

## 10. Sichtbarkeitstransformation und Visible Surface Determination

### *Betrachtungstransformationen (Viewing transformations)*

In der Regel wird bei der Implementierung der Projektion in die Bildschirmenebene auch das Clipping vorbereitet.

Anstelle der Projektion auf eine Ebene und einem Clippen in dieser Ebene transformiert man in ein *kanonisches Sichtvolumen* (Sichtkörper, i.allg. ein Quader) und clippt an dessen Seitenflächen.

Dann: Projektion in Ebene auf einheitliche Weise.

Im Folgenden Darstellung nach van Dam (2001) und Schlechtweg (2001).

### **Der Sichtkörper**

- engl.: view volume
- Sichtkörper enthält alles, was vom COP (oder DOP) aus sichtbar ist (=was die Kamera sieht)
- normalerweise ein Kreiskegel → teure Berechnungen, insbesondere beim Clipping von Objekten gegen eine Kegelfläche
- Approximation durch Pyramide(nstumpf)
  - die Ausgabe ist sowieso rechteckig
  - Clipping durch Lösen eines linearen Gleichungssystems
- Pyramide(nstumpf) = view frustum

# Vereinfachtes Kameramodell

Sechs Parameter zur Spezifikation der Kamera

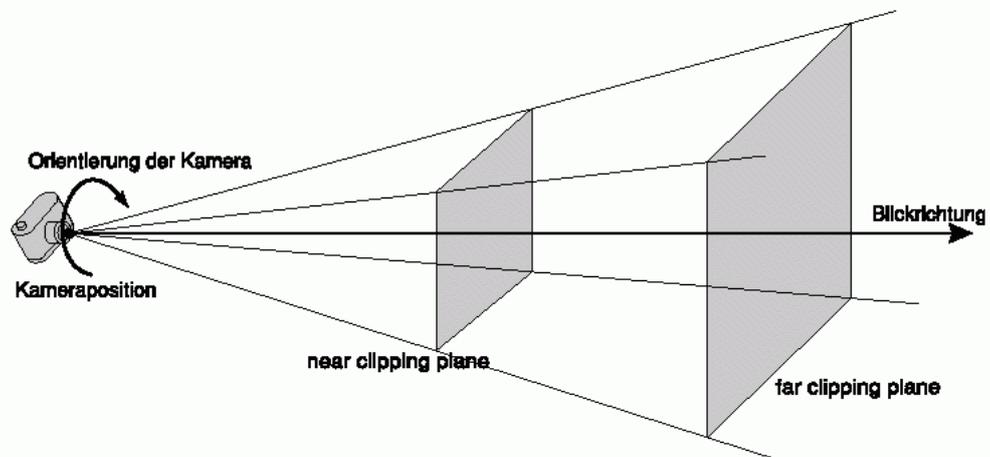
1. Position der Kamera
2. (Blick-)Richtung der Kamera
3. Orientierung der Kamera
4. Seitenverhältnis des zu erzeugenden Bildes
5. Öffnungswinkel
6. near clipping plane und far clipping plane
7. optional: Brennweite

## Orientierung der Kamera

Spezifikation durch zwei Angaben

- Blickrichtung
  - Angabe eines Zielpunktes,  
Berechnung der Richtung aus Zielpunkt – Position
  - Richtung, in die die Kamera „schaut“
  - Spezifikation durch einen Vektor in 3D
- up vector
  - bestimmt die Rotation der Kamera um die Blickrichtung
  - Projektion des up vectors muß in der Ebene senkrecht zur Blickrichtung liegen, d.h. Blickrichtung und up vector dürfen nicht kollinear sein

## Sichtkörper für dieses Modell



## Kameraposition

- analog zur Entscheidung eines Photographen, wo die Kamera aufgestellt wird
- Spezifikation durch einen Punkt in 3D

## Seitenverhältnis

- engl.: aspect ratio
- analog zum Seitenverhältnis des Films in der Photographie
- Verhältnis von Breite zu Höhe
- für quadratische Bilder: 1 : 1
- Kinofilme: 2 : 1
- Fernsehen: 4 : 3
- HDTV: 16 : 9

## Öffnungswinkel

- analog zur Auswahl des Objektivs beim Photographieren
- bestimmt die Stärke der perspektivischen Verzerrung
  - keine Verzerrung – Parallelprojektion
  - sehr starke Verzerrung – Weitwinkellinsen
- eigentlich zwei Öffnungswinkel im view frustum: horizontal und vertikal
- sollten übereinstimmen (Spezifikation eines Winkels, der andere kann aus Seitenverhältnis berechnet werden)

## Clipping planes

- Raum zwischen near clipping plane und far clipping plane bestimmt, was die Kamera sieht
- Position der Ebenen durch Entfernungen entlang der Blickrichtung festgelegt
- Objekte außerhalb dieser Ebenen werden nicht gezeichnet
- diese Ebenen schneidende Objekte müssen geclippt werden

### Gründe für die Verwendung der Clipping planes

- near clipping plane
  - Objekte zu nah an der Kamera sollen nicht dargestellt werden, da:
    - \* Sicht auf Rest der Szene blockiert würde
    - \* zu starke Verzerrung
  - Objekte hinter der Kamera sollen nicht dargestellt werden
- far clipping plane
  - Objekte zu weit von der Kamera entfernt sollen nicht dargestellt werden, da sie visuell nicht signifikant (kaum erkennbar) sind, aber in die Berechnungen einbezogen werden müssen

