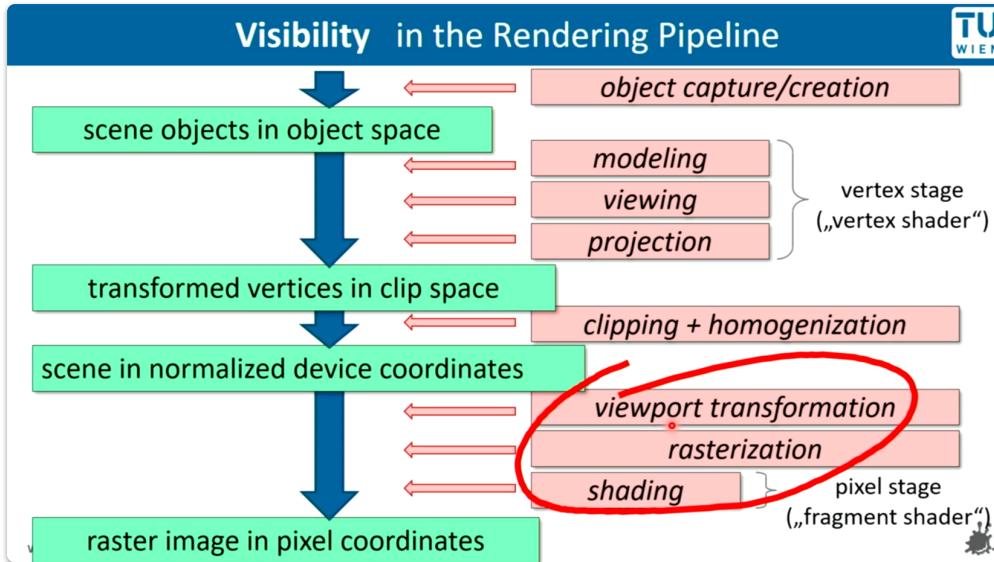


# 8. Sichtbarkeitsverfahren

Das Sichtbarkeitsverfahren kommt hier in dem Bereich der Rendering Pipeline vor:



## 1. Ziel von Sichtbarkeitsverfahren

- **Ziel**: Korrekte und glaubwürdige Darstellung von Szenen, indem unsichtbare Teile der Objekte weggelassen werden.
  - **Unsichtbare Teile**: Rückseiten von Objekten und Teile, die von anderen Objekten verdeckt werden.
- **Begriff**: Hidden-Line- oder Hidden-Surface Eliminierung.

## 2. Ansätze in der Sichtbarkeitsberechnung

- **Objektraum-Methoden**:
  - **Vorgehen**: Vergleichen der Lage der Objekte miteinander.
  - **Ziel**: Nur die vorderen (sichtbaren) Teile der Objekte werden gezeichnet.
- **Bildraum-Methoden**:
  - **Vorgehen**: Für jeden Bildteil wird separat berechnet, was an dieser Stelle sichtbar ist.

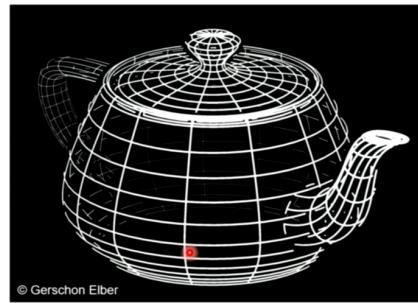
## 3. Berücksichtigung von Transparenz

- Die Erläuterungen zu den Sichtbarkeitsverfahren berücksichtigen **nicht** transparente Objekte.

## Depth Cueing

## 3D Display: Depth Cueing + Visibility

- only visible lines
- intensity decreases with increasing distance



Da geht's darum bei Objekten unsichtbare Polygone anders darzustellen

## Backface Detection (Backface Culling)

### 1. Ziel von Backface Culling

- **Ziel:** Elimination von Polygonen, die sicher nicht sichtbar sind, um den Aufwand nachfolgender Verarbeitungsschritte zu reduzieren.
  - **Nicht sichtbare Polygone:** Polygone, deren Oberflächennormale vom Betrachter weg zeigen.

