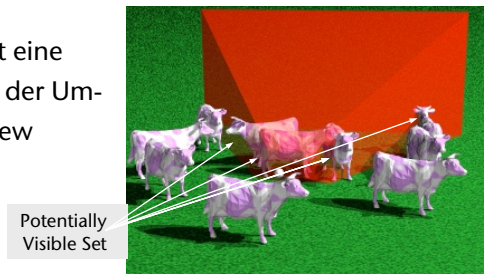
View-Frustum Culling

[Clark 1976]

- In vielen realen Szenen ist eine substanzielle Prozentzahl der Umgebung außerhalb des View Frustums



Bounding Volumes (BVs)

- Test pro Polygon ist zu teuer, wäre langsamer als ohne VFC
- Teste deswegen ganze Objekte (Menge von Polygonen), ob außerhalb des View-Frustums
- Schnelle Tests mit einfachen Hüllkörpern (*bounding volumes*, BVs):

Kugel

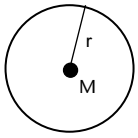
Achsenparallele Bounding Box (AABB)

Orientierte BB (OBB)

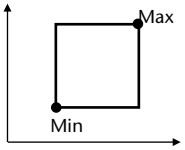
- Das Verfahren ist effizient nur dann, wenn
  Cost(BV-Test) << Cost(Rendern der Gruppe)

Darstellung der BVs

- Kugel := (Mittelpunkt, Radius)

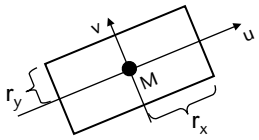


- AABB := (Min, Max) =
  $(x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max})$



- OBB ist definiert durch
  - Mittelpunkt
  - 3 Achsen
  - 3 „Radien“
  - Entspricht 3x4 Matrix:

    $T(M) \cdot R(u,v,w) \cdot S(r_x, r_y, r_z)$



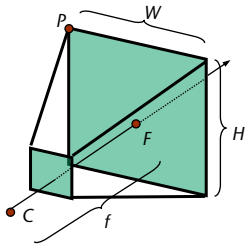
Darstellung des View Frustums

- Vorgehen:
  1. Parameter aus gluPerspective und gluLookAt verwenden
  2. Eckpunkte des Frustums berechnen
  3. Ebenen des Frustums berechnen
- Ecken bestimmen (in Weltkoordinaten):

$$F = C + f \cdot \mathbf{d}$$

$$P = F + \frac{1}{2} H\mathbf{v} - \frac{1}{2} W\mathbf{u}$$

Analog alle anderen Ecken



- Aus den Ecken die Ebenen bestimmen:
  - 3 Punkte genügen (Kreuzprodukt, Aufpunkt einsetzen)
  - Achtung: achte auf konsistente Orientierung der Normalen!
  - Kleine Opt.: Die Normalen der Near- und Far-Plane kennt man schon

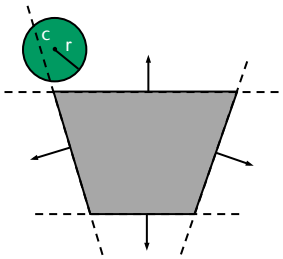
Test Kugel — Frustum

- Gegeben: 6 Ebenengleichungen

$$E_i : x \cdot n_i - d_i = 0$$

eine Kugel

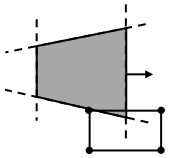
$$(x - c)^2 - r^2 = 0$$



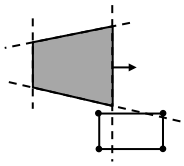
- Frage: befindet sich die Kugel komplett außerhalb des Frustums?
- Ja $\leftrightarrow \exists i : \quad c \cdot n_i - d_i > r$
- Falls $\exists i : \quad |c \cdot n_i - d_i| \leq r$
  dann schneidet die Kugel eine der Ebenen (aber nicht notwendigerweise das Frustum)
- Falls $\forall i : \quad c \cdot n_i - d_i < -r$
  dann ist die Kugel komplett innerhalb des Frustums

Test Box—Frustum

- Achtung: es genügt nicht festzustellen, daß alle Ecken außerhalb des Frustums liegen!
  - Gegenbeispiel:



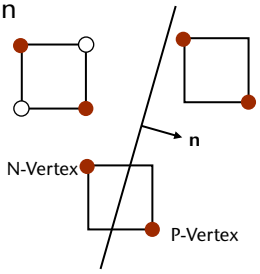
- Einfacher Test:
  Alle 8 Ecken sind auf der positiven Seite derselben Ebene
  → Box ist außerhalb
- Dieser Test produziert sog. „false positives“:



- Die Box ist komplett innerhalb ↔ alle Ecken sind auf der neg. Seite irgend einer Ebene

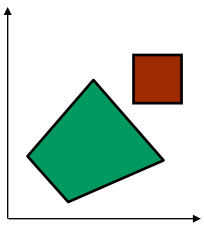
Optimierungen

- Es genügt, 2 Ecken gegen jede Ebene zu testen
  - Wir bezeichnen mit *„N-Vertex“* denjenigen Vertex aller Ecken, der auf der Funktion
    $$f(x) = x \cdot \mathbf{n} - d$$
    das Minimum annimmt. Analog bezeichnet *„P-Vertex“* denjenigen, der das Max annimmt
  - Diese sind (meistens) eindeutig, weil f monoton ist und eine Box konvex ist




```
loop über alle Ebenen i:
  Berechne fi(N-Vertex)
  Falls der N-Vertex auf der pos. Seite:
    → komplette Box ist auf der pos. Seite
    → komplette Box außerhalb des Frustums
  Berechne fi(P-Vertex)
  Falls P-Vertex auf der neg. Seite:
    → komplette Box ist auf der neg. Seite
```


- Wie findet man **schnell** den N- bzw. P-Vertex?
- Falls Box = *axis-aligned bounding box (AABB)* dann geht es schnell
- AABB = $(x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max})$

