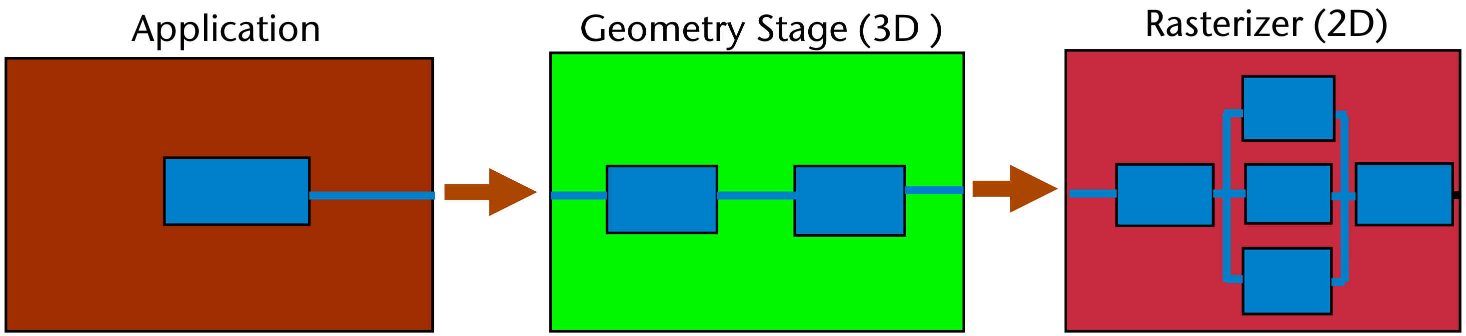
Advanced Computer Graphics
Real-Time Rendering by Advanced Visibility Computations

G. Zachmann
University of Bremen, Germany
cgvr.informatik.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 app. 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

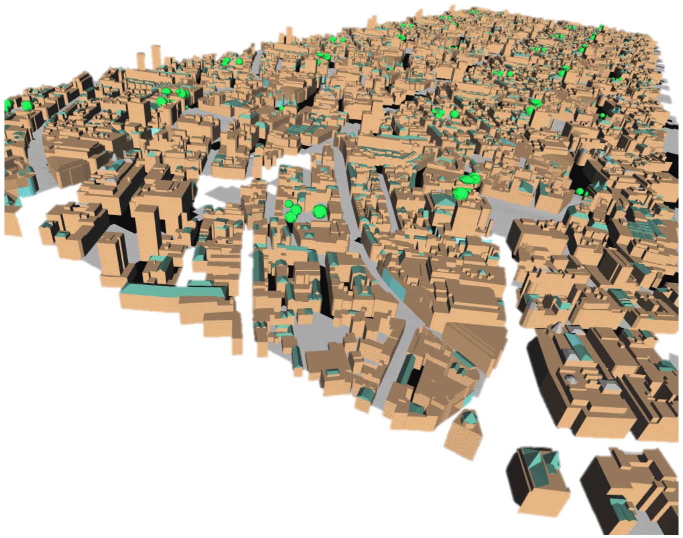
- Problem classes within "visibility computations":
  1. Hidden Surface Elimination: which pixels (parts of polygons) are covered by others?
  2. Clipping: which pixels (parts of polygons) are inside the viewport?
  3. Culling: which polygons cannot be visible? (e.g., because they are located behind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

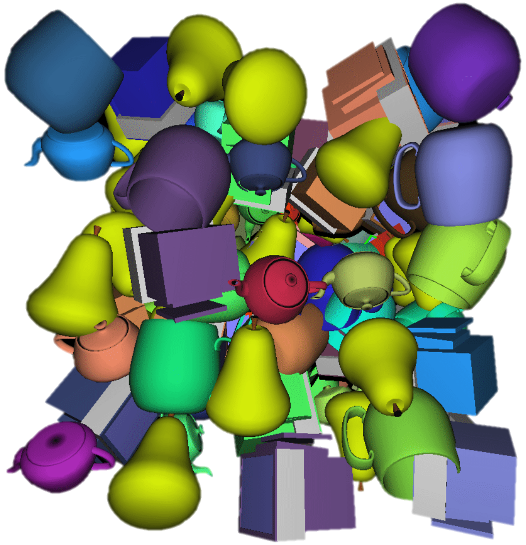
Culling

- Let A = set of all primitives;
  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 the number of primitives is small compared to the number of pixels
  - Reminder: depth complexity

- *"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., $|S| << |A|$)