### 2. Funktionsweise

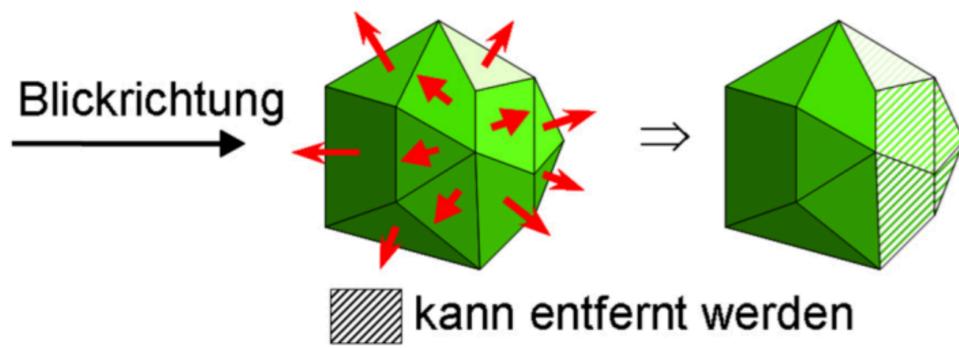
- **Backface Culling** ist kein vollständiges Sichtbarkeitsverfahren, sondern dient als Optimierungsschritt.
  - **Durchschnittliche Reduktion:** Etwa 50% der Polygone werden entfernt.

### 3. Berechnungsverfahren

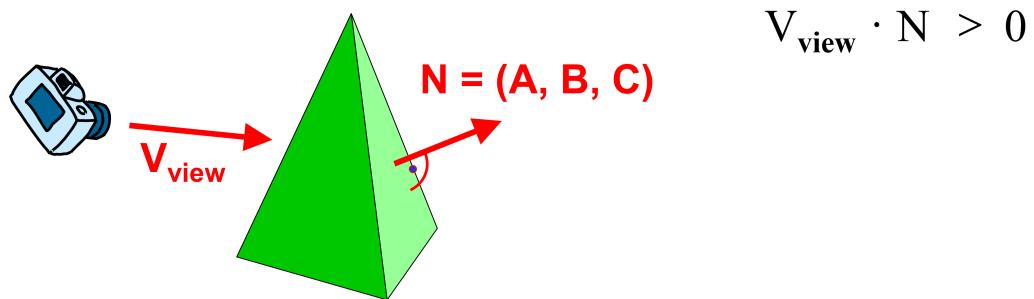
- **Orthographische Projektion:**
  - **Berechnung:** Das Skalarprodukt des Blickrichtungsvektors  $V_{view}$  und der Oberflächennormale  $N$  wird berechnet.
  - **Formel:**  $V_{view} \cdot N > 0 \Rightarrow$  Polygon ist unsichtbar.
- **Perspektivische Projektion:**
  - **Berechnung:** Der Blickpunkt  $(x, y, z)$  wird in die Ebenengleichung eingesetzt.
  - **Formel:**  $Ax + By + Cz + D < 0 \Rightarrow$  Polygon ist unsichtbar.

### 4. Annahmen

- Die gleichen Annahmen wie bei der Verwendung von „Polygonlisten“ werden zugrunde gelegt.



eliminating back faces of closed polyhedra  
 view point  $(x, y, z)$  “inside” a polygon surface if  
 $Ax + By + Cz + D < 0$   
 or polygon with normal  $N = (A, B, C)$  is a back face if

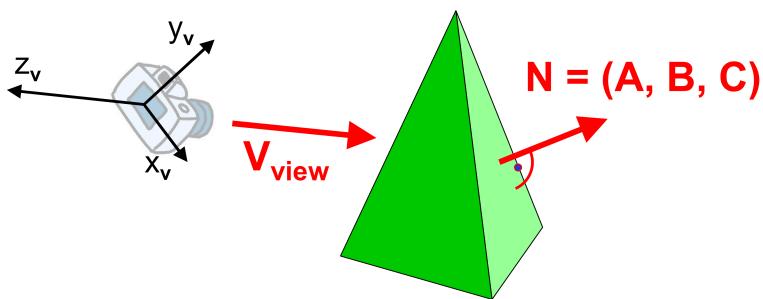


object description in viewing coordinates  $\Rightarrow V_{\text{view}} = (0, 0, V_z)$