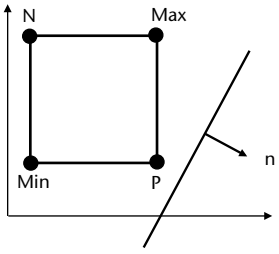


$$P_x = \begin{cases} x_{max} & , n_x \geq 0 \\ x_{min} & , n_x < 0 \end{cases}$$

$$P_y = \begin{cases} y_{max} & , n_y \geq 0 \\ y_{min} & , n_y < 0 \end{cases}$$

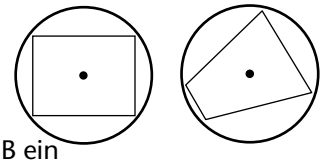
$$P_z = \ldots$$

$$N_x = \begin{cases} x_{min} & , n_x \geq 0 \\ x_{max} & , n_x < 0 \end{cases}$$

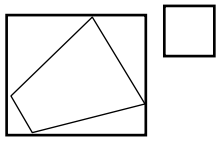


Weitere Optimierungen

- "Meta-BVs": Falls viele Boxes getestet werden müssen, schließe Boxen und Frustum in Kugeln ein



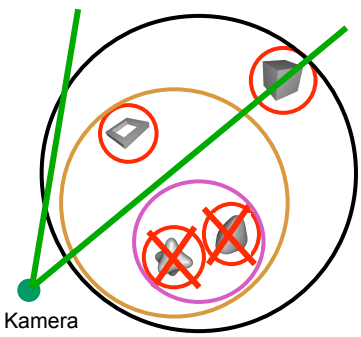
- Oder schließe das Frustum in eine AABB ein

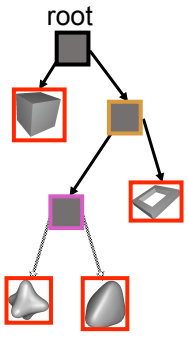


- Zeitliche Kohärenz: wenn Box durch eine bestimmte Ebene gecullt wurde, speichere diese Ebene und teste diese beim nächsten Mal zuerst. Wahrscheinlichkeit ist hoch, daß diese Ebene wieder die Box rauswirft

HierarchicalView Frustum Culling

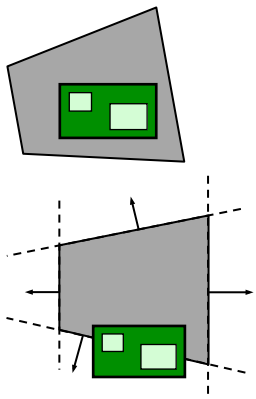
- Erzeuge in jedem Knoten des Szenengraphen ein Bounding-Volumen, das den kompletten Unterbaum einschließt → Bounding-Volumen-Hierarchie (BVH)
- Traversiere diese BVH und teste dabei jeden Knoten





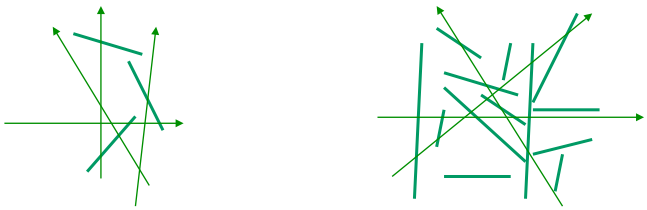
Weitere Optimierung

- *Plane Masking*:
  - Falls eine Box komplett auf der neg. Seite einer Ebene liegt, so auch alle Kinder. Teste diese Ebene also nicht mehr bei den Kindern
  - Falls ein BV vollständig innerhalb, dann auch alle Kinder; teste diese also nicht mehr

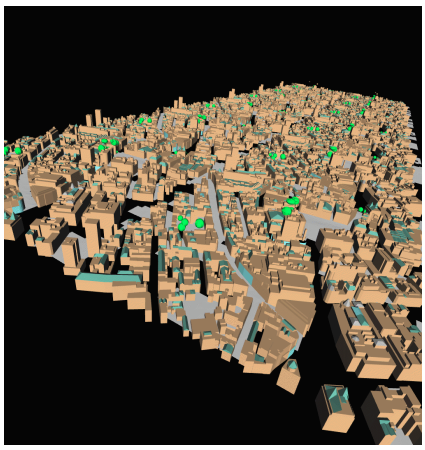


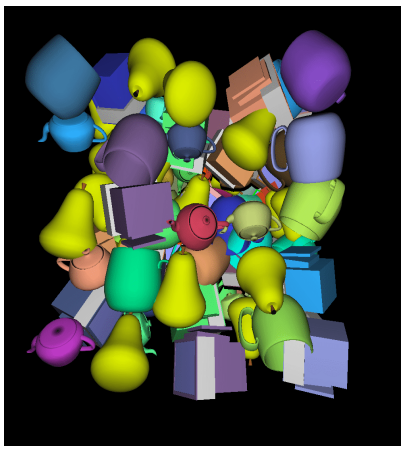
Occlusion Culling

- Occlusion Culling ist immer dann interessant, falls viele Objekte durch einige wenige Objekte verdeckt werden
- Definition: **Depth Complexity** (beobachtungs- und richtungsabhängig)
  - Anzahl Schnittpunkte des Strahls durch die Szene
  - Anzahl Polygone, die auf ein Pixel projiziert werden
  - Anzahl Polygone, die au einem Pixel sichtbar wären, wären alle Polygone transparent



