

10. Visible Surface Determination

auch: *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 (Kameramodell).

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.

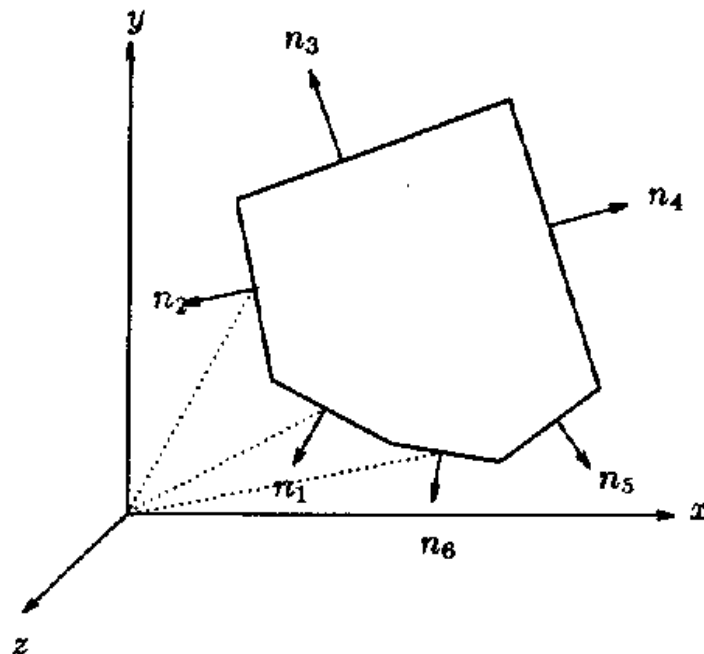
Zwei Möglichkeiten der Betrachtung:

- 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

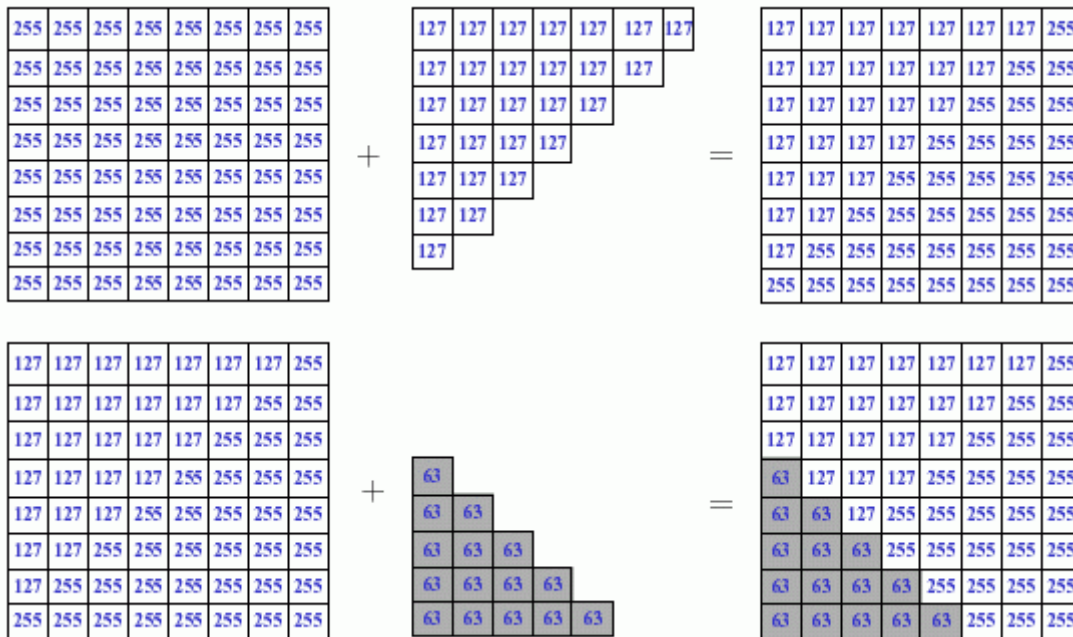
- schon 1974 vorgeschlagen, aber damals nicht realisiert
- heute dominierender Algorithmus für Rastergeräte

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**

$p_z = z$ -Wert des Polygons an Position (x, y)

if $p_z < z$ -Buffer(x, y) **then**

 WritePixel(x, y, c)

 WriteZ(x, y, p_z)

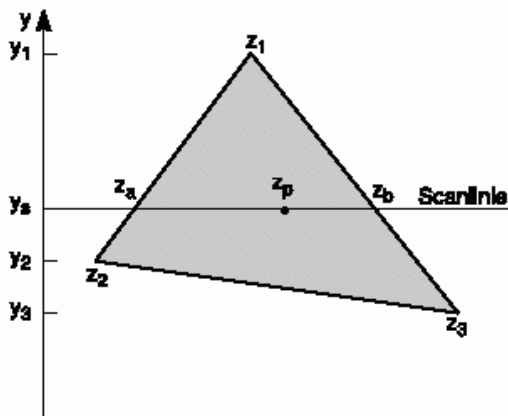
fi

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.

