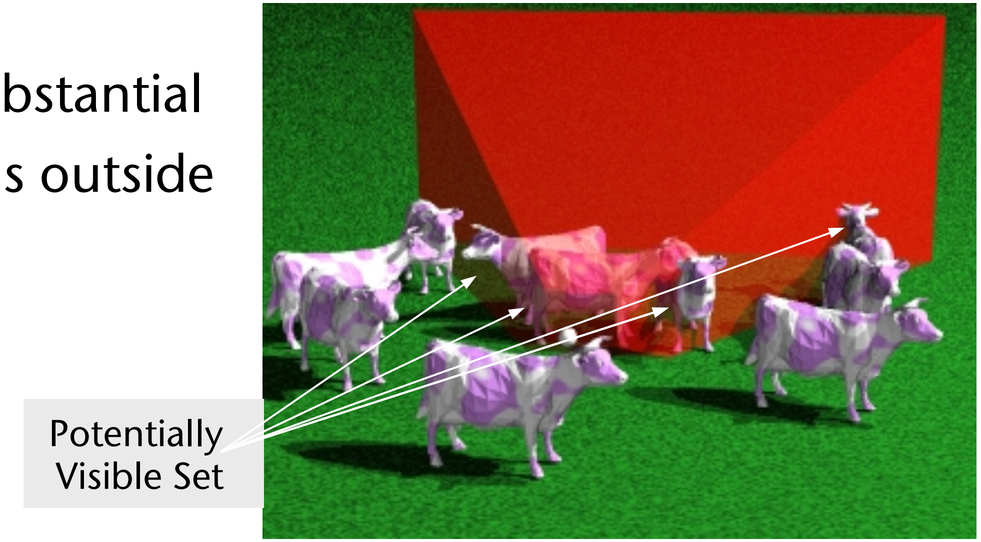
View-Frustum Culling

- In many real scenes, a substantial percentage of the scene is outside the view frustum



Bounding Volumes (BVs)

- Test per polygon is too expensive, overall rendering time would be slower than without VFC
- Therefore, test complete objects (= set of polygons) whether they are outside the view frustum
- Do fast tests with simple *bounding volumes* (BVs):

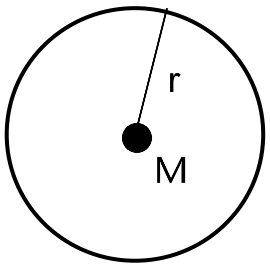
- The process is efficient only if

  Cost(BV test) << Cost(rendering the polygon set)

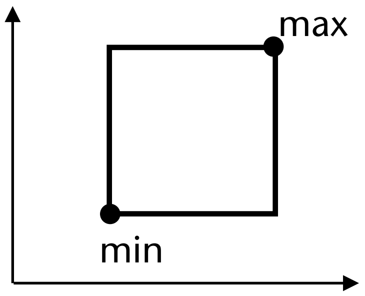
Calculation of OBBs

Representation of BVs

- Sphere := (center, radius)

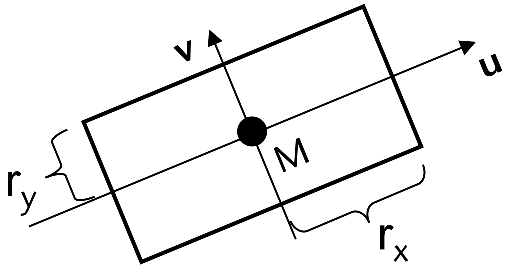


- AABB := (min, max) = $(x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max})$



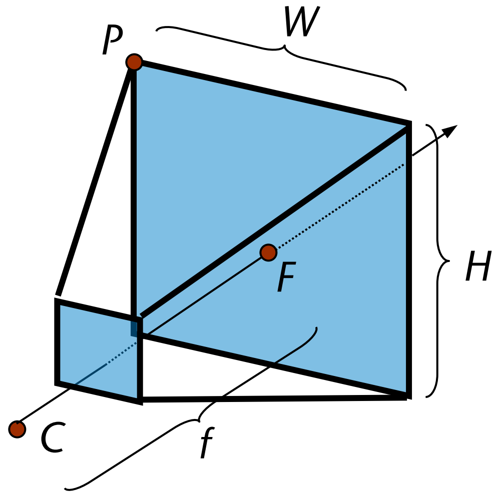
- OBB is defined by
  - center
  - 3 axes
  - 3 „radii“
  - Corresponds to a 3x4 matrix:

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



Representation of the View Frustum

- Procedure:
  1. Use parameters from gluPerspective and gluLookAt
  2. Calculate vertices of the frustum
  3. Calculate the frustum planes



- Determine corners (in world coordinates):

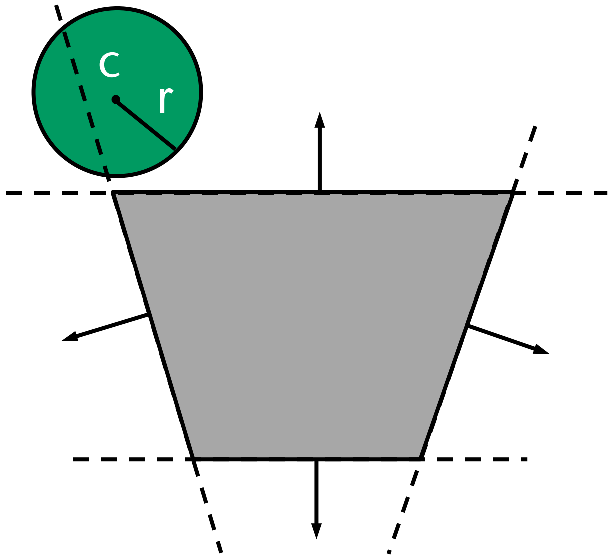
$$F = C + f \cdot \mathbf{d}$$

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

  Analogously, calc all other vertices

- Determine the planes of the corners :
  - 3 points are sufficient (cross product of edges)
  - Note: ensure a consistent orientation of the normals!
  - Small optimization: normals of the near and far plane are known already

Test Sphere v. Frustum For View Frustum Culling



- Given: 6 plane equations

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

  and a sphere

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

- Question: Is the sphere is completely outside the frustum?
- Yes ↔ $\exists i : \quad c \cdot n_i - d_i > r$
- If $\exists i : \quad |c \cdot n_i - d_i| \leq r$

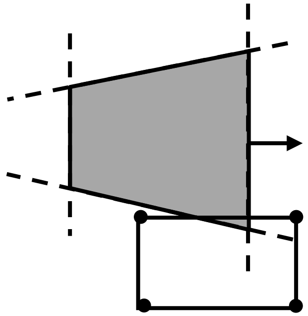
  then one of the planes intersects the sphere (but not necessarily the frustum)

- If $\forall i : \quad c \cdot n_i - d_i < -r$

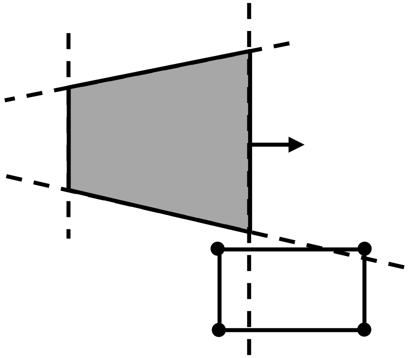
  then the sphere is completely inside the frustum

Test Box v. Frustum

- Warning: it is not sufficient to check that all vertices are outside the frustum!
  - Counterexample:



- A simple, conservative test: All 8 vertices are on the positive side of the *same* plane → box is outside



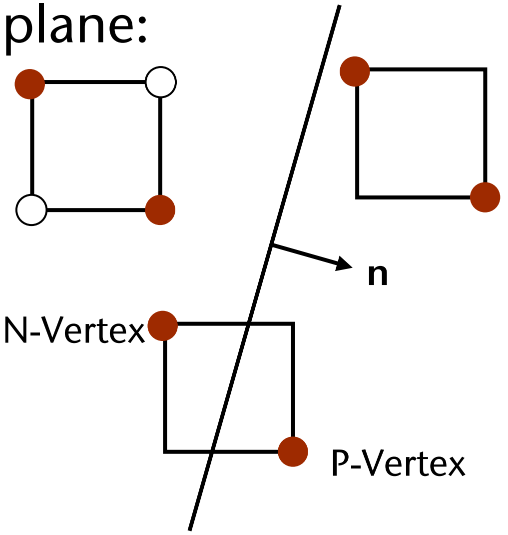
- This test produces so-called "false positives" ⟶ increases the PVS
- The box is completely inside ⇔ all vertices are on the negative side of all planes

Optimizations

- It is sufficient to test two corners against each plane:
  - We denote by *"N vertex"* that vertex of all vertices where the following function assumes the minimum:

$$f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{n} - d$$

  - Analogously define *"P vertex"* (f assumes max)
  - These are (almost always) unique because f is monotone, and a box is convex



```
loop over all planes i:
   calculate fᵢ(N vertex)
   if N nertex is on positive side:
      → complete box is on the positive side
      → complete box is outside the frustum
   calculate fᵢ(P vertex)
   if P vertex is on the negative side:
      → complete box is on the negative side
```

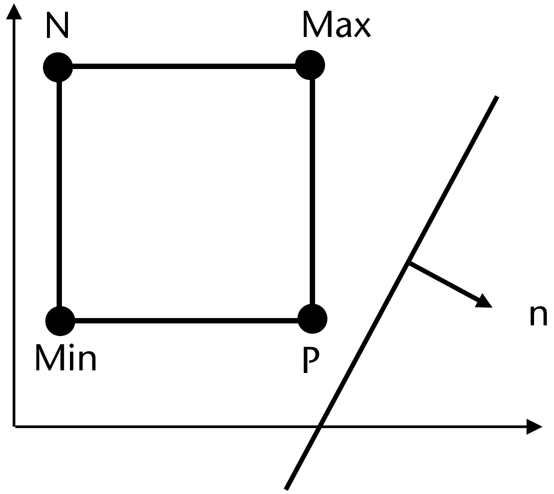
- How to *quickly* find the N- or P vertex?
- If box = *axis-aligned bounding box (AABB),* then it can be done very fast
- AABB = (x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max})

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

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

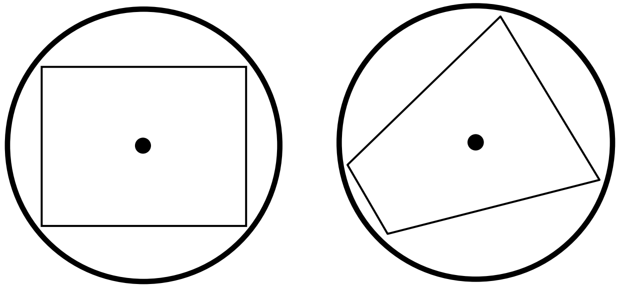
$$P_z = \ldots$$

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

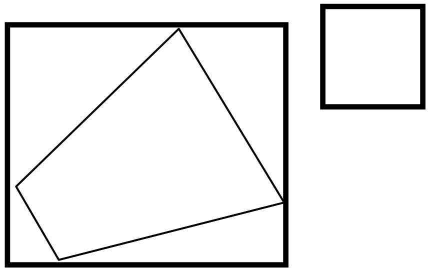


Further Optimizations

- "Meta-BVs": If many boxes need to be tested, enclose boxes and balls in a frustum



