Visibilitätsrechnung

"Hidden Surface Removal" (HSR)
"Visible Surface Determination" (VSD)

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

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

Im Folgenden:
- Problemdefinition und Komplexitätsbetrachtung
- Konservative Sichtbarkeitstests für die Vorverarbeitung
- Bildraumalgorithmen
- Objektraumalgorithmen
- hybride Algorithmen

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.

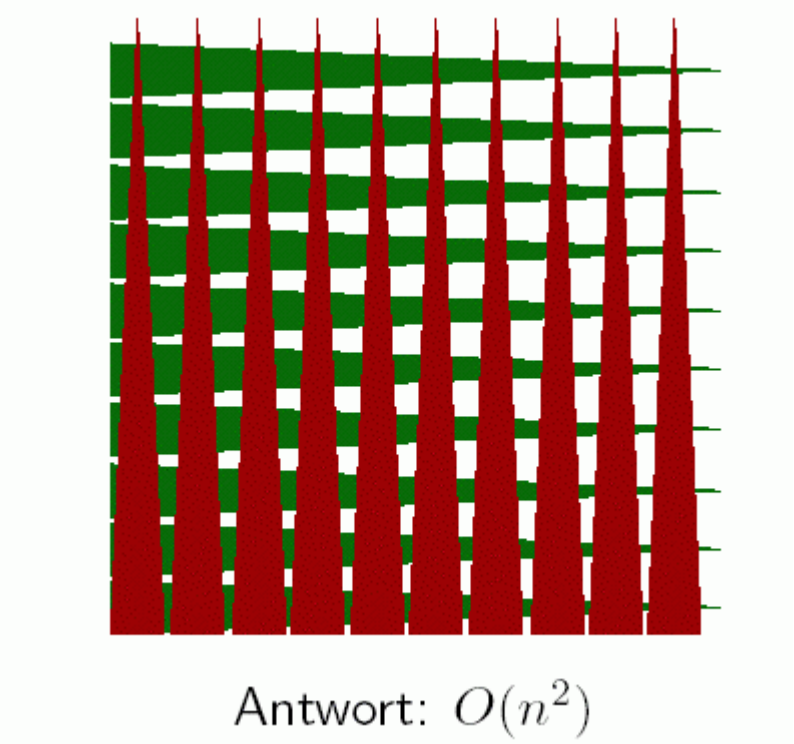
Drei Klassen von Algorithmen:
1. Bildraumalgorithmen ($\rightarrow$ *image precision*)
- Sichtbarkeit wird für diskrete Bildpunkte bestimmt
- Beispiel: *z*-Buffer
2. Objektraumalgorithmen ($\rightarrow$ *object precision*)
- exakte Sichtbarkeitsbestimmung im Modellraum
- Beispiele: Clipping von Polygonen 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.

Unterschiede zwischen den Algorithmen:

- Welche Art geometrischer Modelle werden verarbeitet?
- Wieviele Objekte in der Szene müssen untersucht werden?
- Ist ein Preprocessing notwendig?
- Ist der Algorithmus auch für Animationen geeignet?
- Komplexität (Speicher und Zeit)

Zur Komplexität:
theoretische Schranke
Was ist die minimale "worst-case"-Komplexität für die analytische Berechnung aller sichtbaren Teilflächen durch Clipping von Polygonen an Polygonen (Szene mit *n* Polygonen)?



genauer je nach Typ des Algorithmus:
bei Objektraumalgorithmen:

- Vergleich jedes Objekts mit jedem anderen
- somit theoretisch $O(n^2)$ (n = Anzahl Objekte)
- praktisch besser (der obige "worst case" tritt selten ein)

bei Bildraumalgorithmen:

- Vergleich jedes Objekts mit jedem Pixel
- somit theor. $O(nN)$ (N = Anzahl Pixel)
- praktisch besser als Objektraumalgorithmen wegen Ausnutzung von Kohärenzen

Kohärenzen in einer Szene:

Bestimmte Charakteristiken einer Szene sind lokal relativ konstant. Beispiele:

- *Spannenkohärenz*: Scanlinien enthalten Spannen konstanter Intensität (Nutzung z.B. auch bei der Bildkompression).
- *Scanlinienkohärenz:* ähnliche "Muster" von Scanlinie zu Scanlinie.
- *Kantenkohärenz*: Kanten ändern ihre Sichtbarkeit nur, wenn sie eine sichtbare Kante kreuzen oder eine Fläche durchstoßen.
- *Tiefenkohärenz:* Benachbarte Teile einer Fläche haben gewöhnlich eine ähnliche Tiefe.
- *Framekohärenz:* Bei Animationen ändern sich die Szenen wenig von Frame zu Frame.

