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 diese Wahrsch.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!

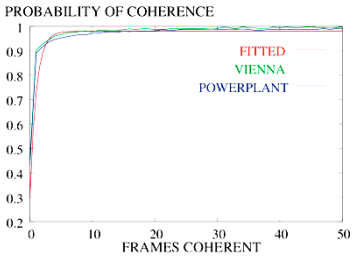
- Definition: Visibility-Persistenz

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

wobei $I(t)$ = Anzahl Objekte, die in den vergangenen t Frames ständig 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!

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



- Sei t_O = Anzahl vergangener Frames, die Objekt O verdeckt war
- Definiere für ein Menge M von Objekten ein "Orakel" $i(M)$:= die "Wahrscheinlichkeit", daß alle Objekte aus M im aktuellen Frame verdeckt (*invisible*) sein werden (ist nur eine Heuristik!):

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

- Definiere damit
  - Kosten (costs) eines Occlusion-Multiquery (im Batch):

$$C(M) = 1 + c_1|M|$$

  - Erwarteter Nutzen (benefit) eines Multiquery:

$$B(M) = c_2 i(M) \sum_{O \in M} \text{num polygons of } O$$

- Definiere damit den erwarteten Wert eines Multiquery:

$$V(M) = \frac{B(M)}{C(M)}$$

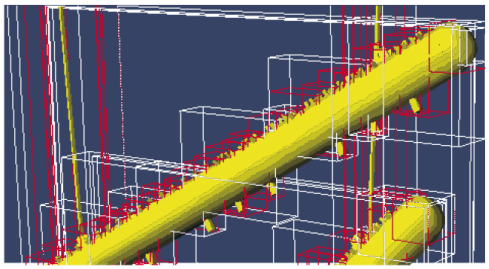
- Wenn die I-Queue dann zu irgend einem Zeitpunkt voll ist:
  - Sortiere die Objekte O_i in der I-Queue nach $t_O \rightarrow \{O_1, \ldots, O_b\}$
  - Suche damit einfach greedy das Maximum

$$\max_{n=1\ldots b} \{V(\{O_1, \ldots, O_n\})\}$$

  - 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



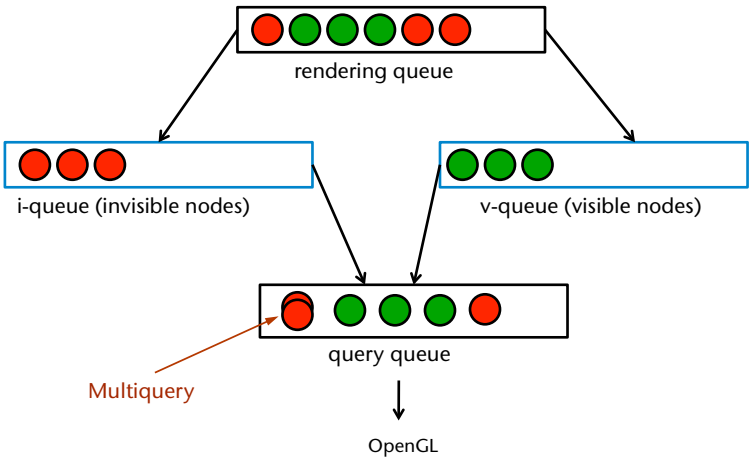
- Frage: wie klein soll man die "kleinen" AABBs (bzw. die Cluster) machen?
- Beobachtung: je größer die Anzahl der kleinen AABBs, ...
  - ... desto größer die Wahrscheinlichkeit, daß "invisible" korrekt erkannt wird; aber
  - ... desto größer die Oberfläche → längere Rendering-Zeit des daraus resultierenden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

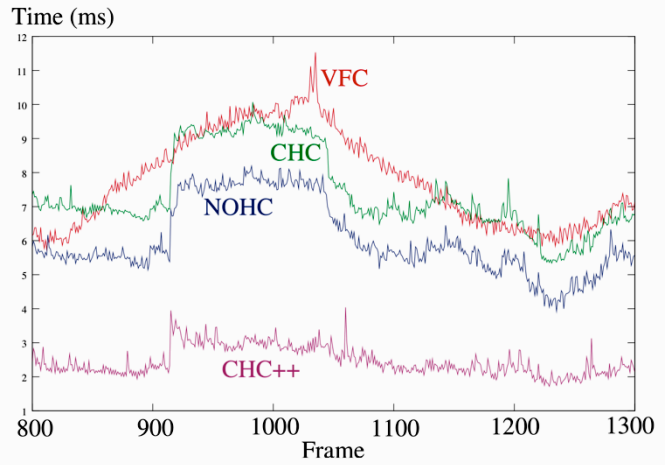
Alles zusammen

- Die Queues in CHC++:



Resultate

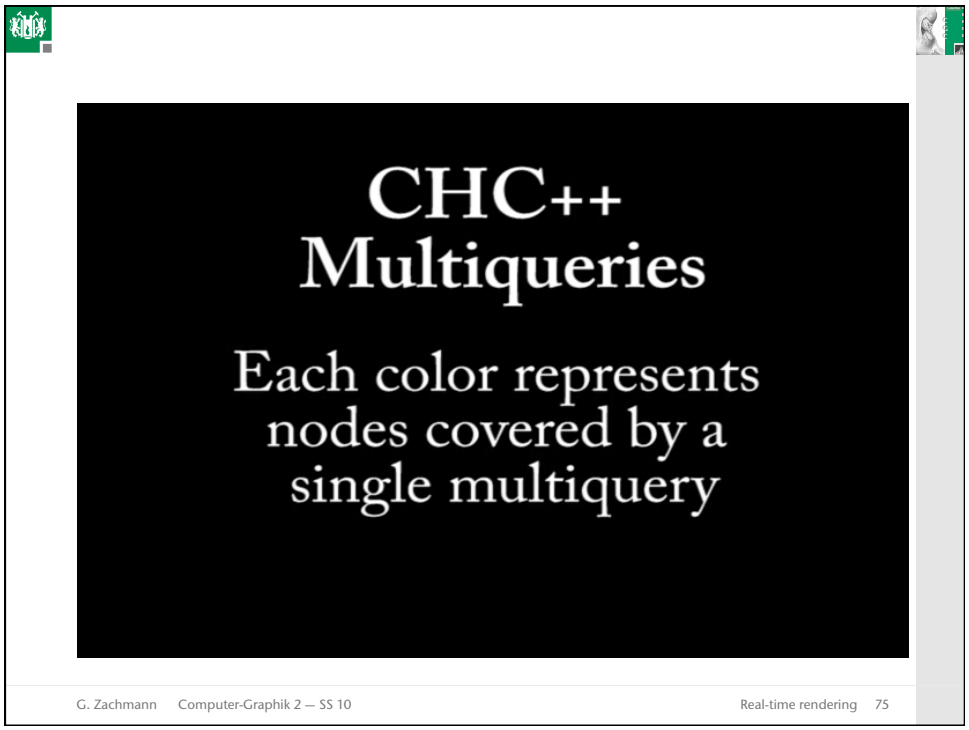
- Walkthrough durch das Power-Plant-Modell:

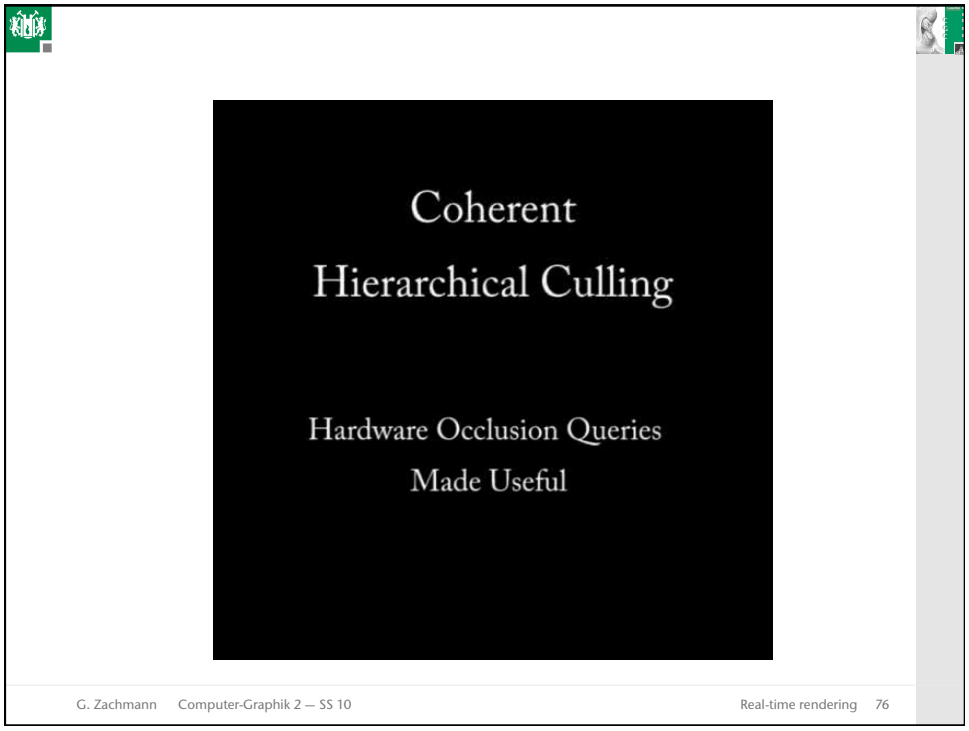


State Changes: CHC vs. CHC++

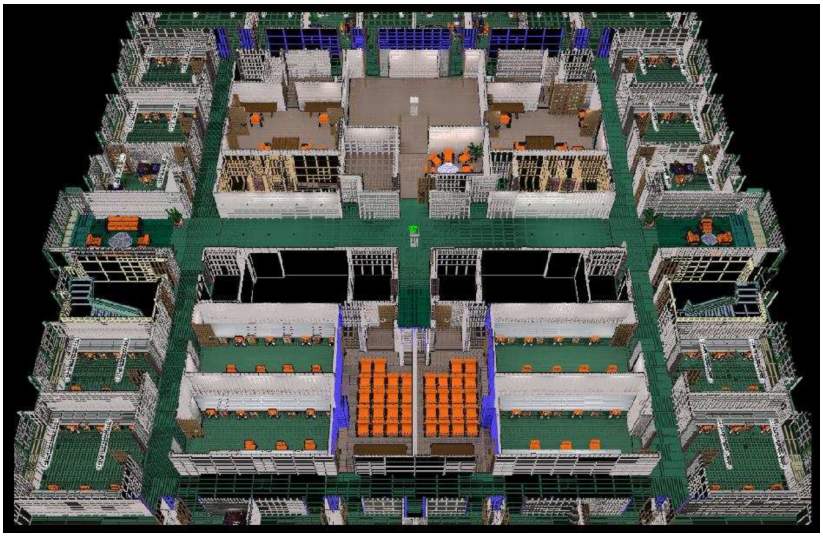
Each color represents a state change required by the algorithm

Powerplant Walkthrough 2


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

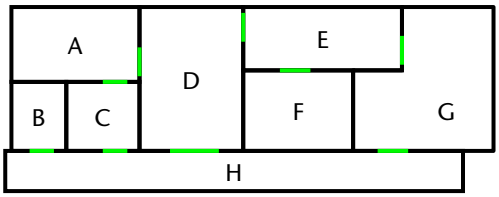

Coherent
Hierarchical Culling
Hardware Occlusion Queries
Made Useful

Weiterer Spezialfall: Architekturmodelle



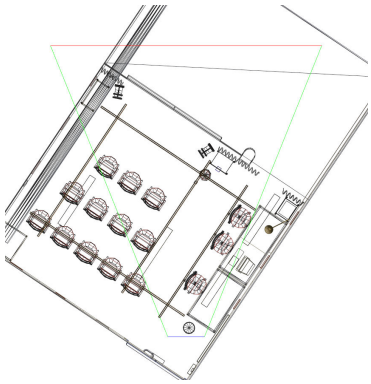
Zellen und Portale (*Portal Culling*)

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



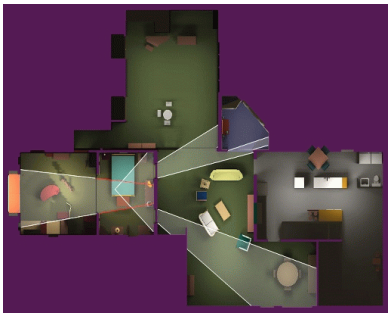
- Welche Zelle sind im PVS enthalten?
  - Die Zelle welche den Viewpoint enthält
  - Und diese Zellen, welche ein Portal zur Ausgangszelle besitzen

Beispiel-Szene



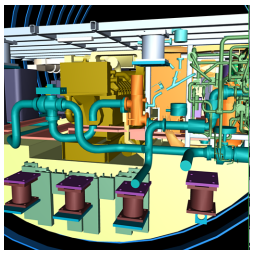
Resultat

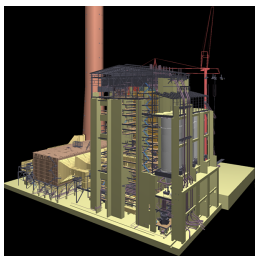
- Beispiel-Szene:



- 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

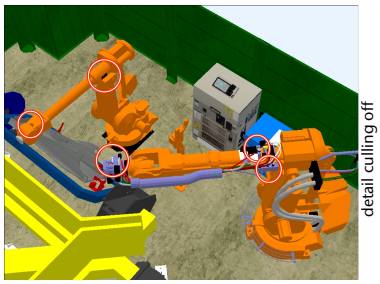
- Anwendungsgebiete
  - Computerspiele
  - Gebäude
  - Städte
  - Schiffe (innen)
- Nicht geeignet für CAD-Daten
  - Flugzeuge
  - Industrieanlagen
- Nicht geeignet für natürliche Objekte
  - Pflanzen
  - Wälder

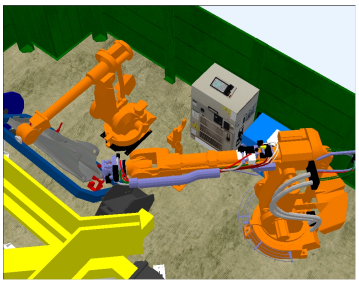




Detail Culling

- 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

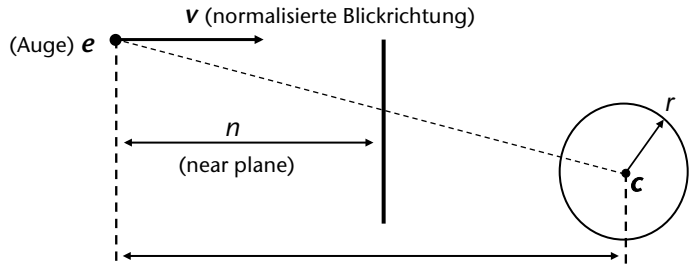




- Besonders geeignet, wenn Kamera in Bewegung (je schneller, desto größere Details können gecullt werden)

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