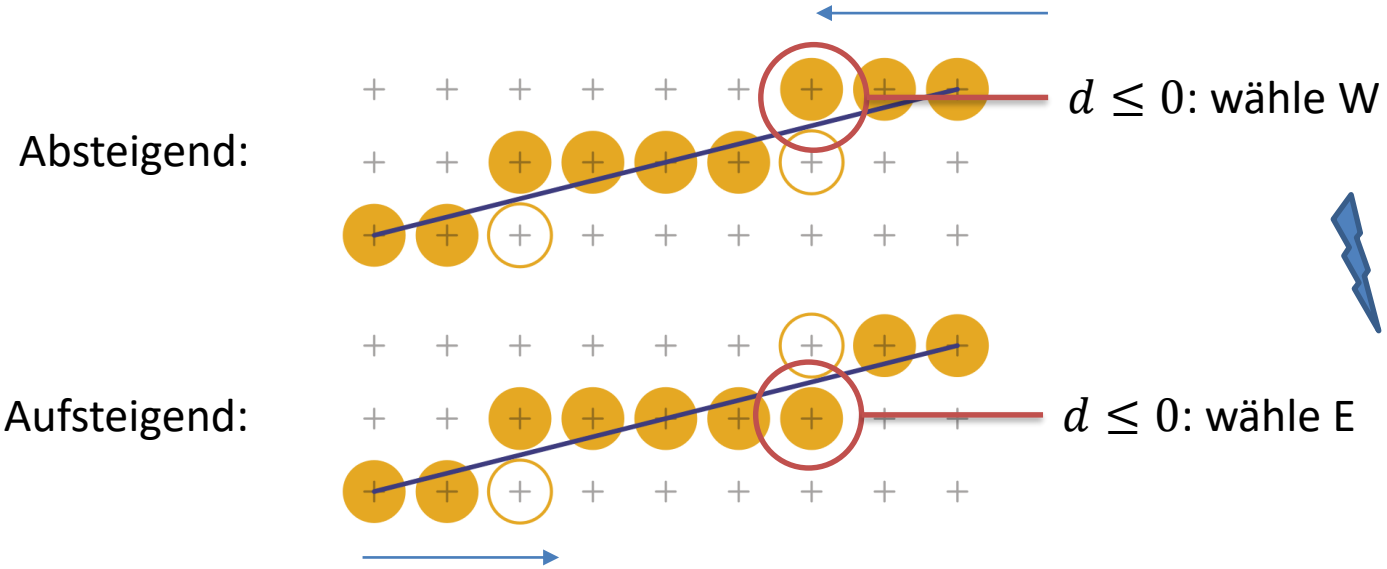
# Midpoint Line Algorithmus

- Unabhängigkeit der Laufrichtung: Was passiert wenn $d = 0$?

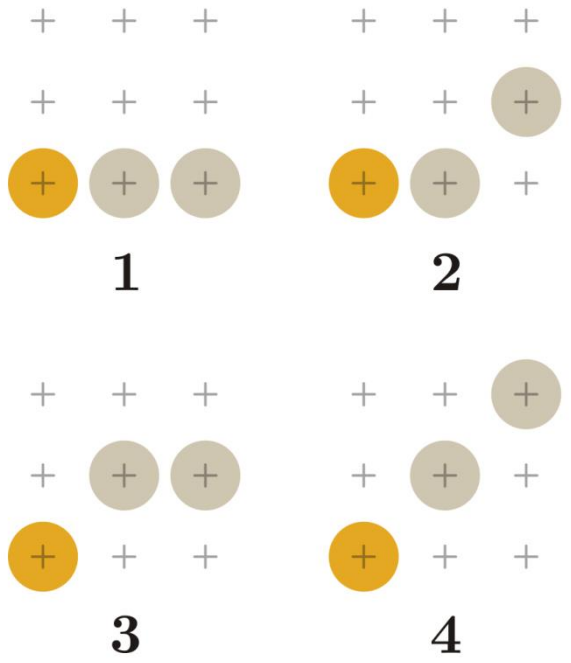



- Anwenden der Konvention für $d = 0$:
  - Wähle **E** wenn die Linie aufsteigend von links nach rechts gezeichnet wird
  - Wähle **SW** wenn die Linie absteigend von rechts nach links gezeichnet wird
- Zur Beschleunigung kann der Algorithmus auch von beiden Endpunkten gleichzeitig gestartet werden

https://de.wikipedia.org/wiki/Rasterung_von_Linien

# N-Schritt-Verfahren

- Grundidee: Schritte von mehreren Pixeln in x-Richtung vollziehen, alle dazwischen liegende Pixel werden auf einmal eingefärbt
- Wahl von verschiedenen Pixelmustern, z.B. **Doppelschrittverfahren**:
  - Wenn Steigung $m \leq ½$: Muster 4 kann nicht auftreten
  - Wenn Steigung $m \geq ½$: Muster 1 kann nicht auftreten




- Entscheidung für Pixelmuster:
  Betrachtung der letzten Pixelspalte
  - Pixel oben → (4)
  - Pixel unten → (1)
  - Pixel in der Mitte → weiterer Test
- Analoge Vorgehensweise wie bei Midpoint Line Algorithmus: Betrachtung der Midpoints $M$

https://de.wikipedia.org/wiki/Rasterung_von_Linien

# N-Schritt-Verfahren

- Pseudocode für den 1. Fall ($m < ½$) :

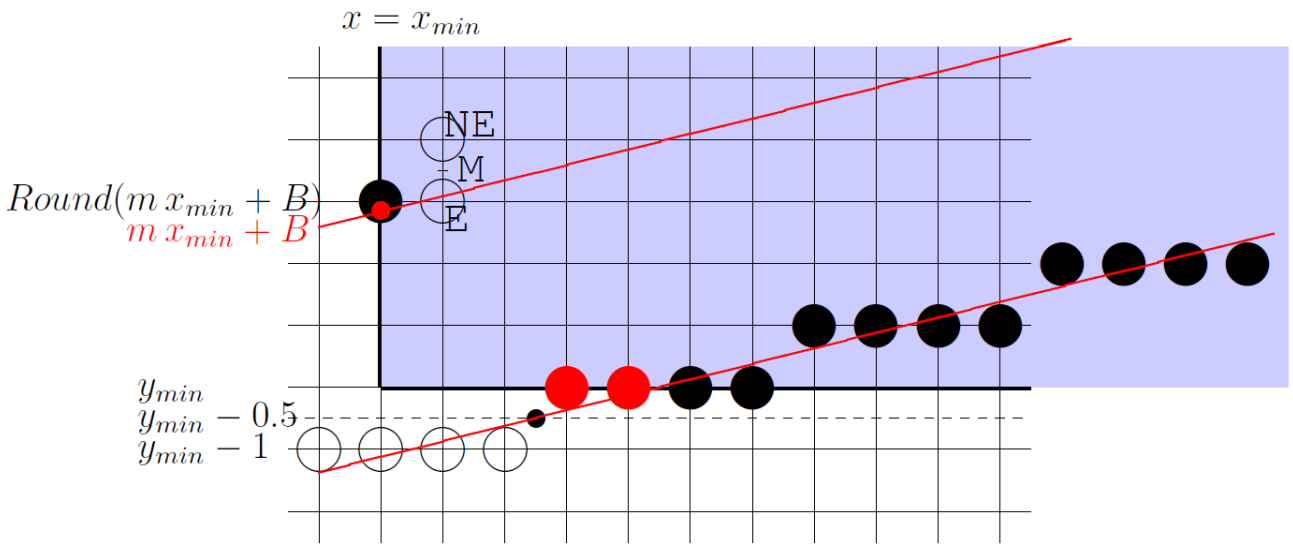
```
// init
dx = x1-x0;
dy = y1-y0;
incr1= 4*dy;
incr2= 4*dy-2*dx;
cond = 2*dy
d = 4*dy - dx;
x = x0;
y = y0;
```

```
while(x < x1) {
   if(d <= 0) {        // Pattern 1
      drawPixels(pattern1,x);
      d += incr1;
   } else {
      if(d < cond) {  // Pattern 2
         drawPixels(pattern2,x);
      } else          // Pattern 3
         drawPixels(pattern3,x);
      }
      d += incr2;
   }
   x += 2;
}
```

- Algorithmen für Dreifach- und Vierfachschritt nach gleichem Prinzip vorhanden
- Midpoint-Line-Algorithmus ist Spezialfall des N-Schritt-Verfahrens mit einer Schrittweite von 1 und zwei Mustern aus denen gewählt wird.

# Schnittpunkt mit Viewport

- Beim Erzeugen von Linien muss auf Schnittpunkte mit dem Viewport geachtet werden

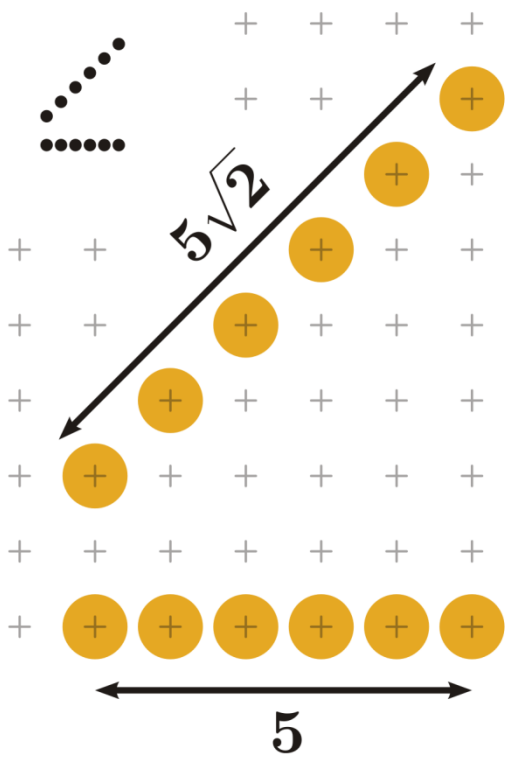



- Initialisierung kann angepasst werden (keine Rundung!):
- Schnitt mit unterer/oberer Kante: Schnitt mit $y_{min} - 0.5$ bzw. mit $y_{max} + 0.5$ verwenden.

