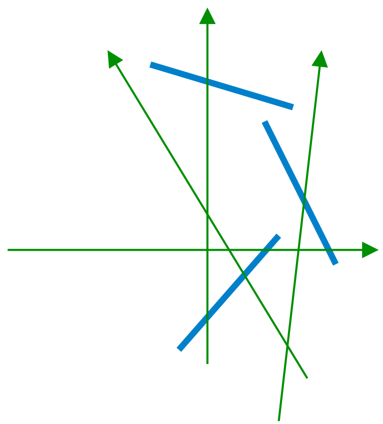
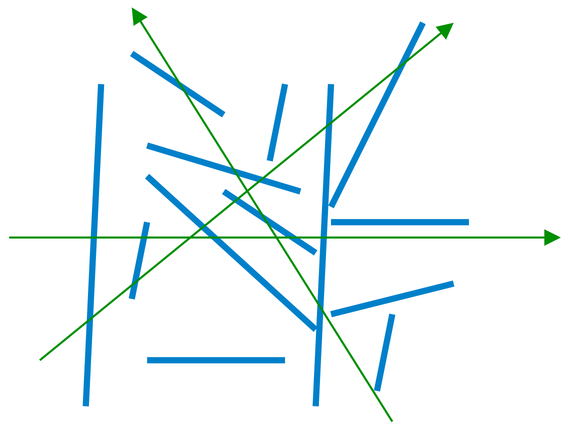
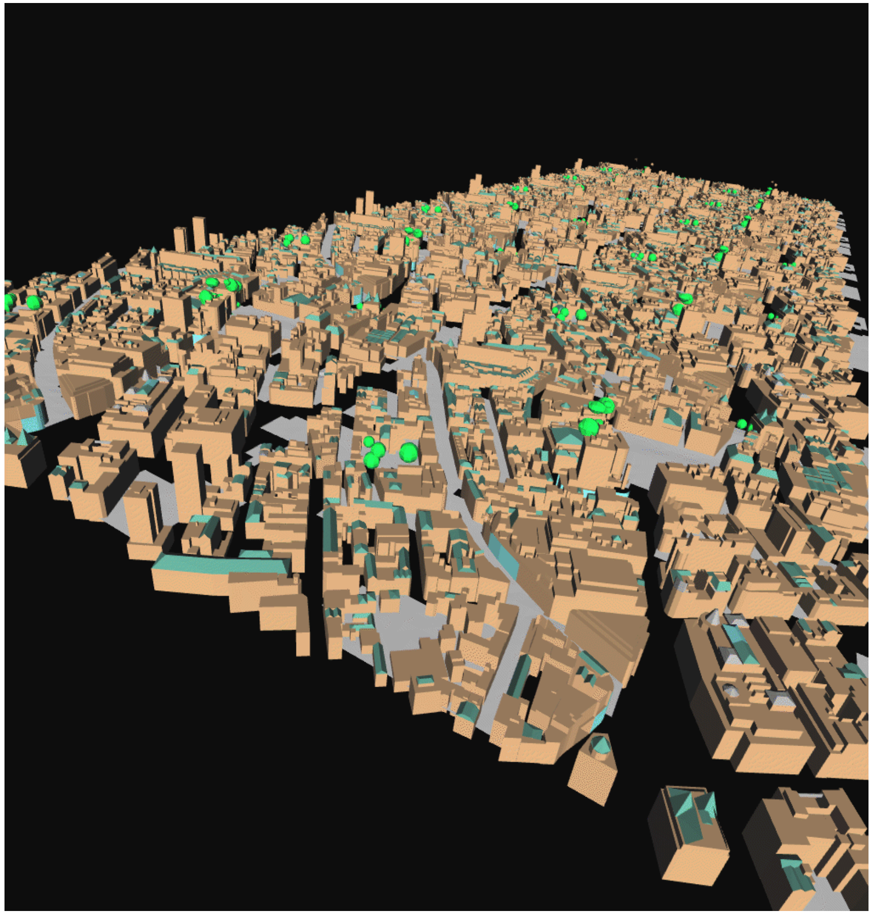
Occlusion Culling

