

- §1 Hardwaregrundlagen
- §2 Transformationen und Projektionen
- §3 Repräsentation und Modellierung von Objekten
- §4 Rasterung
- **§5 Visibilität und Verdeckung**
 - **5.1 Visibilität**
 - 5.2 Verdeckung

§6 Rendering
§7 Abbildungsverfahren
(Texturen, etc.)
§8 Freiformmodellierung
Anhang: Graphiksprachen und
Graphikstandards
Anhang: Einführung in OpenGL

Weitere Themen: Netze, Fraktale,
Animation, ...

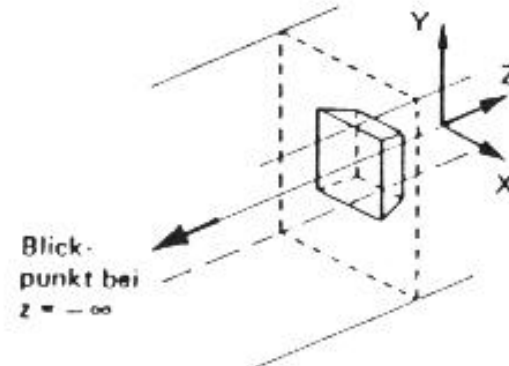
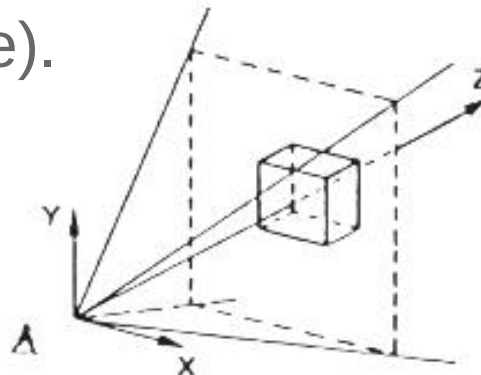
- Ziel ist (möglichst) exakte Bestimmung der von einem **gegebenen Blickpunkt** aus sichtbaren bzw. unsichtbaren Teile der darzustellenden Szene.
- Wünschenswert ist **hohe Interaktionsrate**, so dass Eingaben des Benutzers sich direkt auf die Darstellung auswirken.
- In den meisten Fällen ist **Echtzeitausgabe** der Szene nötig.
- Einteilung der Verfahren:
 - **Objektraumverfahren**
prinzipiell geräteunabhängig,
Rechengenauigkeit ist die Maschinengenauigkeit.
 - **Bildraumverfahren**
geräteabhängig,
Rechengenauigkeit ist die Auflösung des Ausgabegerätes.

Visibilität

- Bestimmung der **sichtbaren Pixel**
 - Welches Pixel liegt **vor dem anderen**?
- **Entfernung verdeckter** Kanten und Flächen
 - Nicht zum Sparen, sondern für **korrekte** Darstellung
 - Wireframe-Darstellung:
Nur Kantendarstellung: **Hidden Line Removal** (HLR)
 - Flächendarstellung:
Solide Fläche: **Hidden Surface Removal** (HSR)
- **Transparenzen** müssen ggf. berücksichtigt werden

Verdeckung

- Dreidimensionale Szene wird auf Bildebene projiziert: **unterschiedliche Objektteile** werden auf **dieselbe Stelle** abgebildet.
- Sichtbar sind diejenigen Objektpunkte, die dem Auge des Betrachters **am nächsten** gelegen sind.
- Nicht nur die (x, y)-Koordinate in der Bildebene wichtig, sondern auch die **Tiefenrelation der Szene** (z-Koordinate).



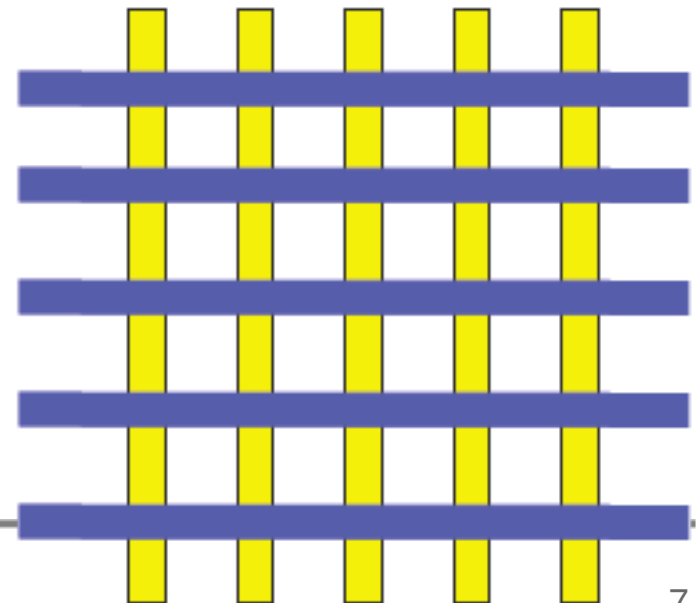
Visibilität vs. Verdeckung

- Visibilität für **Korrektheit**
- Verdeckung für **Beschleunigung** (nicht sichtbare Anteile werden weggelassen - Culling)
 - nutzt Heuristiken
 - In der Regel keine exakte Lösung

Kohärenzen

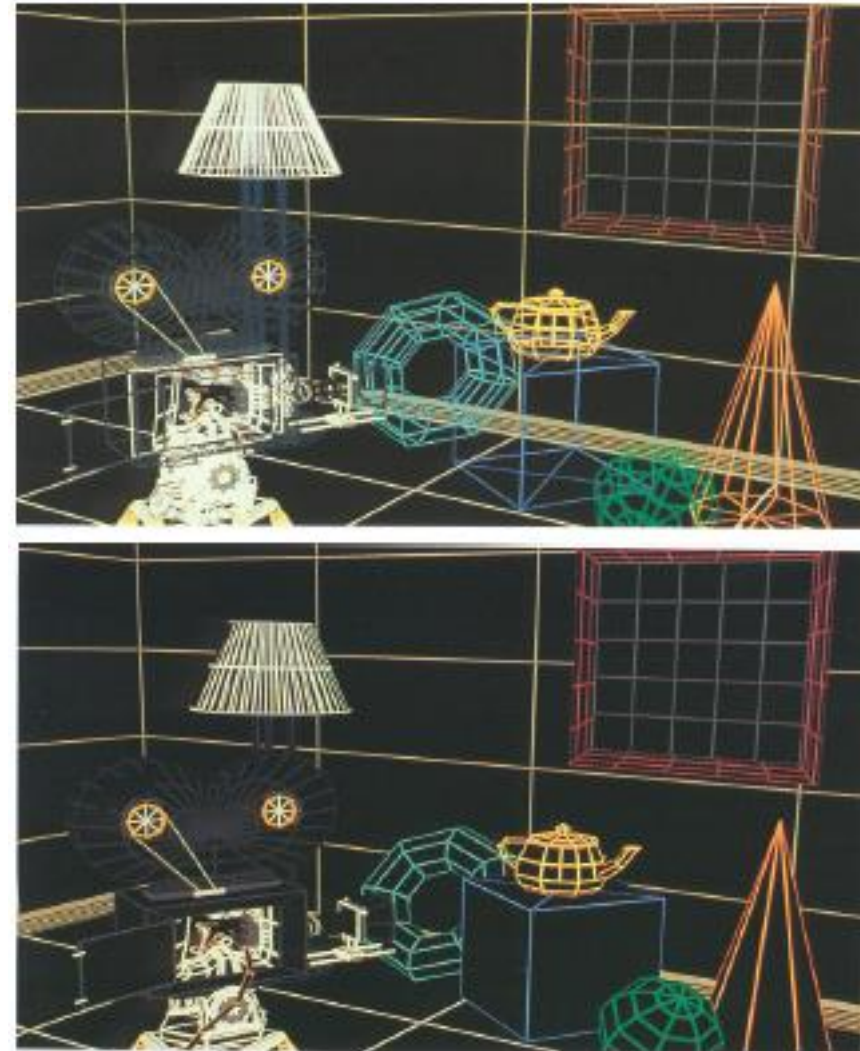
- Ausnutzung lokaler **Ähnlichkeiten**
- **Objekt**kohärenz: schneiden sich Objekte nicht, so müssen auch nicht ihre Flächen miteinander getestet werden.
- **Flächen**kohärenz: Eigenschaften benachbarter Punkte auf einer Fläche ändern sich oft nur unwesentlich.
- **Tiefen**kohärenz: Die Tiefe $z(x,y)$ auf einer Fläche kann oft inkrementell berechnet werden
- **Zeit-/Frame**kohärenz: Oft ändern sich nur wenige Anteile eines Frames

- Polygone werden auf die **Bildebene projiziert**.
- Polygone werden **in Pixel zerlegt** und die Pixel im Framebuffer abgelegt.
- Letztes gerastertes Polygon besetzt **Pixelpositionen im Framebuffer**.
- Aber: welches Polygon **müsste gesehen** werden?
 - Das dem **Beobachter am nächsten** Gelegene!
- **Klare Aussage** nicht immer gegeben.
- Visibilität = **Sortierproblem**



Wireframe-Darstellung

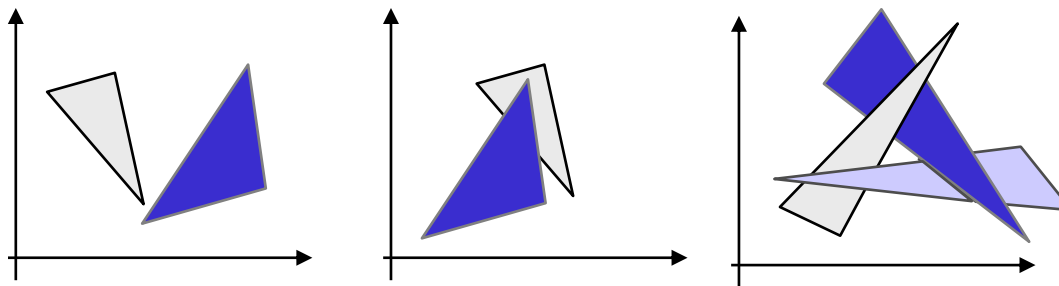
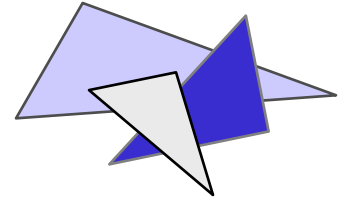
- Kanten der Flächen werden dargestellt.
- Eigentlich verdeckte Kanten **scheinen durch**.
- **Hidden-Line**-Removal:
Entfernen verdeckter Kanten



[Foley et al, 1992]

Painter's Algorithmus

- Male Polygone von **hinten nach vorne**.
- Erfordert **Tiefensortierung!**
- Wenn Tiefenwerte (z-Wert) der Polygone sich überlappen, **müssen Polygone geschnitten** werden (n^2 mögliche Teile).
- Beginne mit (Teil-)Polygon mit **größtem z-Wert**.



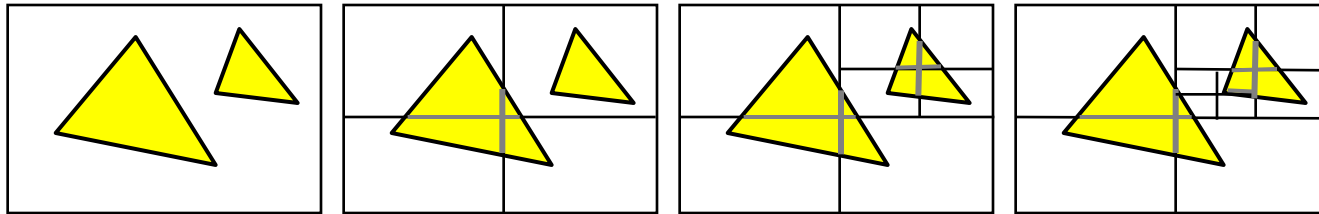
- Komplexität: $O(n^2)$, n ist Anzahl der Dreiecke

Bekannte Verfahren

- Erste Lösung des Hidden-Line-Problems: Roberts, 1963; Objektraumverfahren für konvexe Objekte
- Appels Algorithmus (1967):
Berechnet sichtbare Kanten/Konturen (NPR)
- Area Subdivision (divide-and conquer): Warnock, 1969
Ausnutzung von Flächenkohärenz durch Quadrees
- Sample Spans: Watkins, 1970
Ausnutzung von Rasterzeilenkohärenz
- Depth List: Newell et al., 1972
Prioritätslistenalgorithmus im Objektraum
- Weiler-Atherton-Algorithmus (1977):
Sortiert Polygone näherungsweise in der Tiefe

Divide and Conquer [Warnock 1969]

- **Komplexe Fälle** werden auf **einfache Fälle** zurückgeführt
 - **Nächste Fläche** überdeckt gesamten Rasterbereich
 - Es gibt **maximal eine** Fläche im Rasterbereich
- Ansonsten **rekursive Unterteilung** bis nur noch einfache Fälle



- Aufwand: $O(np)$, $p = \text{\#Pixel}$, $n = \text{\#Polygone}$
- Ggf. muss Unterteilung bis **auf Pixelebene** durchgeführt werden.
 - Fast immer bei $n > p$ (große Szenen)
 - Entspricht dann dem Z-Buffer, aber mit **Overhead**.

Scanline/Sample Span [Watkins 1970]

- y-Scanline γ nach Kanten untersuchen
- Kanten nach Tiefenwerten vergleichen/sortierten
- Kanten-, Polygon-, **ActiveEdge-Tabellen** (AET)
- **Update** der AET für jede neue Scanline γ

- Scanline γ in AET:

AC, AB, DE, FE

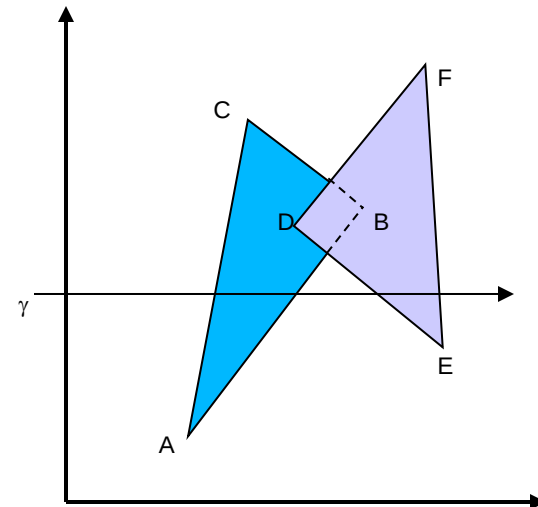
P_1 in

P_1 out

P_2 in

P_2 out

P_1, P_2 Polygone



Bekannte Verfahren

- erste Lösung des Hidden-Line-Problems: Roberts, 1963
- Appels Algorithmus (1967)
- Area Subdivision (divide-and conquer): Warnock, 1969
- Sample Spans: Watkins, 1970
- Depth List: Newell et al., 1972
- Weiler-Atherton-Algorithmus (1977)

- Diese Algorithmen haben sich im Allgemeinen **nicht durchgesetzt**, aber:

Bekannte Verfahren (Frts)

Z-Buffer-Algorithmus [Straßer, 1974, Catmull, 1974]

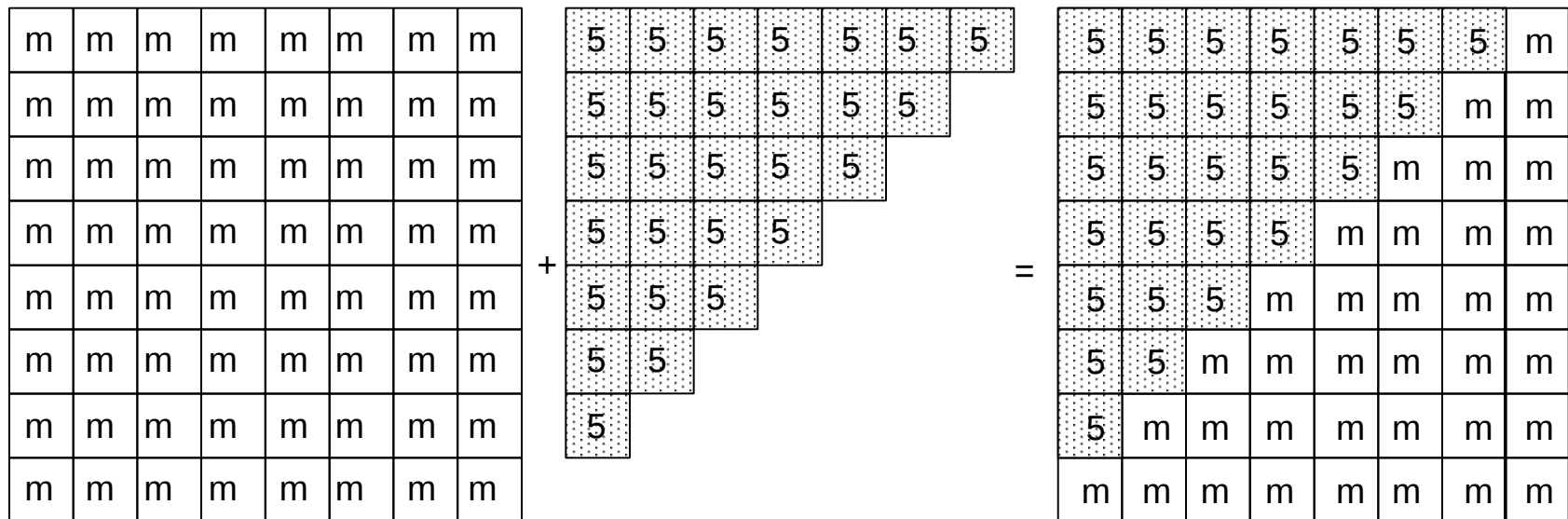
- Bestimmt Sichtbarkeit von Pixeln (**Bildraum**).
- Geeignet für die Bildausgabe auf Rastergeräten
- Einfache **Hardware**-Unterstützung
- Arbeitsweise
 - Sucht für jeden Pixel bei Rasterung nach Polygon mit **kleinstem** (am weitesten vorne liegendem) z-Wert.
 - **Zusätzlicher Speicher** (Z-Buffer/Depthbuffer):
Speichere in jedem Pixel den bisher kleinsten aufgetretenen z-Wert.

