Virtual Reality

Sommer 2012

4 Spezielle Aspekte der Computergraphik

Versionsdatum: 6. Mai 2012

4 Spezielle Aspekte der Computergraphik ...

Fokus: *Computergraphik* zur Erzeugung der Bilder für visuelle Darstellung

VR-spezifische Aspekte der Computergraphik:

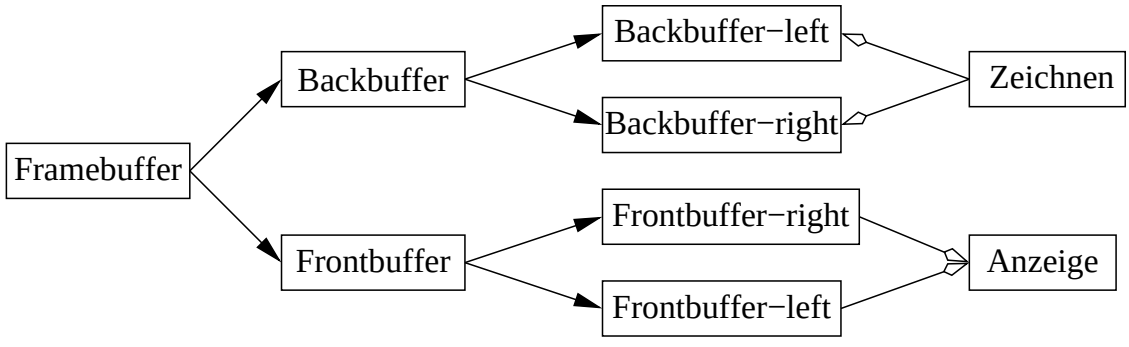
- Stereo-Projektion
  - Erzeugung von Bildern für rechtes und linkes Auge
- Detailierungsgrad (Level of Detail, LOD)
  - Reduktion des Renderingauswandes durch Modellvereinfachung
- Occlusion Culling
  - Reduktion des Renderingauswandes durch Verwerfen unsichtbarer Objekte

4.1 Stereo-Projektion

Ansatz: Rendering und Hardware

Zweimaliges Rendern der Graphik-Pipeline mit leicht veränderten Viewing- oder Perspektiven-Parametern für rechtes /linkes Bild.

Aktiv Stereo: Quadbuffer (Double & Stereo) speichert beide Bilder (synchrone Ausgabekanäle)



Passiv Stereo: Erzeugung der Stereobilder in zwei separaten Graphikspeichern bzw. Graphikkarten (asynochrone Ausgabekanäle)

4.1.1 Erzeugung von Stereobildern

Erinnerung: Tiefenwahrnehmung

Disparität: Abstand eines projizierten Punktes bzgl. korresp. Punkte (Retina)

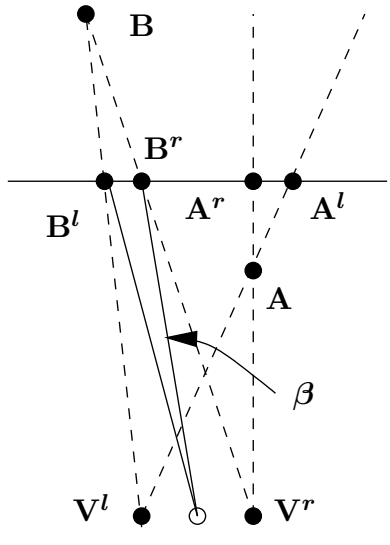
Parallaxe: Abweichung auf der Fokusebene (=Projektionsebene)

Null-Parallaxe: Punkte auf der Fokusebene (*zero parallax, ZP)* erzeugen identische Projektionen

Negative Parallaxe: Punkte $\mathbf{A}$ **vor** der Fokusebene liefern Projektionen $\mathbf{A}^r, \mathbf{A}^l$ mit *negativem* horizontalen Abstand

Positive Parallaxe: Punkte $\mathbf{B}$ **hinter** der Fokusebene liefern Projektionen $\mathbf{B}^r, \mathbf{B}^l$ mit *positivem* horizontalen Abstand

Parallaxenwinkel β: Winkel zwischen rechtem/linken Punkt auf Projektionsebene



Erinnerung: Viewing Transformation

Aufgabe: Abbildung der Objekte aus Welt- in Beobachter-Koordinaten

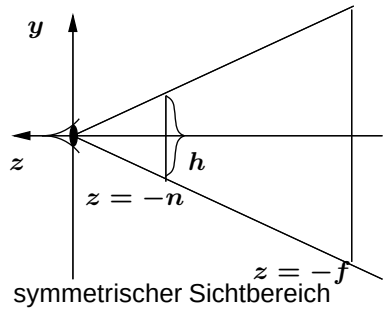
Steuergrössen `gluLookAt`: *Viewpunkt* $\mathbf{V}$, *Lookat-Punkt* $\mathbf{L}$ und *up-Vektor* $\vec{\mathbf{u}}$

Erinnerung: Perspektivische Transformation

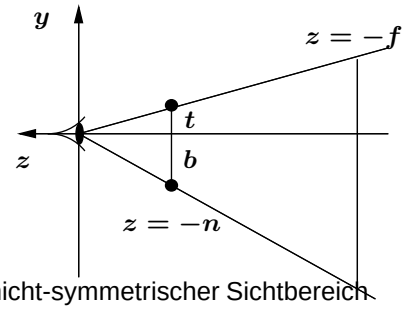
Aufgabe: Festlegung des Sichtbereiches/Viewing-Frustum & Transformation in $[-1,1]^3$

Bislang: Sichtbereich symmetrisch zur z-Achse (=optische Achse)

Nichtsymmetrischer Sichtbereich: l, r, b, t (left, right, bottom, top) auf naher Clippingebene