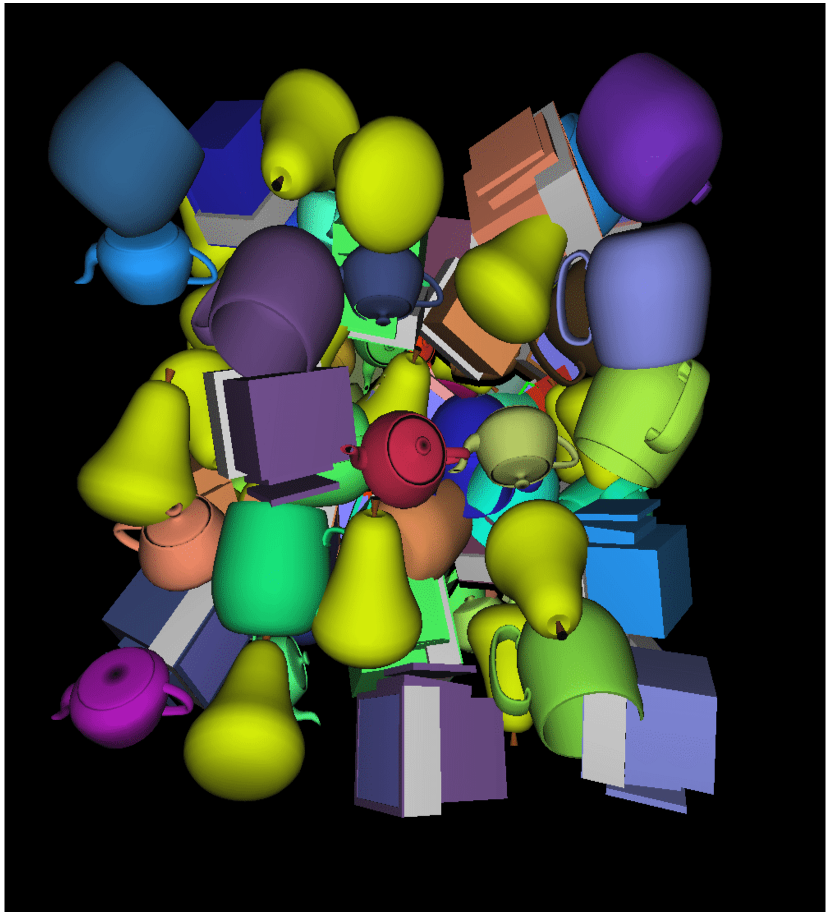
- Occlusion Culling is always interesting, if many objects are hidden by a few other objects
- Definition: **depth complexity**
  - Number of intersections of a ray through the scene
  - Number of polygons projected on a pixel
  - Number of polygons that would be visible at a pixel, if all polygons were transparent
- Comment : depth complexity depends on view point & direction





