abstand zu y_{k+1} ist:

$$d_{upper} = (y_k + 1) - y = y_k + 1 - m * (x_k + 1) - b$$

Nun berechnet man sich die Differenz zwischen d_{lower} und d_{upper} :

$$d_{lower} - d_{upper}$$

- Wenn diese Differenz negativ ist, dann nimmt man den unteren Punkt (x_{k+1}, y_k)
- Wenn positiv den oberen (x_{k+1}, y_{k+1})

Optimierung durch Entscheidungsvariable

Um keine Fließkommaoperationen (Multiplikation/Division) durchführen zu müssen, wird eine **Entscheidungsvariable** eingeführt. Dazu setzt man:

$$m = \frac{\Delta y}{\Delta x} \quad \text{mit} \quad \Delta x = x_1 - x_0, \quad \Delta y = y_1 - y_0$$

Multipliziert man die obige Differenz mit Δx , ergibt sich:

$$p_k = \Delta x \cdot (d_{lower} - d_{upper}) = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c$$

Diese Entscheidungsvariable hat dasselbe Vorzeichen wie $d_{lower} - d_{upper}$, benötigt aber keine Division mehr.

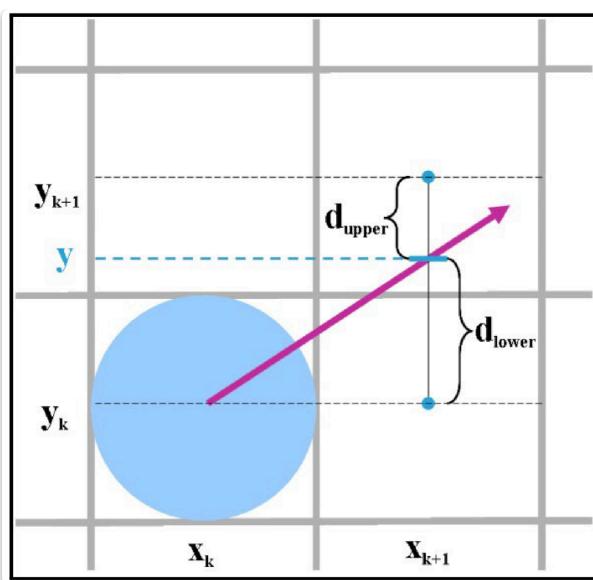
Rekursive Berechnung der nächsten Entscheidungsvariable:

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

Das bedeutet: Die neue Entscheidungsvariable lässt sich **einfach aus der vorherigen berechnen**, je nachdem, ob y erhöht wurde oder nicht – also ganz ohne Neuberechnung des exakten y -Werts.

Startwert:

$$p_0 = 2\Delta y - \Delta x$$



Attribute

Graphikprimitive können mit vielerlei Eigenschaften erzeugt werden, sogenannten Attributen.

Attribute von Punkten und Linien

Neben allgemein bekannten Eigenschaften von Linien, wie Strichdicke, Strichierungsmuster, Farbe oder Pinseltyp, gibt es noch ein paar Attribute, die einem oft weniger bewusst sind. Dazu gehören etwa die Linienenden bei breiteren Linien sowie die Form von Ecken bei breiten Linien:



Weiters ist Antialiasing auch für Linien ein Thema, dazu werden etwas weiter unten Details gebracht.

Attribute von Text

[EVC_Skriptum_CG, p.21](#)

Typische Eigenschaften von Text:

- **Font / Schriftart:**
z. B. *Courier, Helvetica, Times, Fraktal*
- **Stil:**
normal, fett, kursiv, unterstrichen, durchgestrichen etc.
- **Größe:**
in Punkten angegeben (z. B. 12pt)
- **Richtung:**
horizontal, vertikal, rotiert
- **Farbe:**
z. B. RGB-Werte oder vordefinierte Farbnamen

- **Bündigkeit:**
links, rechts, zentriert, Blocksatz

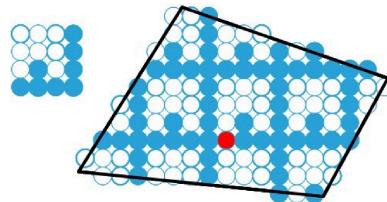
Serifen vs. serifenos

- **Fonts mit Serifen** (z.B. Times):
 - ✓ Besser geeignet für **Fließtext** – durch Serifen wird das Auge entlang der Zeile geführt.
- **Fonts ohne Serifen** (z.B. Helvetica):
 - ✓ Ideal für **plakativen Text**, Überschriften oder Bildschirmdarstellung.