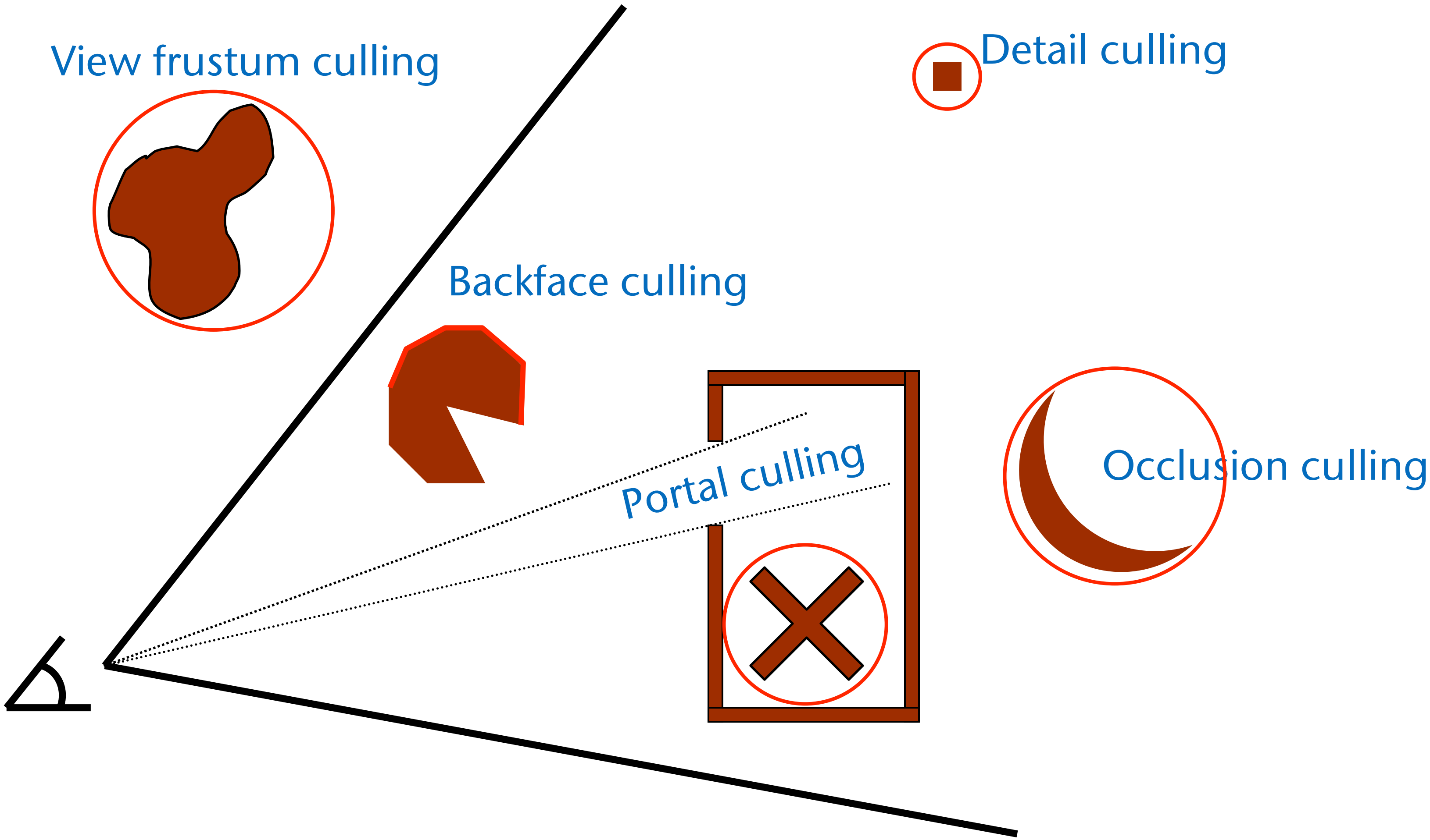




- Culling is an important optimization technique (as opposed to clipping)

- For $|S| << |A|$, existing rendering algorithms are not efficient
- 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
  - Trivial PVS (with trivial effort) is, of course, A