## Kameramodell . . .

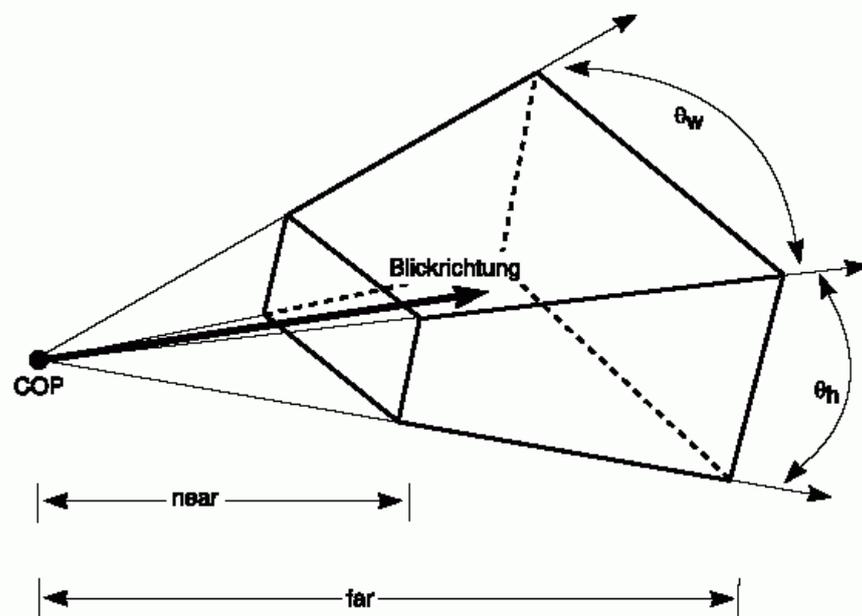
- . . . kann Sichtkörper für folgende Projektionen erstellen:
  - perspektivische Projektion mit positivem Öffnungswinkel
  - Parallelprojektion (Öffnungswinkel ist Null)
- . . . kann nicht für schräge Projektionen verwendet werden

Welche Transformation ist rechnerisch anzuwenden, um die *Betrachtungstransformation* (Abbildung in den Bildschirmausschnitt) durchzuführen?

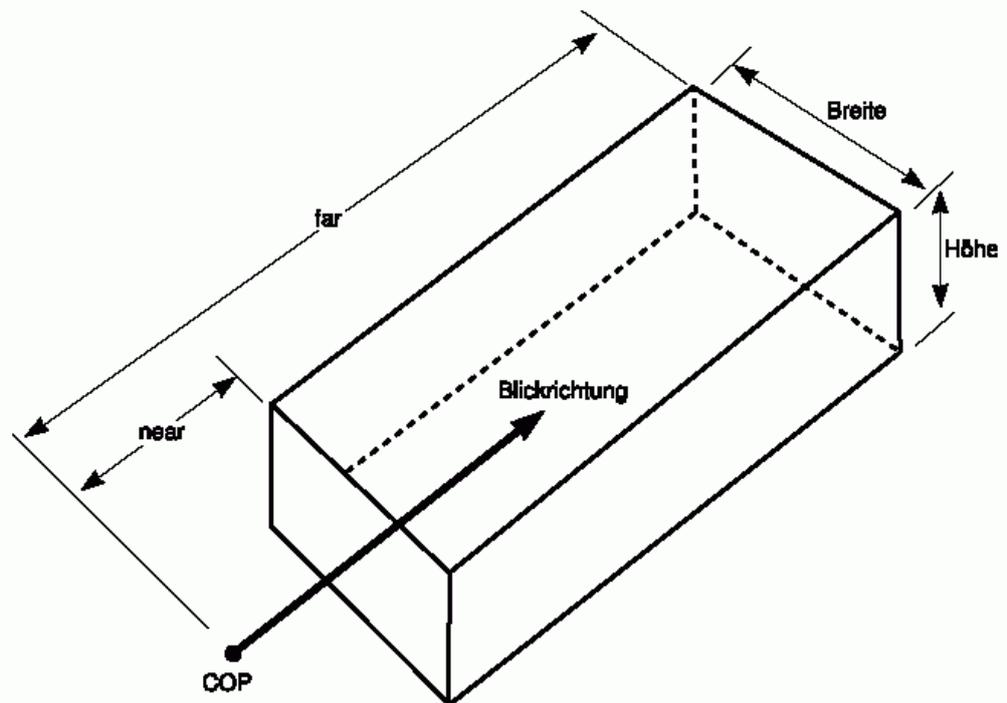
Ziel: Herleitung einer Transformationsmatrix (für Objektdarstellung in homogenen Koordinaten)

- Vorgehensweise:
  - Der kanonische Sichtkörper
  - Spezifikation des Sichtkörpers (bereits behandelt)
  - Transformation des spezifizierten Sichtkörpers in den kanonischen Sichtkörper
  - Projektion und Clipping an diesem kanonischen Sichtkörper

## Sichtkörper für perspektivische Projektionen



## Sichtkörper für Parallelprojektionen

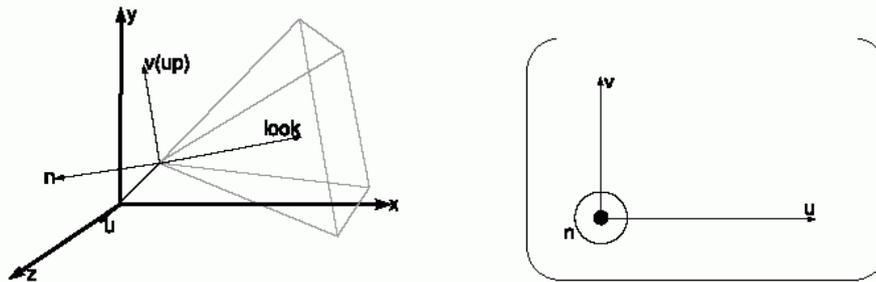


## Spezifikation einer beliebigen 3D-Ansicht

- Platzierung des Sichtkörpers spezifiziert durch:
  - Position (COP)
  - Orientierung (Blickrichtung, up vector)
- Form des Sichtkörpers spezifiziert durch
  - Öffnungswinkel
  - near clipping plane, far clipping plane
- perspektivische Projektion: Projektionsstrahlen schneiden sich im COP
- Parallelprojektion: Projektionsstrahlen parallel zur DOP, schneiden sich nie

