Advanced Computer Graphics
Real-Time Rendering by
Advanced Visibility Computations

G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de
Bottlenecks in the Rendering Pipeline

- Remember the graphics pipeline
- A pipeline always has the throughput of its slowest link!
- Possible bottlenecks in the graphics pipeline:
  - In rasterizer → "fill limited"
  - In geometry stage → "transform limited"
  - Bus between application and graphics hardware → "bus limited"
  - If the graphics card is faster than the application can provide geometry → "CPU limited" (recognizable by 100% CPU usage)
Classification of Visibility Problems

- Problem classes within "visibility computations":
  1. **Hidden Surface Elimination**: which pixels (parts of polygons) are covered by others?
  2. **Clipping**: which pixels (parts of polygons) are inside the viewport?
  3. **Culling**: which polygons cannot be visible? (e.g., because they are located behind the viewpoint)

- Difference: HSE & clipping are rather used to render an **accurate image**, culling is rather used to **accelerate** the rendering of large scenes

- Note: the boundary is blurred
Culling

- Let $A$ = set of all primitives in the scene; let $S$ = set of visible primitives.

- Many rendering algorithms operate on the entire set $A$, i.e., they have a minimum effort of $O(|A|)$

- No problem if $|S| \approx |A|$.

- Also no problem, if $|A|$ is small compared to the number of pixels.
  - Reminder: depth complexity

- Remark: "to cull from" = "sammeln [aus ...] / auslesen"
  "to cull flowers" = Blumen pflücken
• But for complex visual scenes, the number of visible primitives is typically much smaller than the total number of primitives! (i.e., |S| << |A|)

• Culling is an important optimization technique (as opposed to clipping)
• **Culling algorithms** attempt to determine the set of non-visible primitives \( C = A \setminus S \) (or a subset thereof), or the set of visible primitives \( S \) (or superset thereof).

• **Definition**: potentially visible set (PVS) = a superset \( S' \supseteq S \)
  
  • Goal: compute PVS \( S' \) as small as possible, with minimal effort.
Different Kinds of Culling

- View frustum culling
- Detail culling
- Backface culling
- Portal culling
- Occlusion culling
Back-Face Culling

- **Definition:** A solid = closed, opaque object = non-translucent object with non-degenerate volume
- **Observations:**
  - For solids, the back faces are never visible
  - For convex objects, there is exactly one contiguous back side
  - For non-convex solids, there may be several unconnected back sides
• **Backface Culling** = not drawing the surface parts that are on the "far" side, with respect to the viewpoint
  • Only works with **solids**!
• Predicate for backfacing:
  • Compute normal \( \mathbf{n} \) of the polygon
  • Compute **view vector** \( \mathbf{v} \) from the viewpoint to any point \( \mathbf{p} \) of the polygon
  • In perspective projection: \( \mathbf{v} = \mathbf{p} - \text{eye} \)
  • Polygon is back facing, iff angle between \( \mathbf{n} \) and \( \mathbf{v} < 90^\circ \Leftrightarrow \mathbf{n} \cdot \mathbf{v} > 0 \)
Example

\[ N_1 \cdot V = (2, 1, 2) \cdot (-1, 0, -1) \]
\[ = -4 < 0 \]
\[ \Rightarrow N_1 \text{ front facing} \]

\[ N_2 \cdot V = (-3, 1, -2) \cdot (-1, 0, -1) \]
\[ = 5 > 0 \]
\[ \Rightarrow N_2 \text{ back facing} \]
Backface Culling in OpenGL

- Just enable it:

  ```
g_slCullFace( GL_BACK );
glEnable( GL_CULL_FACE );
  ```

- Where does it happen in the GPU's pipeline:
Example
Normal Masks

- Central idea: replace the scalar product by classifying all normals
- Preprocessing: create classes over the set of all normals
  - Enclose the sphere of normals (a.k.a. Gaussian sphere) with cube (direction cube)