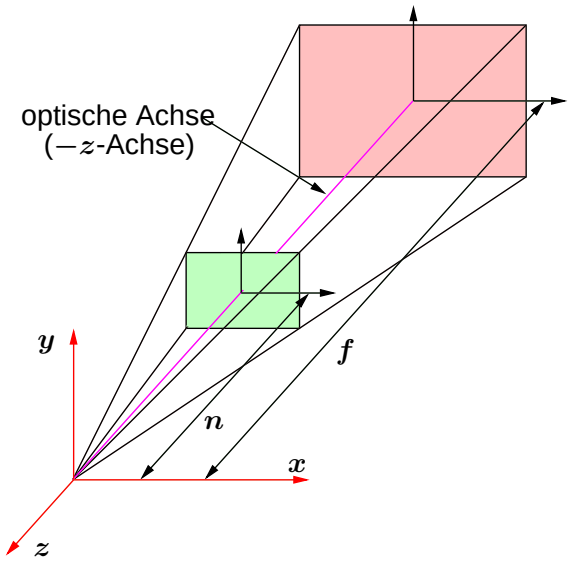
symmetrischer Sichtbereich

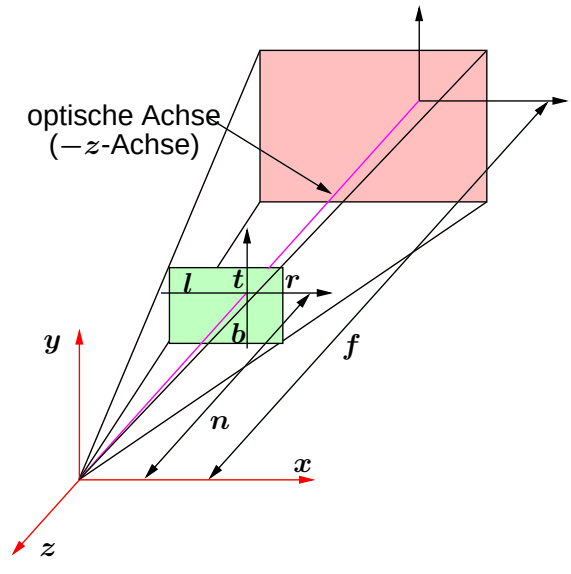


nicht-symmetrischer Sichtbereich

4.1.1 Erzeugung von Stereobildern ...

Perspektivische Projektion



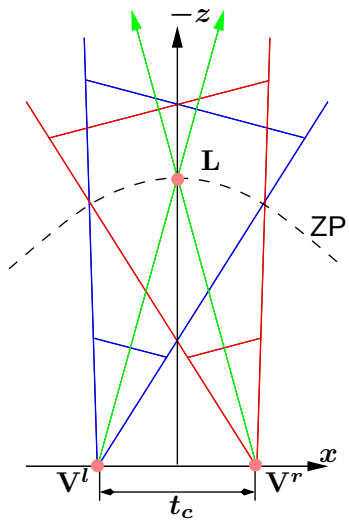


Definition:

Augpunkte $\mathbf{V}^l, \mathbf{V}^r$ für das li./re. Bild; Beobachterpunkt („twin center“): $\mathbf{V} = \frac{1}{2}\left(\mathbf{V}^l + \mathbf{V}^r\right)$

Interaxiale bzw. interpupilare Distanz t_c**:** Abstand $\left\|\mathbf{V}^l - \mathbf{V}^r\right\|$

Sichtachsen: Ergeben sich aus $\mathbf{L}^l - \mathbf{V}^l$ bzw. $\mathbf{L}^r - \mathbf{V}^r$



Fokuspunkt-Ansatz

Idee: Stereobilder durch Verdrehen des Augpunktes mit gemeinsamem LookAt-Punkt $\mathbf{L}$

Vorteil: Einfache Umsetzung (nur $\mathbf{V}$ anpassen)

Nachteil: Die ZP-Punkte liegen nicht auf einer Ebene (Verzerrung am Bildrand)

4.1.1 Erzeugung von Stereobildern ...

Parallele optische Achsen

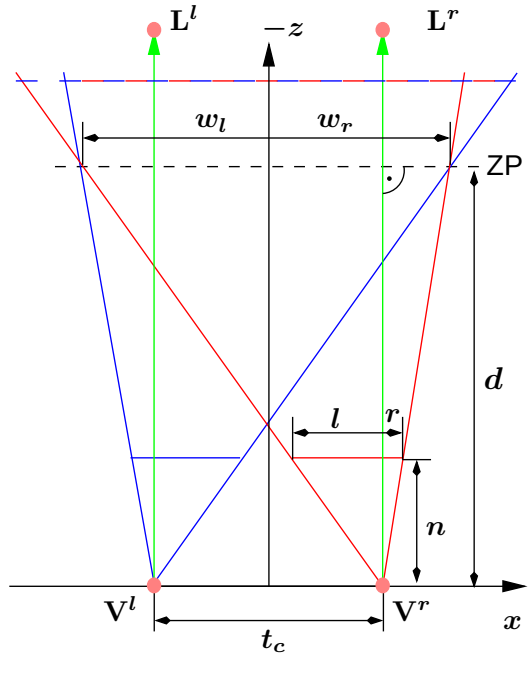
Idee: Parallele Sichtachsen mit verschobenem Viewing-Frustum

Vorteil: ZP-Punkte in einer Ebene

Gegeben sind:

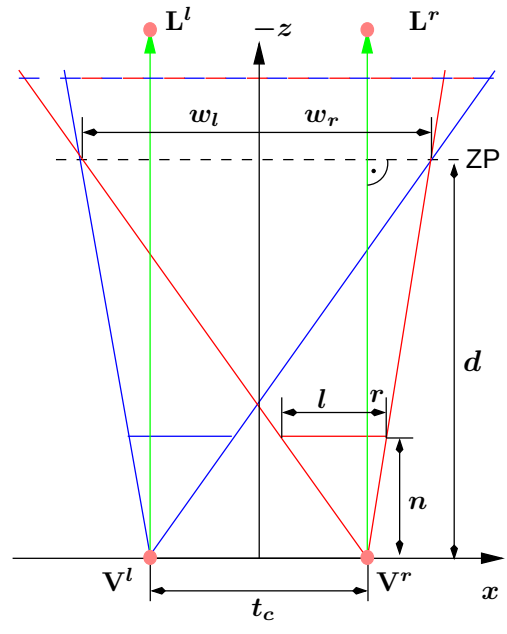
Display-Parameter:
- ❍ Abstand zum Screen d
- ❍ horizontale und vertikale Position der Screenränder w_l, w_r, h_b, h_t (vorzeigenbehaftet)

View-Parameter: $n, f, t_c, \mathbf{V}$



Parallele optische Achsen (Forts.)