Ausnutzung solcher Kohärenzen beim Hidden Surface Removal erhöht die Effizienz.

Konservative Sichtbarkeitstests

"konservativ": vorsichtig, d.h. Sichtbarkeit wird auch noch angenommen in Fällen, wo in Wirklichkeit (Teil-) Verdeckung vorliegt

- nur triviale Fälle, also komplett sichtbar oder nicht
- z.T. auch dazugerechnet: einfache Clipping-Verfahren
- Zweck: Verringerung der Anzahl der zu rendernden Polygone
- Einsatz als Preprocessing für andere Algorithmen

Techniken:

- Clipping am Sichtkörper (bereits behandelt)
- *Back-face culling* (Rückseitenentfernung)
- Portal-Rendering (bes. bei Gebäuden, Dungeons...)
- Occlusion Culling, betrachtet große "Blocker"-Polygone

Back-face culling

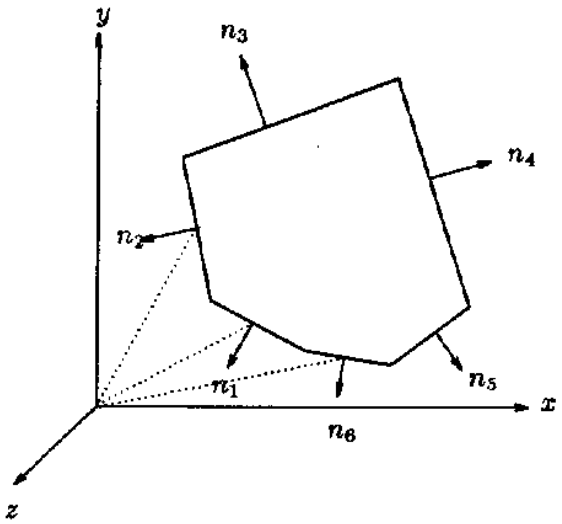
Entfernen von nicht sichtbaren Rückseiten, d.h. von allen Polygonen, die vom Betrachter weg zeigen
(diese Rückseiten machen etwa die Hälfte aller Flächen aus!)

setzt voraus:
- Objekte sind als Polygonnetze modelliert, der Betrachter-standpunkt 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°), dann handelt es sich bei dem Polygon um eine zu entfernende Rückseite.

Rechnerisch: Skalarprodukt prüfen
LOS · Normale > 0 $\Rightarrow$ unsichtbar
LOS · Normale $\leq$ 0 $\Rightarrow$ möglicherweise sichtbar

Halbraum-Interpretation:
Normalenform der Ebenengleichung für die Polygon-Ebene:
(Normale · x) + D = 0
Die Ebene teilt den übrigen Raum in zwei Halbräume:
(Normale · x) + D > 0: x liegt im positiven Halbraum
(Normale · x) + D < 0: x liegt im negativen Halbraum

Polygon zeigt vom Betrachter weg, wenn sich der Betrachterstandpunkt (COP) im negativen Halbraum befindet:
unsichtbar, wenn (Normale · COP) + D < 0

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

Portal-Rendering

eingesetzt in Spielen und Architekturvisualisierung (*walkthroughs*)

Grundidee:
- in Architekturmodellen typischerweise hohe Tiefenkomplexität und häufige Verdeckungen $\Rightarrow$ Anzahl der sichtbaren Objekte eher klein
- man bestimme eine Menge potentiell sichtbarer Objekte, nur diese werden gerendert
- Unterteilung der Szene in *Zellen* und *Portale*

Zelle: Polyeder im Raum
Portal: transparente 2D-Region an einer Zellgrenze, die aneinandergrenzende Zellen verbindet

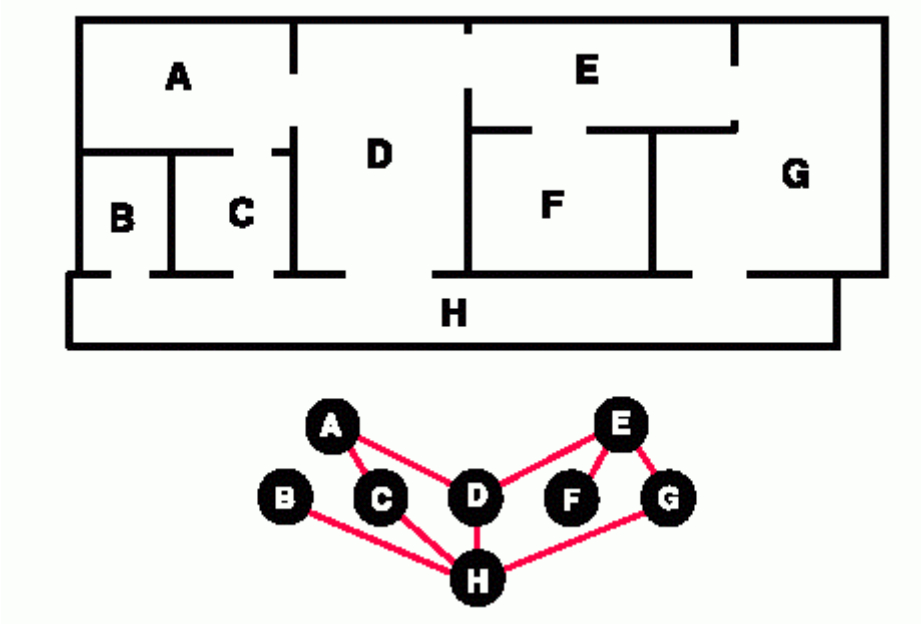
Zellen können andere Zellen nur durch Portale "sehen".