- Results in $6 \cdot N^2$ classes ($N =$ number of partitions along each axis)
- Classification of a normal is very easy
- With each polygon store the class of its normal
Encoding a normal (pre-processing)

- The entire direction cube $\mapsto$ bit string of length $6 \cdot N^2$
- A normal $\mapsto$ bit string with only one 1, otherwise 0
- Encode this as offset + part of the bit string that contains the 1
- E.g.: subdivide bit string in bytes, offset = 1 Byte, results in $256 \times 8 = 2048$ Bits
- Save those 2 bytes for each polygon
- E.g.: choose $N = 16$
- Results in $6 \cdot 16 \cdot 16 = 1536$ classes for the set of all normals

```
typedef struct PolygonNormalMask {
    Byte offset, bitMask;
};
```
Culling (initialization)

- Identify all those **normal classes** whose normals are **all** back-facing
- With orthographic projection:

- With perspective projection: which normals are back-facing depends on normal direction and **position** of the polygon!

- Idea: determine a "conservative" set of classes which are back-facing, regardless of the location of the polygon
• Graphical derivation how to estimate this conservative set of classes:
  
  ![Diagram showing α/2 and α/2 angles]

• In practice:
  - Test each class in all four corners of the view frustum
  - Test for a class = test of 4 normals, which are pointing to the corners of the cell (on the direction cube) that represents that class
• Represent this conservative set of classes as a bit string (e.g. 2048 Bits = 256 Bytes) in a byte array (preprocessing):

```plaintext
Byte BackMask[256];
```

• Culling (at runtime): test for each polygon

```plaintext
if ( (BackMask[polygon.byteOffset] &
     polygon.bitMask) == 0 )
    render polygon
```

• Further acceleration:
  • Divide view frustum into sectors
  • Thus, the angle $\alpha/2$ in each sector is smaller
  • For each sector, compute its own BackMask[]
Example

216 classes ("clusters")

1536 classes ("clusters")

BackMask for the current viewpoint (green = backfacing)
Speedup

Result: speedup factor ~1.5 compared to OpenGL backface culling
Clustered Backface Culling

- Reminder: some simple rules for min/max

\[
\begin{align*}
\max_i \{x_i + y_i\} &\leq \max_i \{x_i\} + \max_i \{y_i\} \\
\max_i \{x_i - y_i\} &\leq \max_i \{x_i\} - \min_i \{y_i\}
\end{align*}
\]

\[
\max_i \{k x_i\} = \begin{cases} 
  k \max_i \{x_i\}, & k \geq 0 \\
  k \min_i \{x_i\}, & k < 0 
\end{cases}
\]

- In the following, \(\mathbf{n}^i\) and \(\mathbf{p}^i\) are the normal and a vertex of a polygon in a cluster (a set) of polygons; let \(\mathbf{e}\) be the viewpoint.

- Attention: in the following, we use the "inverted" definition for backfacing!

\[\mathbf{n} \cdot (\mathbf{e} - \mathbf{p}) \leq 0\]
• Assumption: cluster (= set) of polygons is given

• All polygons in cluster are backfacing if and only if

\[ \forall i : \mathbf{n}^i \cdot (\mathbf{e} - \mathbf{p}^i) \leq 0 \quad \iff \quad \max \{ \mathbf{n}^i \cdot (\mathbf{e} - \mathbf{p}^i) \} \leq 0 \tag{1} \]

• Upper bound for (1) is

\[ \max \{ \mathbf{n}^i \cdot (\mathbf{e} - \mathbf{p}^i) \} \leq \max \{ \mathbf{e} \mathbf{n}^i \} - \min \{ \mathbf{n}^i \mathbf{p}^i \} \tag{2} \]

• Set \( d := \min\{\mathbf{n}^i \mathbf{p}^i\} \) (pre-computation)

