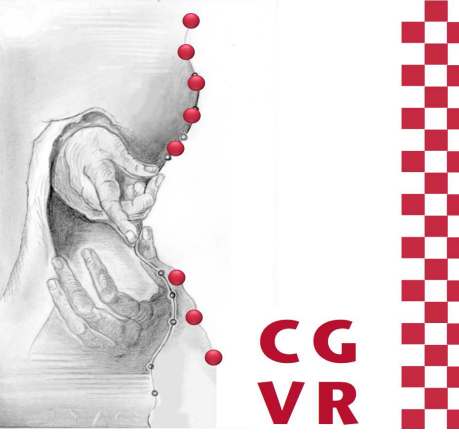
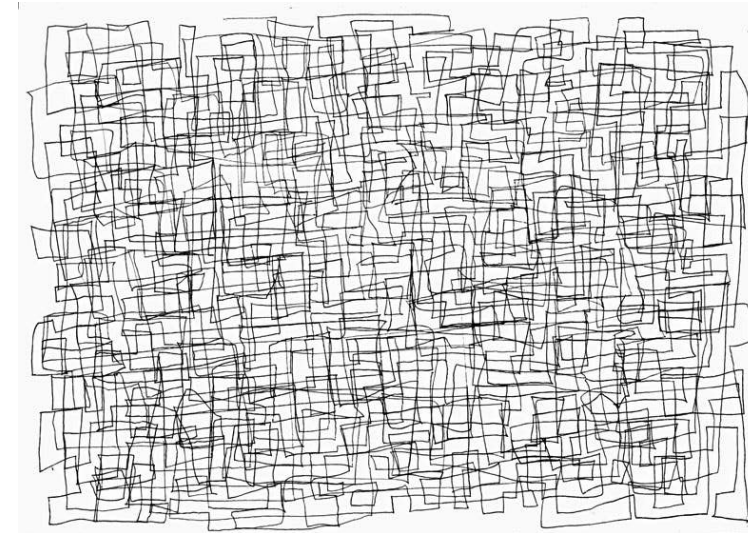




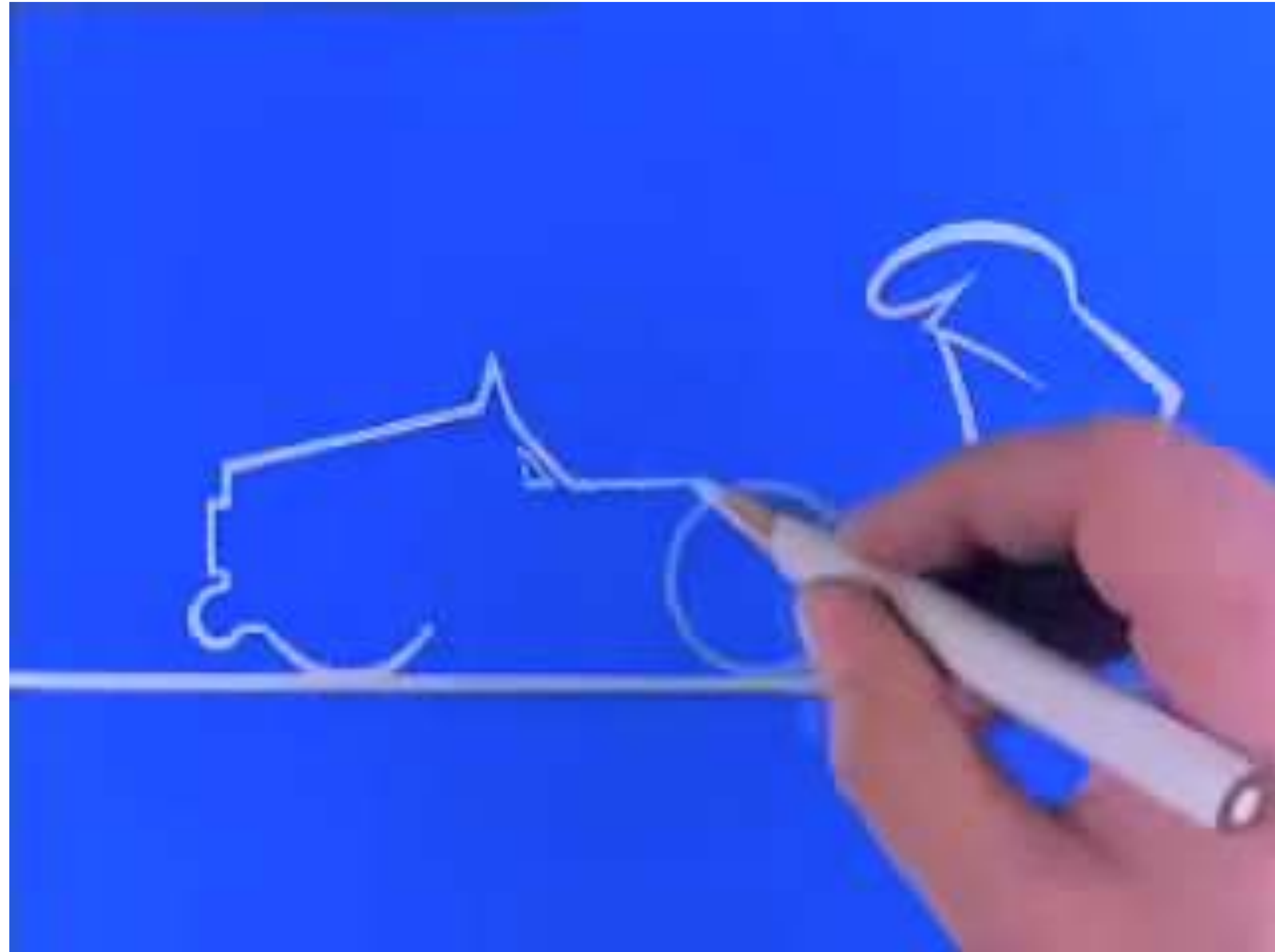
Computergraphik I

Scan Conversion of Lines



G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de





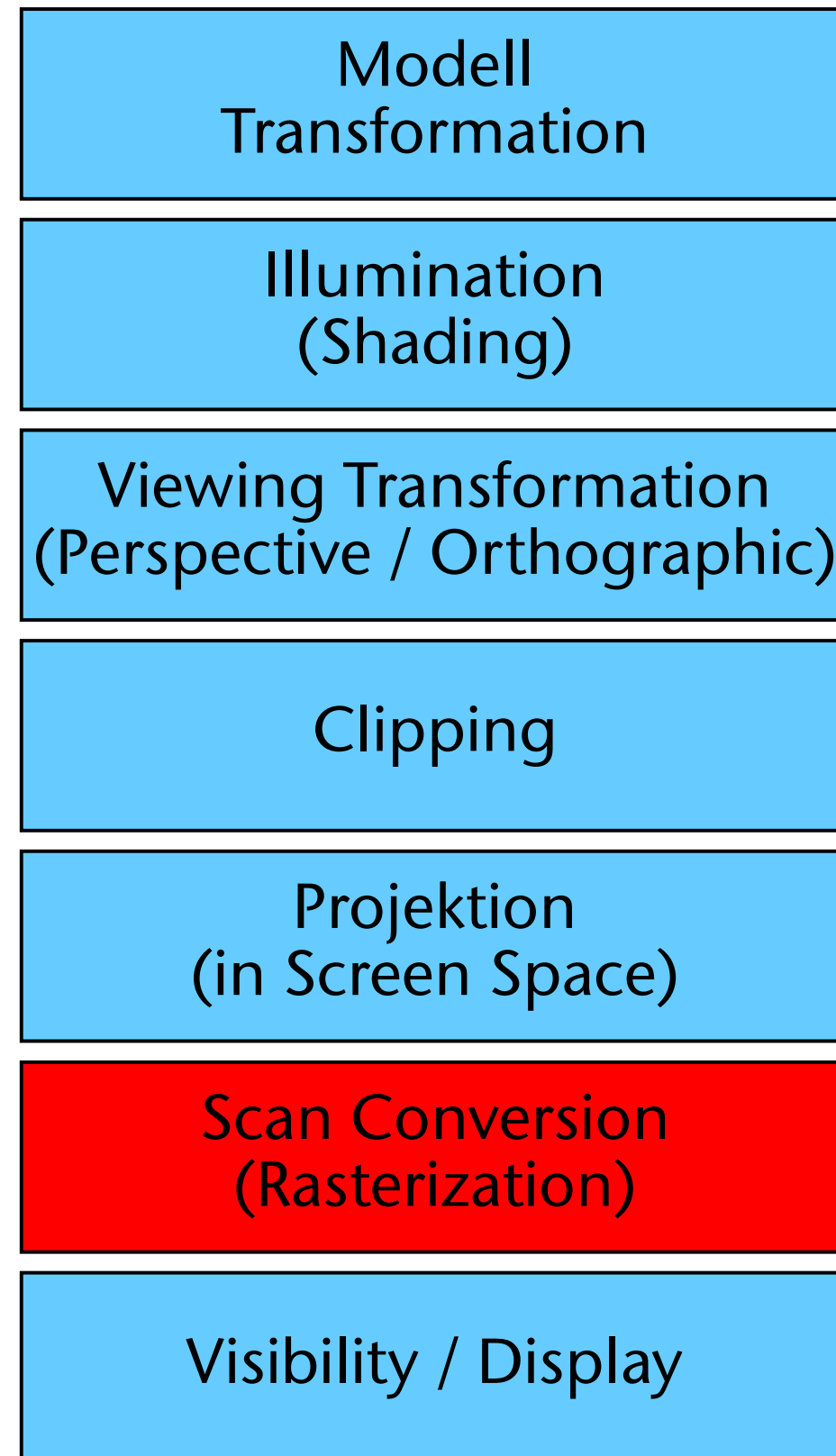
"La Linea"

Das Zeichnen von Linien

- Der Begriff **Scan-conversion** oder **Rasterisierung** bezeichnet allgemein das algorithmische Bestimmen, welche Pixel von dem Primitiv überdeckt werden
 - Der Name kommt von der Scan-Technik der Rasterdisplays
- Was bedeutet es mathematisch?

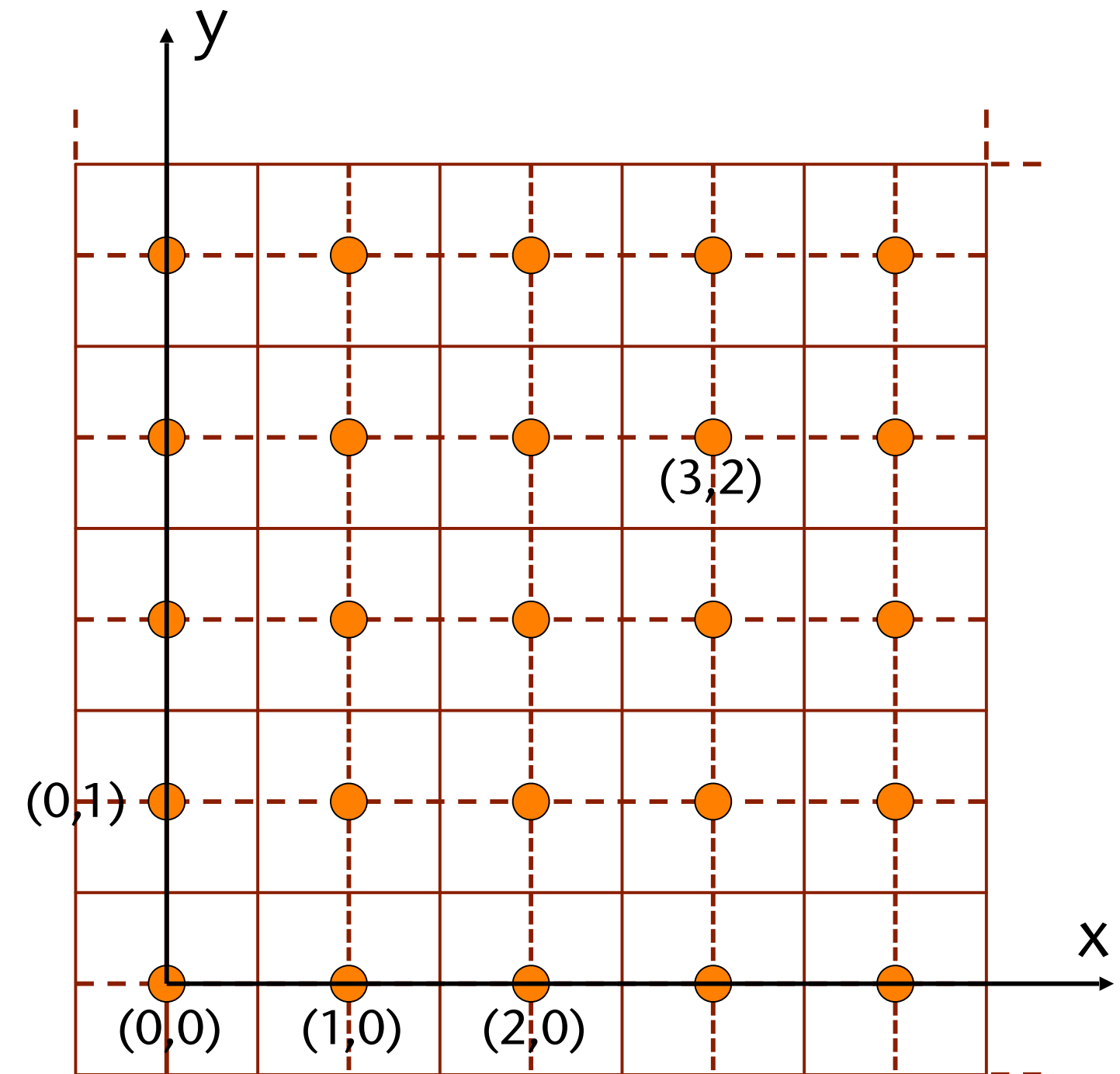
Diskretisierung von **kontinuierlichen** geometrischen Objekten
- Zusätzliche Aufgabe: Ecken-Werte interpolieren (z.B. Farbe, Tiefenwert, ...)
- Scan-Conversion ist grundlegend für 2D und 3D Computergraphik

Einordnung in die Pipeline



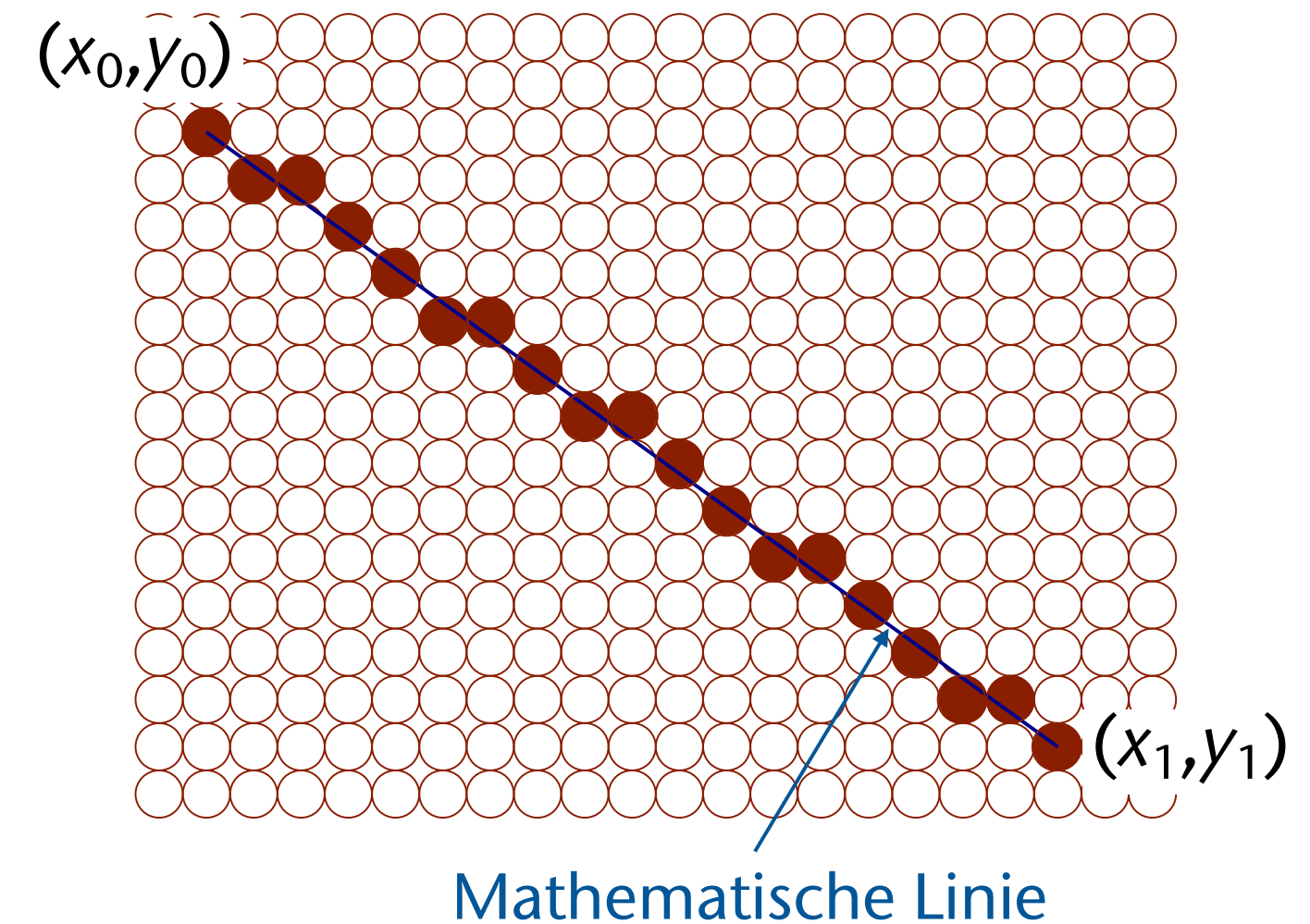
Bildschirmkoordinaten im Frame-Buffer

- Wir verwenden folgende 2D Bildschirmkoordinaten
 - Ganzzahlige Koordinaten für die *Mittelpunkte* der Pixel
 - Senkrecht = Y–Achse, Horizontal = X–Achse



