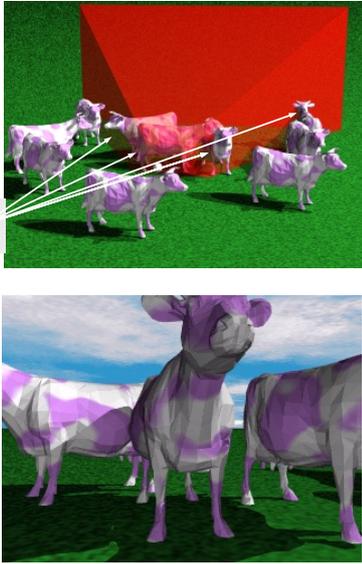


View-Frustum Culling [Clark 1976]

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

Potentially Visible Set



G. Zachmann Computer-Graphik 2 – SS 10
Real-time rendering 30

Bounding Volumes (BVs)

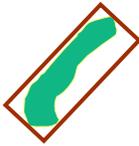
- Test pro Polygon ist zu teuer, wäre langsamer als ohne VFC
- Teste deswegen ganze Objekte (Menge von Polygonen), ob außerhalb des View-Frustums
- Schnelle Tests mit einfachen Hüllkörpern (*bounding volumes*, BVs):



Kugel



Achsenparallele
Bounding Box (AABB)



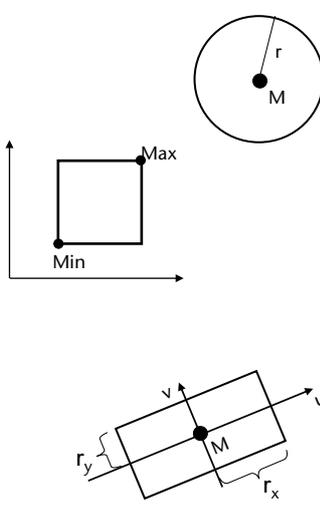
Orientierte BB (OBB)

- Das Verfahren ist effizient nur dann, wenn
 $Cost(BV\text{-Test}) \ll Cost(\text{Rendern der Polygon-Menge})$

G. Zachmann Computer-Graphik 2 – SS 10
Real-time rendering 31

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



G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 32

Darstellung des View Frustum

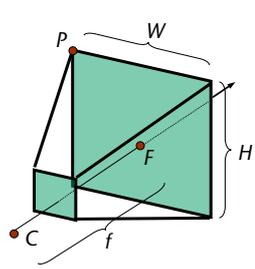
- Vorgehen:
 1. Parameter aus gluPerspective und gluLookAt verwenden
 2. Eckpunkte des Frustum berechnen
 3. Ebenen des Frustum berechnen
- Ecken bestimmen (in Weltkoordinaten):

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

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

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



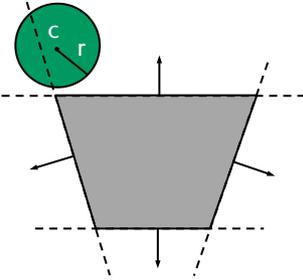
G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 33

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

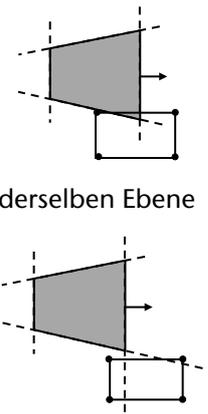


The diagram illustrates a sphere with center c and radius r being tested against a frustum. The sphere is shown intersecting one of the frustum's planes, which is represented by a dashed line. The frustum is a shaded trapezoid with arrows indicating its outward-facing planes.

G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 34

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
 \rightarrow Box ist außerhalb
- Dieser Test produziert sog. „false negatives“:
- Die Box ist komplett innerhalb \Leftrightarrow
 alle Ecken sind auf der negativen Seite aller Ebenen

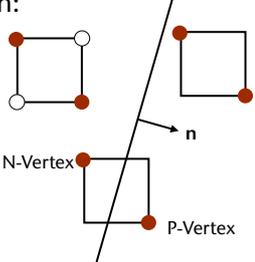


The top diagram shows a box partially inside a frustum. A dashed line represents a plane. All 8 corners of the box are on the positive side of this plane, yet the box is not completely outside the frustum. The bottom diagram shows a box completely inside a frustum. All 8 corners of the box are on the negative side of all planes, indicating it is completely inside.