• Write (2) as

\[ \max \{ \mathbf{n}^i \cdot (\mathbf{e} - \mathbf{p}^i) \} \leq \max \{ e_x n_x^i + e_y n_y^i + e_z n_z^i \} - d \]

\[ \leq \max \{ e_x n_x^i \} + \max \{ e_y n_y^i \} + \max \{ e_z n_z^i \} - d \tag{3} \]
• Assumption: $\mathbf{e}$ is located in the positive octant, i.e., $e_x, e_y, e_z \geq 0$; then we can rewrite (3) as:

$$\max \left\{ \mathbf{n}^i (\mathbf{e} - \mathbf{p}^i) \right\}$$

$$\leq e_x \cdot \max \{ n_x^i \} + e_y \cdot \max \{ n_y^i \} + e_z \cdot \max \{ n_z^i \} - d$$

$$\leq \mathbf{m} \cdot \mathbf{e} - d \ , \ \text{mit} \ \mathbf{m} = \begin{pmatrix} \max \{ n_x^i \} \\ \max \{ n_y^i \} \\ \max \{ n_z^i \} \end{pmatrix}$$

• Analogously for $e_x, e_y, e_z \leq 0$:

$$\max \left\{ \mathbf{n}^i (\mathbf{e} - \mathbf{p}^i) \right\} \leq \mathbf{\bar{m}} \cdot \mathbf{e} - d , \ \text{with} \ \mathbf{\bar{m}} = \begin{pmatrix} \min \{ n_x^i \} \\ \min \{ n_y^i \} \\ \min \{ n_z^i \} \end{pmatrix}$$
• For all other octants, combine min and max appropriately
  • Construct vector $\mathbf{w}_e$, combined from $\mathbf{m}$ and $\bar{\mathbf{m}}$ like this:

  $$\mathbf{w}_e = (w_x, w_y, w_z) \quad \text{with} \quad w_x = \begin{cases} m_x, & e_x \geq 0 \\ \bar{m}_x, & e_x < 0 \end{cases}$$

  • This allows us to write the (conservative) test as:

  $$\mathbf{w}_e \cdot \mathbf{e} - d \leq 0 \quad \Rightarrow \quad \text{cluster is backfacing} \quad (4)$$

• Pre-computation: for each cluster determine $\mathbf{m}$, $\bar{\mathbf{m}}$, and $d$
• Memory requirements per cluster: 28 bytes (2 vectors + 1 scalar)
Geometric Interpretation

- Inequality (4) defines 8 planes (one per octant)
- The 4 planes of adjacent octants intersect at one point, which lies on the coordinate axis "between" the 4 octants
  - Example: consider the 4 planes in the 4 octants with $e_x \geq 0$
  - All 4 planes have normals of the form $n = (m_x, \cdot, \cdot)$
  - So, they all intersect the x-axis at the point $(\frac{d}{m_x}, 0, 0)$
- Those 8 planes form a closed volume, the so-called culling volume
- If the viewpoint is anywhere inside the culling volume, then the cluster is completely backfacing
Further Optimization: Change to Local Coordinates

• Problem: if the polygons are far away from the origin, and the origin is located on the positive side of the normal, then $d$ is very much negative $\rightarrow$ the test is never positive

• Solution: run the test in a *local coordinate system* by translating all polygons in the cluster to a local origin $c$ such that

$$d = \min \{ \mathbf{n}^i \cdot (\mathbf{p}^i - c) \}$$

is as large (and positive) as possible

• Wanted is the optimal $c$
  • In practice: try the center and corners of the BBox of the cluster as $c$
  • Save $c$ with the cluster, then test $\mathbf{w}_{(e-c)} \cdot (e - c) - d \leq 0$

• Question: Will rotation improve this further?
Hierarchical Clustered Backface Culling