- Or enclose the frustum in an AABB, too



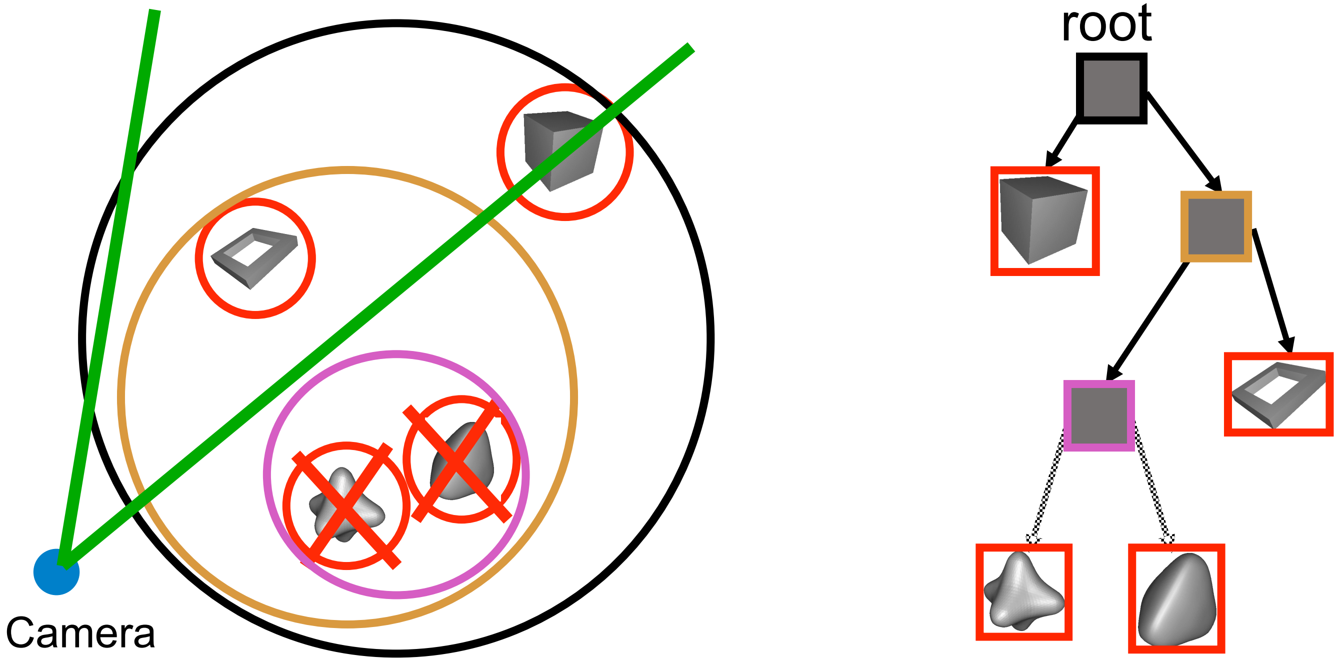
- Produces more false positives, so YMMV
- Exploit temporal coherence: if box has been culled by a certain frustum plane, save that plane and test this *first* in the next frame. Probability is high that this plane culls the box again!

Hierarchical View Frustum Culling

- Generating at each node of the scene graph a bounding volume including the complete subtree →

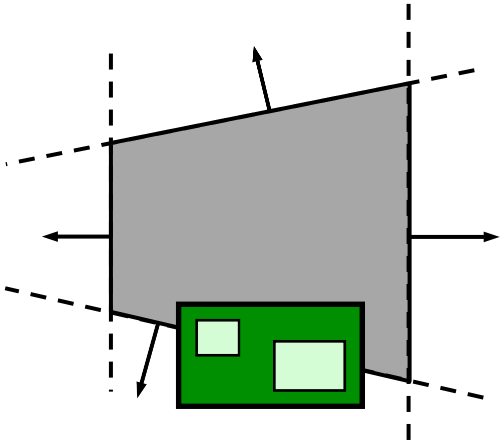
  Bounding-Volume-Hierarchy (BVH)

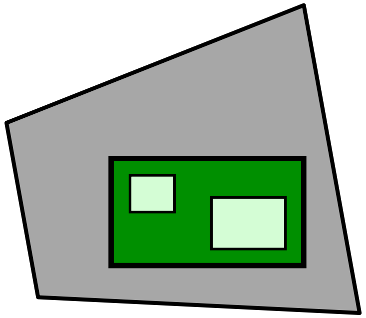
- Traverse this BVH und test all knots



Further Optimization

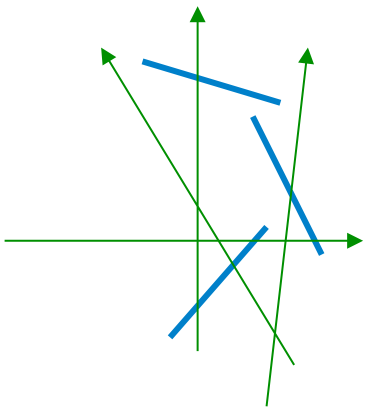
- *Plane Masking*:
  - If a box is completely on the negative side of a plane, then all children too. Do not test this level for the children
  - If a BV is completely inside, then all children are inside

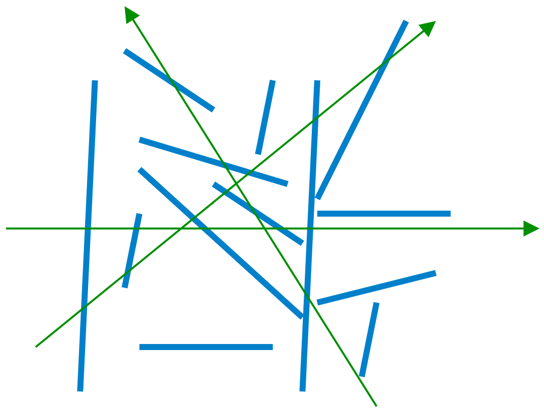




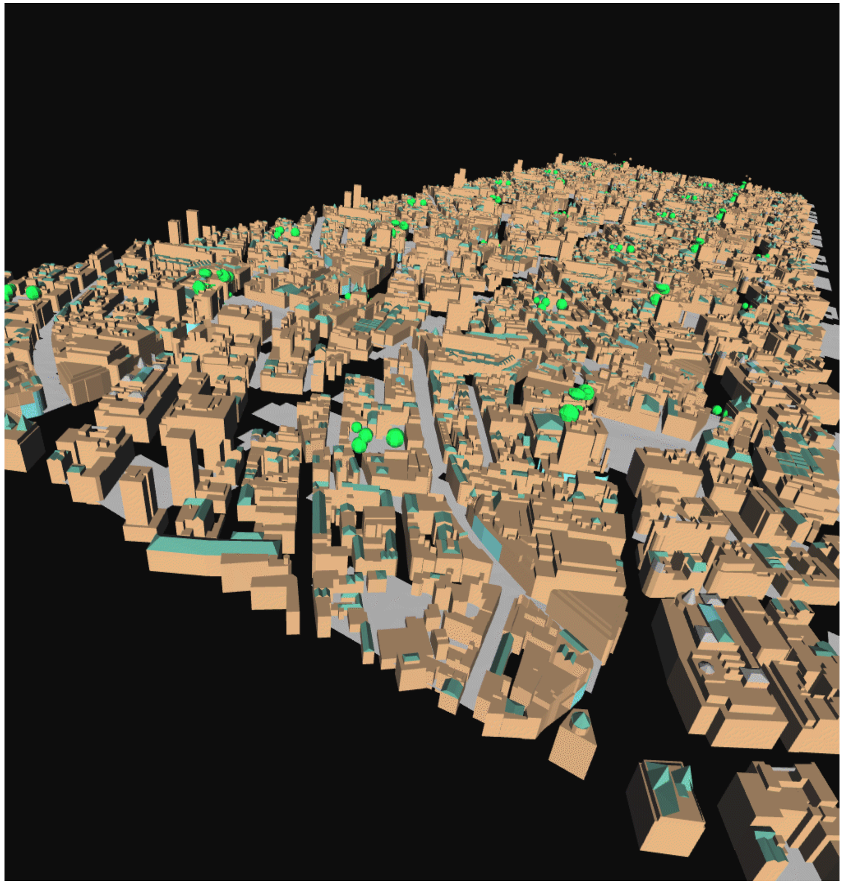
Occlusion Culling

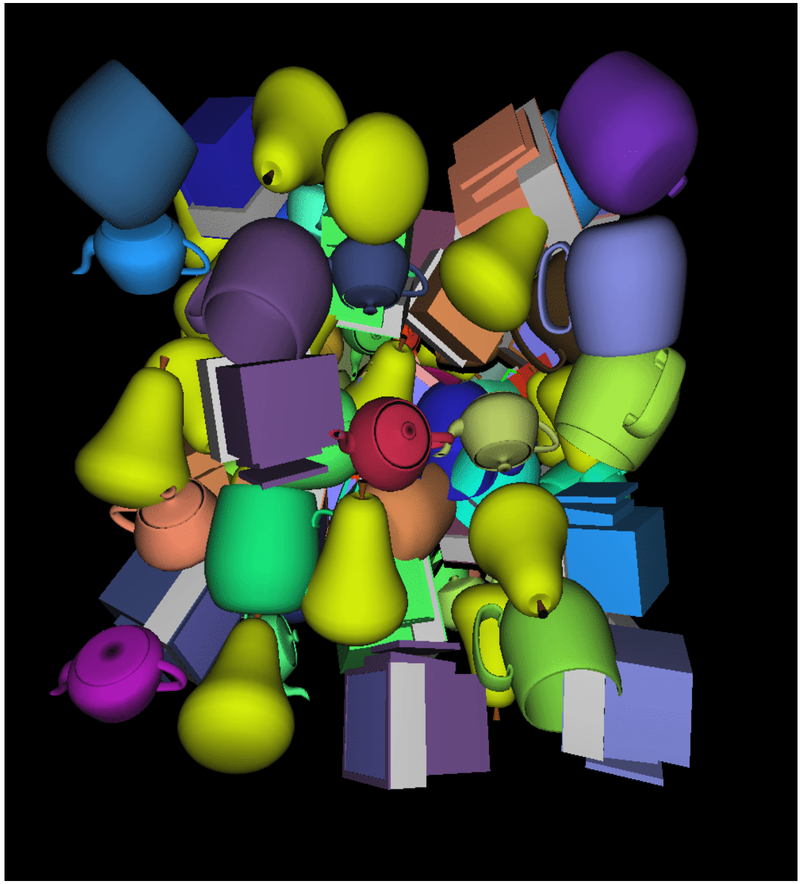
- Occlusion Culling is always interesting, if many objects are hidden by a few objects
- Definition: **Depth Complexity**
  - Number of intersections of the ray in the scene
  - Number of polygons projected on a pixel
  - Number of polygons that would be visible at a pixel, all polygons were transparent
- Comment : Depth Complexity is oberservation and directional