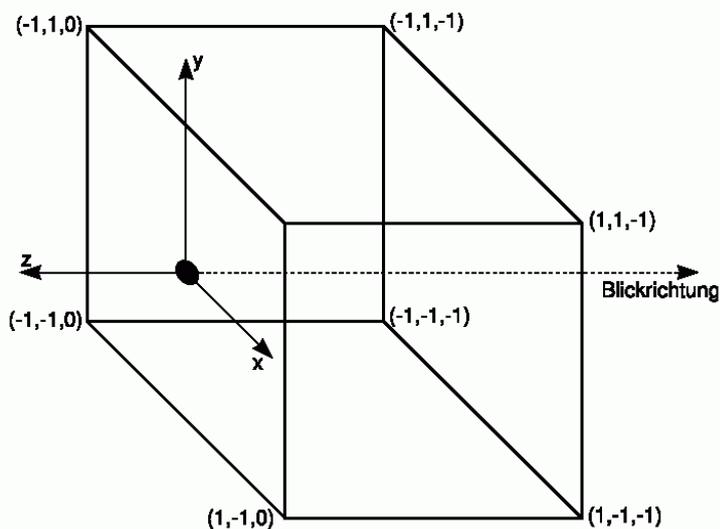
# Koordinatensysteme

- Weltkoordinaten – Definition des Modells, der Kameraparameter
- View-Referenz-Koordinaten (Kamerakoordinaten) – Ursprung im COP, Achsen rotiert je nach Orientierung der Kamera



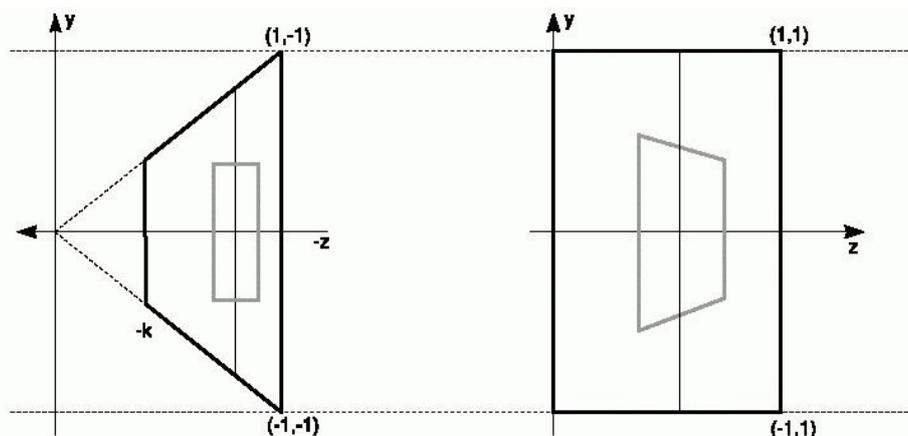
## Der kanonische Sichtkörper

- beliebiger Sichtkörper zu komplex (Clipping und Projektion)
- Bilderzeugung einfacher vom kanonischen Sichtkörper aus:
  - Parallelprojektion
  - Kamera im Ursprung (Position:  $(0, 0, 0)$ )
  - Sicht entlang der negativen  $z$ -Achse (look vector:  $(0, 0, -1)$ )
  - aufrecht orientiert (up vector:  $(0, 1, 0)$ )
  - Ausdehnung der Bildebene zwischen  $-1$  und  $1$  in  $x$  und  $y$
- für perspektivische Projektion ein zusätzlicher Schritt: Umwandlung des perspektivischen Sichtkörpers in den der Parallelprojektion



Vorgehen in einzelnen Schritten:

1. Verschiebung des Projektionszentrums (COP) in den Koordinatenursprung
2. Rotation zur Ausrichtung der Kamera (Drehmatrix: Inverse der Matrix  $M$ , deren Spalten den Orientierungsvektoren der Kamera entsprechen; diese Matrix ist orthogonal  $\rightarrow M^{-1} = M^T$ )
3. Skalierung in der  $xy$ -Ebene, um den Öffnungswinkel zu berücksichtigen
4. äquiforme Skalierung, um die far-clipping-planes anzugleichen
5. falls Sichtkörper für perspektivische Projektion vorlag: dessen *projektive Transformation* in den kanonischen Sichtkörper für die Parallelprojektion (Quader)



[ebenfalls als eine  $4 \times 4$ -Matrix  $D$  in homogenen Koordinaten darstellbar]

genauere Herleitung und Spezifikation der Matrizen siehe  
[http://www-gs.informatik.tu-cottbus.de/cg2\\_v07a.pdf](http://www-gs.informatik.tu-cottbus.de/cg2_v07a.pdf)

## Endergebnis

- endgültige Transformation:

$$p' = DS_{xyz} \left( \frac{1}{\text{far}} \right) S_{xy} M^T T(-\text{COP}) p$$

- Die Matrizen  $D$ ,  $S_{xyz} \left( \frac{1}{\text{far}} \right)$ ,  $S_{xy}$ ,  $M^T$ ,  $T(-\text{COP})$  können alle berechnet werden, wenn die Kameraparameter bekannt sind.
- Erzeugen einer zusammengesetzten Transformationsmatrix, mit der alle Szenenkoordinaten multipliziert werden, um diese von Weltkoordinaten in den Standard-Sichtkörper für Parallelprojektionen zu überführen.

