



3. Rendering

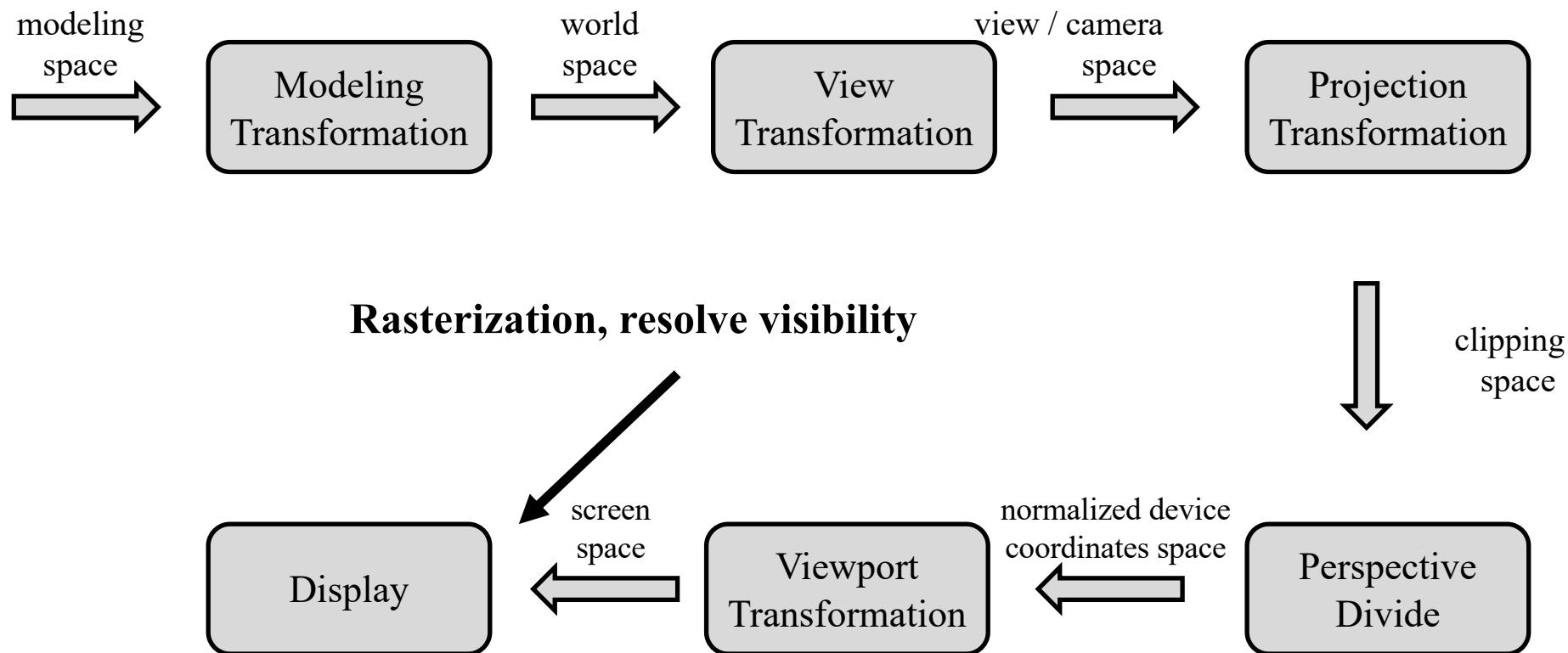
3.6 Sichtbarkeitsberechnung

3. Rendering

3.6 Sichtbarkeitsberechnung



Übliche Koordinatensysteme der 3D-Computergrafik



3. Rendering

3.6 Sichtbarkeitsberechnung



- Werden keine Sichtbarkeitsberechnungen angestoßen, so geben graphische Systeme die Objekte entsprechend der zeitlichen Reihenfolge ihrer Generierung aus.
- Dabei übermalen zuletzt ausgegebene Objekte die zuvor dargestellten Objekte. Diese Vorgehensweise bezeichnet man auch als Painter-Algorithmus.
- Für die anschauliche Darstellung einer 3D Szene sind Sichtbarkeitsberechnungen (HLHSR Hidden Line Hidden Surface Removal) eine wichtige Voraussetzung.

3. Rendering

3.6 Sichtbarkeitsberechnung



- Sichtbarkeitsberechnungen lösen folgendes Problem:
 - Auswahl der (exakten?) Menge von Elementen einer 3D-Szene, die bei gegebenem Blickpunkt sichtbar sind.
- Das Sichtbarkeitsproblem hat eine hohe Komplexität. Deshalb ist es wichtig, diejenigen Objekte auszuschließen, die keinen Beitrag zum Bild liefern können.
- Dies so bezeichnete **Culling** kann nach 3 unterschiedlichen Gesichtspunkten erfolgen:
 - View Frustum Culling
 - Back Face Culling
 - Occlusion Culling

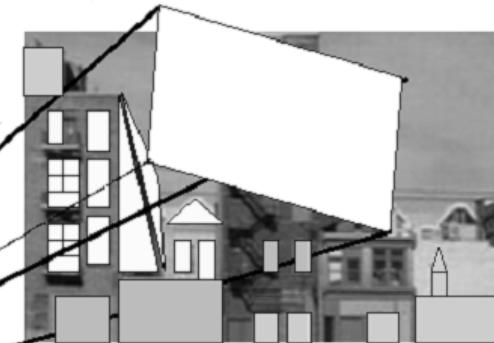
3. Rendering

3.6 Sichtbarkeitsberechnung



Frustum Culling

Ausschluss von Objekten außerhalb des Öffnungswinkels der Kamera



SIGGRAPH 2000,
Course Notes #4



Backface Culling

Ausschluss von Rückseiten

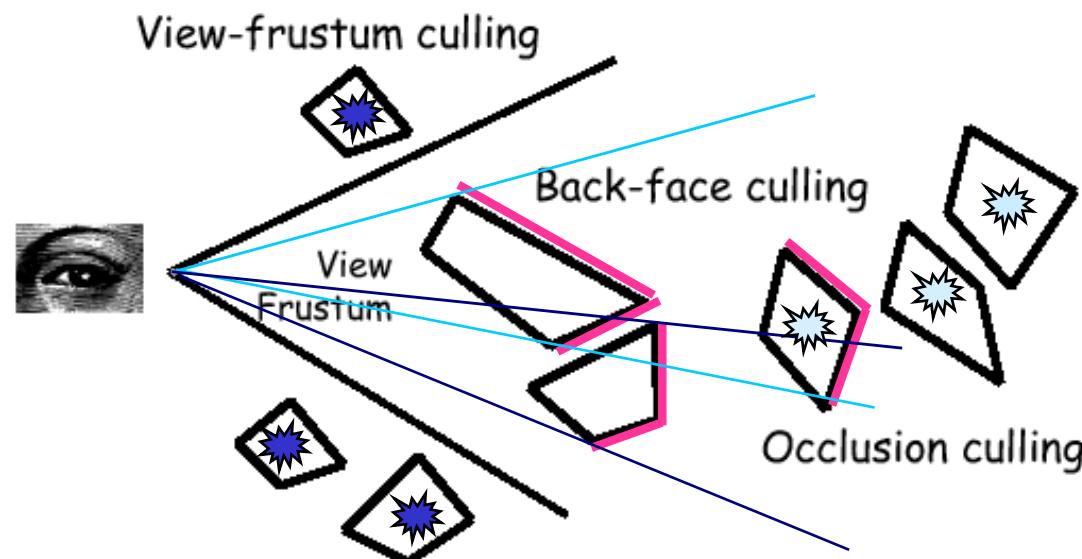


Occlusion Culling

Ausschluss von Objekten, die durch einen Occluder verdeckt sind

3. Rendering

3.6 Sichtbarkeitsberechnung



SIGGRAPH 2000, Course Notes #4

3. Rendering

3.6 Sichtbarkeitsberechnung



■ **Visibility Culling**

- Unser Ziel: *schnelles* Eliminieren von grossen Teilen der Szene, welche im Bild nicht sichtbar sind
- Keine exakte visibility-Berechnung, sondern “quick-and-dirty” konservative Abschätzung
- exakte Berechnung später (Z-buffer)
- Diese konservative Abschätzung wird auch *potentially visible set* oder *PVS* genannt.

3. Rendering

3.6 Sichtbarkeitsberechnung



■ **Zellen & Portale**

- Ziel: laufen durch Architekturmodelle (Gebäude, Städte, Katakomben, Labyrinthe)
- Diese lassen sich einteilen in *Zellen*
 - Räume, Korridore...
- Transparente *Portale* verbinden die Zellen
 - Türen, Fenster, Eingänge...
- Beachte: Zellen können sich nur durch Portale sehen