S. Krömker: Computergrafik I, Universität Heidelberg

# Intensitätsschwankungen

- Konstante Anzahl von Pixeln für unterschiedlich lange Linien (bis Faktor $\sqrt{2}$).
- Intensität der Linie schwankt
- Bei Bilevel-Displays keine Abhilfe möglich
- Sonst: Variieren der Intensität einer Linie als Funktion der Steigung

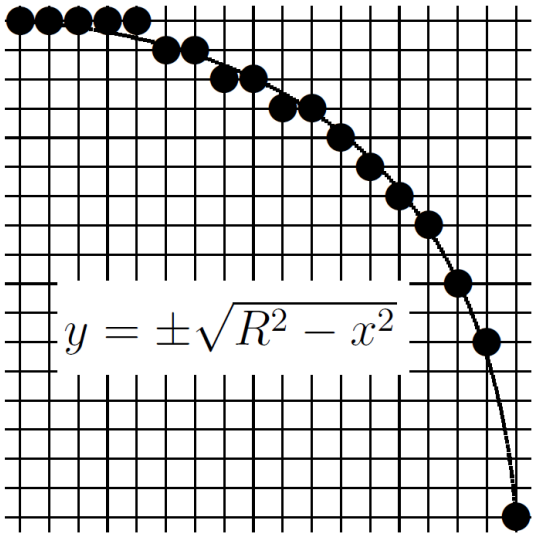


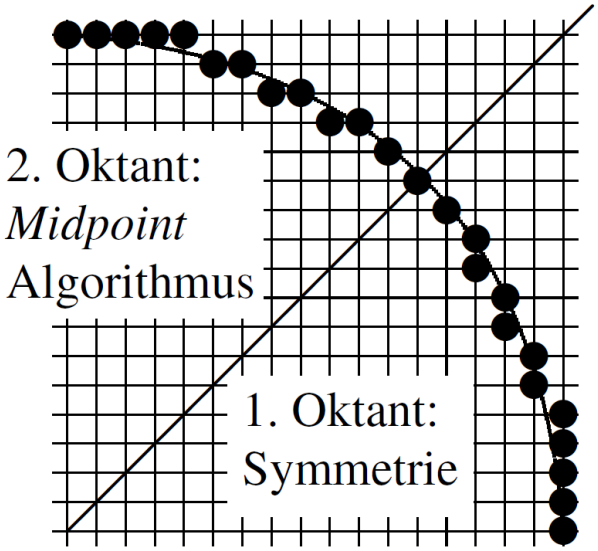

https://de.wikipedia.org/wiki/Rasterung_von_Linien

# 5.2. ZEICHNEN VON KREISEN

# Midpoint Circle Algorithmus

- Zeichnen von Kreisen analog zum Midpoint Line Algoritmus.
- Kreis wird im zweiten Oktanten gezeichnet und die Symmetrieeigenschaft für eine Spiegelung genutzt.

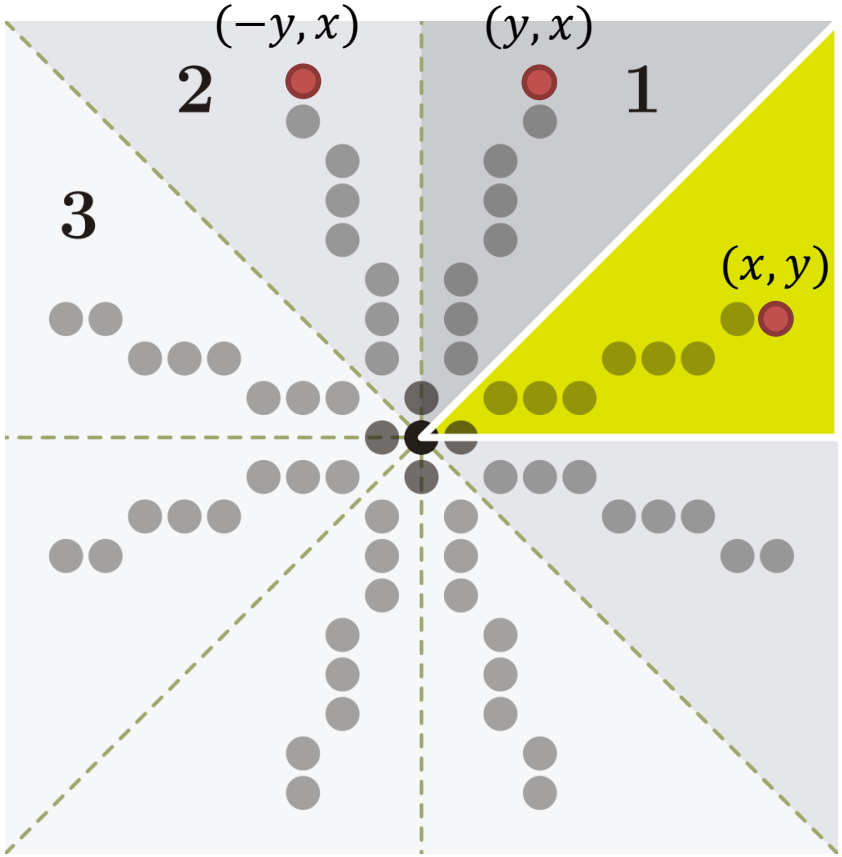



S. Krömker: Computergrafik I, Universität Heidelberg

# Midpoint Circle Algorithmus

- Erinnerung: Ausnutzung der Symmetrie




```
void circlePoint (int x, int y) {
   writePixel( x, y);
   writePixel( y, x);
   writePixel( y,-x);
   writePixel( x,-y);
   writePixel(-x,-y);
   writePixel(-y,-x);
   writePixel(-y, x);
   writePixel(-x, y);
}
```

https://de.wikipedia.org/wiki/Rasterung_von_Linien

# Midpoint Circle Algorithmus

- Gleichung eines Kreises: Implizite Formulierung

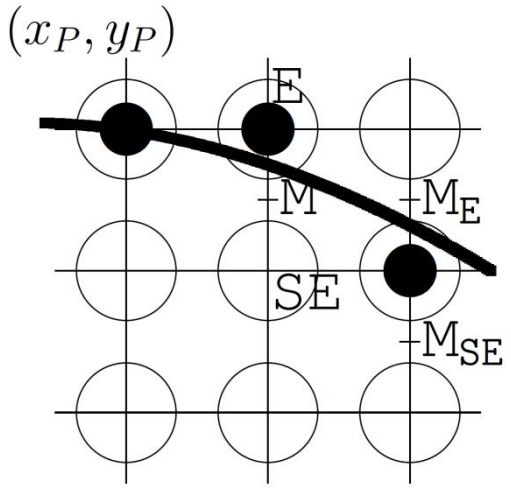
$$F(x, y) = x^2 + y^2 - R^2 = 0$$

- Konvention:
  - Start bei 12 Uhr
  - Zeichnen bis 1.30 Uhr
- Entsprechend wird die Entscheidungsvariable berechnet

$$F(M) = F\left(x_p + 1, y_p - \tfrac{1}{2}\right) = d$$

$$d = (x_p + 1)^2 + (y_p - \tfrac{1}{2})^2 - R^2$$

- Vorzeichen von $d$ entscheidet über nächsten Punkt auf der Linie:
  - $d \geq 0$: wähle SE
  - $d < 0$: wähle E




S. Krömker: Computergrafik I, Universität Heidelberg

# Midpoint Circle Algorithmus

- Die neue Entscheidungsvariable $d_{new}$ wird in Abhängigkeit von der Wahl SE oder E aus der alten Entscheidungsvariablen $d_{old}$ berechnet:
  - Wenn E gewählt wurde:

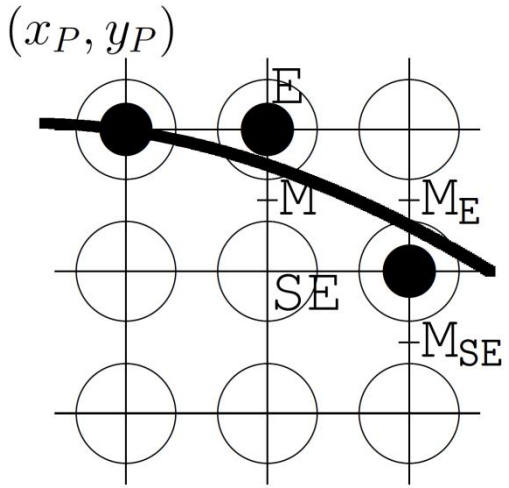
$$d_{new} = F\left(x_p + 2, y_p - \tfrac{1}{2}\right) = (x_p + 2)^2 + (y_p - \tfrac{1}{2})^2 - R^2$$

$$\Rightarrow\ d_{new} = d_{old} + 2x_p + 3$$

  - Wenn SE gewählt wurde:

$$d_{new} = F\left(x_p + 2, y_p - \tfrac{3}{2}\right) = (x_p + 2)^2 + (y_p - \tfrac{3}{2})^2 - R^2$$

$$\Rightarrow\ d_{new} = d_{old} + 2x_p - 2y_p + 5$$




- Initialisierung von *d*:

$$d = F\left(x_0 + 1, y_0 - \tfrac{1}{2}\right) = F\left(1, R - \tfrac{1}{2}\right) = 1 + \left(R^2 - R + \tfrac{1}{4}\right) - R^2$$
$$= \tfrac{5}{4} - R$$

S. Krömker: Computergrafik I, Universität Heidelberg

# Midpoint Circle Algorithmus

- Pseudocode:

```
// init
d = 5/4-radius;       // float!
x = 0;
y = radius;
circlePoint(x,y);     // first pixel

// iterate
while (y > x) {
   if(d < 0) {        // choose E
      d += x*2.0 + 3
      x++;
   } else {           // choose SE
      d += (x-y)*2.0 + 5;
      x++;
      y--;
   }
   circlePoint(x,y);
}
```

# Midpoint Circle Algorithmus

- Vergleich Midpoint Line / Midpoint Circle Algorithmus

| | Midpoint Line | Midpoint Circle |
|---|---|---|
| **Offset** | E: $\Delta y$<br>NE: $(\Delta y - \Delta x)$ | E: $2x_p + 3$<br>SE: $2x_p - 2y_p + 5$ |
| **Inkremente** | Konstant | Lineare Funktion |
| **Operationen** | Integer | Floating Point |