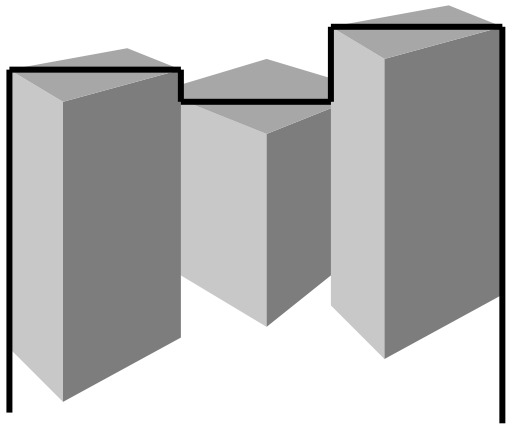
Beispiele für hohe Depth Complexity



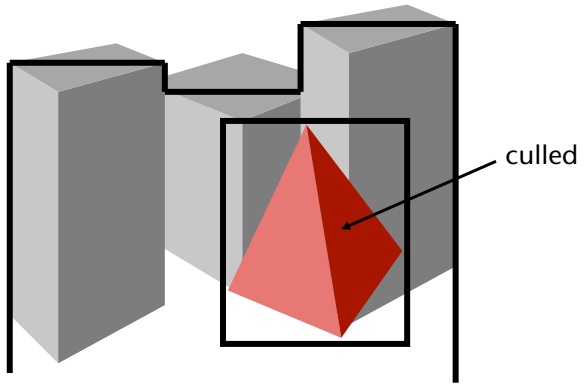


Zunächst: Spezialfall Städte

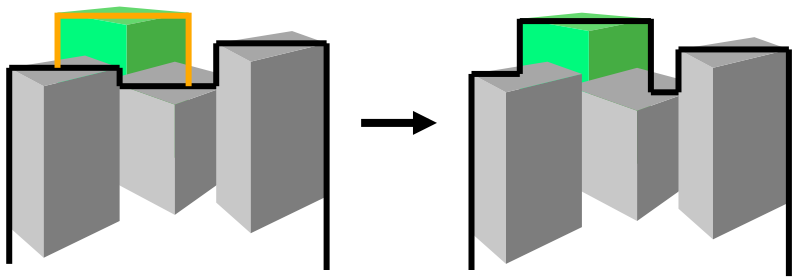
- Rendere die Szene von vorn nach hinten
- Erzeugt einen "occlusion horizon" (schwarze Linie)



- Zum Rendern eines Objektes (hier Tetraeder; liegt hinter den grauen Objekten):
  - Bestimme achsenparallele Bounding-Box (AABB) der Projektion des Obj
  - Vergleiche mit dem Occlusion Horizon

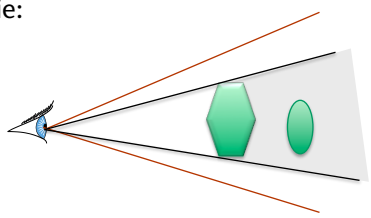


- Falls ein Objekt als sichtbar betrachtet wird:
  - Füge dessen AABB zum bisherigen Occlusion Horizon hinzu

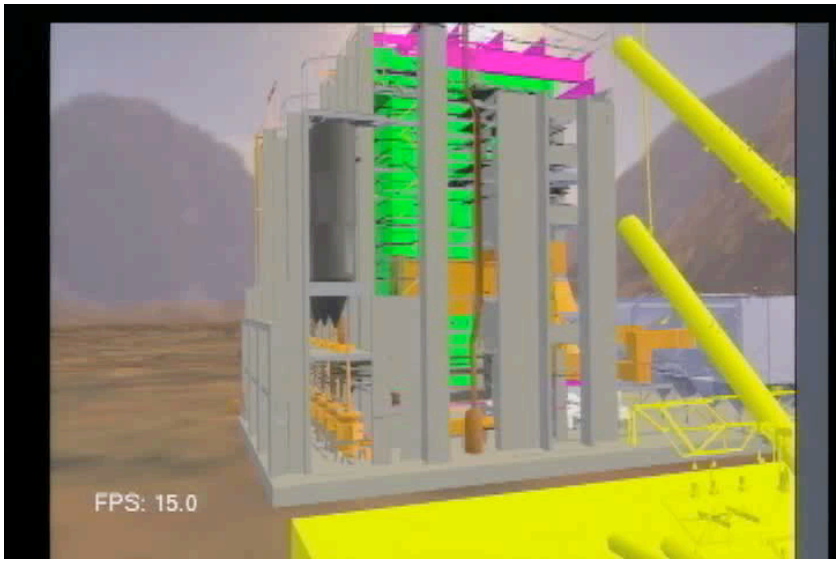


Allgemeines Occlusion Culling

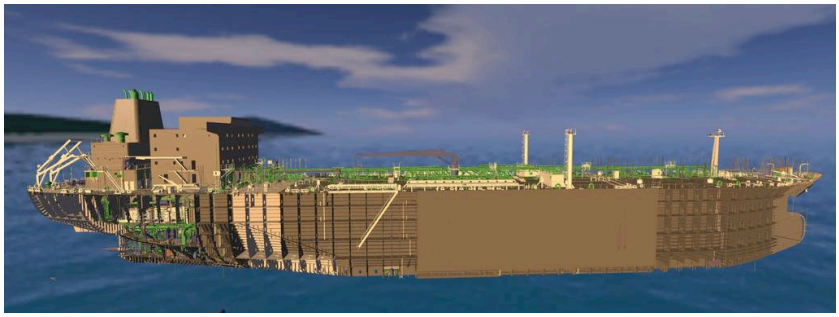
- Gegeben:
  - eine teilweise(!) gerenderte Szene, und
  - ein noch nicht gerendertes Objekt
- Aufgabe:
  - entscheide schnell, ob das Objekt, falls es gerendert würde, Pixel im Framebuffer modifizieren würde;
  - m.a.W.: entscheide schnell, ob das Objekt von der aktuellen Szene komplett verdeckt ist
- Terminologie:



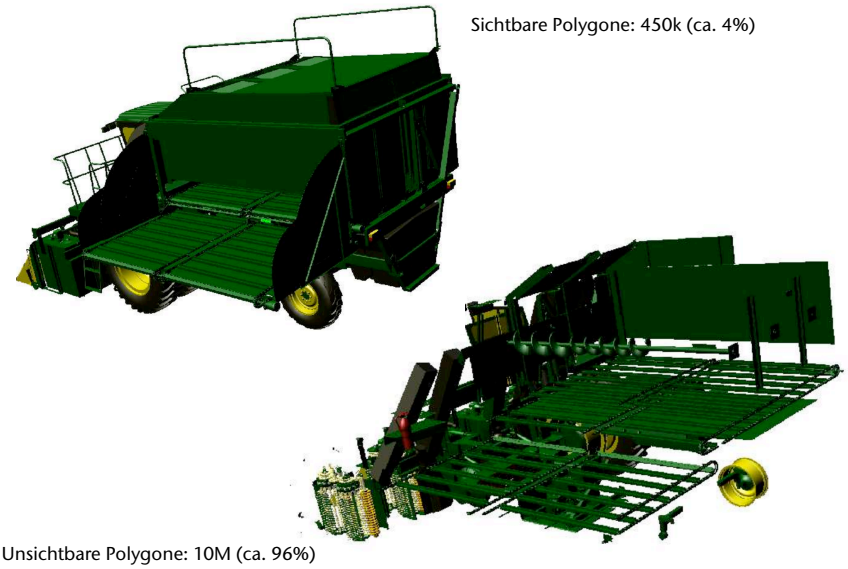
Beispiele für Anwendungen des allg. Occlusion Culling