Attribute von (2D-) Polygonen und Flächen

Klarerweise sind die **Attribute des Randes von Flächen** dieselben wie die von Linien. Dazu kommt nun die **Fläche selbst**, die mit einer **Füllung** versehen werden kann. Muster werden dabei gewöhnlich durch repetitive Aneinanderreihung eines Grundmusters ausgehend von einem **Referenzpunkt** (auch Seed-Point genannt) erzeugt.



In vielen Anwendungen ist es auch notwendig, eine **Kombination** des neu gezeichneten Musters mit dem Hintergrund zu erzeugen. Hier gibt es viele Varianten, die oft auf logischen Verknüpfungen aufbauen: **AND**, **OR**, **XOR**. Das Mischen von Farben erfolgt meist durch Linearkombination der vorhandenen Hintergrundfarbe B mit der zu zeichnenden Vordergrundfarbe F : $P = t \cdot F + (1 - t) \cdot B$

Baryzentrische Koordinaten

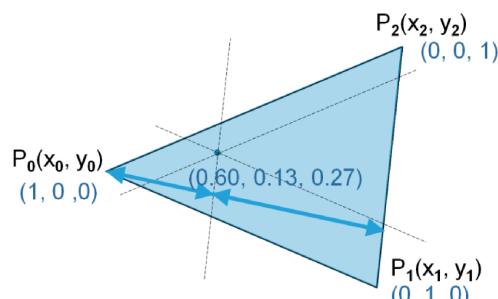
Baryzentrische Koordinaten sind eine Grundlage für die Interpolation von Pixeln in Dreiecken.

Dreiecke rasterisieren

Um Dreiecke zu füllen verwendet man oft **baryzentrische Koordinaten**. Jeder Punkt der Ebene wird dabei als gewichtetes Mittel der drei Eckpunkte des Dreiecks dargestellt:

$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

(α, β, γ) nennt man dann die **baryzentrischen Koordinaten** des Punktes P , wobei immer gilt: $\alpha + \beta + \gamma = 1$. Alle Punkte mit $(0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1)$ liegen innerhalb des Dreiecks; sobald einer dieser Werte negativ oder größer als 1 ist, liegt der Punkt außerhalb des Dreiecks.



Zum Füllen eines Dreiecks berechnet man für jedes Pixel einer (möglichst engen) Umgebung dessen baryzentrische Koordinaten und zeichnet alle für deren Mittelpunkt ($0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1$) gilt. Dabei kann man sehr einfach beliebige Eckpunktattribute (z.B. Farbe) in jedem Pixel mit (α, β, γ) gewichtet berechnen, dies entspricht einer linearen Interpolation dieser Werte.

Baryzentrische Koordinaten beschreiben einen Punkt P im Bezug auf die Eckpunkte eines Dreiecks P_0, P_1, P_2 :

$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

mit der Bedingung:

$$\alpha + \beta + \gamma = 1$$

Punkt liegt innerhalb des Dreiecks, wenn gilt:

$$0 < \alpha < 1, \quad 0 < \beta < 1, \quad 0 < \gamma < 1$$

Füllalgorithmus:

- Für jedes Pixel im Bounding-Box-Rechteck des Dreiecks:
 - Baryzentrische Koordinaten (α, β, γ) des Pixelmittelpunkts berechnen.
 - Liegt der Punkt **innerhalb** des Dreiecks? → **Pixel zeichnen**

Vorteil:

Mit α, β, γ lassen sich beliebige **Attribute** (z. B. Farbe, Tiefe, Textur) linear interpolieren:

$$Attribut(P) = \alpha \cdot Attribut(P_0) + \beta \cdot Attribut(P_1) + \gamma \cdot Attribut(P_2)$$

Beispiel der Baryzentrischen Koordinaten:

Angenommen, du hast ein Dreieck mit den Eckpunkten:

- A,
- B,
- C.

Dann sind die **baryzentrischen Koordinaten** für jeden Eckpunkt wie folgt:

Eckpunkt A:

$$\rightarrow (1, 0, 0)$$

Bedeutet: 100 % bei A, 0 % bei B, 0 % bei C → also exakt Punkt A.

Eckpunkt B:

$$\rightarrow (0, 1, 0)$$

Bedeutet: 100 % bei B → also exakt Punkt B.

Eckpunkt C:

$$\rightarrow (0, 0, 1)$$

Bedeutet: 100 % bei C → also exakt Punkt C.

Also:

- Die baryzentrischen Koordinaten (α, β, γ) eines Punkts P im Dreieck erfüllen immer:
 $P = \alpha A + \beta B + \gamma C$ mit $\alpha + \beta + \gamma = 1$
- Für Punkte innerhalb des Dreiecks sind alle drei Werte positiv.
- Für Punkte auf einer Kante ist eine Koordinate = 0.
- Für die Eckpunkte ist zwei Koordinaten = 0.

General Polygon Fill Algorithms

TU
WIEN

- **triangle rasterization**
- other polygons: what is inside?
- scan-line fill method
- flood fill method

barycentric coordinates!

clean borders between adjacent triangles?