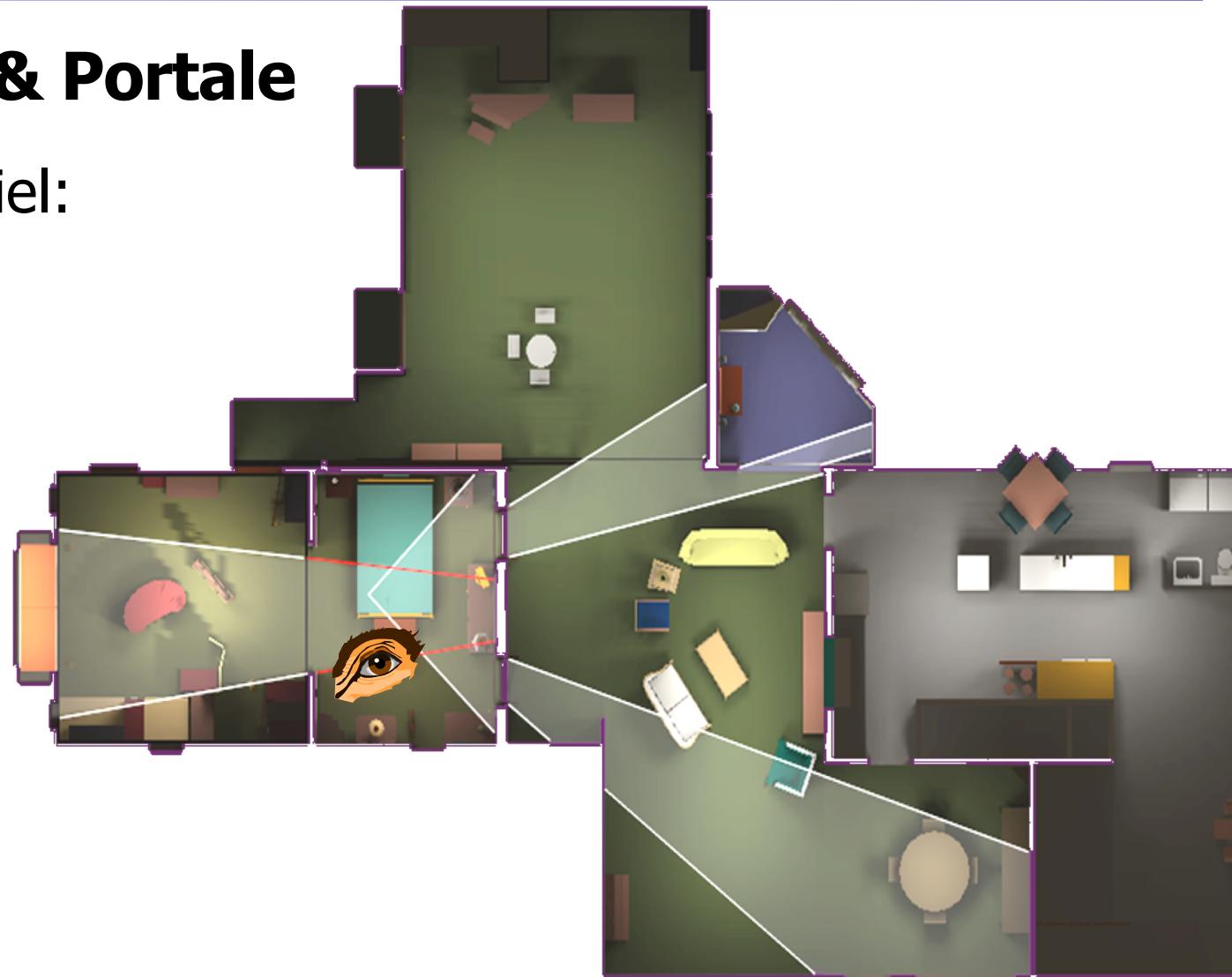
3. Rendering

3.6 Sichtbarkeitsberechnung



- **Zellen & Portale**

- Beispiel:



3. Rendering

3.6 Sichtbarkeitsberechnung



■ **Zellen & Portale**

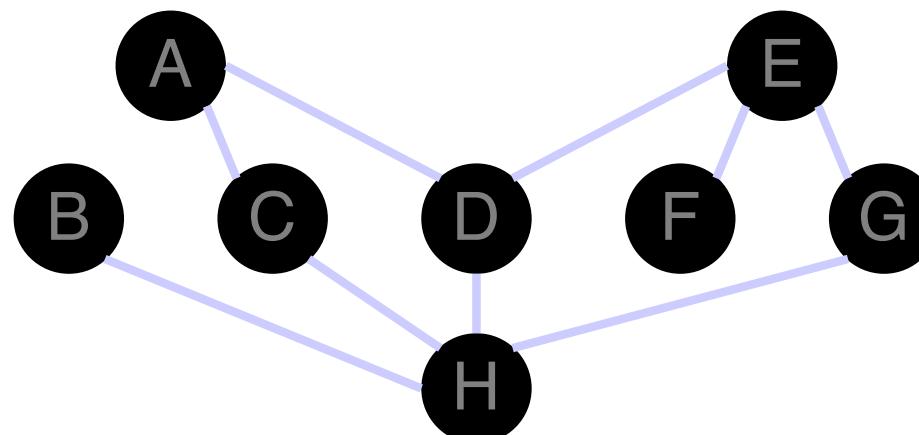
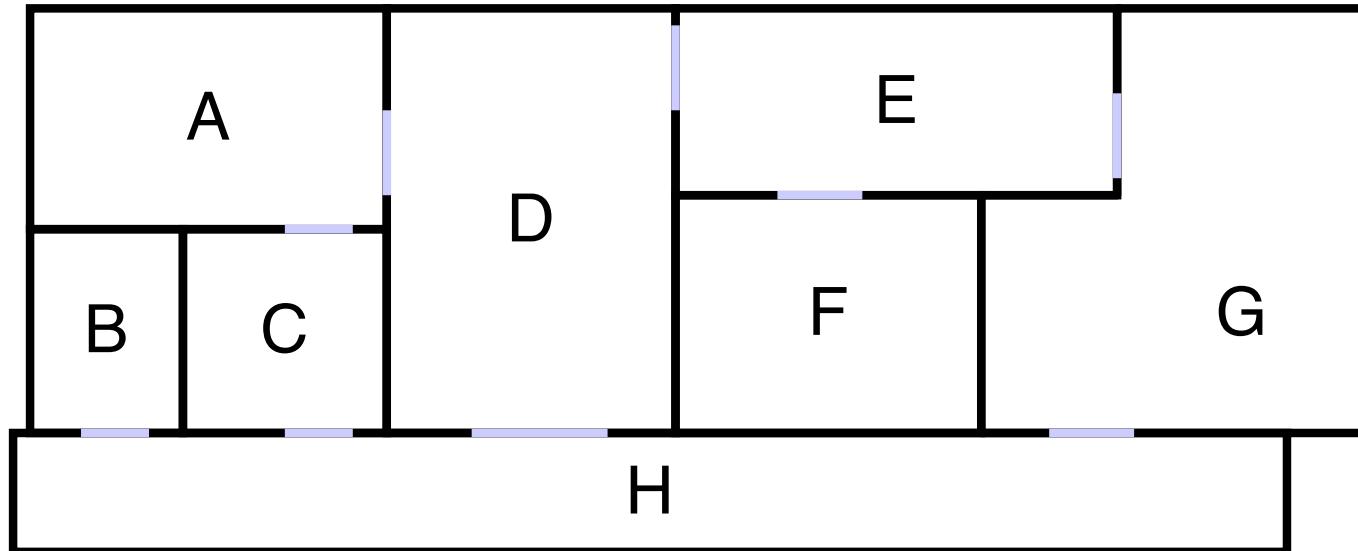
- Idee:
 - Zellen bilden die Basis-Einheit der PVS
 - Bilde einen *adjacency graph* von Zellen
 - Starte mit Zelle, in der der Blickpunkt liegt, traversiere Graph und rendere die sichtbaren Zellen
 - Eine Zelle ist nur sichtbar wenn sie durch eine Folge von Portalen gesehen werden kann

3. Rendering

3.6 Sichtbarkeitsberechnung



- Zellen & Portale

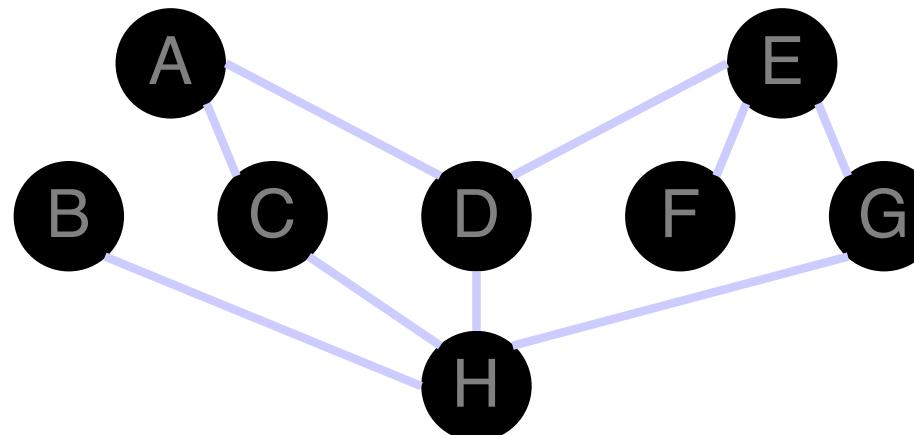
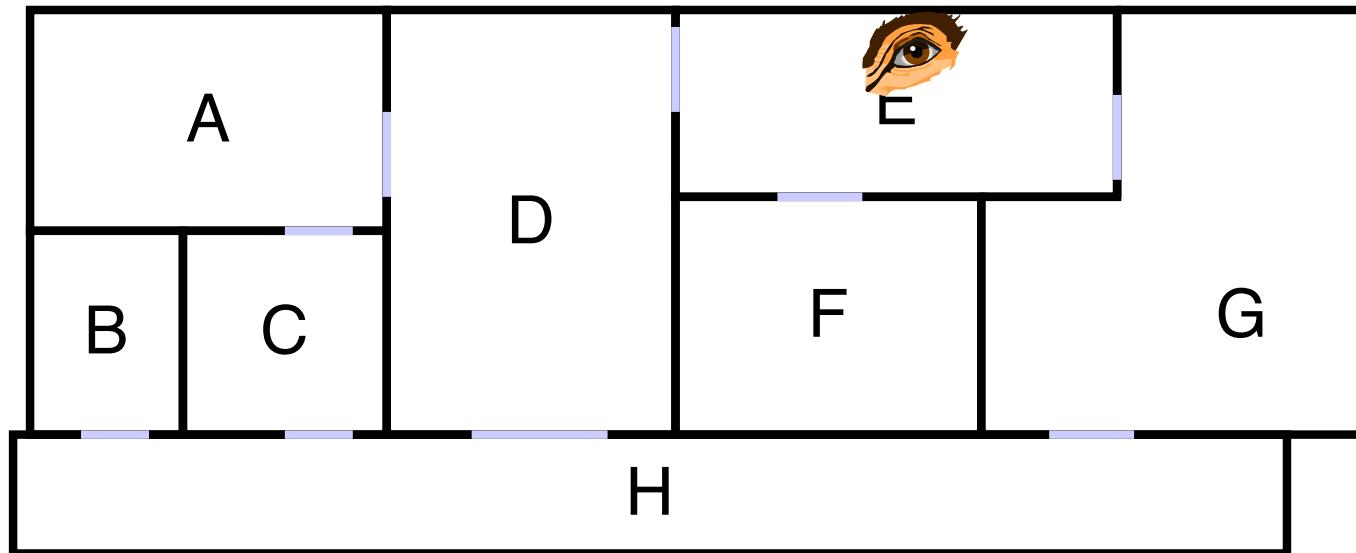