Power plant, 13 million triangles



"Double Eagle", 4 GB, 82M triangles, 127,000 objects



Sichtbare Polygone: 450k (ca. 4%)

Unsichtbare Polygone: 10M (ca. 96%)

Occlusion-Culling in OpenGL

- Früher als Extension **`ARB_occlusion_query`** ,
  heute im OpenGL-Kern ab Version 1.5
- Das Feature: OpenGL fragen, wieviele Pixel von einer Anweisungsfolge "übermalt" werden würden
  - Zeichne eine einfache Repräsentation ("Proxy"), ohne den Color- oder Z-Buffer zu verändern
  - Wurden durch den Proxy keine Pixel gezeichnet, muß das Objekt selbst nicht mehr gezeichnet werden
- Proxy-Geometrie: opfere zunächst ein wenig Rechenkapazität, um möglicherweise danach viel Rechenleistung einzusparen
  - Einigermaßen genaue Bounding Volumes
  - Keine Texturierung, kein Shading, keine Lichtquellen
  - Keine Farben, Texturkoordinaten, Normalen

- Erzeuge zunächst Occlusion-Query bei der Initialisierung:

```
glGenQueries( int count, unsigned int queryIDs[] );
```

- Rendere eine Menge von Objekten (die viel verdecken)
- Schreiben in Z- und Color-Buffer abschalten (optional):

```
glDepthMask( GL_FALSE );
glColorMask( GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE );
```

- Starte Anfrage für eine Menge anderer Objekte:

```
glBeginQuery( GL_SAMPLES_PASSED, unsigned int querynum );
// rendere Proxy-Geometrie, z.B. Bounding Volume ...
glEndQuery( GL_SAMPLES_PASSED );
```

- Lese Ergebnis der Anfrage:

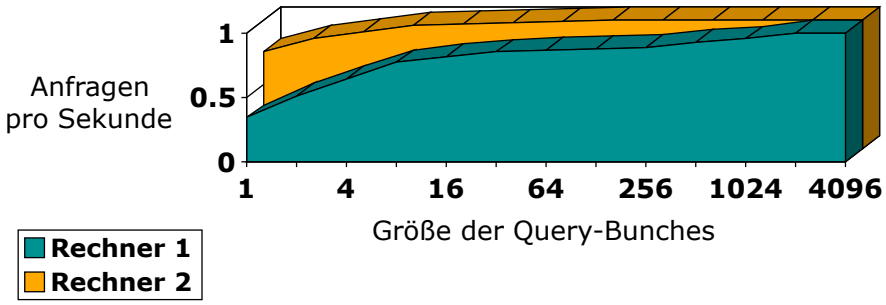
```
glGetQueryObjectiv( int querynum,
                    GL_QUERY_RESULT, int *samplesCounted );
```

Demo

occlusion_query.cpp (~/Work/Lehre/CG1/demos/occlusion_query) - VIM

```
void draw_objects()
{
    glColor3f(1,1,0);
    glPushMatrix();
    glTranslatef(0, -.025, 0);
    glScalef(1, .05, 1);

    // render cube, with occlusion query
    glBeginQueryARB(GL_SAMPLES_PASSED_ARB, oq_plane);
    glutSolidCube(.5);
    glEndQueryARB(GL_SAMPLES_PASSED_ARB);
    glPopMatrix();

    // render sphere, with occlusion query
    glColor3f(1, 0, 0);
    glPushMatrix();
    glTranslatef(0, .25, 0);
    glBeginQueryARB(GL_SAMPLES_PASSED_ARB, oq_sphere);
    glutSolidSphere(.25, 10, 20);
    glEndQueryARB(GL_SAMPLES_PASSED_ARB);
    glPopMatrix();
}

void set_app_info_string()
{
    GLuint plane_samples, sphere_samples;

    // get results of occlusion queries
    glGetQueryObjectuivARB(oq_plane, GL_QUERY_RESULT_ARB, &plane_samples);
    glGetQueryObjectuivARB(oq_sphere, GL_QUERY_RESULT_ARB, &sphere_samples);

    string s;
    char buff[80];

    s = "visible samples\n  plane: ";
    sprintf(buff, "%d", plane_samples);
    s += buff;
    if( plane_samples == 0 )
    {
        s += "  -- no samples visible";
    }
    s += "\n sphere: ";

    sprintf(buff, "%d", sphere_samples);
    s += buff;
    if( sphere_samples == 0 )
    {
        s += "  -- no samples visible";
    }
```

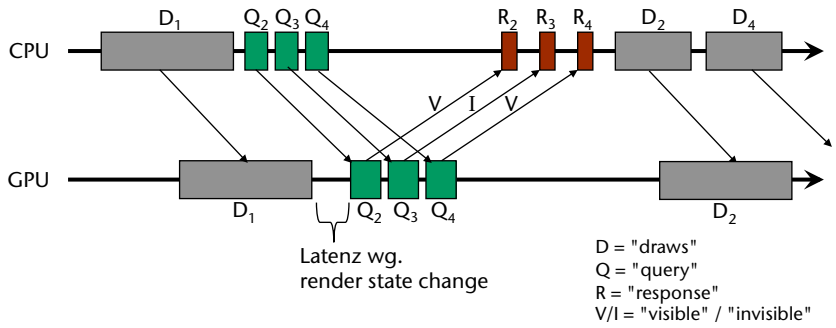

- Bemwerkung: Anfrageergebnisse abholen erfolgt synchron!
- Schicke deshalb einen Folge von Anfragen, lese erst *danach* das Ergebnis der Folge



Der naive "draw-and-wait" Ansatz

```
Sortiere Objekte ungefähr nach Tiefe in Szene
Erzeuge Query-Folge
while einige Objekte noch nicht gerendert:
  For each Objekt in Query-Folge:
    BeginQuery
    Rendere bounding volume
    EndQuery
  For each Objekt in Query-Folge:
    GetQuery
    if #Pixel gezeichnet > 0:
      Rendere Objekt
```


- Probleme des naiven Ansatzes:
  - Sehr hohe Antwortzeit (latency) eines Queries wegen:
    - langer Graphik-Pipeline,
    - etwas Zeit durch das Abarbeiten des Queries (Rasterisierung), und
    - Transfer des Resultats zurück zum Host.