G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 35

Optimierungen

- Es genügt, 2 Ecken gegen jede Ebene zu testen:
 - Wir bezeichnen mit „*N-Vertex*“ denjenigen Vertex aller Ecken, der auf der Funktion $f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{n} - d$ 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

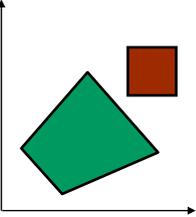


```

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

G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 36

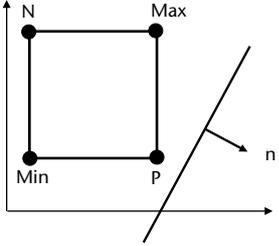
- Wie findet man **schnell** den N- bzw. P-Vertex?
- Falls Box = *axis-aligned bounding box (AABB)* dann geht es schnell
- $AABB = (x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max})$



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

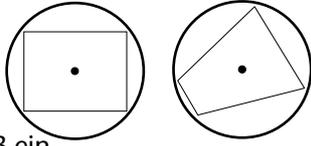
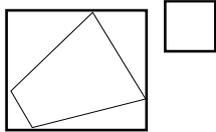
$$P_y = \begin{cases} y_{max} & , n_y \geq 0 \\ y_{min} & , n_y < 0 \end{cases}$$

$$P_z = \dots$$

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


G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 37

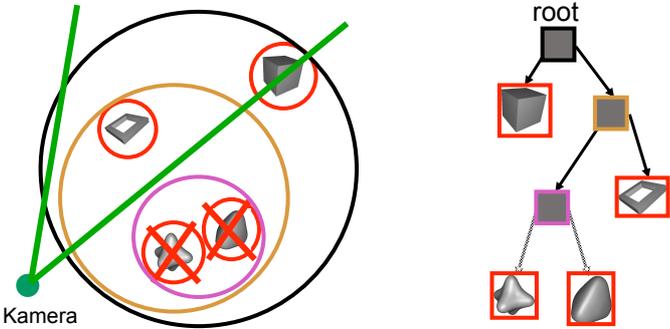
Weitere Optimierungen

- "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

G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 38