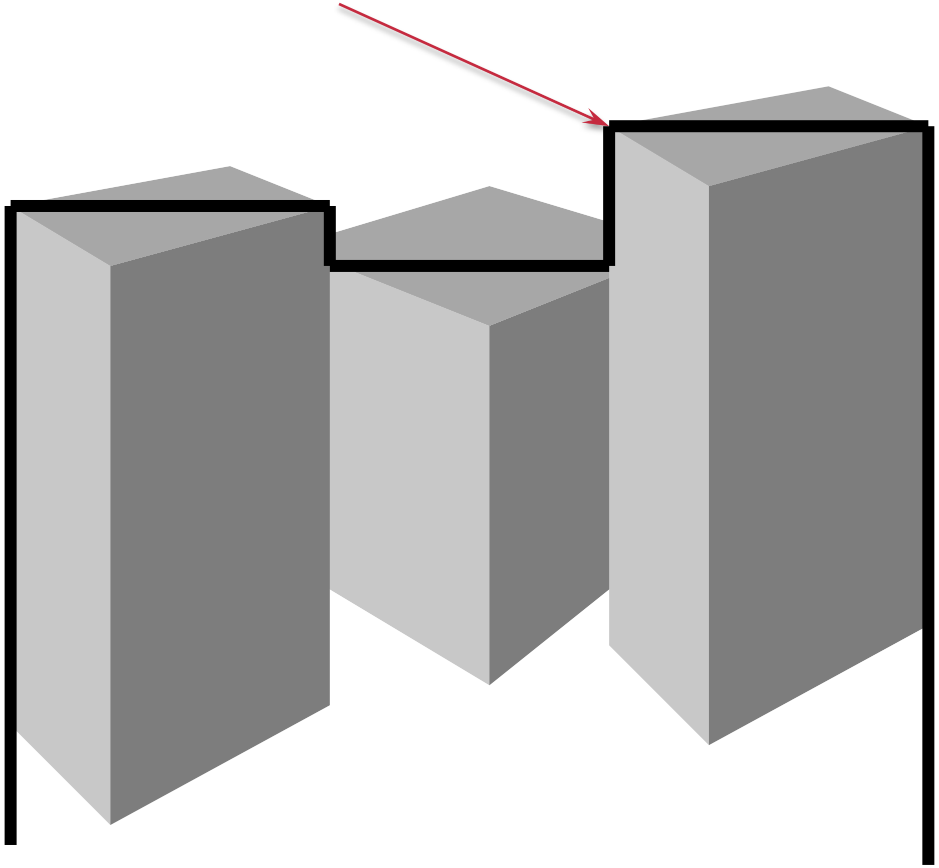
Examples of High Depth Complexity



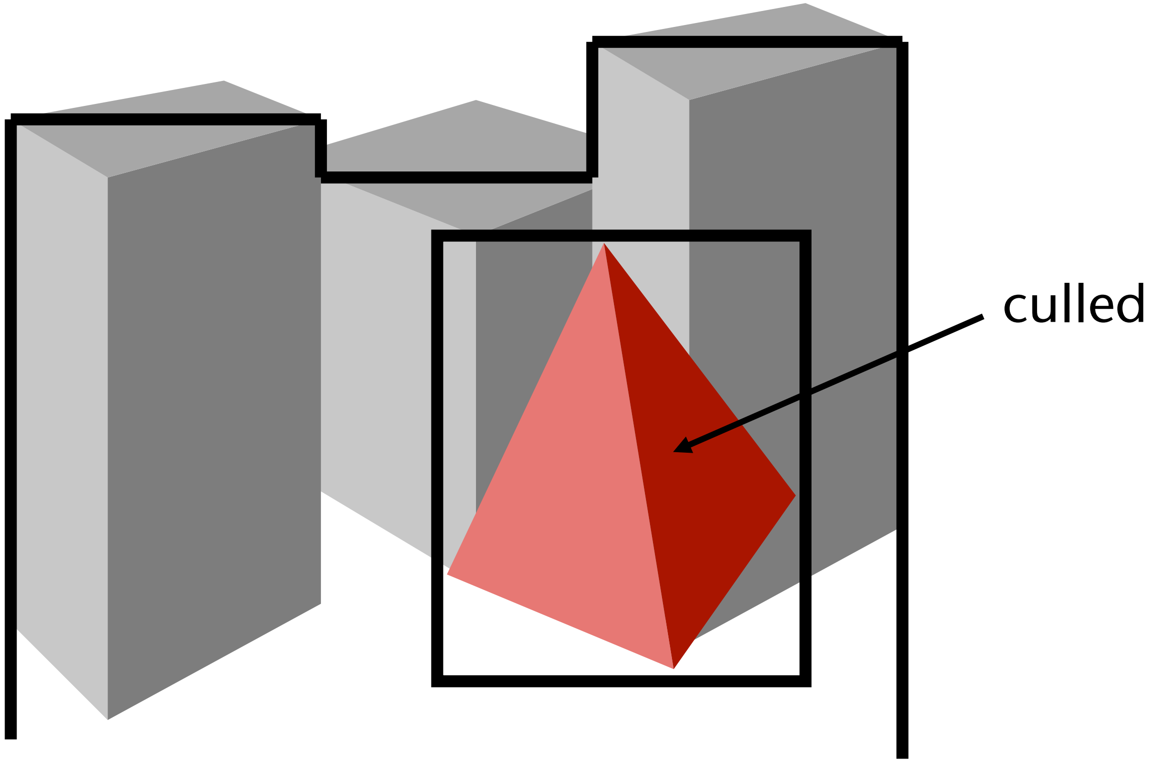


First, the Special Case of “Cities"

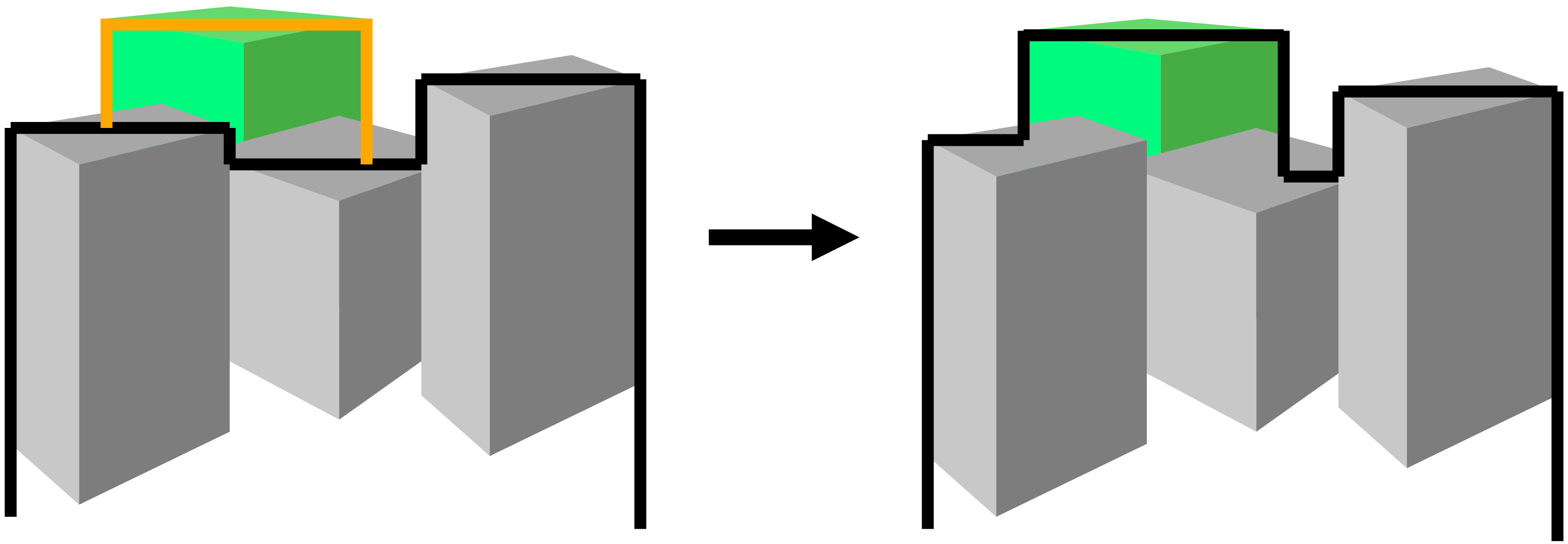
- Render the scene from front to back (reverse Painter's Algorithm)
- Generate an "occlusion horizon"



- Rendering an object (here tetrahedra; behind the gray objects):
  - Determine axis-aligned bounding box (AABB) of the projection of the object
  - Comparison with the occlusion horizon

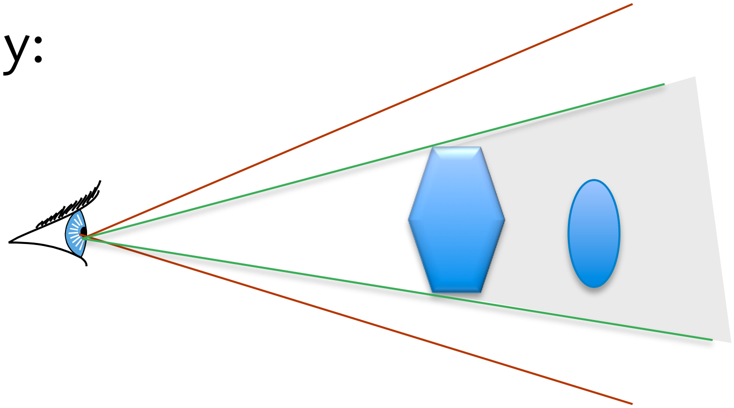


- If an object is considered as visible:
    - Add the AABB with the previous occlusion horizon



General Occlusion Culling

- Given:
  - A partial(!) rendered scene, and
  - not yet rendered object
- Task:
  - Decide quickly whether the object would modify pixels in the frame buffer, if it were rendered;
  - I.o.w.: decide quickly whether the object of the current scene is completely covered
- Terminology:

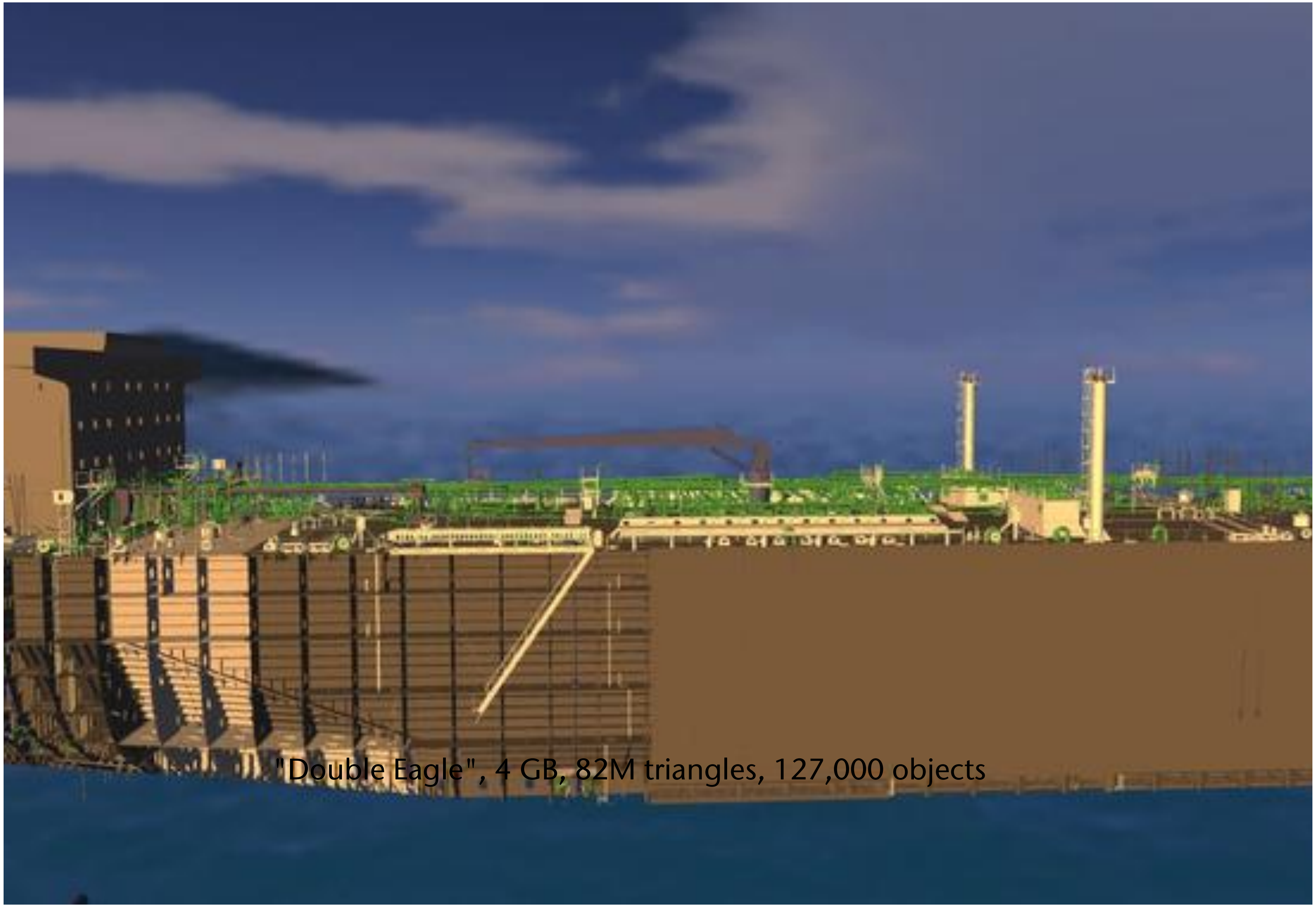


Occluder

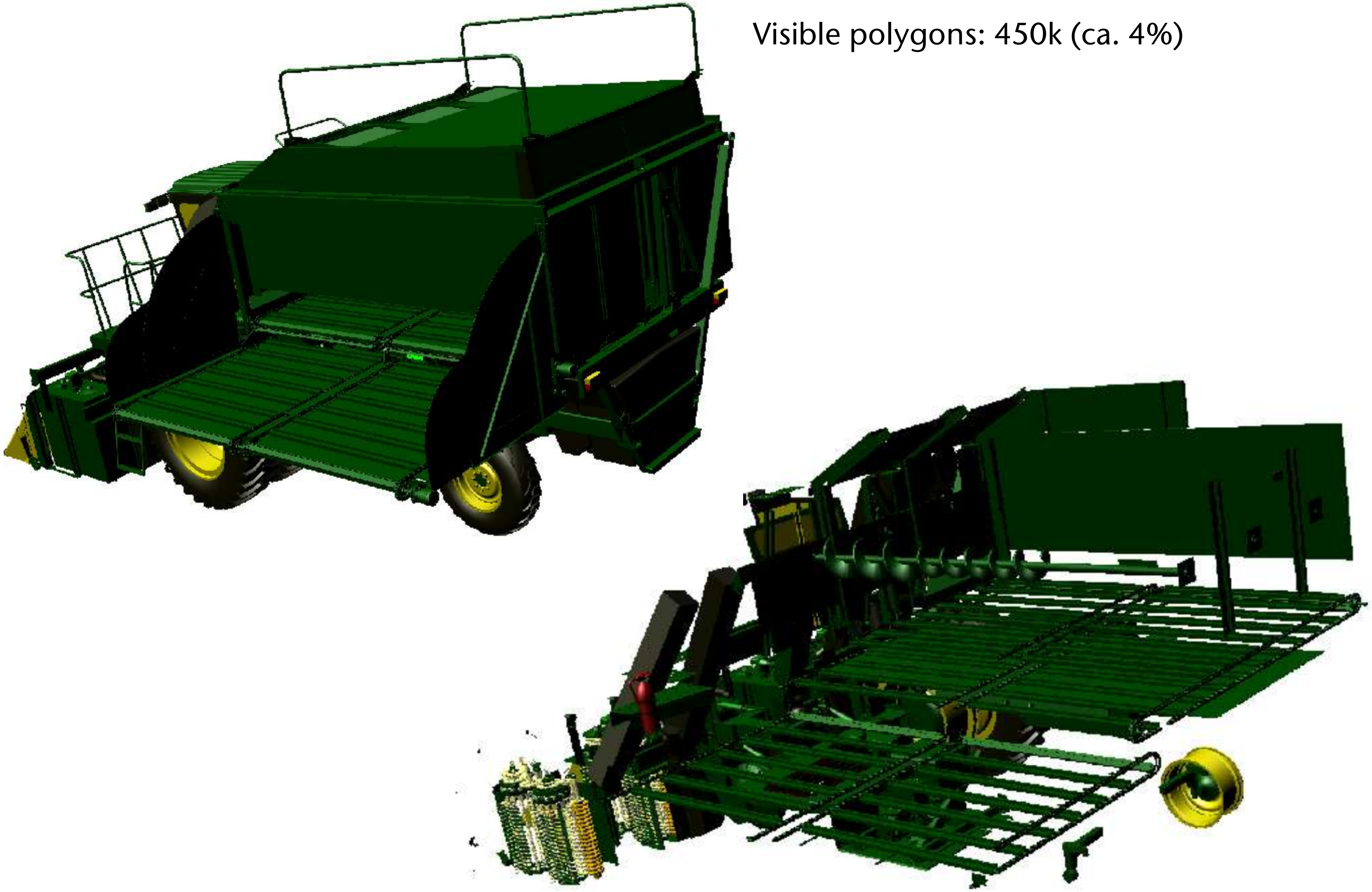
Occluded geometry ("occludee")

Examples of Applications of the General Occlusion Culling

Power plant, 13 million triangles



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



Visible polygons: 450k (ca. 4%)

Invisible polygons: 10M (ca. 96%)

Occlusion-Culling in OpenGL

- Earlier as extension **`ARB_occlusion_query`** , nowadays in OpenGL core from version 1.5
  - Operating mode: Asks OpenGL how many pixels would be "repainted" from a batch
- Appendage: Draw a simple representation ("Proxy"), without changing the color or depth buffer
  - Were no pixels drawn by the proxy, the object itself must not be drawn
- Proxy geometry: first sacrifice a little computing power to possibly save a lot of computing power later
  - Tolerably accurate bounding volumes
  - No texturing, no shading, no light sources
  - No colors, texture coordinates, normals

- First create occlusion query at initialization :

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

- Render a set of objects (hiding a lot)
- Disable writing in Z- and color buffer (optional):

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

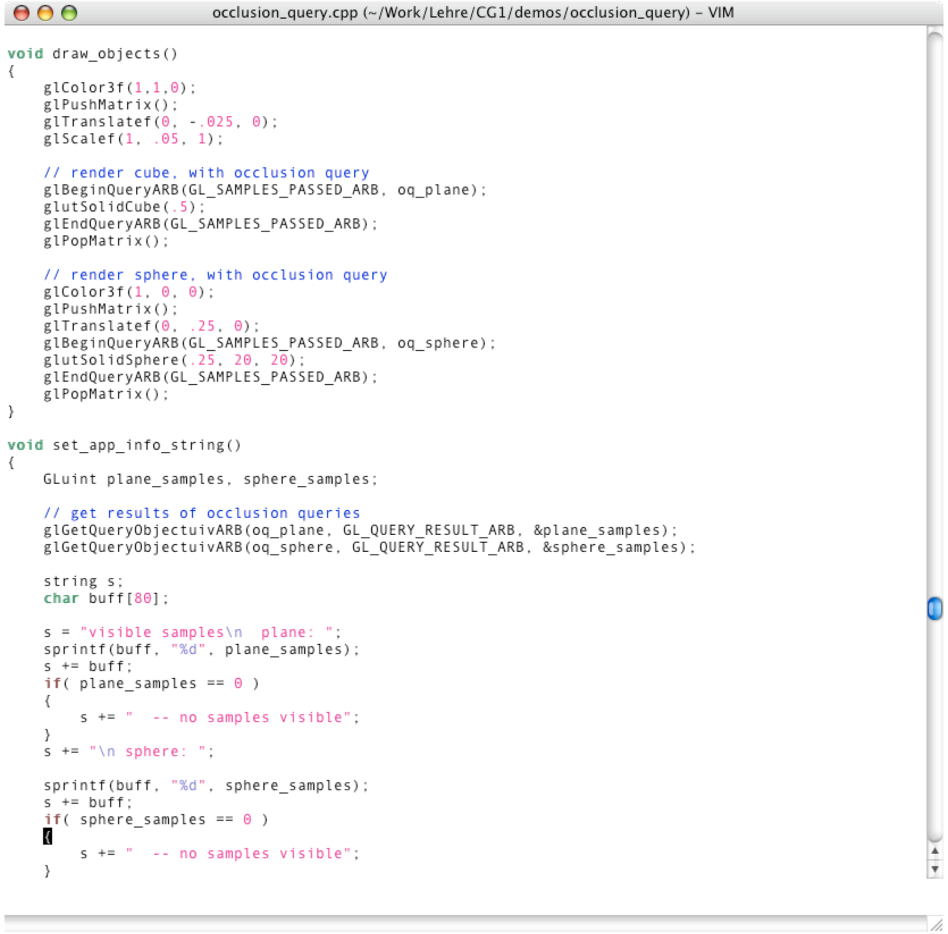
- Create request for a lot of other objects :

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

- Reading result of the request:

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

Demo

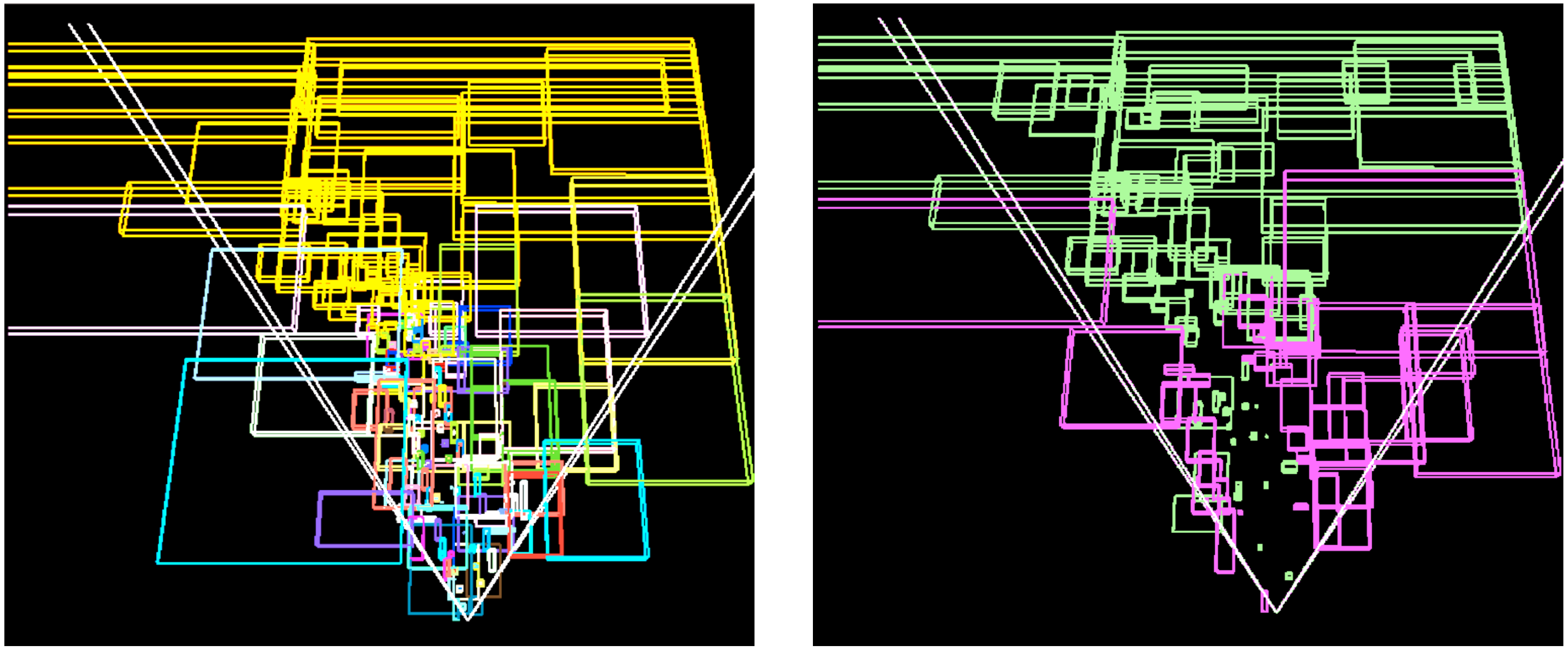


```cpp
void draw_objects()
{
    glColor3f(1,1,0);
    glPushMatrix();
    glTranslatef(0, -.025, 0);
    glScalef(1, .05, 1);

    // 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, 20, 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";
    }
```

Batching Queries

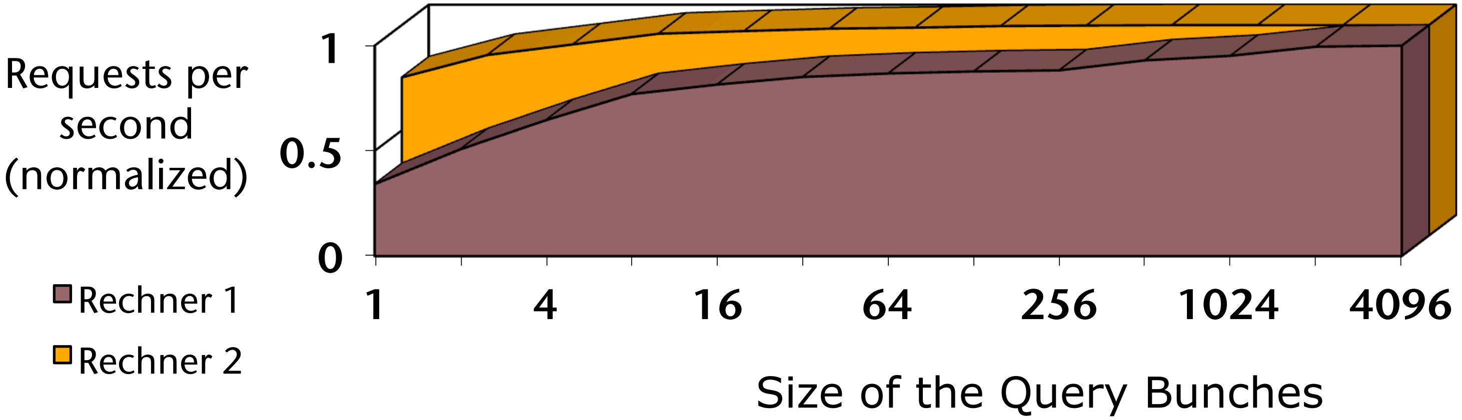
- Problem: A query = expensive State-Changes
  - Before: Disable writing to color-and Z-buffer
  - After: Switch back
  - This overhead takes more time than the actual query!
- Idea: Batching
- Implement 2 additional queues
  - Both contain objects that should to be tested for visibility
  - I-Queue: contains previously “invisible” objects
  - V-Queue: likewise for "visible"
  - Parameters: Batch size *b* (ca. 20-80)
  - Principle: only if batch size is reached, the list of queries is sent to OpenGL
- "Previously visible" objects are still rendered immediately