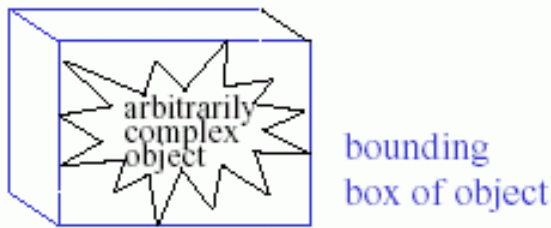
- Polygon zuerst gegen die höchste Ebene testen
- Wenn das Polygon weiter entfernt ist als die Tiefe, die im entspr. (Makro-) Pixel gespeichert ist: verdeckt
- wenn näher: Test gegen die nächstniedrige Ebene, usw.
- Wenn Polygon auf der tiefsten Ebene sichtbar ist, dann zeichnen und z-Pyramide aktualisieren.

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):

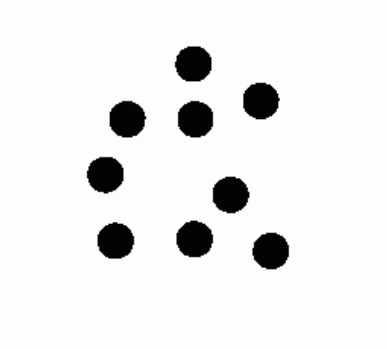
- wähle ein Polygon, Unterteilung erfolgt entlang der Ebene dieses Polygons
- alle Polygone danach unterteilen, ob sie im positiven oder negativen Halbraum dieser Ebene liegen
- wird ein Polygon von der Ebene geschnitten: Zerlegen in Teilpolygone
- Rekursion in den negativen Halbraum
- Rekursion in den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

Konstruktion des BSP-Baumes (Algorithmus):