Z-Buffer-Algorithmus

- Initialisiere Framebuffer (Farbbuffer) mit **Hintergrund**farbe.
- Initialisiere Z-Buffer mit **maximalem z-Wert**.
- **Scan-Conversion** aller Polygone in beliebiger Reihenfolge
 - Berechne z-Wert $z(x,y)$ für jedes Pixel (x, y) im Polygon.
 - $z(x, y) < \text{Z-Buffer}(x, y)$,
dann zeichne Polygonfarbe in Farbbuffer bei (x, y) ein und $\text{Z-Buffer}(x, y) = z(x, y)$.
- Am Ende enthält der Farbbuffer das gewünschte Bild, der z-Buffer dessen **Tiefenverteilung**.

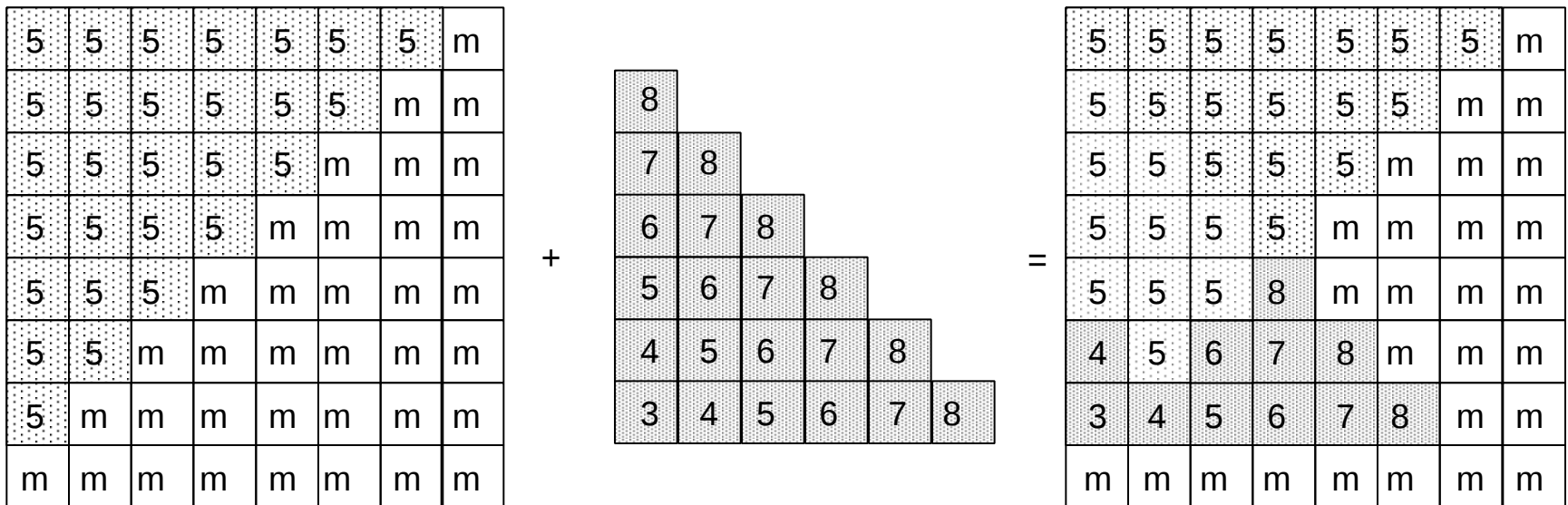
Z-Buffer-Algorithmus – Beispiel

- z-Werte codiert durch Zahlen:
kleinere Zahl => näher am Auge
- Initialisiere Z-Buffer mit max. z-Wert **m** (ganz hinten)
- Addiere ein Polygon mit **konstantem z-Wert 5**.



Z-Buffer-Algorithmus – Beispiel

- Addiere ein Polygon, das das 1. Polygon schneidet
- Artefakte bei Pixeln mit gleichem z-Wert beider Polygone



Z-Buffer-Algorithmus – Berechnung von z bei Polygonen

Zur Berechnung von $z(x,y)$ für ebene Polygone (z.B. Dreiecke) entlang einer Scan-Line:

$$\text{Ebene } Ax+By+Cz+D=0$$

$$\text{Also: } z = (-D - Ax - By) / C$$

$$z(x+dx,y) = (-D-A(x+dx) - By)/C = z(x,y) - dx * A/C$$

Nur eine Subtraktion ist notwendig, da A/C konstant ist und $dx=1$.

Z-Buffer-Algorithmus – Vorteile

- + sehr **einfache Implementierung** des Algorithmus
- + **unabhängig** von der Repräsentation der Objekte;
es muss nur möglich sein, zu jedem Punkt der Oberfläche einen z-Wert zu bestimmen
- + keine **Komplexitäts**beschränkung der Bildszene
- + keine besondere **Reihenfolge** oder Sortierung notwendig

Z-Buffer-Algorithmus – Vorteile

- **Auflösung des Z-Buffers** bestimmt Diskretisierung der Bildtiefe: z.B. 20 Bit genau 2^{20} Tiefenwerte unterscheidbar
 - problematisch sind **weit entfernte** Objekte mit kleinen Details (perspektivische Transformation)

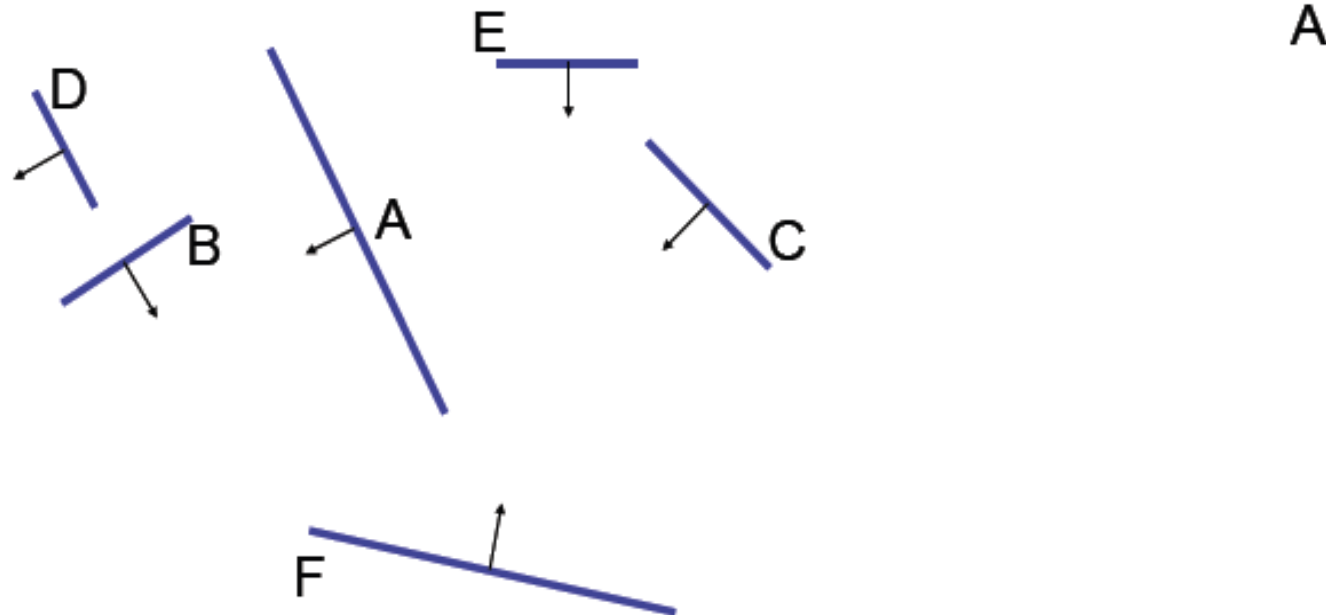
Z-Buffer-Algorithmus – Vorteile

- **Auflösung des Z-Buffers** bestimmt Diskretisierung der Bildtiefe: z.B. 20 Bit genau 2^{20} Tiefenwerte unterscheidbar
- **Transparenz** (Alpha-Buffering) und **Antialiasing** nur durch aufwändige Modifikationen möglich
- (es wird ein großer Speicher benötigt – Abhilfe durch Zerlegung in Teilbilder möglich)

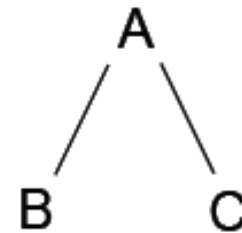
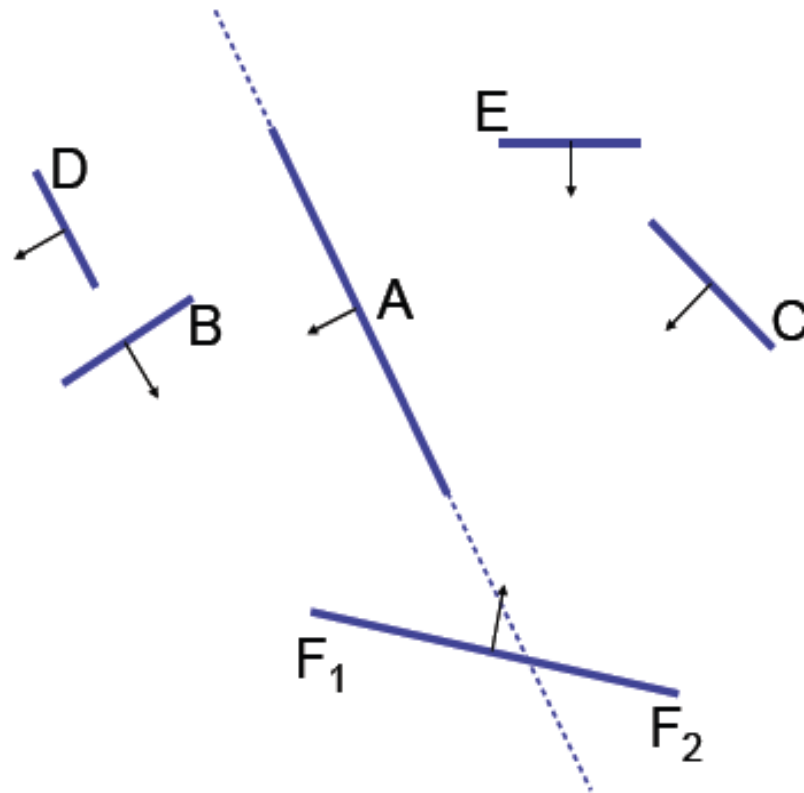
Aufteilung mit dem BSP-Tree

- Jeder Knoten entspricht **Unterteilungsebene**.
- Jeder Knoten teilt Raum in **zwei Halbräume**.
- **Ordnet Raum** bezüglich der Geraden.
- Teilt **Visibilität** an den Unterteilungsebenen auf.

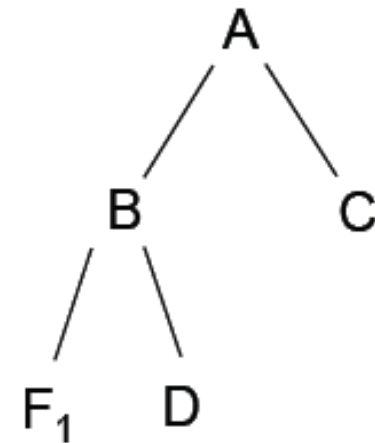
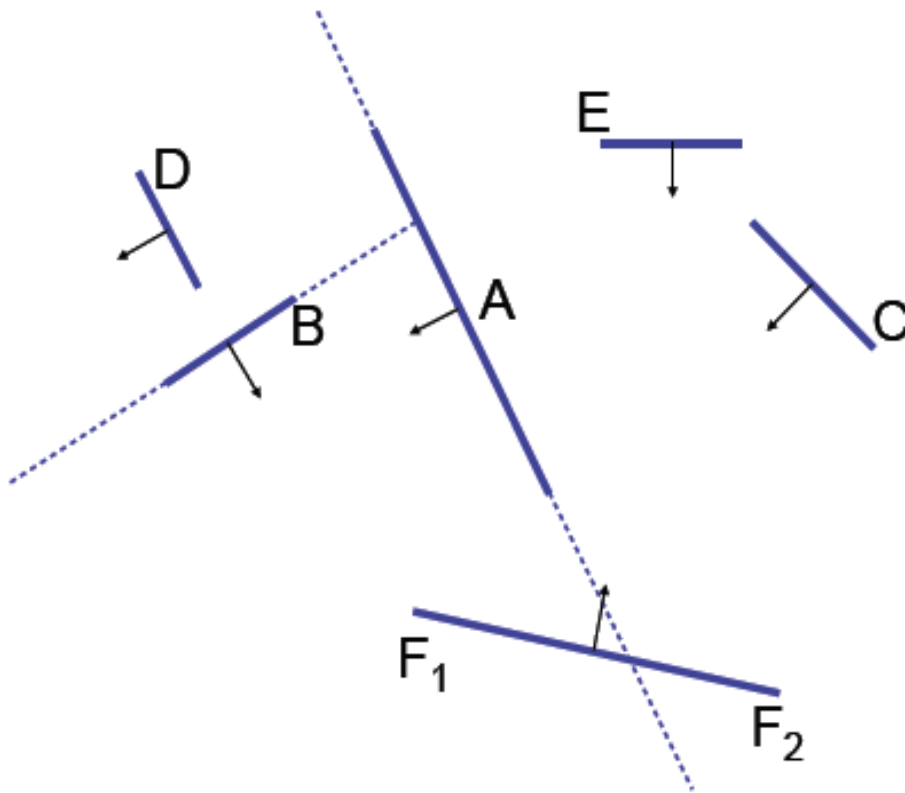
Binary-Space-Partitioning