Hierarchical View 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



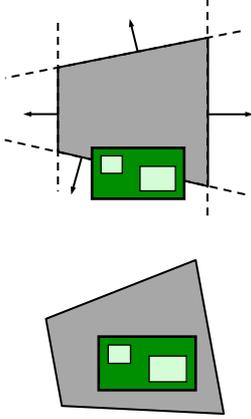
Kamera

root

G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 39

Weitere Optimierung

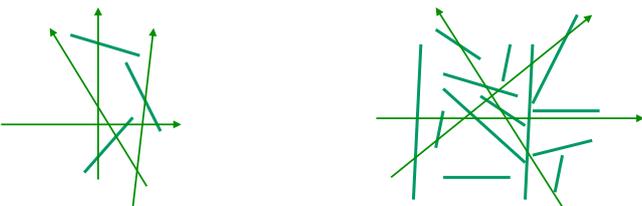
- Plane Masking:**
 - Falls eine Box komplett auf der neg. Seite einer Ebene liegt, so auch alle Kinder. Teste diese Ebene also nicht mehr bei den Kindern
 - Falls ein BV vollständig innerhalb, dann auch alle Kinder; teste diese also nicht mehr



G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 40

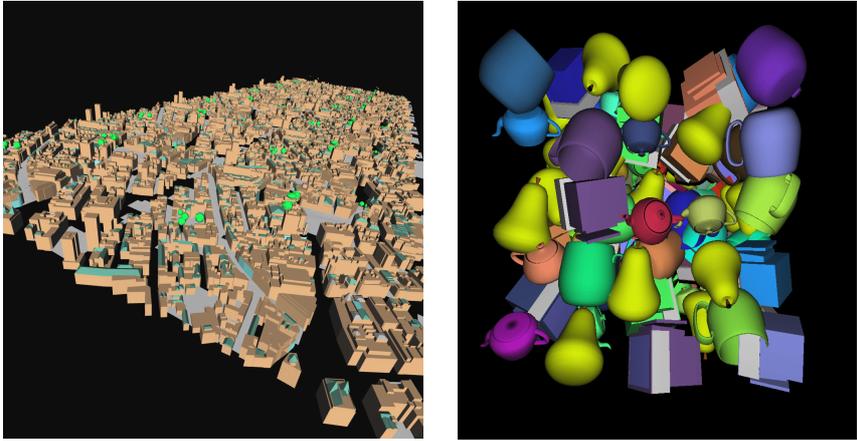
Occlusion Culling

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



G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 41

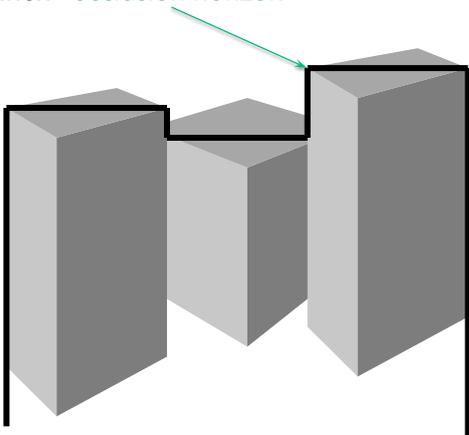
Beispiele für hohe Depth Complexity



G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 42

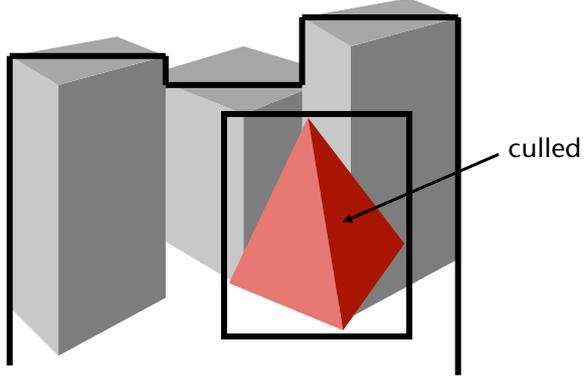
Zunächst der Spezialfall "Städte"

- Rendere die Szene von vorn nach hinten (umgekehrter Painter's Algorithm)
- Erzeugt einen "occlusion horizon"



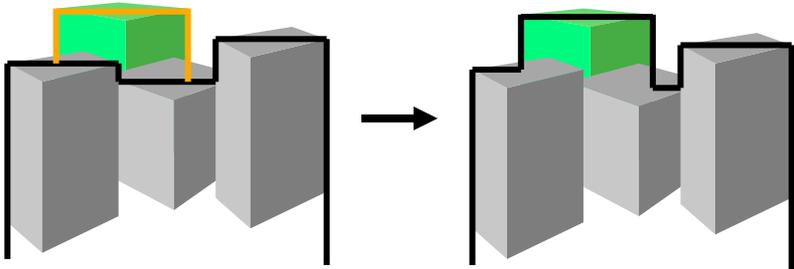
G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 43

- 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



G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 44

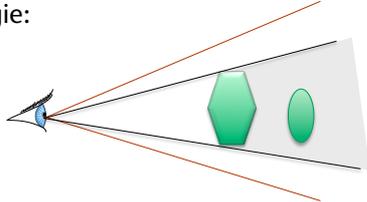