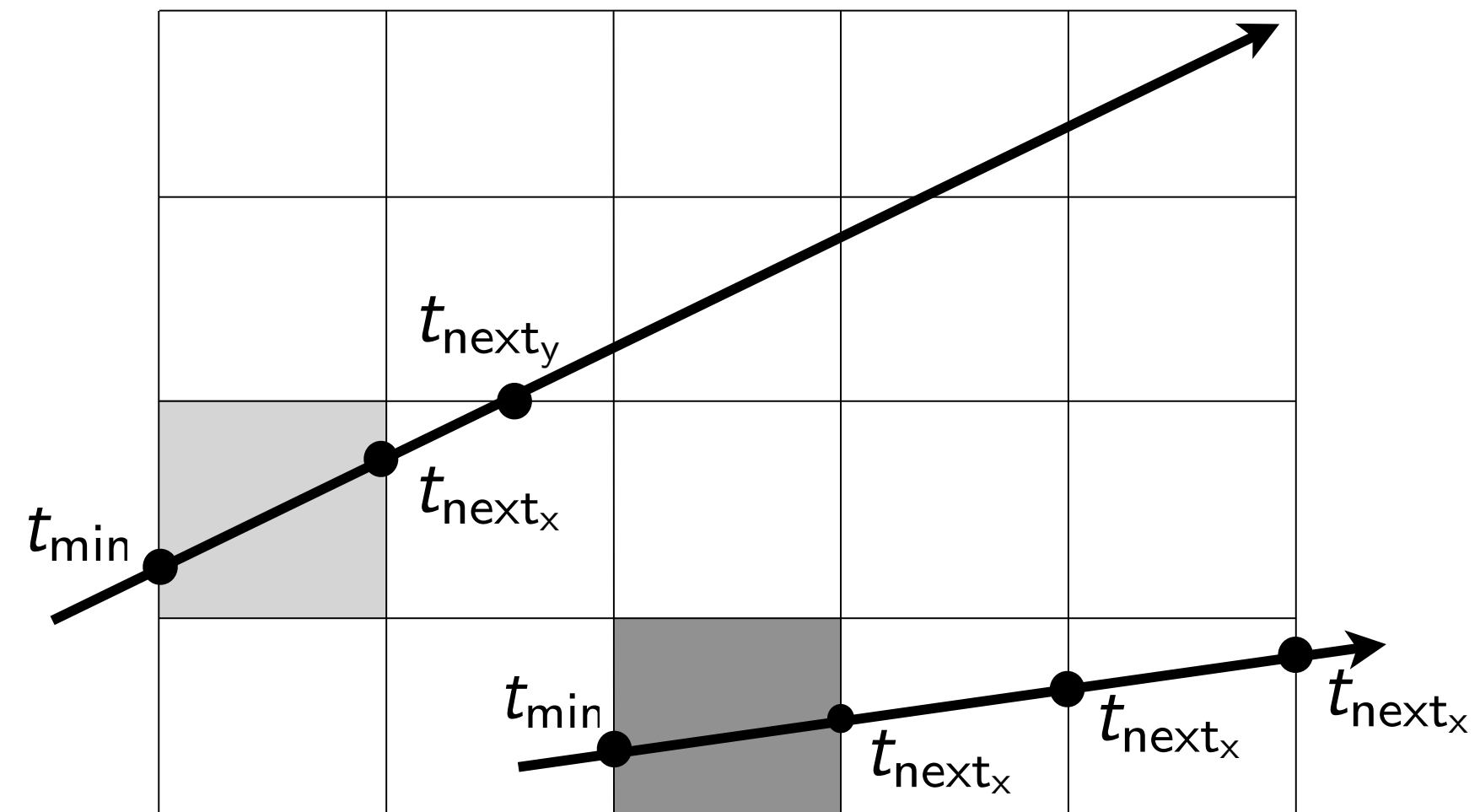
Die Ideale Linie

- Kontinuität: keine Unterbrechungen (diagonale Schritte sind erlaubt)
- Minimalität: setze nur die "nähesten" Pixel an der idealen Linie
- Uniformität: einheitliche Stärke und Helligkeit
- Optimalität: Performanz, d.h., so wenig FLOPs pro Pixel wie möglich
- Invarianz: gegenüber Zeichenrichtung

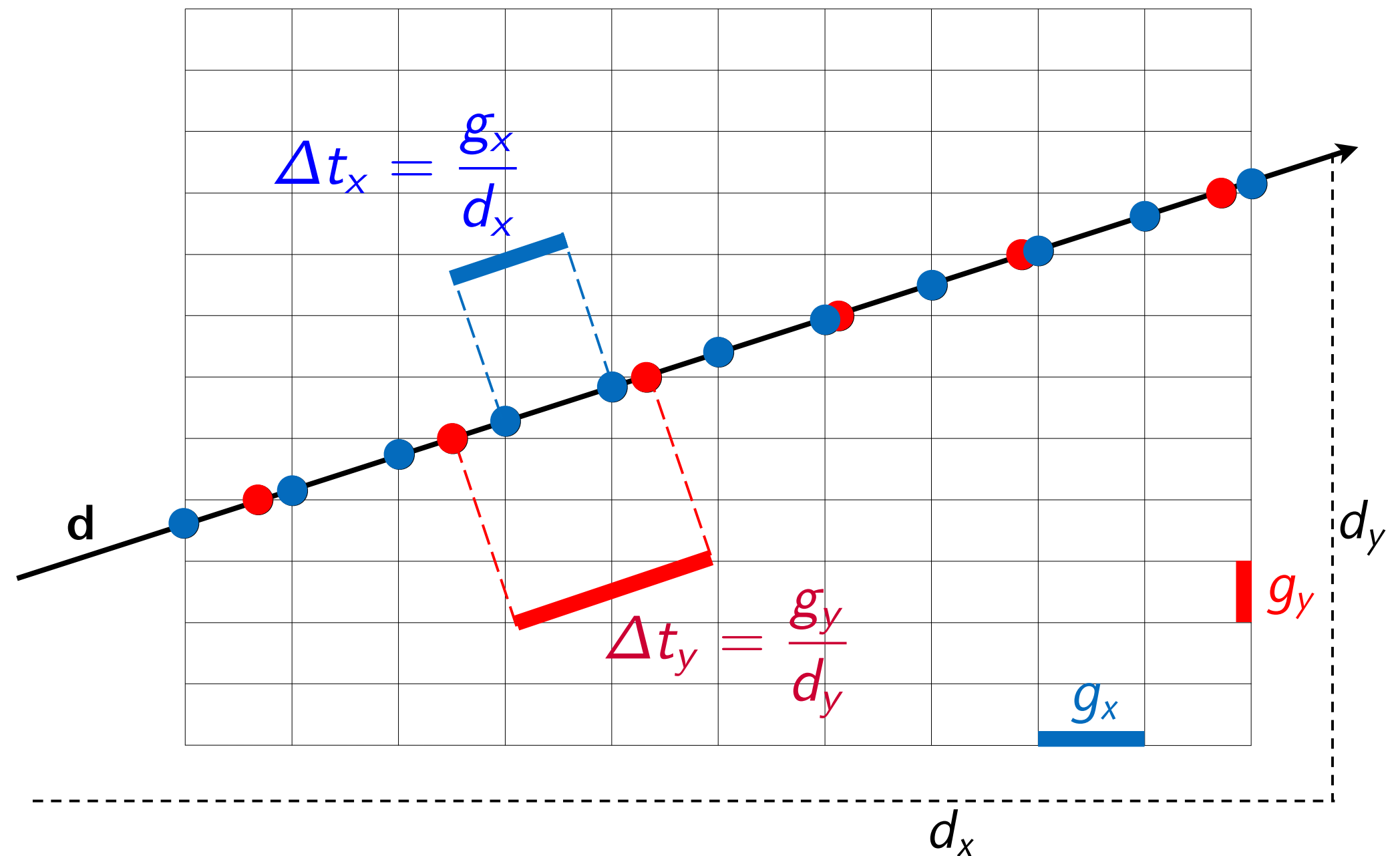


Ein einfacher, inkrementeller Algorithmus

- Ansatz:
 - Verwende Parameterform der Linie
$$P(t) = P_0 + t (P_1 - P_0)$$
 - Berechne Folge von Linienparametern t , an denen die Linie eine senkrechte (x) oder horizontale (y) Gitterlinie schneidet
- Vereinfachung: erster Linienparameter $t_{\min}$ sei gegeben
 - Caveat: Startpunkt liegt normalerweise *innerhalb* eines der Pixel



- Gibt es ein Muster bei den Übergängen?
- Betrachte nur die Schnitte mit den x-Gitterlinien → scheinbar selber Abstand aller dieser Schnittpunkte (t 's)
- Analog: Schnitte mit den y-Gitterlinien

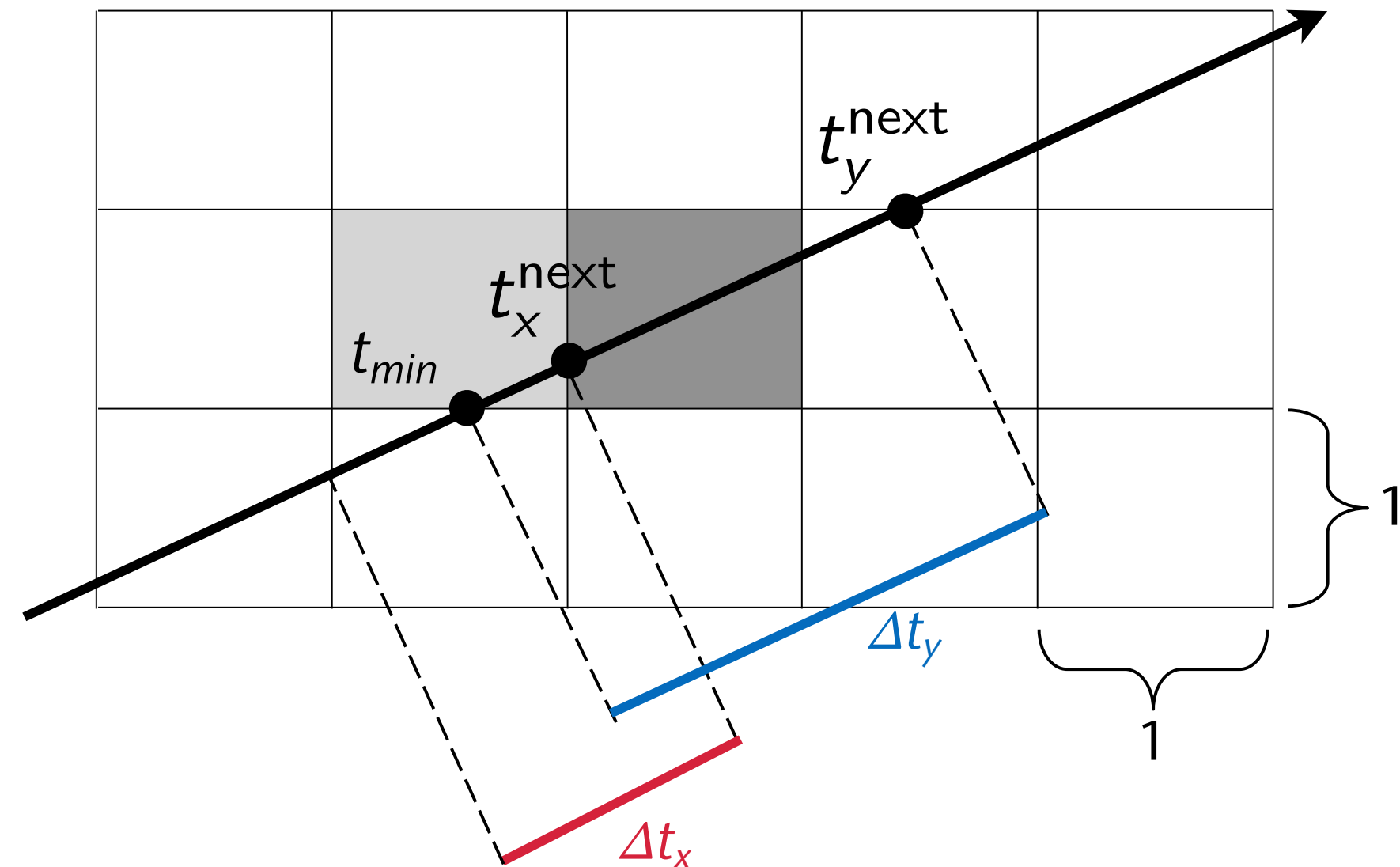


Ein inkrementeller Algorithmus

```

init tmin = 0
calc first tnext_x and tnext_y
calc dtx and dty
init pixel coords [x,y] for P0
while tmin < 1.0 :
    buffer[x,y] = color
    if tnext_x < tnext_y :
        x += 1           // g_x = 1
        tmin = tnext_x
        tnext_x += dtx
    else:
        y += 1           // g_y = 1
        tmin = tnext_y
        tnext_y += dty

```

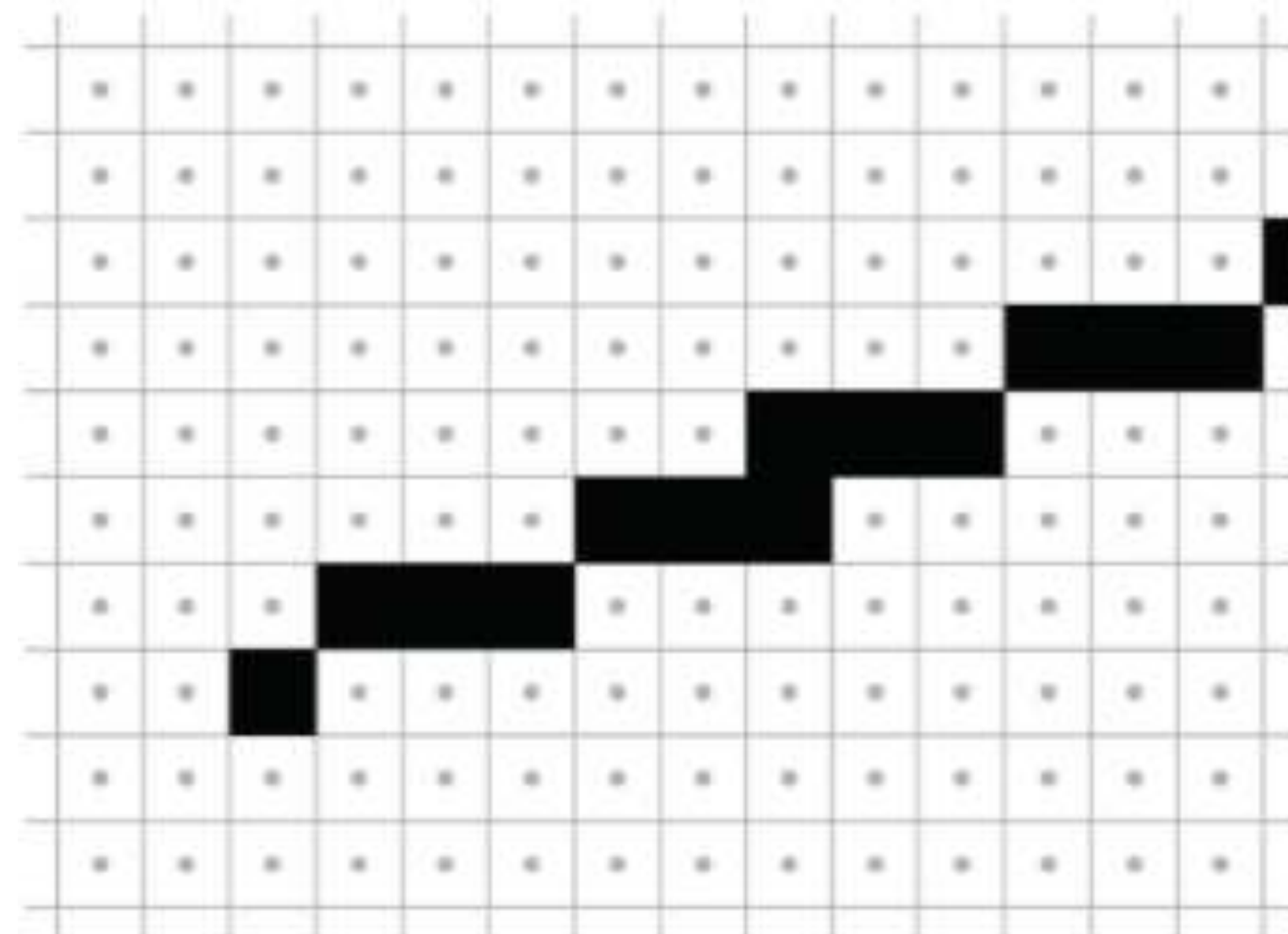


Bitte nicht durch
"look-ahead"
spoilern!