$$V_{\text{view}} \cdot N = V_z C$$

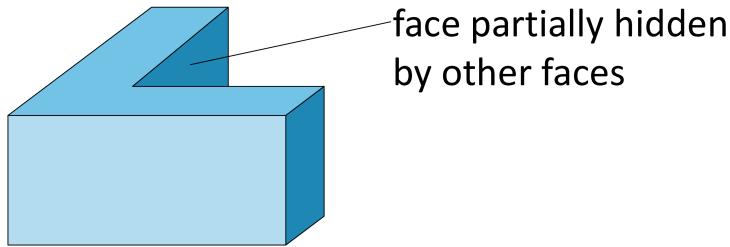
sufficient condition: if  $C \leq 0$  then back-face

↑  
negative !



Der Vektor hat in x und y Koordinaten 0, 0 und nur die Z Koordinate ist relevant.

complete visibility test only for non-overlapping convex polyhedra



... is a preprocessing step for other objects:

about **??** of surfaces are eliminated

## Depth Buffer Verfahren (Z Puffer Verfahren)

[EVC\\_Skriptum\\_CG](#), p.32

### Problemstellung: Sichtbarkeitsproblem

- Ziel: Bestimmung, welche Objekte in einer 3D-Szene für den Betrachter sichtbar sind und welche verdeckt werden.
- Lösung des Z-Puffer-Algorithmus erfolgt pixelweise für eine gegebene Bildauflösung.

### Kernidee des Z-Puffer-Algorithmus

- **Zusätzlicher Speicher:** Neben dem Framebuffer (Farbinformation pro Pixel) wird ein Z-Puffer (oder Tiefenpuffer) benötigt.
- **Z-Wert pro Pixel:** Der Z-Puffer speichert für jedes Pixel die Tiefeninformation (z-Koordinate) des bisher gezeichneten Objekts.
- **Blickrichtung:** Normalerweise in z-Richtung, daher Speicherung des einzelnen z-Wertes ausreichend.

### Speicherbereiche

- **Framebuffer:** Speichert die Farbinformation jedes Pixels.
- **Z-Puffer (Depth Buffer):** Speichert den z-Wert (Tiefe) jedes Pixels.

### Ablauf des Algorithmus

```

for all (x,y)                      // Initialisierung des Hintergrundes
    depthBuff(x,y) = -1             // größtmögliche Entfernung
    frameBuff(x,y) = backgroundColor
for each polygon P                  // Schleife über alle Polygone
    for each position (x,y) on polygon P
        calculate depth z
        if z > depthBuff(x,y) then
    
```

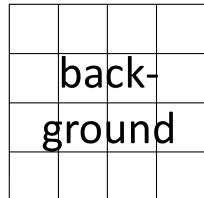
```

depthBuff(x,y) = z
frameBuff(x,y) = surfColor(x,y)
else
    // else nichts !

```

polygons with corresponding z-values

image



depth-buffer

|    |    |           |    |
|----|----|-----------|----|
| -1 | -1 | -1        | -1 |
| -1 | -1 | -1        | -1 |
| -1 | -1 | <b>-1</b> | -1 |
| -1 | -1 | -1        | -1 |

|  |  |       |  |
|--|--|-------|--|
|  |  |       |  |
|  |  | .3 .3 |  |
|  |  | .3 .3 |  |
|  |  |       |  |

|  |  |       |  |
|--|--|-------|--|
|  |  |       |  |
|  |  | .8 .7 |  |
|  |  | .7 .6 |  |
|  |  |       |  |

|  |  |          |  |
|--|--|----------|--|
|  |  |          |  |
|  |  | .6 .5 .4 |  |
|  |  | .6 .5 .4 |  |
|  |  |          |  |

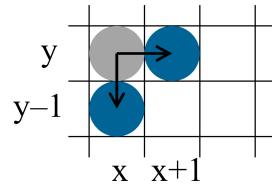
|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## Effizienz

- Inkrementelle Berechnung:** Für ebene Polygone lassen sich die z-Werte natürlich wieder inkrementell effizienter berechnen.