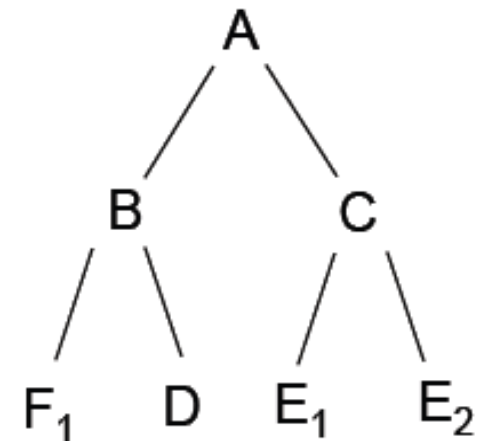
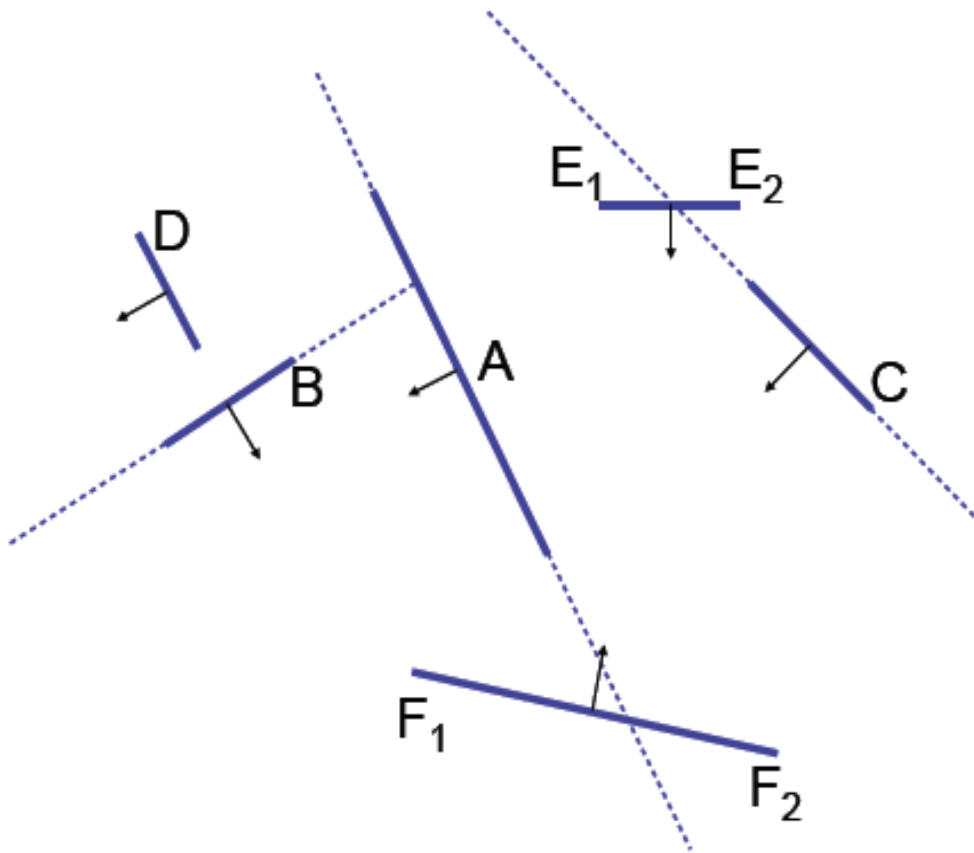
Binary-Space-Partitioning



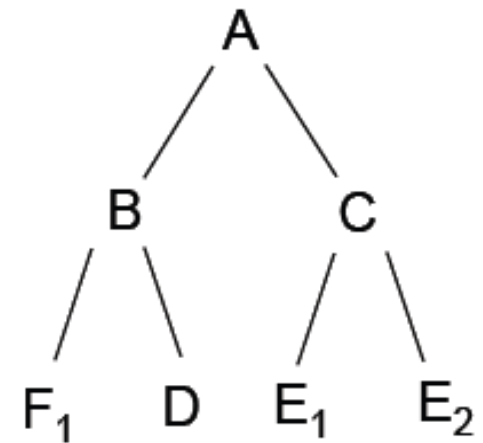
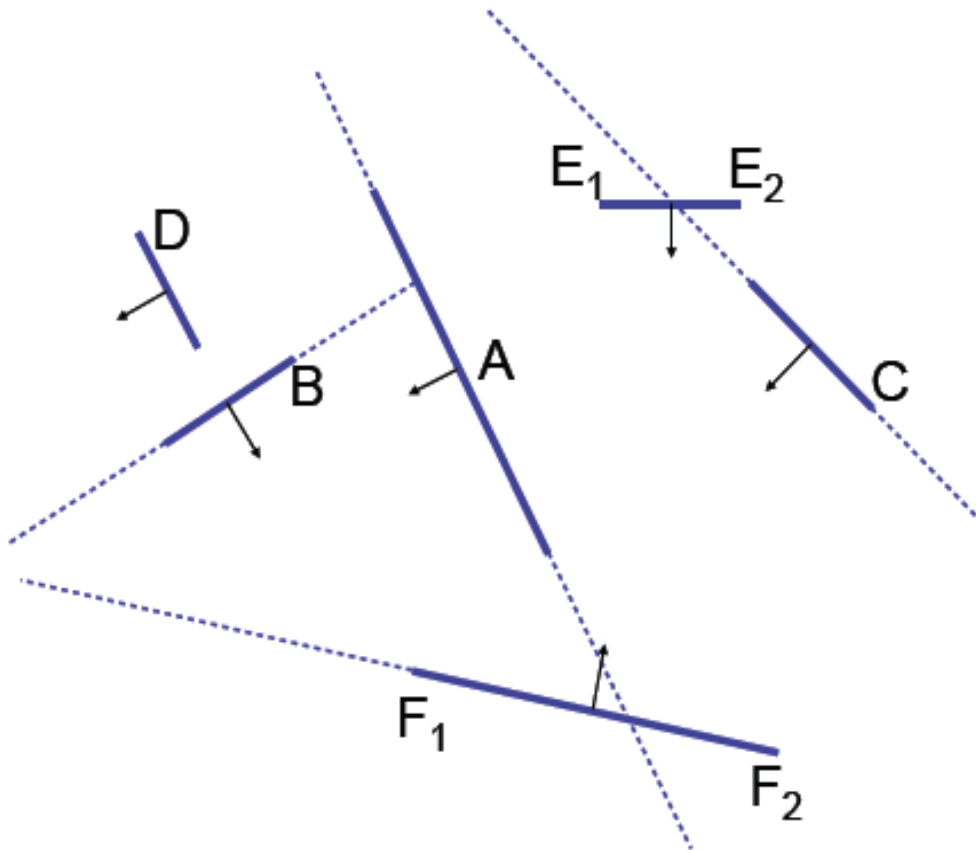
Binary-Space-Partitioning



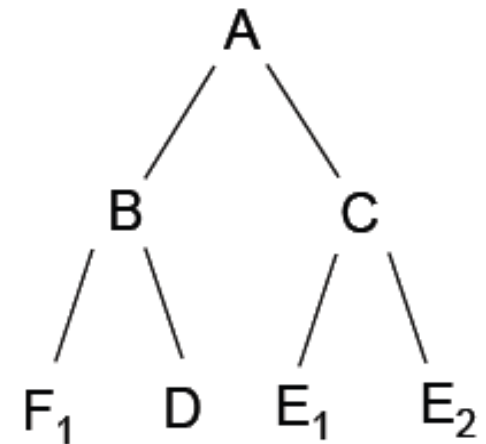
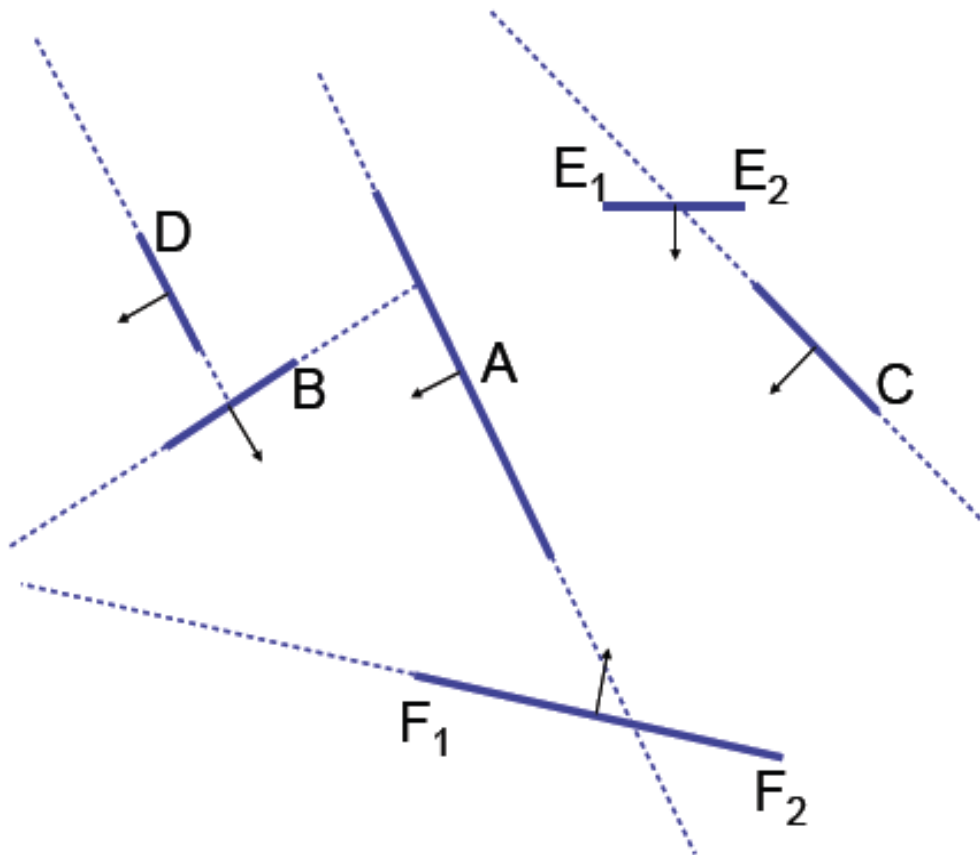
Binary-Space-Partitioning



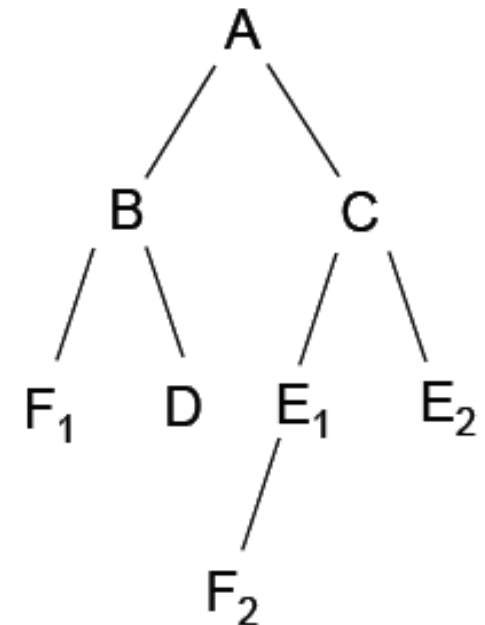
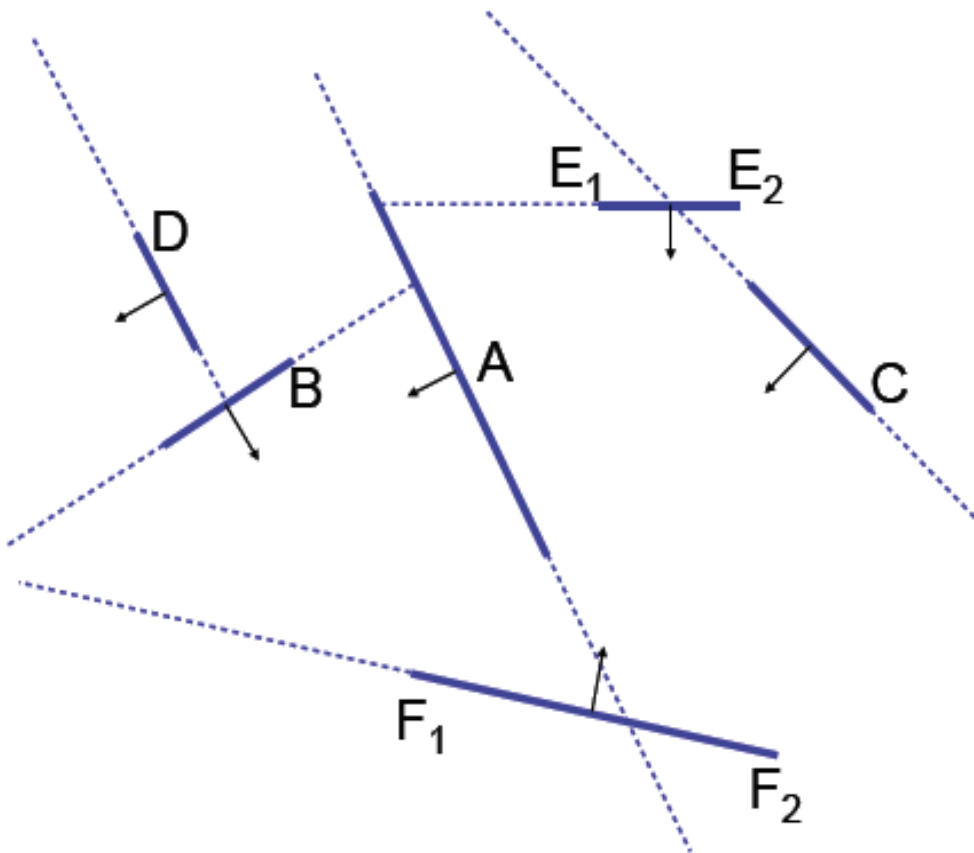
Binary-Space-Partitioning



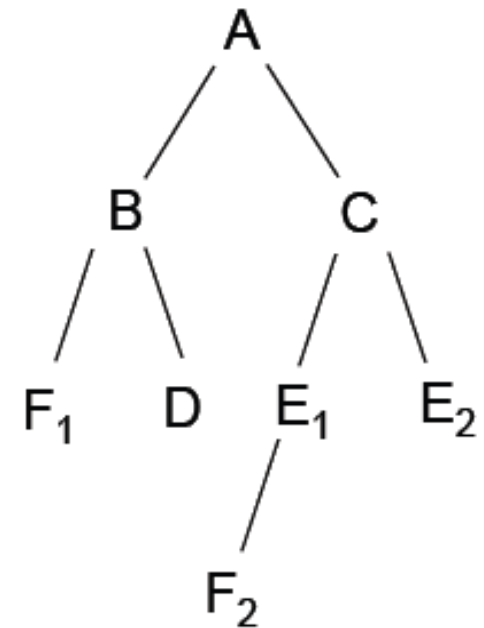
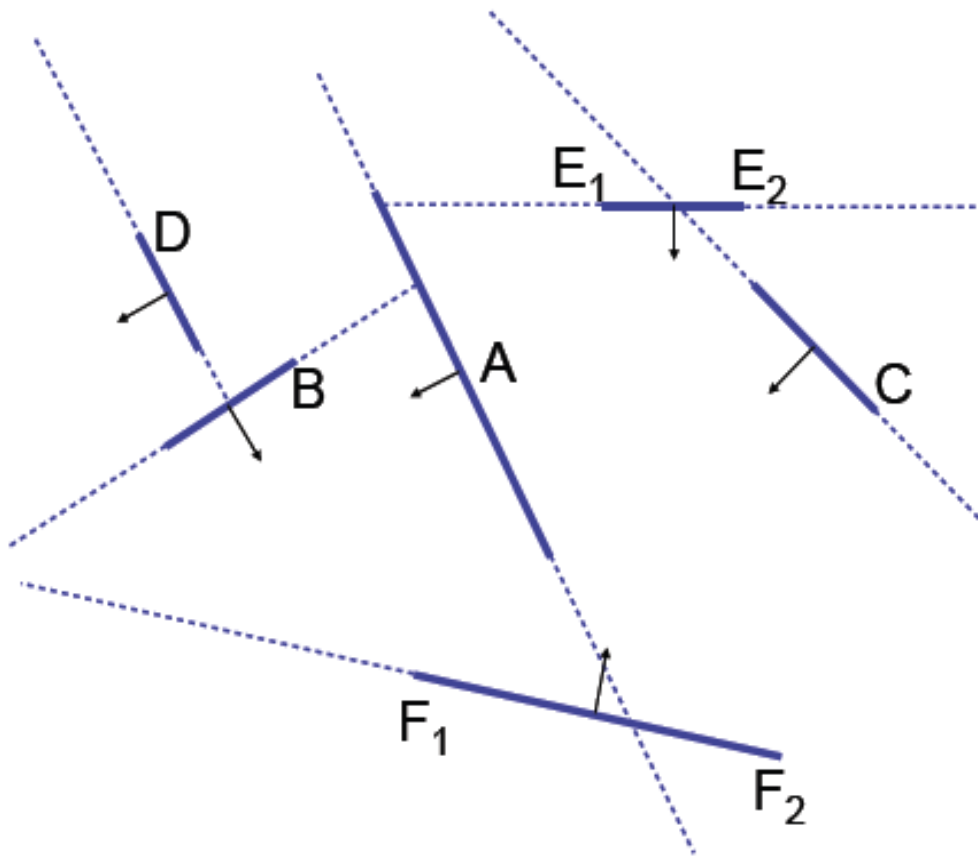
Binary-Space-Partitioning