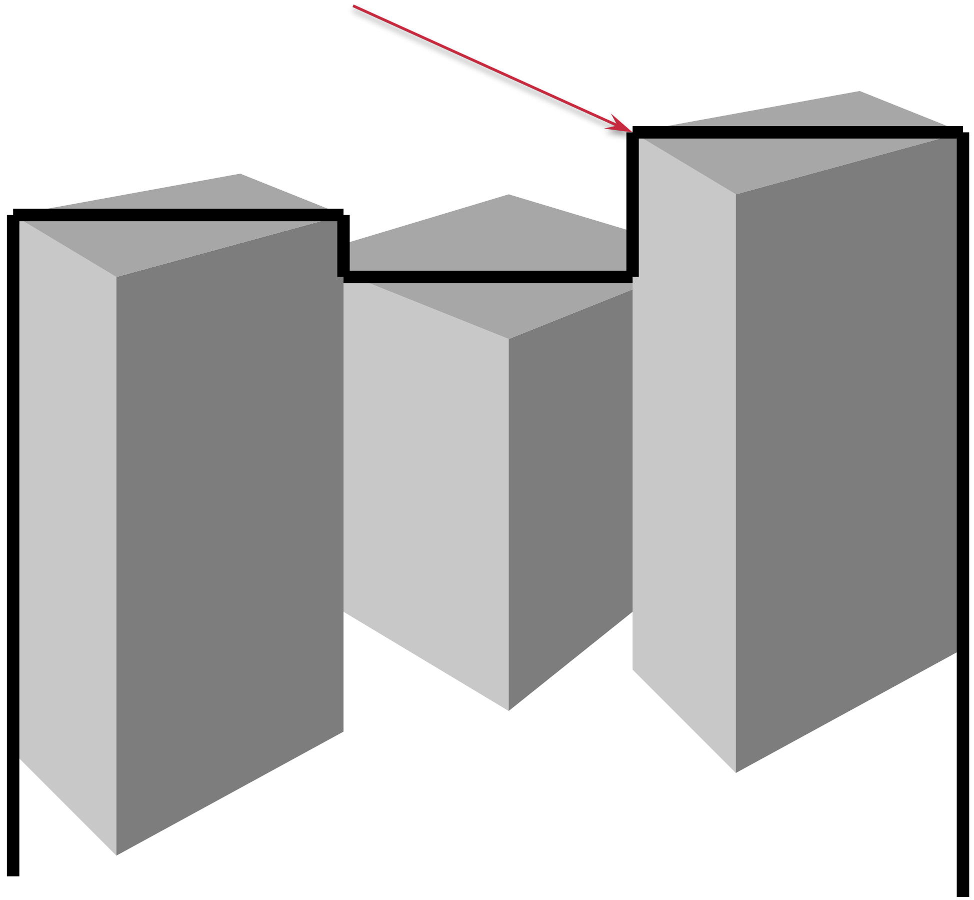
Examples of High Depth Complexity



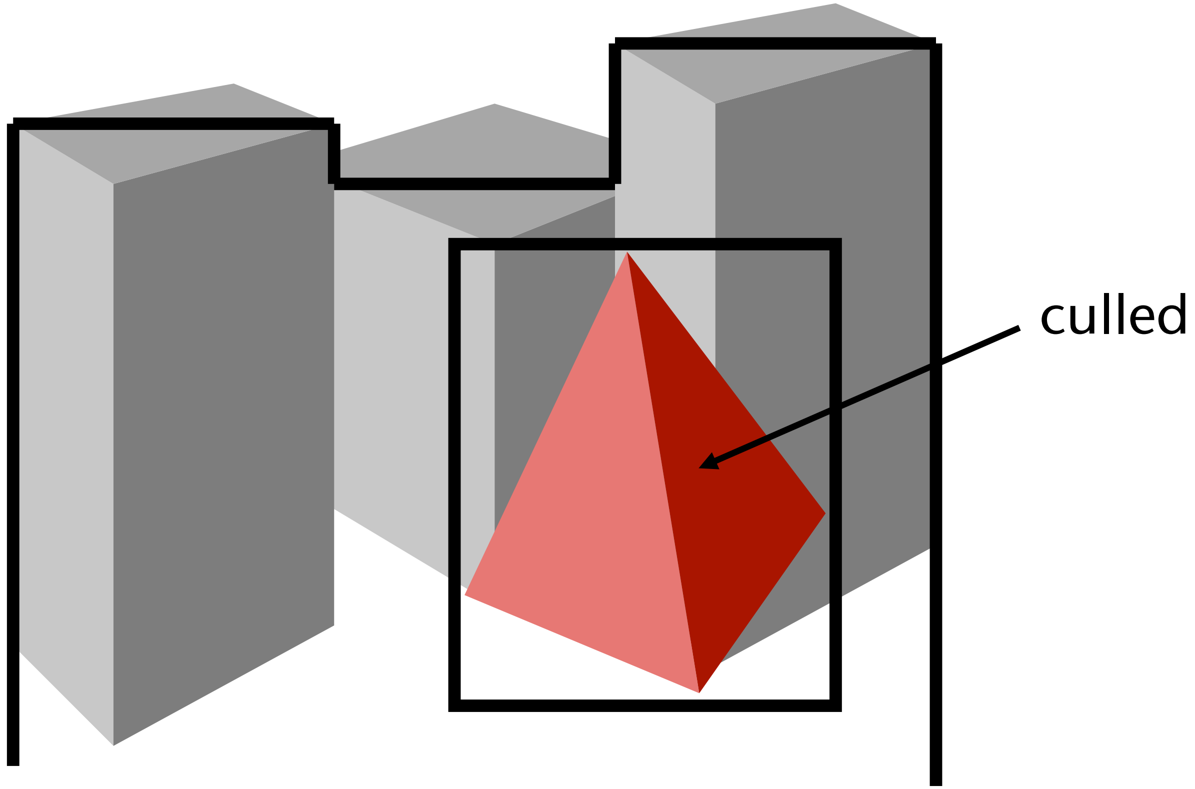


First, the Special Case of “Cities"