• Two clusters can be combined to form a joint cluster:

\[
\mathbf{m} = \begin{pmatrix}
\max(m_x^1, m_x^2) \\
\max(m_y^1, m_y^2) \\
\max(m_z^1, m_z^2)
\end{pmatrix}
\quad \mathbf{\hat{m}} = \begin{pmatrix}
\min(\tilde{m}_x^1, \tilde{m}_x^2) \\
\min(\tilde{m}_y^1, \tilde{m}_y^2) \\
\min(\tilde{m}_z^1, \tilde{m}_z^2)
\end{pmatrix}
\quad \hat{d} = \min(d_1, d_2)
\]

• These two vectors and \( \hat{d} \) provide a conservative estimate

• I.e.: if the joint cluster is back-facing, then the two original clusters are guaranteed to be back-facing, too → cluster hierarchy

• For creating a hierarchy of clusters, define a front-facing test, analogously to the back-facing test

  • Stop testing, if a complete joint cluster is front- or back-facing
  • Otherwise: test the children for being completely front- or back-facing
Generating the Clusters

- For the evaluation of cluster candidates in an algorithm, we need a measure of the "performance" of a cluster.
- Here: probability $P$ that the cluster $C$ will be culled.
- Use a heuristic to calculate $P$:

$$P(C) = \frac{\text{Vol}(\text{culling volume})}{\text{Vol}(\text{all possible viewpoint position})} = \frac{\text{Vol}(C)}{\text{Vol}(U)}$$

- $\text{Vol}(C)$ can be computed exactly.
- For $U$ choose the BBox of the entire scene.
- If local culling coordinates are used: choose $U = c \cdot \text{Bbox}($cluster$)$ ("near-culling probability")
• Question: given two clusters $A$, $B$; 
Is it faster to test and to render (and test) $A$ and $B$ directly, 
or is it faster to test the joint cluster $C = A \cup B$ first? 
(on average!)

• Let $T(A)$ be the expected(!) time to test cluster $A$ and render it in case of 
(possible) visibility. Then 
\[ T(A) = t + (1 - P(A)) R(A) \]

where $P(A) = \text{probability, that cluster } A \text{ gets culled},$
$R(A) = \text{time to render } A \text{ (without further tests), and}$
$t = \text{time for the back-face test of a cluster}$
• So, combining clusters A and B is worthwhile, if and only if

\[ T(C) < T(A) + T(B) \]  \iff \\
\[ t + (1 - P(C)) R(C) < 2t + (1 - P(A)) R(A) + (1 - P(B)) R(B) \]  \iff \\
\[ P(C) > \frac{-t + P(A)R(A) + P(B)R(B)}{R(A) + R(B)} \]  \iff \\
\[ P(C) > \frac{P(A)n_A + P(B)n_B - \frac{t}{r}}{n_A + n_B} \]  \iff Assumption: \( R(A) = n_A r \), \( r = \) constant effort for one polygon

• Ratio \( t/r \) depends on the machine; but can easily be determined experimentally and automatically in advance (depends on graphics card, number of light sources, textures, ...)
View-Frustum Culling

- In many real scenes, a substantial percentage of the scene is outside the view frustum
- Goal: try to calculate a Potentially Visible Set as "tight" and as quickly as possible
Bounding Volumes (BVs)

- Test per polygon is too expensive, overall rendering time would be slower than without any view-frustum culling at all.
- Therefore, test complete objects (= set of polygons) whether they are outside the view frustum.
- Do conservative, but fast tests with simple bounding volumes (BVs).
- The process is efficient only if
  \[ \text{Cost( BV test )} \ll \text{Cost( rendering the polygon set )} \]
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. Get parameters camera matrix, near, far
  2. Calculate vertices of the frustum
  3. Calculate the frustum planes
• Determine vertices (in world coordinates):
  \[ F = C + f \cdot d \]
  \[ P = F + \frac{1}{2} H v - \frac{1}{2} W u \]
  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 \( E_i : \mathbf{x} \cdot \mathbf{n}_i - d_i = 0 \)
  and a sphere \((\mathbf{x} - \mathbf{c})^2 - r^2 = 0\)