Ermittlung von l, r, z.B. für rechtes Auge ($x-z$-Ebene):



rechter Rand: $\underbrace{\mathbf{V} + \begin{pmatrix} \frac{t_c}{2} \\ 0 \end{pmatrix}}_{=\mathbf{V}^r} + \alpha_r \begin{pmatrix} w_r - \frac{t_c}{2} \\ -d \end{pmatrix}$

$$\alpha_r = \frac{n}{d} \Rightarrow r = \frac{n(w_r - \frac{t_c}{2})}{d}$$

linker Rand: $\mathbf{V}^r + \alpha_l \begin{pmatrix} w_l - \frac{t_c}{2} \\ -d \end{pmatrix}$

$$\alpha_l = \frac{n}{d} \Rightarrow l = \frac{n(w_l - \frac{t_c}{2})}{d}$$

analog: $b = \frac{nw_b}{d}, \quad t = \frac{nw_t}{d}$

4.2 Level of Detail (LOD)

Bemerkung:

Herausforderung: VR stellt extreme Ansprüche an die Graphik-Hardware

Beispiel: Zwei Stereo-Screens $\Rightarrow$ 4 Bilder/Frame $\Rightarrow$ 25% Szenenkomplexität!

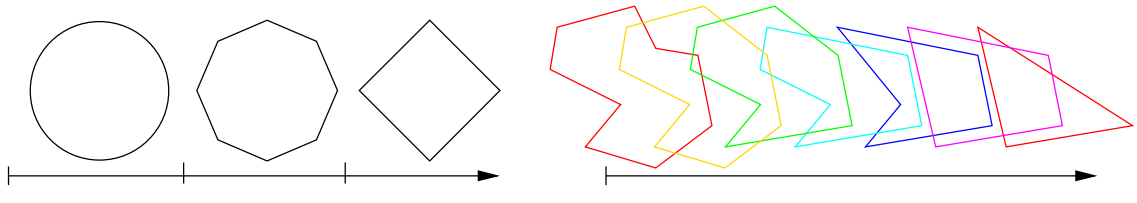
Optimierung-Ansätze:

LOD: Unterschiedliche Detailierungsgrade entspr. Abstand zum Beobachter

Culling: Verwerfen von (teilweise) unsichtbaren Objekten

Mesh-Reduktion: siehe späteres Kapitel

Beispiel Mesh-Reduktion



LOD-Techniken

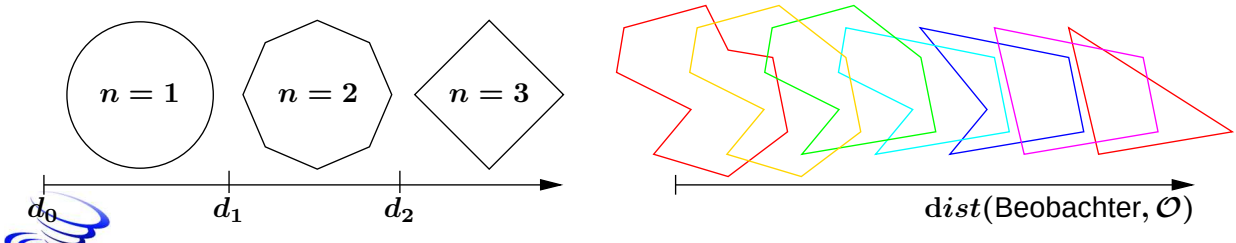
Unterscheide zwei Arten von LOD-Ansätzen für ein Objekt $\mathcal{O}$:

Diskreter LOD: n feste Detailgrade mit Entfernungen
$d_0 = 0 < d_1 < \ldots < d_{n-1} < d_n = \infty$

Verwende Detailgrades $g \Leftrightarrow dist(\text{Beobachter}, \mathcal{O}) \in [d_{g-1}, d_g[$

- ❍ Relativ einfache Umsetzung
- ❍ erhöhtes Datenvolumen bei Übertragung der Modelle
- ❍ „Plop-Effekt“ beim Umschalten meist sichtbar

Kontinuierlicher LOD: Einfügen bzw. Entfernen von Polygoneckpunkten entspr. des Abstandes zum Beobachter (siehe Kapitel Mesh-Reduktion)



4.3 Culling-Techniken

Ziel: Aufwandsoptimierung durch Reduktion der Datenmenge beim Durchlaufen der Graphik-Pipe.

Ansatz: Verwerfen von Objekten, die (mit hoher Wahrscheinlichkeit) nicht sichtbar sind

Culling-Techniken: Man unterscheidet u.a. folgende Techniken:

Backface-Culling: Zeichne nur Polygone, die zum Beobachter weisen

Viewfrustum-Culling: Entferne Objekte außerhalb des Viewing-Frustums

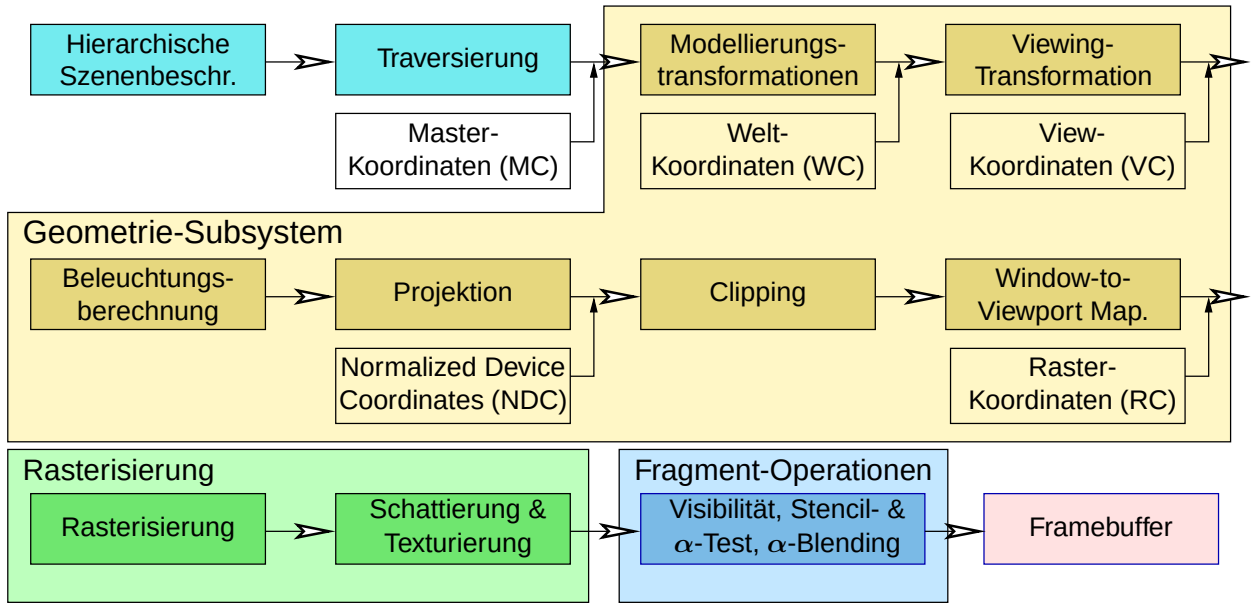
Portal-Culling: Reduziere Teilszenen auf Texturen (z.B. Blick ins Nachbarzimmer)

Occlusion-Culling: Entferne Objekte, die von anderen Objekten verdeckt sind

Grundsatz:

Culling-Aufwand muß kleiner sein, als der eingesparte Renderaufwand!

Erinnerung: Standard Graphik-Pipeline



- Clipping entspricht einem Per-Triangle Culling auf unterer Ebene

4.3.1 Backface-Culling

Erinnerung: Außenseiten von Polygonen durch Orientierung der Punkte definiert

Culling-Ansatz: bei geschlossenen Körpern nur Front-facing Polygone rasterisieren

OpenGL**:** Backface-Culling ist fester Bestandteil von OpenGL.

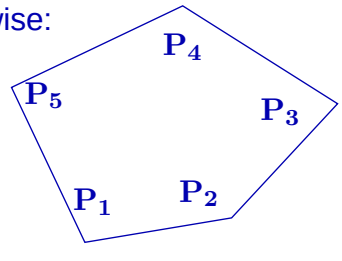
Relevante Funktionen:

`glCullFace()`: legt fest, ob Front- oder Backfaces gecullt werden

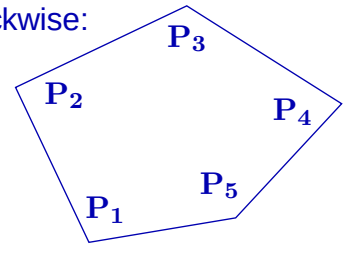
`glFrontFace()`: Orientierung Front-Faces: counter-clockwise (`CCW`), clockwise (`CW`))

`glEnable(GL_CULL_FACE)` aktiviert das Culling.

Counterclockwise:



Clockwise:



4.3.2 Viewfrustum-Culling

Idee: Entfernen ganzer Objekte vor der Projektions-Transformation

Ansatz: Objekte werden von *Bounding-Spheres* umschlossen

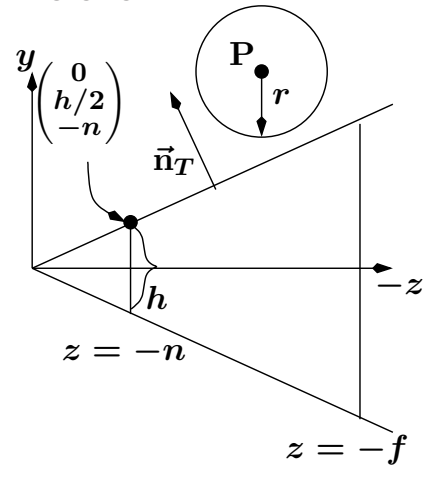
Kugel ausserhalb Viewing-Frustum $\Longrightarrow$ Objekt unsichtbar

Berechnung: Kugel mit Mitte $\mathbf{P}$ und Radius r. Prüfung gegen obere Ebene:

$$\text{Normale: } \vec{\mathbf{n}}_T = \begin{pmatrix}1\\0\\0\end{pmatrix} \times \begin{pmatrix}0\\h/2\\-n\end{pmatrix} = \begin{pmatrix}0\\n\\h/2\end{pmatrix}, \quad \hat{\mathbf{n}}_T = \frac{\vec{\mathbf{n}}}{\|\vec{\mathbf{n}}\|}$$

$$\text{Hesse Normalform: } E_T : \left(\hat{\mathbf{n}} \cdot \begin{pmatrix}x\\y\\z\end{pmatrix}\right) = 0$$

Kugel oberhalb: $(\hat{\mathbf{n}} \cdot \mathbf{P}) > r$



4.3.3 Portal Culling

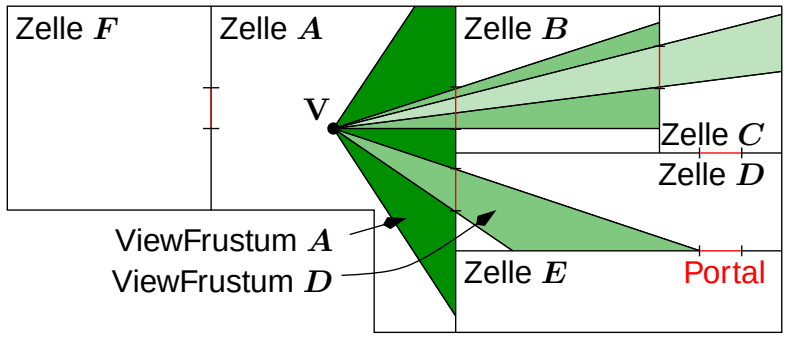
Ausgangslage: Beobachter befindet sich in einem (komplexen) ***architektonischen Modell*** (Gebäude)