<https://www.menti.com/ojndv9fafk>

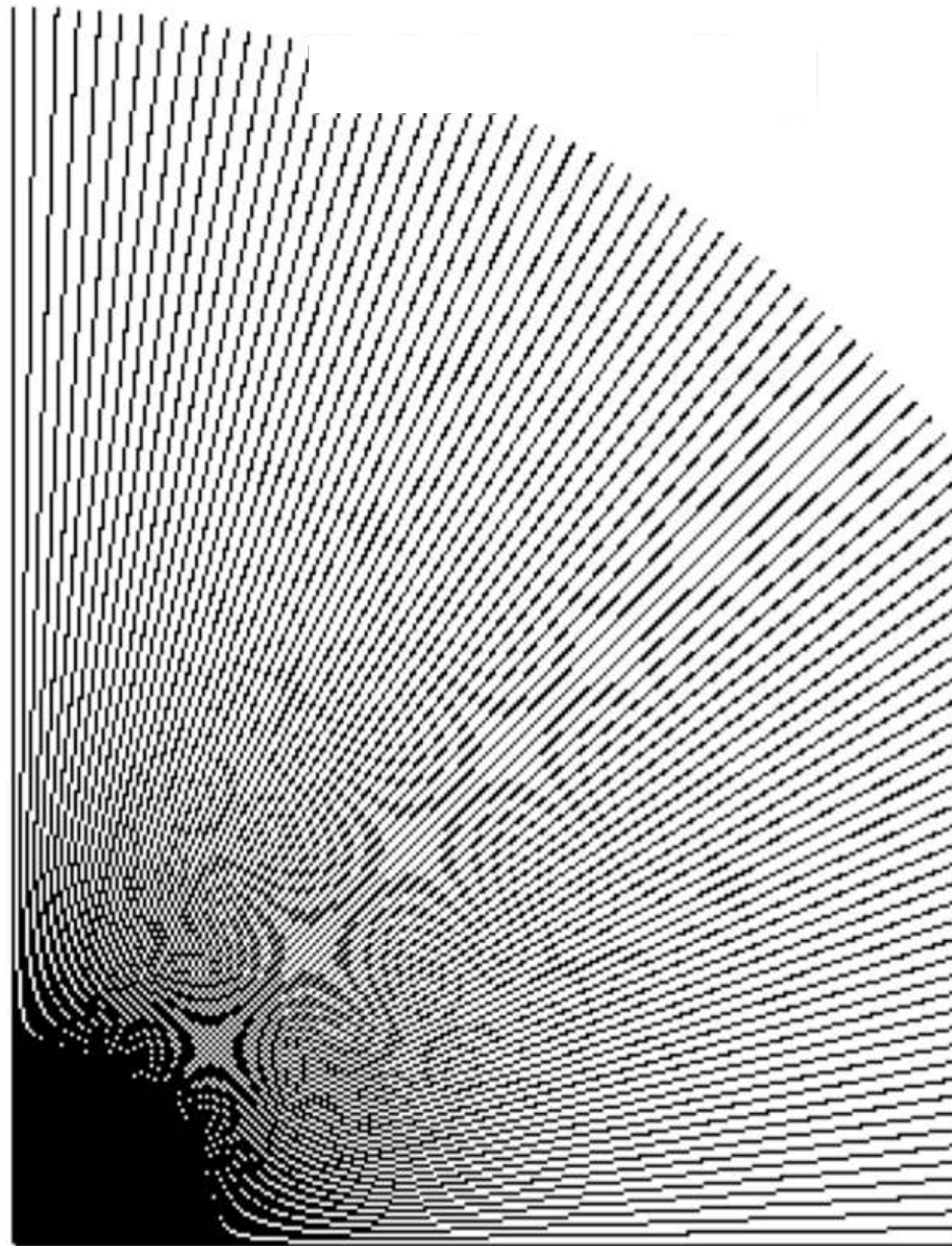
Einer der Nachteile des Algorithmus



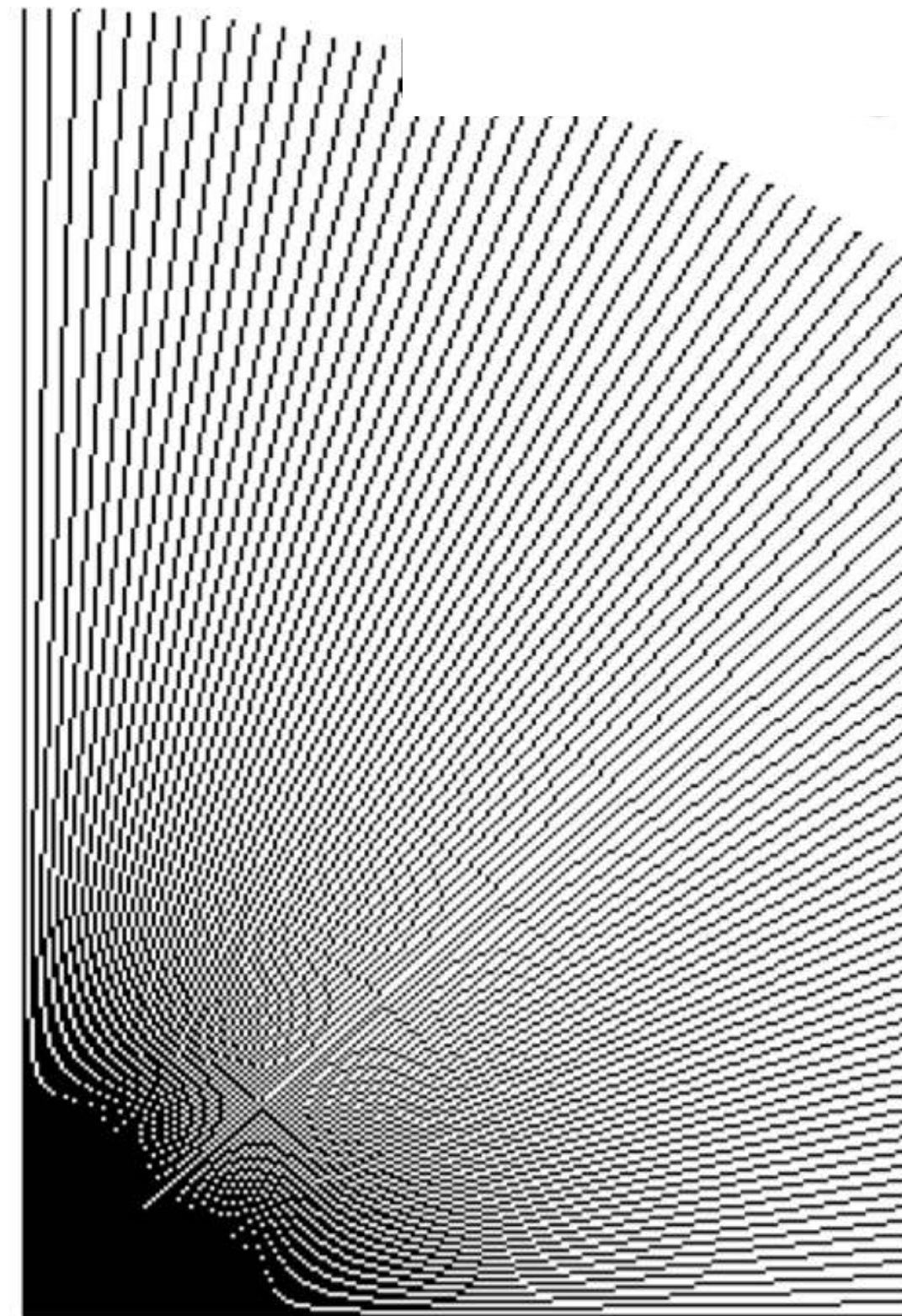
Manchmal werden vertikal übereinander liegende Pixel gesetzt → unterschiedliche scheinbare Linienstärke

Zweite Konsequenz des Problems: Moiré Patterns

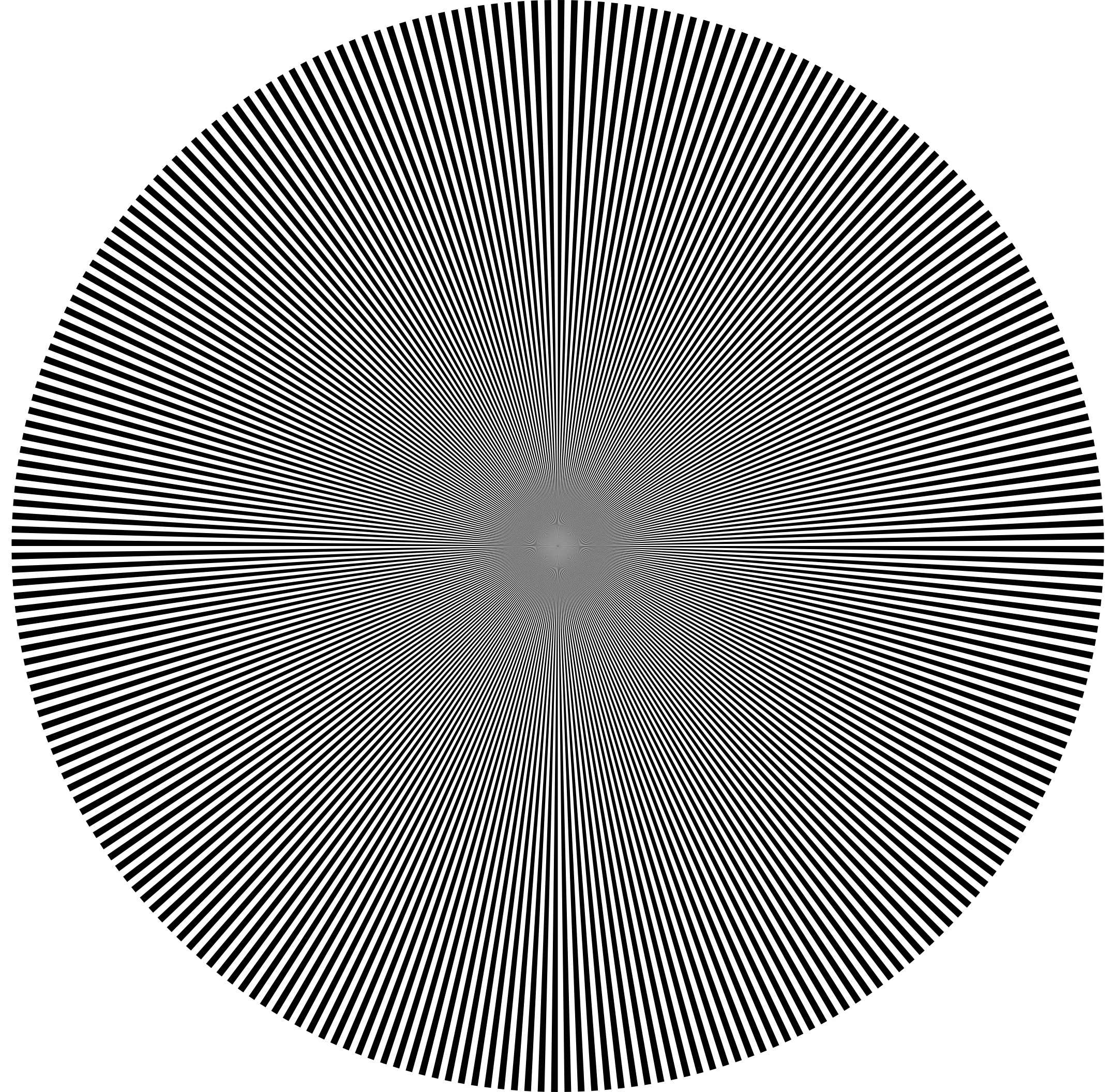
Aussehen mit
unserem
Algorithmus



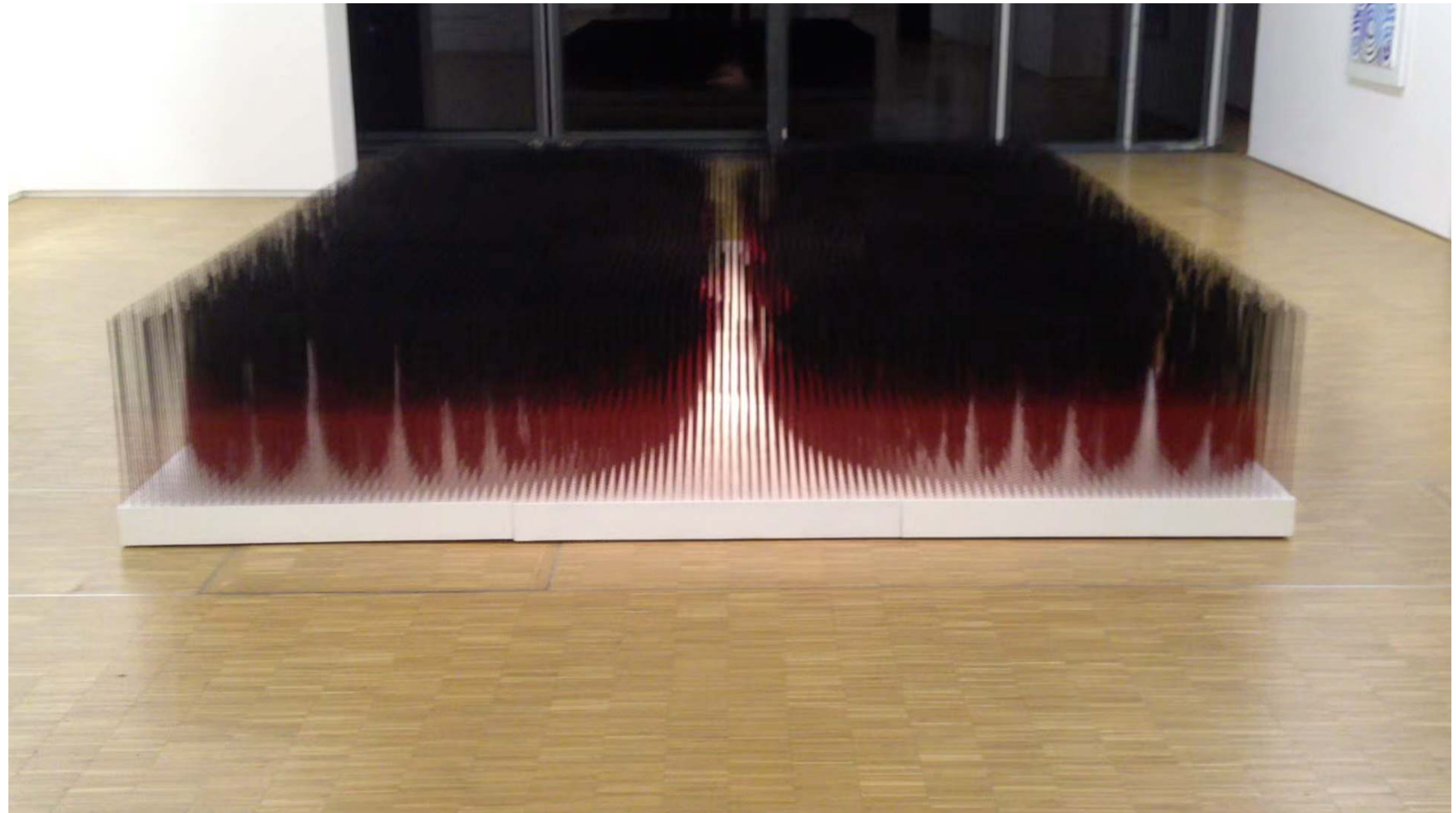
Eher
gewünschtes
Aussehen



Siemensstern



Moiré-Effekt in der Kunst



Centre
George
Pompidou

Komplexität der Rasterisierung von Linien

- Anzahl Operationen:

$$O(2 \cdot l + c) = O(l)$$

l = Länge der Linie in Pixel, c = Konstante für Setup

- Geht es schneller?
- Ja: Algorithmen, die nur Integer-Operationen enthalten!
- Noch schneller: Span-basierter Algorithmen [Henning & Stephenson, 2004]

$$O\left(\frac{l}{r}\right)$$

n = Anzahl Zellen (Pixel) auf dem Strahl, r = mittlere Span-Länge

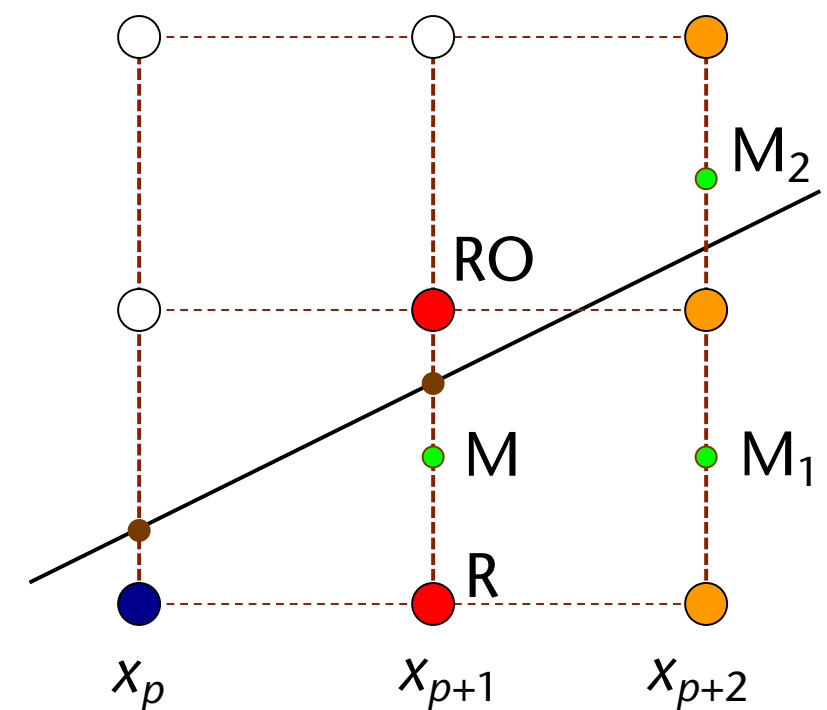
- Im Mittel: ca. Faktor 2 schneller, gemittelt über alle mögliche Orientierungen

FYI: Hintergrund-Infos zu Integer-Algos

- Es gibt Algorithmen mit reiner Integer-Arithmetik, sind auch inkrementell, und nur 1 arithmetische Integer-Operation pro Iteration!
- Ansatz:
 - Berechne **vorzeichenbehaftete** Abstände der Gitterpunkte von der Linie
 - Aktueller Gitterpunkt liegt oberhalb → gehe nach rechts; aktueller Gitterpunkt unterhalb → gehe nach recht oben
 - **Bresenham**: verwende Gitterpunkte "zwischen" Pixeln
 - **Midpoint- Algo**: Mittelpunkte der Pixel-Kanten (einfacher)
- Heutzutage auf der CPU wirklich schneller?