- Folge: "CPU stalls" und "GPU starvation"

Coherent Hierarchical Culling (CHC & CHC++) [2008]

- Hier allerdings vereinfachte Darstellung (u.a. ohne Hierarchie)
- Gegeben: Menge von Objekten
  - Hier: Objekt = Menge von sinnvoll zusammenhängenden Polygonen
- Ideen:
  - Führe eine Queue mit, in der abgesetzte Hardware-Occlusion-Queries gespeichert werden
  - Annahme zunächst: falls ein Objekt im letzten Frame sichtbar war, dann ist es auch im aktuellen Frame sichtbar
  - Falls ein Objekt unsichtbar war, checke zuerst dessen Visibility
  - Warte nicht auf das Resultat, sondern gehe weiter die Liste durch
  - Bearbeite Query-Resultate sobald sie verfügbar werden

Der Algorithmus

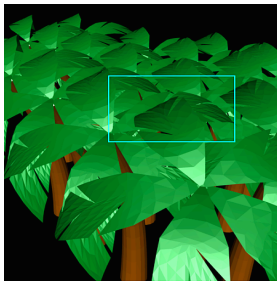
```
L = list of all objects (incl. BVs)
Q = queue for occlusion queries (init'ly empty)

sort L from front to back with respect to current viewpoint

repeat:

  // process list of objects
  if L not empty:
    O = L.front
    if O inside view frustum:
      issue occlusion query with BV(O)
      append O to Q
      if O is marked "previously visible":
        render O
  end if
...
```

```
...
  // process queries
  while Q not empty and
        result of occlusion query Q.front available
    V = Q.pop
    if # visible pixels of query V > threshold:
      V.obj = "visible"
      render V.obj
    else:
      V.obj = "invisible"
  end while
until Q empty and L empty
```

Im Folgenden: schrittweise Verbesserung dieses Algorithmus'

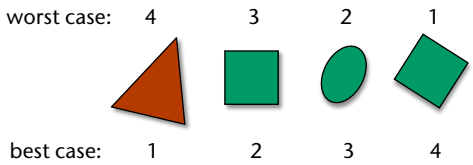
Aggressive approximate Culling

- Oft nur konservatives Culling:
  - Wenn auch nur ein Pixel des BVs sichtbar ist, kann auch ein Pixel des Objektes sichtbar sein → Objekt zeichnen
  - Nachteil: oft sind äußere Teile der BVs sichtbar, an denen sich keine Objektpixel befindet
- Idee: ignoriere kaum sichtbare Objekte
  - Objekt wahrschl.(!) nicht sichtbar, wenn nur wenige Pixel des BVs sichtbar
  - Heuristik: zeichne Objekt nur dann, wenn Query Ergebnis ≥ n
  - Eventuell „kleine“ Löcher in einem oder zwischen Objekten

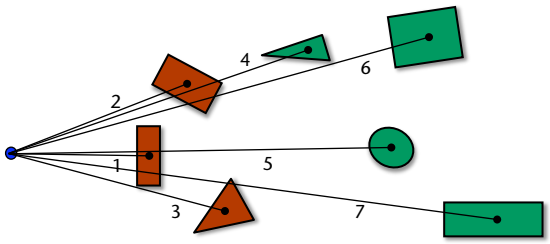


Sortieren der Objekt-Liste

- Beobachtung: je nach dem, in welcher Reihenfolge man die Objekte rendert, bekommt man eine hohe Culling-Rate oder nicht



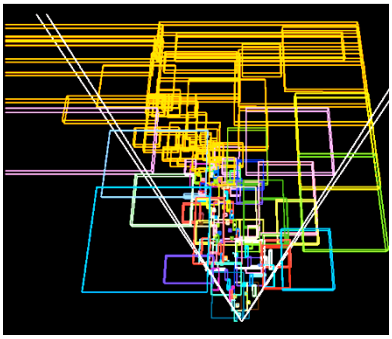
- Lösung: sortiere die Objekt-Liste nach Entfernung zum Viewpoint



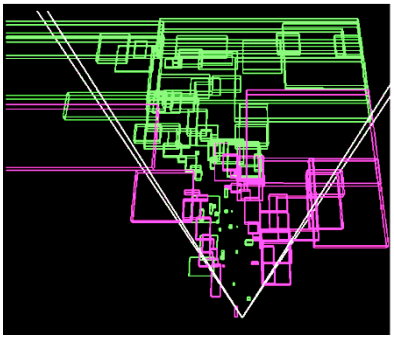
Batching Queries

- Problem: ein Query = teure State-Changes
  - Vorher: das Schreiben in Color- und Z-Buffer abschalten
  - Nachher: wieder einschalten
  - Dieser Overhead kostet mehr Zeit als der eigentliche Query!
- Idee: Batching
- Führe 2 zusätzliche Queues ein
  - Beide enthalten Objekte, die auf Visibility getestet werden sollen
  - I-Queue: enthält Objekte, die vorher "invisible" waren
  - V-Queue: dito für "visible"
  - Parameter: Batch-Größe b (ca. 20-80)
  - Grundsätzlich: erst, wenn Batch-Größe erreicht, wird Liste der Queries an OpenGL abgeschickt
- "Previously visible" Objects werden weiterhin sofort gerendert

- Beispiel: jede Farbe = ein State-Change



Naiv



CHC++

Fusion (potentiell) verdeckter Geometrie

- Beobachtung:
  - Wenn wir wüssten, daß eine Menge von Objekten im aktuellen Frame verdeckt ist, dann könnten wir dies durch genau ein Occlusion-Query verifizieren
  - Objekte, die viele Frames verdeckt waren, sind sehr wahrscheinlich auch im aktuellen Frame verdeckt (*temporal coherence of visibility*)
- Idee:
  - Erfinde ein "Orakel", das für eine gegebene Menge von Objekten mit großer Wahrscheinlichkeit vorhersagen kann, ob die coherence of visibility erfüllt ist
  - Falls W.keit hoch genug, teste diese Menge durch 1 Query:

```
glBeginQuery( GL_SAMPLES_PASSED, q );
  rendere BVs der Menge von Objekten ...
glEndQuery( GL_SAMPLES_PASSED );
glGetQueryObjectiv( q, GL_QUERY_RESULT, *samples );
```

Dies nennen wir im folgenden Multiquery!