- Weitere Bresenham-Algorithmen:
  - Ellipsen
  - Bézierkurven
  - …

http://members.chello.at/~easyfilter/bresenham.html

# 5.3. ANTI-ALIASING BEI LINIEN

# Anti-Aliasing bei Linien

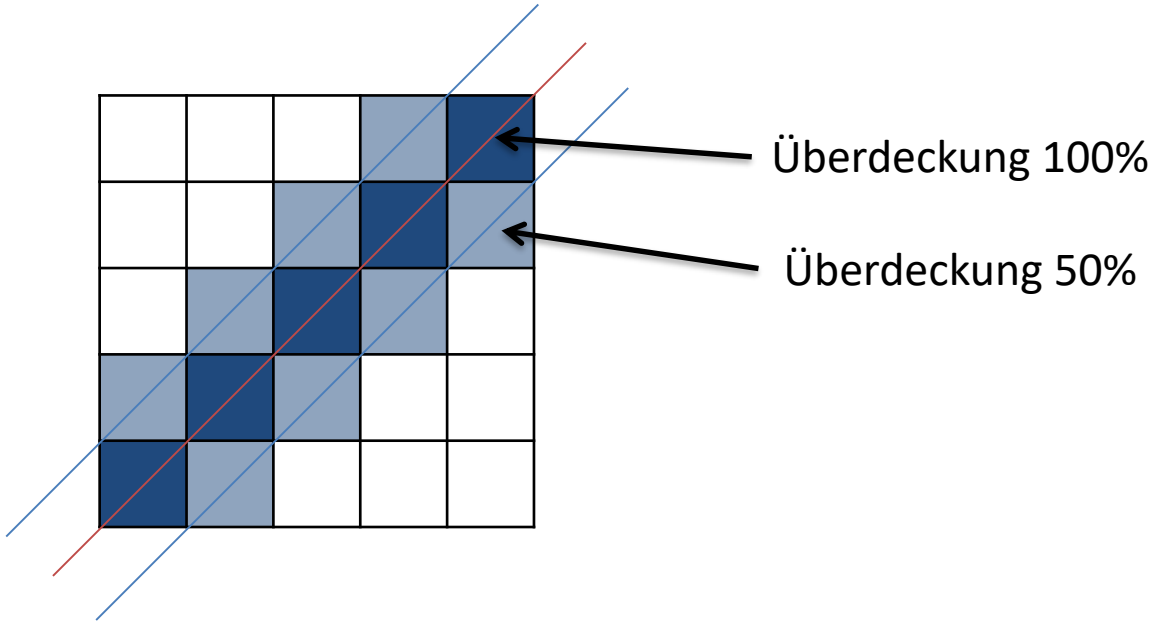
- Binäre Entscheidung führt zu Treppen-Effekten von Linien/Kreisen
  - Mögliche Abhilfe: Erhöhung der Bildschirmauflösung
- Anti-Aliasing: Variation der Intensität der Pixel entsprechend des Anteils am jeweiligen Objekt.
  - Meist beschränkt auf Linien/Kanten

- Alternative: Szenen-Antialiasing durch z.B. geschickte Nutzung des Accumulationbuffers *-- folgt im nächsten Kapitel*

https://commons.wikimedia.org/wiki/Category:Anti-aliasing

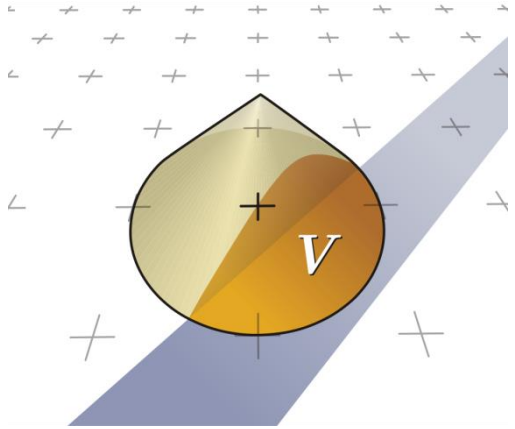
# Anti-Aliasing: ungewichtete Abtastung

- Ideale Linie hat keine Breite
- Beim Rastern geht man von einer Breite von mind. 1 Pixel aus
- Berechnung der Flächenüberdeckung für alle Pixel.
  - Intensität des Pixels = lineare Funktion der am Objekt beteiligten Fläche
  - Abstand der angeschnittenen Fläche zum Pixelmittelpunkt geht nicht mit ein.



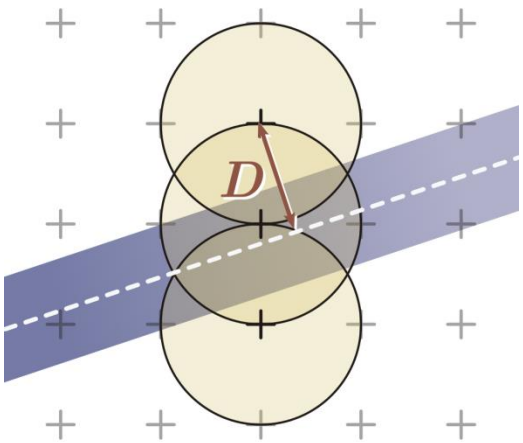
# Anti-Aliasing: Gupta-Sproull Methode

- 3D-Kegel als Glättungskern
- Intensität proportional zum überdeckten Volumen




- Volumen $V$ abhängig von Abstand des Pixelmittelpunktes zur Mittellinie ($D$)

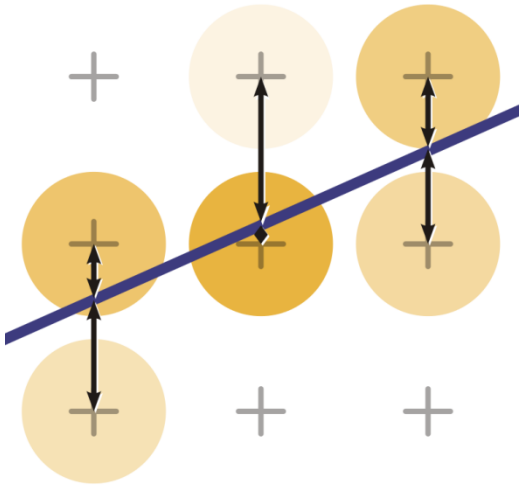
- Erweiterung des Bresenham-Algorithmus
  - Berechnet zusätzlich Distanz $D$, für jedes Pixel, dessen Glättungskern mit der Linie überlappt.
  - Effiziente Implementierung: Reduktion auf 24 mögliche Werte für $D$. Intensität wird aus Lookuptabelle entnommen.




https://de.wikipedia.org/wiki/Rasterung_von_Linien

# Anti-Aliasing: Wu-Methode

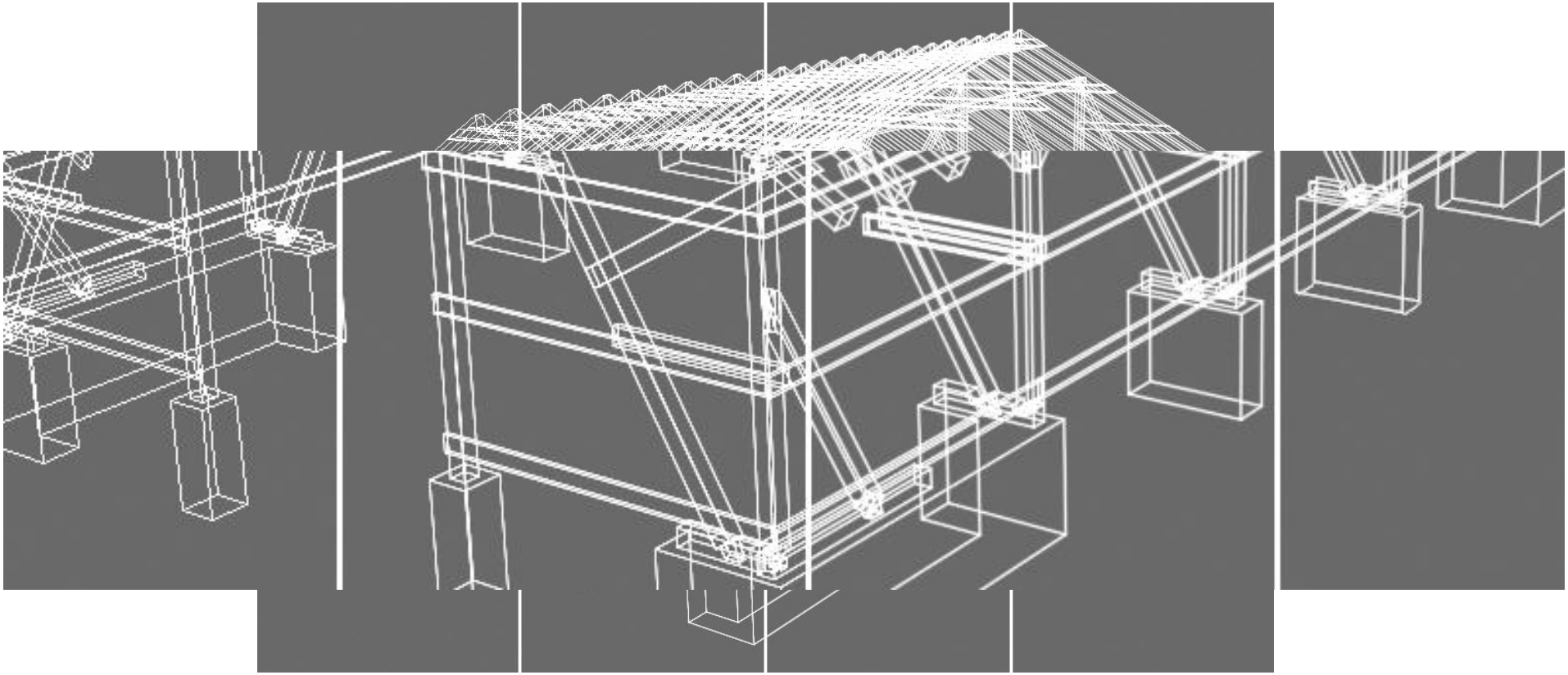
- Basiert auf Fehlermaß des Bresenham-Algorithmus
  - Statt binärer Linie werden beliebige Farbwerte zugelassen
  - Pixel erhalten Farbwerte proportional zur vertikalen Distanz zur idealen Linie



- Sehr performante Implementierung als Erweiterung des Bresenham-Algorithmus mit Kontrollvariable

# Anti-Aliasing bei Linien

- Vergleich der Methoden



Bresenham | Gupta-Sproull | ungewichtet | Wu

https://de.wikipedia.org/wiki/Rasterung_von_Linien

# Anti-Aliasing bei Linien

- Punkt/Linien/Polygon-Antialiasing in OpenGL:

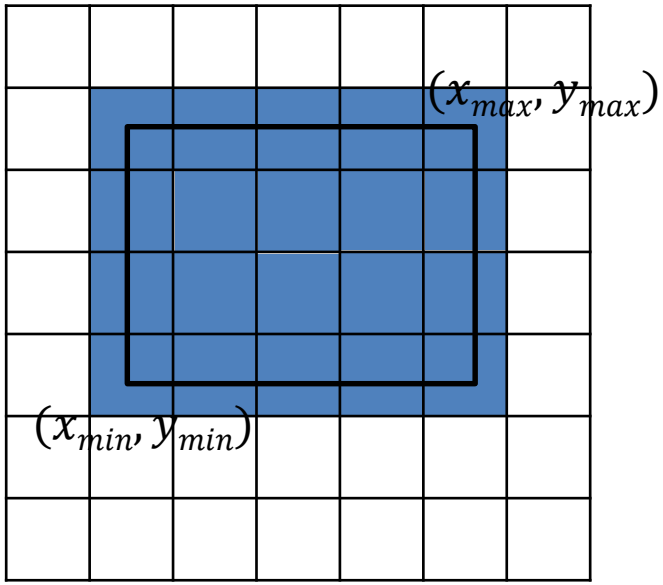
```
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glHint (GL_LINE_SMOOTH_HINT, GL_NICEST);
glHint (GL_POINT_SMOOTH_HINT, GL_NICEST);
glHint (GL_POLYGON_SMOOTH_HINT, GL_NICEST);


glEnable (GL_BLEND);
glEnable (GL_POINT_SMOOTH);
glEnable (GL_LINE_SMOOTH);
glEnable (GL_POLYGON_SMOOTH);
```

- Nutzt Alpha Blending – *sehen wir später in der Vorlesung*
- *Hints*: Hinweise an den Treiber schnellere Berechnung oder qualitativ hochwertigere Berechnung durchzuführen (Umsetzung obliegt dem Treiber)

# 5.4. FÜLLEN VON POLYGONEN

# Füllen von Rechtecken

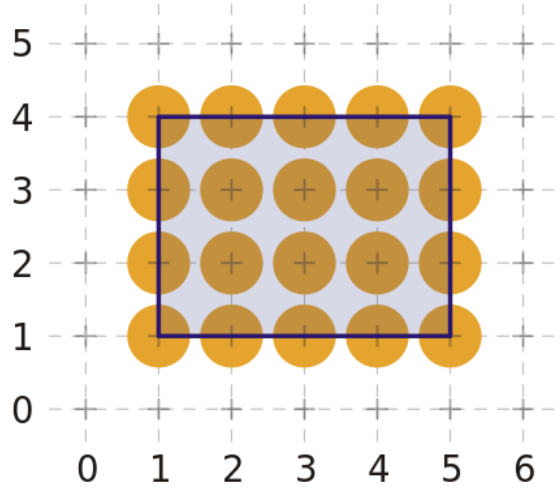
- Füllen eines achsenparallelen Rechtecks trivial:
  - Pixel zwischen $(x_{min}, y_{min})$ und $(x_{max}, y_{max})$ setzen.
  - i.d.R durch Scanline



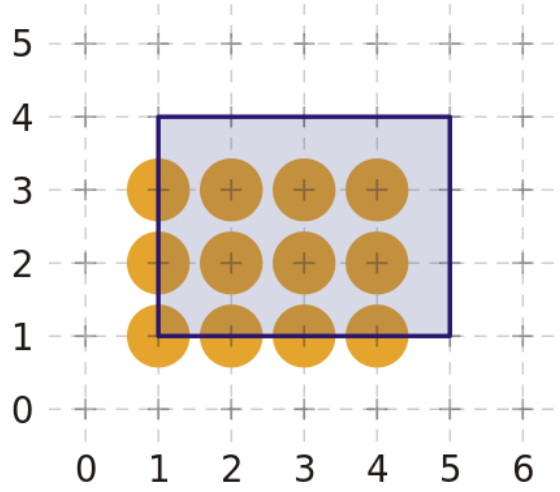

```
int x, y;
int color;

for (y=ymin; y <= ymax; y++) {
   for (x=xmin; x <= xmax; x++) {
      setPixel(x,y,color);
   }
}
```

# Füllen von Rechtecken

- Interpretation der Koordinaten:




*Rechteck größer als definiert*




- Abhilfe:
  - Rasterung um einen halben Pixelabstand nach links und nach unten verschieben.
  - Rastern von $x \geq x_{min} - 0{,}5$ bis $x \leq x_{max} - 0{,}5$ bzw. $y \geq y_{min} - 0{,}5$ bis $y \leq y_{max} - 0{,}5$
  - Rechteck rechts, mit den Koordinaten $(0{,}5, 0{,}5)$ und $(4{,}5, 3{,}5)$, wird korrekt gezeichnet.

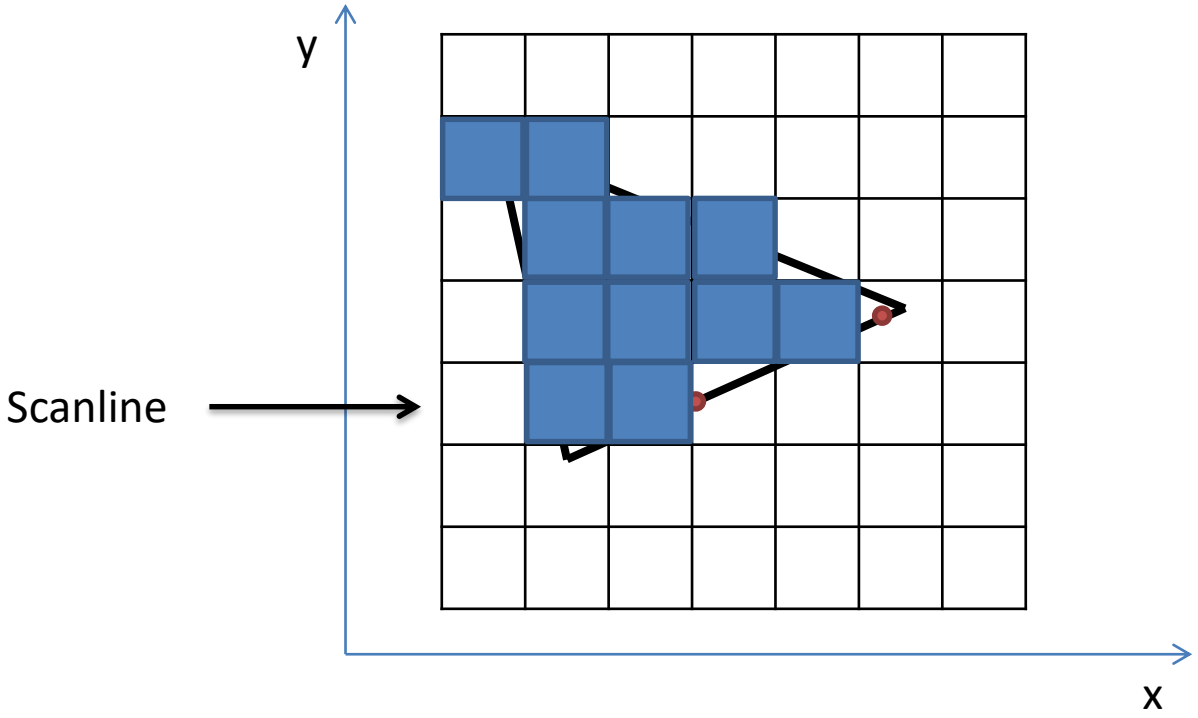
https://de.wikipedia.org/wiki/Rasterung_von_Polygonen

# Füllen von (beliebigen) Polygonen

- Für jede Kante des Polygons Bestimmung des Schnittpunktes mit der Bildzeile (Scanline)
- Horizontale Kanten werden ignoriert
- Sortierung der Schnittpunkte:
  - Nach *y*-Koordinaten sortiert (Scanline)
  - Bei gleichen *y*-Koordinaten wird aufsteigend nach *x*-Koordinaten sortiert
- Liste der Schnittpunkte enthält immer eine gerade Anzahl von Werten mit gleicher *y*-Koordinate
- Zeichnen:
  - Punktpaare aus der Liste von der Form $(x_1;\ y)\ (x_2;\ y)$
  - Zeichnen aller Pixel der entsprechenden Bildzeile deren *x*-Koordinate sich im Intervall $[x_1 - 0{,}5\ x_2 - 0{,}5]$ befindet.
- Hoher Sortieraufwand!

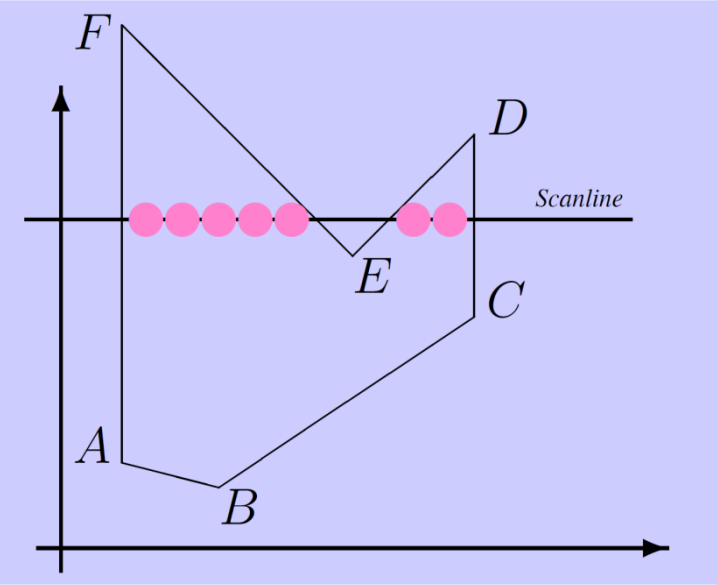
# Füllen von Polygonen

- Füllen des Innern des Polygons per Scanline anhand sortierter Schnittpunkte



# Füllen von Polygonen

- Modifikation:
  1. Für jeden Schnittpunkt einer Polygonkante mit einer Bildzeile: Einfärben des ersten Pixels mit $x > s_x + 0{,}5$.
  2. Füllen des Polygons durch Negation von innerhalb/außerhalb



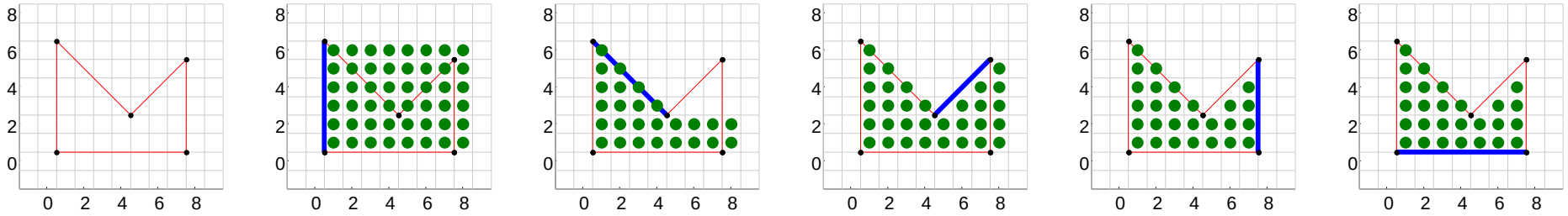

Bei bereits gezeichnetem Umriss:

```
Für jede Bildzeile y, die das Polygon schneidet
  Innerhalb = Falsch
  Für jedes x von links bis rechts
    Wenn Pixel (x, y) eingefärbt ist   // Umriss
      Innerhalb negieren
    Wenn Innerhalb
      Pixel (x, y) einfärben
    ansonsten
      Pixel (x, y) auf Hintergrundfarbe zurücksetzen
```