3. Rendering

3.6 Sichtbarkeitsberechnung



- Zellen & Portale

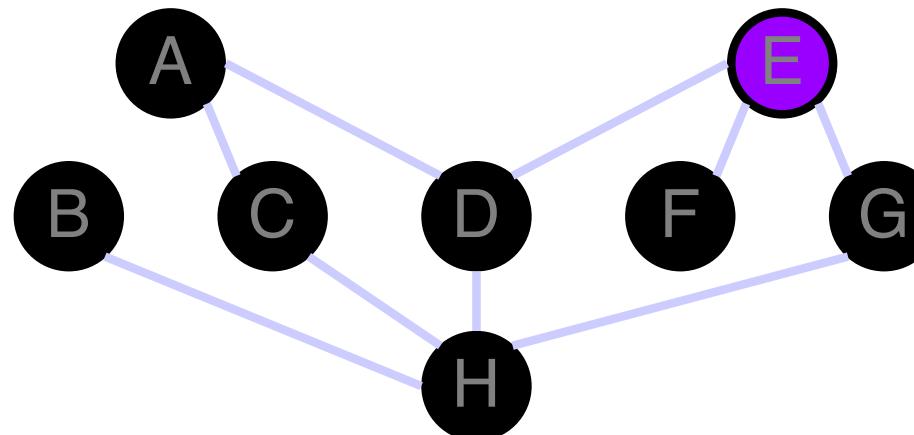
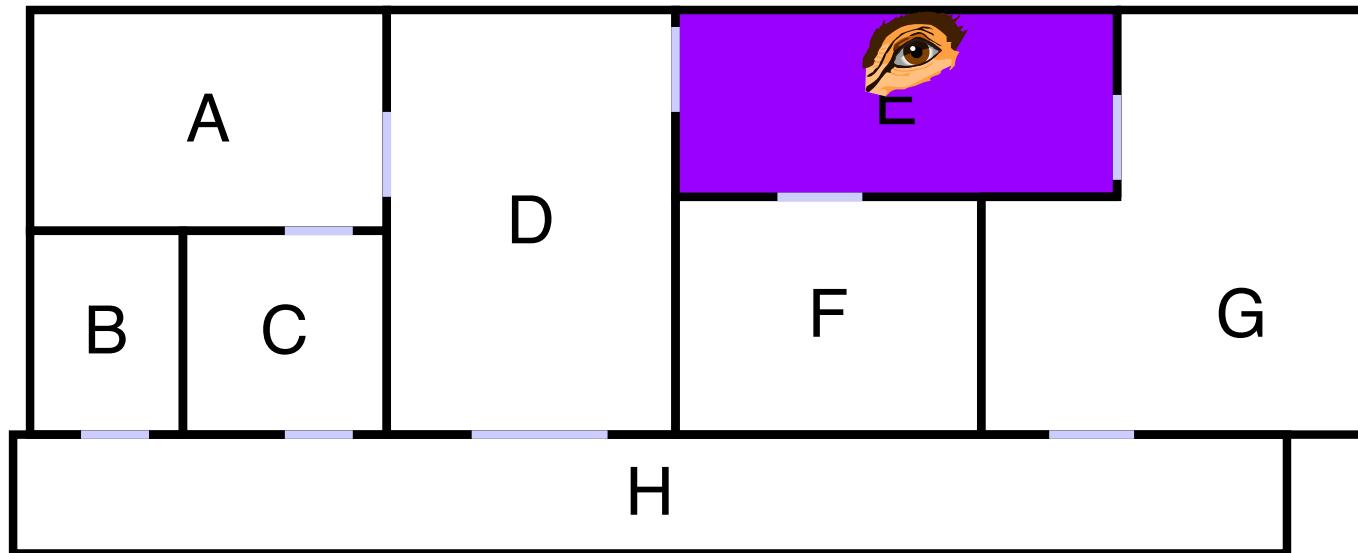


3. Rendering

3.6 Sichtbarkeitsberechnung



- Zellen & Portale

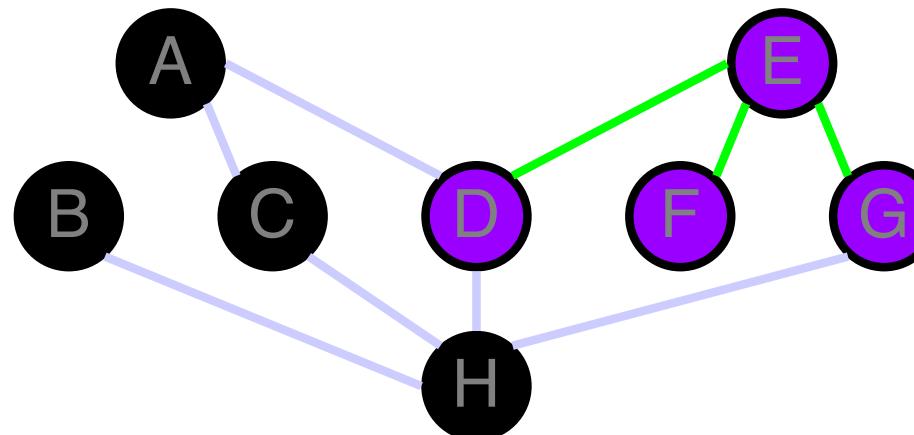
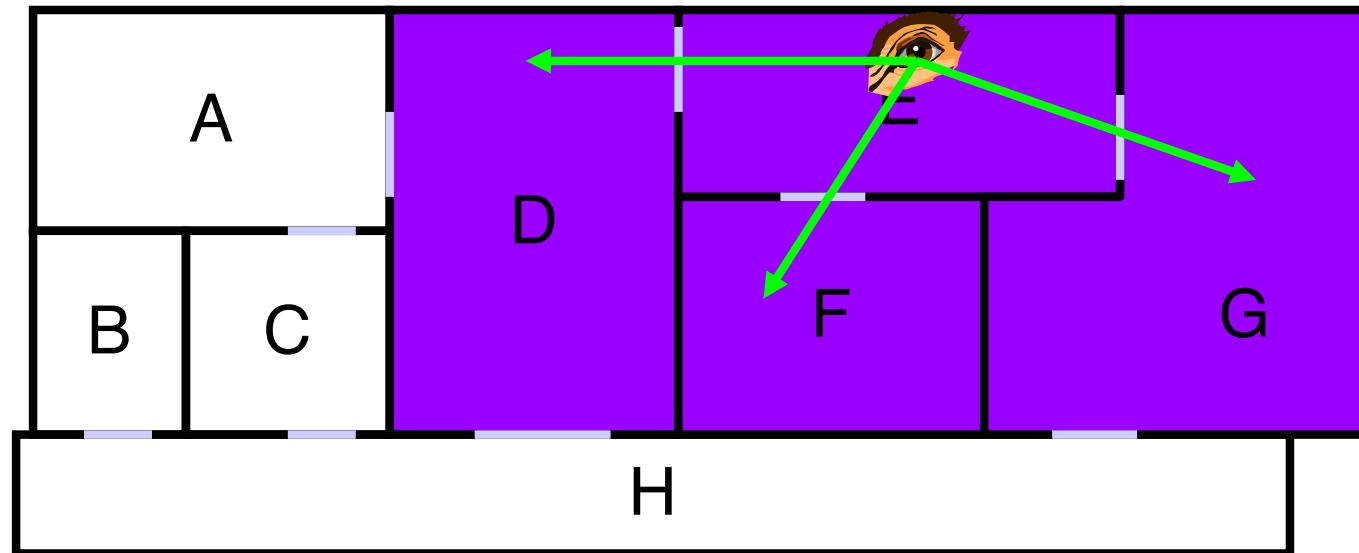


3. Rendering

3.6 Sichtbarkeitsberechnung



- Zellen & Portale

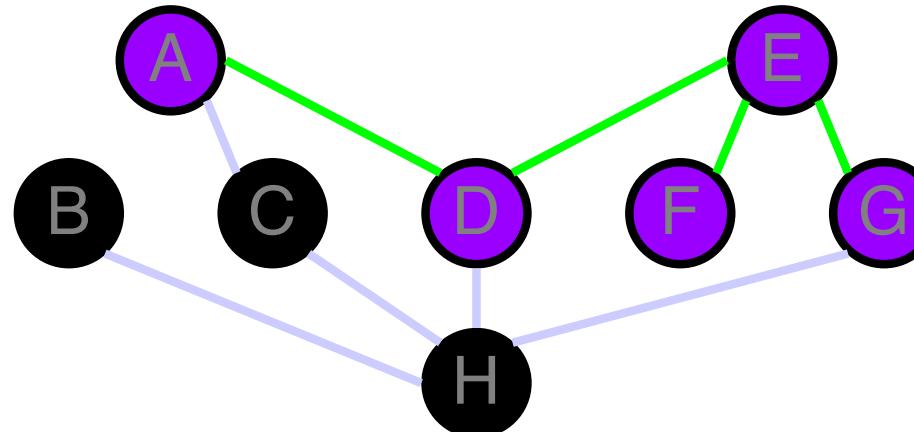
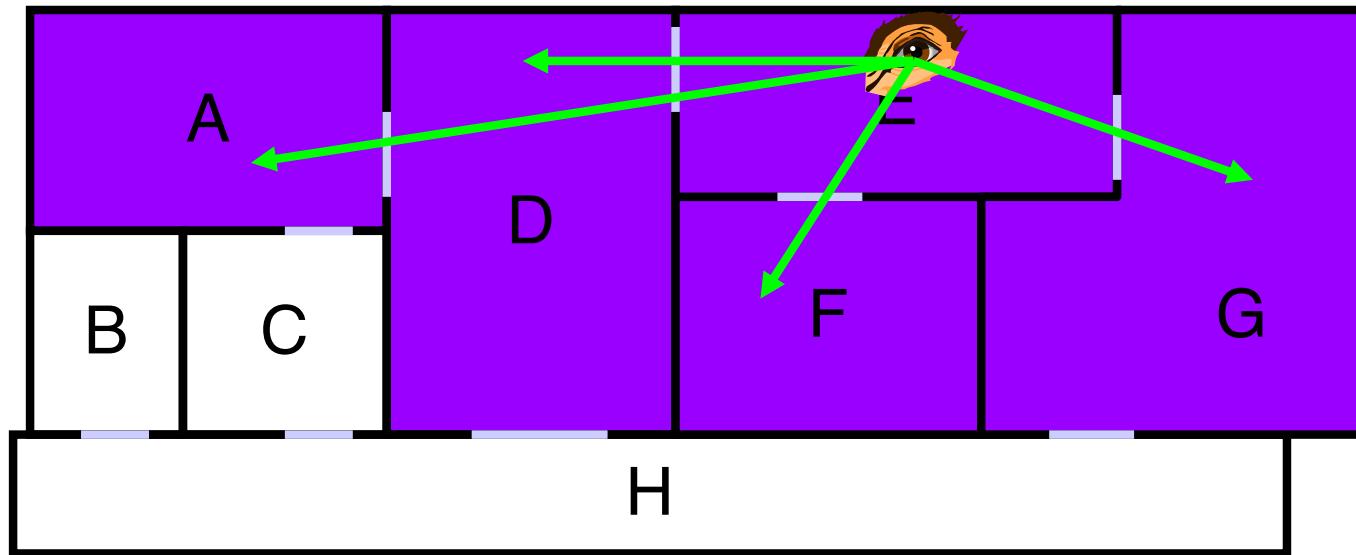