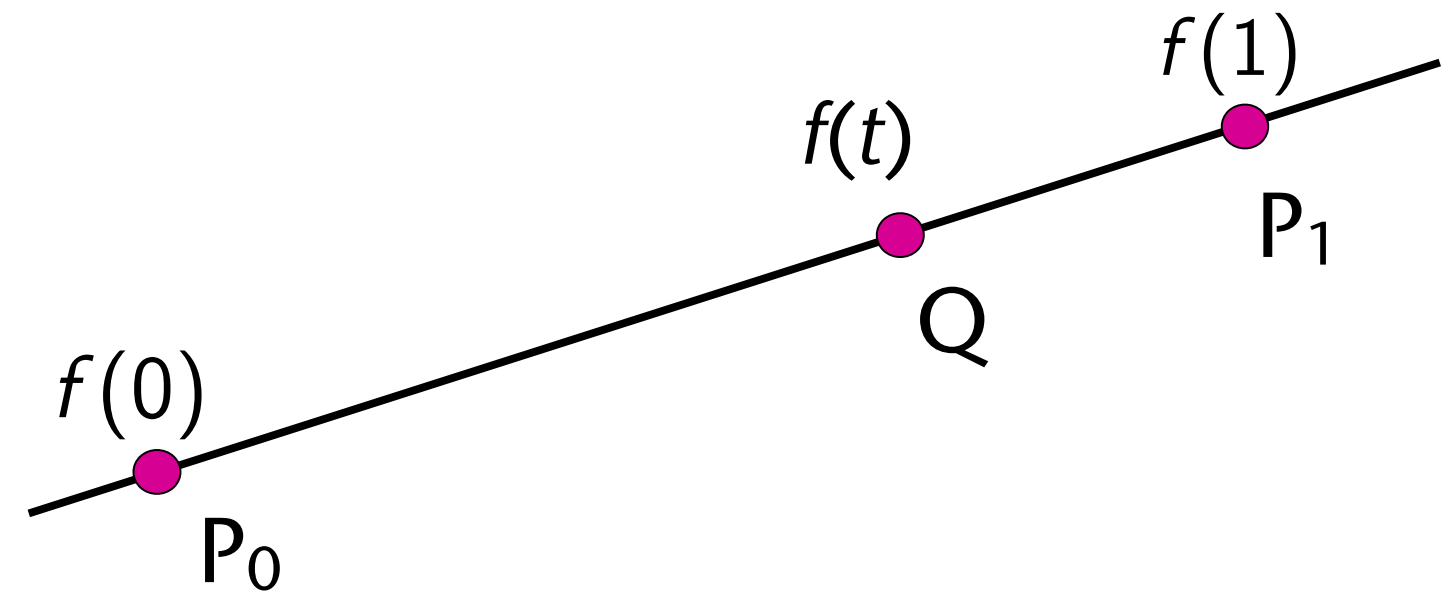


Clip from Bresenham's Keynote Talk at WSCG 2003



Interpolation von Attributen

- Häufig haben Eckpunkte weitere Attribute (außer der Position), z.B. verschiedene Farben
- Ziel: ein gleichmäßiger Farbverlauf f entlang der Linie
- Gesucht: $f(t)$ im Punkt Q
- Ansatz: lineare Interpolation
- Erinnerung: im 2D ist $t \in [0,1]$ gerade die normierte(!) Distanz zwischen P_1 und P_0

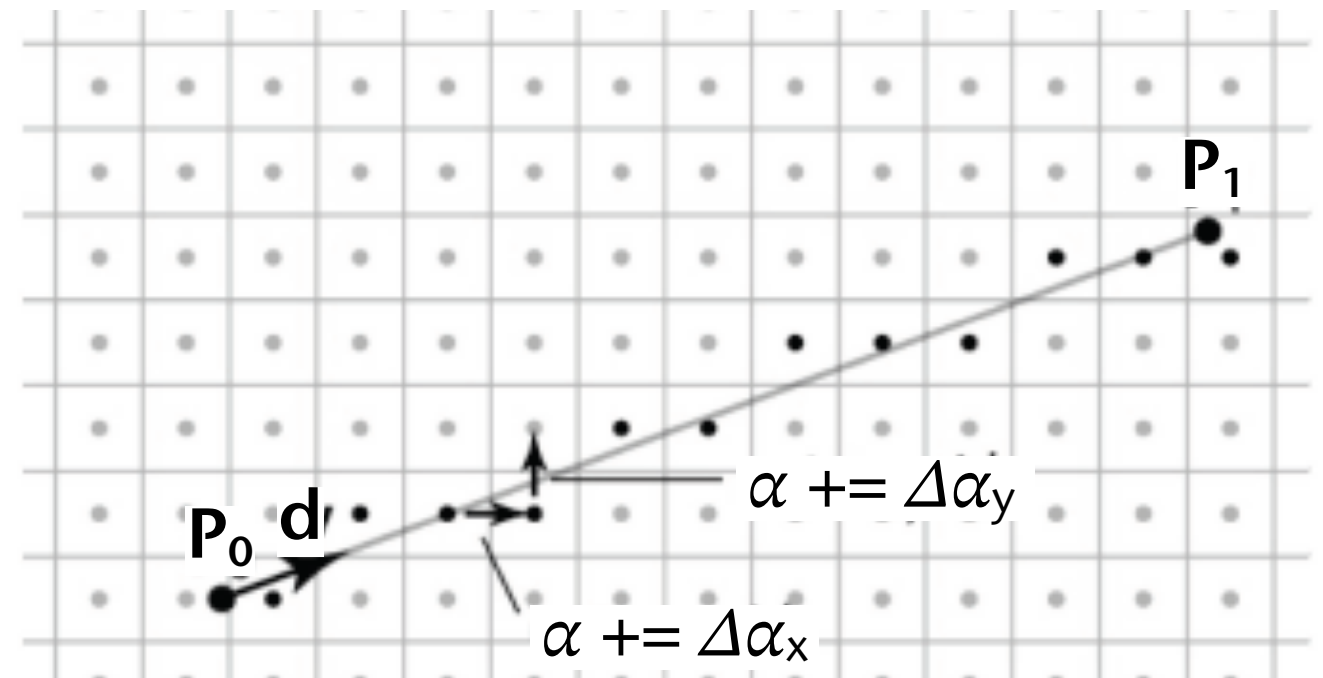
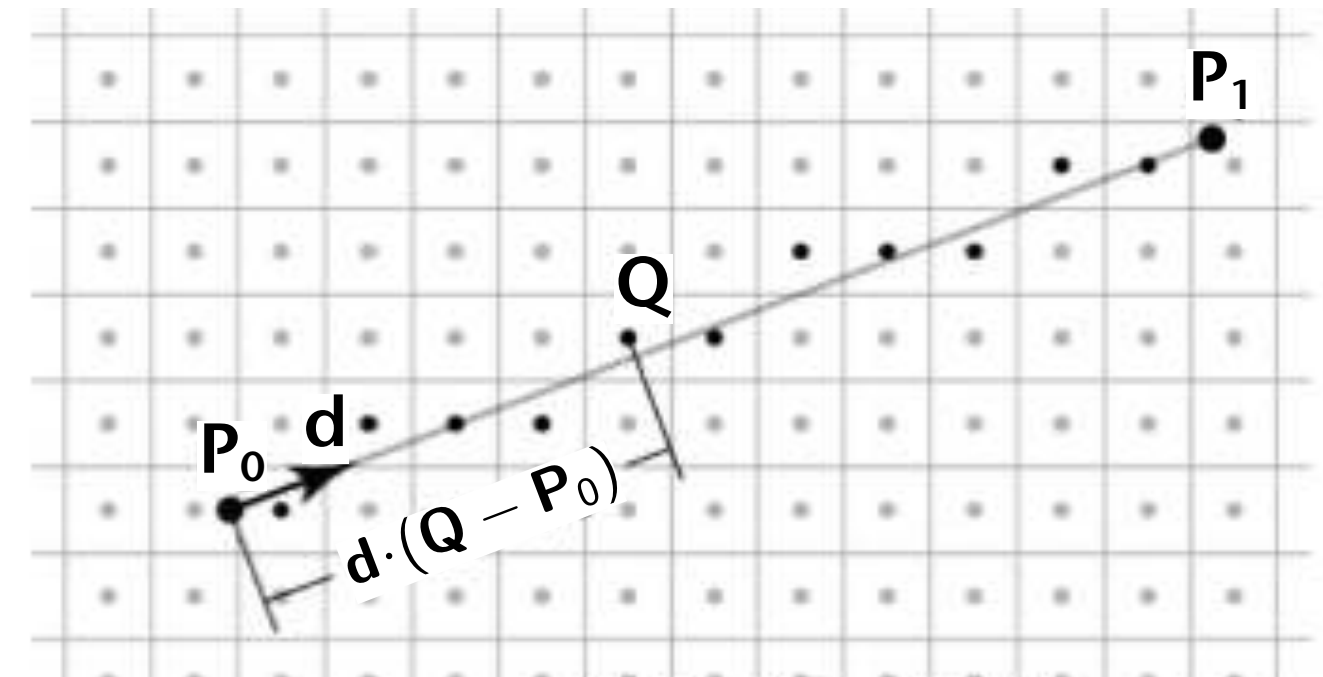


- Problem: die Pixel(-mittelpunkte) Q liegen i.A. *nicht* genau auf der Linie
- Definiere Funktion zur Projektion *auf* die Linie:

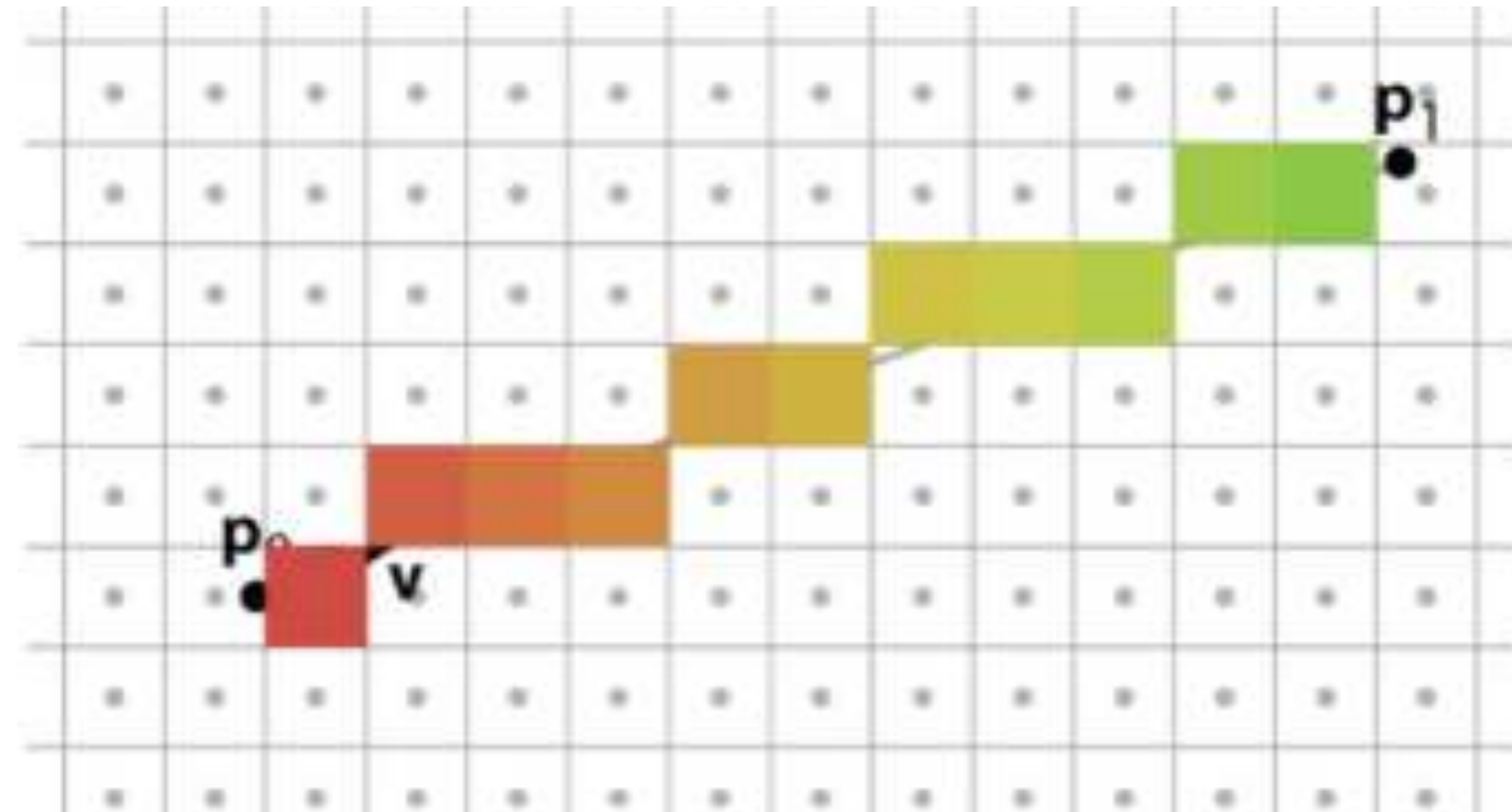
$$\mathbf{d} = \frac{\mathbf{P}_1 - \mathbf{P}_0}{\|\mathbf{P}_1 - \mathbf{P}_0\|} \quad \alpha = \frac{\mathbf{d} \cdot (\mathbf{Q} - \mathbf{P}_0)}{\|\mathbf{P}_1 - \mathbf{P}_0\|}$$

$$f(\alpha) = (1 - \alpha) \begin{pmatrix} r_0 \\ g_0 \\ b_0 \end{pmatrix} + \alpha \begin{pmatrix} r_1 \\ g_1 \\ b_1 \end{pmatrix}$$

- Beobachtung: $\alpha = \alpha(Q_x, Q_y) \rightarrow f$ ist *linear* in Q_x und Q_y
- Verwende inkrementelle Berechnung von f



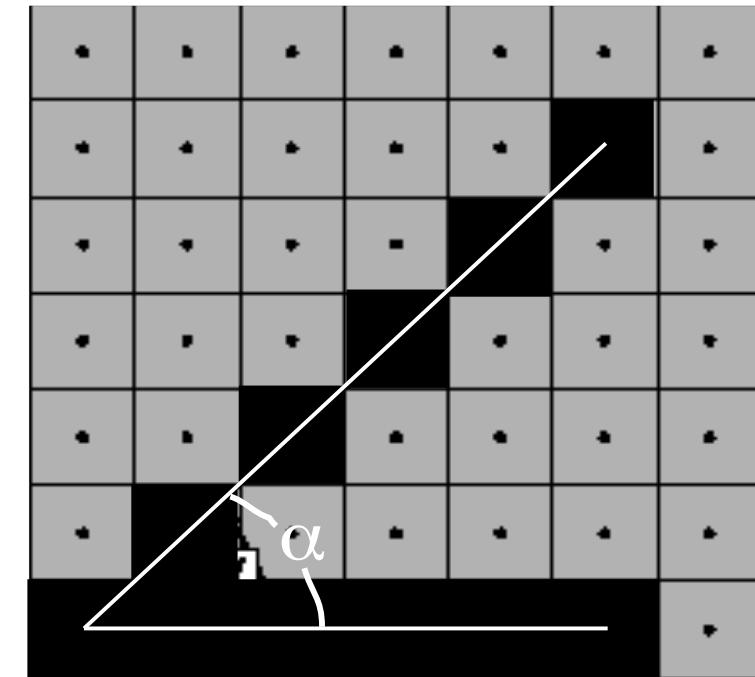
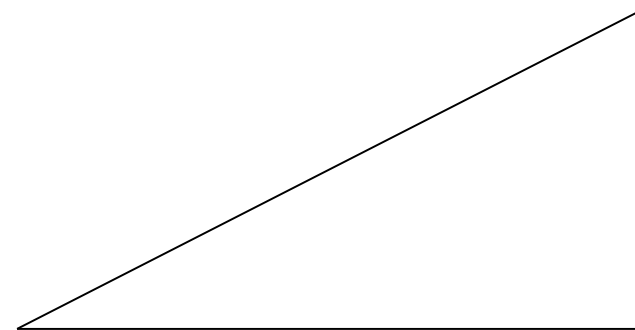
Resultat