- Falls ein Objekt als sichtbar betrachtet wird:
 - Füge dessen AABB zum bisherigen Occlusion Horizon hinzu



G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 45

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:

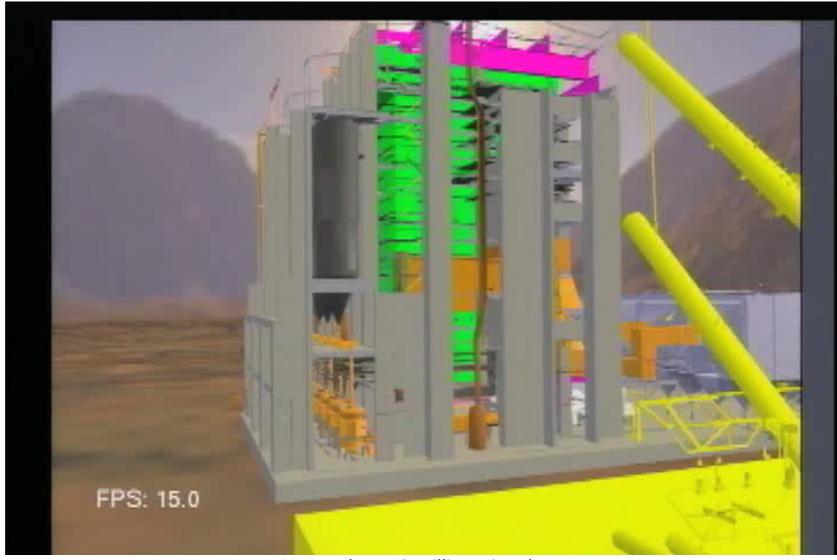


Occluder

Occluded geometry ("occludee")

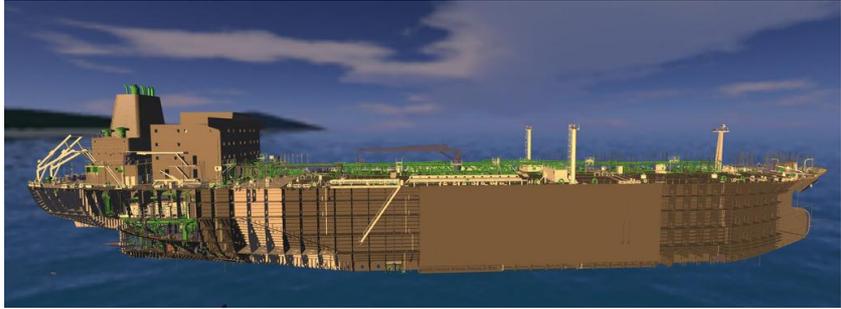
G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 46

Beispiele für Anwendungen des allg. Occlusion Culling



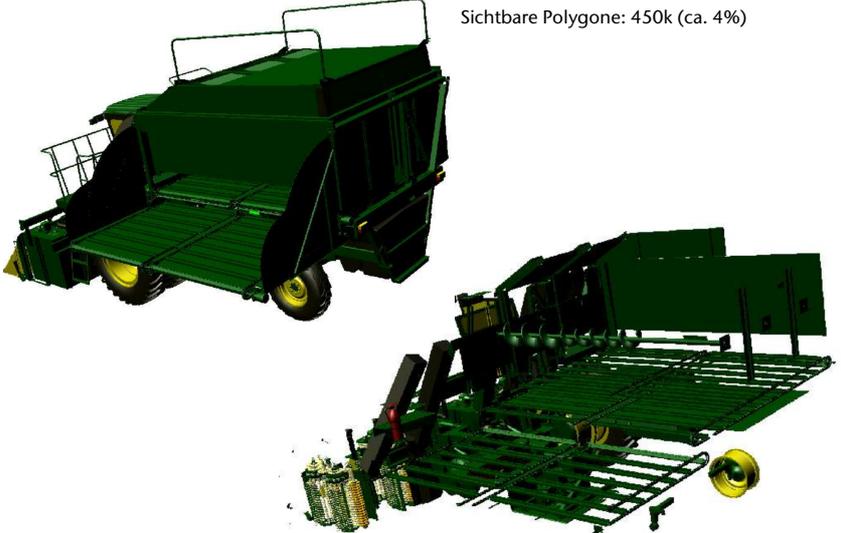
Power plant, 13 million triangles

G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 47



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

G. Zachmann Computer-Graphik 2 — SS 10 Real-time rendering 48



Sichtbare Polygone: 450k (ca. 4%)

Unsichtbare Polygone: 10M (ca. 96%)

G. Zachmann Computer-Graphik 2 — SS 10 Real-time rendering 49

Occlusion-Culling in OpenGL

- Früher als Extension **ARB_occlusion_query** , heute im OpenGL-Kern ab Version 1.5
 - Funktionsweise: fragt OpenGL, wieviele Pixel von einer Anweisungsfolge "übermalt" werden würden
- Ansatz: 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

G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 50

- 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):

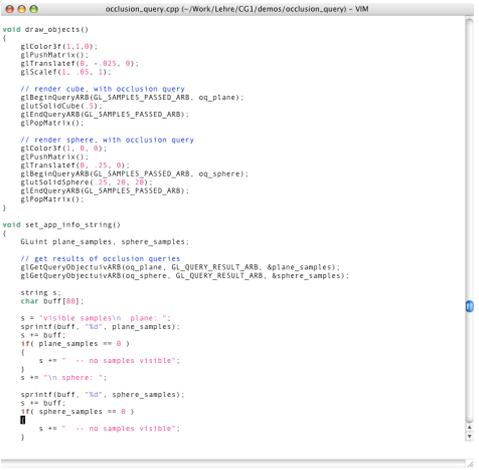