3. Rendering

3.6 Sichtbarkeitsberechnung



- Zellen & Portale

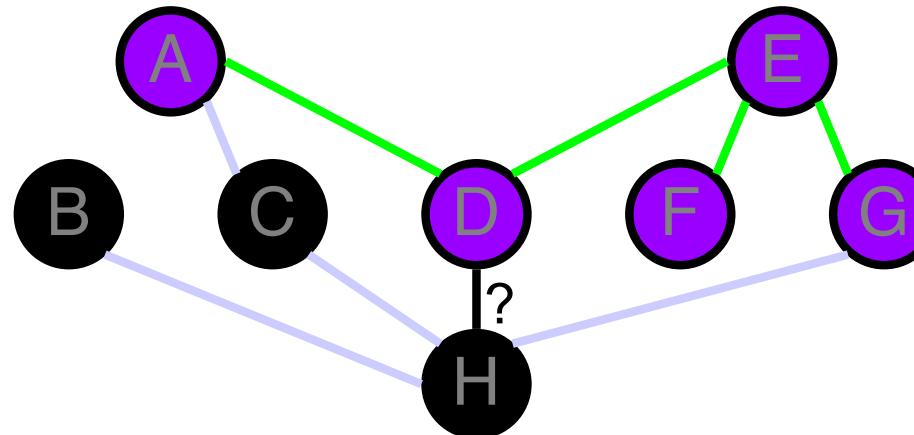
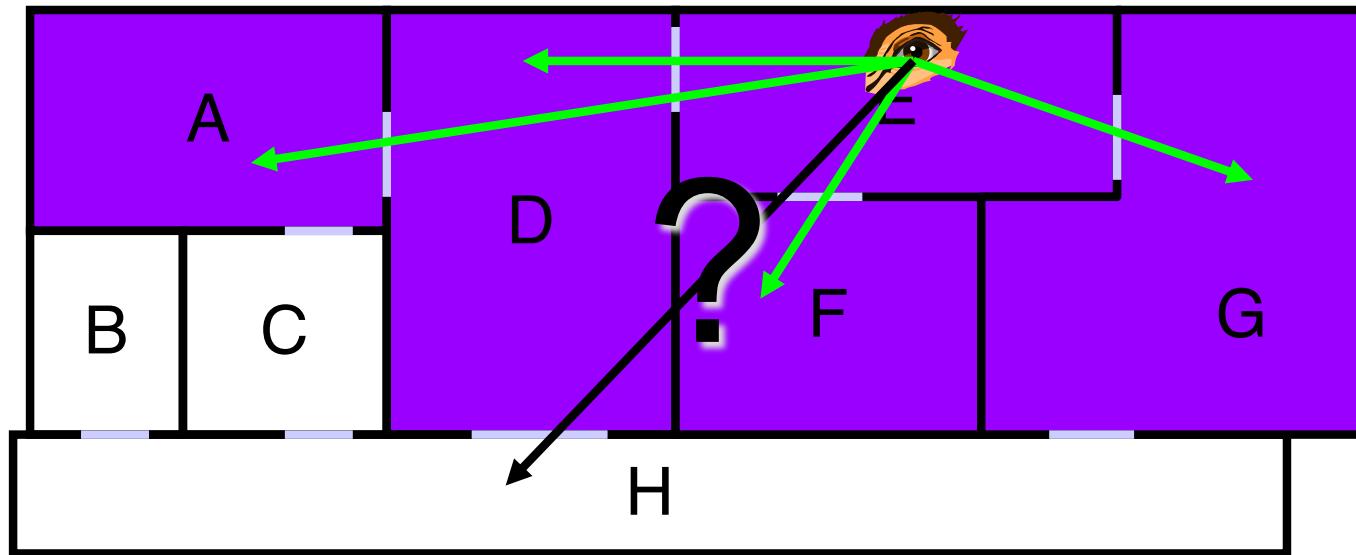


3. Rendering

3.6 Sichtbarkeitsberechnung



- Zellen & Portale

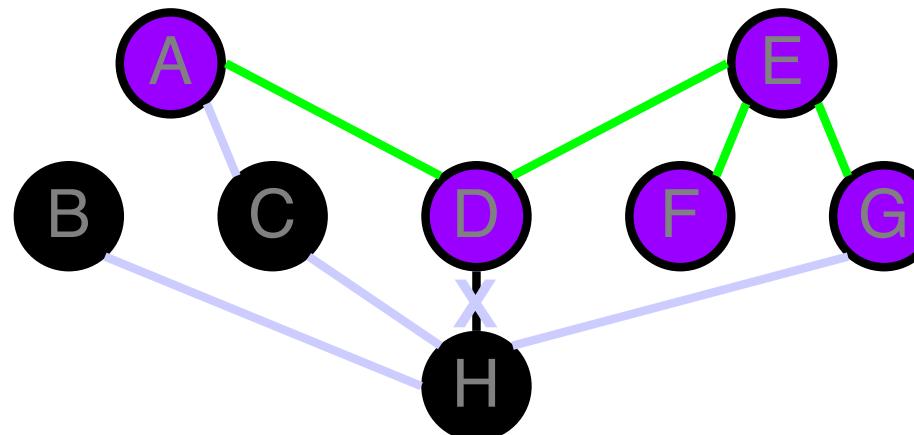
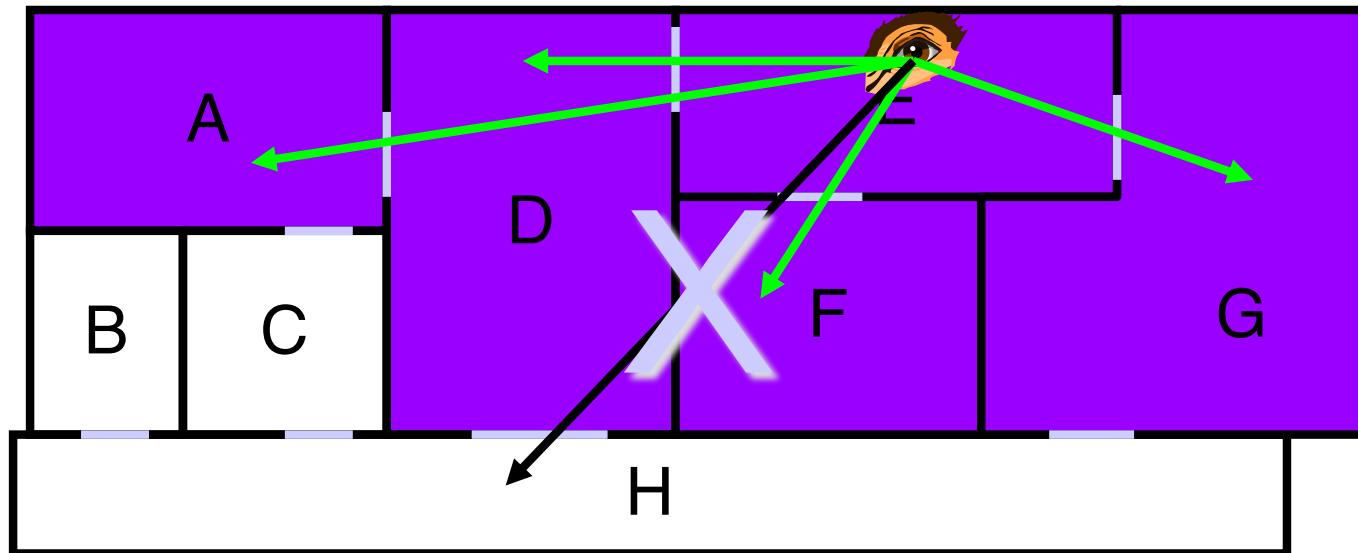


3. Rendering

3.6 Sichtbarkeitsberechnung



- Zellen & Portale



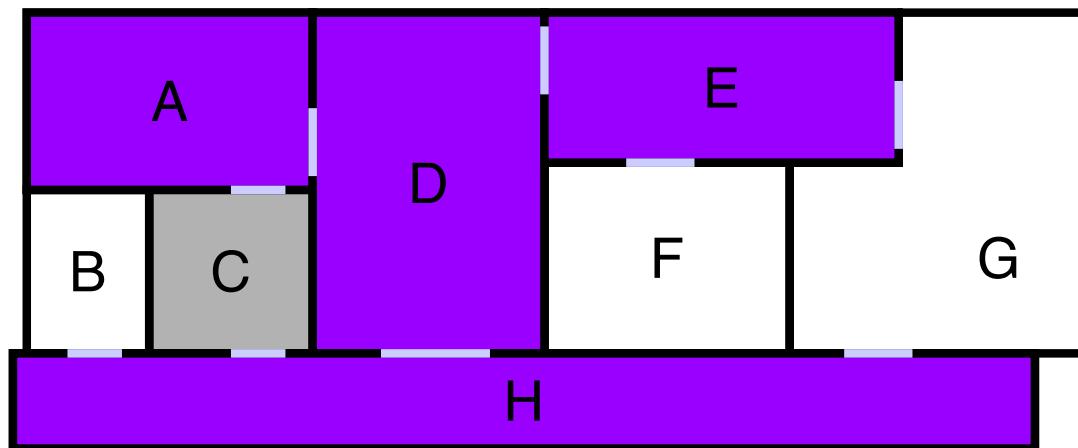
3. Rendering

3.6 Sichtbarkeitsberechnung



- **Zellen & Portale**

- *Blickpunktunabhängige Lösung:* finde alle Zellen welche von einer bestimmten Zelle aus möglicherweise gesehen werden können