Beobachtung: Nur der kleinste Teil der Gesamtgeometrie sichtbar

Idee: ❍ unterteile Gebäude entsprechend der Räume in Zellen

- ❍ Beobachter befindet sich in einer Zelle $\Rightarrow$ Rendering der Objekt in Zelle mit Viewing-Frustum Culling
- ❍ betrachte (rendere) benachbarte Räume (auch Aussenwelt) durch ***Portale*** (Fenster, Türen)

Anpassung der Viewing-Frusten für Portale & Viewfrustum-Culling



Portal Culling Algorithmus

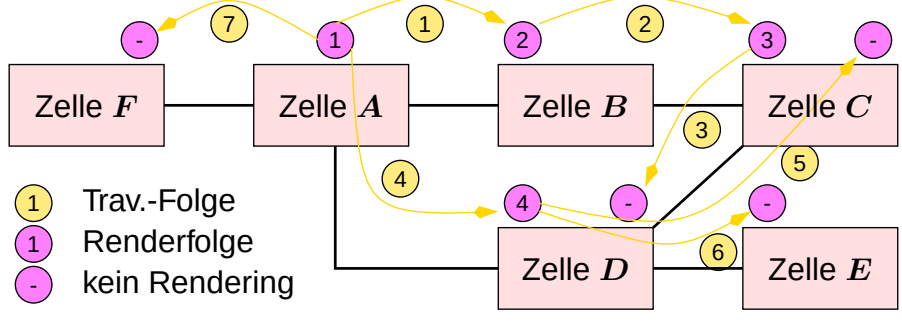
Ansatz: Zellen (Knoten) und Portale (Kanten) bilden einen Graphen, der, beginnend von der Zelle des Beobachters traversiert wird.

Algorithmus:

1. Lokalisation der Zelle Z, in der sich Beobachter befindet
2. Rendern der Objekte der Zelle Z (View-Frustum Culling)
3. Foreach Nachbarzelle N von Z mit Portal zu Z:
   - 3.1. projiziere Portal P auf Bildebene
   - 3.2. falls Portal auf Bildebene sichtbar (auch teilweise und über Portal-Sequenzen) $\Rightarrow$ Portal P'
     - 3.2.1. bestimme korrigiertes View-Frustum für P'
     - 3.2.2. rendere Objekte der Zelle N mit Viewfrustum-Culling
     - 3.2.3. gehe zu Schritt (3) (rekursive Verarbeitung der Portale)

Beispiel: Portal Culling Algorithmus

Graph zu dem Beispiel von der vorhergehenden Folie



Beachte: Knoten werden i.d.R. mehrfach betrachtet

Bemerkung: Aussenbereiche

- Unterteilung des Aussenbereichs in Zellen & Zuordnung der Objekte, z.B. an natürlichen Grenzen (Gebäuden etc.)
- verwende kombiniertes Portal- und Frustumculling
- zusätzlich: Verarbeite Zellen Front-to-Back & Occlusion Culling der Zelle

4.3.4 Occlusion Culling

Bislang: Entfernen von Objekten ausserhalb des Sichtbereiches

Ansatz: Erkennen von verdeckten Objekten innerhalb des Sichtbereiches

455 874 polygons
(4.3%) visible

10 154 292 polygons
(95.7%) occluded

Z-Buffer-Algorithmus löst Sichtbarkeitsproblem für beliebige 3D-Szenen auf Pixelebene

Problem Z-Buffer: Häufiges Prüfen und ggf. Überzeichnen auf Pixelebene

Depth Complexity (Map) = Anzahl der Überzeichnungen pro Pixel

4.3.4 Occlusion Culling ...

Grundsätzliche Betrachtungen

Point-based Occlusion Culling: Bewertung der Sichtbarkeit anhand aktueller Beobachterposition $\mathbf{V}$

Cell-based Occlusion Culling: Sichtbarkeit innerhalb einer „Bewegungszelle“ $\mathcal{R}$; i.a. rechenaufwendiger

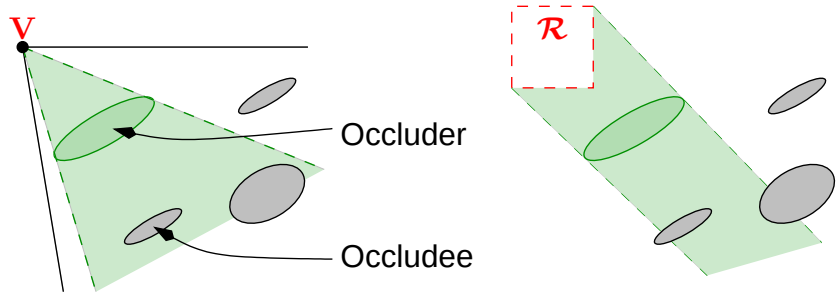


Image Space Occlusion Culling: Entscheidung der Sichtbarkeit in 2D-Bildebene (approximativ auf Pixel-Gitter)

Object Space Occlusion Culling: Entscheidung der Sichtbarkeit im 3D-Objektraum (exakt)

Algorithmus: Occlusion Culling allgemein

$\mathcal{G}$: Menge der Objekte in der Szene

$\mathcal{P}$: Menge potentieller Occluder (initial leer)

$\mathcal{O}$: Occluder-Repräsentation (initial leer)

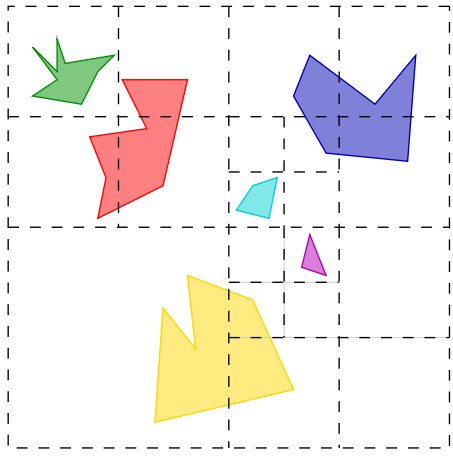
```
foreach g in G do {          // object g in scene objects G
  if ( isOccluded(g, O) ) continue;
  else {
    render(g);
    P.add(g);
    if ( largeEnough(P) ) { // not always update O !
      updateOccluderRepresentation(O , P); // rather expensive
      P.clear();            // remove pot. occluder
    }
  }
}
```