- Question: Is the sphere completely outside the frustum?
  - Yes \(\leftrightarrow\) \(\exists i : \mathbf{c} \cdot \mathbf{n}_i - d_i > r\)
  - If \(\exists i : |\mathbf{c} \cdot \mathbf{n}_i - d_i| \leq r\)
    then one of the planes intersects the sphere (but not necessarily the frustum)
  - If \(\forall i : \mathbf{c} \cdot \mathbf{n}_i - d_i < -r\)
    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 only two corners against each plane:
  • We denote by "N vertex" that vertex of all vertices where
    \[ f(x) = x \cdot n - d \]
    assumes the minimum
  • Analogously define "P vertex" (f assumes max)
    • Note: these are (almost always) unique because f is monotone, and a box is convex

• Algorithm for test box v. frustum:

```python
loop over all planes i:
calculate \( f_i(N \text{ vertex}) \)
if N vertex is on positive side:
  → complete box is on the positive side
  → complete box is outside the frustum
calculate \( f_i(P \text{ vertex}) \)
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 = \( (x_{\text{min}}, y_{\text{min}}, z_{\text{min}}, x_{\text{max}}, y_{\text{max}}, z_{\text{max}}) \)

\[
\begin{align*}
P_x &= \begin{cases} 
  x_{\text{max}}, & n_x \geq 0 \\
  x_{\text{min}}, & n_x < 0 
\end{cases} \\
N_x &= \begin{cases} 
  x_{\text{min}}, & n_x \geq 0 \\
  x_{\text{max}}, & n_x < 0 
\end{cases} \\
\end{align*}
\]

\[
\begin{align*}
P_y &= \begin{cases} 
  y_{\text{max}}, & n_y \geq 0 \\
  y_{\text{min}}, & n_y < 0 
\end{cases} \\
N_y &= \begin{cases} 
  y_{\text{min}}, & n_y \geq 0 \\
  y_{\text{max}}, & n_y < 0 
\end{cases} \\
\end{align*}
\]

\[
P_z = \ldots
\]
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.
Hierarchical View Frustum Culling

- At each node of the scene graph, store a bounding volume enclosing the complete subtree → 

  **Bounding-Volume-Hierarchy (BVH)**

- Traverse this BVH und test all nodes against the view frustum
Further Optimization

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

- Occlusion Culling should be considered, if many objects are hidden by a few other objects
- Definition: depth complexity of a scene
  - Number of intersections of a ray with the scene, averaged over "all possible rays"
  - Number of polygons projected onto the same pixel, on average
  - Number of polygons that would be visible at a pixel, if all polygons were transparent
- Comment: depth complexity depends, more or less, 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 (red object is behind the gray objects):
  • Determine axis-aligned bounding box (AABB) of the projection of the object
  • Compare with the occlusion horizon
• If an object is determined as visible:
  • Merge its 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 (or any other API)

- Idea:
  - First, draw a simple representation ("proxy") of an object, without changing the color or depth buffer
  - Then, check if no pixels would have been overwritten by the proxy (were it really drawn), then the object itself need not be drawn
  - Rationale: spend a bit computing power upfront, in order to (hopefully) save a lot of computing power later

- Proxy geometry:
  - Use bounding volumes as proxies (again: tightness versus effort)
  - During proxy rendering: no texturing, no shading, no light sources, no colors, texture coordinates, normals

- Occlusion query = ask OpenGL how many pixels would be overwritten in the framebuffer by a specific OpenGL sequence
The OpenGL API

1. First create occlusion query at initialization:
   
   `glGenQueries( int count, unsigned int queryIDs[] );`

2. Render a set of objects (try to start with those occluding a lot of the rest)

3. Disable writing in Z- and color buffer:
   
   `glDepthMask( GL_FALSE );`
   `glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );`

6. Start occlusion query request for some of the later, possibly occluded, objects:
   
   `glBeginQuery( GL_SAMPLES_PASSED, unsigned int query_id );`
   `// render proxy geometry, e.g. bounding volumes ...`
   `glEndQuery( GL_SAMPLES_PASSED );`

9. Read result of the request:
   
   `glGetQueryObjectiv( int query_id, GL_QUERY_RESULT, int *samplesCounted );`