C kann *nur* A, D, E, und H sehen

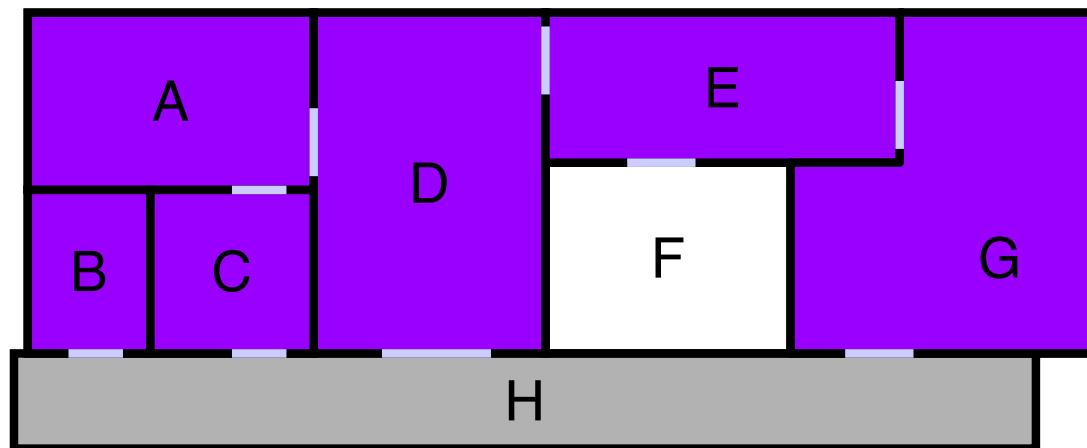
3. Rendering

3.6 Sichtbarkeitsberechnung



- **Zellen & Portale**

- *Blickpunktunabhängige Lösung:* finde alle Zellen welche von einer bestimmten Zelle aus möglicherweise gesehen werden können



H kann niemals F sehen

3. Rendering

3.6 Sichtbarkeitsberechnung



- **Zellen & Portale**

- **Problem:**

- *Wie können wir feststellen ob eine gegebene Zellen von einem Blickpunkt aus sichtbar ist?*
 - *Wie können wir blickpunkt-unabhängige Sichtbarkeit zwischen den Zellen feststellen?*

- **Idee:**

- Diese Probleme lassen sich reduzieren zu Auge-Portal- und Portal-Portal- Sichtbarkeit.

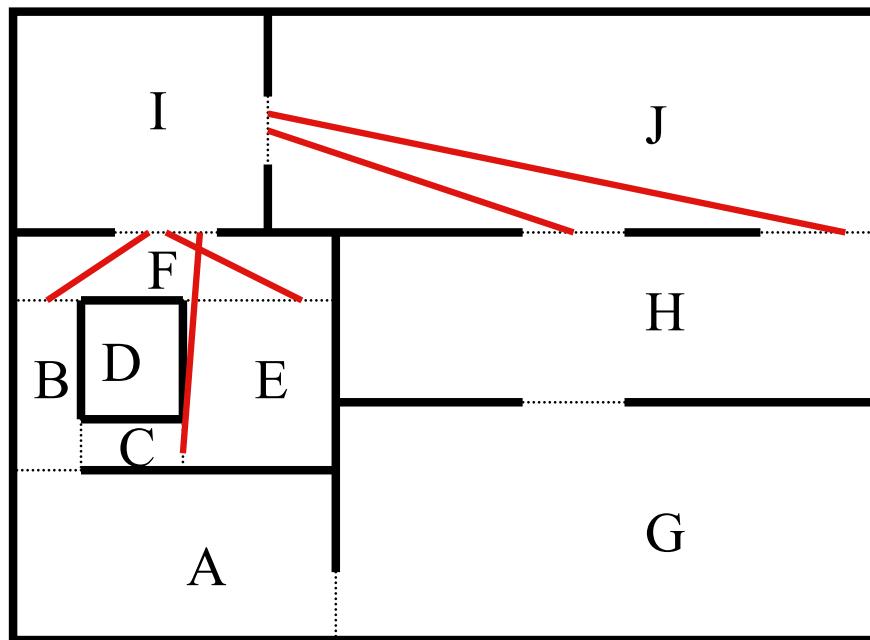
3. Rendering

3.6 Sichtbarkeitsberechnung



■ Zelle-zu-Zelle PVS

- Zelle A ist in der PVs von Zelle B wenn es eine *stabbing line* gibt, welche am portal B beginnt und in der Zelle A endet.
 - Eine *stabbing line* ist eine Strecke, welche Portale verbindet.
 - Benachbarte Zellen sind trivialerweise in der PVS



PVS für I enthält:
B, C, E, F, H, J

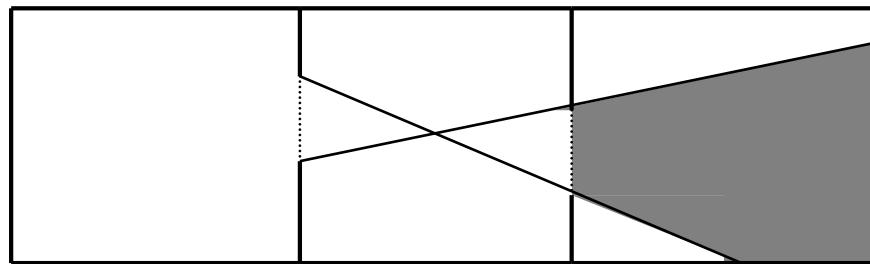
3. Rendering

3.6 Sichtbarkeitsberechnung



■ Zelle-zu-Region PVS

- Identifiziere welche *Regionen* sichtbar sind von einer Zelle aus
 - Füge Objekte innerhalb der Region der PVS zu
- Idee: separierende Ebenen (oder Linien in 2D):
 - Linien, welche durch linke Kante eines Portals und rechte Kante des anderen Portals gehen, und umgekehrt
 - Potentiell sichtbare Zone wird durch Ebenen begrenzt
 - In 3D müssen die maximale Ebenen gefunden werden (Die die Regionen am grössten machen)



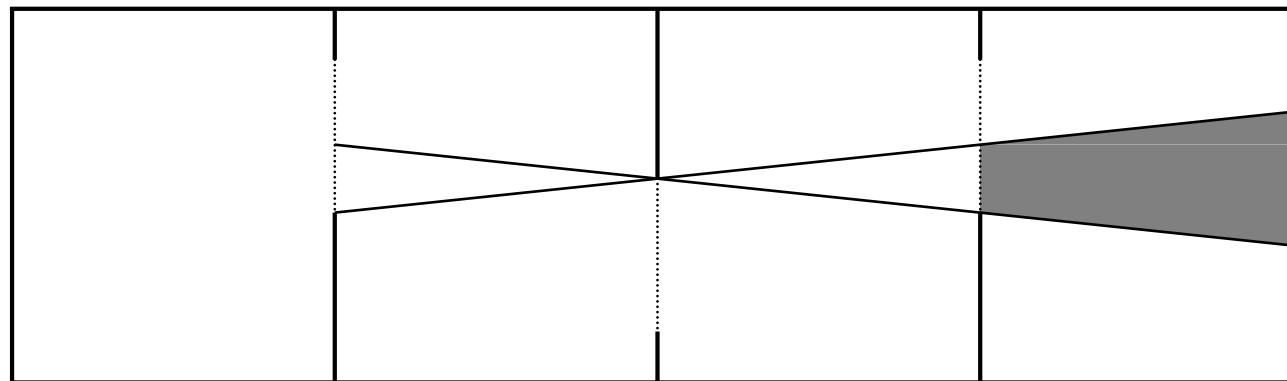
3. Rendering



3.6 Sichtbarkeitsberechnung

- **Zelle-zu-Region**

- Wenn eine Zelle mehrere Portale hat: finde maximale separierende Linien



- Nicht-trivial, ähnliche Probleme gibt es an verschiedenen anderen Stellen der Computergraphik (Schatten-Berechnung, Mesh-Generierung für Radiosity...)

3. Rendering

3.6 Sichtbarkeitsberechnung



- **Eigenschaften PVS:**
 - **Fast die gesamte Erstellung ist preprocess!**
 - Zur run-time muss der PVS-Baum nur durchlaufen werden
 - Display List für jede Zelle kann vorberechnet werden: schnelles rendering!
 - **Die meisten Algorithmen gehen noch weiter als Zelle-zu-Zelle PVS**
 - Sehr konservative Abschätzung – PVS schliesst 90% des Modells aus, 99.6% ist wirklich unsichtbar, bessere Algorithmen kommen auf 98% (Teller 91)
 - **Zelle-zu-Zelle PVS sind gut für dynamische Objekte**
 - Objekt wird mit der Zelle assoziiert, in dem es sich gerade befindet
 - Rendere ein sich bewegendes Objekt nur wenn seine Zelle sichtbar ist

3. Rendering

3.6 Sichtbarkeitsberechnung



■ **PVS Probleme**

- Keine Verwendung des Blickpunktes: auch Dinge, die möglicherweise nicht gesehen werden können, bestehen den Test.
- Nicht gut für dynamische Zellen/Portale
 - “Öffnen einer Tür”
- Pre-processing Zeit kann sehr lang sein!
- Beeinflusst Entwicklungszeit für Spiele – turnaround Zeiten für Änderungen ist lang
- Es gibt Algorithmen, die dies angehen

3. Rendering

3.6 Sichtbarkeitsberechnung



■ Keine Zellen oder Portale?