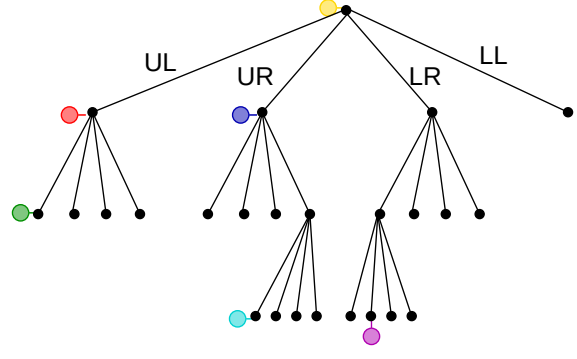

4.3.4 Occlusion Culling ...

Hierarchical Z-Buffering, HZB (Greene '93)

Datenstrukturen: 1. Octree zur Verwaltung der Szenen-Objekte
2. hierarchischer Z-Buffer

Octree: Adaptive räumliche Datenstruktur





Octree Erzeugung

Ausgangslage: Bounding-Box der gesamten Szene U^0

Octree Unterteilung am Box-Mittelpunkt in *acht gleichgroße Teilbereiche*:

$$\text{initial: } U^0 = \dot{\bigcup}_{k=1}^{8} U_k^1 \qquad \text{rekursiv: } U_I^i = \dot{\bigcup}_{k=1}^{8} U_{(I,k)}^{i+1}$$

mit Multi-Index $I = (k_1, \ldots, k_i),\ k_j \in \{1, \ldots, 8\}$

Abbruchkriterium: Zwei gängige Alternativen

1. Teilbereich enthält (auch teilweise) max. eine Obergrenze n_{max} von Objekten
2. max. Unterteilungstiefe erreicht

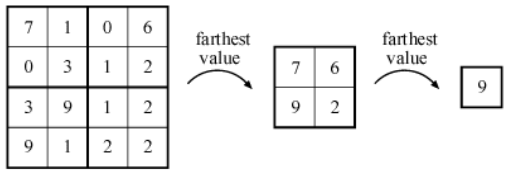
Objekt-Zuordnung: Knoten/Blättern bilden kleinstmögliche Bounding Box für zugeordnete Objekte in der Hierarchie

Beachte: Aufwand für die Erstellung des Octree $\Rightarrow$ statische Szenen

Hierarchischer z-Buffer/z-Pyramide

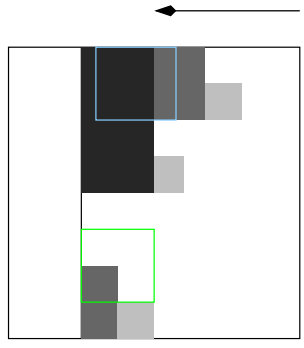
z-Pyramide: ❍ Rekursive Vergröberung des akt. Depth-Buffer mit Fraktor **2**

❍ jeweils größter Tiefenwert wird übernommen



Rendering:

1. Bestimme Octree-Zelle für Beobachter
2. Verarbeite Octree-Node (Init: Root-Node)
   - 2.1. teste Node-Verdeckung in z-Pyramide
   - 2.2. falls nicht verdeckt
     - 2.2.1. rendere zugeordnete Objekte
     - 2.2.2. update z-Pyramide
     - 2.2.3. gehe zu (2) für Child-Nodes („front-to-back“)



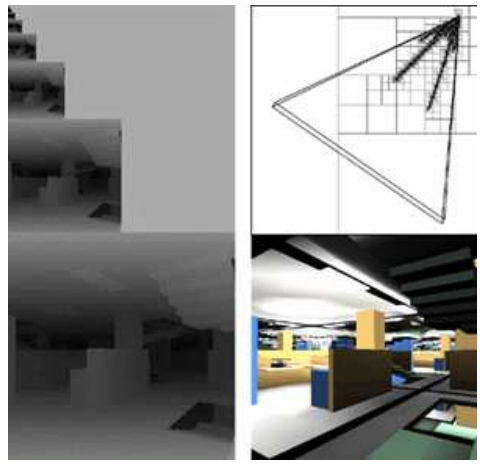
Hierarchischer z-Buffer/Node-Verdeckungstest

Umsetzung der Node-Verdeckung auf Basis der Polygone der AABB

```
bool covered(Polygon p, int l,Pixel px) {
  d  = closest distance of p in px in level l
  if ( d > zPyramid(l, px) ) return true;
  else return false;
}

foreach polygon p of AABB {
  px = polygon location
  l  = coarsest level in zPyramid covering p

  bool visible = true;
  do {
    if ( covered(p, l, px) ) visible = false;
    else {
      visible =
        ! covered(p->subdiv(LL), l+1, px) ||
        ! covered(p->subdiv(LR), l+1, px) ||
        ! covered(p->subdiv(UL), l+1, px) ||
        ! covered(p->subdiv(UR), l+1, px);
  } while ( visible && ++l < MAX_LEVEL );
}
```



Beispielszene mit z-Buffer und Octree

4.3.4 Occlusion Culling ...

Hardware basiertes Occlusion Query

Ansatz: Graphik-Hardware gibt Information,

- ob spezifische Geometrie den Depth-Test teilweise passiert hat (boolsch)
- oder wieviele Pixel der Geometrie den Depth-Test passiert haben

Beispiel: HP OpenGL-Extension `HP_occlusion_test`

- ermittelt, ob Geomtrie (teilweise) Depth-Test passiert hat (boolean)
- Stop-and-Wait Problem: Nach jedem Test muß das Ergebnis abgeholt werden

Beispiel: OpenGL-Occlusion Query Funktionen (urspr. von NVIDIA)

- ermittelt, ob Anzahl der Pixel, die Depth-Test passiert haben
- mehrere Occlusion Tests nacheinander möglich (asynchron)