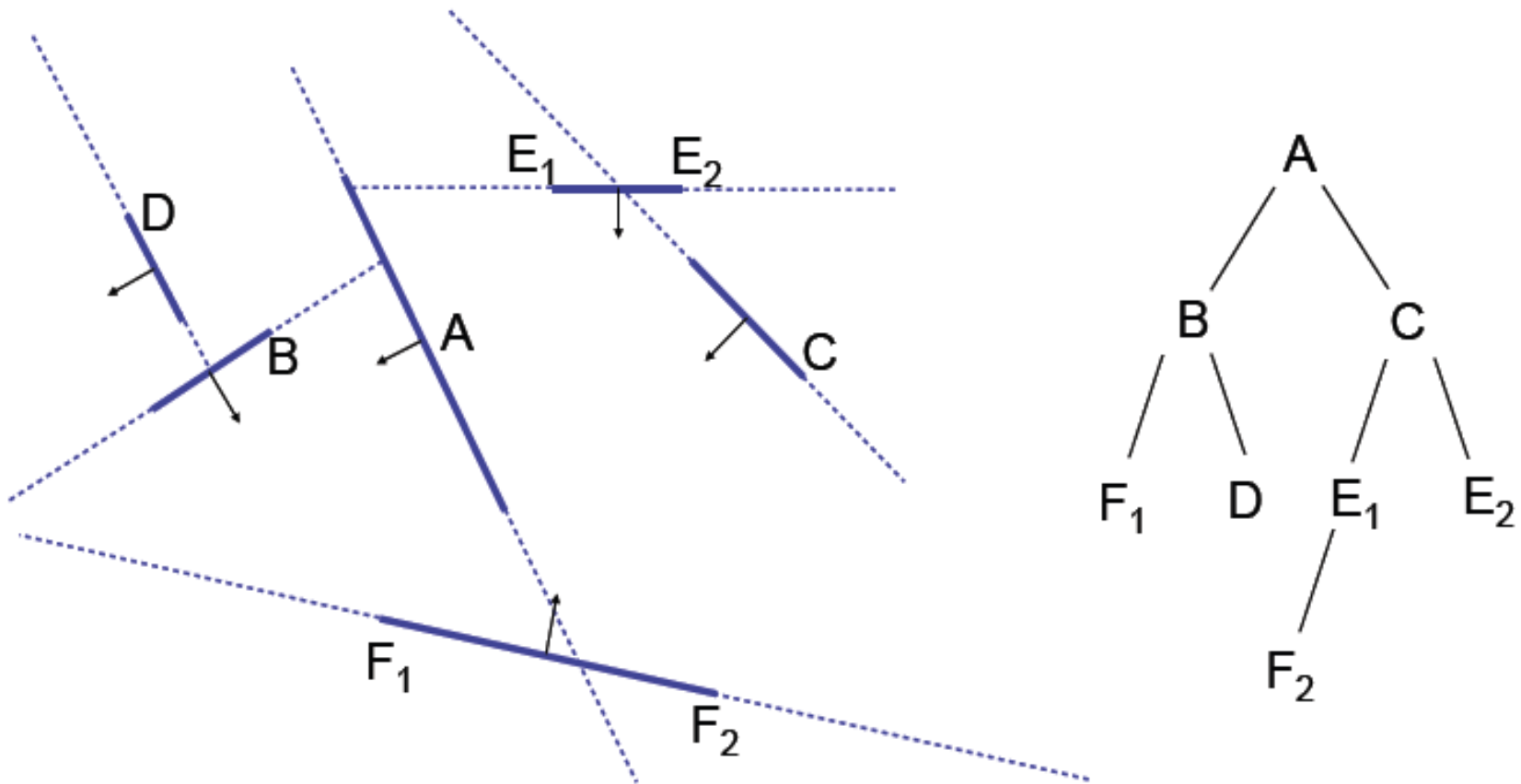
Binary-Space-Partitioning



Binary-Space-Partitioning

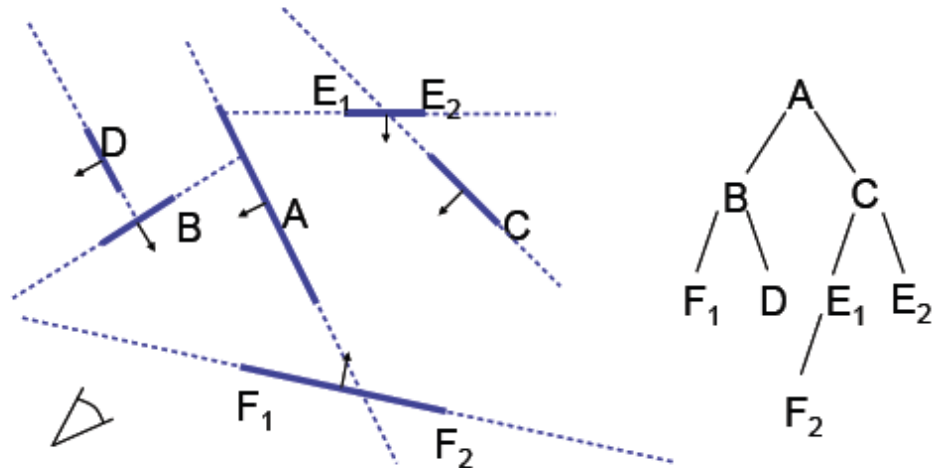


Binary-Space-Partitioning



Binary-Space-Partitioning

- **Tiefensortierte** Liste von Polygonen
- Identifiziere Halbraum H_b in dem der **Augpunkt** liegt
- Traversierungs**reihenfolge**
 - H_{1-b}
 - Knoten (Polygone)
 - H_b
- $E_2, C, E_1, F_2, A, D, B, F_1$



Painter's Algorithmus mit BSP

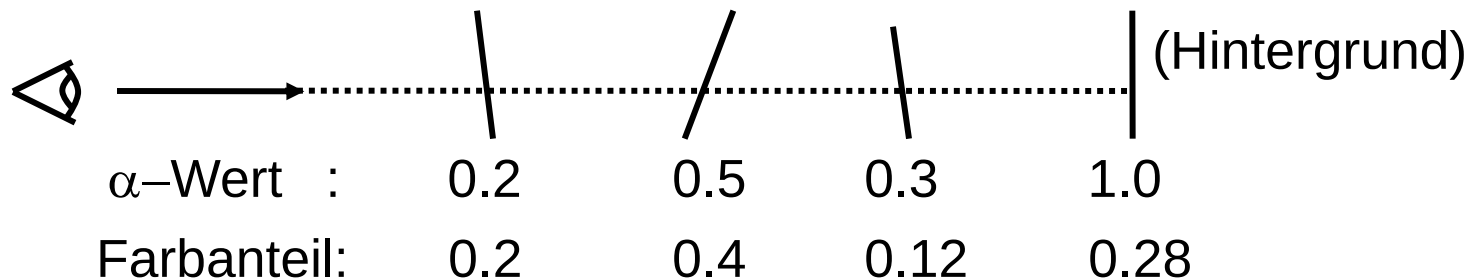
- Objekt im Halbraum **gegenüber des Augpunkts** können Objekte des anderen Halbraums nicht verdecken, können also **gezeichnet werden**.
- BSP ist tiefensortiert: **Blätter** zeichnen
- Traversierungsreihenfolge (Links/Rechts) durch **Position des Augpunkts** gegeben
- Kosten assoziiert mit **Größe/Tiefe** des BSP-Baums

Painter's Algorithmus mit BSP

```
painters(T, PAuge): T BSP-Baum, PAuge = Augpunkt  
  v = root(T);  
  if blatt(v) then zeichne Objektfragmente in S(v)  
  else if PAuge ∈ Hv+ then // Hv+ Linker Halbraum von v  
    painters(T-(v), PAuge);  
    zeichne Objektfragmente in S(v)  
    painters(T+(v), PAuge);  
  else if PAuge ∈ Hv- then // Hv- Rechter Halbraum von v  
    painters(T+(v), PAuge);  
    zeichne Objektfragmente in S(v)  
    painters(T-(v), PAuge);  
  else  
    painters(T+(v), PAuge);  
    painters(T-(v), PAuge);
```

α -Buffer-Algorithmus

- Transparente Flächen besitzen neben den Farbattributen noch einen lokalen Wert für die **Undurchsichtigkeit** (Opazität, Opacity)
 $\alpha \in [0,1]$ (0 = transparent, 1 = undurchsichtig)
- **Pixelfarbe**: Polygonfarbe $\cdot \alpha$ und Hintergrundfarbe $\cdot (1 - \alpha)$



α -Buffer-Algorithmus

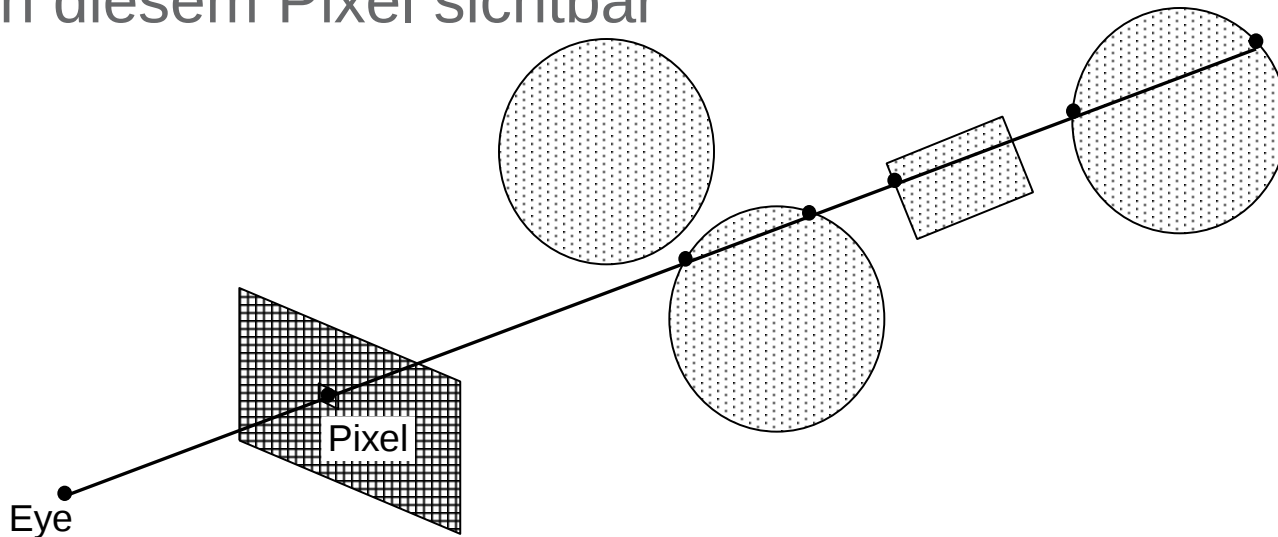
- Transparente Flächen besitzen neben den Farbattributen noch einen lokalen Wert für die **Undurchsichtigkeit** (Opazität, Opacity)
 $\alpha \in [0,1]$ (0 = transparent, 1 = undurchsichtig)
- **Pixelfarbe**: Polygonfarbe $\cdot \alpha$ und Hintergrundfarbe $\cdot (1 - \alpha)$
- α -Buffering funktioniert analog wie z-Buffering, aber:
 - Szene muss von **hinten nach vorne** aufgebaut werden
 - **Tiefensortierung** der Polygone ist notwendig.

Strahlverfolgungs-Algorithmus (RayTracing / RayCasting)

- RayCasting
 - Löst die **Sichtbarkeit** (Strahlperspektive)
 - **Sendet** (to cast) **Strahlen** (ray) durch Datensatz
 - **Akkumuliert** Beiträge entlang des Strahls
- RayTracing
 - Ray Casting + Weiterverfolgung **reflektierter** bzw. **gebrochener** Strahlen
 - **globales** Beleuchtungsmodell
- **Bildraum**algorithmen

RayTracing

- **Verfolge** (to trace) Strahlen vom Augpunkt durch **alle Pixel** der Bildebene
- Berechne **Schnittpunkte** mit allen Objekten der Szene
- RayCasting: Objekt mit **nächst gelegenen** Schnittpunkt ist in diesem Pixel sichtbar



RayTracing

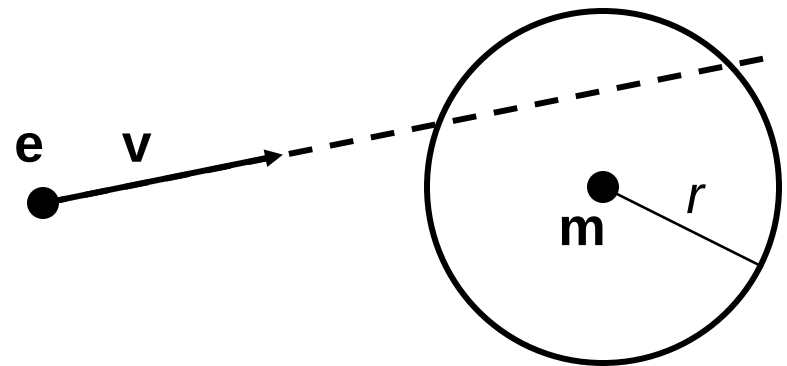
- Schnittpunktberechnung mit dem Strahl

$$r(t) = e + t v$$

e Augpunkt

v Sichtrichtung (Pixel - e)

t Strahlparameter



- Beispiel 1: Schnitt mit einer impliziten Kugel

$$\|x - m\|^2 - r^2 = 0$$

Einsetzen des Strahls $r(t)$ für x liefert:

$$\| e + t v - m \|^2 - r^2 = 0$$

$$(e + t v - m) \cdot (e + t v - m) - r^2 = 0$$

$$((t v) + (e - m)) \cdot ((t v) + (e - m)) - r^2 = 0$$

$$t^2 v \cdot v + 2 t v \cdot (e - m) + (e - m) \cdot (e - m) - r^2 = 0$$

- Lösen der quadratischen Gleichung nach t liefert (max. zwei) **Parameter der Schnittpunkte**

$$s_{1,2} = r(t_{1,2}) = e + t_{1,2} \cdot v$$

- Schnittpunkt mit **kleinstem $t > 0$** liegt Augpunkt am nächsten.