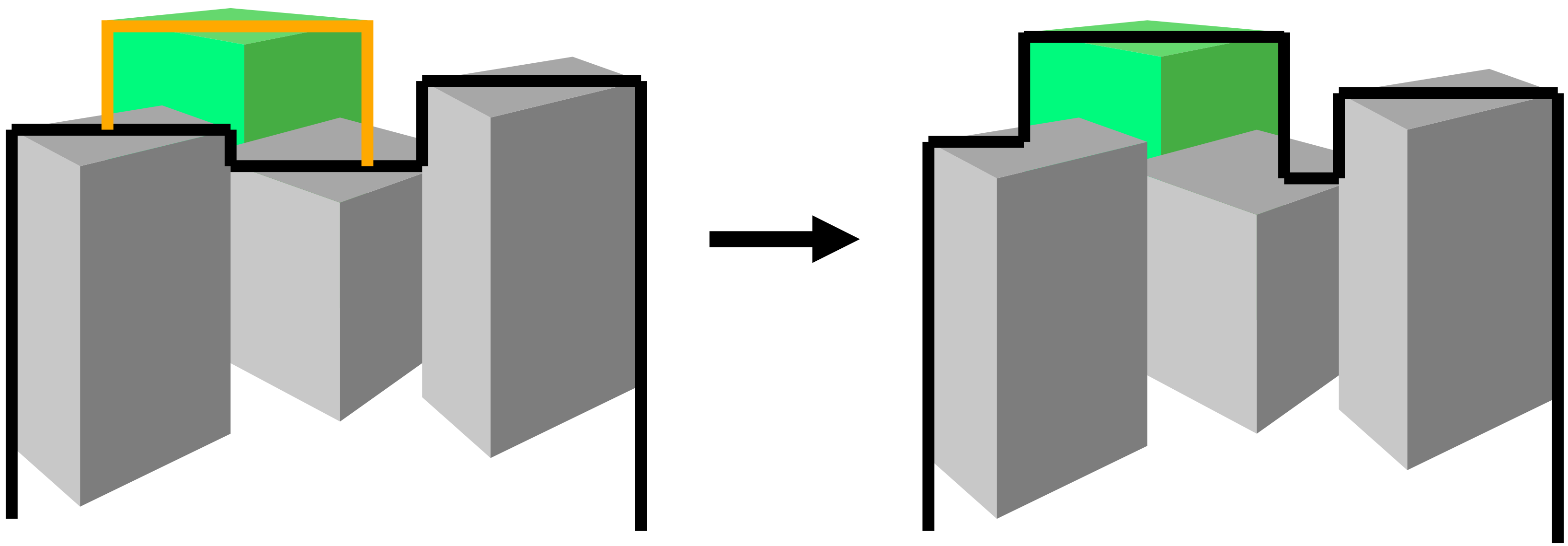
- Render the scene from front to back (reverse Painter's Algorithm)
- Generate an "occlusion horizon"



- 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

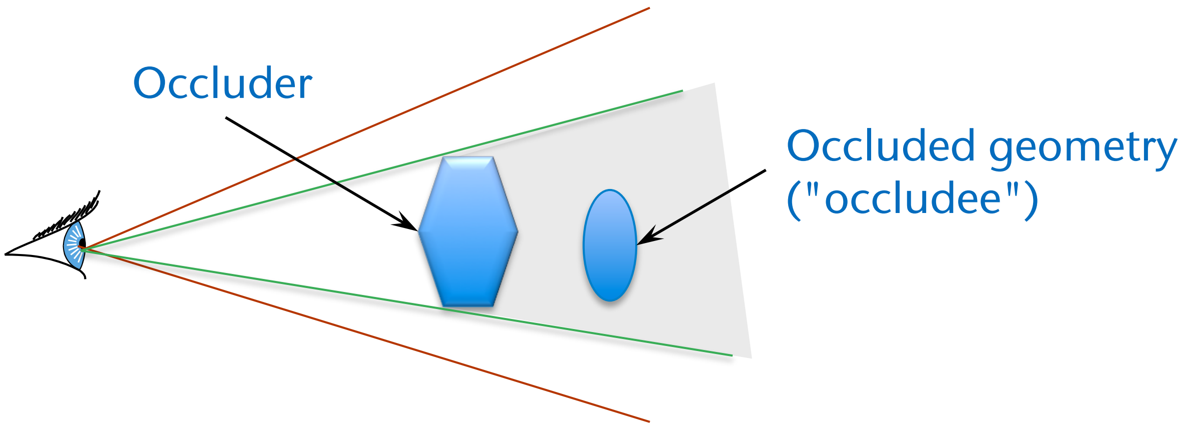


- If an object is considered as visible:
  - Add the AABB with the previous occlusion horizon