Batching Queries

• Problem: occlusion query → expensive state changes
  • Before query: wait for pipeline to become empty; disable writing to color- and Z-buffer
  • After query: wait for empty pipeline, retrieve occlusion results, then enable all previous states again
  • This overhead takes more time than the actual query!

Naive approach. Colors represent query ID's
• Goal: Reduce the number of state changes, and thus the time required per occlusion query

• Idea: batching: implement 2 additional queues
  • Both contain objects that should be tested for visibility
  • I-Queue: contains previously invisible objects
  • V-Queue: contains previously visible objects
  • Parameter: batch size $b$ (ca. 20-80)

• During rendering, send a sequence of requests (when batch size reached), read the result of the sequence afterwards

• "Previously visible" objects are still rendered immediately
The Naive "Draw-and-Wait" Approach

Sort items along their approx. 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  // wait here until result available
        if # pixels drawn > 0:
            Render object
Sort the Object List

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

   worst case:
   
   4  3  2  1

   1  2  3  4

   best case:

• Solution: sort the list by distance to the object Viewpoint
Problem of the naive approach

- Very high response time (latency) for a query, because of ...
  - long graphics pipeline,
  - time for the execution of the queries itself (rasterization), and
  - transfer the result back to the host.

- Consequence: **CPU stalls** and **GPU starvation**

![Diagram showing the flow of data between CPU and GPU, with labels D, Q, R, V/I indicating draws, queries, responses, visible/invisible respectively.](image)

Latency due to render state change

D = "draws"
Q = "query"
R = "response"
V/I = "visible" / "invisible"
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 $\rightarrow$ draw object
  - Disadvantage: Often outer parts of the BVs are visible where no object pixels 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 $\geq$ threshold
  - Potentially "small" holes in or between objects
Coherent Hierarchical Culling (CHC & CHC++)

FYI - Not Relevant for Exam

- Here in a simplified representation (a.o. without hierarchy)
- Given: set of objects
  - Here: object = a set of (usually) contiguous polygons
- Ideas:
  - Maintain a queue with occlusion queries that were already issued
  - Assumption (for the moment): if an object was visible in the last frame, it is also visible in the current frame
  - If an object was invisible in the last frame, then check its visibility in this frame before rendering it (occlusion query)
  - Do not wait for the result; instead continue going through the list
  - Act upon query results as soon as they become available
The Algorithm

$L = \text{list of all objects (incl. BVs)}$
$Q = \text{queue for occlusion queries (initially empty)}$

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 $\text{BV}(O)$
    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 for Q.front is available
    O = Q.pop
    if #visible pixels of query for O > threshold:
        mark O as "visible"
        if O is not marked "previously visible":
            render O
    else:
        mark O as "invisible"
end while

until Q empty and L empty

In the following: gradual improvement of this algorithm
Fusion of (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:

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

This will be called a multiquery.
• Definition: visibility persistence of a set of objects

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

where \( I(t) \) = number of objects (out of the set of objs), which were constantly occluded in the previous \( t \) frames

• Interpretation: \( p(t) = \) "probability ", that one object, which was occluded \( t \) frames, will also be occluded in the following frames

• Observation: \( p(t) \) is amazingly independent from object and scene

• Consequence: can be approximated well by an analytic function, e.g.

\[ p(t) \approx 0.99 - 0.7e^{-t} \]