Beispiel 2: Schnitt mit einer Ebene

p Punkt der Ebene

n Normalenvektor

- Einsetzen des Strahls in die Normalenform der Ebene liefert:

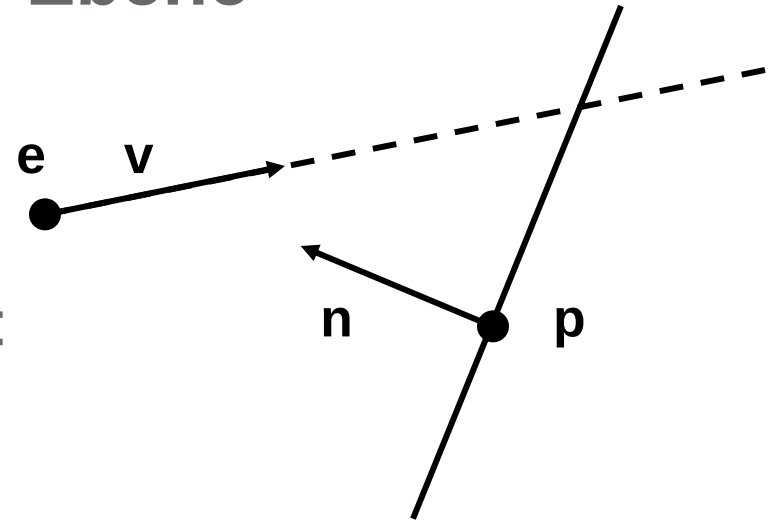
$$(x - p) \cdot n = 0$$

$$(e + t v - p) \cdot n = 0$$

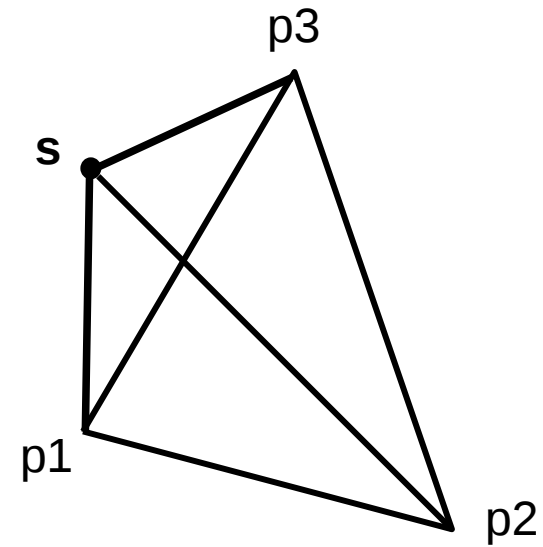
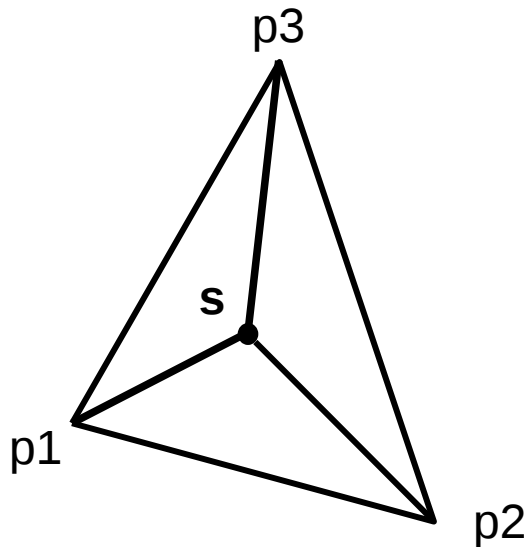
$$t v \cdot n + (e - p) \cdot n = 0$$

$$t = (p - e) \cdot n / v \cdot n$$

- Schnittpunkt: $s = r(t) = e + t \cdot v$



- Beim Schneiden von Polygonen (Dreiecken) ist noch die Gültigkeit des Schnittpunktes zu verifizieren
 - Bestimme **Summe der Flächeninhalte** der Teildreiecke.
 - Ist diese **Summe größer** als der Flächeninhalt des ursprünglichen Dreiecks, so liegt der Punkt **außerhalb**.



RayTracing - Nachteile

- Für **jeden Strahl** muss **jedes Objekt** der Szene daraufhin getestet werden, ob der Strahl das **Objekt schneidet**.
- Bei einer Auflösung 1024×1024 mit 100 Objekten in der Szene müssen **100 Millionen Schnittpunkt-berechnungen** durchgeführt werden!
- Bis zu **95% der Rechenzeit** werden für Schnittpunkt-berechnungen bei typischen Szenen verbraucht.

RayTracing: Beschleunigungsansätze

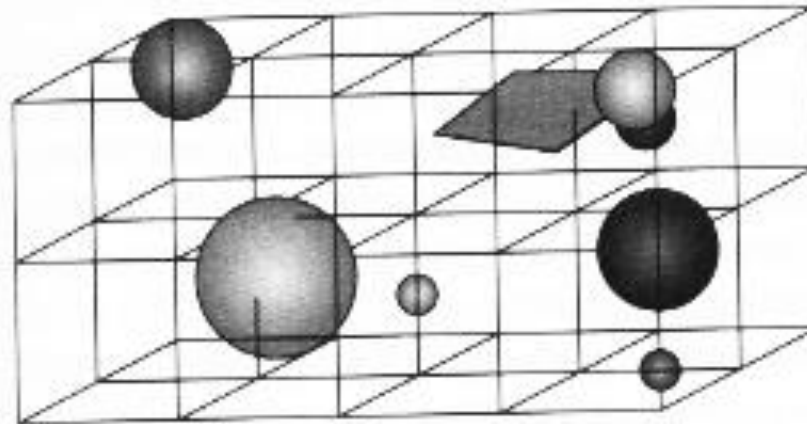
- Transformation der Strahlen **auf die z-Achse**
 - Werden die Objekte mit der gleichen Transformation verschoben, so tritt ein Schnittpunkt immer bei $x=y=0$ auf.
- Bounding-Box/Spheres
 - Komplexe Objekte werden mit **einfacher zu testenden** Hüllvolumen (zB. Bounding-Boxes/Spheres) umschlossen.
 - Liegt **kein Schnittpunkt mit Hüllvolumen** vor, so liegt auch kein Schnittpunkt mit darin enthaltenen Objekten vor.
- Vermeidung von **unnötigen** Schnittpunktberechnungen
 - Hierarchien
 - Raumteilung

RayTracing: Hierarchien

- Baumartige Strukturen von Hüllvolumen
 - Blätter: **Objekte** der Szene (Geometrie)
 - Innere Knoten: **Hüllvolumen** um Objekte der Unterbäume
- Schneidet ein Strahl das **Hüllvolumen eines inneren** Knotens nicht, so entfällt ein Test der untergeordneten Teilbäume.
- **Problem:** Generierung **guter Hierarchien** ist schwierig
 - Geometrisch: Nach **Szenenausdehnung** unterteilen
 - Dichte: Nach **Polygonschwerpunkten** sortieren und unterteilen

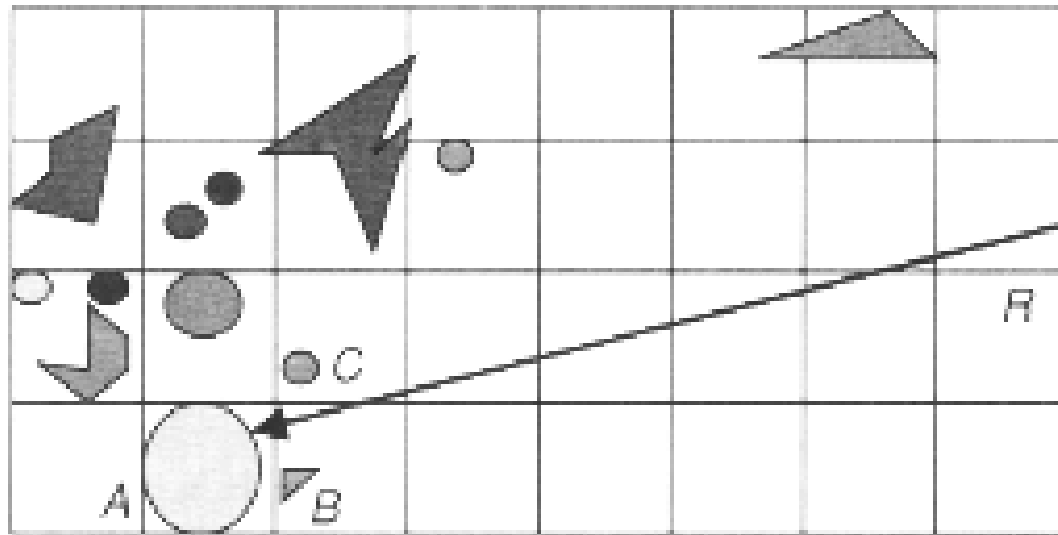
RayTracing: Raumteilung

- **Top-down**-Ansatz
- Zuerst wird **Hüllvolumen der Szene** berechnet.
- Anschließend wird diese in **gleich große Teile** zerlegt.
- Jede Unterteilung enthält eine **Liste mit allen Objekten**, die in der Partition **komplett** oder auch nur **teilweise** enthalten sind.



RayTracing: Raumteilung

- Nur wenn ein Strahl eine **Partition schneidet**, müssen Schnittpunktberechnungen mit den **assozierten Objekten** durchgeführt werden.
- **Durchlaufen der Partitionen** nach Richtung des Strahls



- Große Szenen: #Sichtbarer Primitive << #Primitive
- Wesentliche Zielsetzung: **Zeitersparnis**
- Test muss **erheblich „billiger“** sein als normales Rendern und Visibility-Testen.
 - Pipeline **Flush** (Unterbrechung)
 - Test **+ Rendern** (falls sichtbar)
 - Sehr **häufige Testung** pro Frame
- Also muss Test **einfach** sein.
 - **Komplexen Test** auf wenige Einzelfälle beschränken.

Zwei verwandte Probleme:

- Was ist **Sichtbarkeit**?
Ist ein Objekt sichtbar von einem bestimmten Blickpunkt?
- Was ist **Verdeckung**?
Ist ein Objekt verdeckt (nicht sichtbar) von einem bestimmten Blickpunkt?

Nur aufwendig lösbar

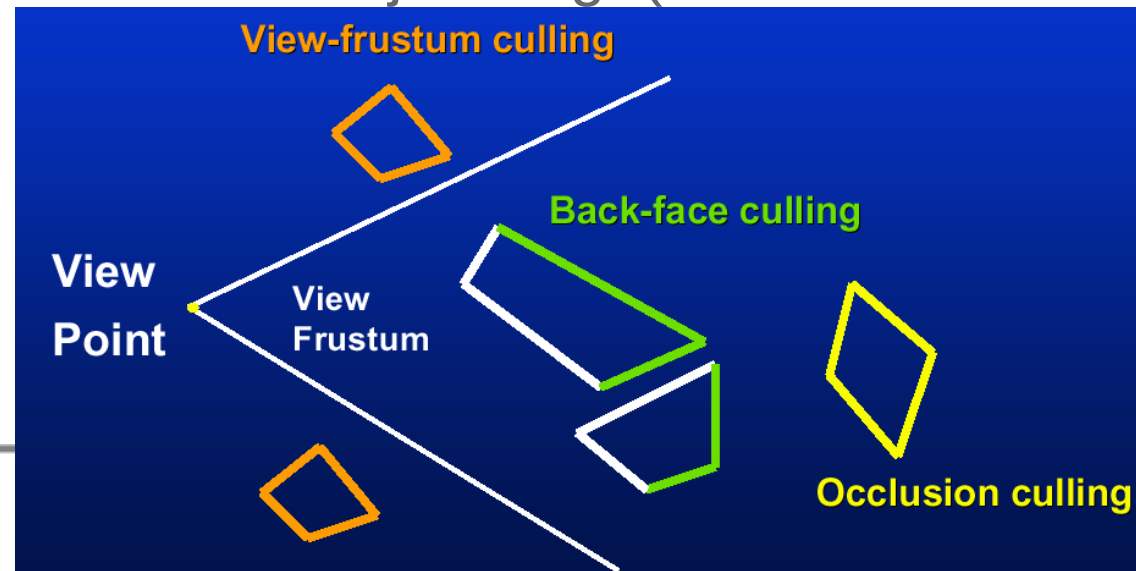
Relativ einfacher lösbar durch Heuristiken

Nicht verdeckt $\neq$ sichtbar!

Unterschiedliche Heuristiken für Verdeckung

Nicht sichtbar, wenn:

- Wenn **Normale nach hinten** zeigt, sehen wir Rückseite (Backface-Culling)
- Wenn Objekt **außerhalb des Blickfeldes** liegt (View-Frustum-Culling)
- Wenn Objekt **hinter anderem** Objekt liegt (Occlusion-Culling)



Unterschiedliche Heuristiken für Verdeckung

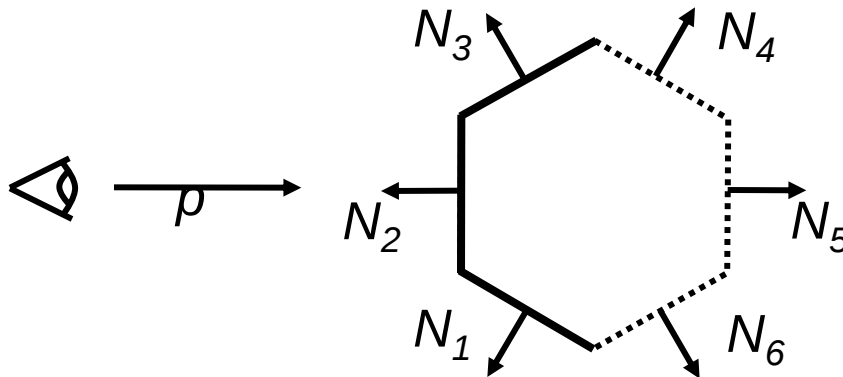
- Objekte von **vorne nach hinten** verarbeiten (Tiefensortierung)
- Verwende **Grenzhüllen/Hüllvolumen**: Wenn Hüllen nicht sichtbar sind, dann auch nicht in ihr eingeschlossene Geometrie
- **Konservativ**: Nicht exakt, aber immer auf der sicheren Seite
 - Objekte, die auf jeden Fall nicht sichtbar sind
 - Sonst darstellen
- **Quantitativ/Approximativ**: Wenn nur ein geringer Anteil sichtbar, als verdeckt behandeln (nichtkonservativ)

Backface-Culling

- **Rückseiten** von undurchsichtigen Objekten nicht sichtbar
- **Seitenorientierung über Normalen** kodiert: konsistente Berechnung wichtig
 - Bei Inkonsistenzen Löcher in Objekten
- Wird von **OpenGL** unterstützt
- Sehr **einfache** Operation

Backface-Culling - Klassifikation der Rückseiten

- **Normalenvektoren N_i** aller Flächen benötigt - müssen ggf. berechnet werden.
- Bei Rückseite zeigt **Normalenvektor N_i in Blickrichtung**, d.h. Skalarprodukt aus Blickrichtungsvektor p und Normale N_i : $p \cdot N_i > 0$



$$p \cdot N_1 < 0 \quad p \cdot N_4 > 0$$

$$p \cdot N_2 < 0 \quad p \cdot N_5 > 0$$

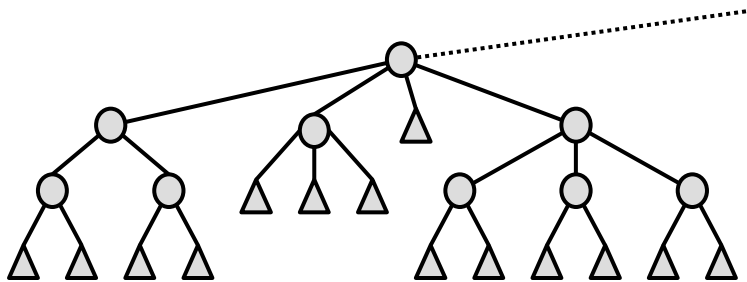
$$p \cdot N_3 < 0 \quad p \cdot N_6 > 0$$

Backface-Culling – Eigenschaften

- Anzahl der Objektpolygone wird durch Entfernen der Rückseiten durchschnittlich **etwa um die Hälfte reduziert**
- Aufwand zur Berechnung des Skalarprodukts **sehr gering**
- Besteht die Szene nur aus **einem einzelnen konvexen** Polyeder, so löst Backface-Culling bereits das Visibilitätsproblem!
- Bei konkaven Polyedern oder Szenen, an denen mehrere Objekte beteiligt sind, kann es zu **Selbst- und/oder Fremdverdeckung** kommen.
 - Hier werden **aufwändigere Verfahren** benötigt.