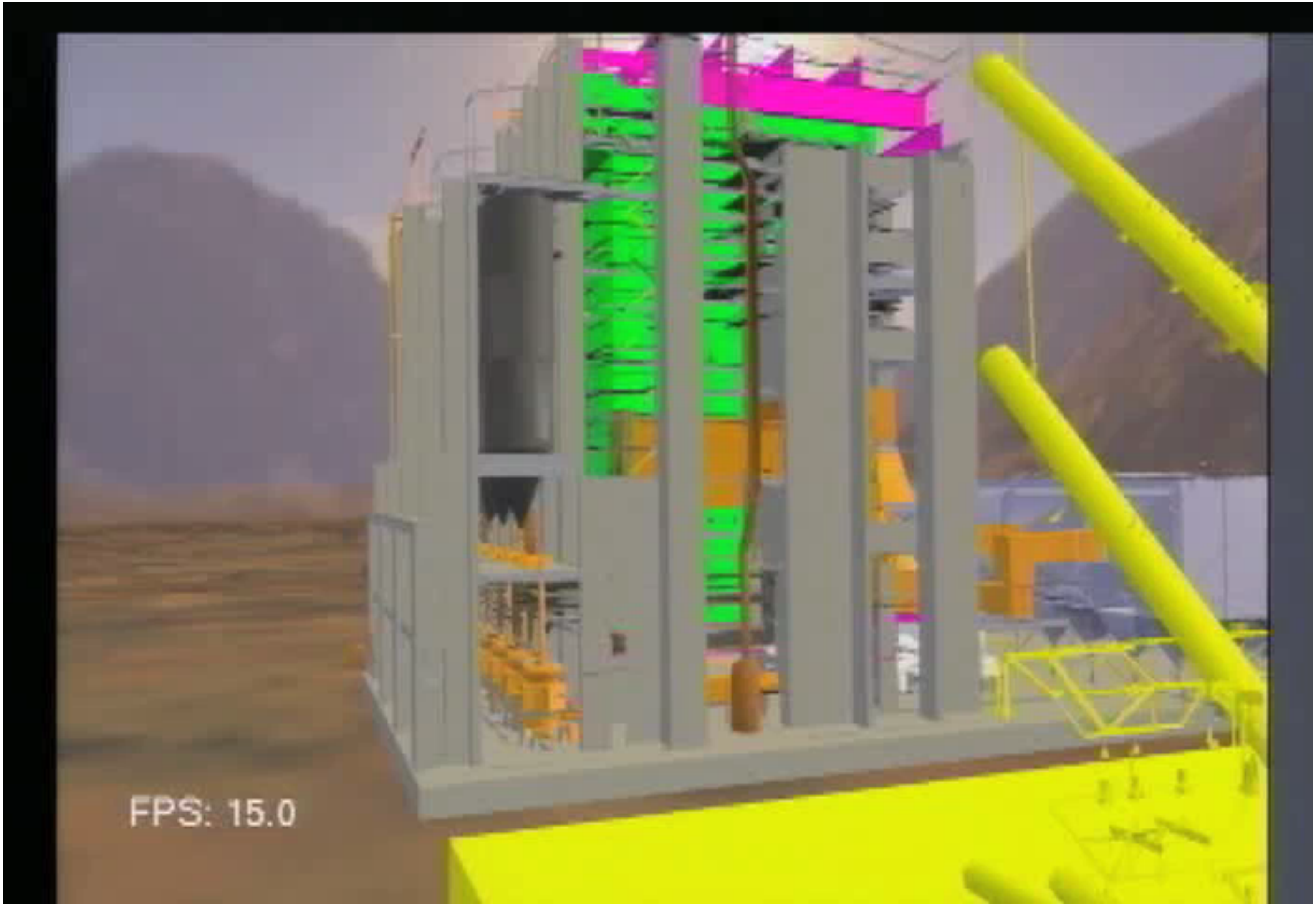


General Occlusion Culling

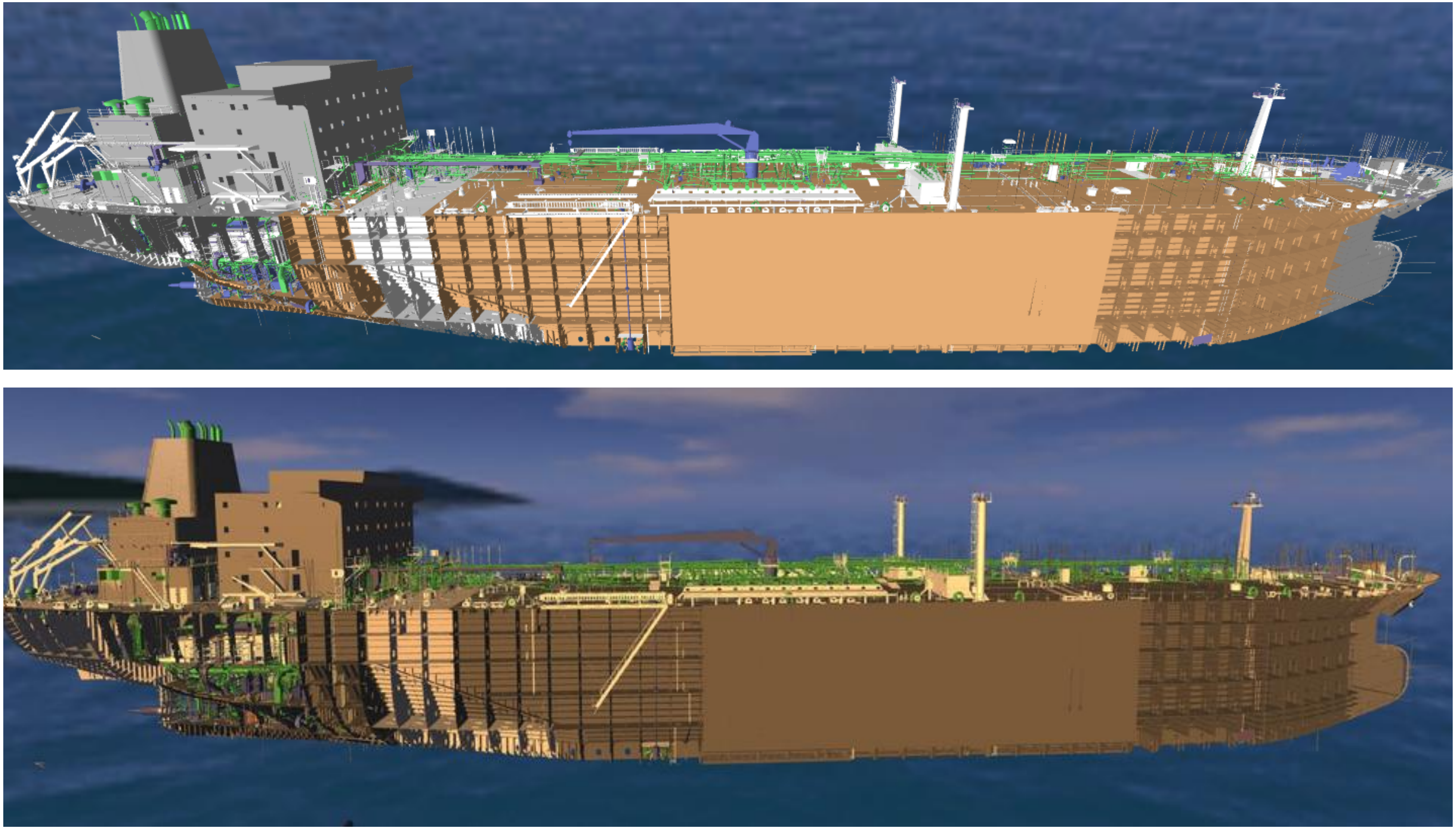
- Given:
  - A partially(!) rendered scene, and
  - A not yet rendered object
- Task:
  - Decide quickly whether the object would modify pixels in the frame buffer, if it were rendered;
  - In other words, decide quickly whether the object is completely covered by the current scene
- Terminology:



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

- Idea:
  - Draw a simple representation ("proxy") of an object, without changing the color or depth buffer
  - If no pixels would have been overwritten by the proxy (were it really drawn), then the object itself need not be drawn
- Proxy geometry: spend a bit computing power upfront, in order to hopefully save a lot of computing power later
  - Use bounding volumes as proxies (again: tightness versus effort)
  - During proxy rendering: no texturing, no shading, no light sources, no colors, texture coordinates, normals
- OpenGL: occlusion query = ask OpenGL how many pixels would be overwritten in the framebuffer by a specific OpenGL sequence
- Nowadays in OpenGL core