$$Ax + By + Cz + D = 0$$

depth at  $(x, y)$ :

$$z = \frac{-Ax - By - D}{C} \quad \text{constants !}$$

depth at  $(x+1, y)$ :

$$z' = \frac{-A(x+1) - By - D}{C} = z - \frac{A}{C}$$

depth at  $(x, y-1)$ :

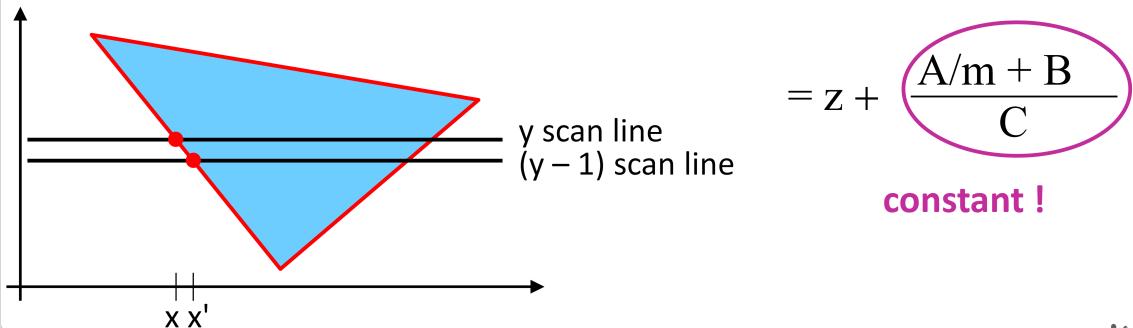
$$z'' = \frac{-Ax - B(y-1) - D}{C} = z + \frac{B}{C}$$

## Vorteile

- Keine Sortierung der Objekte notwendig:** Polygone können in beliebiger Reihenfolge gezeichnet werden.

$$z = \frac{-Ax-By-D}{C}$$

$$y' = y - 1 \\ x' = x - 1/m \Rightarrow z' = \frac{-A(x-1/m)-B(y-1)-D}{C}$$



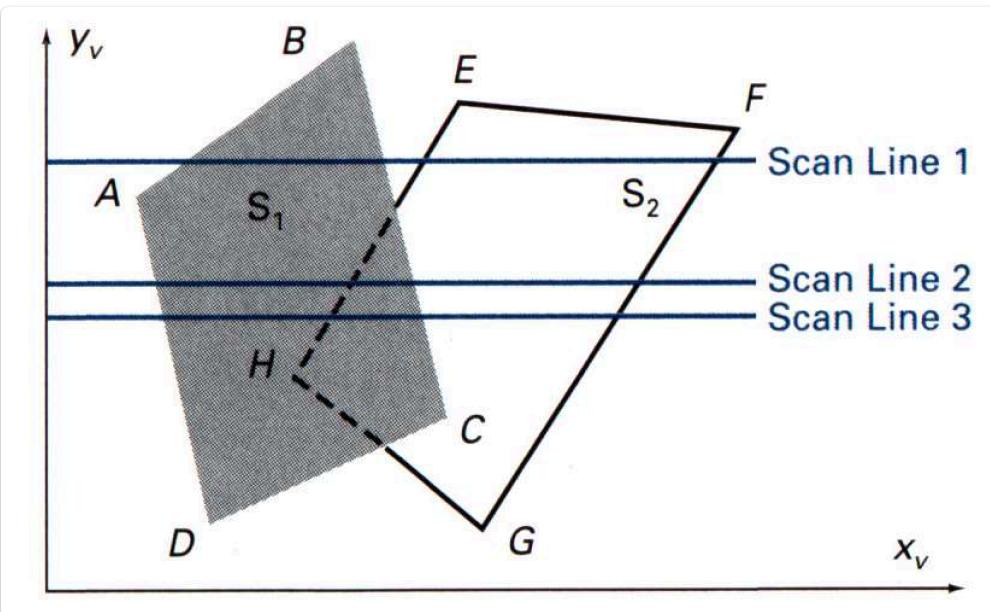
## Anmerkung

- **Viewport-Transformation:** Oft wird  $z$  mit  $-1$  multipliziert bevor das  $z$ -Puffer-Verfahren angewendet wird.