- Exemple: each color = one state change



Naive CHC++

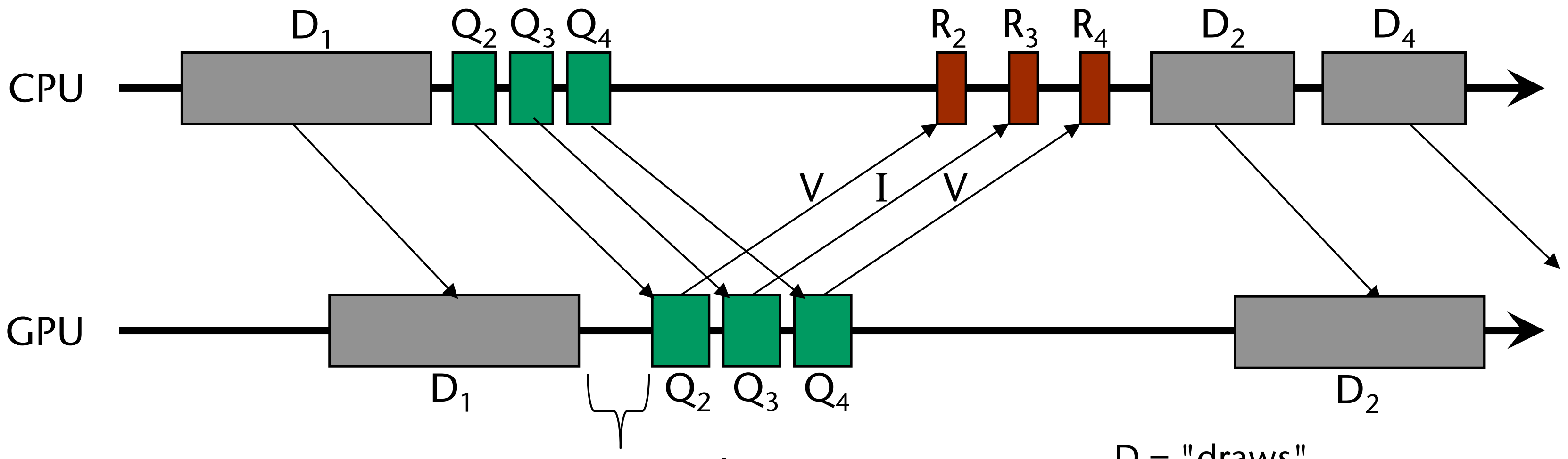
- Goal: Reduce the number state changes, and thus the time required per Occlusion Query
- Therefore, send a sequence of requests, read the result of the sequence afterwards



The Naive "draw-and-wait" Approach

```
Sort items about the depth in
Create query sequence
while some objects are not rendered:
  For each object in query sequence:
    BeginQuery
    Render bounding volume
    EndQuery
  For each object in query sequence:
    GetQuery
    if #pixel drawn > 0:
      Render object
```

- Problems of the naive approach:
  - Very high response time (latency) for a query:
    - long graphics pipeline,
    - some time by the execution of the queries (rasterization), and
    - transfer the result back to the host.



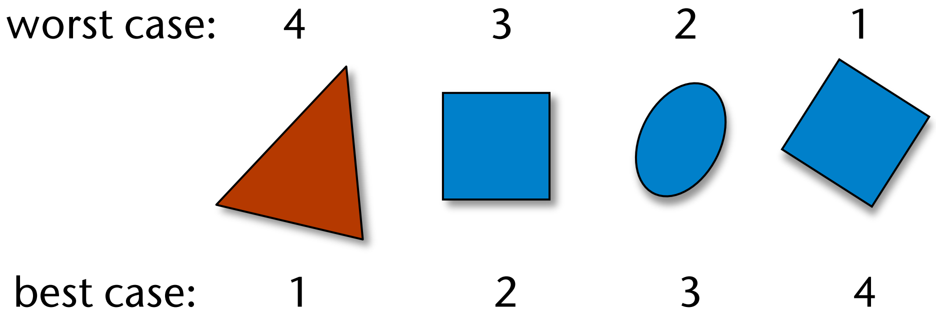
Latency due to
render state change

D = "draws"
Q = "query"
R = "response"
V/I = "visible" / "invisible"

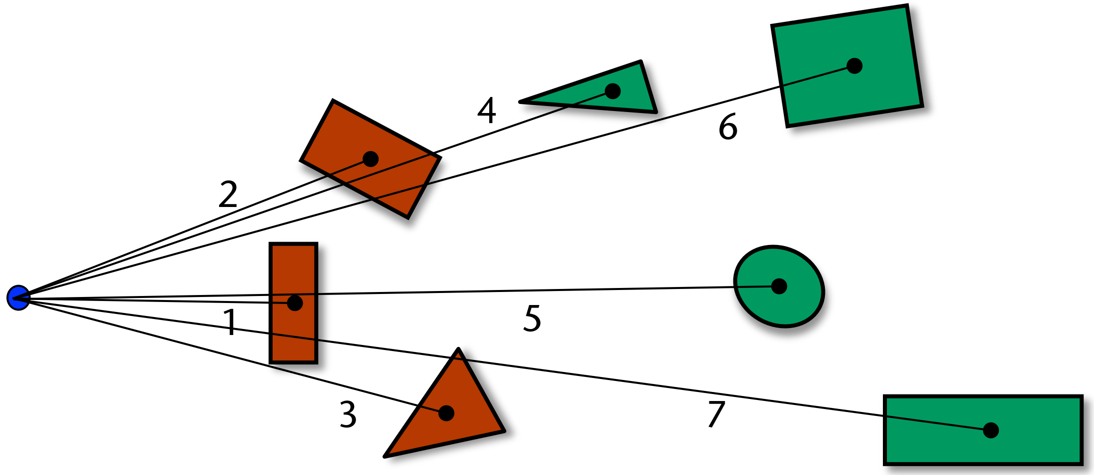
- Sequence: "CPU stalls" and "GPU starvation"

Sort the Object List

- Observation: Depending on the order in which you render the objects, you get a high culling rate or not

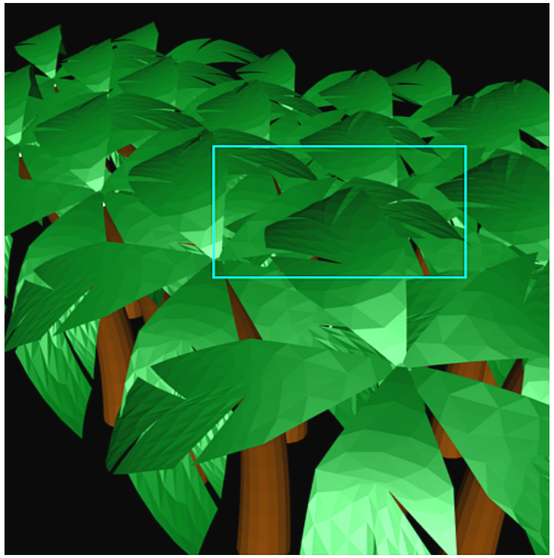


- Solution: sort the list by distance to the object Viewpoint



Aggressive Approximate Culling

- Often only conservative culling :
  - Even if only one pixel of the BVs is visible also one pixel of the object can be visible → draw object
  - Disadvantage: Often outer parts of the BVs are visible where no object pixel are located
- Idea: Ignore barely visible objects
  - Object probably (!) not visible if only a few pixels of the BVs are visible
  - Heuristics: Draw object only if query result ≥ threshold
  - Potentially "small" holes in or between objects



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

- Here in a simplified representation (a.o. without hierarchy)
- Given: Set of objects
  - Here: Object = Amount of useful contiguous polygons
- Ideas:
  - Perform a queue with stored hardware occlusion queries
  - First assumption: if an object was visible in the last frame, it is also visible in the current frame
  - If an object was invisible, first check its visibility
  - Do not wait for result, go further through the query
  - Edit query results as soon as they are available

The algorithm

```
L = list of all objects (incl. BVs)
Q = queue for occlusion queries (initally 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
```

Below: gradual improvement of this algorithm

Fusion (Potentially) Hidden Geometry

- Observation :
  - If we knew that a lot of objects in the current frame is hidden, then we could verify this by exactly one occlusion query
  - Objects that were hidden in many frames are probably obscured in the current frame (*temporal coherence of visibility*)
- Idea:
  - Invent an "oracle" that can predict for a given set of objects with high probability whether the coherence of visibility is satisfied
  - If the probability is high enough, test this set by 1 Query:

```
glBeginQuery( GL_SAMPLES_PASSED, q );
  render BVs with the set of objects ...
glEndQuery( GL_SAMPLES_PASSED );
glGetQueryObjectiv( q, GL_QUERY_RESULT, *samples );
```

This will be called in the following: Multiquery!

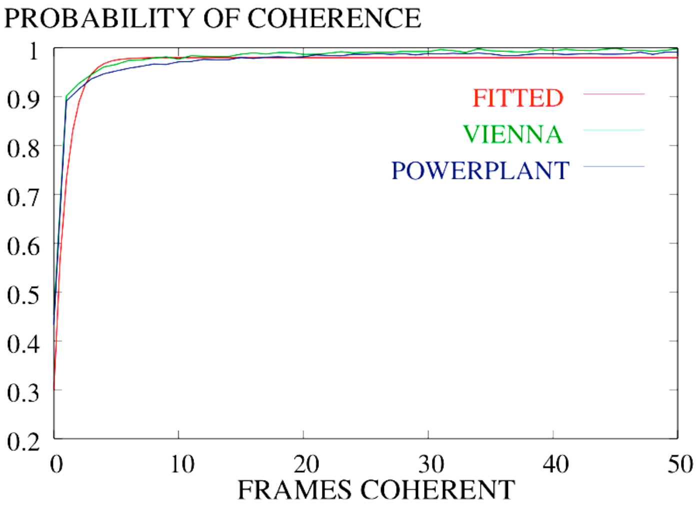
- Definition: Visibility persistence

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

where $I(t)$ = number of objects, which were constantly covered in previous t frames

- Interpretation: p(t) = " probability ", that one object, which was covered t frames, will also be covered in the following frames
- Observation: is amazingly independent from object and scene
- Consequence: can be approximated well by analytic function!