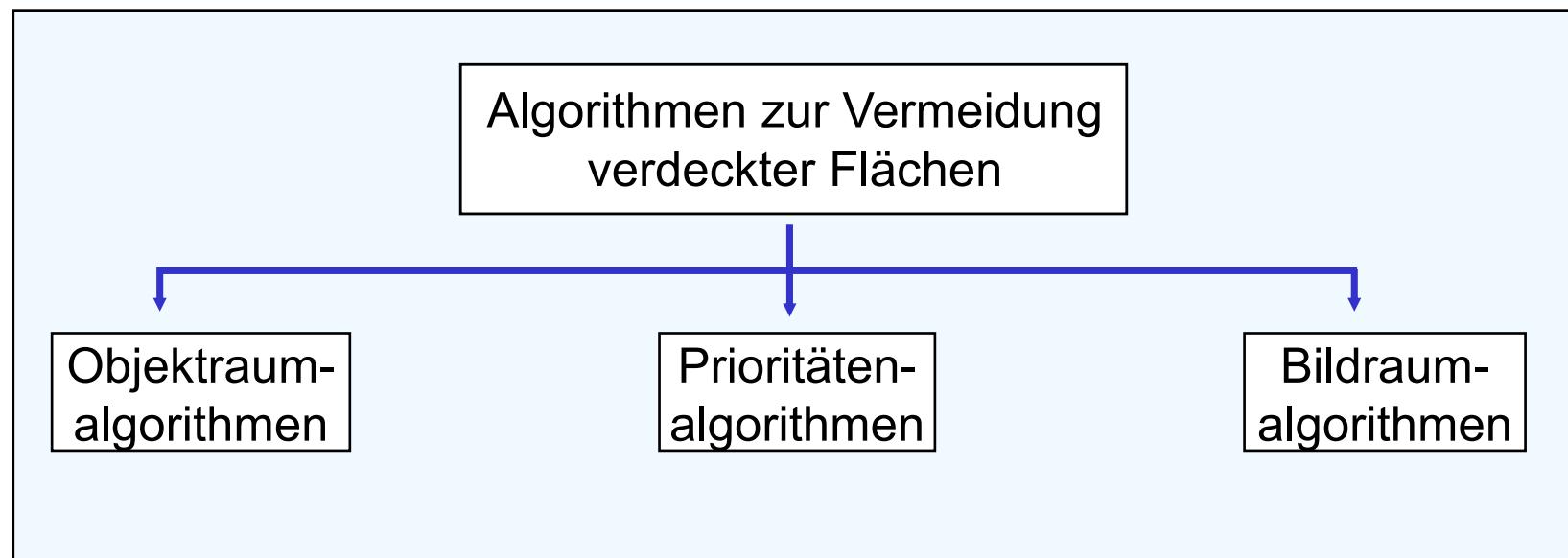
- Viele Szenen haben keine Zellen-Portale-Struktur
- Szenen ohne grosse koplanare Polygone, welche als Blocker oder Wände wirken
 - Beispiel: Wald – man kann nicht durchsehen, aber kein einzelnes Blatt ist verantwortlich
- Was kann man dann tun?
 - Finde *occluders* und nutze sie um Geometrie zu cullen

3. Rendering

3.6 Sichtbarkeitsberechnung



- Nach dem Culling werden die verbliebenen Elemente der 3D-Szene auf Sichtbarkeit getestet. Die hierzu eingesetzten HLHSR-Algorithmen lassen sich in 3 Gruppen klassifizieren:



3. Rendering

3.6 Sichtbarkeitsberechnung



- **Objektraumalgorithmen**

führen die Sichtbarkeitsberechnungen explizit auf Grund der Lagebeziehungen der Objekte durch; typische Beispiele sind Algorithmen zur Vermeidung verdeckter Kanten.

- **Bildraumalgorithmen**

berechnen die Sichtbarkeit pixelweise;

- **Prioritätenalgorithmen**

sortieren die Objekte einer Szene nach bestimmten Kriterien (z.b. Abstand zum Betrachter) und geben die Objekte in Abhängigkeit dieser Sortierung aus.

3. Rendering

3.6 Sichtbarkeitsberechnung



- Ein Vertreter von **Objektraumalgorithmen** ist der **BSP (binary space partition)-Baum**
- BSP-Baum ist eine blickpunkt-unabhängige Datenstruktur
- wird als Preprozess aufgebaut
- ermöglicht schnelle Entscheidung der Sichtbarkeit für konkreten Blickpunkt

3. Rendering

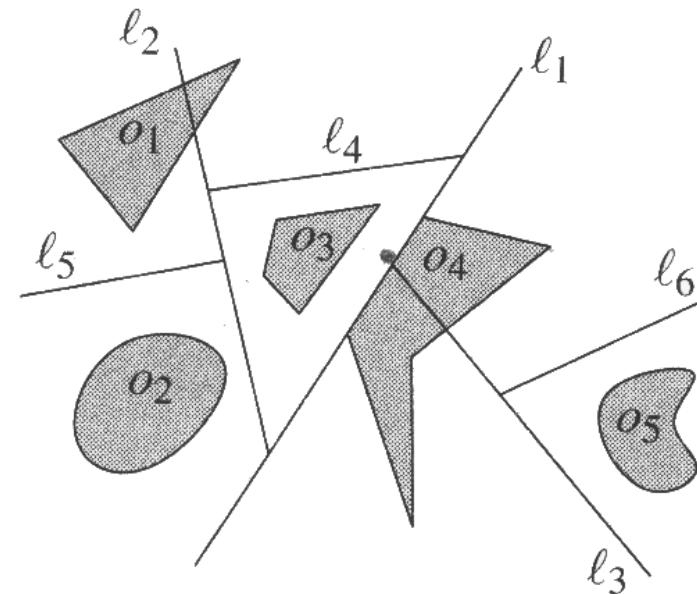
3.6 Sichtbarkeitsberechnung



- **BSP - The Big Picture**

- Annahme: Objekte überlagern sich nicht.

Nutze Ebenen, um den Objektraum rekursiv zu splitten, speichere die Baumstruktur dieser Splittings



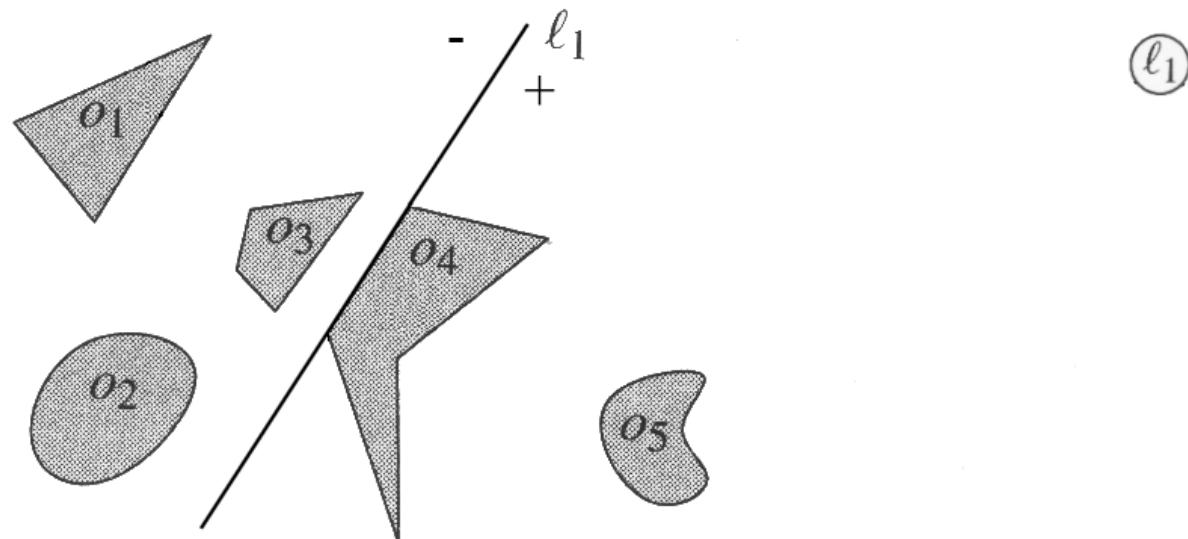
3. Rendering

3.6 Sichtbarkeitsberechnung



- **Auswahl einer Splitting Line**

- Die Auswahl einer Splitting line teilt die Objekte in 3 Klassen auf: auf jeder Seite der Ebene, und in der Ebene.



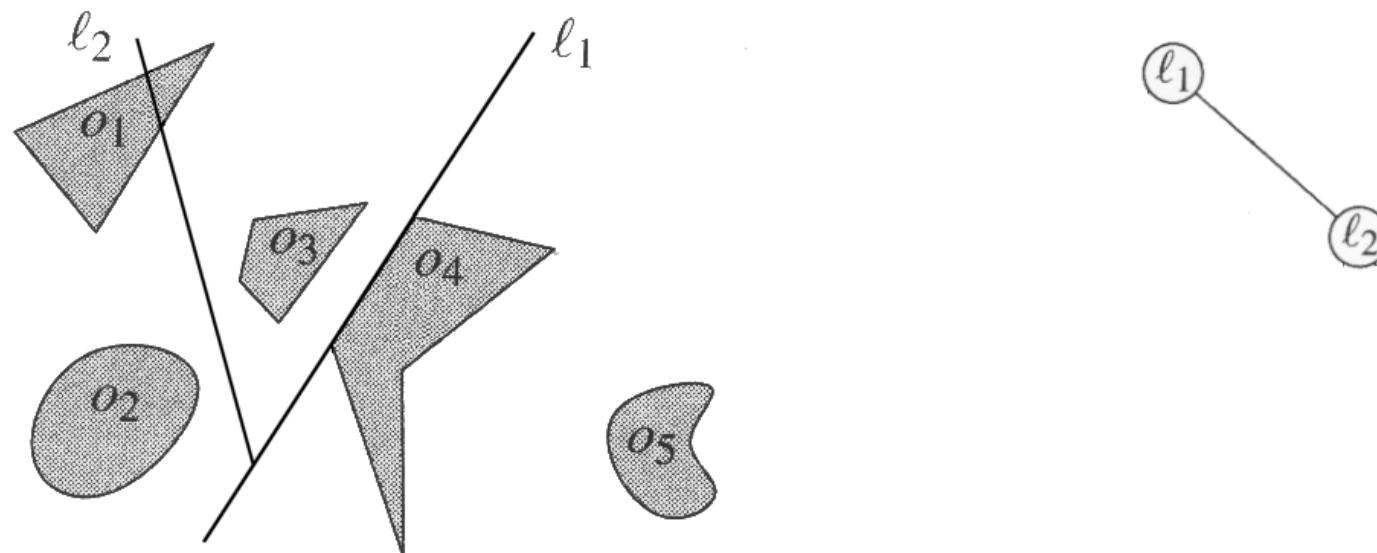
3. Rendering

3.6 Sichtbarkeitsberechnung



■ Mehrere Splitting-Ebenen

- Was tun wenn ein Objekt (z.B. Objekt 1) durch eine Splitting plane geteilt wird?
- Teilen in 2 Objekte!