Gleichmäßiger Helligkeitsverlauf

- Gewünscht: einheitliche Stärke und Helligkeit
- Bei gleicher Pixelzahl sind schräge Linien länger als horizontale
- Ändere Intensität der Linie gemäß der Steigung
- Skaliere den Grauwert um den Faktor

$$\cos(45^\circ - \alpha), \quad \alpha = 0^\circ \dots 45^\circ$$



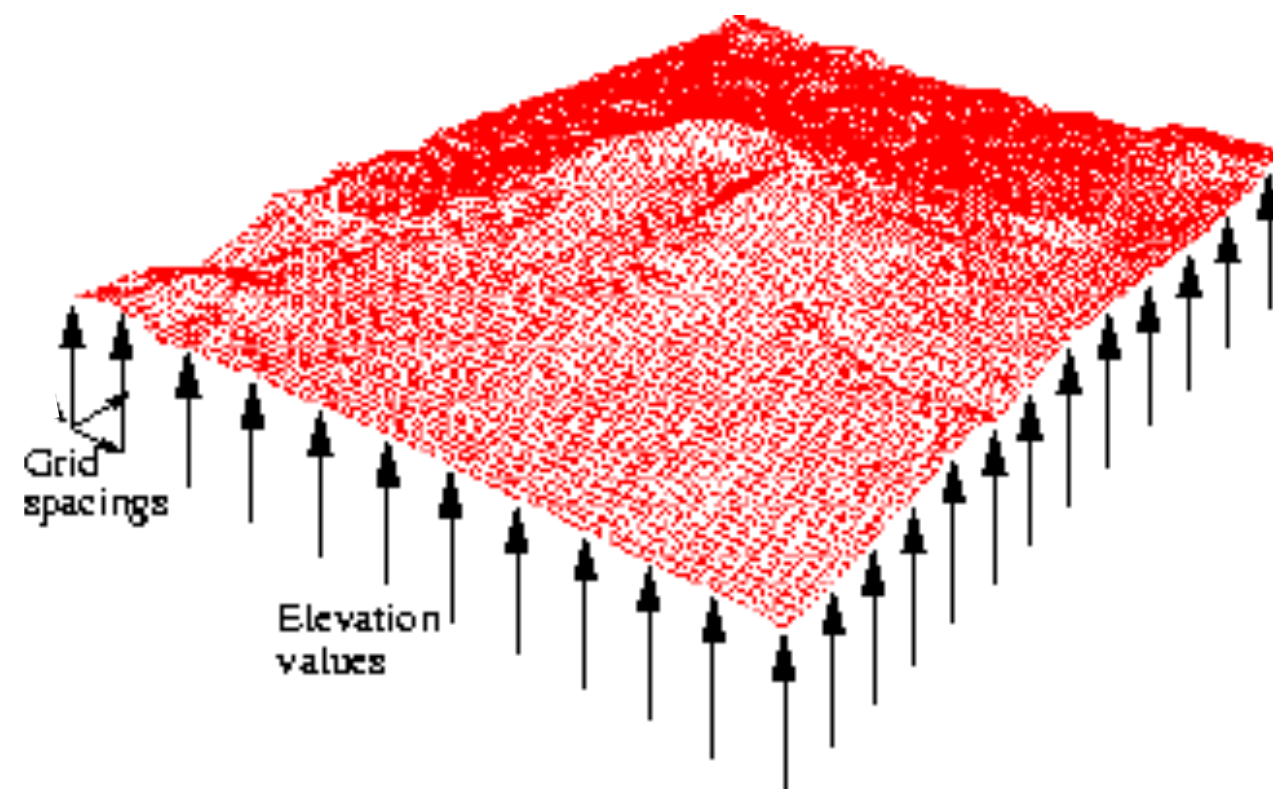
Ray-Tracing Height Fields

- Height Field = alle Arten von Flächen, die sich als eine Funktion

$$z = f(x, y)$$

schreiben lassen

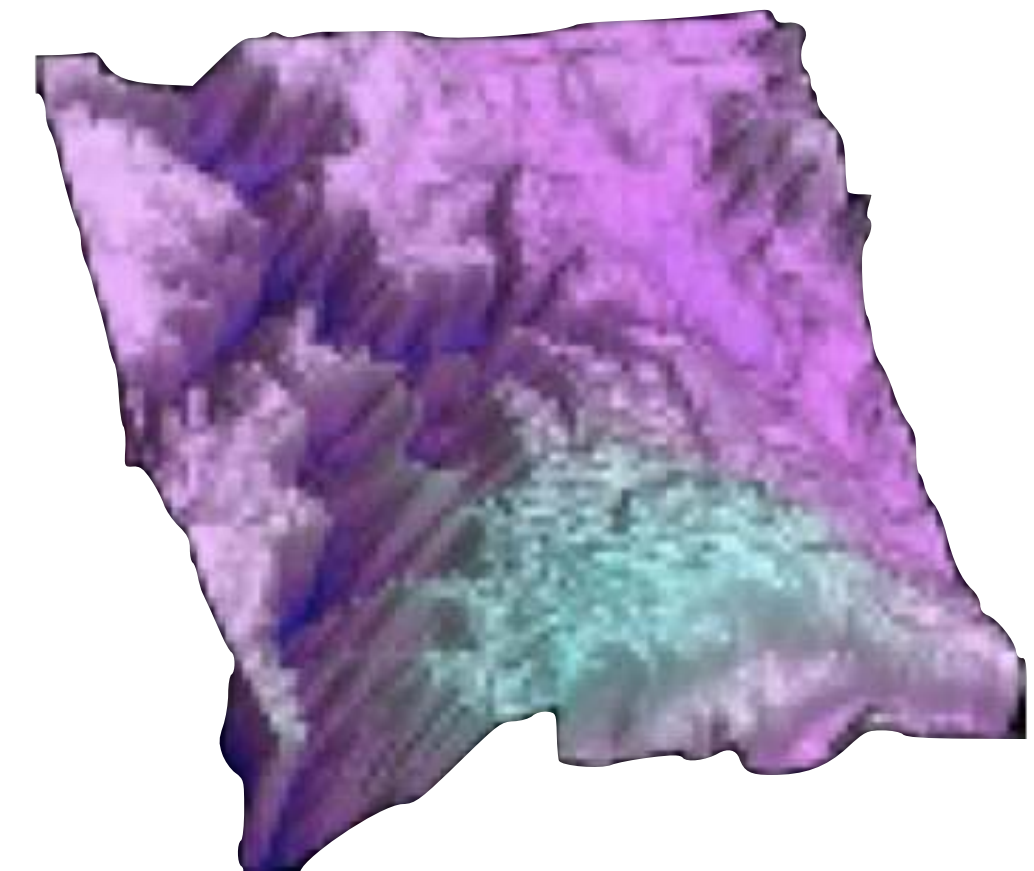
- Z.B.: Terrain, Meßwerte über einer Ebene, 2D-Skalarfeld, Reliefs, ...



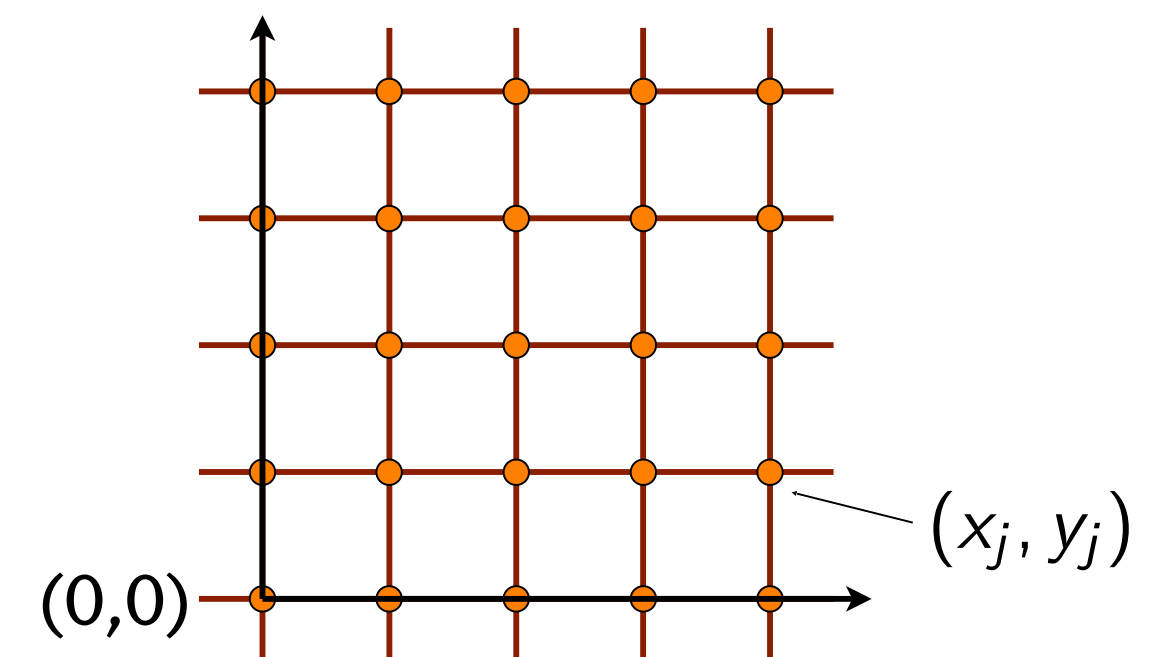
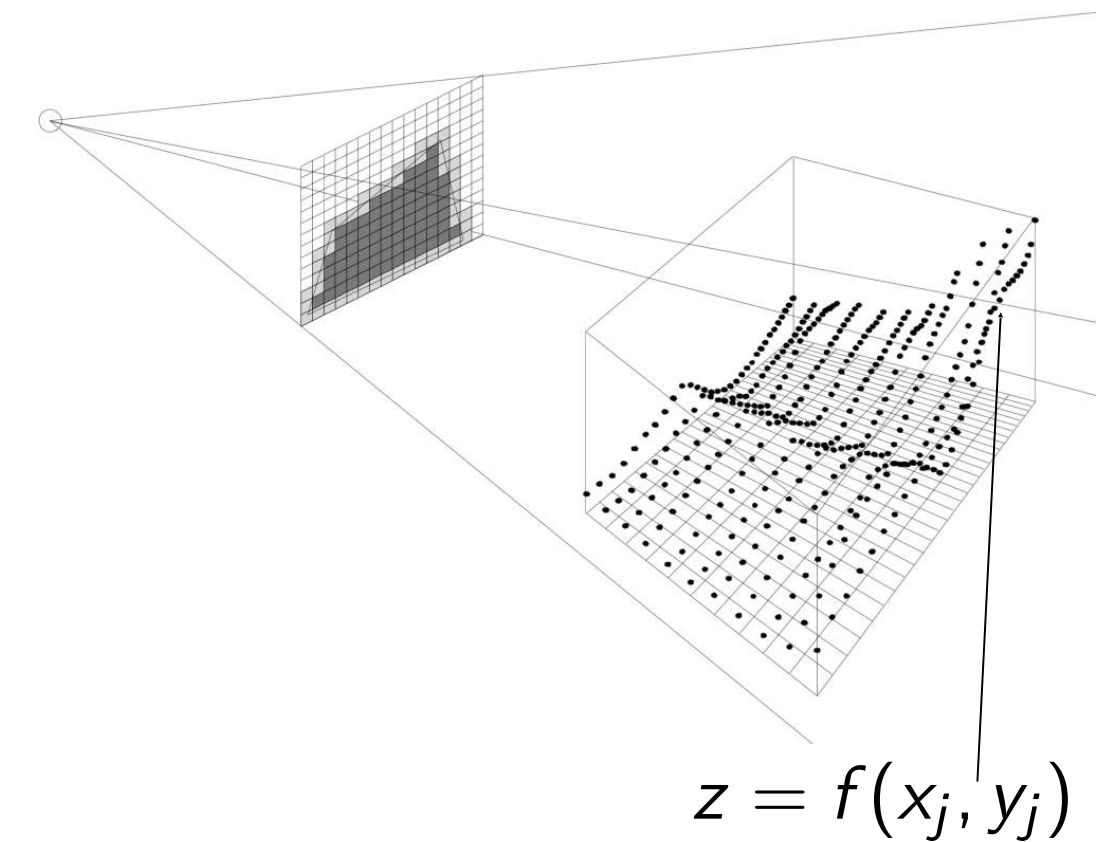
Height field (= Bitmap)



Rendered



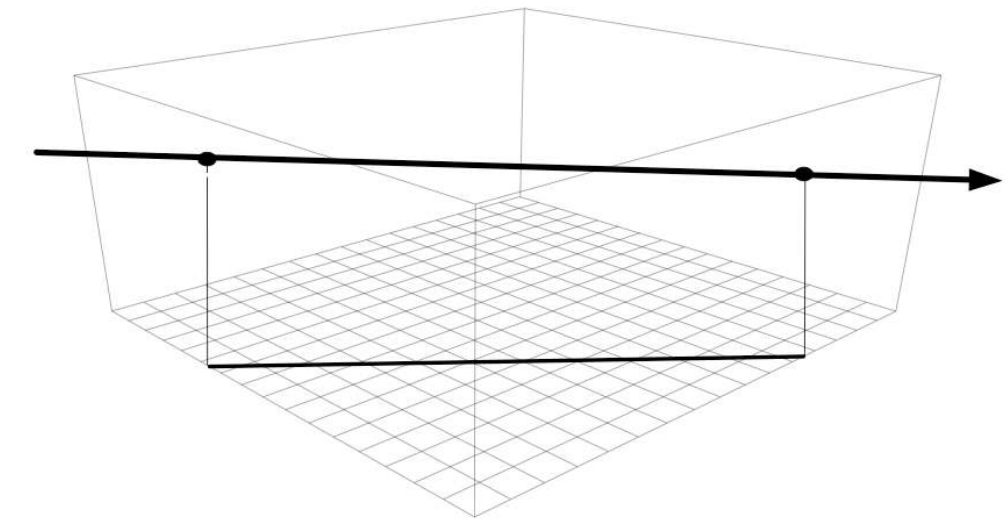
- Die naive Methode, ein Height-Field zu raytracen:
 - Konvertiere das $n \times n$ Feld in $2n^2$ Dreiecke, teste Strahl gegen jedes davon
 - Probleme: langsam, benötigt viel Speicher
- Ziel: direktes Ray-Tracing des Height-Fields aus dem 2D-Array
- Gegeben:
 - Strahl in 3D
 - Gitter $[0 \dots n] \times [0 \dots n]$ als Float-Array
 - Höhenwerte liegen auf den Gitterknoten vor



Das Verfahren

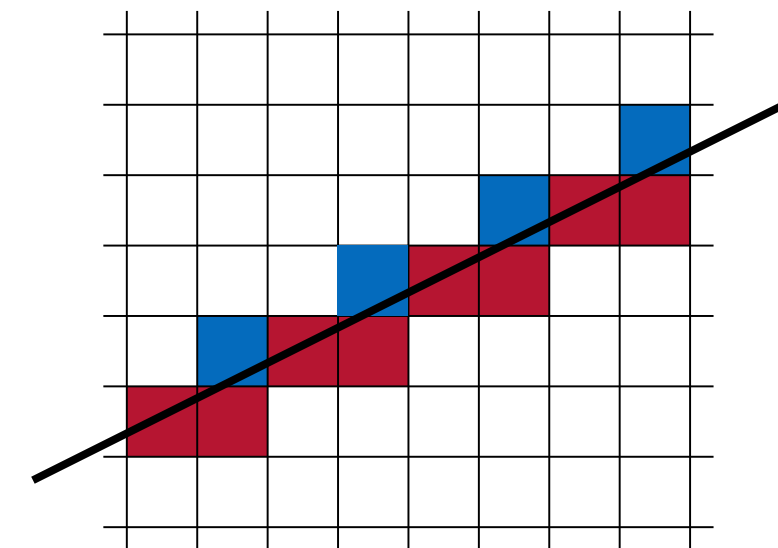
1. Dimensionsreduktion

- Projiziere Strahl in xy-Ebene

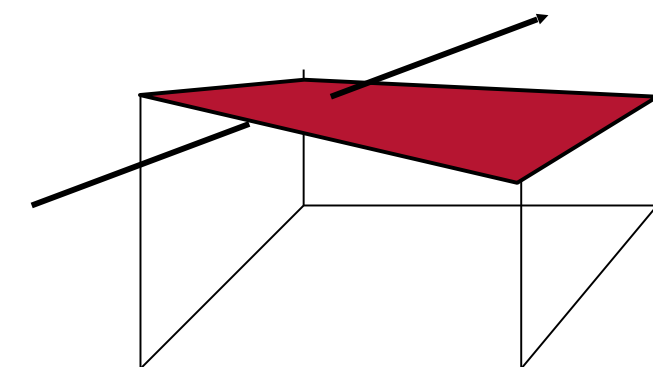


2. Alle Zellen der Reihe nach besuchen, die vom Strahl geschnitten werden (und nur diese)

- Wie Scan-Conversion von Linien, hier auch mit den "blauen" Zellen

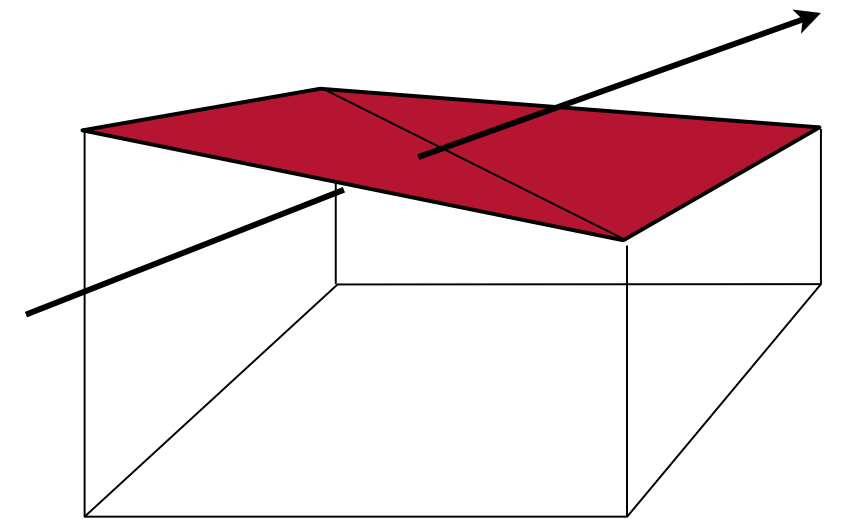
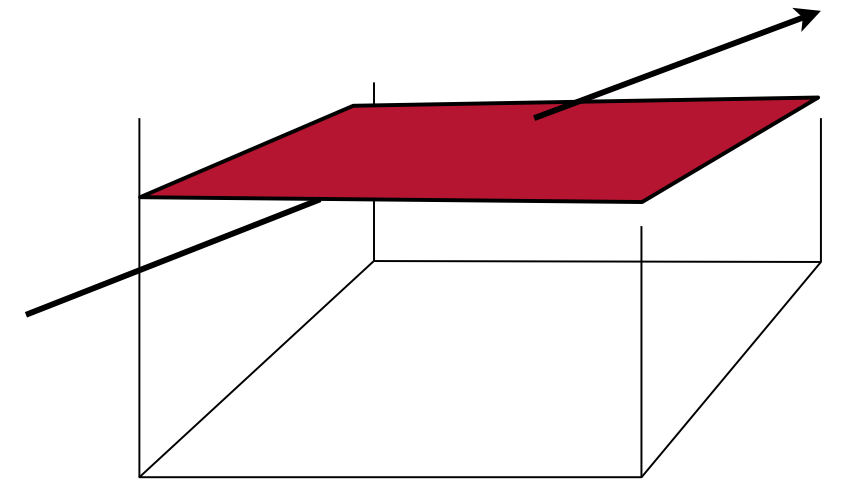