S. Krömker: Computergrafik I, Universität Heidelberg

# Füllen von Polygonen

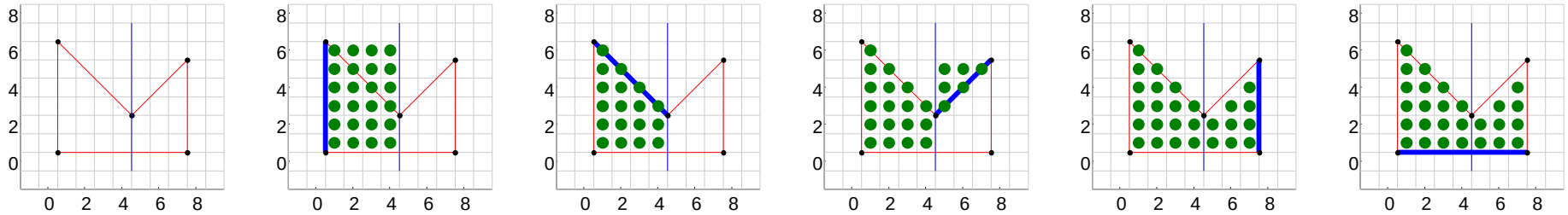
**Edge-Fill-Algorithmus**




- Verschiebung der Vertices um $\Delta x = -0{,}5$ und $\Delta y = -0{,}5$
- Für jede Bildzeile $y$, die bei Schnittpunkt $s_x$ eine Polygonkante schneidet: alle Pixel mit $x > s_x$ invertieren (binäre Wertzuordnung)
- Reihenfolge der Polygonkanten-Abarbeitung beliebig.
- Nachteil: viele Pixel müssen mehrmals geändert werden.

# Füllen von Polygonen

## Fence-Fill-Algorithmus



- Erweiterung des Edge-Fill-Algorithmus:
  - Verschiebung der Vertices um $\Delta x = -0{,}5$ und $\Delta y = -0{,}5$
  - Vertikale Gerade durch Schnittpunkt zweier Polygonkanten $(= fence)$, Beispiel hier: bei $4{,}5$ (nach Verschiebung der Vertices)
  - Für alle Schnittpunkte auf der linken Seite des Zauns werden alle Pixel $x > s_x$ bis $x < fence$ invertiert.
  - Für alle Schnittpunkte auf der rechten Seite des Zauns werden alle Pixel $x > fence$ bis $x < s_x$ invertiert.
- Reduziert Invertierungsvorgänge

# ZUSAMMENFASSUNG

# Zusammenfassung

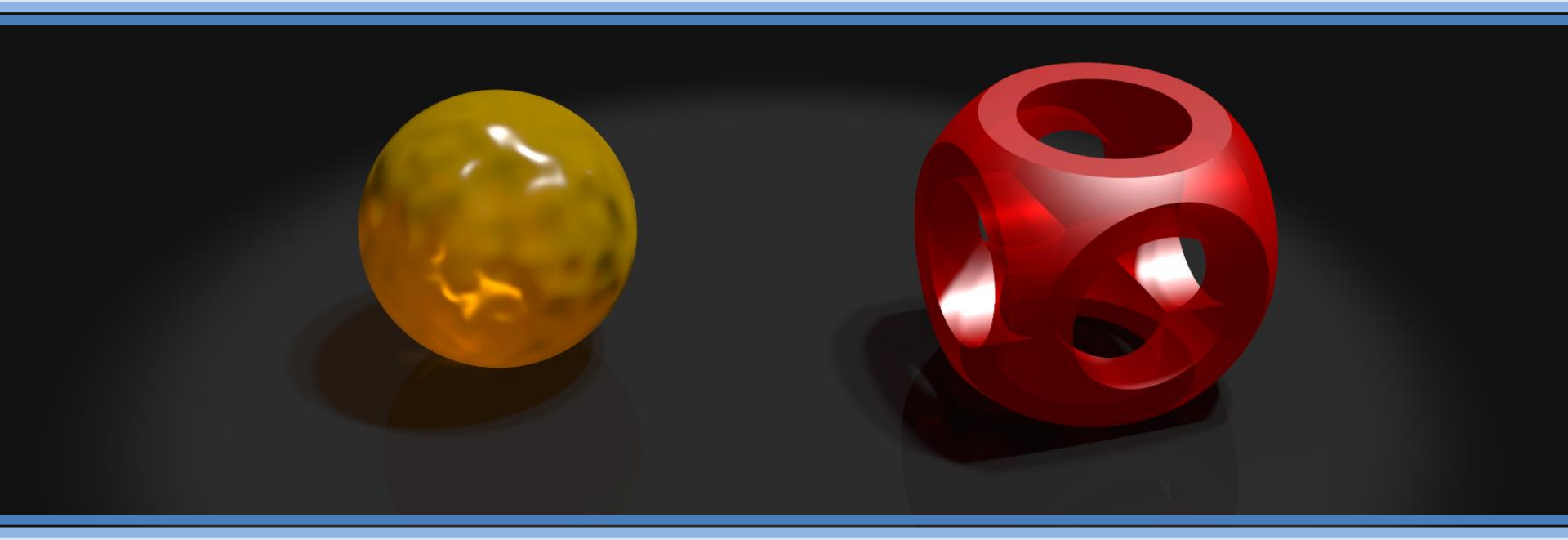
- Rastern: Umsetzung der kontinuierlichen Projektion der Szene in diskrete Pixel auf dem Bildschirm
- Clipping = Zuschneiden von Objekten an einem vorgegebenen Bereich.
- Zeichnen von Linien:
  - Midpoint Line Algorithmus, N-Schritt-Verfahren.
- Zeichnen von Kreisen:
  - Midpoint Circle Algorithmus
- Anti-Aliasing verringert optisch Treppeneffekte beim Zeichnen
  - Intensität von geschnittenen Pixel anhand Flächenüberdeckung
  - Ungewichtete Abtastung, Gupta-Sproull Methode, Wu-Methode
- Füllen von Rechtecken und Polygonen
  - Schnitt von Kanten mit Scanline
  - Negierung entlang der Scanline
  - Edge-Fill-Algorithmus, Fence-Fill-Algorithmus

# Übungsfragen Kapitel 5

- Was ist Clipping?
- Warum kommt bei Linien unterschiedlicher Steigung zu Intensitätsschwankungen? Wie kann man Abhilfe schaffen?
- Beschreiben Sie ein Verfahren zum Antialiasing von Linien

# Computergrafik

T. Hopp



# Themenübersicht

1. Einführung
2. Programmierbibliotheken / OpenGL
3. Geometrische Repräsentation von Objekten
4. Koordinatensysteme und Transformationen
5. Zeichenalgorithmen
6. **Buffer-Konzepte**
7. Farbe, Beleuchtung und Schattierung
8. Texturen
9. Animationen
10. Raytracing
11. Volumenvisualisierung

# Buffer in der Computergrafik

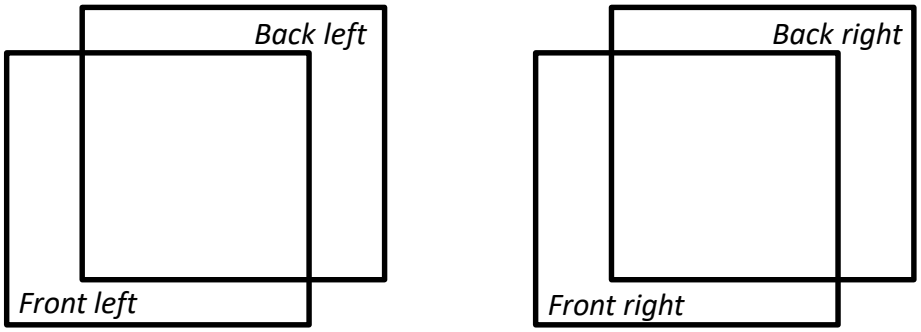
- Gleichmäßig für alle Pixel gespeicherte Daten nennt man Buffer (Pufferspeicher)
- Der Frame-Buffer (auch Color Buffer) umfasst alle anderen Buffer
  - = Bildspeicher
  - Entspricht einer digitalen Kopie des Monitorbildes
  - Größe abhängig von verwendeter Auflösung und Farbtiefe

- Wir betrachten einige Buffer-Konzepte, die zusammen den Frame-Buffer füllen:
  - Double und Stereo Buffer
  - Depth Buffer
  - Stencil Buffer
  - Accumulation Buffer

# 6.1. DOUBLE U. STEREO BUFFERING

# Double Buffering

- Single Buffering: Löschen & Bildaufbau dauert oft zu lange um den Neuaufbau des Bildes für das Auge unsichtbar zu machen
  - Typisches „Flackern“ / Verwischen: vor allem bei Animationen/Bewegung störend
- Double Buffering schafft Abhilfe:
  - Unterteilung des Frame-Buffers in Front-Buffer und Back-Buffer
  - Front-Buffer wird ausgelesen und auf dem Bildschirm dargestellt
  - Back-Buffer wird zeitgleich gelöscht und mit neu berechnetem Bild gefüllt
  - Anschließend Austausch von Front- und Back-Buffer (*Swap*)
- Bei stereofähigem Grafiksystem: zwei zusätzliche Buffer für links und rechts:



# Double Buffering

- In OpenGL:
  - Setzen des Buffermodus beim Initialisieren eines Fensters:

    **`glutInitDisplayMode(…|GLUT_SINGLE|…)`**

  - GLUT_SINGLE: Single Buffering
  - GLUT_DOUBLE: Double Buffering
  - GLUT_STEREO: Stereo Buffering

- Unterstützte Modi des Systems können mit **`glGetBooleanv(GL_DOUBLEBUFFER)`** bzw. **`glGetBooleanv(GL_STEREO)`** abgefragt werden
- Jedes System hat mindestens einen Colorbuffer (Frame Buffer): *Front left*
- Der Tausch zwischen Front- und Back-Buffer erfolgt am Ende der Zeichenroutine durch den Aufruf **`glutSwapBuffers();`**
- Neuzeichnen der Szene beginnt mit Löschen des jeweiligen Buffers