## Clipping

- Transformierte Szene paßt in einen Quader nahe des Koordinatenursprungs
- noch notwendig: Clipping der Szene gegen die Seiten des Sichtkörpers
- normalisierter Sichtkörper mit folgender Ausdehnung:

$$-1 \leq x \leq 1$$

$$-1 \leq y \leq 1$$

$$0 \leq z \leq 1$$

- Clipping ist jetzt sehr einfach!!!  $x$ - und  $y$ -Komponenten gegen  $\pm 1$  testen,  $z$ -Komponenten gegen 0 und 1

es kann eine Variante des Cohen-Sutherland-Algorithmus angewandt werden:

6-Bit-Binärcode anstatt 4-Bit im planaren Fall  
Bit 5: gesetzt für Punkte  $(x, y, z)$  oberhalb des Sichtvolumens  
Bit 4: unterhalb  
Bit 3: rechts  
Bit 2: links  
Bit 1: hinter dem Sichtvolumen  
Bit 0: vor dem Sichtvolumen

damit dann analoge Oder- und Und-Abfrage wie im planaren Cohen-Sutherland-Algorithmus.

(Vgl. Rauber 1993, S. 147 f.)

## Projektion

- Projektion in die  $xy$ -Ebene durch „Weglassen“ der  $z$ -Koordinate
- dabei Mapping auf (Pixel-)Koordinaten des Bildschirms
- Seien  $(x, y, z)$  Koordinaten nach Normalisierungstransformation und Clipping, dann ergeben sich die Screen-Koordinaten  $(x', y')$  aus:

$$x' = \left\lfloor \frac{W}{2}(x + 1) \right\rfloor$$
$$y' = \left\lfloor \frac{H}{2}(y + 1) \right\rfloor$$

mit  $W$  als Bildschirmbreite und  $H$  als Bildschirmhöhe

## Zusammenfassung

- Reduktion des gesamten Problems auf Multiplikation der Koordinaten mit einer zusammengesetzten Matrix, die aus der Spezifikation der Kameraparameter gewonnen werden kann
- Zusätzlich dazu noch die Modelltransformationen
- $\rightarrow$  Transformation lokaler Koordinaten des Modells  $p$  in Bildschirmkoordinaten  $p'$  ist Matrixmultiplikation:

$$p' = N M p$$

mit der Normalisierungstransformation  $N$  und der zusammengesetzten Modell-Transformationsmatrix  $M$

## *Visibilitätsrechnung*

"Hidden Surface Removal" (HSR)

"Visible Surface Determination" (VSD)

*Problem der Verdeckung* (Unsichtbarkeit von Teilen einer Szene) entsteht dadurch, dass Projektionen nicht injektiv sind

⇒ mehrere Objektpunkte haben denselben Bildpunkt ("Projektionsäquivalenz"), aber nur einer davon ist "sichtbar" (= bestimmt Farbe u. Intensität des betr. Pixels).

Im Folgenden:

- Problemdefinition
- Konservative Sichtbarkeitstests für die Vorverarbeitung
- Algorithmen (hier nur drei, es gibt sehr viel mehr!)

### *Problemdefinition:*

Gegeben sei eine Menge von 3D-Objekten und die Spezifikation einer Ansicht dieser Szene (Kamera-modell).

*Problem:* Bestimme, welche Linien oder Flächen der gegebenen Objekte unter der gegebenen Ansicht sichtbar sind.

Drei Klassen von Algorithmen:

1. Bildraumalgorithmen ( $\rightarrow$  *image precision*)

- Sichtbarkeit wird für diskrete Bildpunkte bestimmt
- Beispiel: z-Buffer

2. Objektraumalgorithmen ( $\rightarrow$  *object precision*)

- exakte Sichtbarkeitsbestimmung im Modellraum
- Beispiele: Clipping von Polygonen an Polygonen, 3D-Tiefensortierung, BSP-Bäume

3. Hybride Algorithmen

- arbeiten sowohl im Objekt- als auch im Bildraum

Zusätzlich: "konservative Sichtbarkeitstests"

(Vorverarbeitung): Aussondern von mehr oder weniger "trivialen" Fällen. Beispiel: Back-face culling.

*Back-face culling*

Entfernen von nicht sichtbaren Rückseiten, d.h. von allen Polygonen, die vom Betrachter weg zeigen (diese Rückseiten machen etwa die Hälfte aller Flächen aus!)

setzt voraus:

- Objekte sind als Polygonnetze modelliert, der Betrachterstandpunkt liegt außerhalb
- Polygone sind orientiert: Festlegung von "innen" und "außen" hinsichtlich des modellierten Körpers.

Testgrundlage: nach außen zeigende Normalen der Polygone.

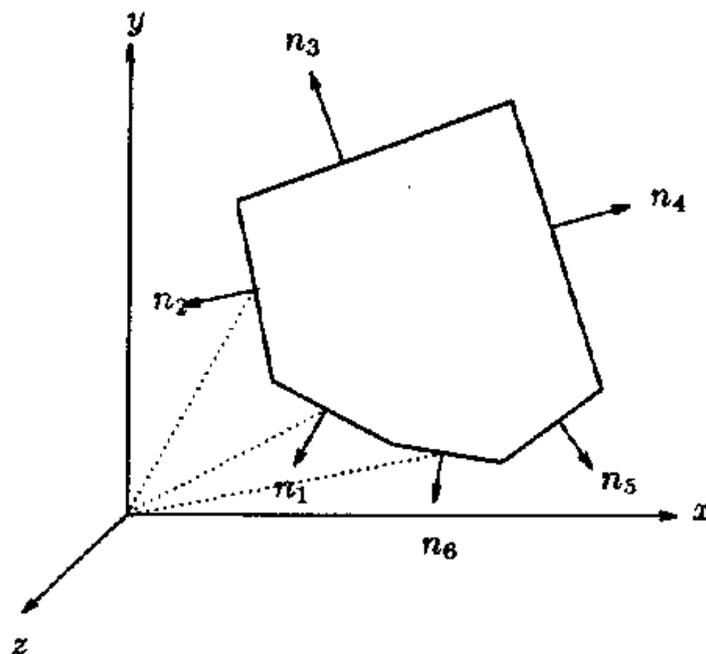
Zwei Möglichkeiten der Betrachtung:

- Line-of-Sight-Interpretation
- Halbraum-Interpretation

*Line of Sight* (LOS):

Strahl vom Betrachterstandpunkt (COP) zu einem beliebigen Punkt des Polygons

(bei Parallelprojektion ist die LOS identisch mit der DOP)



Zeigt die Normale zur selben Seite wie die LOS (Winkel  $< 90^\circ$ ), dann handelt es sich bei dem Polygon um eine zu entfernende Rückseite.

Rechnerisch: Skalarprodukt prüfen

$\text{LOS} \cdot \text{Normale} > 0 \Rightarrow$  unsichtbar

$\text{LOS} \cdot \text{Normale} \leq 0 \Rightarrow$  möglicherweise sichtbar

Halbraum-Interpretation:

Normalenform der Ebenengleichung für die Polygon-Ebene:

$$(\text{Normale} \cdot x) + D = 0$$

Die Ebene teilt den übrigen Raum in zwei Halbräume:

$(\text{Normale} \cdot x) + D > 0$ :  $x$  liegt im positiven Halbraum

$(\text{Normale} \cdot x) + D < 0$ :  $x$  liegt im negativen Halbraum

Polygon zeigt vom Betrachter weg, wenn sich der Betrachterstandpunkt (COP) im negativen Halbraum befindet:

unsichtbar, wenn  $(\text{Normale} \cdot \text{COP}) + D < 0$

beide Interpretationen sind letztlich äquivalent (LOS lässt sich aus COP und Polygonpunkt berechnen).

# Algorithmen zur Visible Surface Determination

## 1. Z-Buffer-Algorithmus

häufig in der Hardware implementiert

z-Buffer (z-Puffer, Tiefenpuffer): pixelbezogener Speicher für die z-Werte

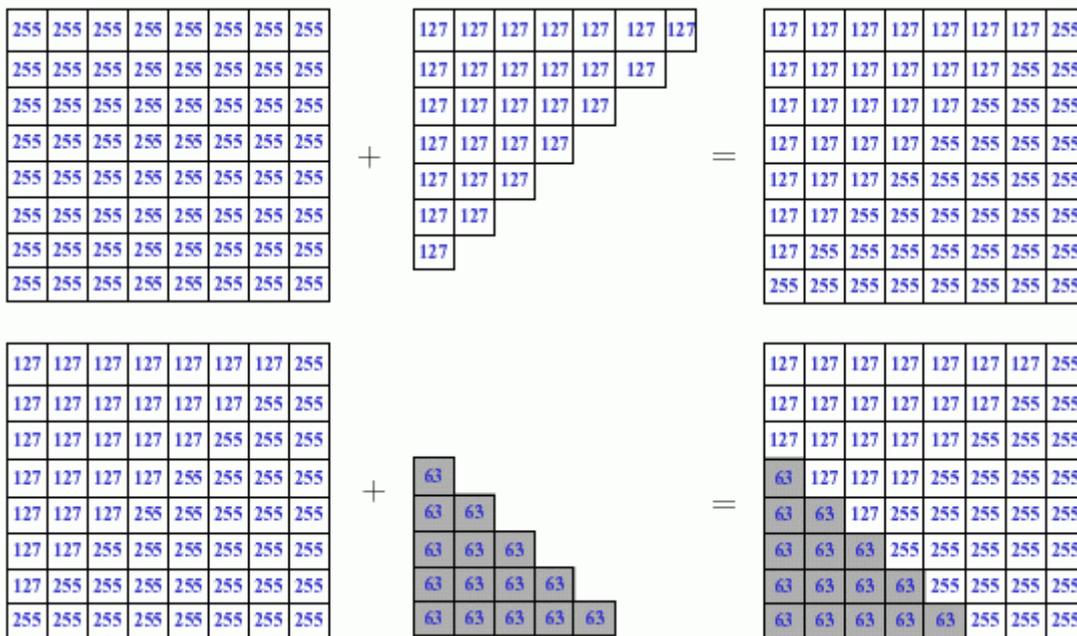
- schon 1974 vorgeschlagen, aber damals nicht realisiert
- heute dominierender Algorithmus für Rastergeräte

Vorgehensweise:

- zusätzlich zum Framebuffer (Bild) noch eine zweite Bitmap, die die z-Werte speichert
- z-Buffer initialisiert mit Hintergrundwert (Tiefe der am weitesten entfernten Ebene des Sichtkörpers,  $z = 1$ )
- z-Wert jedes Pixels wird mit dem Wert im z-Buffer verglichen
  - $z$  aus Ebenengleichung des aktuellen Polygons berechnen
  - besser:  $z$  an den Eckpunkten berechnen und interpolieren
- ist aktueller z-Wert kleiner als der an dieser Position gespeicherte: aktuellen Wert in den z-Buffer speichern und Pixel zeichnen
- sonst keine Änderungen

die Polygone können in jeder beliebigen Reihenfolge gezeichnet werden

Beispiel:



(hier z-Werte nicht zwischen 0 und 1, sondern zwischen 0 und 255; implementationsabhängig)

Algorithmus:

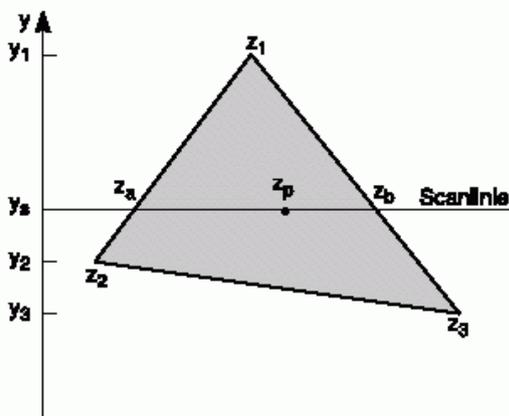
```

initialisiere z-Buffer
foreach Polygon do
    foreach Pixel in der Projektion des Polygons do
         $p_z = z\text{-Wert des Polygons an Position } (x, y)$ 
        if  $p_z < z\text{-Buffer}(x, y)$  then
            WritePixel( $x, y, c$ )
            WriteZ( $x, y, p_z$ )
        fi
    od
od

```

Effiziente Implementierung: z-Werte im Inneren der Polygone durch Interpolation bestimmen

- ähnliche Vorgehensweise wie beim Scanlinien-Algorithmus zum Polygonfüllen
- Interpolation zunächst entlang der Polygon-Kanten, dann entlang der Scanlinien im Inneren



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

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

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

- erweiterbar zu inkrementellem Verfahren zur Bestimmung der z-Werte

- Vorteile des z-Buffer-Algorithmus

- einfach auch in Hardware zu implementieren → **schnell**
- Polygone können in beliebiger Reihenfolge bearbeitet werden
- jedes Polygon einzeln behandelt
- kann auch für nicht-polygonale Flächen genutzt werden

- Nachteile des z-Buffer-Algorithmus

- Genauigkeitsproblem, da z-Werte durch perspektivische Verkürzung „komprimiert“ werden
- kein Antialiasing
- *alle* Polygone müssen behandelt werden

- Transparenz ist nicht realisierbar

Verbesserung:

### *Hierarchischer z-Buffer-Algorithmus*

Ersetze z-Buffer durch "z-Pyramide"

tiefste Ebene: z-Buffer in maximaler Auflösung

höhere Ebenen: Jedes Pixel repräsentiert die maximale Tiefe der vier Pixel "unter" ihm

Grundidee: hierarchische Rasterung des Polygons;  
früher Abbruch, wenn Polygon verdeckt ist.

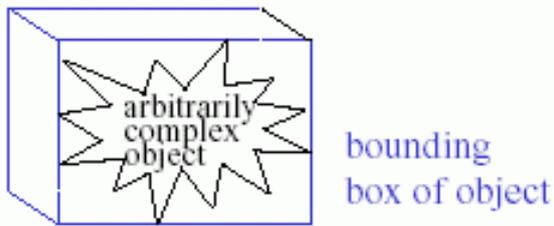
- Polygon zuerst gegen die höchste Ebene testen
- Wenn das Polygon weiter entfernt ist als die Tiefe, die im entspr. (Makro-) Pixel gespeichert ist: verdeckt
- wenn näher: Test gegen die nächstniedrige Ebene, usw.
- Wenn Polygon auf der tiefsten Ebene sichtbar ist, dann zeichnen und z-Pyramide aktualisieren.

Nutzung von Kohärenz in zweierlei Hinsicht möglich:

- Ein in einem Pixel verdecktes Polygon ist wahrscheinlich auch in benachbarten Pixeln verdeckt (Bildraum-Kohärenz, wird inhärent von der Pyramide genutzt)
- Polygone nahe einem verdeckten Polygon sind möglicherweise auch verdeckt (Objektraum-Kohärenz).

Verbesserung des Algorithmus durch Ausnutzung der Objektraum-Kohärenz:

- Unterteile die Szene durch einen *Octree*
- Geometrische Objekte in einem Knoten des Octrees sind in einem Würfel enthalten (*bounding volume*).
- Bevor der Inhalt des Würfels gerendert wird: Teste die Seiten des Würfels gegen die z-Pyramide!



- Wenn die Seitenflächen des Würfels verdeckt sind, dann ist auch die Geometrie im Inneren des Würfels verdeckt – gesamten Inhalt ignorieren.

(Dieser Ansatz ist ein Beispiel für eine "bounding volume hierarchy - Technik", wovon es noch andere Varianten gibt.)

## 2. BSP-Bäume (Binary Space Partition)

Objektraum-Verfahren für die Sichtbarkeitsrechnung, entwickelt von Fuchs, Kedem & Naylor um 1980

- sehr effizient für statische Szenen und möglicherweise wechselnden Betrachterstandpunkt
- zeit- und speicherintensives Preprocessing, aber lineare Zeit für Display
- Preprocessing nur einmal nötig, kann für alle Betrachterstandpunkte genutzt werden (wichtig für walk-through)

Grundlegende Idee: Clustering

- betrachten eine Szene als Ansammlung von *clusters* (Mengen von Polygonen)
- Wenn eine Ebene gefunden werden kann, die eine Cluster-Teilmenge von der anderen trennt, dann:
  - Cluster auf der gleichen Seite dieser Ebene wie der Betrachterstandpunkt können nicht von Clustern auf der anderen Seite verdeckt werden (aber umgekehrt)
- Jede Teilmenge kann weiter unterteilt werden (wenn eine Ebene gefunden werden kann).
- → Binärbaum, der die Unterteilung repräsentiert
  - innere Knoten: Ebenen
  - Blätter: Regionen im Raum

Konstruktion des BSP-Baumes (Prinzip):