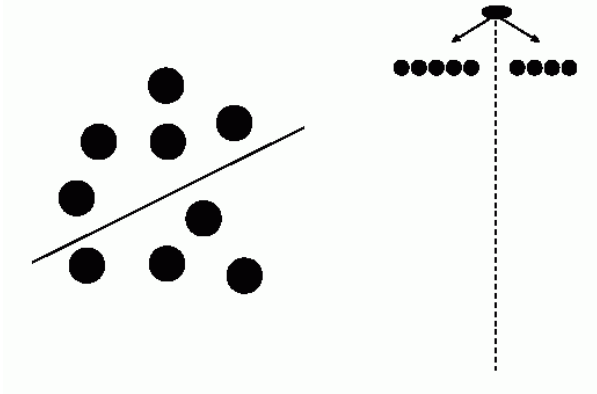
```
if Polygonliste leer
  then BSPTree := NULL
  else begin
    SelectPolygon(Polygonliste, Wurzel)
    backlist := NULL
    frontlist := NULL
    foreach Polygon p in der Polygonliste do
      if p liegt vor Wurzel
        then AddToBSPList(p, frontlist)
        else if p liegt hinter Wurzel
     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

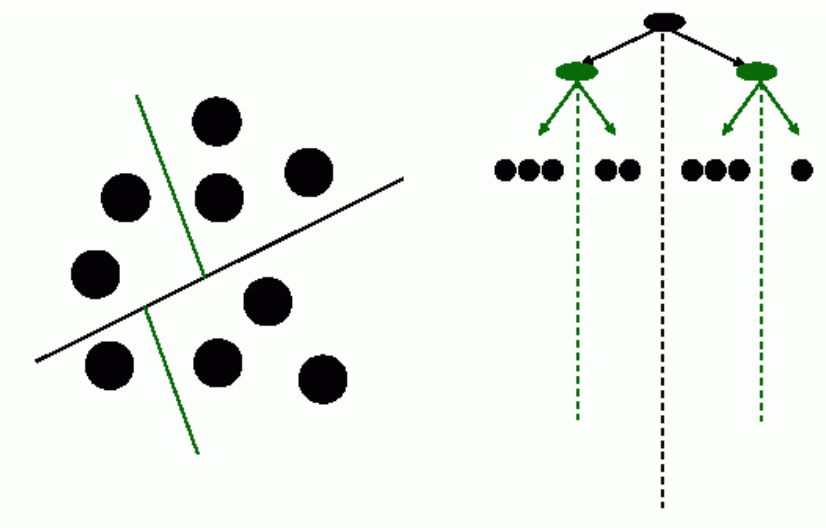
Polygone der Szene



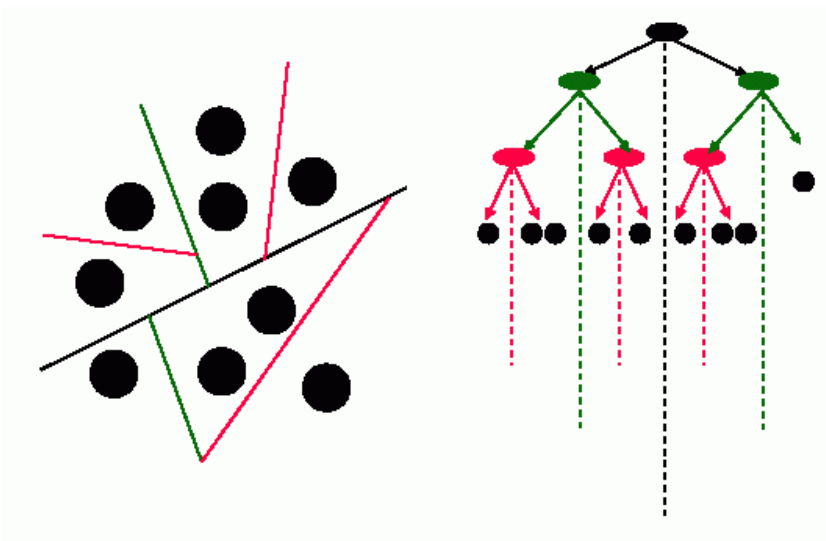
erste Unterteilung



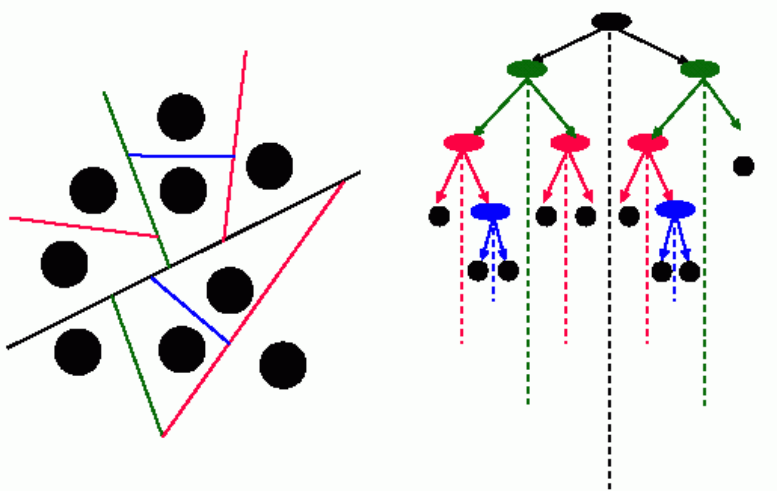
zweite Unterteilung:



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

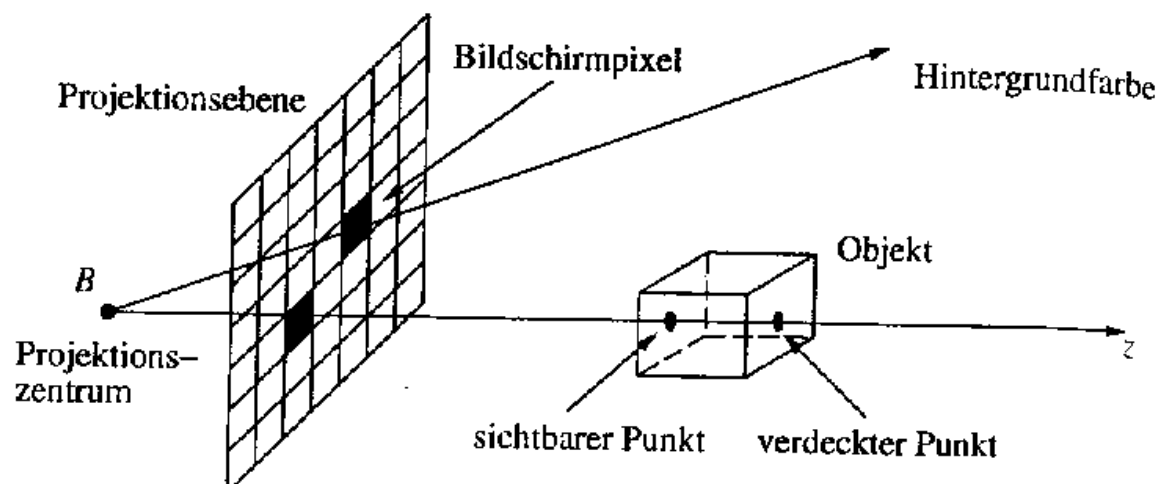
- Vorteile:
 - einfaches, elegantes Schema
 - benutzt nur den Framebuffer (kein zusätzlicher z -Buffer)
- Nachteile:
 - aufwendige Preprocessing-Phase schränkt Anwendung auf statische Szenen ein
 - worst-case-Komplexität zum Aufbau des Baumes: $O(n^3)$
 - Zerteilen der Polygone erhöht deren Anzahl (auch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