# Tausch-Operation

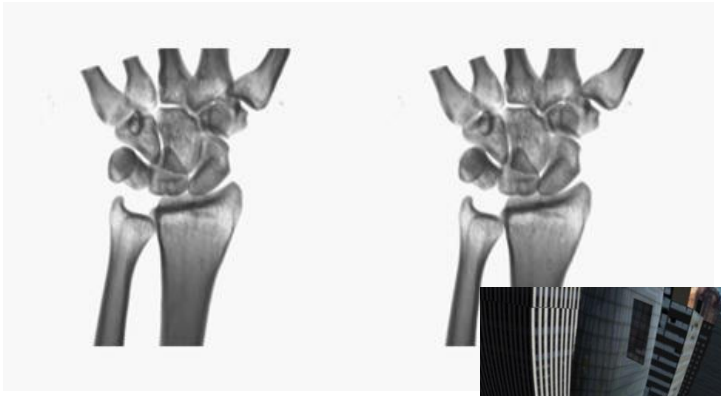
- Wichtig: Synchronisation zwischen Anzeige-Framerate und Generierungs-Framerate
- Ohne gesonderte Behandlung (bei deaktiviertem VSYNC):
  - Sofortiger Tausch zwischen Front- und Back-Buffer sobald der Back-Buffer gefüllt ist
  - Ist das Bild auf dem Bildschirm noch nicht vollständig aufgebaut, wird ab der aktuellen Stelle auf dem Bildschirm bereits das neue Bild dargestellt: Tearing-Artefakte
- Vertikale Synchronisation (VSYNC): verhindert Aktualisierung der Bilddaten, während der Bildschirm das Bild aufbaut.
- Bei aktiviertem VSYNC:
  - Tausch findet erst statt wenn der Bildschirm das Bild vollständig aufgebaut hat
  - Künstliche Reduktion der Leistung der GPU
  - Kompensation durch Dreifach-Pufferung
- Typische Bildwiederholrate:
  - 60Hz , 75 Hz bei LCD Bildschirmen

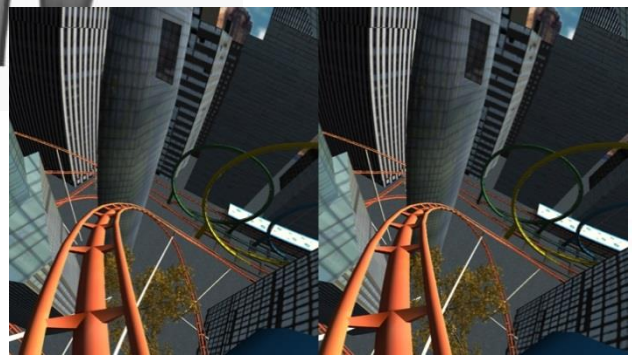
http://www.geforce.com/hardware/technology/adaptive-vsync/technology

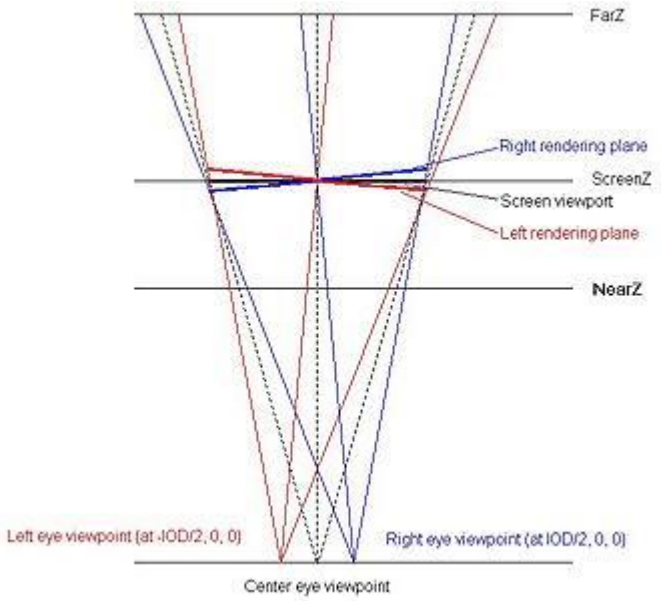
# Stereo Buffering

- Bei Stereo-Darstellungen werden Bilder für das linke und rechte Auge aus unterschiedlicher Kameraposition erzeugt. → Stereosehen
  - Vergl. **`gluLookAt`**-Befehl
- Der Buffer in den gezeichnet werden soll wird ausgewählt durch:

```
glDrawBuffer(GL_BACK_RIGHT);
```








http://www.orthostereo.com/geometryopengl.html
http://www.pcadvisor.co.uk/test-centre/gadget/12-best-google-cardboard-apps-2016-uk-3585299/

# Umsetzung von Stereo-Darstellungen

**Passiv-Stereo:** Trennung der Information für linkes und rechtes Auge durch Filter

- Farb-Filter: meist rot-grün Filter
  - Farbinformation geht u.U. verloren

- Polfilter: Vertikal- und horizontal polarisiertes Licht
  - Farbinformation bleibt erhalten
  - Betrachter darf Kopf nicht neigen. Abhilfe: zirkular polarisiertes Licht

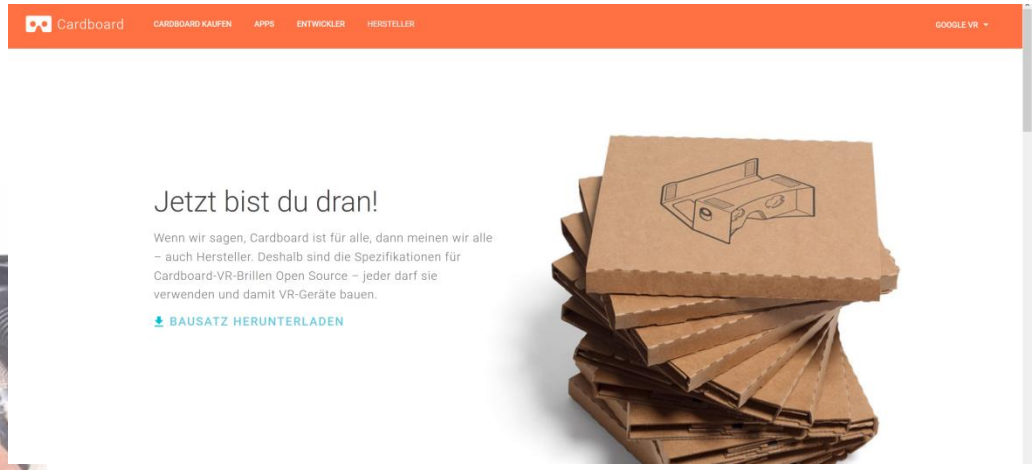
**Aktiv-Stereo:** Abwechselnde Darstellung von Vollbildern für das linke und rechte Auge

- Trennung der Bilder für das Auge durch synchronisierte Shutterbrille

https://www.3d-brillen.de/3d-brillen/3d_brille_anaglyph_rot_gruen.html
https://www.conrad.de/de/passive-3d-brille-hama-3d-polfilterbrille-943701.html
http://www.hifi-forum.de

# Umsetzung von Stereo-Darstellungen

- … oder ganz einfach durch zwei getrennte Displays für die Augen



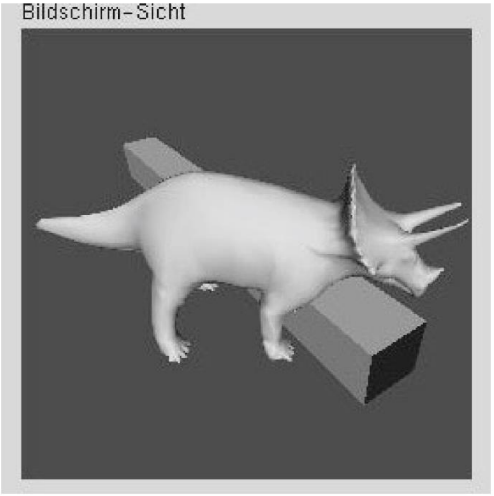

*https://vr.google.com/intl/de_de/cardboard/manufacturers/*

http://www.pcwelt.de/ratgeber/Die-besten-VR-Apps-fuer-Handy-und-Cardboard-9976062.html
https://de.wikipedia.org/wiki/Oculus_Rift

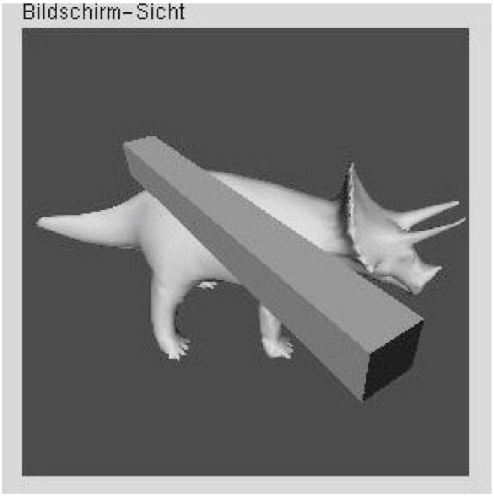
# 6.2. DEPTH- BZW. Z-BUFFER

# Verdeckung

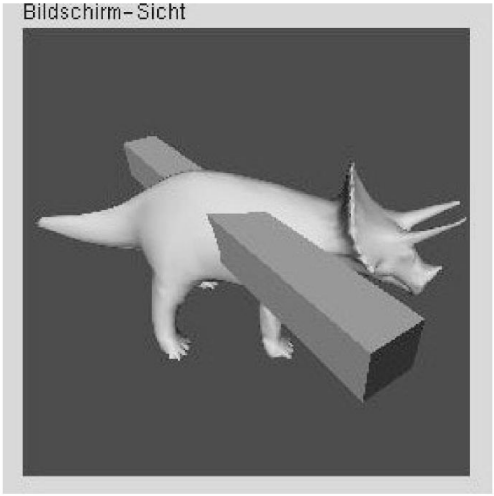
- Gegenseitige Verdeckung von Objekten gibt räumlichen Eindruck und Hinweis auf Entfernung der Objekte vom Augpunkt.
- I.a. werden Objekte in der Reihenfolge ihrer Definition gezeichnet
  - D.h. Pixel erhält ohne weitere Behandlung die Farbe des zuletzt gezeichneten Objektes