3. Strahl testen gegen das Flächenstück, das von den 4 Höhenwerten an den Ecken aufgespannt wird



Schnittest Strahl—Flächenstück in der Zelle

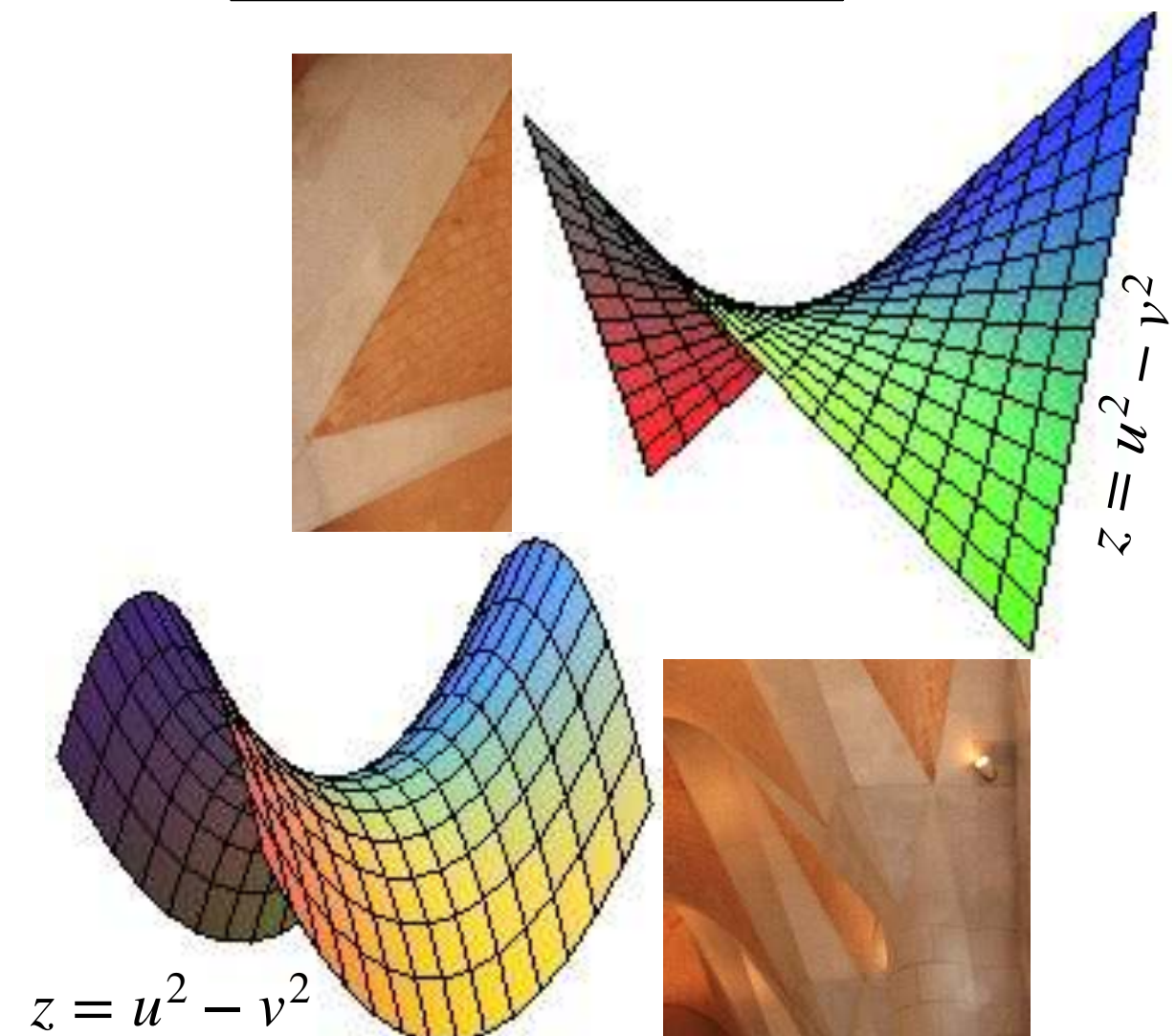
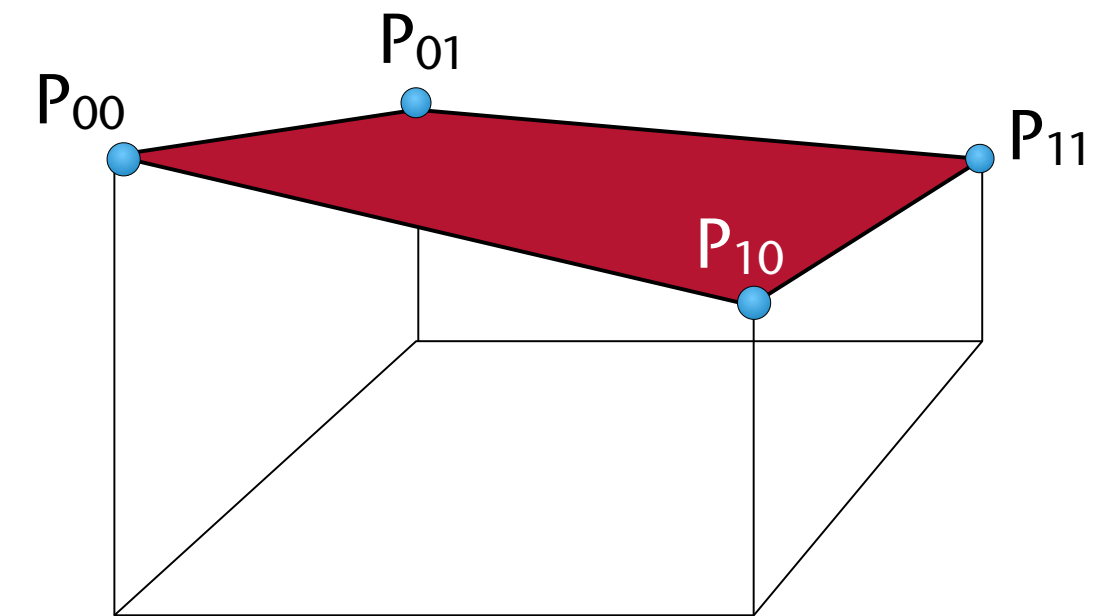
- Naive Methode "piecewise constant heightfield":
 - Bestimme die mittlere Höhe aus den 4 Höhenwerten an den Ecken
 - Schneide Strahl gegen horizontales Quadrat mit dieser mittleren Höhe
 - Sehr ungenau
- Naive Methode "2 Dreiecke":
 - Konstruiere 2 Dreiecke aus den 4 Punkten über den Ecken
 - Knick innerhalb der Zelle
 - Aufteilung in Dreiecke nicht eindeutig



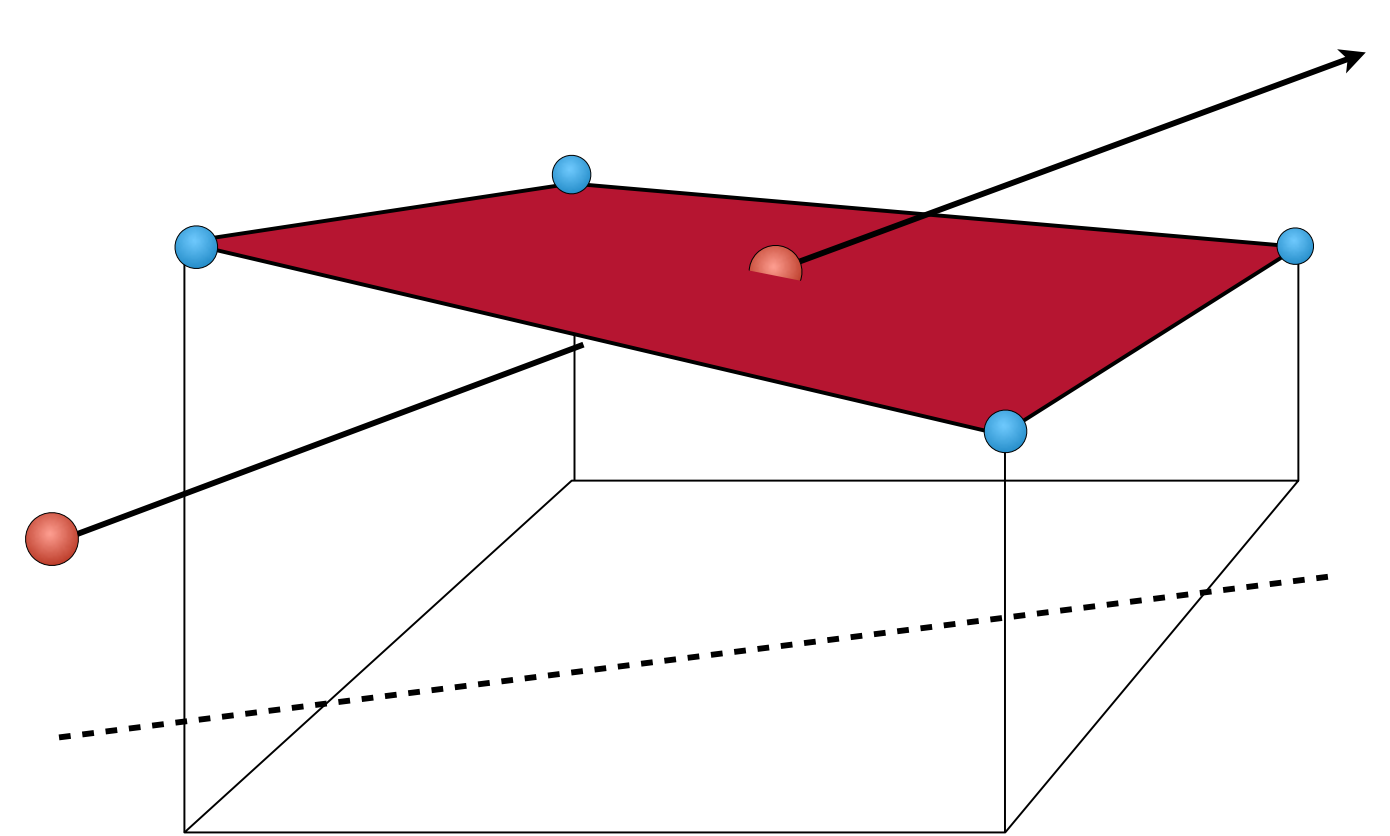
Bilineare Interpolation

- Betrachte Fläche, die entsteht, wenn man eine Gerade durch die Punkte P_{00} und P_{01} legt, und dann auf beiden Seiten gleichzeitig Richtung P_{10} und P_{11} gleiten lässt
 - Konstruktion heißt "Regelfläche" (*ruled surface*)
 - Bezeichnung: parabolisches Hyperboloid
- O.B.d.A. $P_{00} = (0, 0, h_{00})$, $P_{10} = (1, 0, h_{10})$, etc.
- Gleichung der Fläche:

$$h(u, v) = (1 - v) \left[(1 - u) \cdot h_{00} + u \cdot h_{10} \right] + v \left[(1 - u) \cdot h_{01} + u \cdot h_{11} \right]$$



Sagrada Familia, Barcelona (Gaudi)



Lösung mit bilinearer Interpolation

$$Q(t) = Q + t \cdot \vec{r} = \begin{pmatrix} q_x \\ q_y \\ q_z \end{pmatrix} + t \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix}$$

Bei bestimmten t hat die Linie die (u,v) -Koord $\begin{pmatrix} u(t) \\ v(t) \end{pmatrix} = \begin{pmatrix} q_x + t r_x \\ q_y + t r_y \end{pmatrix}$ erreicht, und die "Höhe" $z(t) = q_z + t r_z$.

Gesucht ist t^* , bei dem

$$z(t^*) = h(u(t^*), v(t^*)) \quad (1)$$

Näherung: $h(u,v)$ umformen

In (1) einsetzen:

$$\begin{aligned} q_z + t r_z &= h(q_x + t r_x, q_y + t r_y) \\ &= h_{00} + (q_x + t r_x) h_u + (q_y + t r_y) h_v \\ &\quad + (q_x + t r_x)(q_y + t r_y) h_{uv} \\ &= h_{uv} r_x r_y t^2 + (\dots) \cdot t + (\dots) \end{aligned}$$

$\Rightarrow$ quadratische Gleichung in $t \Rightarrow t_{1/2}$

Fälle: keine Lsg

$$t_1 = t_2 \Rightarrow (\text{Tangente})$$

$$t_1 < t_2 < 0 \Rightarrow (\text{obdA } t_1 < t_2, \text{ Schn. pkt vor } Q)$$

$$t_1 < 0 < t_2 \Rightarrow (\text{Schn. pkt } Q(t_2), t^* = t_2)$$

$$0 < t_1 < t_2 \Rightarrow \text{Schn. pkt } Q(t_1), t^* = t_1$$

$$(u(t^*), v(t^*)) \notin [0,1]^2 \Rightarrow \text{Schn. pkt außerhalb des Quadrates}$$

Näherung:

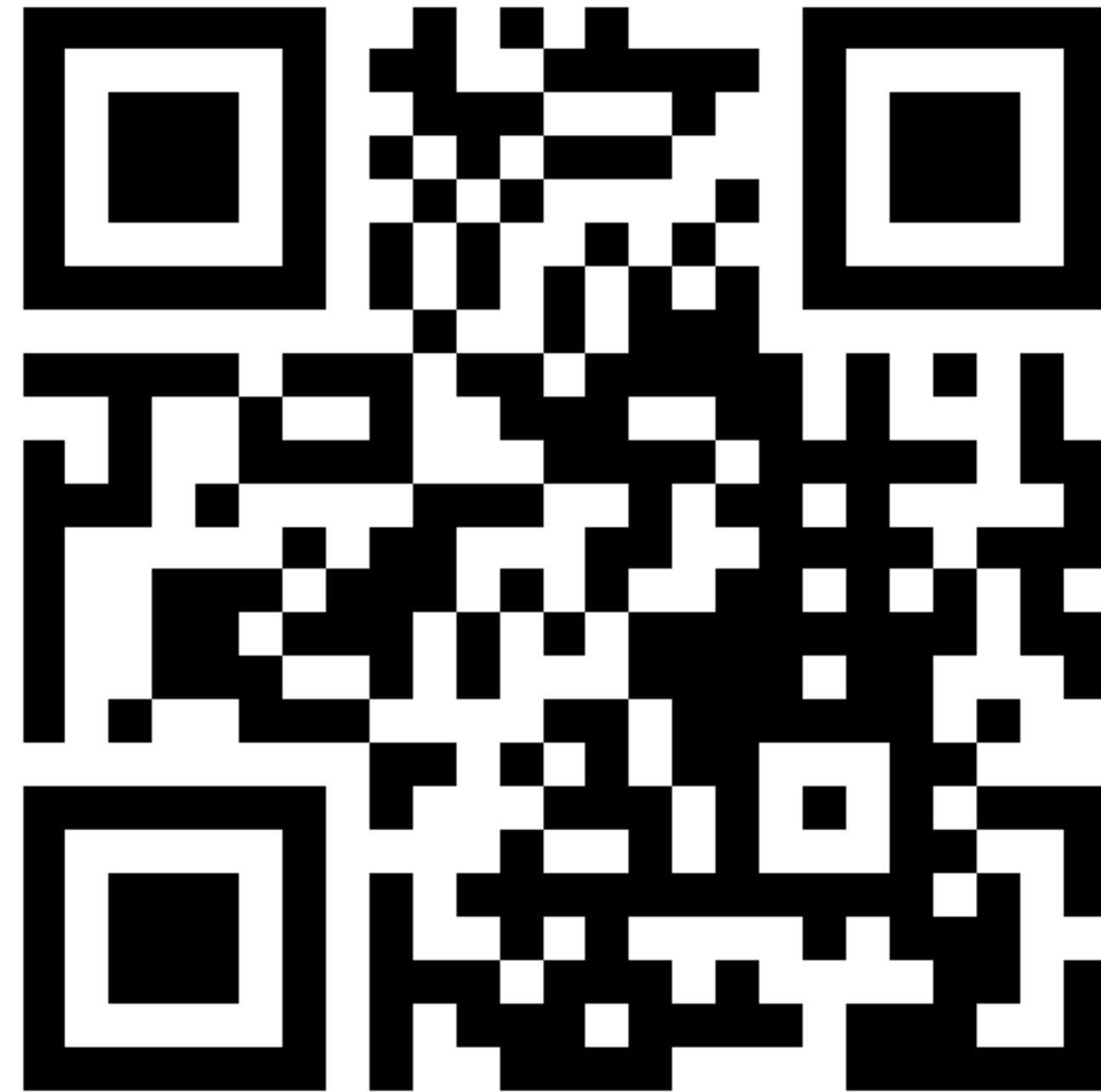
$$(1-v) \left[h_{00} + u(h_{10} - h_{00}) \right] + v \left[h_{01} + u(h_{11} - h_{01}) \right]$$

$$\begin{aligned} &= 1 \cdot [h_{00} + u(h_{10} - h_{00})] + \\ &\quad v \cdot ([h_{11} + u(h_{11} - h_{11})] - [h_{00} + u(h_{10} - h_{00})]) \end{aligned}$$

$$\begin{aligned} &= h_{00} + u h_{10} (h_{10} - h_{00}) \\ &\quad + v(h_{01} - h_{00}) + u \cdot v \cdot [(h_{11} - h_{01}) - (h_{10} - h_{00})] \end{aligned}$$

$$= h_{00} + u h_u + v h_v + u \cdot v \cdot h_{uv}$$

Welche Fälle gibt es bei/nach der Berechnung des/der t's ?