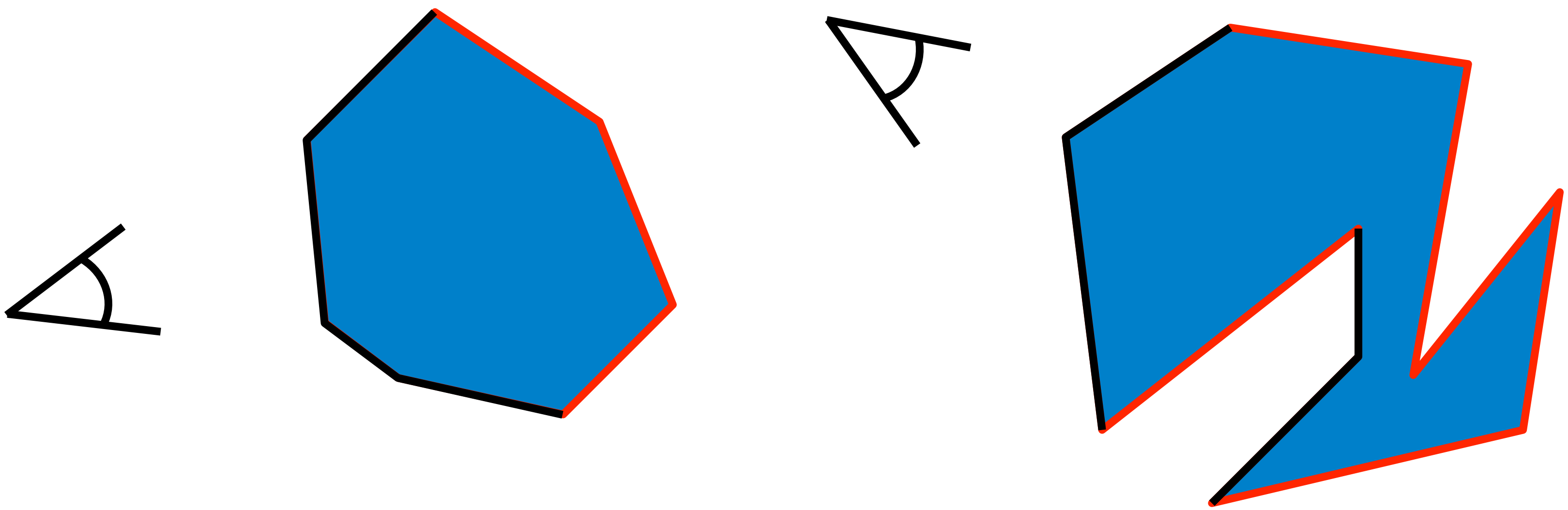
Kinds of Culling



Bremen

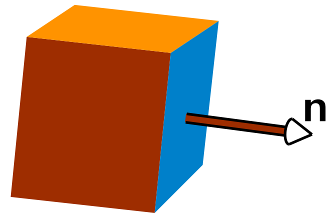
Back-Face Culling

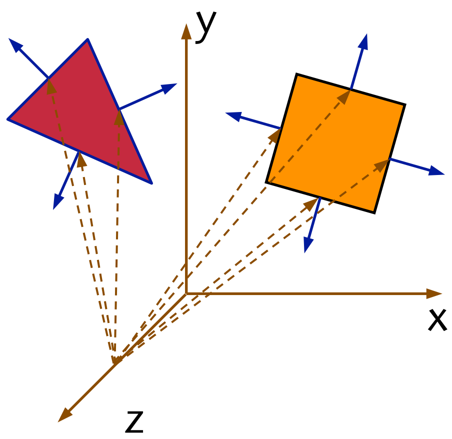
- Definition: a solid = closed, opaque object = non-translucent object with non-degenerate volume
- Observations:
  - With solids, the back faces are never visible
  - For convex objects, there is exactly one *contiguous* back side
  - For non-convex solids, there may be several unconnected back sides



- Backface Culling = not drawing the surface parts that are on the far side, with respect to the viewpoint
  - Only works with solids!
- Compute normal **n** of the polygon
- Compute view vectors **v** from the viewpoint to all points **p** of the polygon
  - Perspevtive projection: **v** = **p** – **eye**
  - Orthogonal projection: **v** = $[0\ 0\ 1]^T$
- Polygon is back facing (don't draw), iff angle between **n** and **v** < 90°