## Scanline-Methode

[EVC\\_Skriptum\\_CG](#), p.32

- **Ziel:** Korrekte Sichtbarkeitsberechnung pixelzeilenweise.
- **Ablauf:** Zeilenweises Vorgehen (z.B. von oben nach unten,  $y$  fallend).
- **Vorteil:** Nutzt die Kohärenz zwischen aufeinanderfolgenden Scanlines (geringe Änderung des Sichtbarkeitsverhaltens).



## Depth-Sorting-Methode (Painter's Algorithm)

[EVC\\_Skriptum\\_CG](#), p.33

- **Grundprinzip:**

- Sortiere alle Polygone nach ihrer Tiefenlage (von hinten nach vorne).
- Zeichne die Polygone in dieser sortierten Reihenfolge.
- **Logik:** Spätere (weiter vorne liegende) Polygone übermalen frühere (weiter hinten liegende) und erzeugen so korrekte Sichtbarkeit.
- **Hauptaufwand:** Sortierung der Polygone.
- **Herausforderung:** Sicherstellen, dass kein Polygon ein anderes verdeckt, das in der Sortierreihenfolge später kommt (weiter vorne liegt).
- **Vorgehensweise:**
  1. **Grobsortierung:** Schnelle erste Sortierung der Polygone.
  2. **Überprüfung:** Testen, ob die Sortierung korrekt ist (keine fehlerhaften Verdeckungen).
  3. **Umsortierung (ggf.):** Bei Bedarf Anpassung der Reihenfolge, um korrekte Sichtbarkeit zu gewährleisten.

surfaces sorted in order of decreasing depth (viewing in  $-z$ -direction)

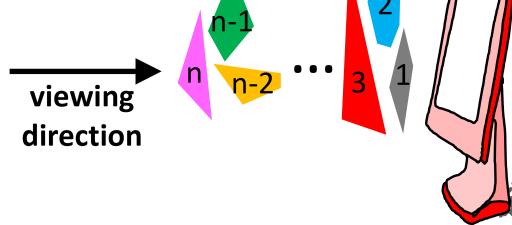
- (1) "approximate"-sorting using smallest  $z$ -value (greatest depth)
- (2) fine-tuning to get correct depth order

surfaces scan converted in order

sorting both in image and object space

scan conversion in image space

also called "painter's algorithm"



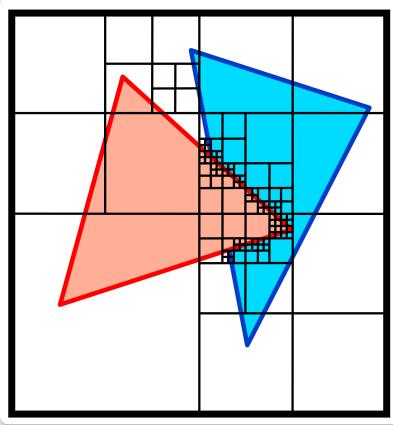
Werner Purgathofer

23

## Area-Subdivision Methode

[EVC\\_Skriptum\\_CG](#), p.33

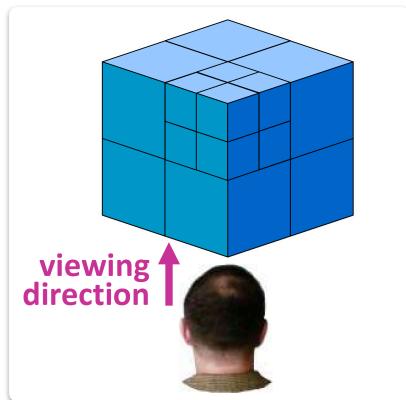
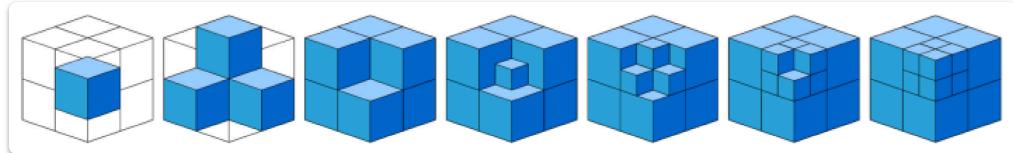
- **Grundidee:** Divide-and-Conquer-Ansatz für das Sichtbarkeitsproblem.
- **Analogie:** Ähnlich der Quadtree-Repräsentation von Bildern.
- **Vorgehensweise:**
  1. **Einfache Fälle:** Sichtbarkeitsproblem wird in grober Auflösung gelöst.
  2. **Komplizierte Fälle:**
    - Unterteilung der aktuellen Bildfläche in vier gleich große Viertel.
    - Rekursive Anwendung der Methode auf jede der vier Teilstufen.
- **Garantie:** Die rekursive Unterteilung bis zur maximalen Bildauflösung stellt eine pixelgenaue Lösung des Sichtbarkeitsproblems sicher.