*Dinosaurier zuletzt gezeichnet*




*Quader zuletzt gezeichnet*




*Korrekte Darstellung (z-Buffer)*

A. Nischwitz et al. „Computergrafik und Bildverarbeitung“, Band 1: Computergrafik“

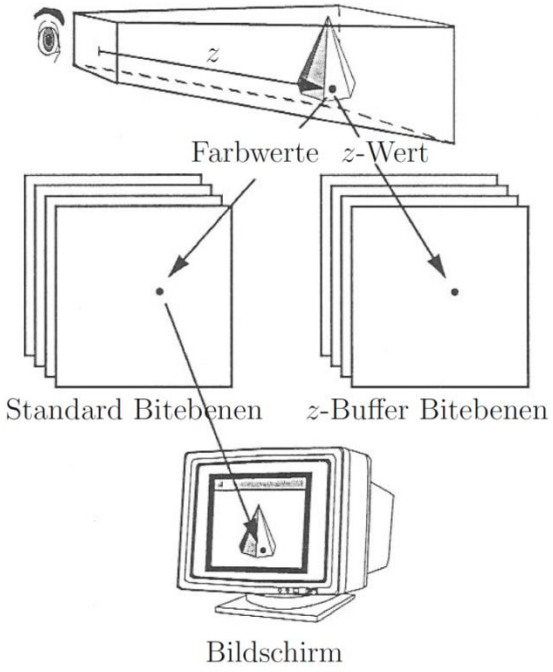
# Maler-Algorithmus

- Ermöglicht Unabhängigkeit von Reihenfolge der Definition
- Einfacher zweistufiger Ablauf:
  1. Sortieren aller Objekte/Polygone in Bezug auf ihren Abstand zum Augpunkt
  2. Zeichnen aller Objekte/Polygone in der neuen Reihenfolge, beginnend mit entferntestem

- Nachteile:
  - Hoher Sortieraufwand, speziell bei großer Objekt-/Polygonanzahl
  - Bei Objekt- oder Augpunktbewegung Neu-Sortierung erforderlich
  - Gegenseitige Durchdringung von Objekten/Polygonen kann nicht abgebildet werden

# Z-Buffer

- Grundidee: nutzen zusätzlichen Speichers für Tiefeninformation (z-Wert) für jedes Pixel
  - Für jedes Pixel eines Objektes: Test ob es näher am Augpunkt liegt als das vorher gezeichnete. → Wenn ja: in den Color-Buffer schreiben.



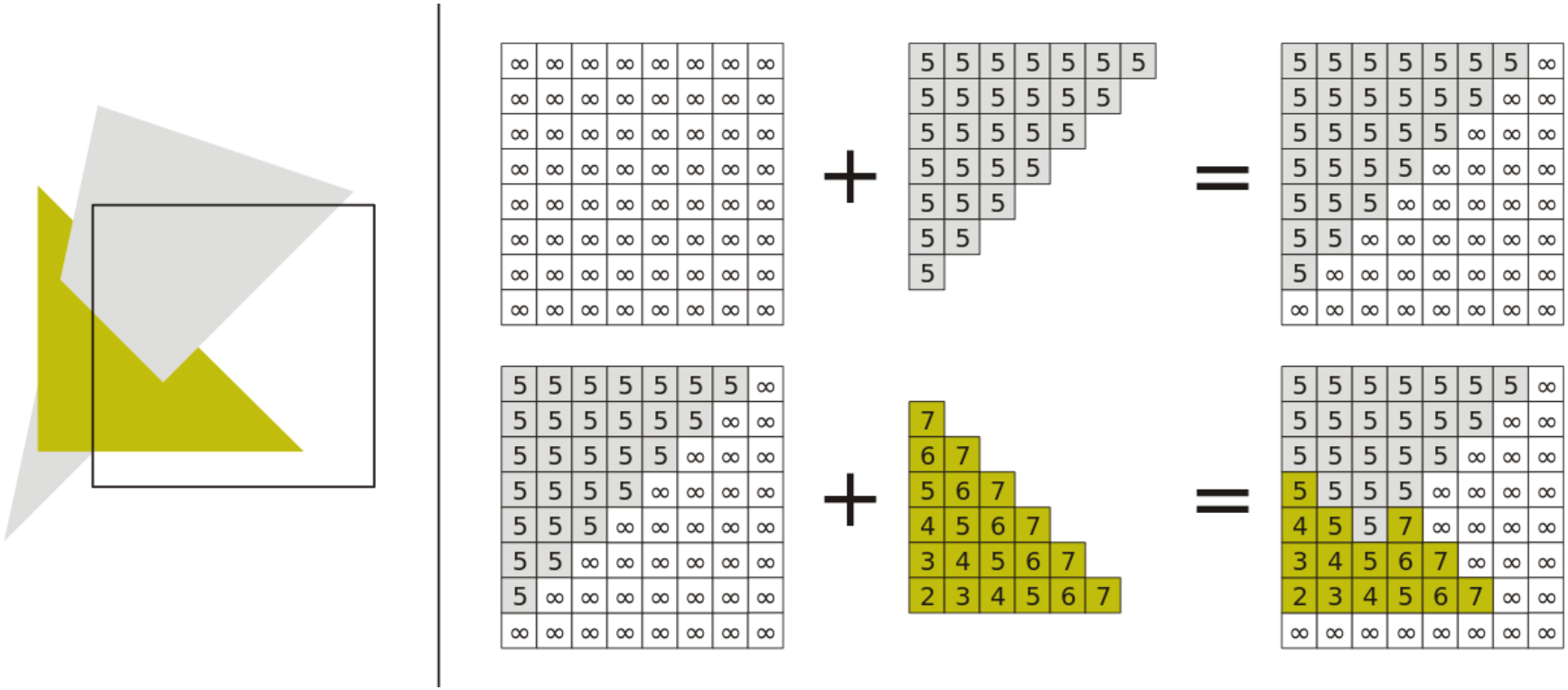

A. Nischwitz et al. „Computergrafik und Bildverarbeitung“, Band 1: Computergrafik“

# Z-Buffer-Algorithmus

- Pseudocode:

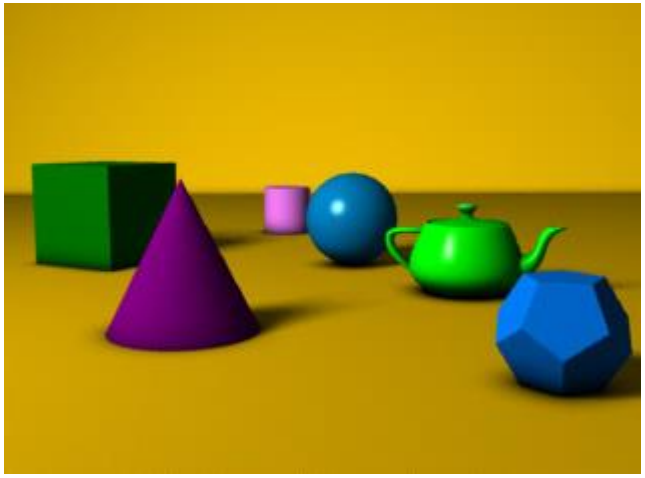
```
void zBuffer() {
   int x, y;
   for (y = 0; y < YMAX; y++){
      for (x = 0; x < XMAX; x++){
         writePixel(x,y, BACKGROUND_COLOR);  /* Clear color */
         writeZ(x,y, DEPTH);                 /* Clear depth */
      }
   }

   for(eachPolygon){
      for(eachPixel in polygon's projection){
         pz = polygon's z-value at pixel's coordinates(x,y);
         if (pz <= readZ(x,y)){              /* New point is nearer */
            writeZ(x,y,pz);
            writePixel(x,y, polygon's color at (x,y));
         }
      }
   }
}
```

# Z-Buffer-Algorithmus




https://de.wikipedia.org/wiki/Z-Buffer

# Z-Buffer-Algorithmus



*Generiertes Bild*

*Zugehöriger z-Buffer*

https://de.wikipedia.org/wiki/Z-Buffer

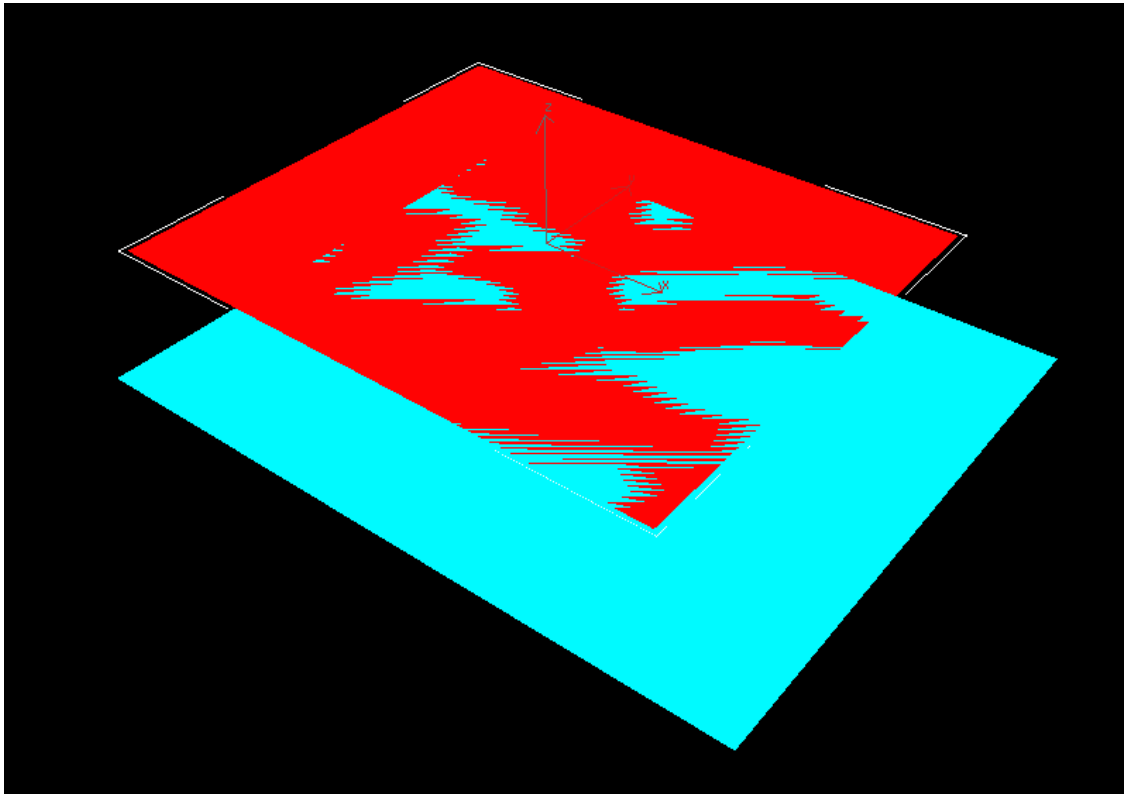
# Z-Buffer-Algorithmus

- Vorteile:
  - Kein aufwändiges Sortieren von Objekten nötig
  - Pixelgenaue Verdeckung von sich durchdringenden Objekten
  - Z-Wert-Berechnung einfach und schnell
    - Eckpunkte von Polygonen schon vorhanden durch Transformationskette
    - Lineare Interpolation für jedes Pixel eines Polygons
  - einfache Parallelisierung möglich