Architekturmodelle:

- Zellen = Räume, Grenzen der Zellen = Wände
- Portale: Fenster, Türen, auch Spiegel!

Prinzip: Zerlegung des Raumes in Zellen; vor dem Rendern feststellen, welche Zellen für den Betrachter aus seiner aktuellen Position sichtbar sind.

Beispiel (aus Schlechtweg 2001):



Der Betrachterstandpunkt liege in Zelle E.
Die Menge potentiell sichtbarer Objekte enthält dann:

- alle Objekte in Zelle E
- alle Objekte in Zellen, die mit E durch ein Portal verbunden sind (D, F, G)
- alle Objekte in Zellen, die wiederum mit diesen verbunden sind?

Nein! Nur die Objekte in denjenigen Zellen, deren Portale durch die Portale der Zelle E direkt sichtbar sind.

- Bestimmung benachbarter (durch Portale verbundener) Zellen durch Adjazenzgraphen
- Reduktion der Anzahl der beim Rendering zu betrachtenden Polygone durch Berechnung der Sichtkegel durch die Portale

(hier nur die Grundidee gezeigt, der eigentliche Algorithmus ist etwas schwieriger)

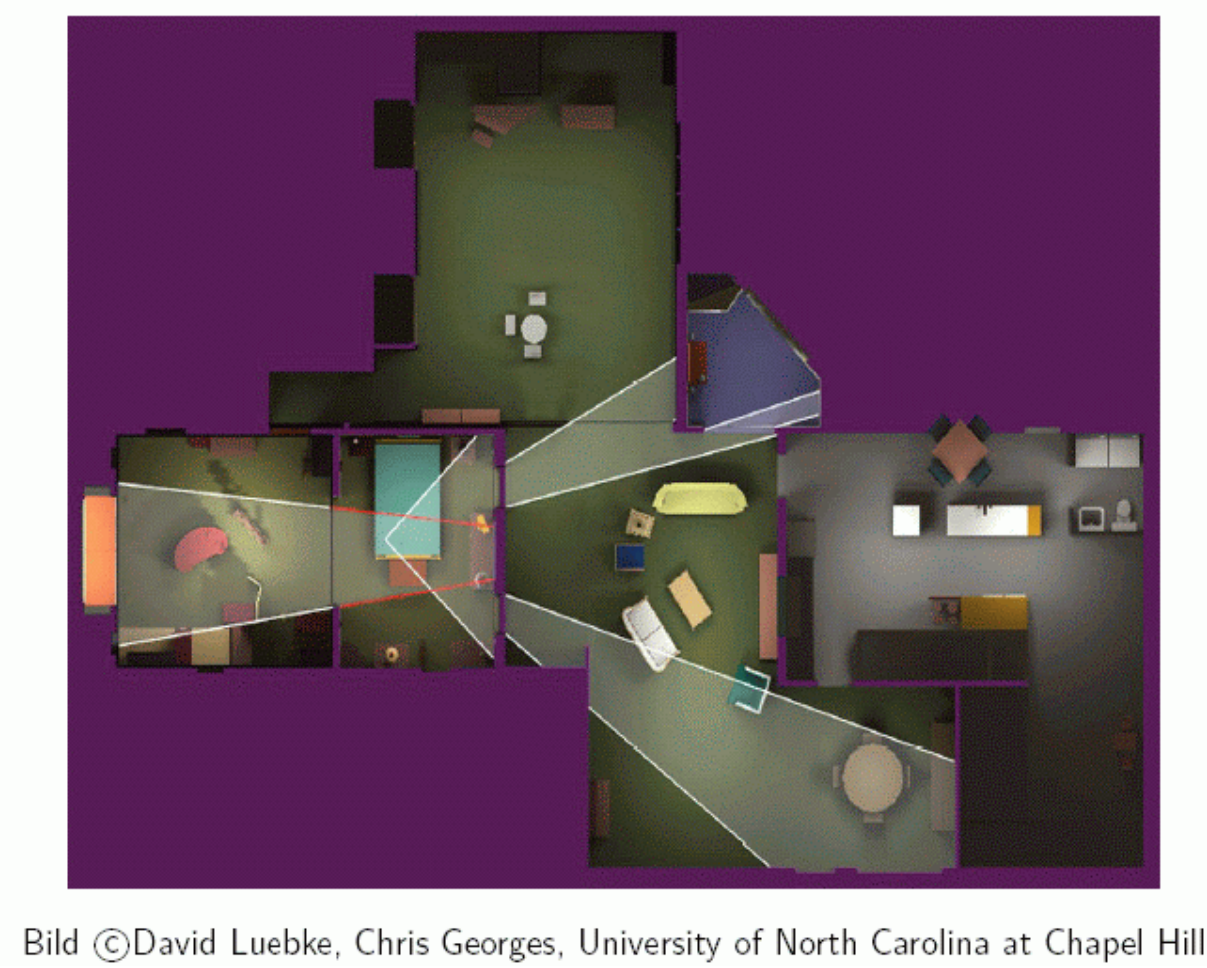


Bild ©David Luebke, Chris Georges, University of North Carolina at Chapel Hill

Algorithmen zur Visible Surface Determination

Z-Buffer-Algorithmus

häufig in der Hardware implementiert
z-Buffer (*z*-Puffer, Tiefenpuffer): pixelbezogener Speicher für die *z*-Werte

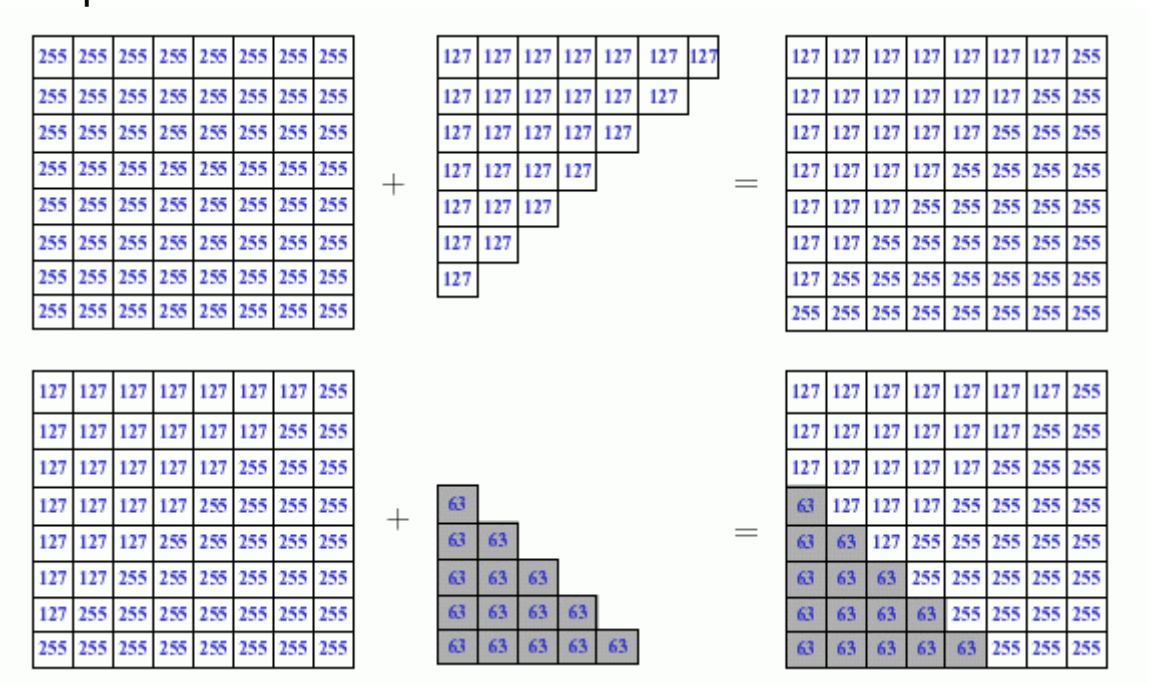
- schon 1974 vorgeschlagen, aber damals nicht realisiert
- heute dominierender Algorithmus für Rastergeräte

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:



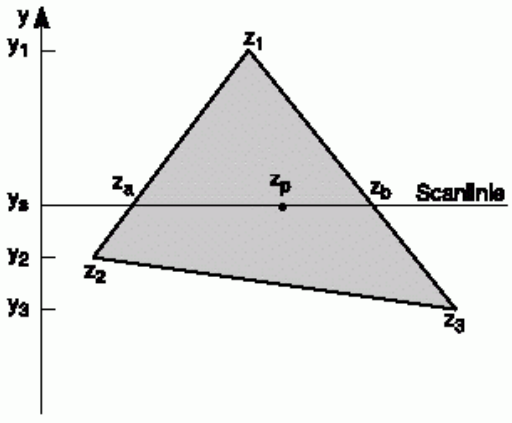
(hier z-Werte nicht zwischen 0 und 1, sondern zwischen 0 und 255; implementationsabhängig)

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.

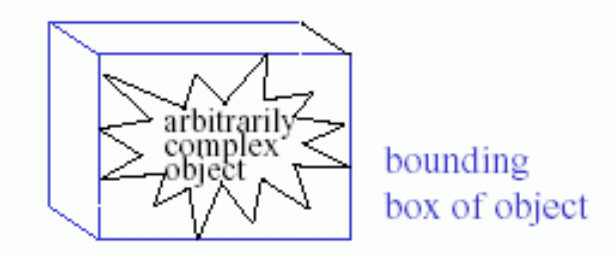
- Polygon zuerst gegen die höchste Ebene testen
- Wenn das Polygon weiter entfernt ist als die Tiefe, die im entspr. (Makro-) Pixel gespeichert ist: verdeckt
- wenn näher: Test gegen die nächstniedrige Ebene, usw.
- Wenn Polygon auf der tiefsten Ebene sichtbar ist, dann zeichnen und *z*-Pyramide aktualisieren.

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.)

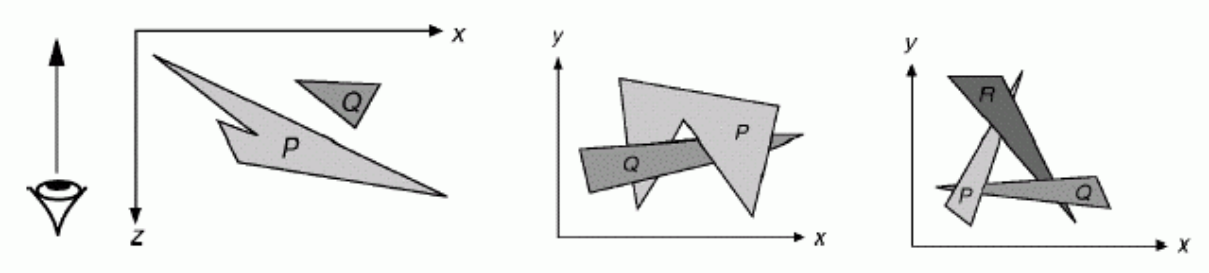
Listenprioritäts-Algorithmen

Idee: "Sichtbarkeits-Reihenfolge" (Priorität) aller Objekte wird im Objektraum bestimmt

- werden Objekte in dieser Reihenfolge gezeichnet → korrektes Bild
- Spezialfall "2 1/2 D-Darstellungen": Alle Objekte haben konstante *z*-Koordinaten oder nehmen disjunkte *z*-Intervalle ein

→ dann eindeutige Sortierung nach der *z*-Koordinate möglich

Probleme, wenn das nicht der Fall ist, z.B.:



- Nach welchem *z*-Wert ist zu sortieren?
- was passiert mit zyklischen Überlappungen?

Tiefensortierung

- Zeichne Objekte in Reihenfolge ihrer Entfernung vom Betrachterstandpunkt
- Schrittfolge:
  1. Sortiere alle Polygone nach ihrer kleinsten (am weitesten entfernten) z-Koordinate
  2. Löse alle Mehrdeutigkeiten (bei Überlappngen) auf, teile Polygone – wenn nötig – auf
  3. Rastere die Polygone in aufsteigender Reihenfolge der z-Koordinate (d. h. von hinten nach vorne)
- auch genannt: *Painter's Algorithm*

(Maler-Algorithmus) – weil Ölmaler so Polygone malen würden: die entferntesten zuerst.

Auflösen von Mehrdeutigkeiten (wenn sich die Bereiche der *z*-Koordinaten von Polygonen überlappen):

Sei P nach der kleinsten *z*-Koordinate weiter entfernt als Q.
Führe folgende Tests durch:
1. Überlappen sich die Bereiche der *x*-Koordinaten nicht?
2. Überlappen sich die Bereiche der *y*-Koordinaten nicht?
3. Liegt P komplett auf der gegenüberliegenden Seite der Ebene von Q (vom Betrachterstandpunkt aus gesehen)?
4. Liegt Q komplett auf der gleichen Seite der Ebene von P wie der Betrachterstandpunkt?
5. Überlappen sich die Projektionen der Polygone auf die *xy*-Ebene nicht?
$\rightarrow$ wenn einer dieser Tests positiv: P verdeckt Q nicht.