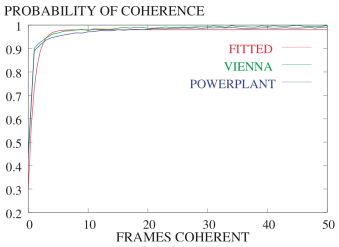
- Definition: Visibility-Persistenz

$$p(t) = \frac{I(t+1)}{I(t)}$$

wobei $I(t)$ = Anzahl Objekte, die in den vergangenen t Frames verdeckt waren

- Interpretation: p(t) = Wahrscheinlichkeit, daß ein Objekt, das t Frames lang verdeckt war, auch im kommenden Frame verdeckt sein wird
- Beobachtung: ist erstaunlich unabhängig von Obj und Szene
- Folge: läßt sich gut approximieren durch analytische Funktion!

$$p(t) \approx 0.99 - 0.7e^{-t}$$



- Motivation für die Beobachtung:

- Sei Menge M von Objekten gegeben;
  sei t_O = Anzahl vergangener Frames, die $O \in M$ verdeckt war
- Definiere "Orakel" c für die Wahrscheinlichkeit, daß alle Objekte aus M im aktuellen Frame verdeckt sein werden:

$$c(M) = \prod_{O \in M} p(t_O)$$

- Definiere damit
  - Kosten eines Multiquery:

$$C(M) = 1 + (1 - c(M)) \cdot |M|$$

  - Nutzen eines Multiquery:

$$B(M) = |M|$$

- Definiere damit den Wert eines Multiquery:

$$V(M) = \frac{B(M)}{C(M)} = \frac{1}{1 + (1 - c(M))}$$

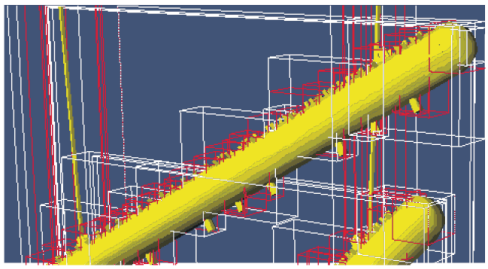
- Wenn die I-Queue dann zu irgend einem Zeitpunkt voll ist:
  - Sortiere die Objekte O_i in der I-Queue nach $t_O \rightarrow \{O_1, \ldots, O_b\}$
  - Suche damit einfach greedy das Maximum

$$\max_n \{ V(\{O_1, \ldots, O_n\}) \}$$

  - Setze eine Multiquery für diese ersten n Objekte aus der I-Queue ab
  - Wiederhole, bis die I-Queue leer ist

Tighter Bounding Volumes

- Beobachtung: je größer das BV im Verhältnis zum Objekt, desto wahrscheinlicher liefert ein Occlusion-Query ein "*false positive*" (behauptet "visible", ist in Wahrheit aber "invisible")
- Ziel: möglichst enge BVs
- Randbedingungen:
  - BVs müssen sehr schnell zu rendern sein
  - BVs dürfen nicht viel Speicher kosten
- Idee:
  - Zerlege Objekt in einzelne Stücke (Cluster von Polygonen)
  - Lege um jeden Cluster eine BBox (AABB)
  - Verwende als BV des Objektes die Vereinigung der "kleinen" BBoxes



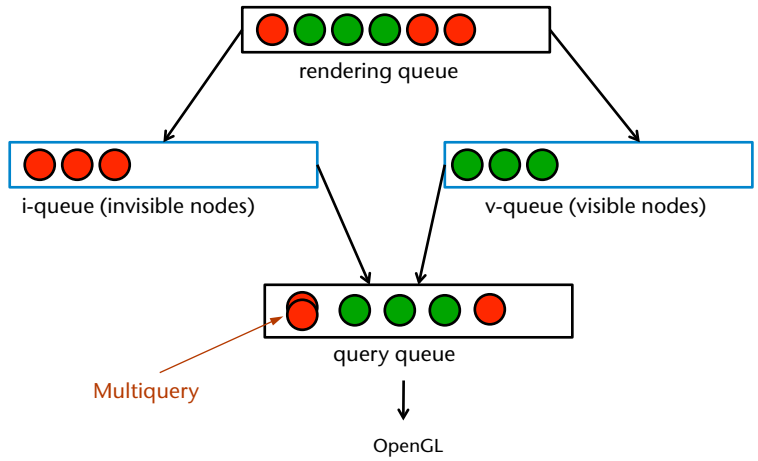
- Frage: wie klein soll man die "kleinen" AABBs (bzw. die Cluster) machen?
- Beobachtung: je größer die Anzahl der kleinen AABBs, ...
  - ... desto größer die Wahrscheinlichkeit, daß "invisible" korrekt erkannt wird; aber
  - ... desto größer die Oberfläche → längere Rendering-Zeit des daraus resultierenden Occlusion-Queries
- Strategie zur Konstruktion der "engen AABBs":
  - Unterteile die Cluster rekursiv
  - Abbruchkriterium: falls

$$\sum \text{Oberfläche der kleinen AABBs} > \sigma \cdot \text{Oberfläche der großen AABB}$$

  - Parameter σ hängt ab von der Graphikkarte ($\sigma \approx 1.4$ scheint OK)