<https://www.menti.com/ojndv9fafk>

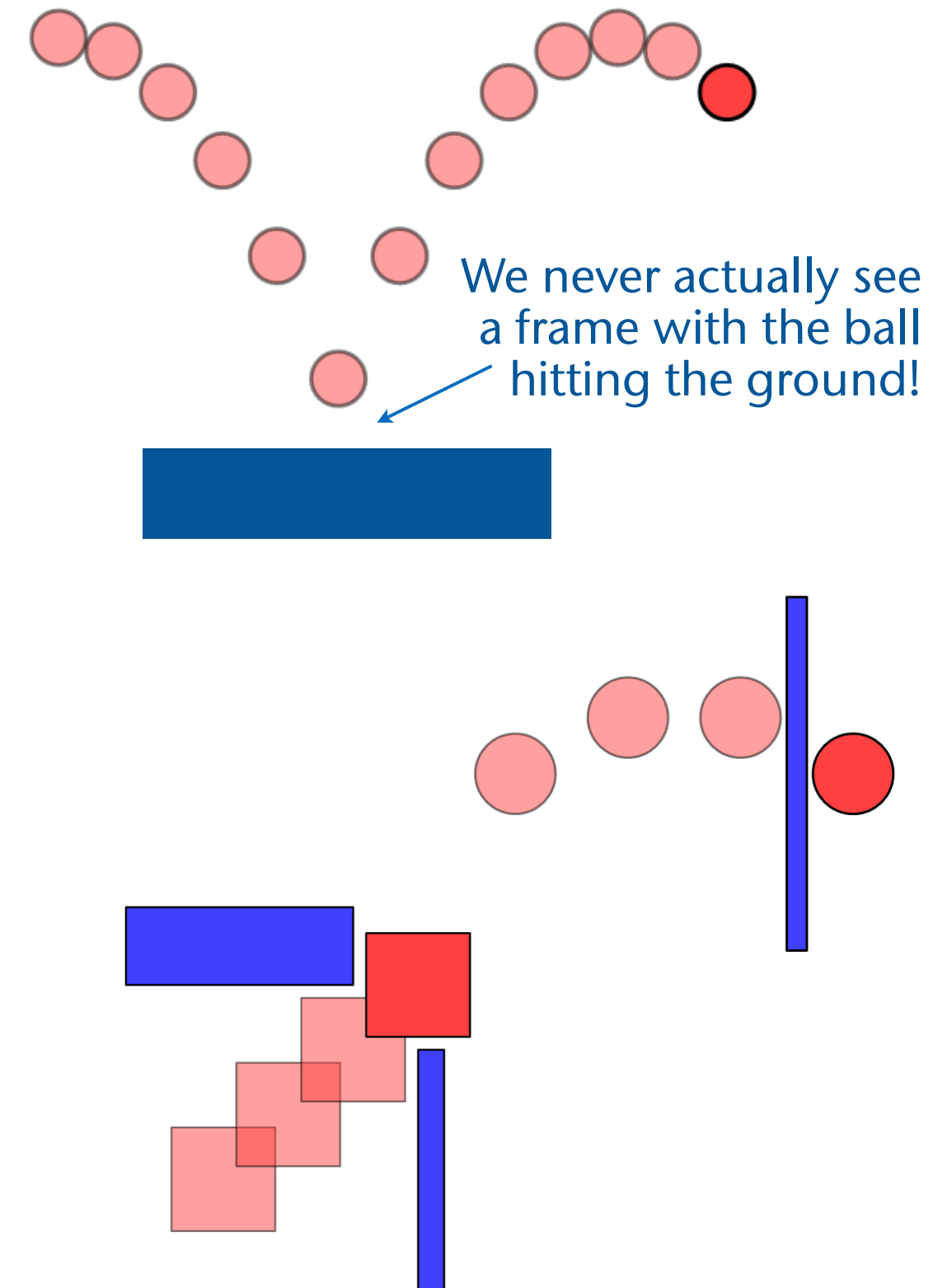


Valles Marineris, Mars - <http://mars.jpl.nasa.gov>

Anwendungsfall: Continuous Collision Detection

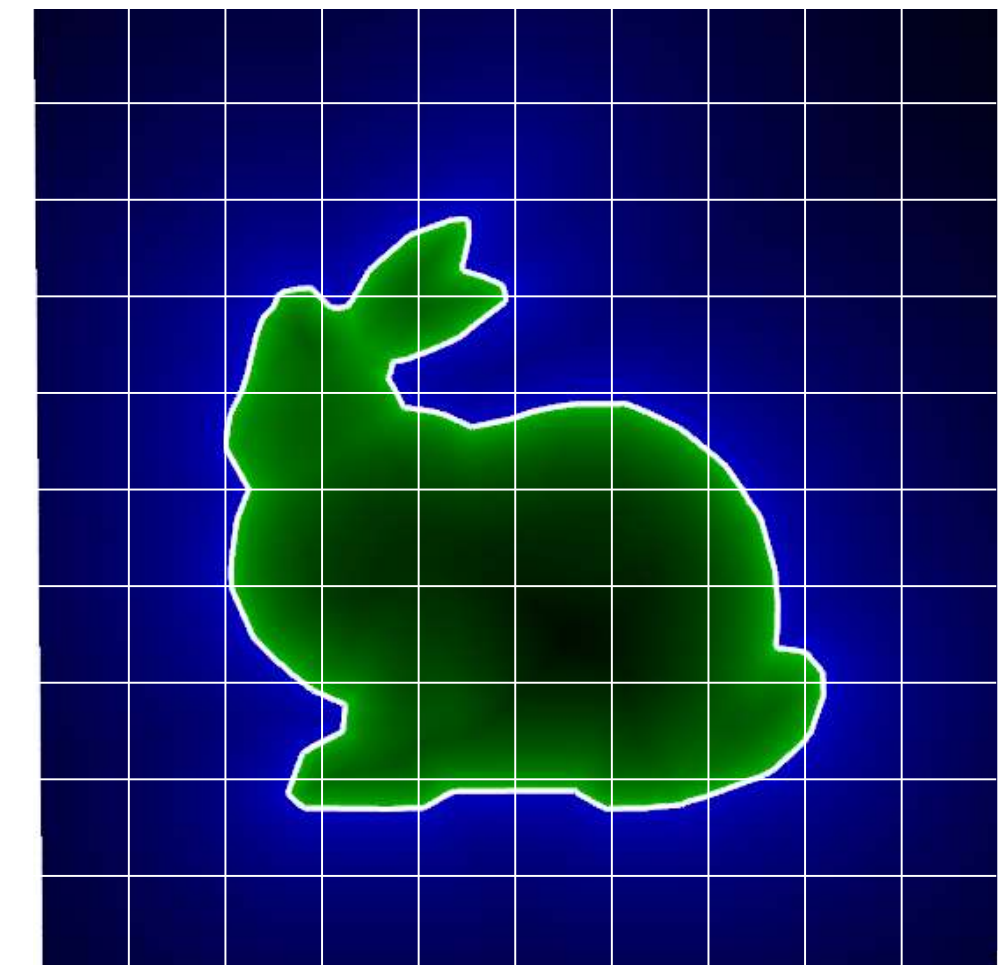
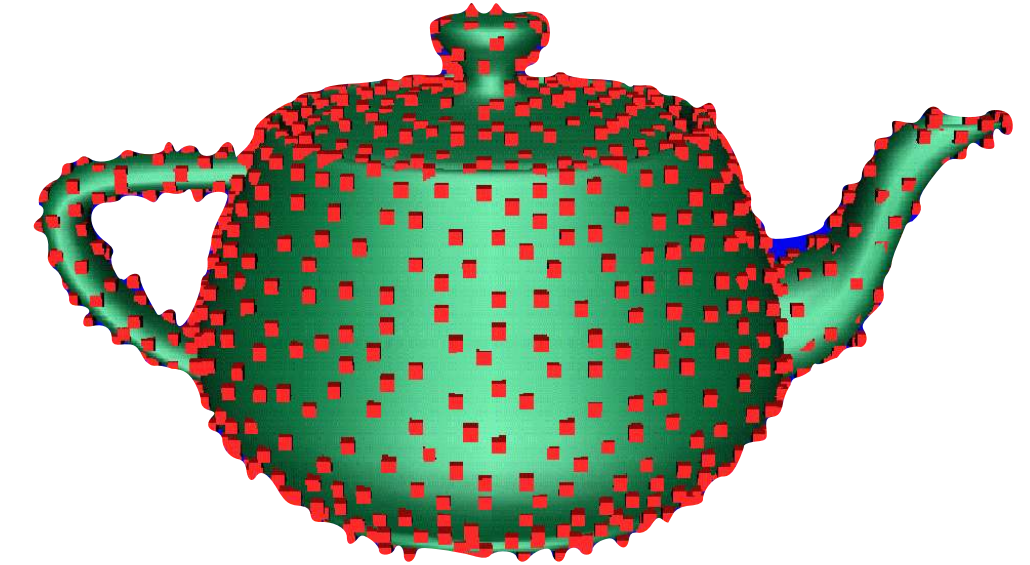
[Xu, Barbic 2014]

- Probleme diskreter Collision Detection:
 - Diskretisierung der Zeit
 - Tunneling
- Limits für min. Objekt-Größe, max. Geschwindigkeit und min. Framerate helfen nur bedingt
- Problemstellung bei CCD:
 - Gegeben zwei Objekte A, B, bewegt durch eine Simulation
 - Annahme: zum Zeitpunkt t_1 sind $A(t_1), B(t_1)$ kollisionsfrei
 - Bestimme, ob im Zeitintervall $[t_1, t_2]$ eine Kollision vorliegt und, falls ja, den frühesten Zeitpunkt $t \in [t_1, t_2]$, zu dem sie sich gerade "berühren"



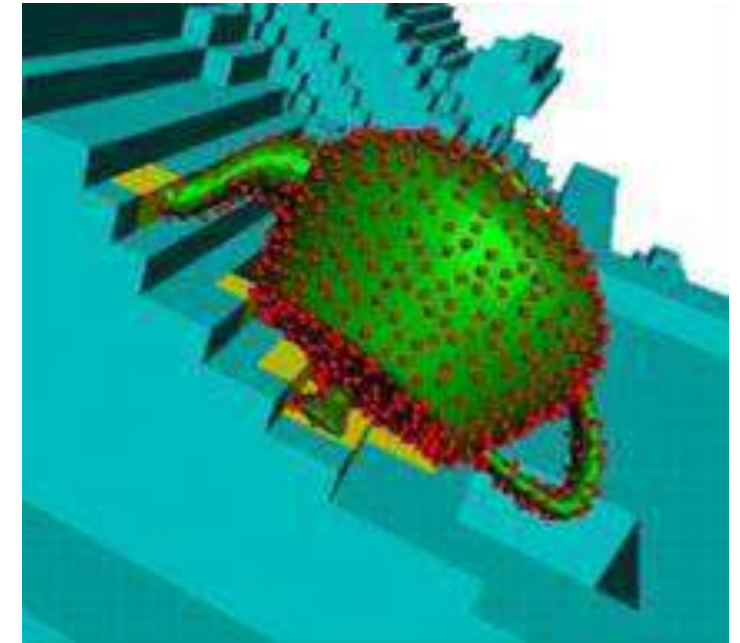
Das Voxmap-Pointshell-Verfahren

- Eine Lösung für Collision Detection
- Pointshell = Sampling der Oberfläche mit Points
- Voxmap:
 - 3D Gitter für die statische Szene, Zelle = "Voxel"
 - Einfachster Fall: 1 = innerhalb, 0 außerhalb
 - Besser: **signed distance field** = Gitterknoten (nicht Mittelpunkte!) speichern Distanz von Gitter-Punkt zu nächstem Punkt auf Oberfläche, positiv/negativ = außerhalb/innerhalb



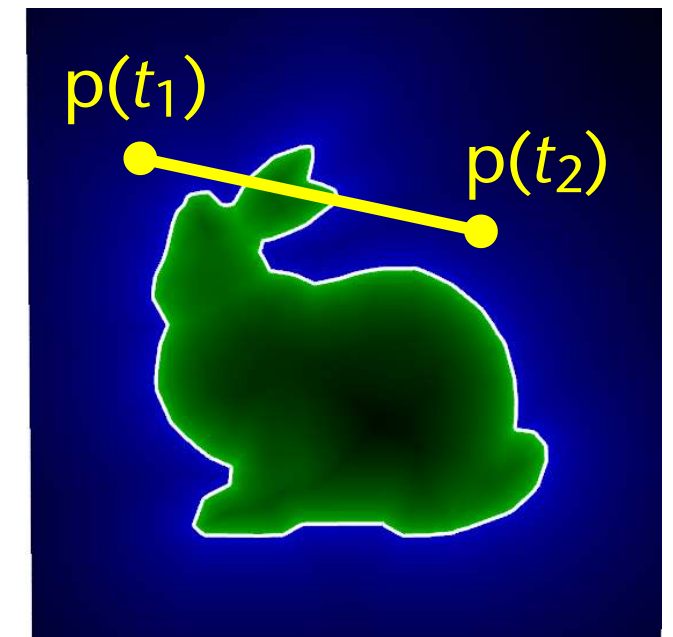
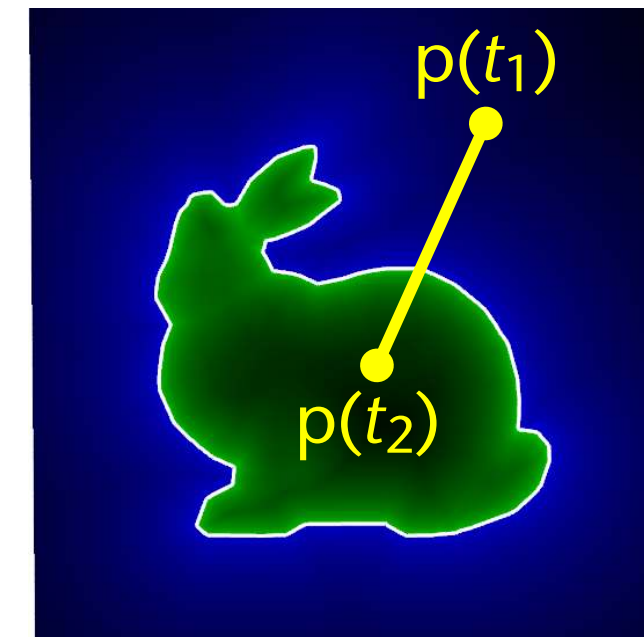
- Discrete collision detection:

```
for all points p in A:
    check voxel(p) in B
```



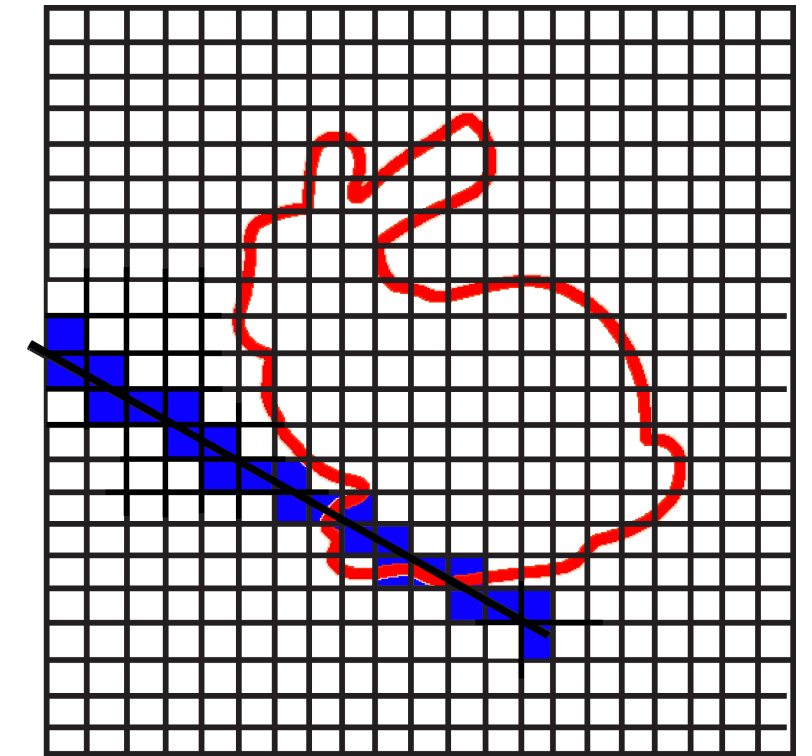
- Continuous collision detection → test line segments

```
for all points p in A:
    check whether line segment  $[p(t_1), p(t_2)]$ 
    goes from pos. to neg. distance ...
    .. somewhere in-between
```