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



- If t_O = Number of previous frames which the object O was covered
- Define an "oracle" for a set M of objects $i(M)$:= the „probability“ that all objects from M in the actual frame will be invisible (only a heuristic!):

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

- Define that
  - Costs of an occlusion multiquery (in the batch):

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

  - Expected benefits of a multiquery:

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

- Thus defining the expected value of a multiquery:

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

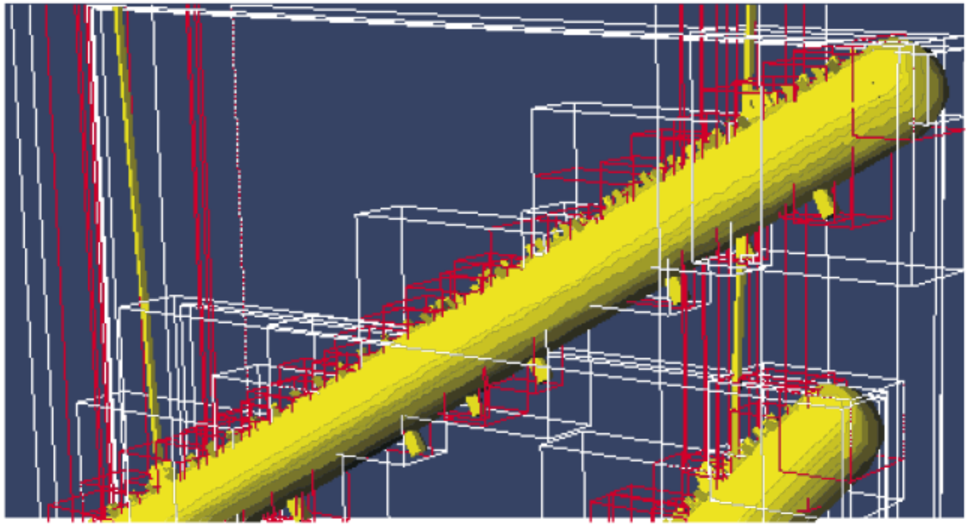
- f the I-queue is full at some point:
  - Sort the objects O_i in the I-queue t_O → $\{O_1, \ldots, O_b\}$
  - Simple greedy search the maximum

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

  - Set on a multiquery for these first n objects from the I-queue
  - Repeat until the I-queue is empty

Tighter Bounding Volumes

- Observation: As greater the BV in relation to the object as more probable that the occlusion query returns a "*false positive*" (claims "visible", but in truth "invisible")
- Objective: close as possible BVs
- Boundary conditions:
  - BVs must be very fast to render
  - BVs may not cost a lot of memory
- Idea:
  - Decompose object into cluster (cluster of polygons)
  - Wrap a BBox around each clustert (AABB)
  - Use as BV foreach object the union of the small BBoxes



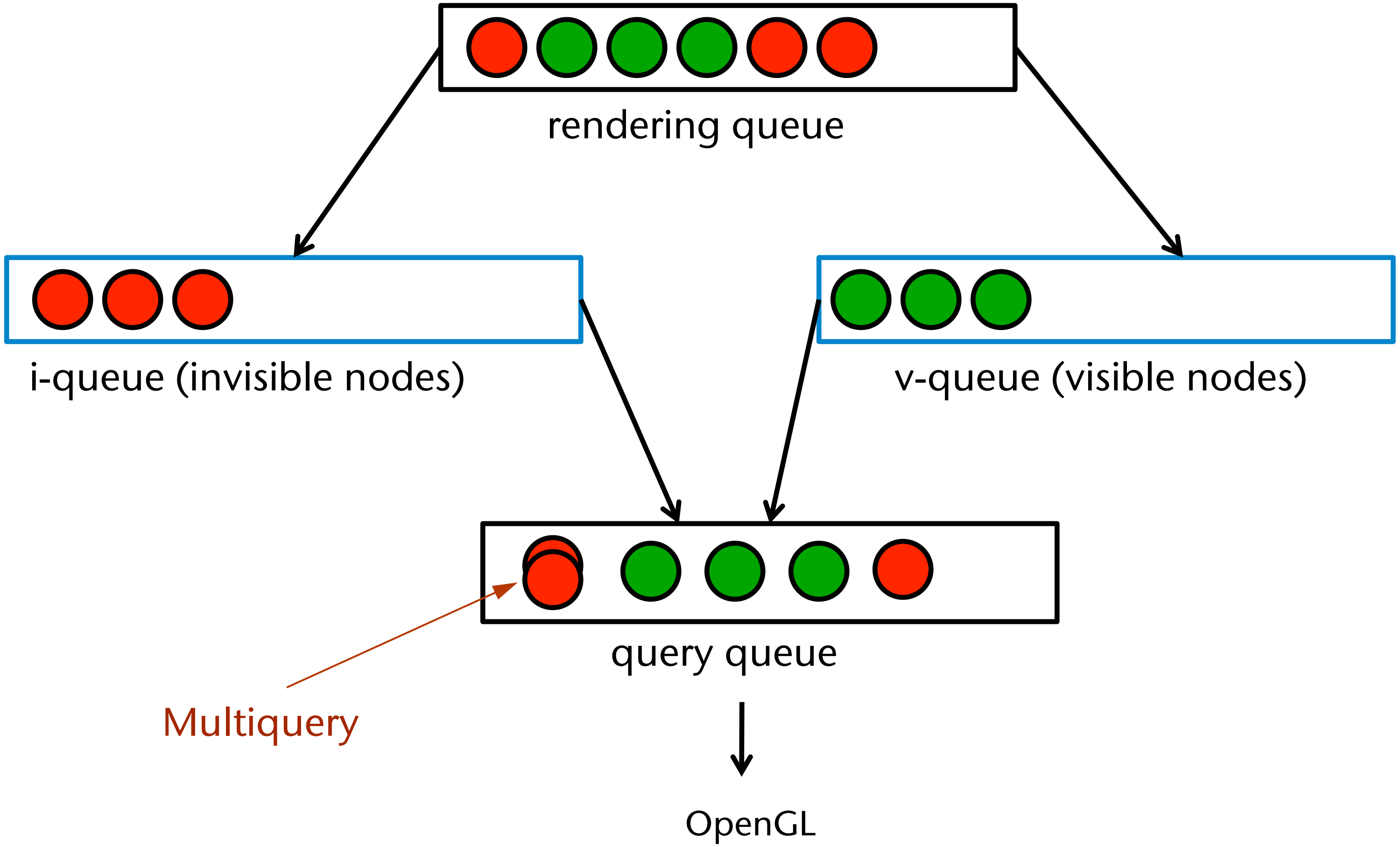
- Question: how small should the "small" AABBs (or cluster) be?
- Observation: the greater the number of small AABBs, ...
  - ... the greater the probability that "invisible" is correctly recognized, but
  - ... the greater the surface → onger rendering time of the resulting occlusion queries
- Strategy for the construction of the "narrow AABBs":
  - Devide clusters recursively
  - termination criterion: if

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

  - Parameter σ depends on the graphics card (σ ≈ 1.4 seems OK)

Altogether

- The queues in CHC++:



rendering queue

i-queue (invisible nodes)

v-queue (visible nodes)

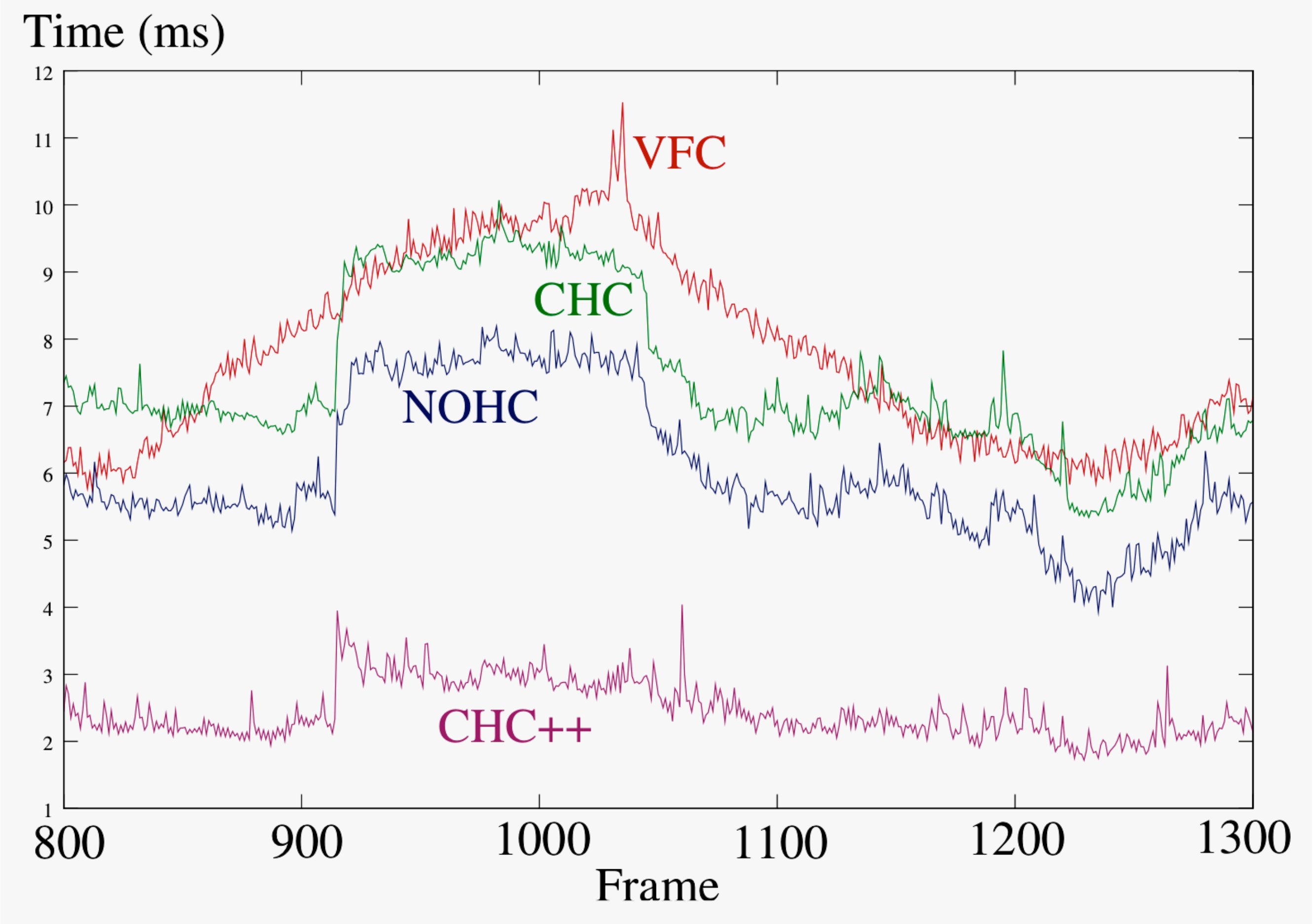
query queue

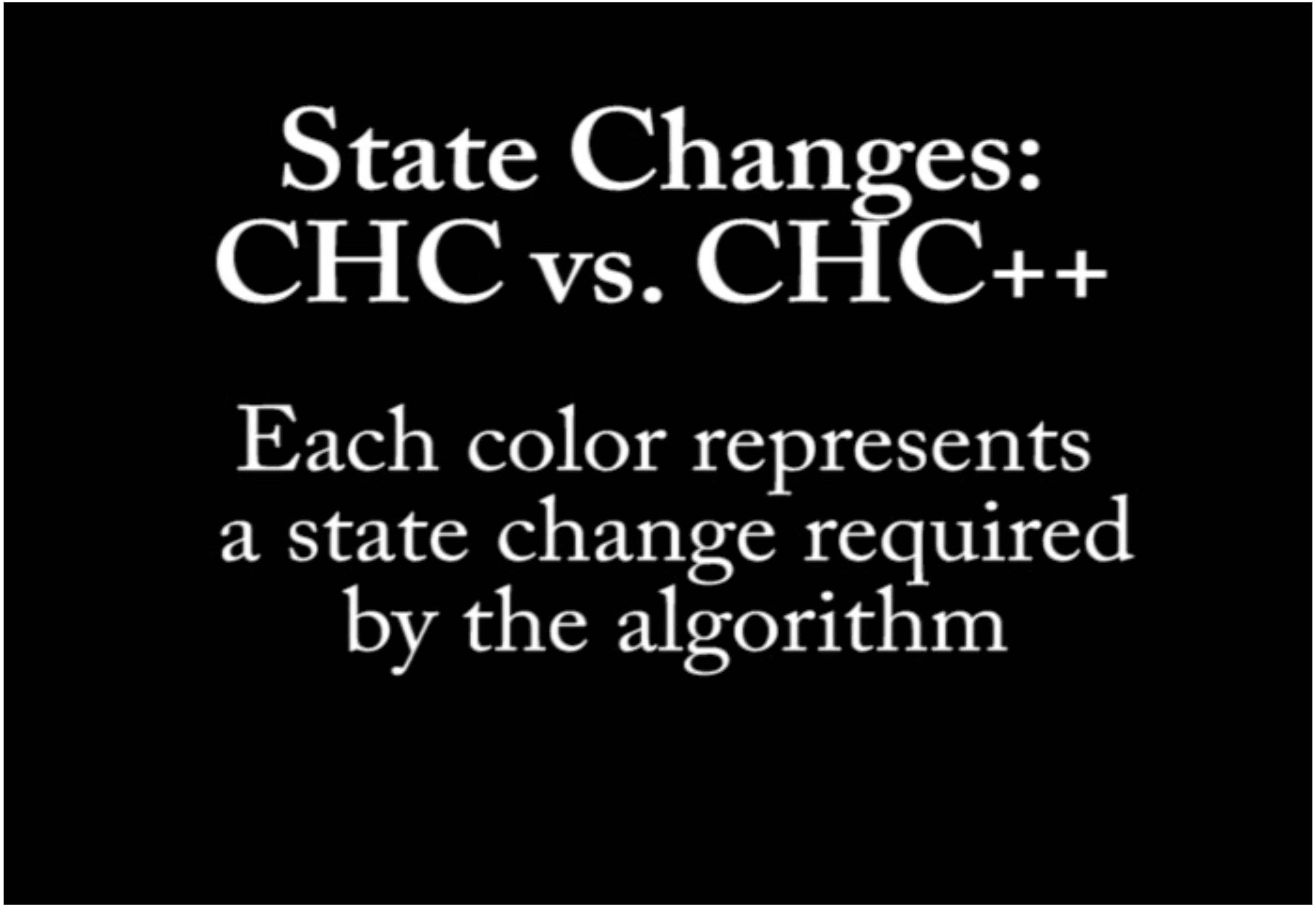
Multiquery

OpenGL

Results

- Walkthrough the power plant model:

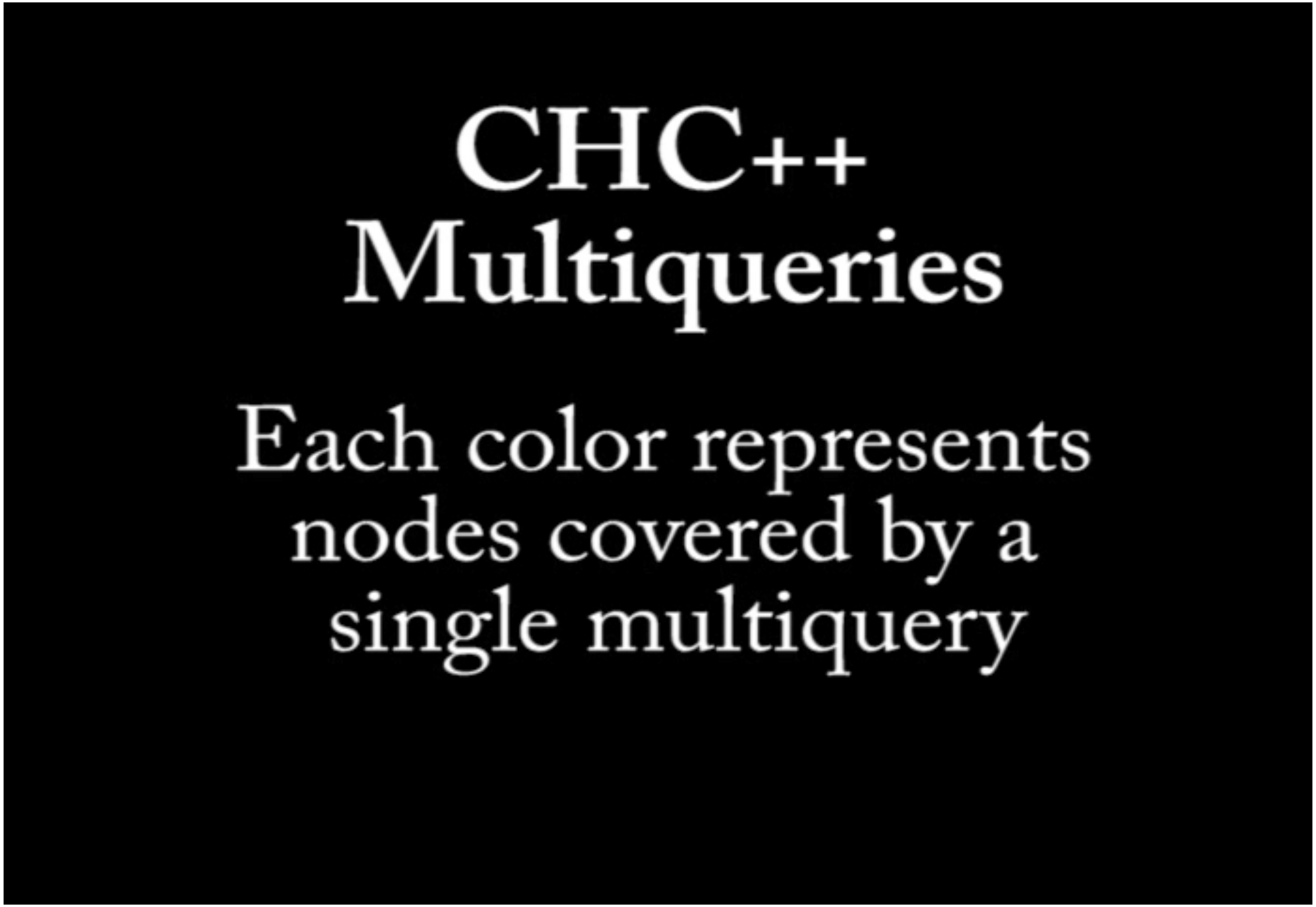




State Changes: CHC vs. CHC++

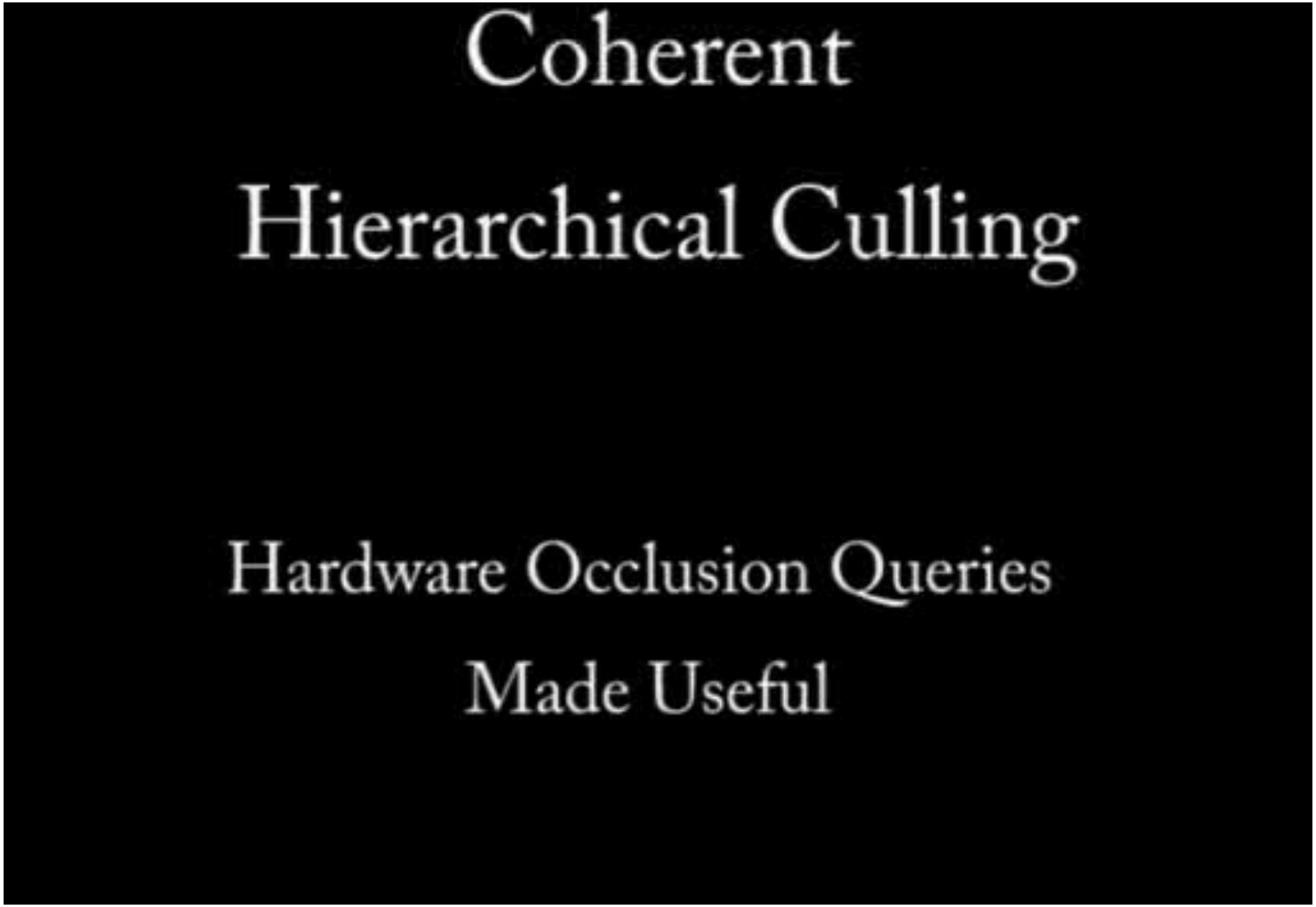
Each color represents
a state change required
by the algorithm