## Octree-Methode

EVC\_Skriptum\_CG, p.33

- **Datenstruktur:** Repräsentation der Szene als Octree (raumorientierter Baum).
- **Vorteil der Datenstruktur:** Implizites Wissen über die Tiefenordnung (was vorne und hinten ist) für jede Blickrichtung.
- **Rendering-Strategien:**
  - **Von hinten nach vorne:**
    - Rekursives Durchlaufen der Octree-Knoten.
    - Rendern der Teilwürfel in der Reihenfolge: *entferntester* -> 3 nächstnäheren -> nächste 3 -> vorderster.
    - Beispiel für frontale Blickrichtung

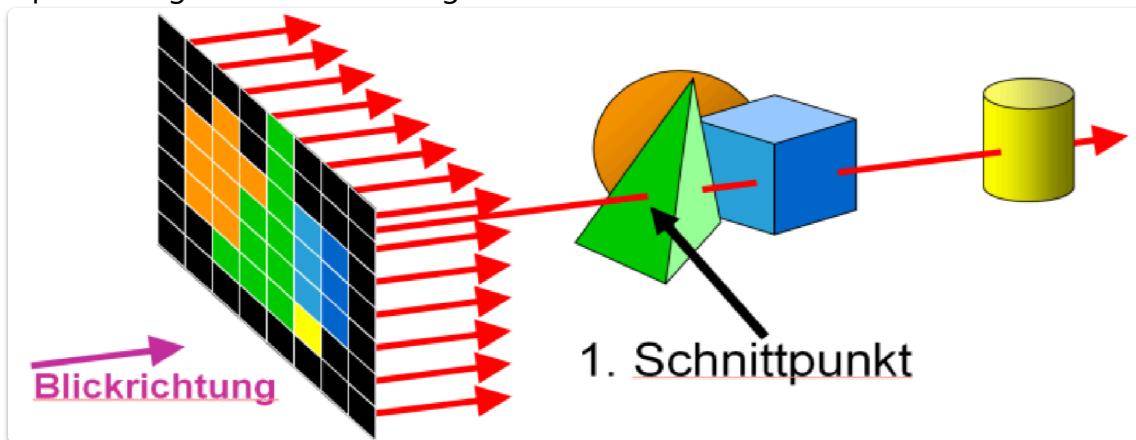


- **Von vorne nach hinten:**
  - Rekursives Durchlaufen der Octree-Knoten.
  - Zeichnen nur der sichtbaren Bereiche.
  - Notwendigkeit, sich bereits gezeichnete Bereiche zu merken, um Overdraw zu vermeiden.
- **Genereller Vorteil:** Das inhärente Wissen über die Tiefenordnung im Vergleich zu anderen Datenstrukturen vereinfacht die Sichtbarkeitsbestimmung.