Algorithmus: OpenGL Occlusion Query

```
GLuint    queryID[N];

glGenQueries(N, queryID); // generate N occl.-IDs
...            // render major occluders
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE); // no color update
glDepthMask(GL_FALSE);                               // no depth update
for (i = 0; i < N; i++) {
    glBeginQuery(queryID[i]);
    // render bounding box for object i
    glEndQuery();
}
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);     // activate update
glDepthMask(GL_TRUE);                                // activate update
// do some stuff
for (i = 0; i < N; i++) {
    do {
        glGetQueryObjectuiv(queryID[i], GL_QUERY_RESULT_AVAILABLE,
                            &available);
    } while ( !available );
    glGetQueryObjectuiv(queryID[i], GL_QUERY_RESULT, &pixelCount);
    if (pixelCount > EPSILON)
        // render object i
}
```

Uniform Hardware Occlusion Query (Hillesland '02)

Gegeben: Statische Szene bestehend aus Polygon-Modellen

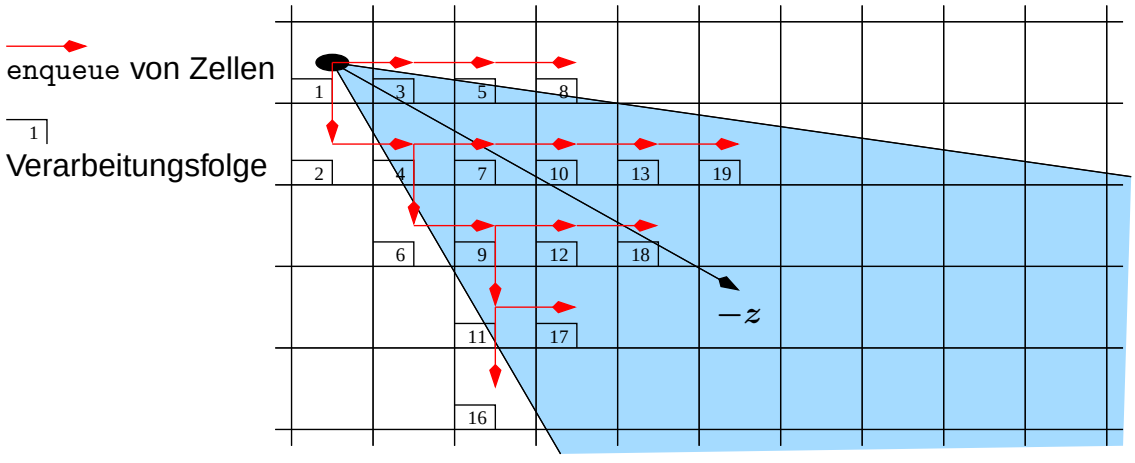
Vorverarbeitung: Dreiecke in reguläres Gitter einordnen (Zellen $\mathcal{Z}$ mit Zentren $\mathcal{C}$ in View-Koordinaten)

Algorithmus: Front-to-Back Grid-Rasterisierung mittels ***Priority-Queue*** $\mathcal{Q}$
Sortier-Kriterium für Zellen in $\mathcal{Q}$ ist $\mathcal{C}_z$ (absteigend)

1. Identifiziere Zelle $\mathcal{Z}$ der Kamera, füge $\mathcal{Z}$ in $\mathcal{Q}$ ein
2. Solang $\mathcal{Q}$ nicht leer
   - 2.1. Hole oberstes Element: $\mathcal{Z} = \mathcal{Q}$.`dequeue()`
   - 2.2. HW Occlusion Query von $\mathcal{Z}$
     - 2.2.1. rasterisiere Zelle $\mathcal{Z}$ (Würfel) (kein Color- & Depthbuffer Update)
     - 2.2.2. HW-Query-Ergebnis $\geq$ Schwellwert $\Rightarrow$ rendere Dreiecke in $\mathcal{Z}$
   - 2.3. Markiere $\mathcal{Z}$ als `visited`
   - 2.4. Füge Nachbarzellen $\mathcal{Z}'$ von $\mathcal{Z}$ in $\mathcal{Q}$ ein ($\mathcal{Q}$.`enqueue(`$\mathcal{Z}'$`)`), sofern
     - ❍ $\mathcal{Z}'$ ist noch unbesucht und Viewing-Frustum Culling negativ
     - ❍ „in Blickrichtung“ $\Leftrightarrow (\mathcal{C}' - \mathcal{C})_z \leq 0$

Uniform Hardware Occlusion Query (Hillesland ’02) (Forts.)

Beispiel:



Bemerkung: ❍ Der Algorithmus kann auch mehrere Zellen parallel auf Verdeckung prüfen (siehe Occlusion Queries)

❍ Das Vorgehen entspricht einer ***breadth first*** Traversierung

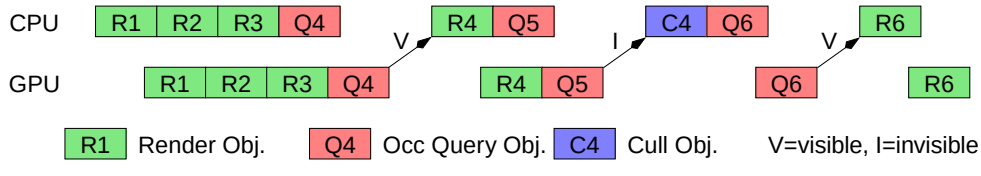
4.3.4 Occlusion Culling ...