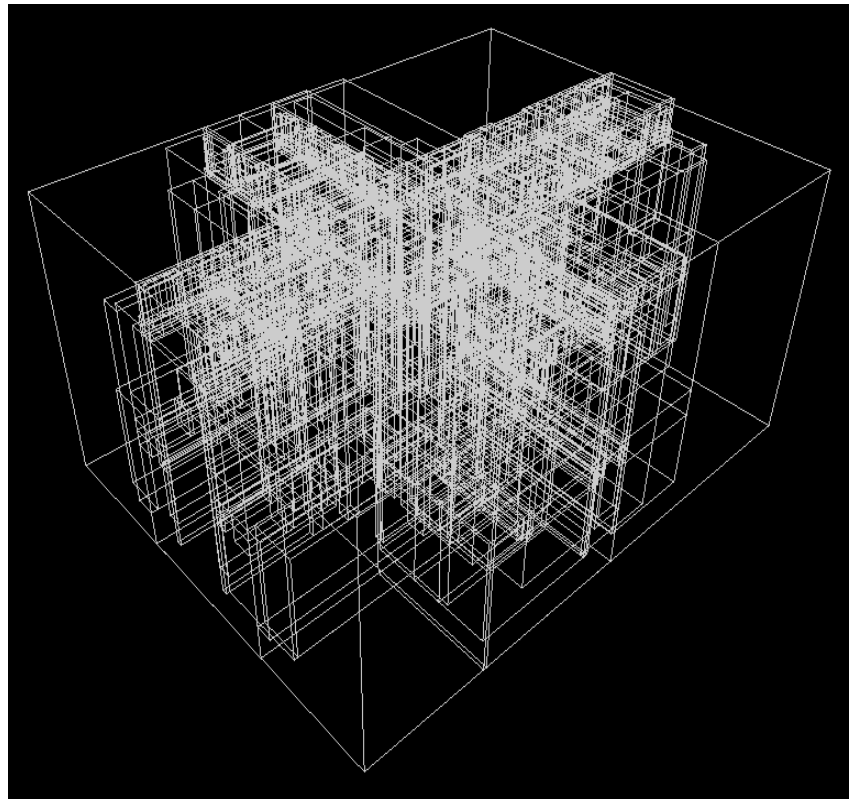
Hierarchische Organisation

- AABB-Hierarchie (Kathedrale)

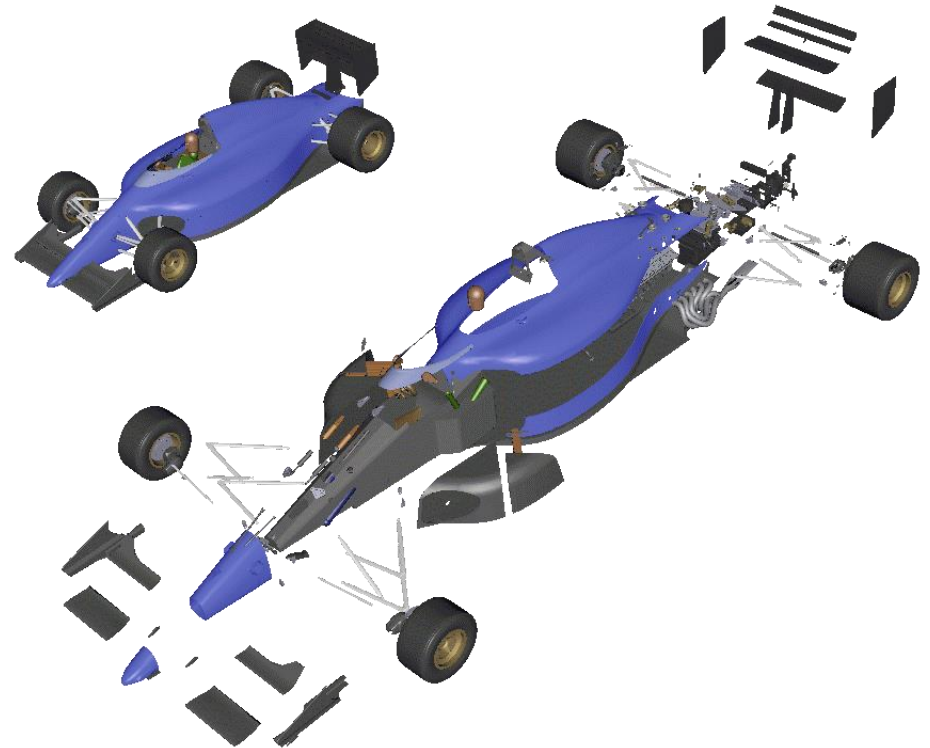
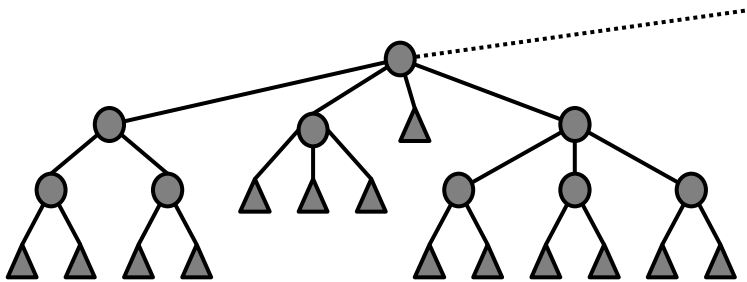


○ Innere Knoten

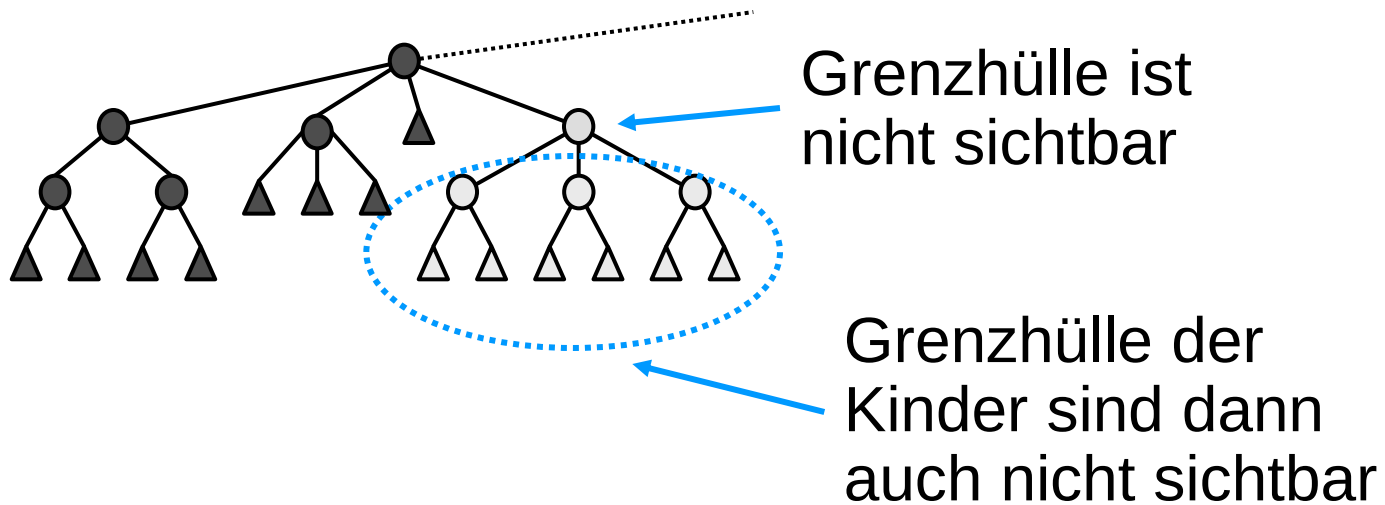
Δ Geometrieblätter



Baumhierarchie Beispiel

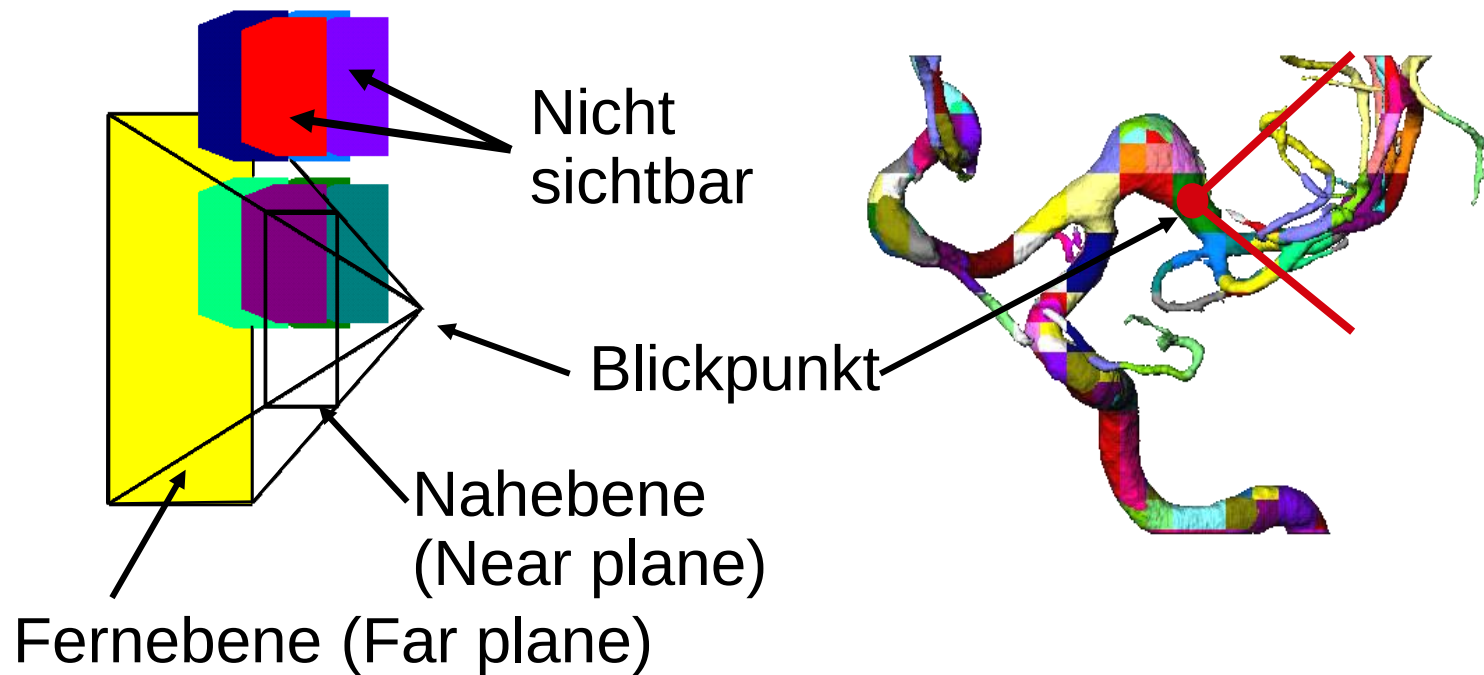


- Hierarchische Verdeckungsrechnung testet Baum **beginnend an der Wurzel**
- Die **Grenzhüllen der inneren Knoten** werden Verdeckungstests unterworfen
- Grenzhülle des inneren Knoten **umhüllen** alle Grenzhüllen der Kinderknoten
- **Abhängig** von diesem Test werden deren **Kinder entfernt, traversiert** oder **dargestellt**



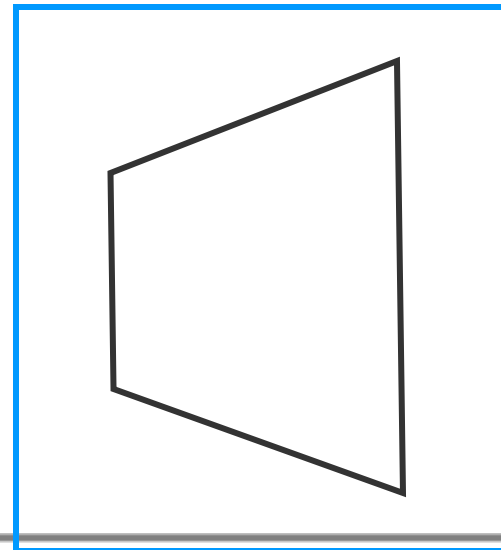
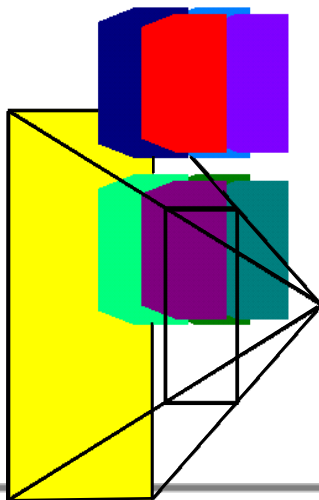
Blickfeldtest (View-Frustum-Culling)

- Liegt Grenzhülle im Sichtvolumen?



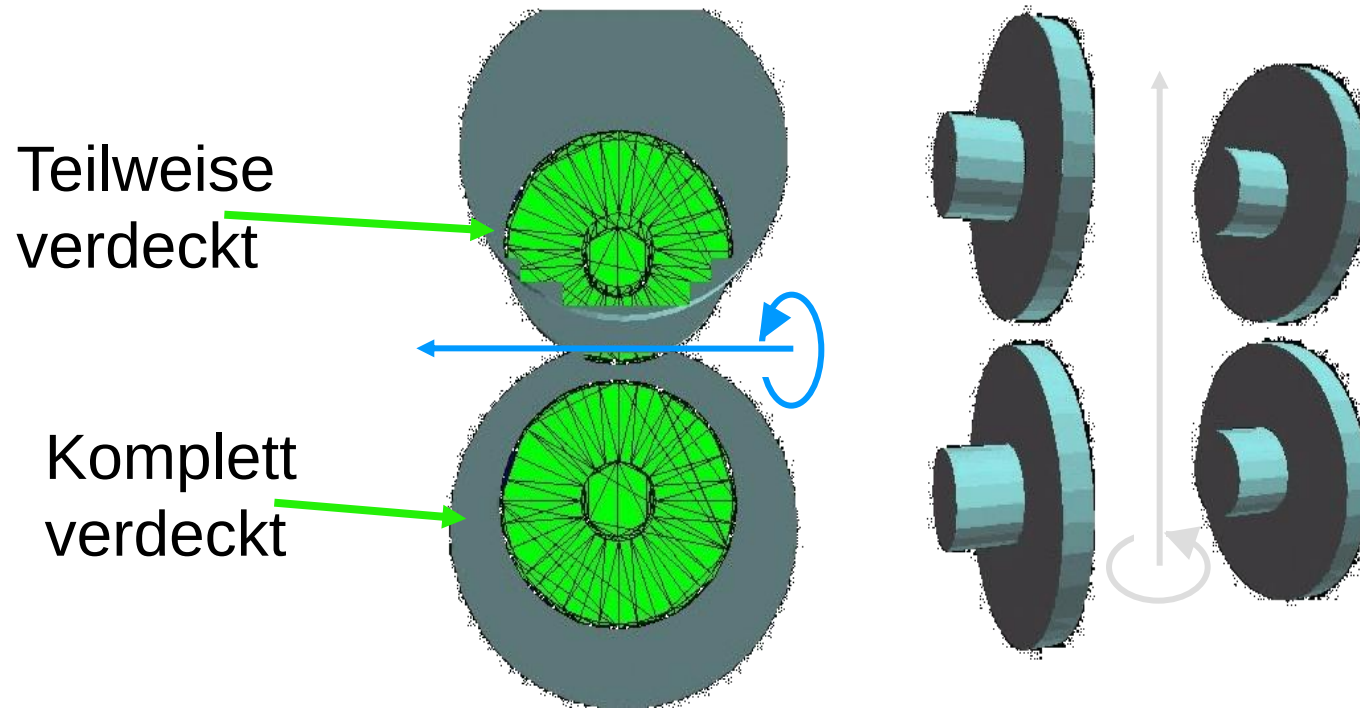
Blickfeldtest (Viewfrustum-Culling)

- **Schneide** Grenzhüllen mit Sichtvolumen/Frustum
- Einfach nach **perspektivischer** Transformation (Einheitswürfel)
- Sonderfall: Grenzhülle **umfasst** Sichtvolumen
- Hierarchischer oder linearer Test



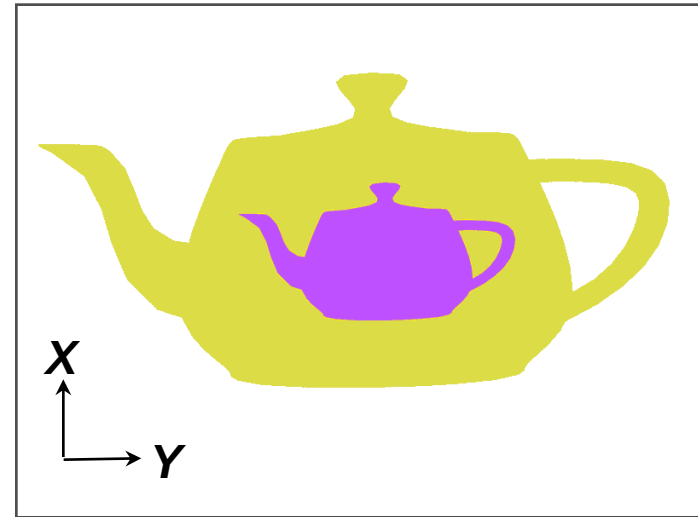
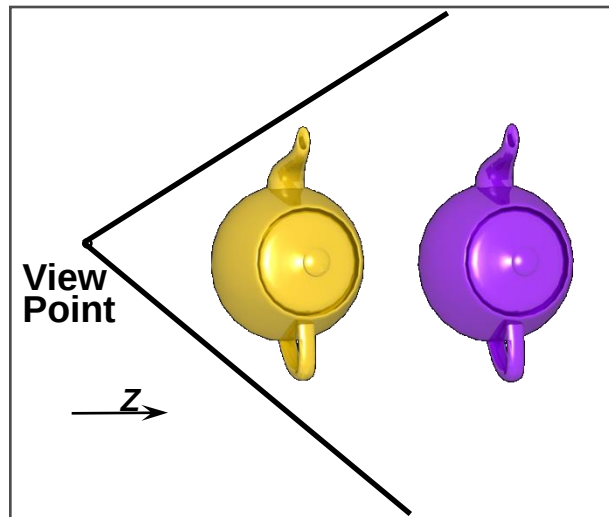
Verdeckungstest (Occlusion-Culling)

- Ist Geometrie von Geometrie verdeckt?