# Ray-Casting

EVC\_Skriptum\_CG, p.33

- **Grundprinzip:** Berechnung der Sichtbarkeit für jedes Pixel einzeln.
- **Ablauf:**
  1. **Blickstrahl:** Vom Pixel in Blickrichtung wird eine gerade Linie (Blickstrahl) in die Szene projiziert.
  2. **Schnittpunktberechnung:** Der Blickstrahl wird mit allen Objekten/Polygonen in der Szene geschnitten.
  3. **Nächster Schnittpunkt:** Aus der Menge der Schnittpunkte wird derjenige ausgewählt, der am nächsten zum Betrachter liegt.
  4. **Pixelfarbe:** Die Farbe der Oberfläche am nächsten Schnittpunkt bestimmt die Farbe des betrachteten Pixels.
- **Vorteile:**
  - Ermöglicht das Rendern verschiedenster Oberflächen (nicht nur Polygone), sofern der Schnitt mit einer Geraden berechenbar ist (z.B. Freiformflächen).
- **Benötigte Information (für Schattierung):** Oberflächennormale am Auftreffpunkt des Strahls.
- **Nachteile:**
  - **Hoher Rechenaufwand:** Für jedes Pixel müssen Schnittberechnungen mit potenziell vielen Objekten durchgeführt werden (Millionen von Pixeln und möglicherweise tausende bis Millionen von Objekten).
  - **Notwendigkeit:** Effiziente Implementierung der Schnittpunkttests und weitere Optimierungen zur Reduzierung des Rechenaufwands sind unerlässlich.