Coherent Hierarchical Culling (CHC) (Bittner ’04)

Zielsetzung: Verbesserung bzgl. folgender Punkte

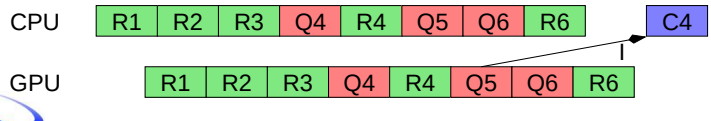
❍ Hierarchische Struktur zur adaptiven Anpassung an (statische!) Szene

❍ Vermeidung von Warten der CPU und Unterauslastung der GPU unter Nutzung ***zeitlicher Kohärenz*** aufgrund kontinuierlicher Beobachterbewegung



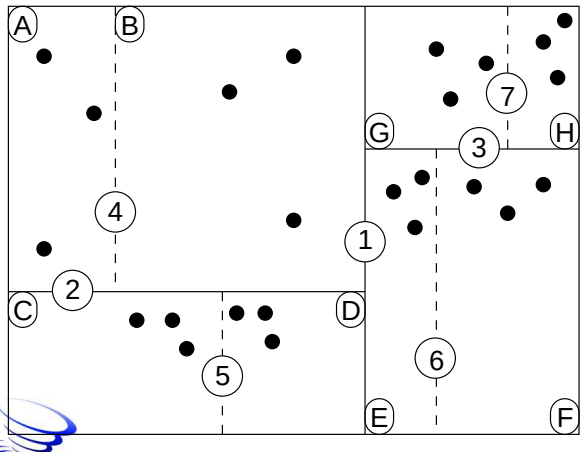
Ansatz: Vermeidung von CPU-Warten und GPU-Unterauslastung:

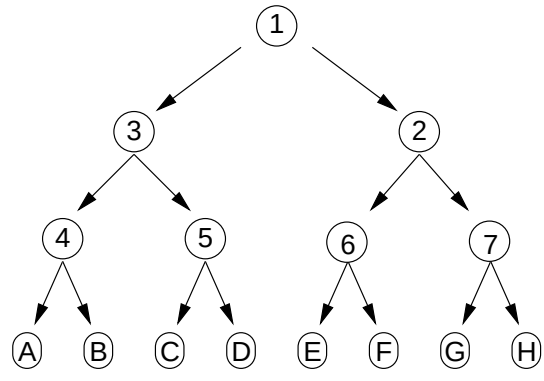
❍ direktes Rendern von im letzten Frame sichtbaren Objekten (hier Objekt 4 und 6)



CHC: Raumunterteilung mit kd-Baum

- achsen-parallele Quader durch Unterteilung entlang einer Koord.-Achse
- Unterteilung halbiert Anzahl der Objekt
- Nächste Unterteilung-Achse ist längste Quaderkante
- Es entsteht ein („nahezu“) ausgeglichener Binärbaum, Objekte werden in Blättern gespeichert





4.3.4 Occlusion Culling ...

CHC: Nutzung der zeitlichen Kohärenz

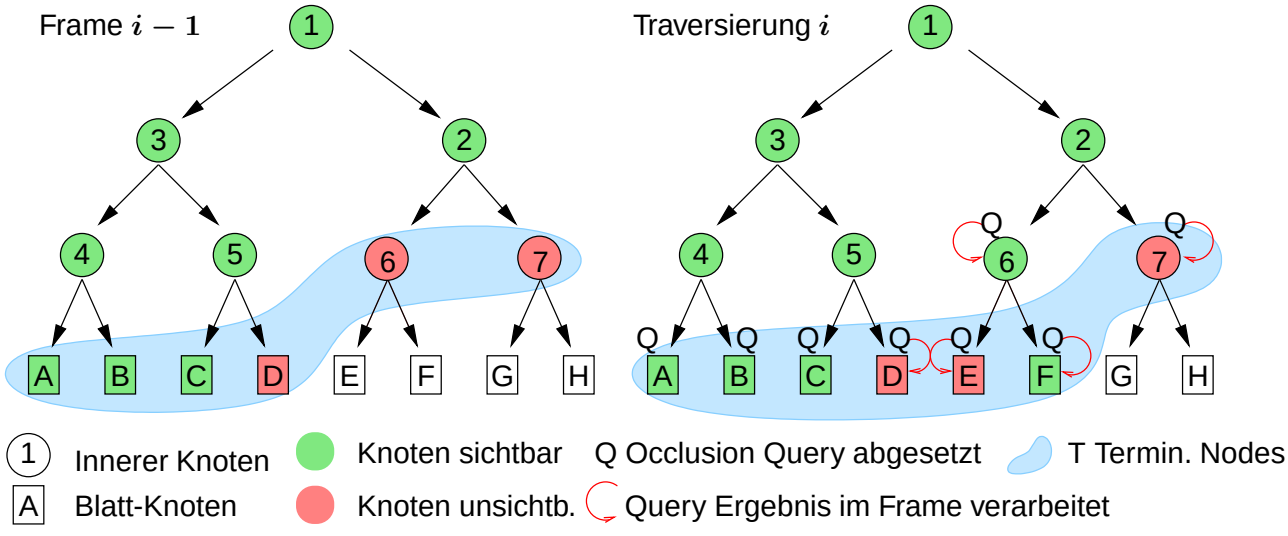
Gegeben: kd-Baum des letzten Frames $i-1$ mit

- ***open nodes*** $\mathcal{O}_{i-1}$: Sichtbare, innere Knoten
- ***termination nodes*** $\mathcal{T}_{i-1}$: Abbruch der Traversierung (Blattknoten oder unsichtbarer innerer Knoten

Traversierung im Frame i in Back-to-Front Ordnung

- Überspringe open nodes $N \in \mathcal{O}_{i-1}$
- Verarbeite termination nodes $N \in \mathcal{T}_{i-1}$:
  - N ist Blatt: Starte Occ.Query, speichere Occ.Query in ***query queue*** $\mathcal{Q}$ & rendere Objekte (ohne Prüfung des Query-Ergebnisses!)
  - N ist innerer Knoten: Starte Occ.Query, speichere Occ.Query in $\mathcal{Q}$
- Verarbeite Query-Ergebnisse (nur innere Knoten):
  - Wenn N Blattknoten, hole Query-Ergebnis erst zum Schluss
  - Wenn N innerer Knoten: Starte Occ.Query für die Kinder & speichere Occ.Query in $\mathcal{Q}$

Beispiel: CHC: Traversierung (Teil I)



Beachte: Occlusion Queries werden nur für Termination Nodes abgesetzt.

Offener Punkt: Behandlung von Knoten, die unsichtbar werden

4.3.4 Occlusion Culling ...

Beispiel: CHC: Traversierung (Teil II)

Pull-Up: Beide Kinderknoten sind/werden unsichtbar $\Rightarrow$ Elternknoten wird unsichtbar und zum Termination Node.