Werner Purgathofer 31

Berechnen der baryzentrischen Koordinaten

Sei $g_{12}(x, y) = a_{12}x + b_{12}y + c_{12} = 0$ die Trägergerade durch die Punkte P_1 und P_2 , dann berechnet sich α des Punktes $P(x_p, y_p)$ zu
 $\alpha = g_{12}(x_p, y_p)/g_{12}(x_0, y_0)$.

β und γ werden analog berechnet.

Um das doppelte Zeichnen der Kanten aneinander grenzender Dreiecke zu vermeiden, werden nur Pixel gezeichnet, deren Mittelpunkt innerhalb eines (exakten) Dreiecks liegen. Pixel genau auf einer Kante sind speziell zu behandeln, z.B. durch Regeln wie „Kanten unten und rechts werden gerendert, Kanten oben und links nicht“. Dadurch stellt man sicher, dass jedes Kantenpixel nur einmal behandelt wird.

Gegeben: ein Dreieck mit Eckpunkten $P_0 = (x_0, y_0)$, $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$

Und ein beliebiger Punkt $P = (x, y)$, z.B. der Mittelpunkt eines Pixels

Schritt 1: Trägergeraden der Dreiecksseiten

Zu jeder Seite des Dreiecks wird eine lineare Funktion $g_{ij}(x, y)$ bestimmt, die null ist, wenn der Punkt auf der Geraden zwischen P_i und P_j liegt:

$$g_{ij}(x, y) = a_{ij}x + b_{ij}y + c_{ij}$$

Diese Gerade ist die Kante gegenüber von Punkt P_k .

Zum Beispiel:

- $g_{12}(x, y)$: Gerade durch P_1 und $P_2 \rightarrow$ gegenüber von $P_0 \rightarrow$ liefert α

- $g_{20}(x, y)$: Gerade durch P_2 und $P_0 \rightarrow$ gegenüber von $P_1 \rightarrow$ liefert β
- $g_{01}(x, y)$: Gerade durch P_0 und $P_1 \rightarrow$ gegenüber von $P_2 \rightarrow$ liefert γ

Die Formel für $g_{ij}(x, y)$ ist:

$$g_{ij}(x, y) = (y_i - y_j)x + (x_j - x_i)y + (x_i y_j - x_j y_i)$$

Schritt 2: Baryzentrische Koordinate berechnen

Um z.B. α zu berechnen (also Anteil von P_0), setzt man den Punkt $P = (x, y)$ in g_{12} ein:

$$\alpha = \frac{g_{12}(x, y)}{g_{12}(x_0, y_0)}$$

Analog für β und γ :

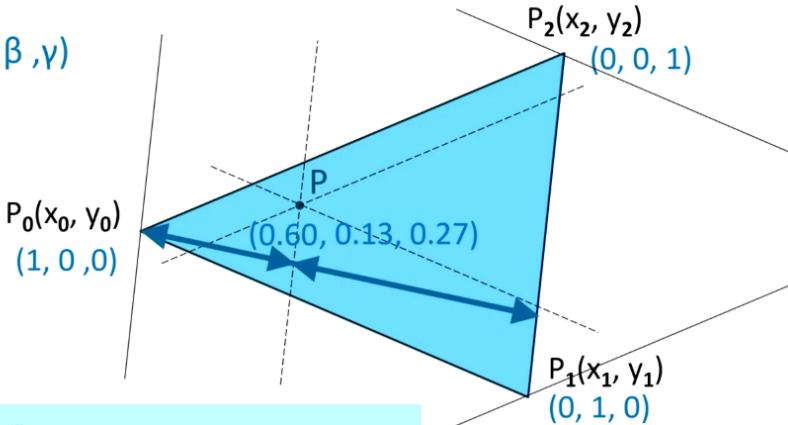
$$\beta = \frac{g_{20}(x, y)}{g_{20}(x_1, y_1)}, \quad \gamma = \frac{g_{01}(x, y)}{g_{01}(x_2, y_2)}$$

Der Nenner ist der jeweilige Wert, den die Gerade beim zugehörigen Eckpunkt liefert. Da die Punkte P_0, P_1, P_2 nicht auf ihren gegenüberliegenden Seiten** liegen, ist das kein Problem.

kompaktere Version hier: [5. FS - Rasterisierung > Barzentrische Koordinaten berechnen](#)

Triangles: Barycentric Coordinates

notation: (α, β, γ)



$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

$$\text{triangle} = \{P \mid \alpha + \beta + \gamma = 1, 0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1\}$$