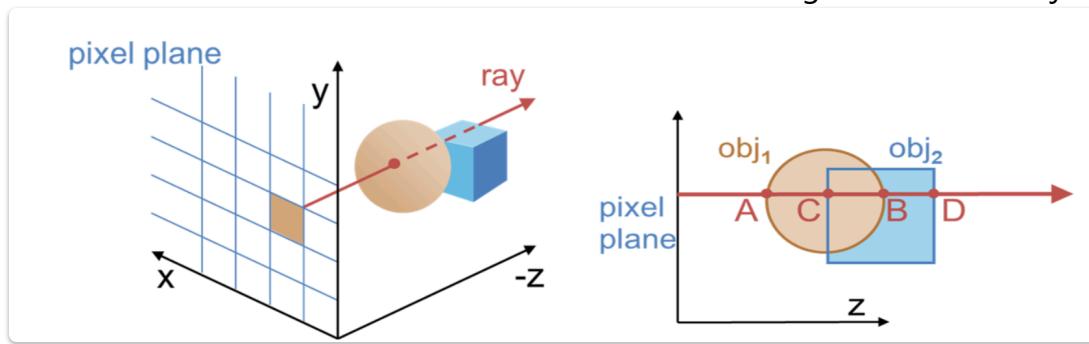


## Ray-Casting von CSG-Objekten

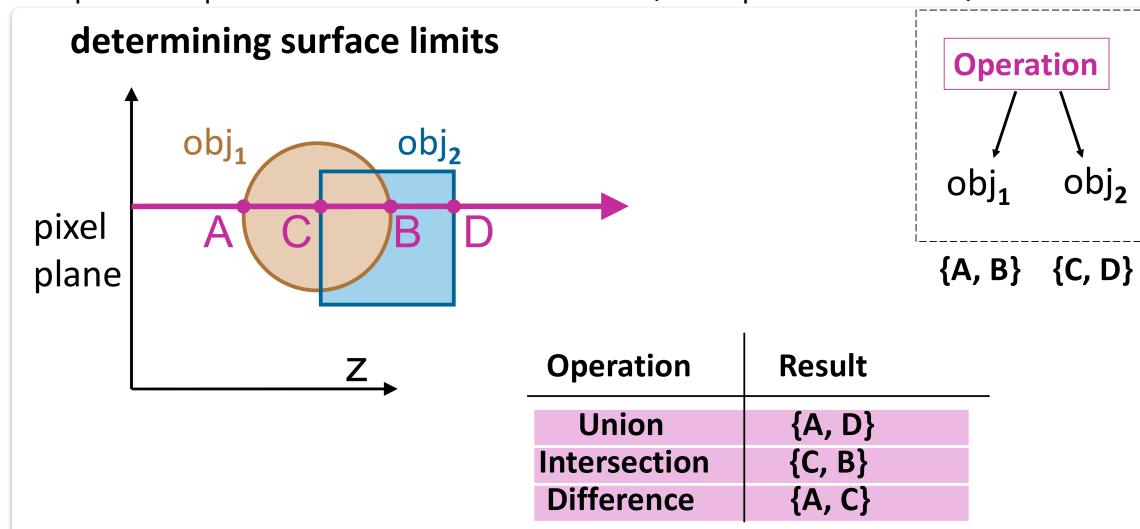
EVC\_Skriptum\_CG, p.33

- **Gebräuchliche Methode:** Pixelweise Berechnung des Bildes durch Ray-Casting.
- **Ablauf pro Pixel:**
  1. **Ray-Erzeugung:** Ein Blickstrahl (Ray) wird in Blickrichtung "ausgeworfen".
  2. **Schnitt mit allen Objekten:** Der Ray wird mit allen Objekten der Szene geschnitten.

3. **Vorderster Schnittpunkt:** Der Schnittpunkt, der am nächsten zum Betrachter liegt, bestimmt das sichtbare Objekt.
4. **Pixelfarbe:** Das Pixel erhält die Farbe des am vordersten geschnittenen Objekts.



- **Rekursive Berechnung bei CSG-Bäumen:**
  - **Endknoten (Primitive Objekte):** Direkte und einfache Berechnung aller Ray-Objekt-Schnittpunkte.
  - **Zwischenknoten (Boolesche Operationen):**
    - Die Schnittpunktlisten der beiden Kindknoten werden entsprechend dem Booleschen Operator verknüpft:
      - **Vereinigung (Union):** Kombination der Schnittpunktlisten (Beispiel: A, D).
      - **Durchschnitt (Intersection):** Schnittmenge der Schnittpunktlisten (Beispiel: C, B).
      - **Differenz (Subtraction):** Schnittpunkte des ersten Objekts, die nicht im zweiten Objekt liegen (Beispiel: A, C).
  - **Wurzelknoten:** Der erste Punkt (der dem Betrachter am nächsten liegt) der resultierenden verknüpften Schnittpunktliste wird ausgewählt.
- **Relation zu Ray-Tracing:** Ray-Casting ist eine vereinfachte Version von Ray-Tracing, das komplexere optische Effekte simulieren kann (wird später behandelt).



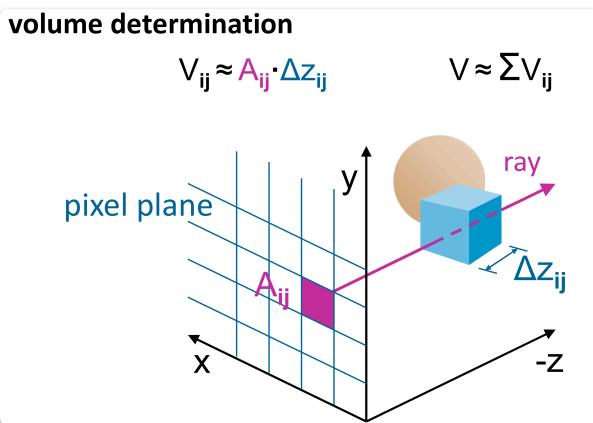
Ray-Casting ist eine vereinfachte Version von Ray-Tracing, mit dem noch viele weitere optische Effekte simuliert werden können. Dazu kommen wir in einem späteren Kapitel.

**Ray-Casting** = für jedes Pixel der Darstellungsfläche:

- erzeuge eine Gerade durch das Pixel in Blickrichtung („Blickstrahl“)
- schneide den Blickstrahl mit allen Objekten
- wähle aus der Schnittpunktliste den zum Betrachter nächsten Punkt
- färbe das Pixel mit der Farbe der Oberfläche dieses Punktes

## Ergänzung von den Slides

Man kann das auch verwenden um das Volumen zu berechnen



## Klassifizierung der Verfahren

[EVC\\_Skriptum\\_CG](#), p.34

Wir wollen nun noch überlegen, welche Verfahren im Objektraum arbeiten und welche im Bildraum. Dies ist nicht immer ganz eindeutig klassifizierbar, aber im Großen und Ganzen gilt:

### Objektraum-Verfahren:

- Backface Detection
- Depth Sorting
- Octree-Methode

### Bildraum-Verfahren:

- Z-Puffer
- Scanline-Methode
- Area Subdivision
- Ray Casting