Powerplant
Walkthrough 2



CHC++
Multiqueries

Each color represents nodes covered by a single multiquery



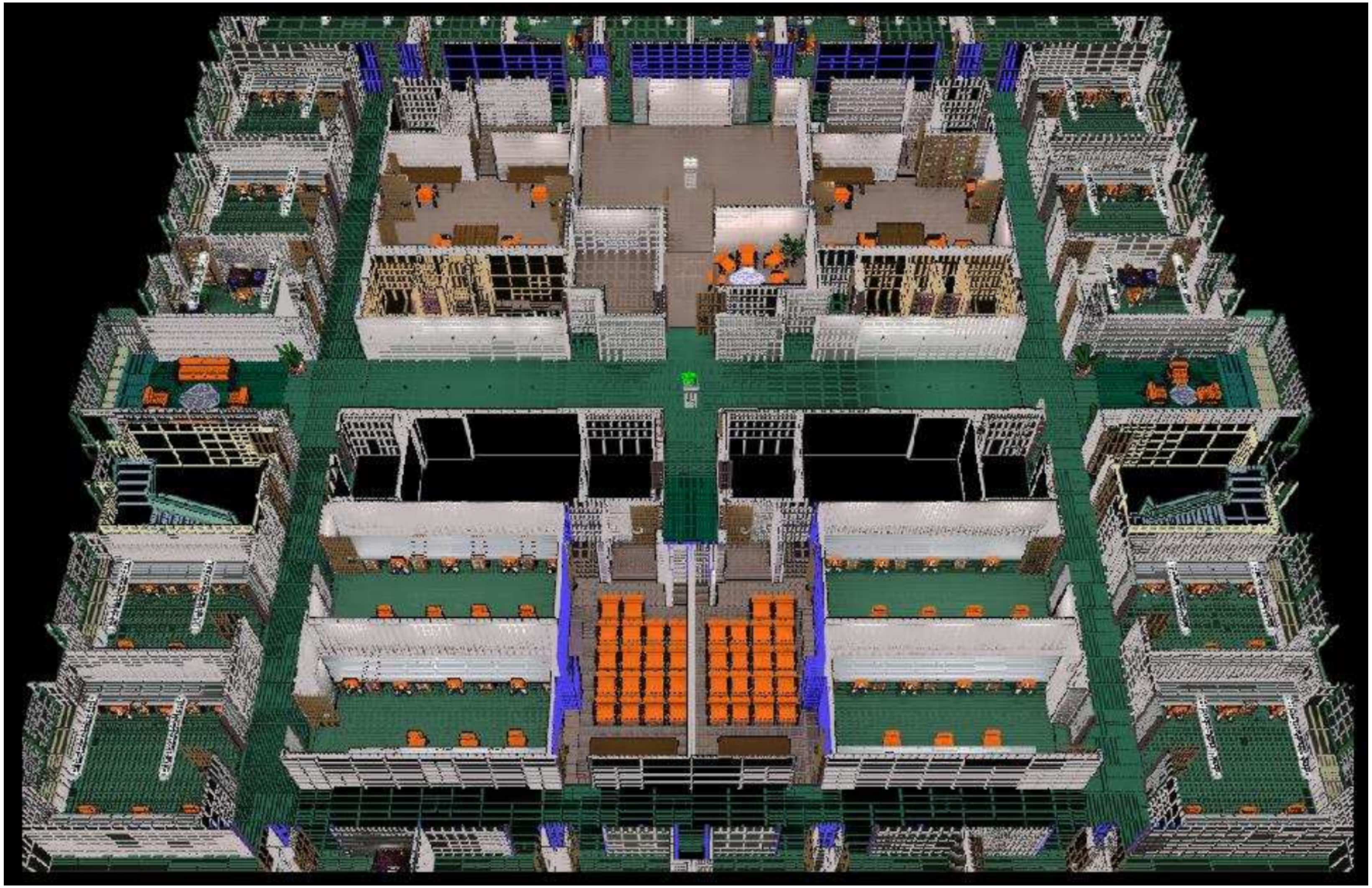
Coherent

Hierarchical Culling

Hardware Occlusion Queries

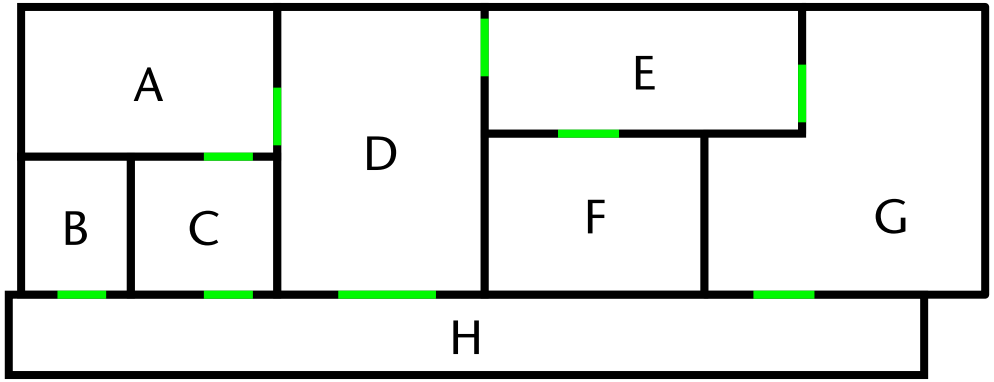
Made Useful

Another Special Case: Architectural Models



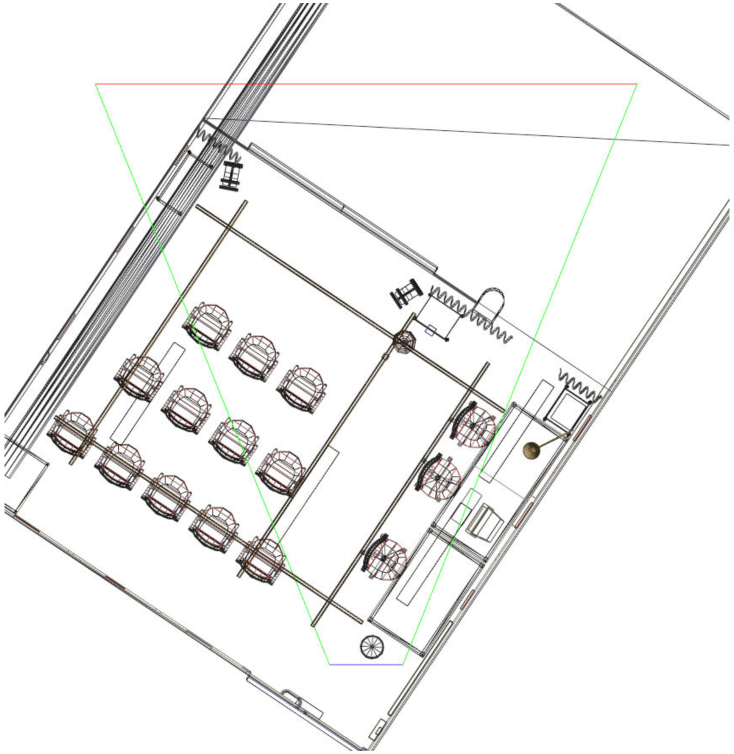
Cells and Portals (*Portal Culling*)

- Scenario: Walkthrough of buildings and cities
- Transparent portals connect the cells
    - doors, windows, holes, …
- Observation: cells see each other only through the portals



- Which cell is included in the PVS?
    - The cell which contains the Viewpoint
    - And these cells, which have a portal to the initial cell

Example scene

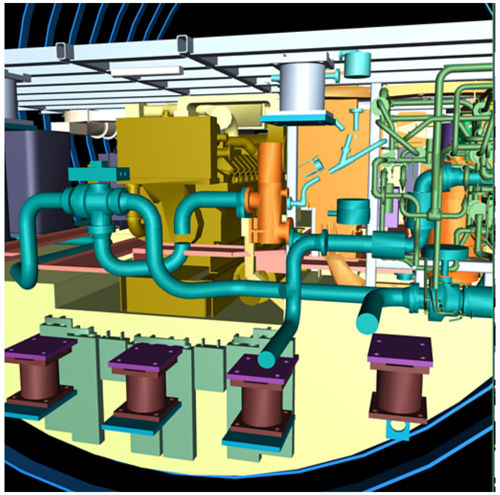


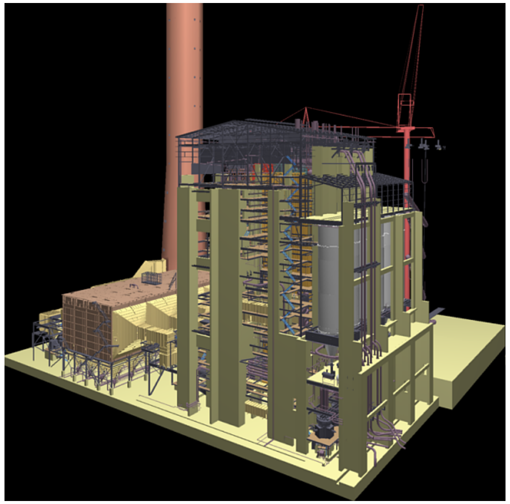
Result

- Example scene :

- Speedup is highly dependent on the model and viewpoint
  - Frame rate is 1-10-fold the frame rate without Cells & Portals method
  - For typical viewports the method removes 20 - 50% from the model

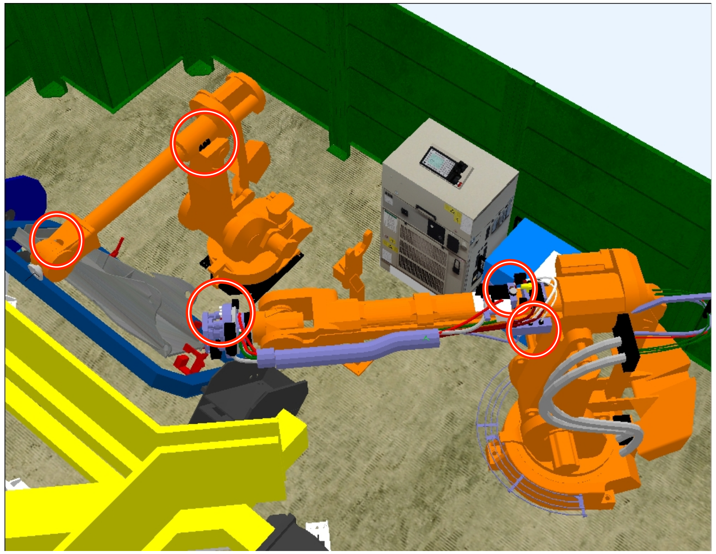
- Field of applications
  - Computer games
  - Buildings
  - Cities
  - Ships (inside)
- Not suitable for CAD data
  - Aircrafts
  - Industrial facilities
- Not suitable for natural objects
  - Plants
  - Forests



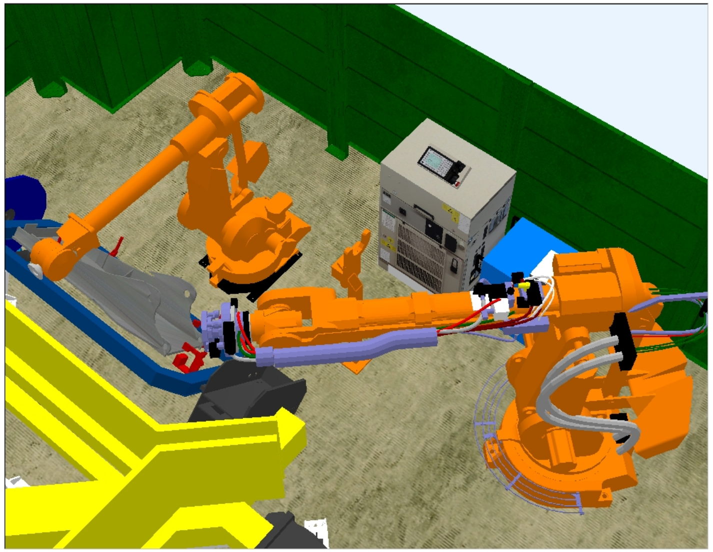


Detail Culling

- Idea: Objects that occupy less than N pixels in the projection are not shown
- This approach also removes parts that would be visible in the final image
- Advantage: trade-off quality / speed



detail culling off

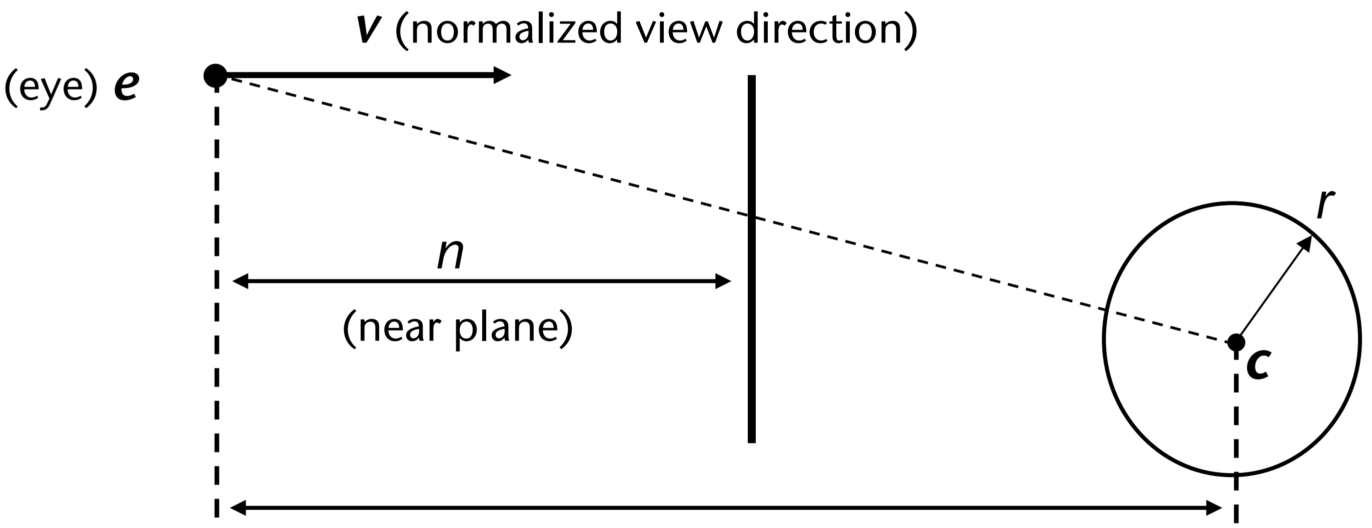


detail culling on

- Particularly suitable for camera motion (the faster, the more details can be culled)

Estimating the Projected Size of an Object

- Estimate the size of the BVs in screen space from:



$d = \mathbf{v} \cdot (\mathbf{c} - \mathbf{e})$ (distance along *v*)

$\hat{r} = r \cdot \frac{n}{d}$ (Estimate of the projected radius)

$\pi \hat{r}^2 =$ estimated area of the projected sphere