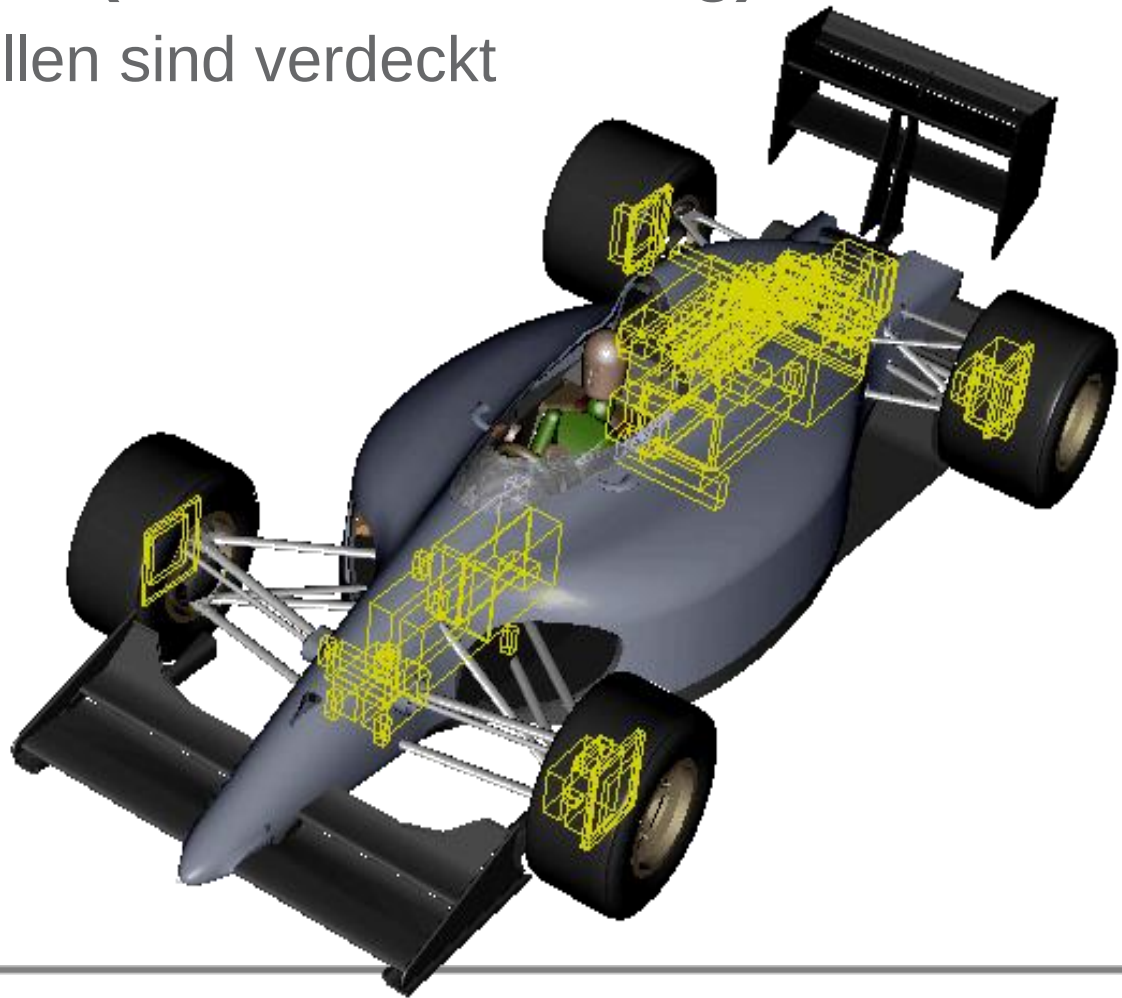
Verdeckungstest (Occlusion-Culling)

- Ist Geometrie von Geometrie verdeckt?



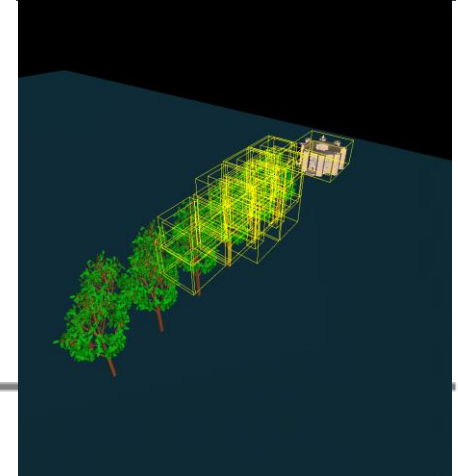
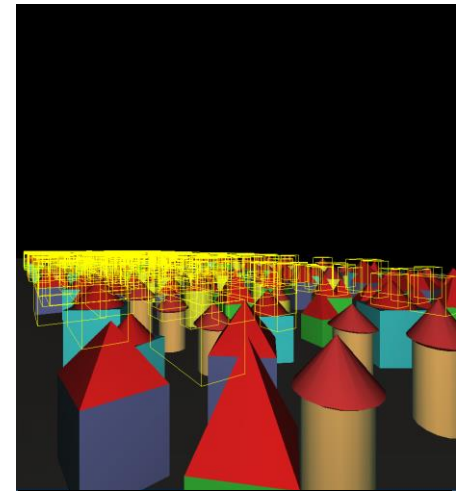
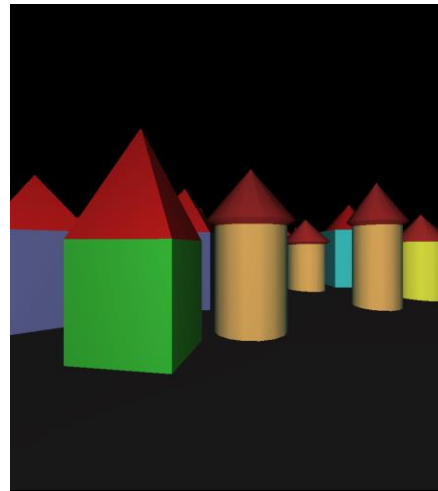
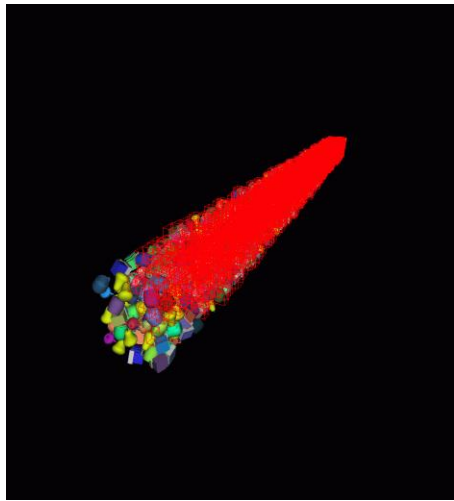
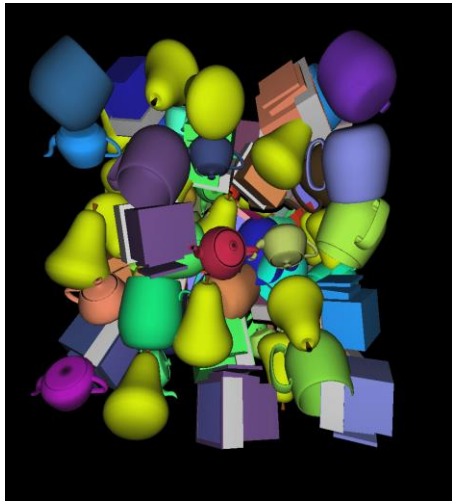
Verdeckungstest (Occlusion-Culling)

- Gelbe Grenzhüllen sind verdeckt



Verdeckungstest (Occlusion-Culling)

- Gelbe/rote Grenzhüllen sind verdeckt



Verdeckungstest (Occlusion-Culling)

- Wie wird **Verdeckung erkannt**?
- **Wo ändert** sich etwas, wenn man Objekt rendern würde.
- **Objekt**raum und **Bildraum**verfahren
 - Hierarchical Z-Buffer (Greene 1993)
 - Hierarchical Occlusion Maps (Zhang 1997)
 - Virtual Occlusion Buffer (Bartz 1999)
- Für allgemeines OC sind **Softwareverfahren zu teuer**
- 1998: Einführung HP OpenGL Extension **Occlusion-Flag** (fx4)
- 2001: NVIDIA OpenGL Extension **Occlusion-Query** (NV 20)
- 2003: Occlusion-Queries in OpenGL 1.5

Verdeckungstest (Occlusion-Culling)

Hardware-Tests in OpenGL (u.a.)

- Occlusion-Flag: **Binäre** Antwort, ob etwas sichtbar wird.
- Occlusion-Queries: **Quantitative** Antwort (wieviel)
- **Stop-and-Wait**-Paradigma
 - Pro Frame, pro Objekt:
 - Teste Hüllvolumen
 - Warte Occlusion-Antwort ab
 - Wenn sichtbar, rendere assoziierte Geometrie
 - Sonst entferne Objekt (culling)
 - Warten (Pipeline-Flush) führt zu Pipeline-Stall

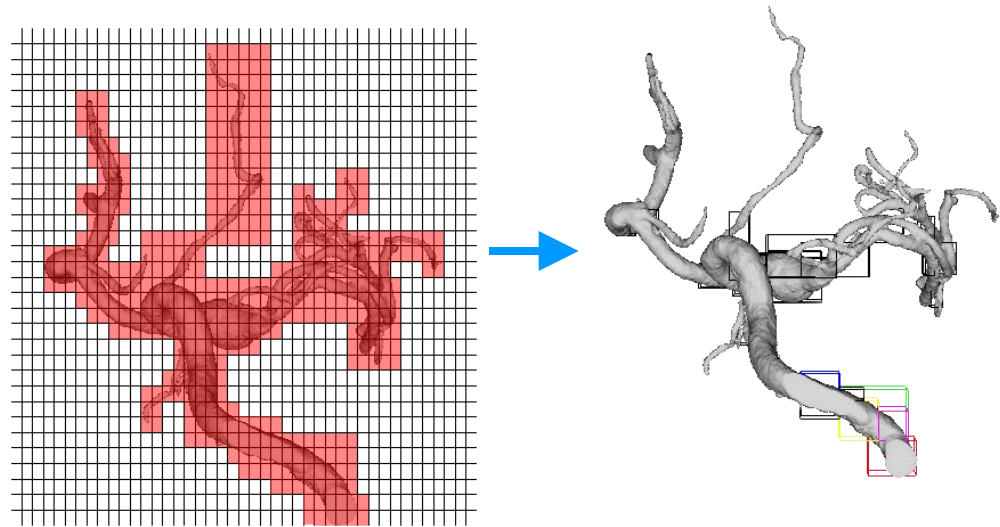
Verdeckungstest (Occlusion-Culling)

Hardware-Tests in OpenGL (u.a.)

- **Stop-and-Wait**-Paradigma
 - Occlusion-Flag: **Binäre** Antwort ob etwas sichtbar wird
 - Occlusion-Queries: **Quantitative** Antwort (wieviel)
- Multiple Queries: Wartet nicht auf **Zwischenergebnisse**
 - **Gebündelte** Queries
 - Kein Pipeline-Stall
 - Nutzt aber auch **keine Zwischenergebnisse** aus
 - Leider: Gesparte Stalls viel billiger als zuviel gerenderte Geometrie: 1/10, 1/20
 - Also: **Vorsortierung** in gegenseitig unabhängige Objekte (sich nicht gegenseitig verdeckend)

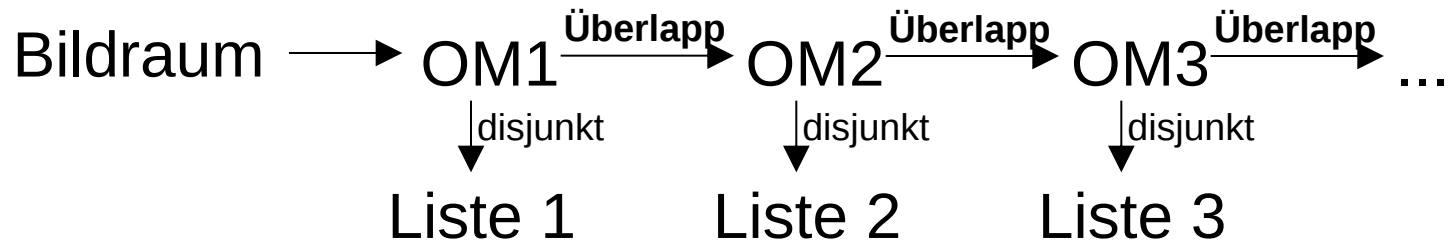
Verdeckungstest (Occlusion-Culling)

- Vorsortierung in gegenseitig unabhängige Objekte
- Vorsortierung mit **OccupancyMap** Hierarchie [Staneker 2003]: Nutze Kohärenz und **reduziere Falsch-Positive**
- Günstiger Vorab-**Overlap**-Test
- Spart auch **redundante** Occlusion-Queries



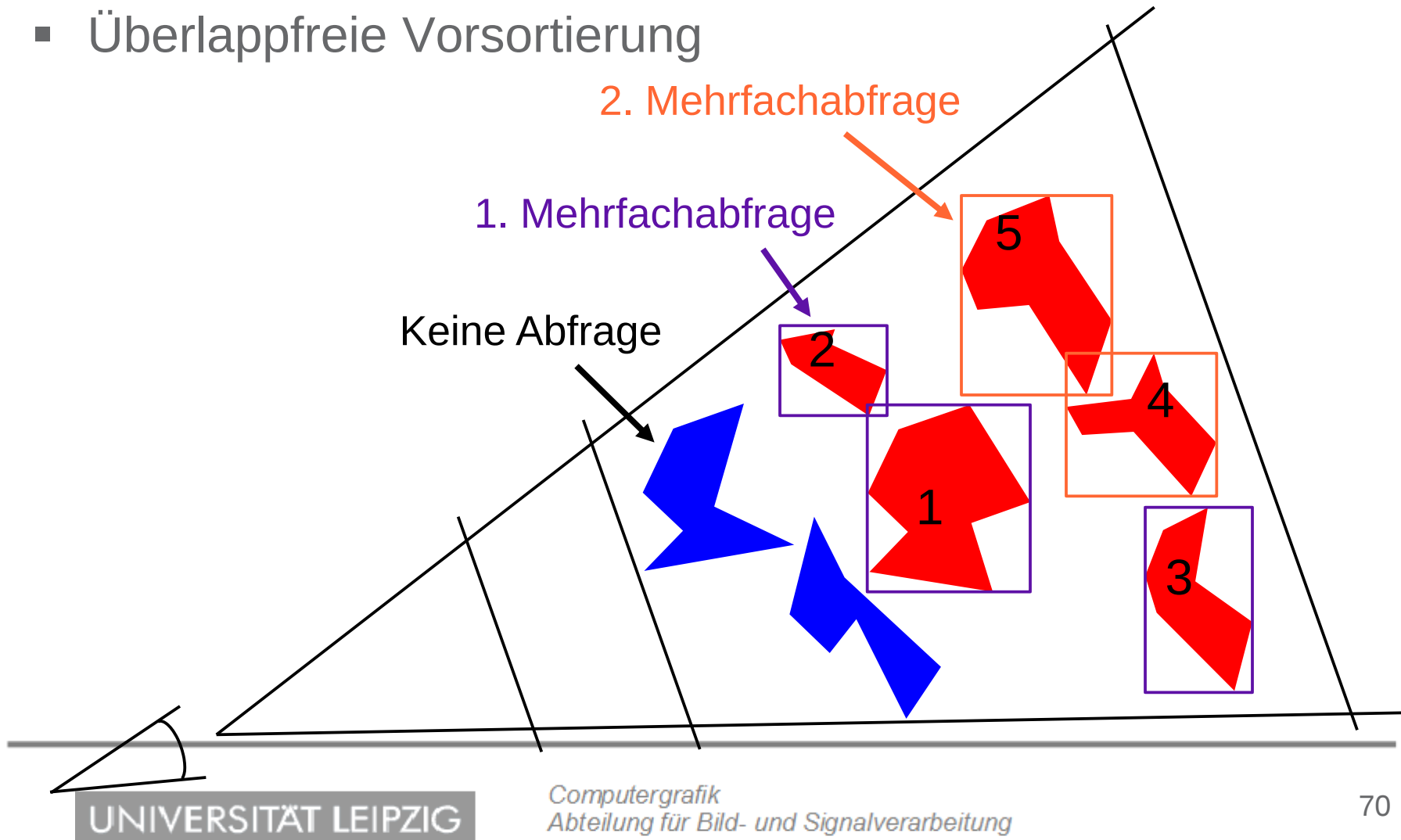
Verdeckungstest (Occlusion-Culling)