```
glDepthMask( GL_FALSE );
glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );
```
- Starte Anfrage für eine Menge anderer Objekte:


```
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 );
```

G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 51

Demo



```

occlusion_query.cpp (~Work/Lehre/CG1/demos/occlusion_query) - VIM
void draw_objects()
{
    glColor3f(1,0,0);
    glPushMatrix();
    glTranslatef(0, -0.25, 0);
    glScalef(1, 0.5, 1);

    // render cube with occlusion query
    glBeginQueryARB(GL_SAMPLES_PASSED_ARB, og_plane);
    glTranslatef(0, 0, 0);
    glEndQueryARB(GL_SAMPLES_PASSED_ARB);
    glPopMatrix();

    // render sphere with occlusion query
    glColor3f(0, 0, 1);
    glPushMatrix();
    glTranslatef(0, 0.25, 0);
    glBeginQueryARB(GL_SAMPLES_PASSED_ARB, og_sphere);
    glTranslatef(0, 0, 0);
    glEndQueryARB(GL_SAMPLES_PASSED_ARB);
    glPopMatrix();
}

void set_app_info_string()
{
    GLuint plane_samples, sphere_samples;

    // get results of occlusion queries
    glGetQueryObjectuARB(og_plane, GL_QUERY_RESULT_ARB, &plane_samples);
    glGetQueryObjectuARB(og_sphere, GL_QUERY_RESULT_ARB, &sphere_samples);

    string s;
    char buff[80];
    s = "visible samples in plane: ";
    sprintf(buff, "%d", plane_samples);
    s += buff;
    if (plane_samples == 0)
    {
        s += " -- no samples visible";
    }
    s += " in sphere: ";
    sprintf(buff, "%d", sphere_samples);
    s += buff;
    if (sphere_samples == 0)
    {
        s += " -- no samples visible";
    }
}

```

G. Zachmann Computer-Graphik 2 — SS 10 Real-time rendering 52

Batching Queries

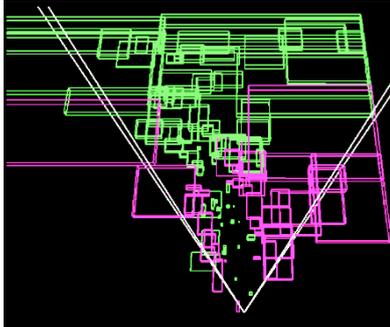
- Problem: ein Query = teure State-Changes
 - Vorher: das Schreiben in Color- und Z-Buffer abschalten
 - Nachher: wieder einschalten
 - Dieser Overhead kostet mehr Zeit als der eigentliche Query!
- Idee: Batching
- Führe 2 zusätzliche Queues ein
 - Beide enthalten Objekte, die auf Visibility getestet 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

G. Zachmann Computer-Graphik 2 — SS 10 Real-time rendering 53

▪ Beispiel: jede Farbe = ein State-Change



Naiv

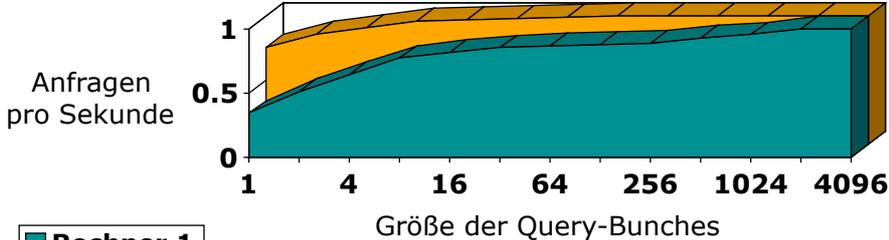


CHC++

G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 54

▪ Bemerkung: Anfrageergebnisse abholen erfolgt **synchron!**

▪ Schicke deshalb einen Folge von Anfragen, lese erst *danach* das Ergebnis der Folge



Anfragen pro Sekunde

Größe der Query-Bunches

■ Rechner 1
■ Rechner 2

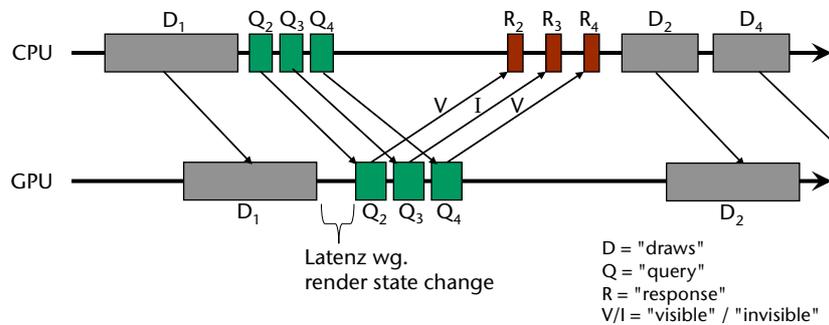
G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 55

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"

Sortieren der Objekt-Liste

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

worst case: 4 3 2 1

best case: 1 2 3 4

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

G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 58

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 \geq Threshold
 - Eventuell „kleine“ Löcher in einem oder zwischen Objekten

G. Zachmann Computer-Graphik 2 – SS 10 Real-time rendering 59

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 (initially 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 num. visible pixels of query V > threshold:
        V.obj = "visible"
        if V.obj is not marked "previously visible":
            render V.obj
    else:
        V.obj = "invisible"
end while
until Q empty and L empty
```

Im Folgenden: schrittweise Verbesserung dieses Algorithmus'