- First create occlusion query at initialization :

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

- Render a set of objects (try to start with those occluding a lot of the rest)
- Disable writing in Z- and color buffer (optional):

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

- Start occlusion query request for some of the later, possibly occluded, objects :

```
glBeginQuery( GL_SAMPLES_PASSED, unsigned int querynum );
// render proxy geometry, e.g. bounding volumes ...
glEndQuery( GL_SAMPLES_PASSED );
```

- Reading result of the request:

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

Demo

Auf GLFW umstellen

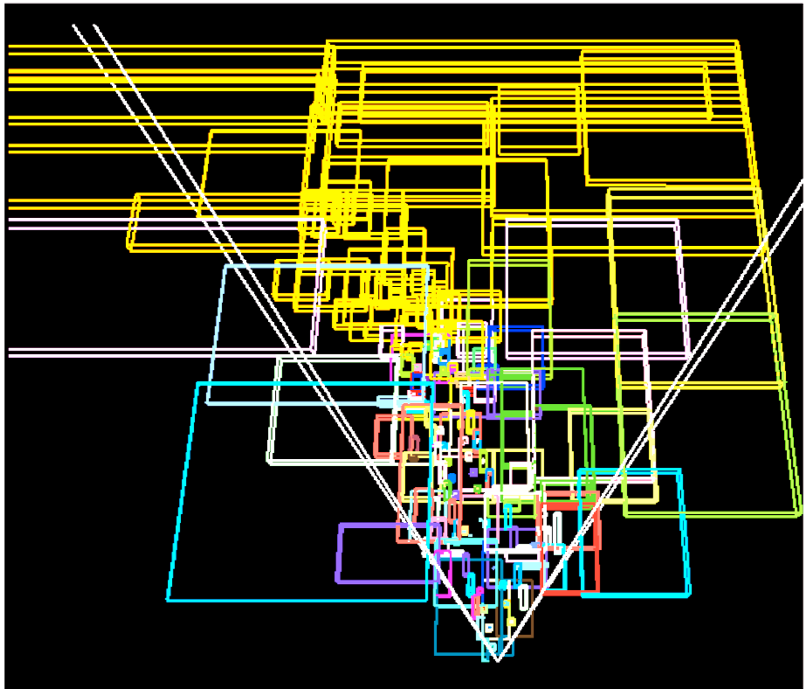
occlusion_query.cpp (~/Work/Lehre/CG1/demos/occlusion_query) - VIM

```
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, 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: an occlusion query = expensive state changes
  - Before: disable writing to color- and Z-buffer
  - After: enable all this again