- Probleme:
  - Begrenzte Genauigkeit des z-Wertes: Z-Buffer-Flimmern („Z-Fighting“)
  - Transparente Oberflächen werden nicht korrekt berücksichtigt

# Z-Buffer-Algorithmus

- Z-Fighting



https://de.wikipedia.org/wiki/Z-Buffer

# Z-Buffer-Algorithmus

- Anwendung in OpenGL:
  - Aktivierung des z-Buffers: **`glEnable(GL_DEPTH_TEST);`**
  - Anfordern des zusätzlichen Bildspeichers: **`glutInitDisplayMode(…|GLUT_DEPTH|…);`**

- Initialisierung des z-Buffers:
  - Default: z-Werte zwischen 0.0 (*near clipping plane*) und 1.0 (*far clipping plane*).
  - Zuweisen eines Initialisierungswertes: **`glClearDepth(Gldouble depth);`**
  - In der Regel der Maximalwert
  - Initialisierungswert der Hintergrundfarbe:

    **`glClearColor(Gldouble r, Gldouble g, Gldouble b);`**
  - Löschen ist eine der teuersten Aktionen im Darstellungsprozess: Jedes Pixel muss angesteuert und zurückgesetzt werden!
  - Gleichzeitiges Zurücksetzen von Farbe und z-Buffer:

    **`glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`**
  - Einschränkung des Wertebereichs, falls nötig:

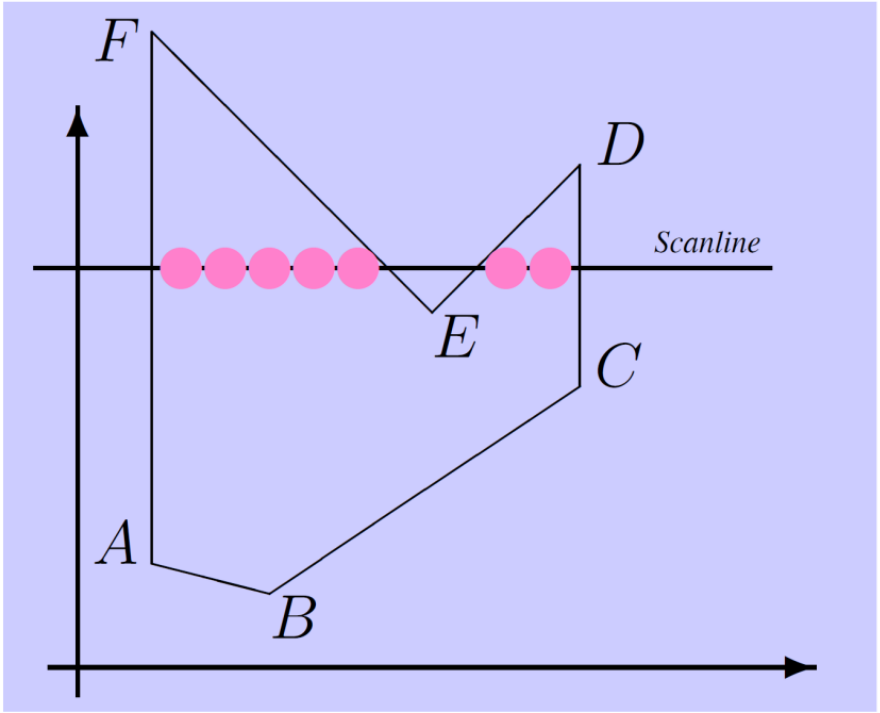
    **`glDepthRange(Gldouble near, Gldouble far);`**

# Z-Buffer-Algorithmus

- Auswahl der Vergleichsoperation: **`glDepthFunc(Glenum operator)`**
- Folgende Vergleichsoperationen stehen zur Verfügung:
  - **`GL_LESS:`** <, kleiner (Default Operation)
  - **`GL_NEVER:`** 0, liefert immer den Wahrheitswert **`false.`**
  - **`GL_EQUAL:`** =, gleich
  - **`GL_LEQUAL:`** <=, kleiner oder gleich
  - **`GL_GREATER:`** >, größer
  - **`GL_GEQUAL:`** >=, größer oder gleich
  - **`GL_LESS:`** <, kleiner
  - **`GL_NOTEQUAL:`** ≠, ungleich
  - **`GL_ALWAYS:`** 1, liefert immer den Wahrheitswert **`true`**

# Z-Wert-Bestimmung

- Zur Erinnerung: Zeichnen per Scanline:




- z-Werte nach Anwendung der Transformationskette bekannt für jeden Vertex

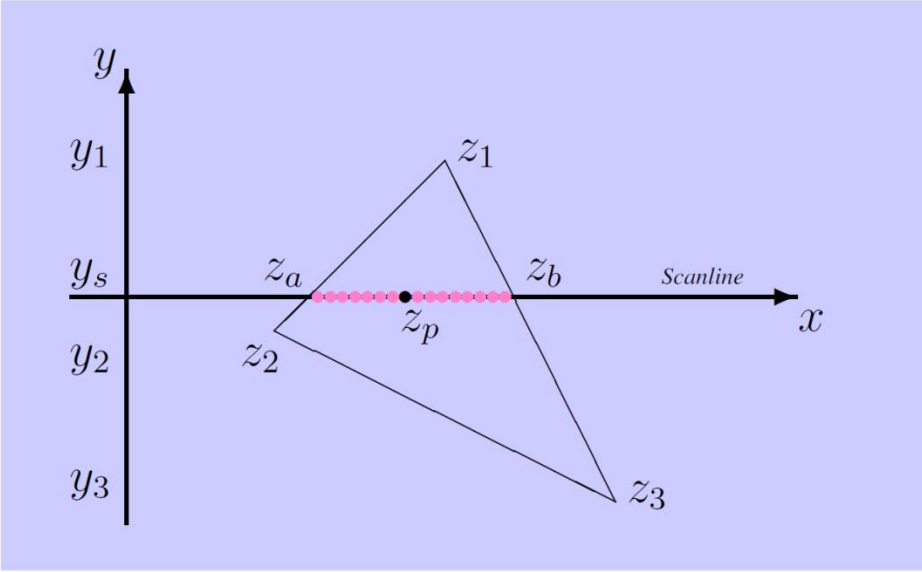
S. Krömker: Computergrafik I, Universität Heidelberg

# Z-Wert-Bestimmung

- Lineare Interpolation der z-Werte entlang der Scanline für alle Pixel:

$$z_a = z_1 - (z_1 - z_2)\frac{y_1 - y_s}{y_1 - y_2}$$
$$z_b = z_1 - (z_1 - z_3)\frac{y_1 - y_s}{y_1 - y_3}$$

$$z_p = z_b - (z_b - z_a)\frac{x_b - x_p}{x_b - x_a}$$




S. Krömker: Computergrafik I, Universität Heidelberg

# Z-Wert-Bestimmung: Beschleunigung

- Ebenengleichung zur Darstellung des Polygons:

$$Ax + By + Cz + D = 0 \quad \Rightarrow \quad z(x, y) = \frac{-Ax - By - D}{C}$$

  - $(A, B, C)$ stellt den Normalenvektor der Ebene dar, $D$ ist Abstand vom Ursprung
- Beschleunigung durch Berechnung aus dem vorherigen Wert:

$$z(x + \Delta x, y) = z(x, y) - \frac{A}{C}(\Delta x)$$

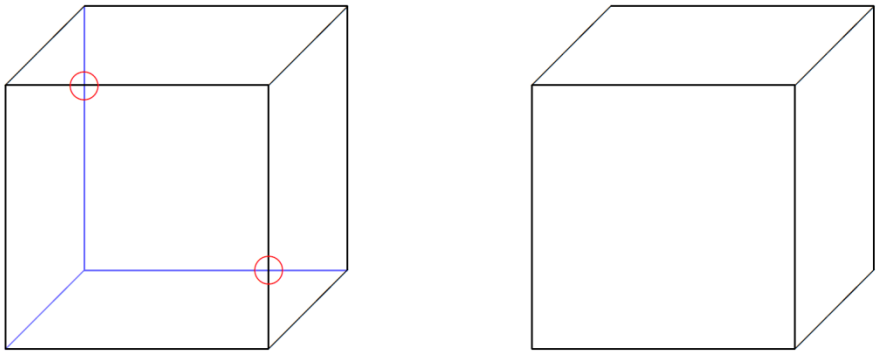
  - Typischerweise ist $\Delta x = 1$ und $^{A}/_{C}$ konstant.
- So ergibt sich eine Subtraktion einer Konstanten entlang einer Scanline:

$$z(x + \Delta x, y) = z(x, y) - const$$

- Gleiches gilt für die Berechnung des ersten z-Wertes der darauffolgenden Scanline!
- Bei vielen Polygonen: Beschleunigung durch Vorsortierung der Objekte nach z-Werten

# Z-Buffer: Drahtgittermodelle

- Drahtgittermodelle: Nur Schnittpunkte der Linien werden durch z-Buffer berücksichtigt.



- Hidden Line-Darstellung (gefüllte Objekte mit Umriss):
  - Objekt als gefülltes Polygon zeichnen: **`glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);`**
  - Objekt erneut als Linienzug zeichnen: **`glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);`**
- Problem: benachbarte Pixel an Polygonkante haben (nahezu) gleiche z-Werte („Stitching“ durch „z-Fighting“)

S. Krömker: Computergrafik I, Universität Heidelberg

# Hidden-Line

- Abhilfe für Stitching: Polygonoffset
  - `glEnable(GL_POLYGON_OFFSET_FILL);`
  - `glEnable(GL_POLYGON_OFFSET_LINE);`
  - `glEnable(GL_POLYGON_OFFSET_POINT);`
- Addiert zum jeweiligen z-Wert einen Offset, der wie folgt berechnet wird:

$$o = m \cdot factor + r \cdot units$$

Aktuell auflösbarer z-Wert

Maximaler z-Unterschied des Objektes

- *factor* und *units* werden vom Anwendungsprogramm vorgegeben:

```
glPolygonOffset(Glfloat factor, Glfloat units);
```

# Hidden-Line

- Wahl von *factor* und *units*:
  - Standardmäßig *factor=1.0* und *units=1.0*
  - Bei größerer Linienstärke sollte *factor* erhöht werden
  - Bei perspektivischer Projektion: *factor* an z-Wert koppeln (Offset größer für weiter entfernte Objekte)
  - Besser zu viel Offset als zu wenig

- Für Hidden-Line-Darstellung zwei Möglichkeiten:
  - Offset für gefülltes Polygon >0 → Polygon erscheint weiter hinten
  - Offset für Linie <0 → Linie erscheint weiter vorne