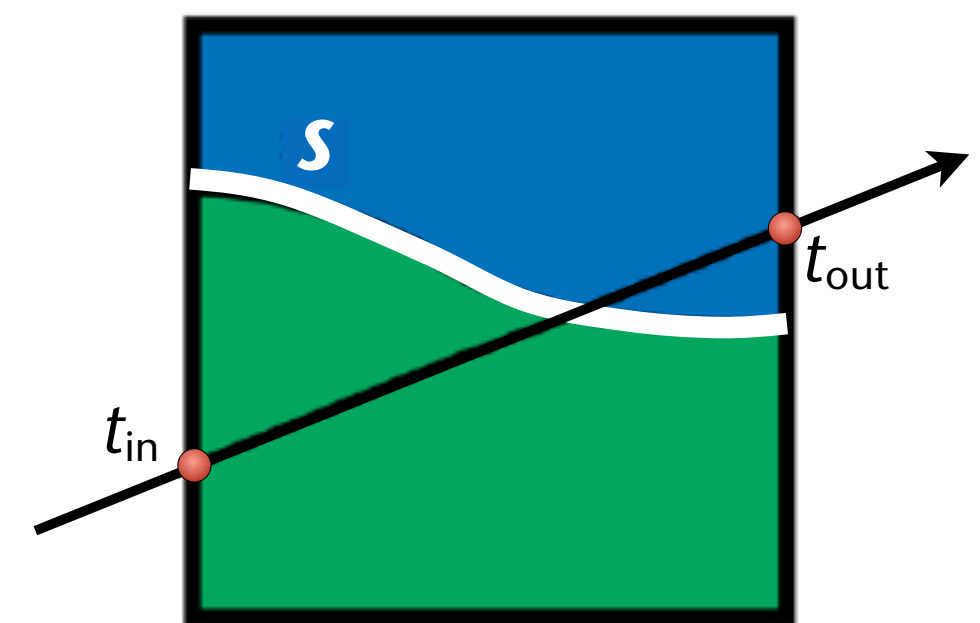
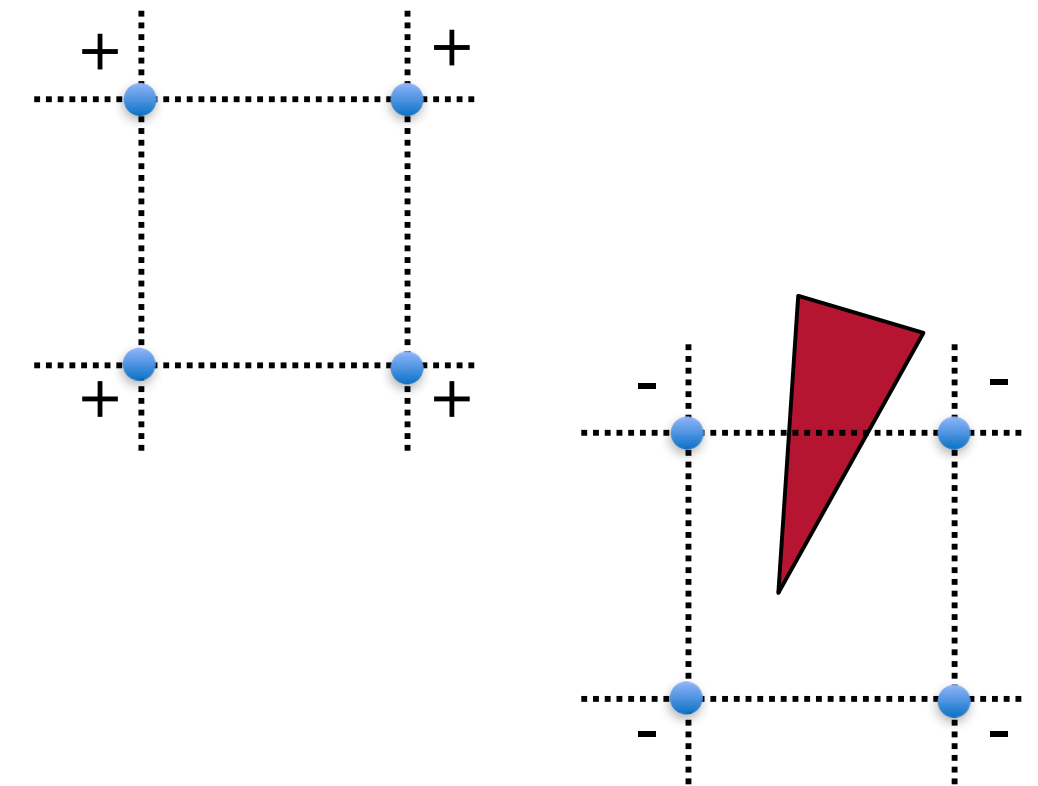
- Caveat: es genügt nicht, die Voxel $p(t_1)$ und $p(t_2)$ zu testen!

- Verfahren:
 1. Besuche alle Voxel, die vom Liniensegment getroffen werden
 2. In jedem Voxel: teste auf Null-Durchgang
- Zu 1:
 - Einfache Erweiterung des Linien-Scanline-Algorithmus' auf 3D Gitter
 - Berechne Folge von t 's für Schnitte mit xy-, xz-, und yz-Ebene
 - Mache Schritt zu Nachbarzelle mit "nächstem" t

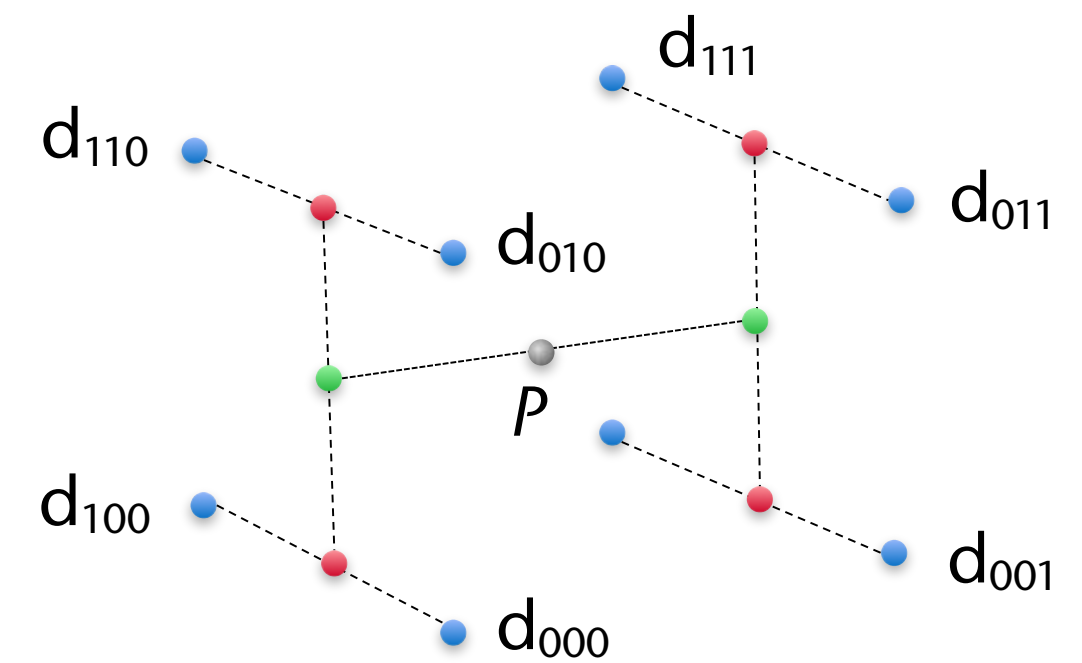
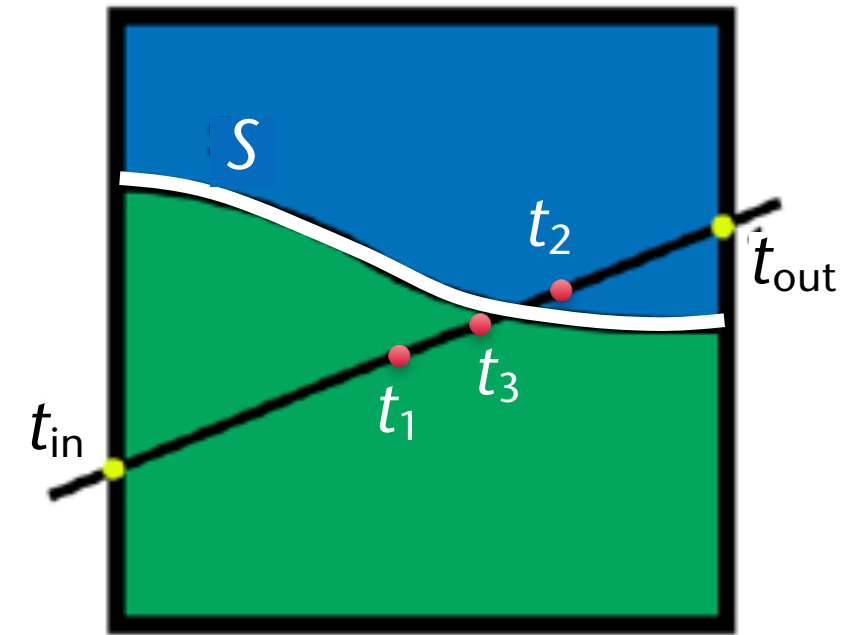


- Zu 2:
 - Erstes Ausschlusskriterium: alle Voxel-Ecken sind positiv, oder alle Ecken negativ
 - Verpasst zwar Details – die aber im Distanzfeld schon verloren gegangen sind
 - Zweites Ausschlusskriterium:

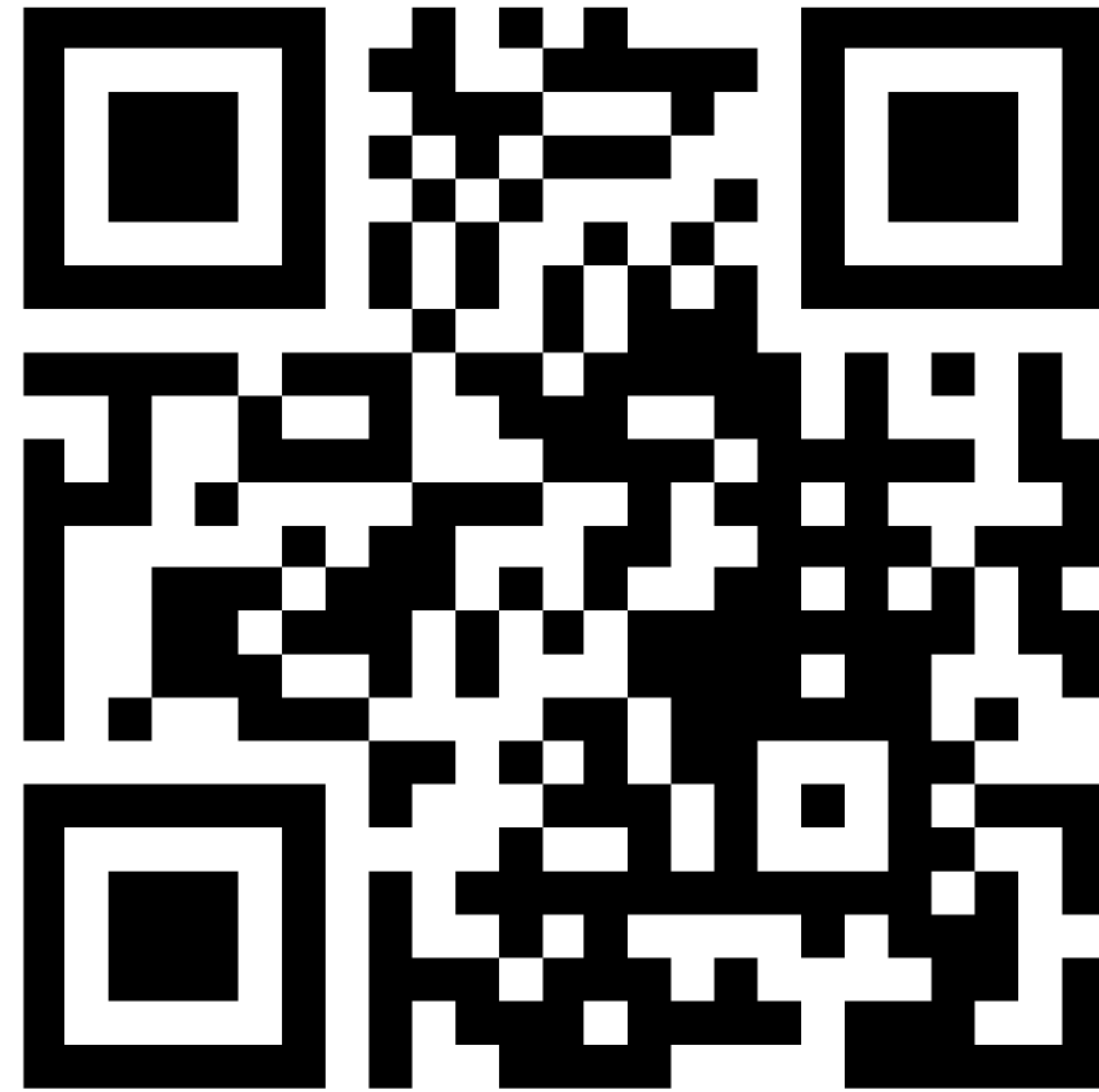
$$\text{sign}(P(t_{\text{in}})) = \text{sign}(P(t_{\text{out}})) \Rightarrow \text{no intersection}$$
 - Verpasst weitere "ungünstige" Fälle (welche?)



- Falls beide Vortests passiert wurden → Nullstellensuche: gesucht wird t mit $d(P(t)) = 0$
 - Z.B. mittels Intervall-Halbierung
 - Könnte dadurch auch einen Null-Durchgang verpasst werden?
- Aufgabe: Distanzwerte $d(P)$ für Punkte im Inneren eines Voxels berechnen (z.B. Testpunkte auf der Geraden)
- Lösung: **trilineare Interpolation**
 - OBdA: Voxel = Einheitswürfel $[0,1]^3$
 - Distanzwerte an den Ecken = d_{abc} , $a,b,c \in \{0,1\}$
 - Gesucht: d_{xyz} = Distanz am Punkt $P = (x, y, z)^T$



Soll ich die trilineare Interpolation vorrechnen?



<https://www.menti.com/ojndv9fafk>

Trilineare Interpolation

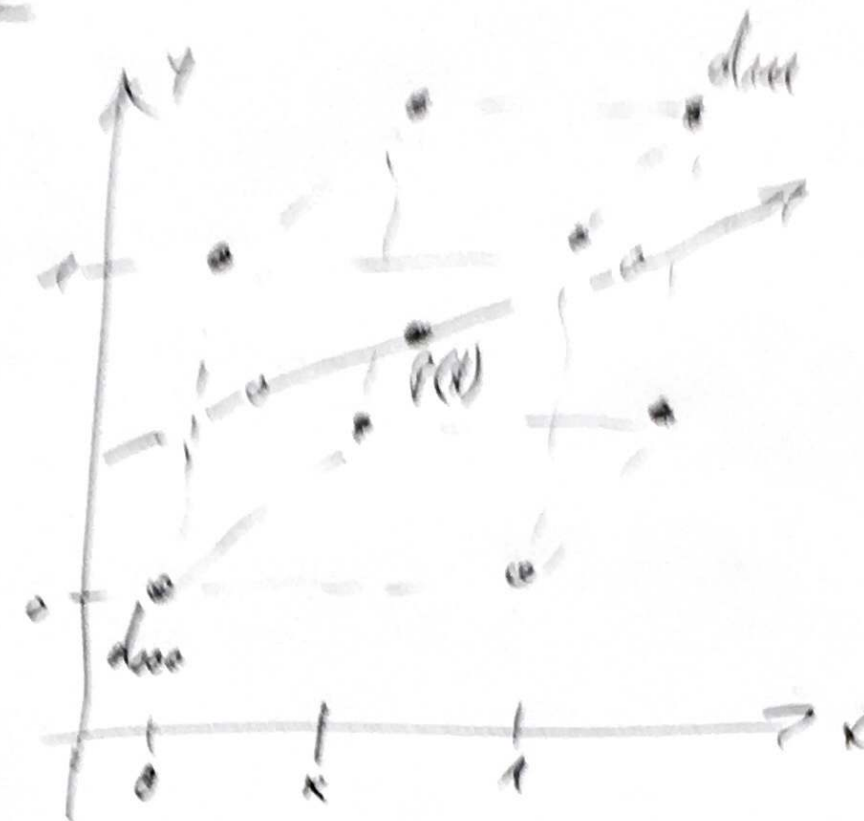
Trilineare Interpolation

$d_{abc} \in \mathbb{R}^3$ Diskretwerte
 $a, b, c = 0, 1$

$$t = t(x) = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

gültig für

$$\text{valid: } x, y, z \in [0, 1]$$



$$d_{xbc} = (1-x)d_{abc} + x \cdot d_{1bc} \quad , b=0,1, c=0,1 \quad | 4x$$

$$d_{xyz} = (1-y)d_{x0c} + y \cdot d_{x1c} \quad , c=0,1 \quad | 2x$$

$$d_{xyz} = (1-z)d_{xy0} + z \cdot d_{xy1} \quad | 1x$$

Bemerkungen

- Die Funktion $d(P(t))$ ist ein Polynom vom Grad 3 $\rightarrow$ löse mit analytischer Formel
- Wir haben hier – als Nebenprodukt – eine Fläche S "implizit" definiert:

$$S = \{X \mid F(X) = 0\}$$

- Solche Flächen heißen oft **implizite Flächen**, oder **Iso-Surfaces**