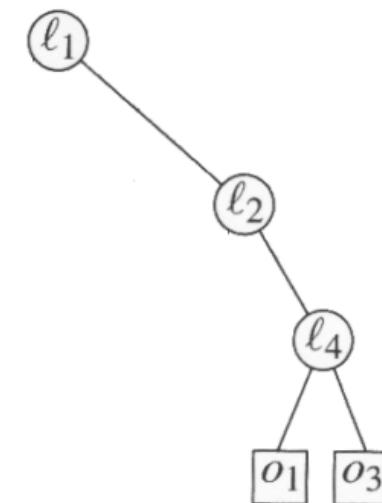
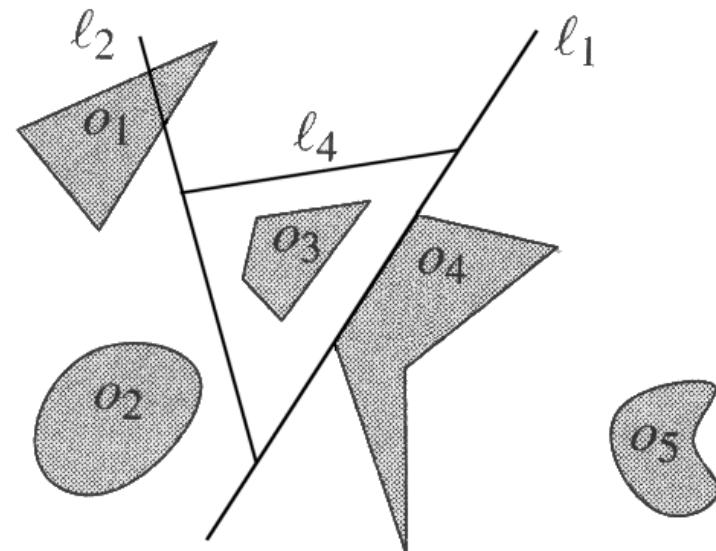
3. Rendering

3.6 Sichtbarkeitsberechnung



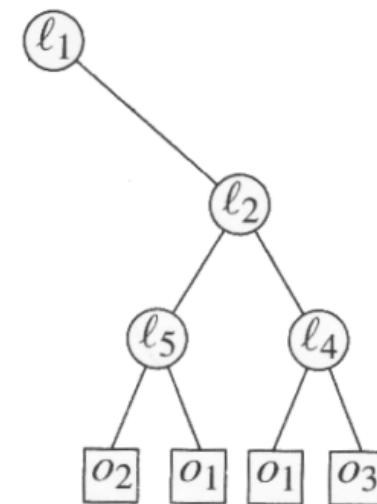
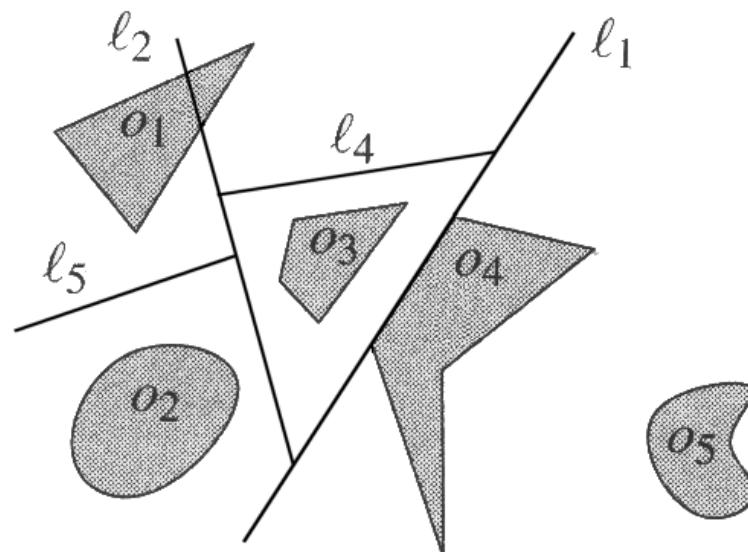
■ Rekursives Splitting

- Wenn wir einen Raum erreichen, der kein oder nur ein Objekt enthält: stoppe splitting



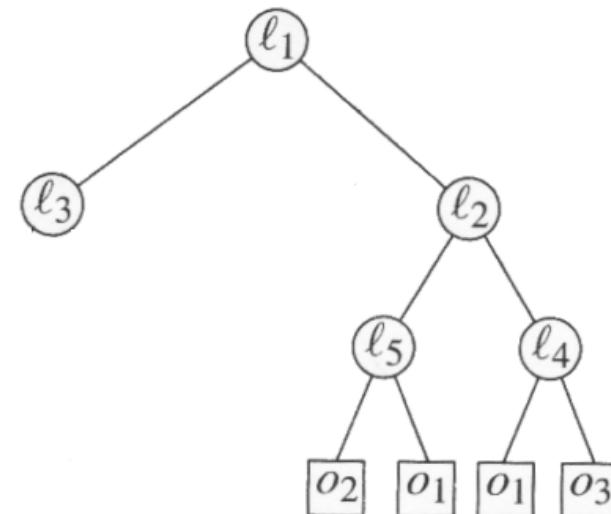
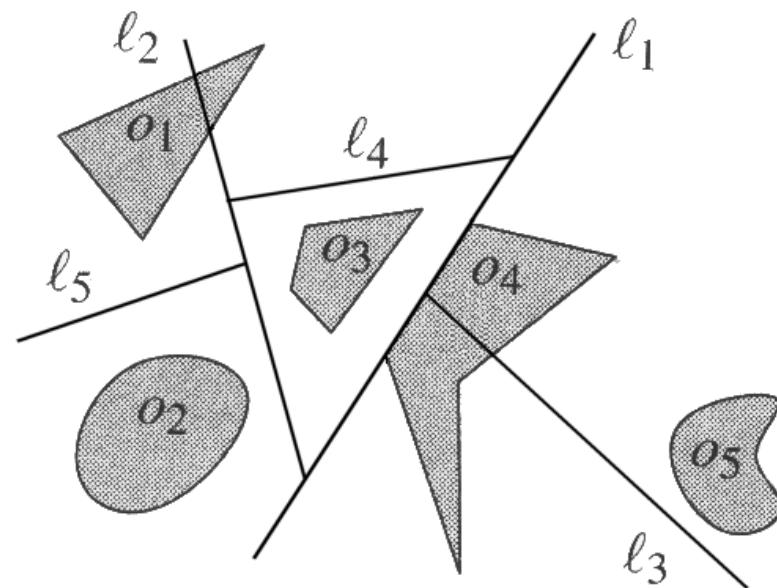
3. Rendering

3.6 Sichtbarkeitsberechnung



3. Rendering

3.6 Sichtbarkeitsberechnung

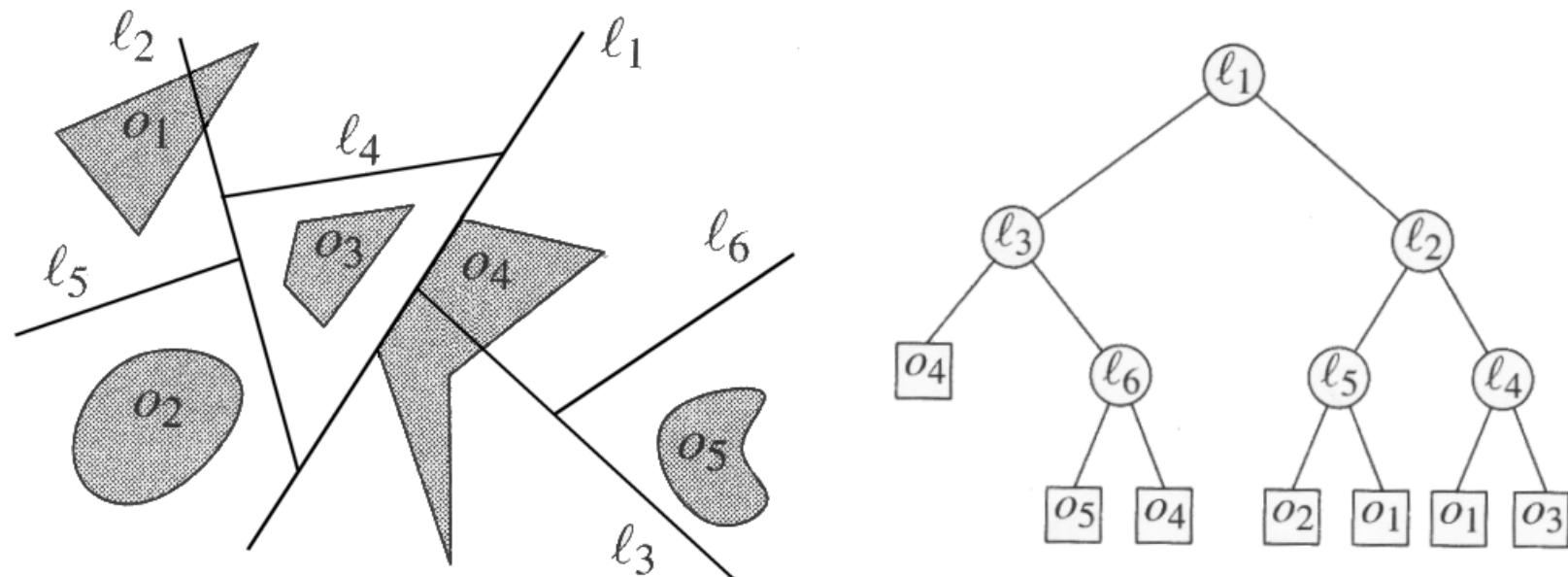


3. Rendering

3.6 Sichtbarkeitsberechnung



- **Beendet**
 - Wenn der Baum vollständig aufgebaut ist, beschreibt jeder root-to-leaf Pfad einen einzelnen konvexen Unterraum