  - This overhead takes more time than the actual query!
- Idea: batching
- Implement 2 additional queues
  - Both contain objects that should be tested for visibility
  - I-Queue: contains previously "invisible" objects
  - V-Queue: likewise for "visible"
  - Parameter: batch size *b* (ca. 20-80)
  - Send list of queries to OpenGL only, when batch size is reached
- "Previously visible" objects are still rendered immediately

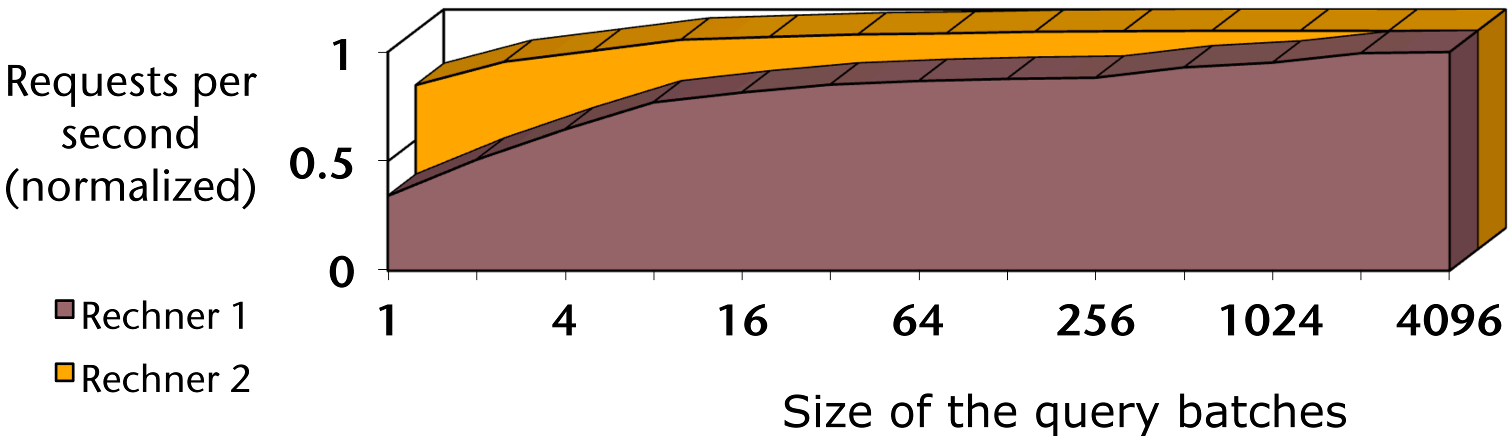
- Example: each color = one state change



Naive

CHC++

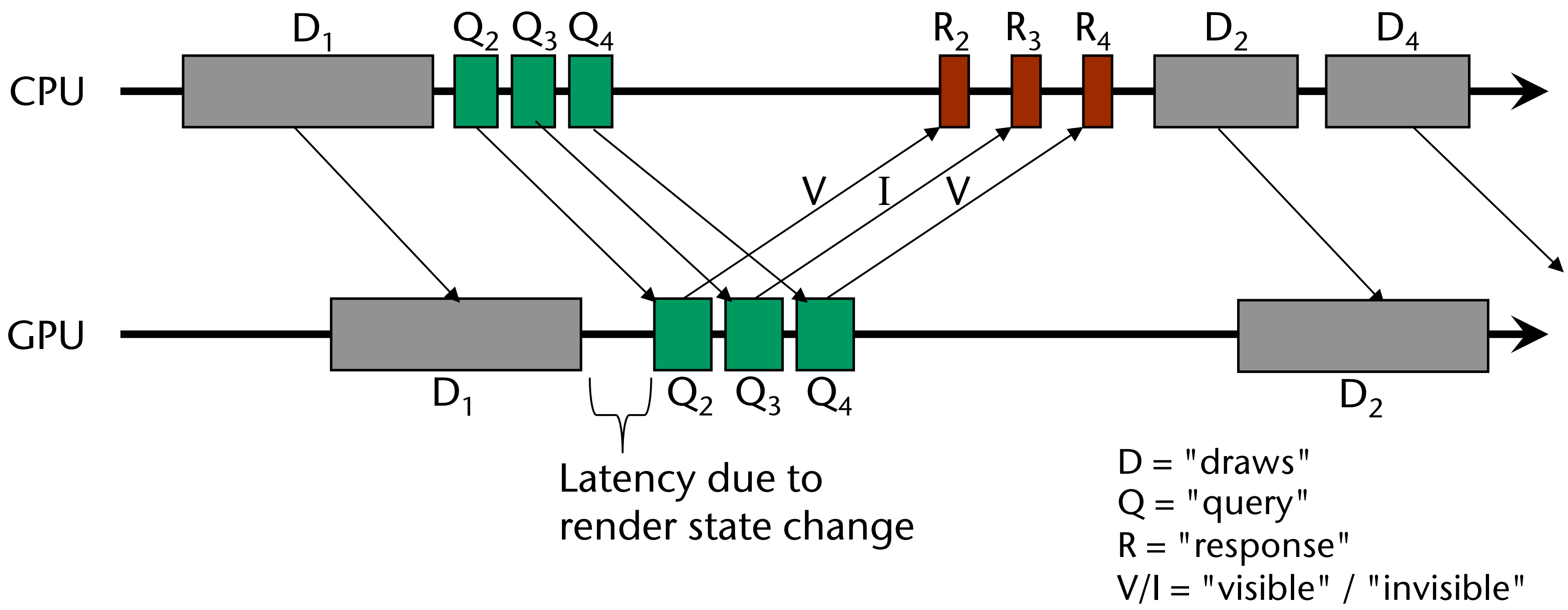
- Goal: Reduce the number of state changes, and thus the time required per occlusion query
- Therefore, send a *sequence* of requests, read the result of the sequence afterwards



The Naive "Draw-and-Wait" Approach

```
Sort items along the depth in the scene
Create query sequence
while some objects are not yet 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.



- Consequence: CPU stalls and GPU starvation