- Let $t_O$ = number of previous frames which the object $O$ was occluded
- Define an "oracle" for a set $M$ of objects
  $i(M) :=$ the "probability" that all objects in $M$ will be occluded in the current frame (this is just a heuristic!):

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

- Based on that, define:
  - Costs of an occlusion multiquery (issued as a batch):

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

    $$B(M) = c_2 i(M) \sum_{O \in M} \text{num polygons of } O$$
• **Expected value** of a multiquery:

\[ V(M) = \frac{B(M)}{C(M)} \]

• If the I-queue is full at some point:
  
  • Sort the objects \( O_i \) in the I-queue according to \( t_O \rightarrow \{O_1, \ldots, O_n\} \)
  
  • Greedily find \( \max_{b=1\ldots n}\{ V(\{O_1, \ldots, O_n\}) \} \)
  
  • Issue an occlusion multiquery for these first \( b \) objects in the I-queue
  
  • Repeat until the I-queue is empty
Tighter Bounding Volumes

• Observation: the larger the BV relative to the object, the more likely it is that the occlusion query returns a "false positive" (claims "visible", but "invisible" in reality)
• Objective: BVs as tight as possible
• Conditions:
  • BVs must be very fast to render
  • BVs must not cost a lot of memory
• Idea:
  • Decompose object into clusters (of polygons)
  • Compute a BBox around each cluster (e.g., AABB)
  • Use as BV for an object the union of the small BBoxes

FYI - Not Relevant for Exam
• Question: how small should the "small" AABBs (or clusters) be?

• Observation: the larger the number of small AABBs, ...
  • ... the larger the probability that "invisible" is correctly recognized, but
  • ... the larger the surface → longer rendering time of the resulting occlusion queries

• Strategy for the construction of tight AABBs:
  • Divide clusters recursively
  • Termination criterion: if
    \[ \sum \text{surface area of small AABBs} > \sigma \cdot \text{surface area of the big AABB} \]
    • Parameter \( \sigma \) depends on the graphics card (\( \sigma \approx 1.4 \) seems OK)
Altogether

- The queues in CHC++:

  ![Diagram of queues](image)

  - Rendering queue
  - V-queue (visible nodes)
  - I-queue (invisible nodes)
  - Query queue
  - Multiquery
  - OpenGL
Results

- Walk-through through the power plant model:

![Time (ms) vs Frame graph](image-url)
Comparison of the Number of State Changes

FYI - Not Relevant for Exam
Visualization of Multi-Queries

FYI - Not Relevant for Exam
Another Special Case: Architectural Models
Cells and Portals (*Portal Culling*)

- **Scenario:** walkthrough of buildings and cities
- **Partition space (and geometry) into** cells; cells are connected by portals (doors, windows, ...) → neighborhood graph
- **Question:** exactly what is a cell?
- **Observation:** viewpoint can see other cells only through the portals
- **Which cell is included in the PVS?**
  - The cell which contains the viewpoint
  - Cells that have a portal to the initial cell
  - Others?
Approach

• Compute neighborhood graph between cells
• Start with cell A (containing the viewpoint), traverse the graph using BFS
• For each reached cell X, check whether portal "entering" X and portal "leaving" A are visible to each other
• Do this for each cell → visibility table for each cell (preprocessing)
How to Check Portal-Portal Visibility

• Exact solution involves beam tracing
  • Beam = generalized cone
  • Portals "cut out" a piece from the silhouette of a beam

• Heuristic solution:
  • Sample border of leaving and entering portals
  • Draw line between all pairs of sample points
  • Check whether line is inside all other portal polygons ("stabs" them)
Example Scene
Detail Culling

- Idea: objects that occupy less than $N$ pixels on the screen are not shown
- Rationale: users might not notice, or don't care (especially, when viewpoint is moving fast, e.g., in games)
- Advantage: trade-off quality vs. speed
Estimating the size of an object in screen space

\[ d = \mathbf{v} \cdot (\mathbf{c} - \mathbf{e}) \]  
Distance along \( \mathbf{v} \)

\[ \hat{r} = r \cdot \frac{n}{d} \]  
Estimate of the projected radius

\[ \pi \hat{r}^2 = \]  
Estimated area of the projected sphere