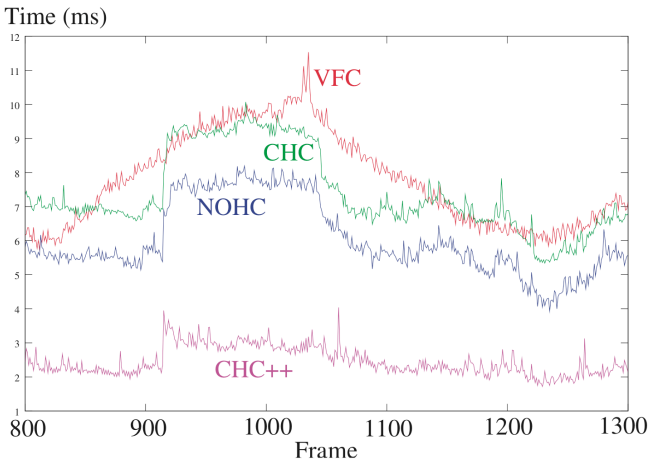
Alles zusammen

- Die Queues in CHC++:



Resultate

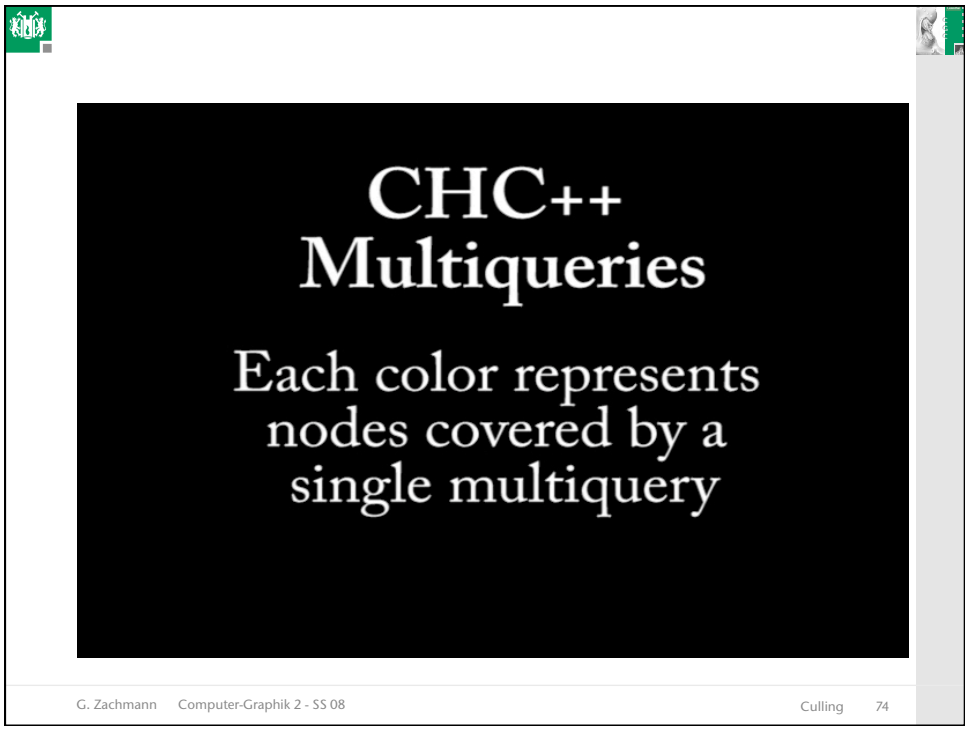
- Walkthrough durch das Power-Plant-Modell:

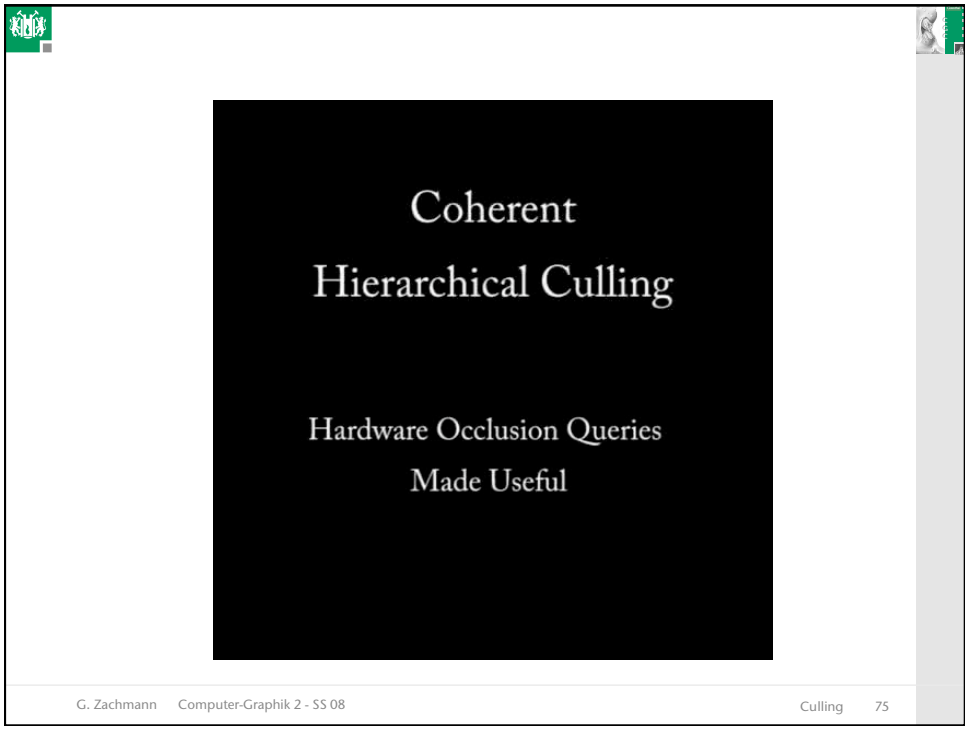


State Changes: CHC vs. CHC++

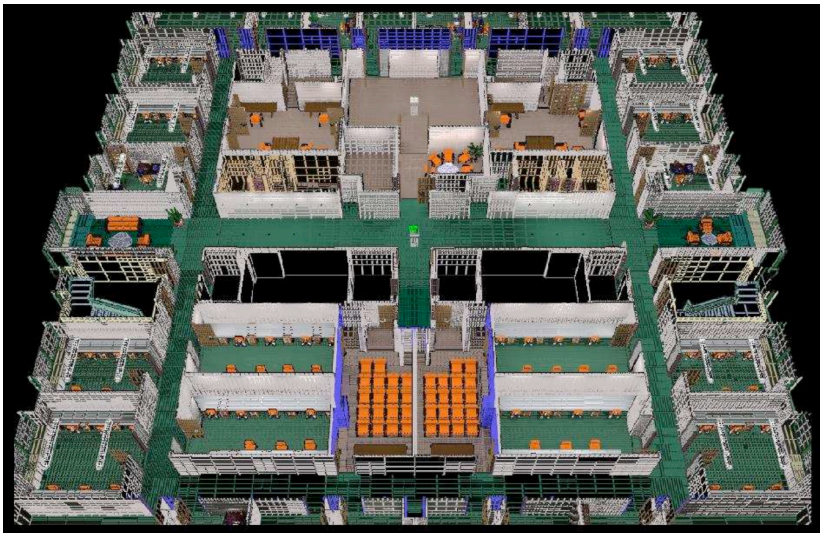
Each color represents a state change required by the algorithm

Powerplant Walkthrough 2


CHC++
Multiqueries
Each color represents
nodes covered by a
single multiquery

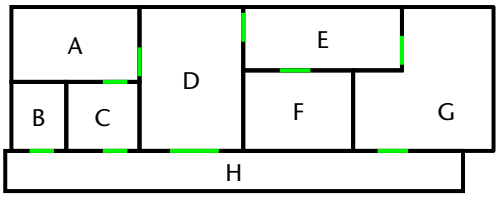

Coherent
Hierarchical Culling
Hardware Occlusion Queries
Made Useful

Weiterer Spezialfall: Architekturmodelle



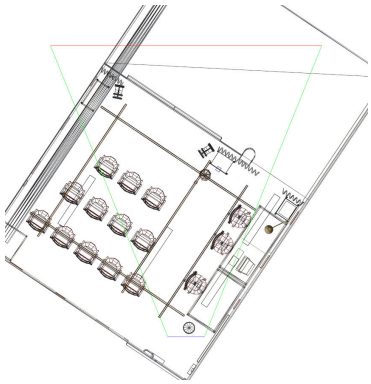
Zellen und Portale (*Portal Culling*)

- Szenario: Walk-through durch Gebäude und Städte
- Durchsichtige Portale verbinden die Zellen
  - Türen, Fenster, Öffnungen, ...
- Beobachtung: Zellen sehen einander nur durch die Portale



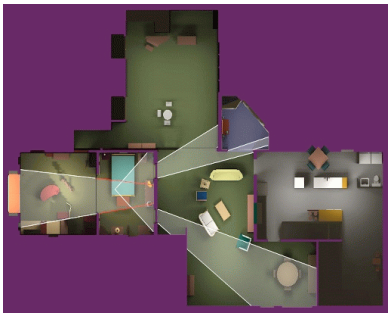
- Welche Zelle sind im PVS enthalten?
  - Die Zelle welche den Viewpoint enthält
  - Und diese Zellen, welche ein Portal zur Ausgangszelle besitzen

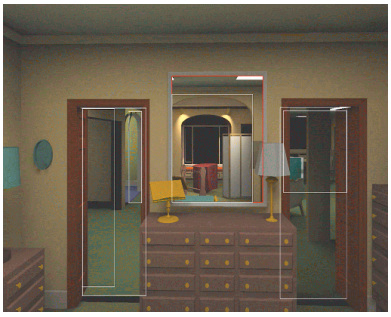
Beispiel-Szene



Resultat

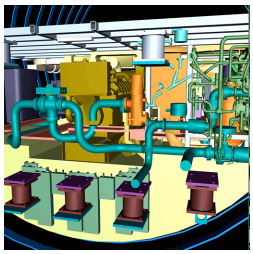
- Beispiel-Szene:





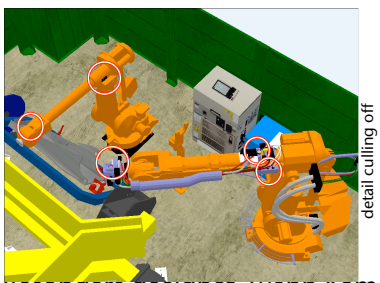
- Speedup ist stark vom Modell und Viewpoint abhängig
  - Framerate ist 1-10—faches der Framerate ohne Cells-&-Portals-Methode
  - Für typische Viewpoints entfernt die Methode 20% – 50% des Modells

- Anwendungsgebiete
  - Computerspiele
  - Gebäude
  - Städte
- Nicht geeignet für CAD-Daten
  - Flugzeuge
  - Schiffe
  - Industrieanlagen
- Nicht geeignet für natürliche Objekte
  - Pflanzen
  - Wälder



Detail Culling

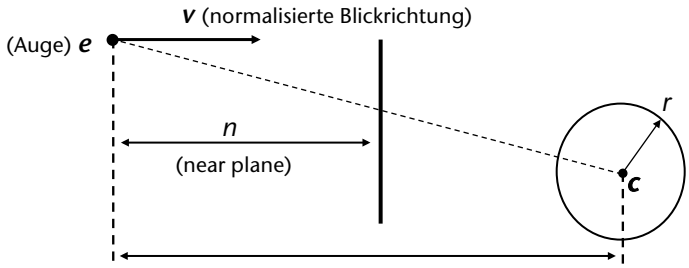
- Idee: Objekte, die bei der Projektion weniger als *N* Pixel belegen, werden nicht dargestellt
- Diese Annäherung entfernt auch Teile, die möglicherweise im endgültigen Bild sichtbar wären
- Vorteil: Trade-Off Qualität/Geschwindigkeit



- Besonders geeignet, wenn Kamera in Bewegung (je schneller, desto größere Details können gecullt werden)

Abschätzung der projizierten Größe eines Objektes

- Schätze Größe des BVs in Screen-Space ab:



$d = \mathbf{v} \cdot (\mathbf{c} - \mathbf{e})$ (Distanz entlang *v*)

$\hat{r} = r \cdot \frac{n}{d}$ (Schätzung des projizierten Radius)

$\pi \hat{r}^2 =$ geschätzte Fläche der projizierten Kugel