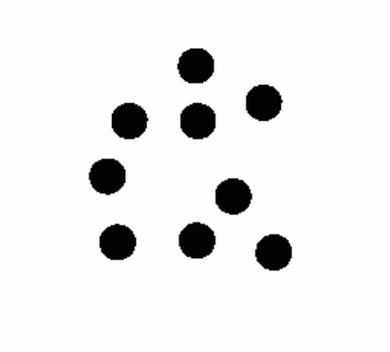
- wähle ein Polygon, Unterteilung erfolgt entlang der Ebene dieses Polygons
- alle Polygone danach unterteilen, ob sie im positiven oder negativen Halbraum dieser Ebene liegen
- wird ein Polygon von der Ebene geschnitten: Zerlegen in Teilpolygone
- Rekursion in den negativen Halbraum
- Rekursion in den positiven Halbraum

Konstruktion des BSP-Baumes (Algorithmus):

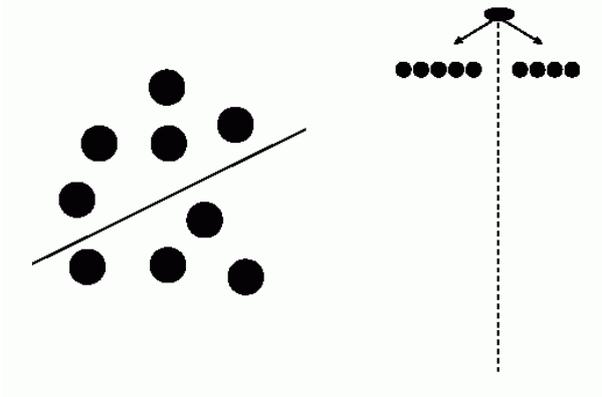
```
if Polygonliste leer
  then BSPTree := NULL
  else begin
    SelectPolygon(Polygonliste, Wurzel)
    backlist := NULL
    frontlist := NULL
    foreach Polygon p in der Polygonliste do
      if p liegt vor Wurzel
        then AddToBSPList(p, frontlist)
        else if p liegt hinter Wurzel
          then AddToBSPList(p, backlist)
          else
            SplitPolygon(p, Wurzel, pFront, pBack)
            AddToBSPList(pFront, frontlist)
            AddToBSPList(pBack, backlist)
          fi
        fi
      fi
      Combine(frontlist, Wurzel, backlist, BSPTree)
    od
  end
fi
```

Beispiel:

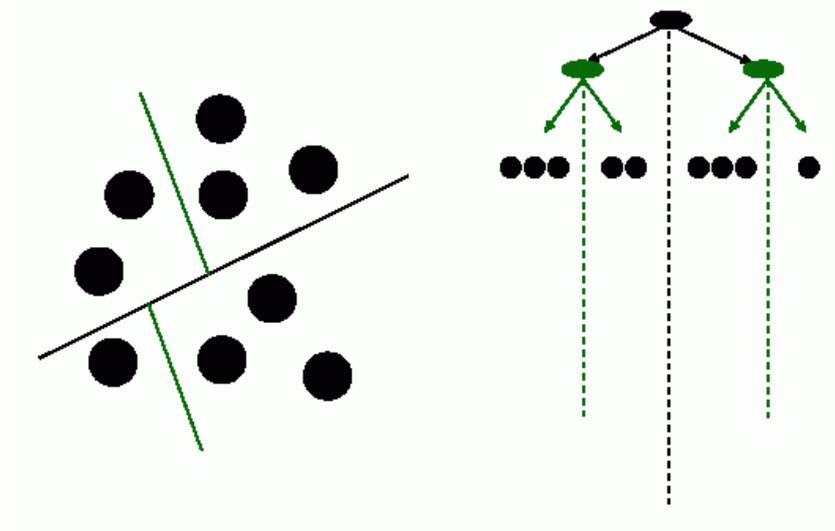
Polygone der Szene



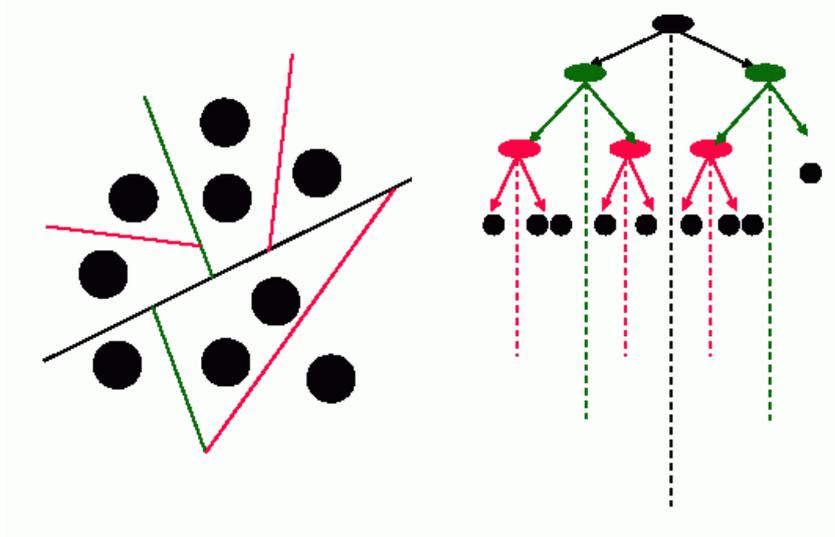
erste Unterteilung



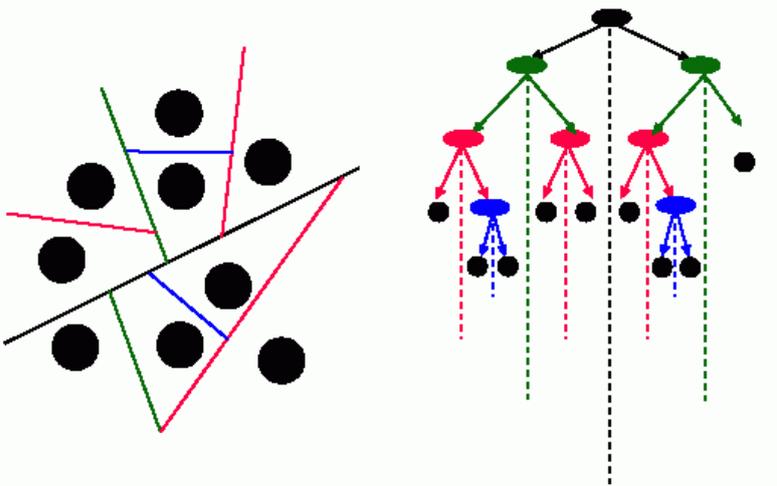
zweite Unterteilung:



dritte Unterteilung:



vierte (letzte) Unterteilung:



Der BSP-Baum ist unabhängig vom Betrachterstandpunkt.

Display des BSP-Baumes:

- ebenfalls rekursiv
- es muss jeweils die räumliche Beziehung des Betrachterstandpunktes zur Wurzel des Baumes bekannt sein (Prüfung, ob vor oder hinter – vgl. back face culling)
- Erzeugen der sichtbaren Flächen: Traversierung des Baumes (rekursiv) in "in-order"
- zeichne die sichtbaren Polygone von hinten nach vorne

## Algorithmus "DisplayBSPTree":

```
if BSPTree nicht leer
  then if Betrachterstandpunkt liegt vor dem Wurzel-Polygon
    then
      DisplayBSPTree(backBranch)
      DisplayBSPTree(rootPolygon)
      DisplayBSPTree(frontBranch)
    else
      DisplayBSPTree(frontBranch)
      DisplayBSPTree(rootPolygon)
      DisplayBSPTree(backBranch)
  fi
fi
```

## Zusammenfassung BSP-Bäume

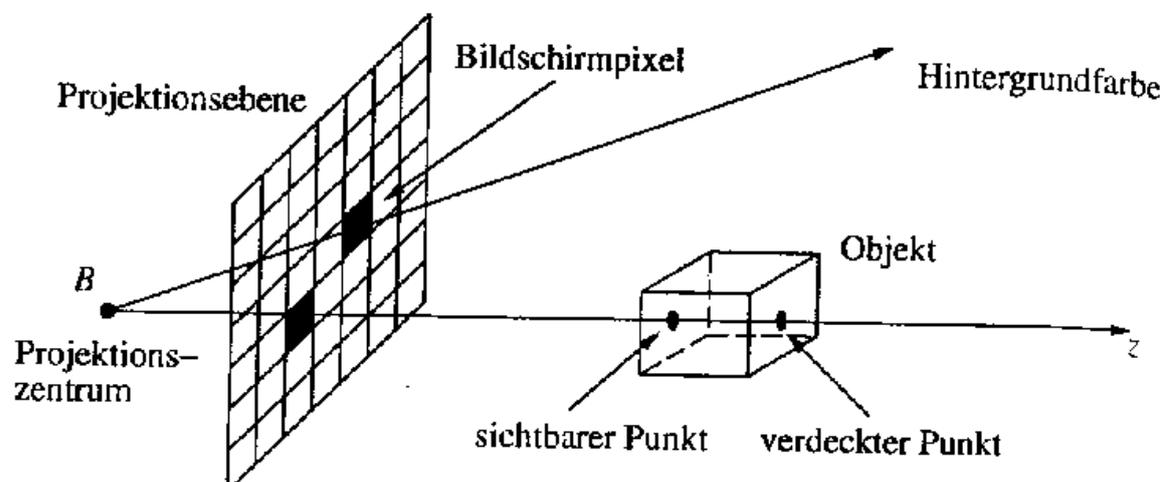
- Vorteile:
  - einfaches, elegantes Schema
  - benutzt nur den Framebuffer (kein zusätzlicher  $z$ -Buffer)
- Nachteile:
  - aufwendige Preprocessing-Phase schränkt Anwendung auf statische Szenen ein
  - worst-case-Komplexität zum Aufbau des Baumes:  $O(n^3)$
  - Zerteilen der Polygone erhöht deren Anzahl (auch hier  $O(n^3)$  im schlechtesten Fall)

### 3. Das Ray-Casting-Verfahren

- Hybrid Objektraum-Bildraum (aber mehr im Objektraum)
- Prinzip der Strahlverfolgung (vgl. Raytracing in der Beleuchtungsrechnung, siehe später)
- vom Betrachterstandpunkt wird je ein Strahl durch jedes Pixel in die 3D-Szene verfolgt.
- Berechnung der Schnittpunkte der Strahlen mit den Objektoberflächen

Fallunterscheidung dabei:

1. es gibt keinen Schnittpunkt: verwende die Hintergrundfarbe.
2. es existieren Schnittpunkte: bestimme den zum Betrachter nächstgelegenen, dieser ist sichtbar, die anderen verdeckt.



Nachteil:

hoher Rechenaufwand durch häufige Schnittpunktberechnungen (ca. 95 % der Gesamtrechenzeit bei typischen Szenen)

## Vorteile:

- leicht parallelisierbar (bei  $p$  Prozessoren teile Bildschirm in  $p$  disjunkte Rechtecke – allerdings muss bei diesem Ansatz die 3D-Szene auf jedem Prozessor vollständig zur Verfügung stehen)
- Effizienzverbesserung durch *boundary volume*-Techniken möglich (hierarchische Konzepte, Clustering)
- Kombination mit klassischem Raytracing (globale Beleuchtungsrechnung; siehe später) möglich