- komplizierte Tests
- viele Sonderfälle
- mögliche zyklische Überlappungen $\rightarrow$ Endlosschleifen
- bei Überlappungen: Polygon in mehrere Teilpolygone aufteilen

BSP-Bäume (Binary Space Partition)

Verfahren 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 walkthrough)

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).
- $\rightarrow$ Binärbaum, der die Unterteilung repräsentiert
  - innere Knoten: Ebenen
  - Blätter: Regionen im Raum

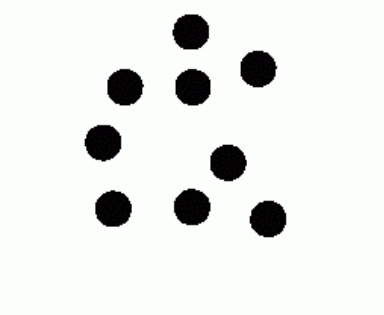
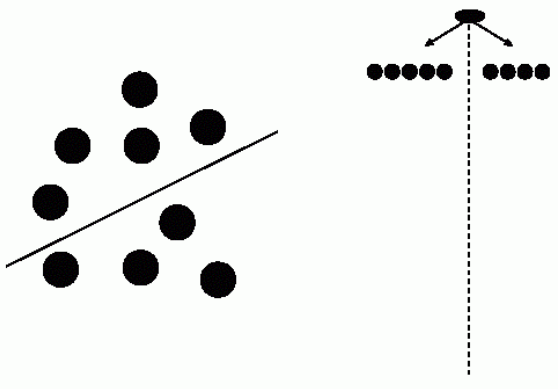
Konstruktion des BSP-Baumes (Prinzip):

- 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

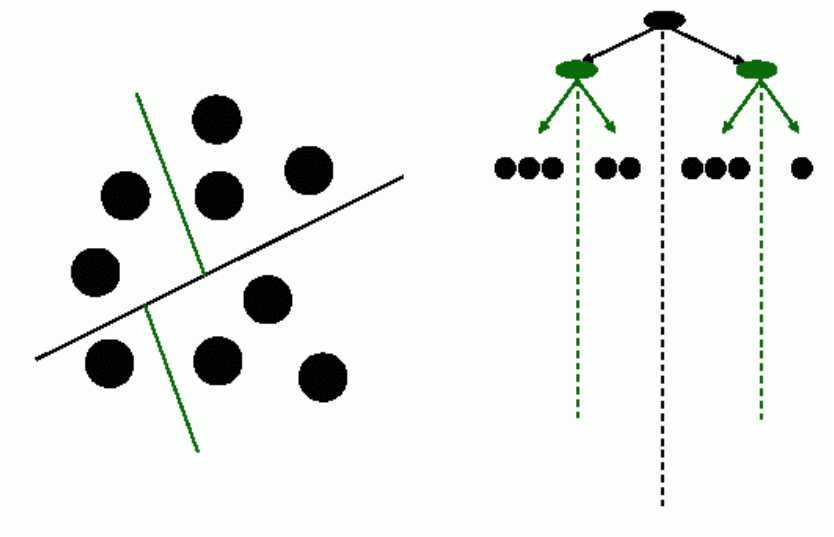
Konstruktion des BSP-Baumes (Algorithmus):

```
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
            Combine(frontlist, Wurzel, backlist, BSPTree)
          od
        end
fi
```

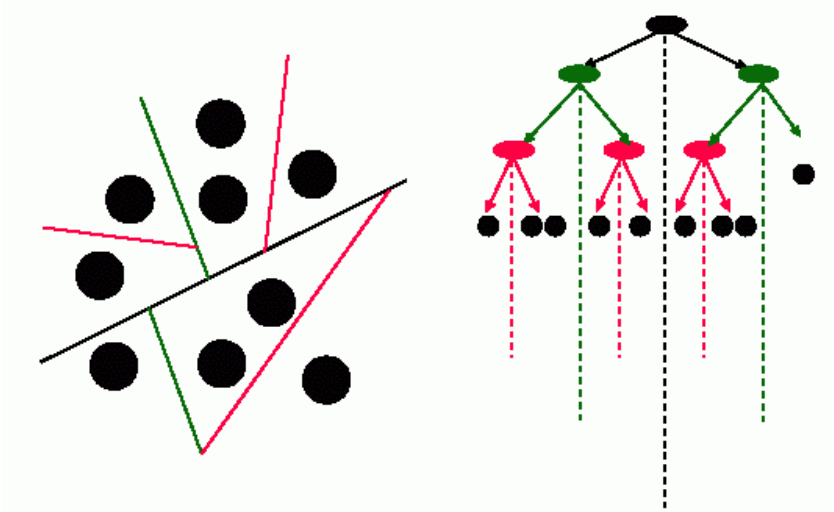
Beispiel:

Polygone der Szene	erste Unterteilung
	

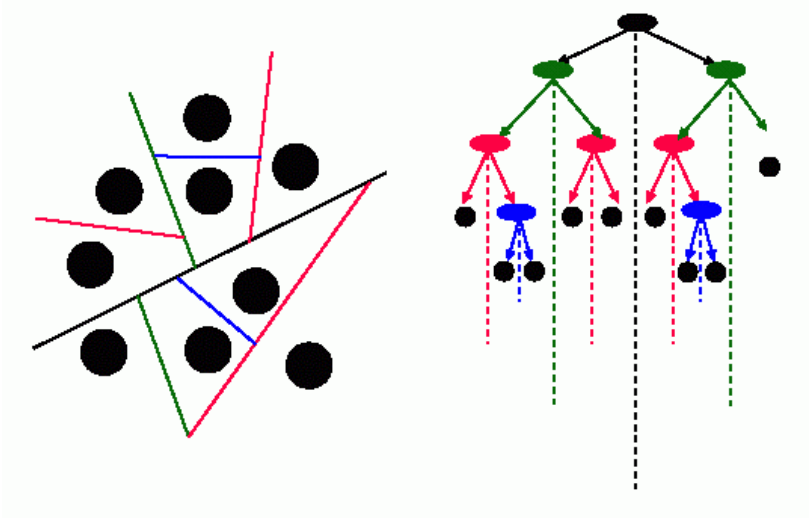
zweite Unterteilung:



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 Betrachter-standpunktes 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 (wie bei Listenprioritäts-Algor.)