- Vorsortierung in **nicht überlappende** Objekte
- Nutze **mehrere** OccupancyMaps (OM)



Verdeckungstest (Occlusion-Culling)

- Überlappungsfreie Vorsortierung



5.2 Verdeckungsrechnung

Occupancy Map für
Gesamtfenster



Occupancy Map für
1. Mehrfachabfrage



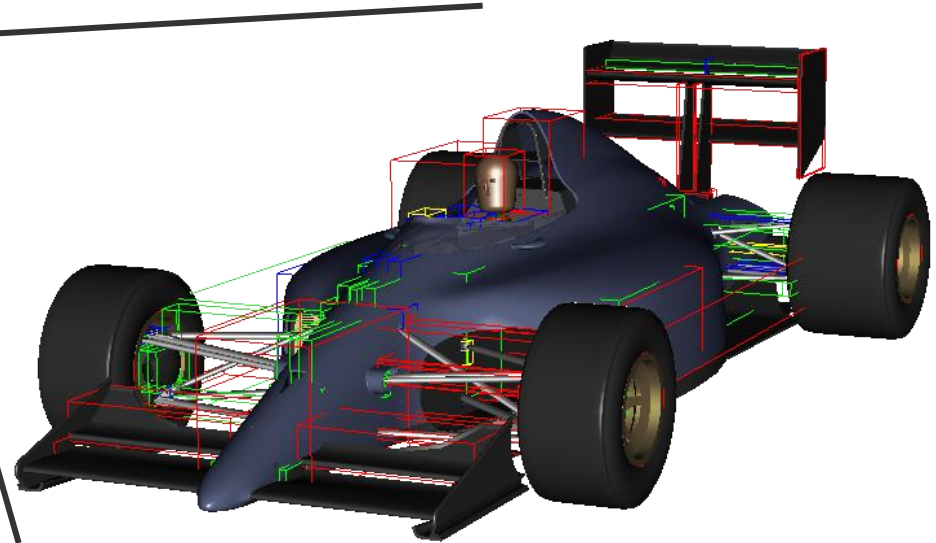
Occupancy Map für
2. Mehrfachabfrage



Occupancy Map für
3. Mehrfachabfrage

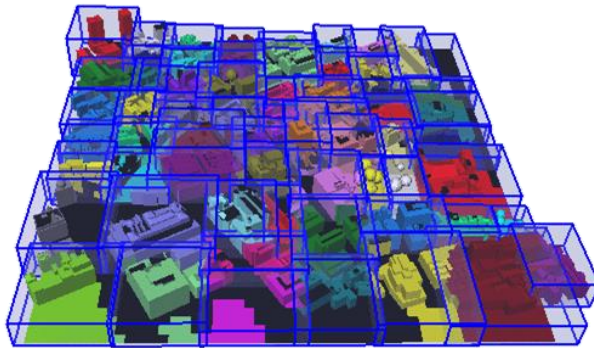


Occupancy Map für
4. Mehrfachabfrage

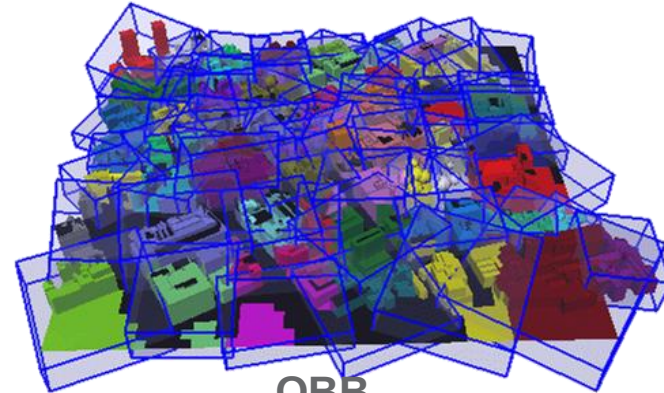


Optimiere Hüllvolumen

- Reduziere **Falsch-Positive** [Bartz 2001]



AABB



OBB



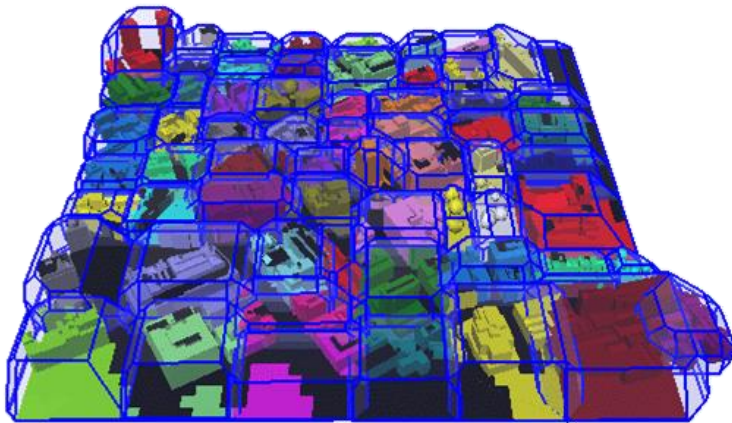
Bounding Spheres



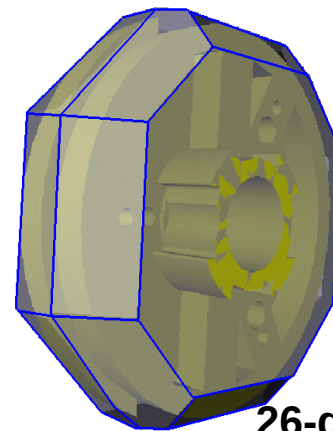
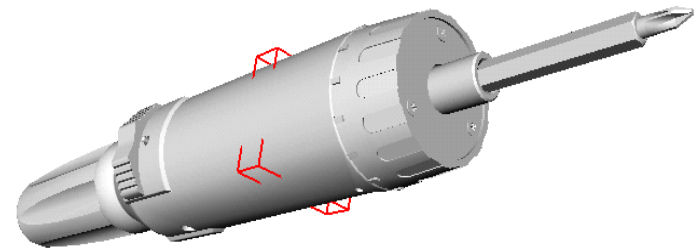
Konvexe Hüllen

Optimiere Hüllvolumen

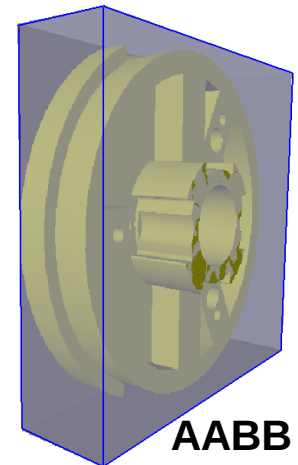
- Reduziere **Falsch-Positive** [Bartz 2001]



k-DOPs



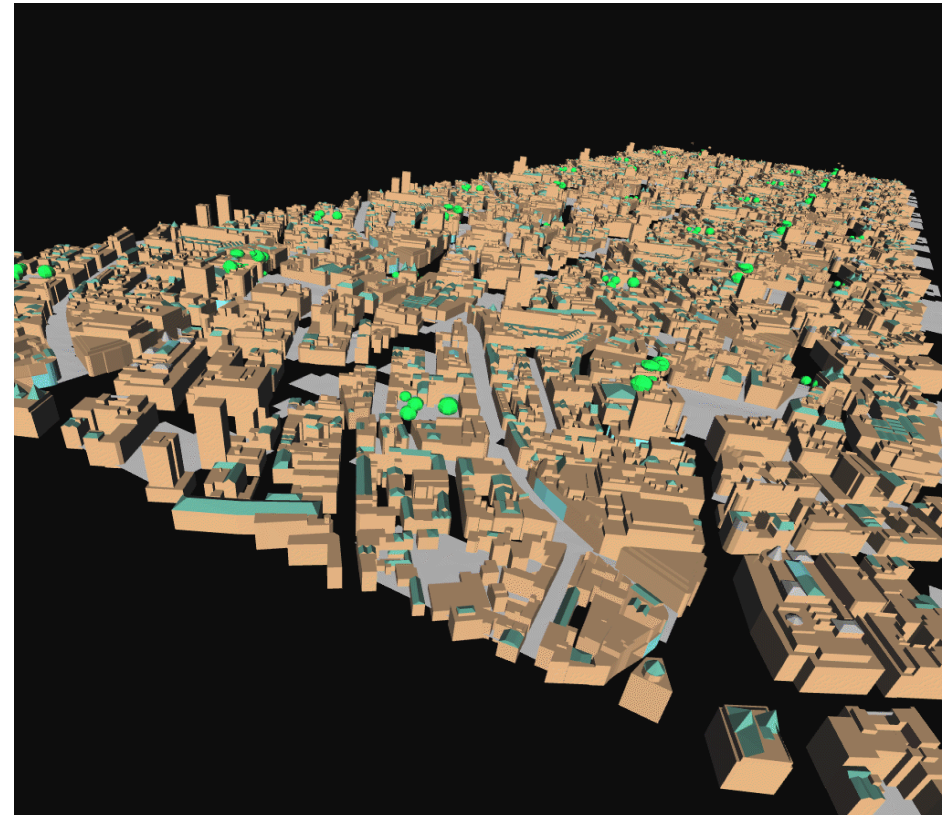
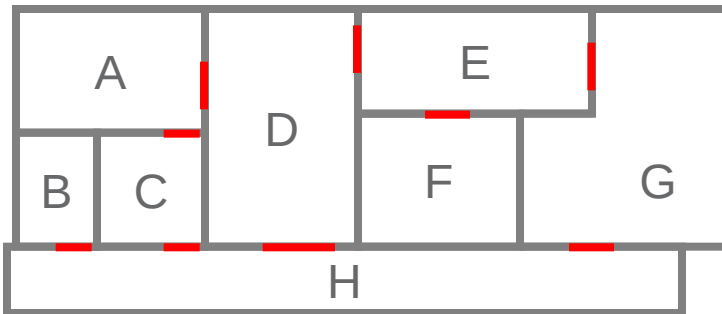
26-dop



AABB

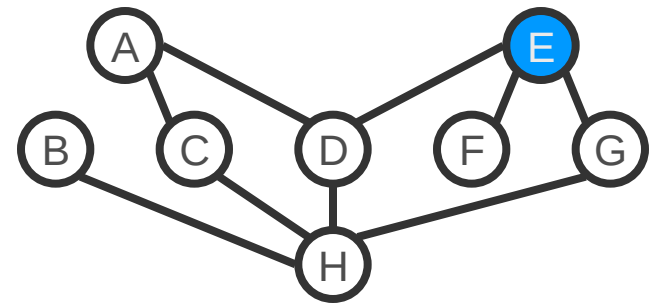
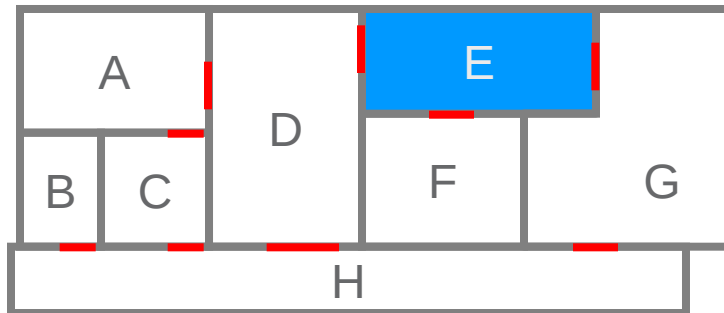
Verdeckungstest (Occlusion-Culling)

- Stadtszenen bieten viel Verdeckung
- Häuser haben feste
Zimmer und Öffnungen:
Cells and Portals
- Sichtbarkeit durch
Portale bestimmt: PVS



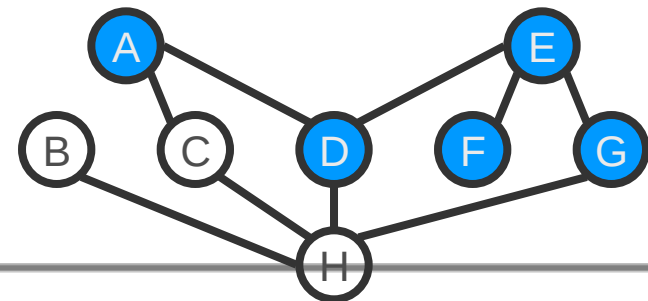
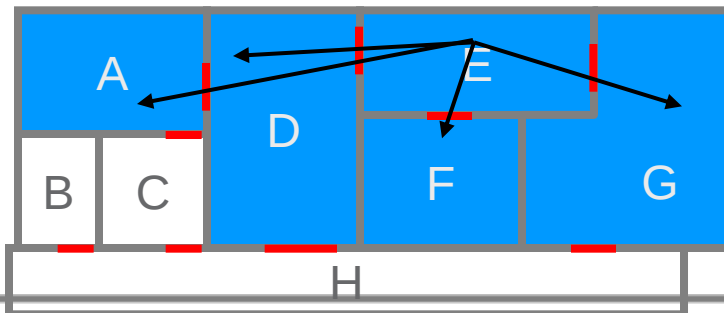
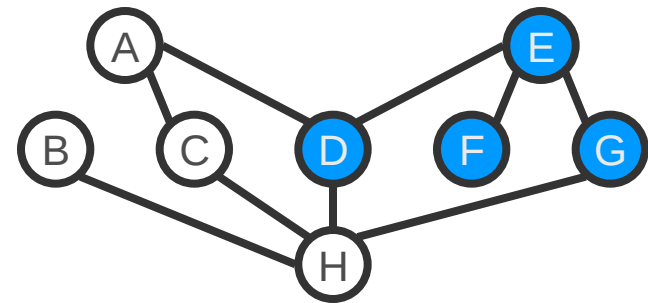
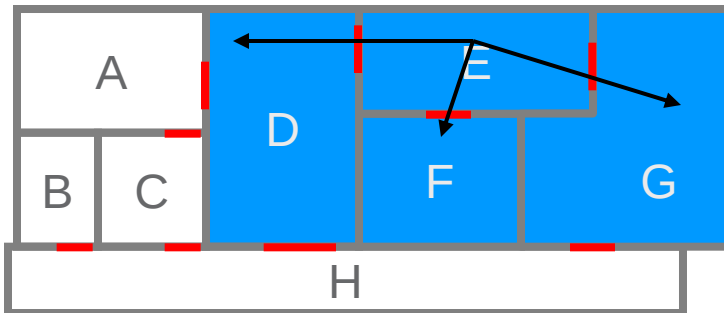
Verdeckungstest (Occlusion-Culling)

- Zellen und Portale [Teller 1992, Luebke 1995]
- Berechne **Nachbarschaftsgraph** zwischen Zellen (durch Portale)
- Zelle ist nur dann sichtbar, wenn sie durch (eine Folge von) Portale(n) sichtbar ist: Sichtlinie



Verdeckungstest (Occlusion-Culling)

- Zellen und Portale
- Speicher für jede Zelle **alle möglicherweise sichtbaren** Zellen
- **Potentially Visible Sets** (PVS)
- Sichtbarkeit für alle Standpunkte: **AbstractGraph**



- Computergraphik, Universität Leipzig
(Prof. D. Bartz)
- Graphische Datenverarbeitung I, Universität Tübingen
(Prof. W. Straßer)
- Graphische Datenverarbeitung I, TU Darmstadt
(Prof. M. Alexa)