3. Rendering

3.6 Sichtbarkeitsberechnung



■ **Abarbeiten des Baumes**

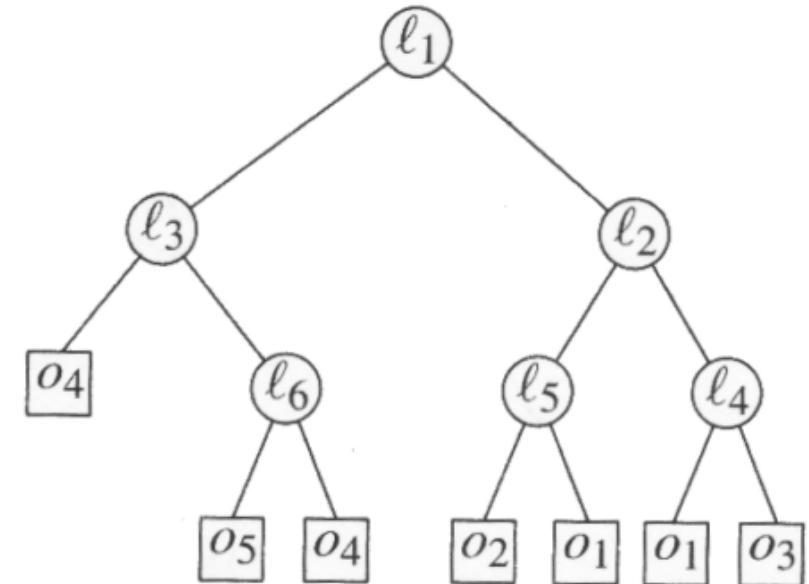
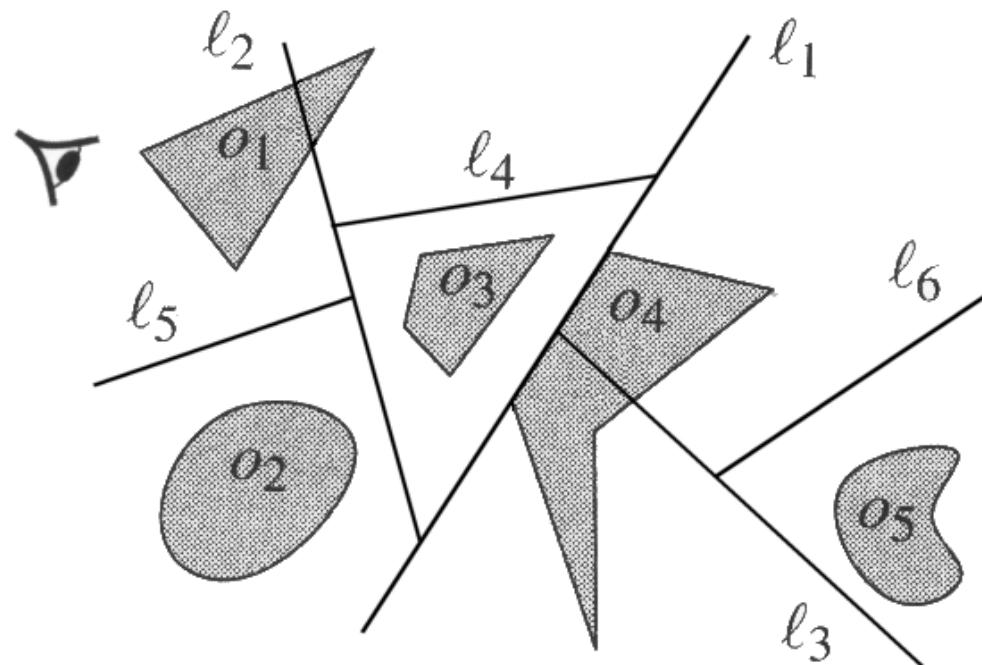
- Platziere Augenpunkt im Baum
- Traverse Baum von der Wurzel:
- Rendere erst alle Objekte, die sich in der Halbebene befinden, in dem der Augenpunkt nicht ist, dann alle Objekte in der cutting plane, dann alles in der Halbebene mit Augenpunkt!

3. Rendering

3.6 Sichtbarkeitsberechnung



- Beispiel:



3. Rendering

3.6 Sichtbarkeitsberechnung



■ **Aufbau eines BSP-Baums**

- Welche cutting planes, welche zuerst?
- Bei Dreiecksnetzen: nur cutting planes die Dreieck enthalten
- Wahl der optimalen Cutting planes ist NP-vollständig!
- Heuristik: zufällige Auswahl der nächsten Cutting plane

3. Rendering

3.6 Sichtbarkeitsberechnung



- Der am häufigsten verwendete **Bildraumalgorithmus** ist der **z-Buffer-Algorithmus**. Dieser arbeitet mit 2 Speichern:
 - Bildspeicher (frame buffer)
enthält die Farbwerte der jeweils sichtbaren Flächen.
 - Tiefenspeicher (z-Buffer)
enthält die z-Werte der jeweils sichtbaren Flächen.

A 10x10 grid representing a Z-buffer for a single polygon. The values are mostly 9 (representing infinity or background), with some values from 1 to 18 highlighted in yellow. The highest value, 18, is at the top right corner. Other values include 16, 17, 15, 16, 13, 14, 15, 16, 10, 11, 12, 13, 14, 6, 7, 8, 9, 10, 11, 12, 13, 4, 5, 6, 7, 8, 9, 10, 11, 12, 3, 4, 5, 6, 7, 8, 9, 11, 12, 1, 2, 0, 1, 2.

9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9

Z-Buffer für Polygon 1

A 10x10 grid representing a Z-buffer for a second polygon. The values are mostly 9 (representing infinity or background). A diagonal line of values from 1 to 18 is highlighted in yellow, representing the vertices of the polygon. The highest value, 18, is at the top right corner of the diagonal.

9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9

Z-Buffer für Polygon 2

A 10x10 grid representing the combined Z-buffer for both polygons. The values are mostly 9 (representing infinity or background). The diagonal line of values from 1 to 18 is highlighted in yellow, representing the vertices of the two polygons. The highest value, 18, is at the top right corner of the diagonal. The bottom row contains values 7, 8, 9, 10, 11, 12, 1, 2, 0, 1, 2.

9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9

Z-Buffer für Polygon 1 + 2

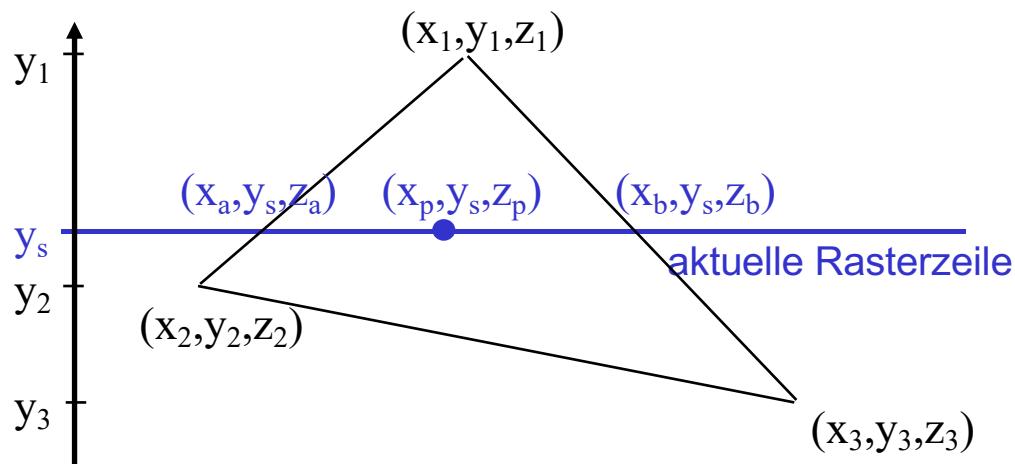
3. Rendering

3.6 Sichtbarkeitsberechnung



▪ Prinzip:

- Objekte durchlaufen nacheinander die Viewing-Pipeline und werden gerastert.
- Für jedes Pixel wird zusätzlich der zugehörige z-Wert berechnet.
- Ein Pixel wird nur genau dann in den Bildspeicher geschrieben, wenn der zugehörige z-Wert größer als der für dieses Pixel im Tiefenspeicher enthaltene z-Wert ist.
- In diesem Fall wird auch der Tiefenspeicher aktualisiert.



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

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

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

3. Rendering

3.6 Sichtbarkeitsberechnung



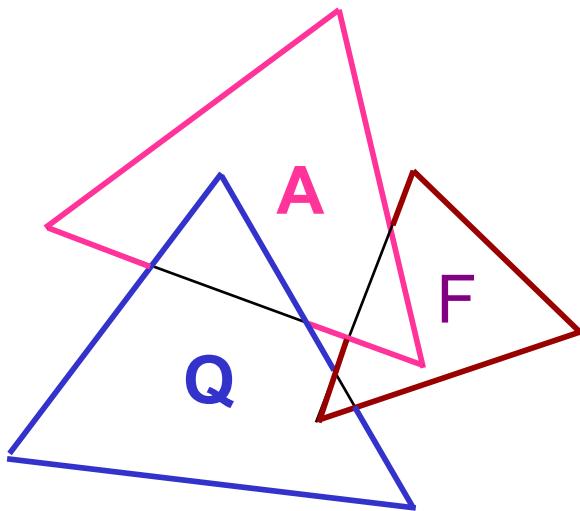
- Ein Verfahren, das mit Prioritäten arbeitet ist der **Tiefensortungsalgorithmus**.
- **Prinzip:**
(Sortierung der Flächen einer Szene (Polygone) in Abhängigkeit vom Abstand zum Betrachter und Ausgabe mit Painter-Algorithmus)
 - Es wird eine Ausgangssortierung entsprechend des minimalen **z** -Werts jeder Fläche hergestellt.
 - Beginnend mit der Fläche, die den kleinsten **z** -Wert aufweist, wird die aktuell bearbeitete Fläche **F** überprüft, ob sie sich mit den Nachfolgerflächen in der Tiefe überlappt.
 - Ist dies nicht der Fall, wird die Fläche **F** als "bearbeitet" markiert und in den Bildspeicher eingetragen.

3. Rendering

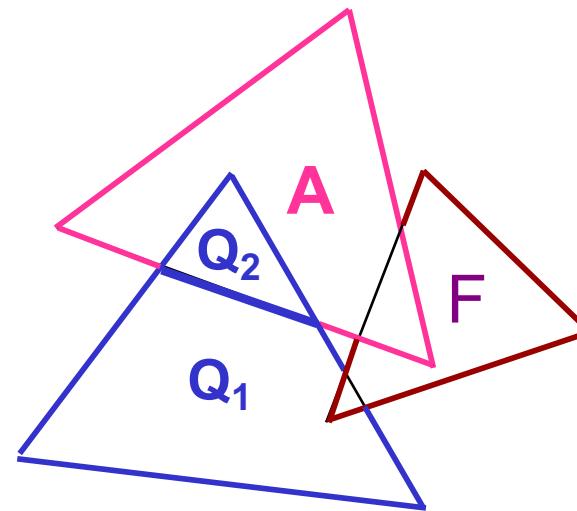
3.6 Sichtbarkeitsberechnung



- Tritt eine Überlappung mit einer Fläche Q auf, so wird geprüft, ob eine Unterteilung der Fläche Q (im Falle eines Konfliktes) oder eine Umsortierung notwendig ist.



Beispiel für Konfliktfall



Lösung des Konfliktfalls

3. Rendering

3.6 Sichtbarkeitsberechnung



- **Zusammenfassung Sichtbarkeit**

- 1. Schnelles und konservatives Ausschliessen der nicht-sichtbaren Objekte/Primitive: visibility culling.**

- Backface culling
- view frustum culling
- occlusion culling
- PVS

- 2. Exakte Berechnung der Sichtbarkeit für übrigbleibende Primitive**

- Objektraumverfahren
 - z.B. BSP
- Bildraumverfahren
 - z.B. z-Buffer
- Prioritätenverfahren