Algorithmus:

```
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

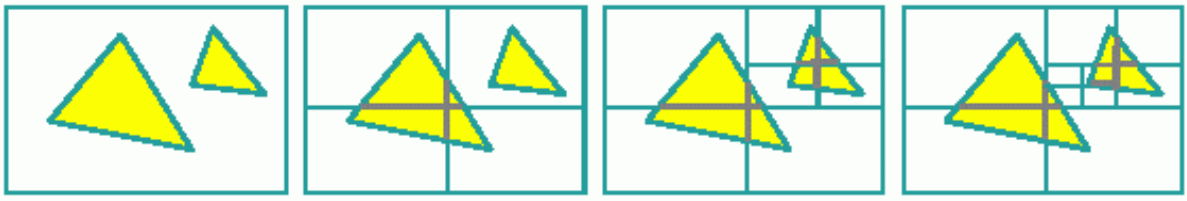
- Vorteile:
  - einfaches, elegantes Schema
  - benutzt nur den Framebuffer (kein zusätzlicher z-Buffer)
- Nachteile:
  - aufwendige Preprocessing-Phase schränkt Anwendung auf statische Szenen ein
  - worst-case-Komplexität zum Aufbau des Baumes: $O(n^3)$
  - Zerteilen der Polygone erhöht deren Anzahl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

Internet-Quelle:
ftp://ftp.sgi.com/other/bspfaq/faq/bspfaq.html

Der Warnock-Algorithmus

Flächenunterteilungsalgorithmus.
Grundidee: Divide and Conquer.
Hybrider Algorithmus Objekt-/Bildraum.
Ausnutzung räumlicher Kohärenz.

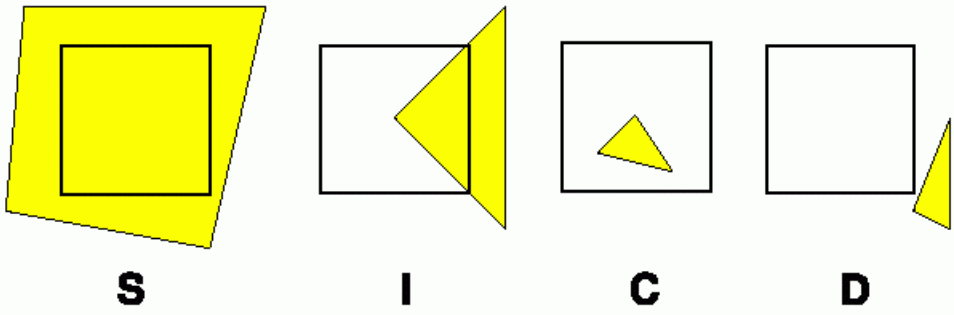
- Unterteilen des Bildes in kleine rechteckige Bereiche (Fenster)
- nur die Polygone betrachten, die in das jeweilige Fenster fallen
- wenn Sichtbarkeit nicht eindeutig: weiter unterteilen
- Unterteilung (spätestens) beendet, wenn Fenster Pixelgröße haben



Grundprinzip:
if <= 1 Polygon ragt ins Fenster
  then zeichne Polygon im Fenster
  else unterteile Fenster und Rekursion

Rekursion endet spätestens, wenn Fenster Pixelgröße hat.
Jedes Pixel wird nur einmal gesetzt.

Beziehungen zwischen betrachtetem Fenster und einem Polygon:



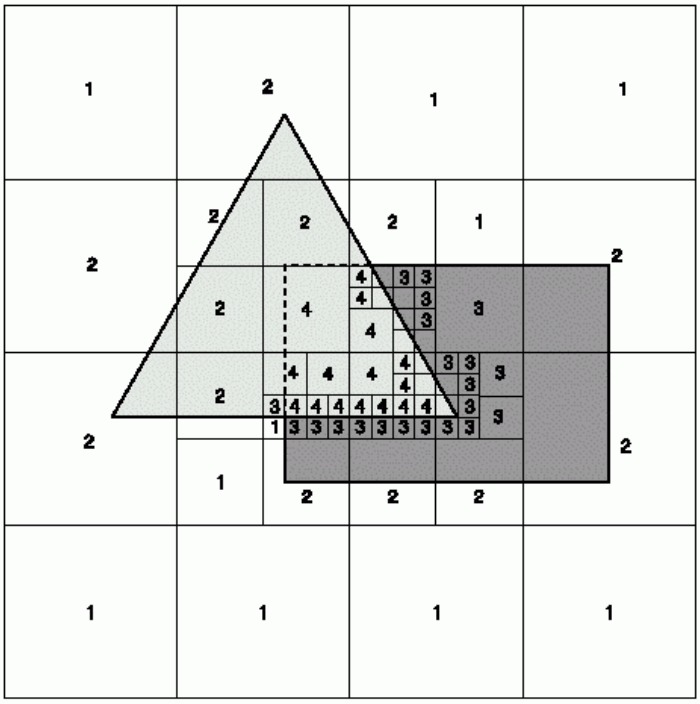
- Polygon umschließt das Fenster (**S** = surrounds)
- Polygon schneidet das Fenster (**I** = intersects)
- Polygon ist komplett innerhalb des Fensters (**C** = is contained)
- Polygon ist komplett außerhalb des Fensters (**D** = is disjoint)

Wann ist die Sichtbarkeit klar?

- Alle Polygone sind außerhalb des Fensters.
- Nur ein Polygon schneidet das Fenster oder ist im Fenster enthalten.
- Ein umschließendes Polygon liegt *vor* allen anderen.

Der Algorithmus kann wahlweise im Bildraum oder im Objekt-raum durchgeführt werden (bei Berechnung im Objektraum sind Ergebnisse in Gerätekoordinaten zu transformieren, Äquivalenz zur Größe eines Pixels festzulegen).

Beispiel:



1. alle Polygone außerhalb
2. nur ein schneidendes oder enthaltenes Polygon
3. nur ein umschließendes Polygon
4. mehrere Polygone schneiden das Fenster, sind darin enthalten oder umschließen es, aber ein umschließendeds Polygon vor allen anderen

- Bei der Unterteilung des Fensters könen einige Relationen zwischen Polygon und Fenster „berechnet“ werden:
  - **S⇒S** für alle Unterteilungen
  - **D⇒D** für alle Unterteilungen
  - sonst: Bestimmung möglich, aber nicht so einfach
- Unterteilung muß nicht in Hälften erfolgen, auch an Polygonvertices möglich

Zusammenfassung zum Warnock-Algorithmus:

- kein Preprocessing notwendig
- Antialiasing kann eingebaut werden (Rekursion weitergehen lassen bis auf Subpixel-Ebene)
- Bei häufigen Unterteilungen und häufigen Schnittberechnungen (Test auf Fall I) ziemlich aufwendig
- relativ einfacher Algorithmus
- schwer in Hardware realisierbar

- Aufwand: $O(nN)$, wobei n = Anzahl der Polygone, N = Anzahl der Pixel
- in Fällen, wo immer bis auf 1 Pixel unterteilt werden muss, ist das Ergebnis ähnlich zum z-Buffer: Das ist für große Szenen ($n>N$) häufig der Fall. Der Overhead des Unterteilens lohnt sich dann nicht mehr.