Triangle Rasterization Algorithm



for all x

```

    for all y          /* use a bounding box! */
        {compute ( $\alpha, \beta, \gamma$ ) for (x, y) ;
         if ( $0 < \alpha < 1$ ) and ( $0 < \beta < 1$ ) and ( $0 < \gamma < 1$ )
         {
             draw pixel (x, y)
         }
     }
```

$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

$$\text{triangle} = \{P \mid \alpha + \beta + \gamma = 1, 0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1\}$$

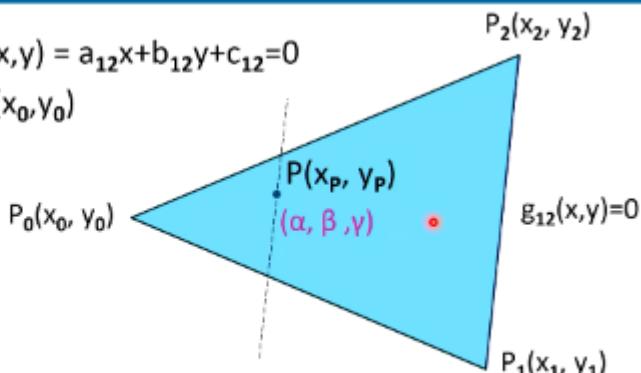


Computing (α, β, γ) for $P(x_p, y_p)$



line through P_1, P_2 : $g_{12}(x, y) = a_{12}x + b_{12}y + c_{12} = 0$

then $\alpha = g_{12}(x_p, y_p) / g_{12}(x_0, y_0)$



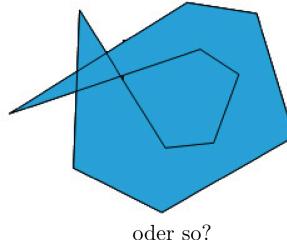
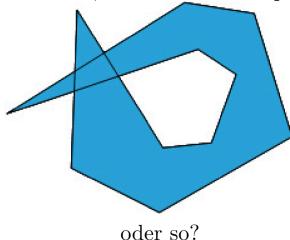
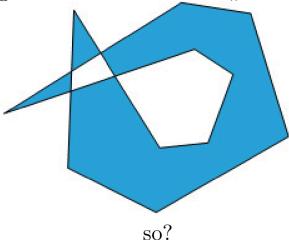
$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

$$\text{triangle} = \{P \mid \alpha + \beta + \gamma = 1, 0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1\}$$

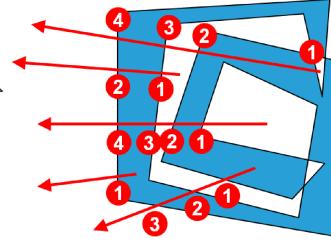
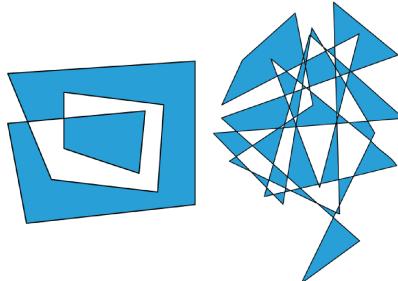


Was ist in einem Polygon innen?

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

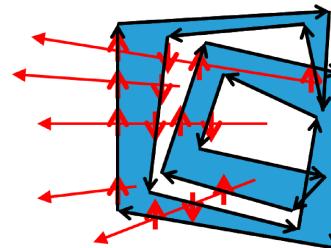
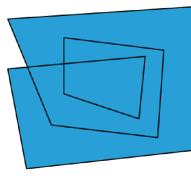


Odd-Even-Rule Zieht man von einem Punkt aus einen beliebigen Halbstrahl, so ist der Punkt innerhalb, wenn die Zahl der Schnitte mit der Kurve ungerade ist, ansonsten ist der Punkt außerhalb (in Abb. oben links, sowie alle Bilder rechts). Jede Kante hat also eine Seite innen und die andere außen.



Nonzero-Winding-Number-Rule

Rule Punkte sind außerhalb, wenn sich auf einem beliebigen Halbstrahl gleich viele im Uhrzeigersinn und gegen den Uhrzeigersinn verlaufende Kurvenkanten befinden, ansonsten innerhalb (in Abb. oben Mitte, sowie alle Bilder rechts).



All-In-Rule Alles, was irgendwie umschlossen ist, ist innen. Wird selten verwendet, meist beim Pokern \odot (in Abb. oben rechts).

Ein Polygon heißt *konvex* wenn alle inneren Winkel kleiner als 180° sind (oberes Bild), andernfalls *konkav* (unteres Bild). Da konvexe Polygone viel weniger Sonderfälle erzeugen, sind viele Algorithmen für konvexe Polygone ausgelegt (oft sogar nur für Dreiecke). Daher braucht man auch Methoden um konkave Polygone in mehrere konvexe Polygone zu zerlegen (oft in Dreiecke).