$$\Leftrightarrow \quad \mathbf{n} \cdot \mathbf{v} > 0$$





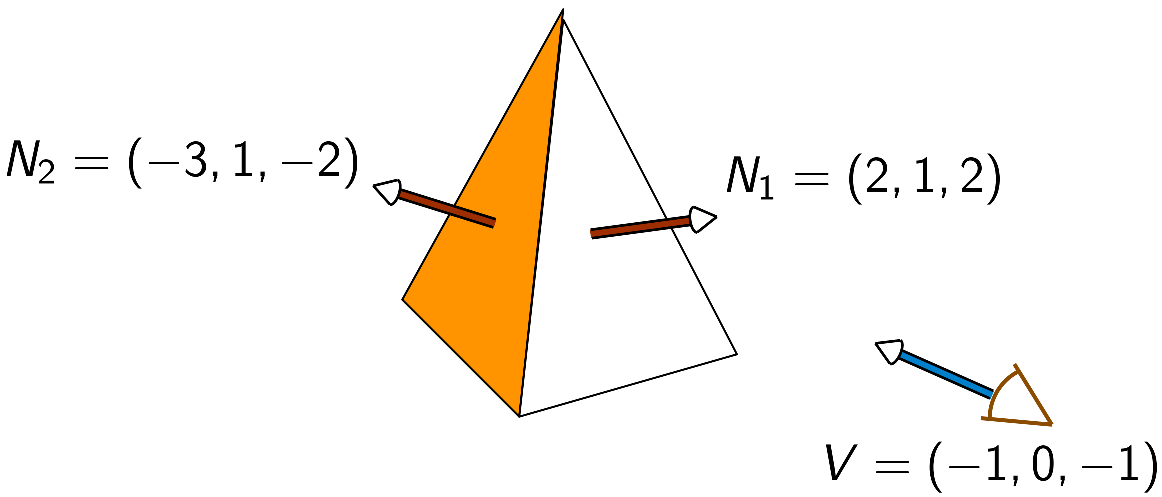
Example

$$N_1 \cdot V = (2,1,2)\cdot(-1,0,-1) = -4 < 0$$

$\Rightarrow N_1$ front facing

$$N_2 \cdot V = (-3,1,-2)\cdot(-1,0,-1) = 5 > 0$$

$\Rightarrow N_2$ back facing

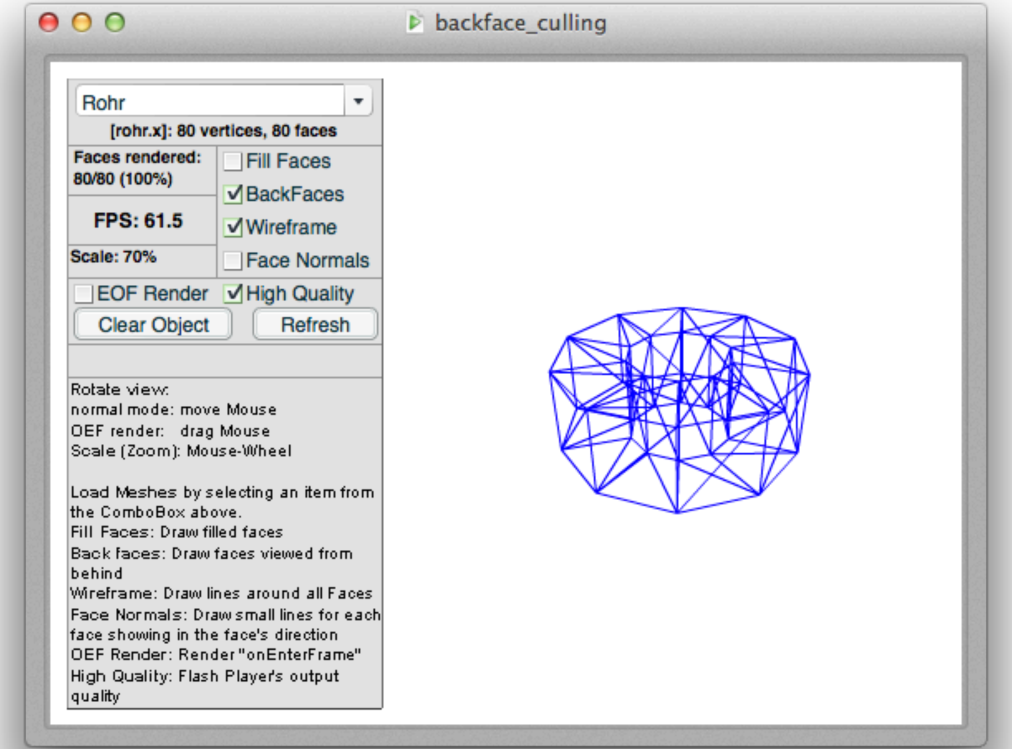


Backface Culling in OpenGL

- Just enable it:

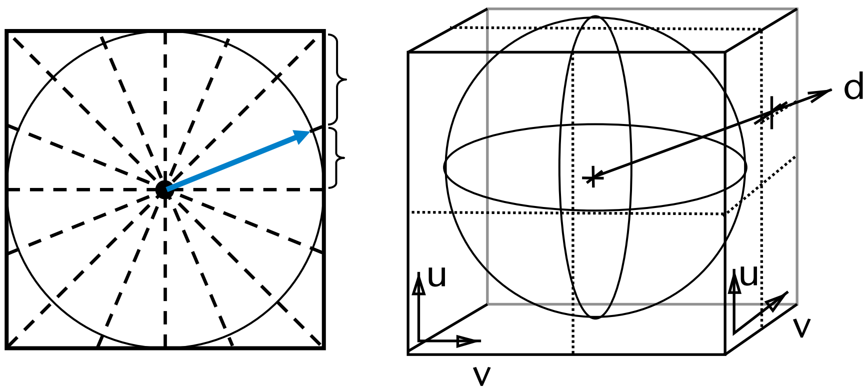
```
glCullFace( GL_BACK );
glEnable( GL_CULL_FACE );
```

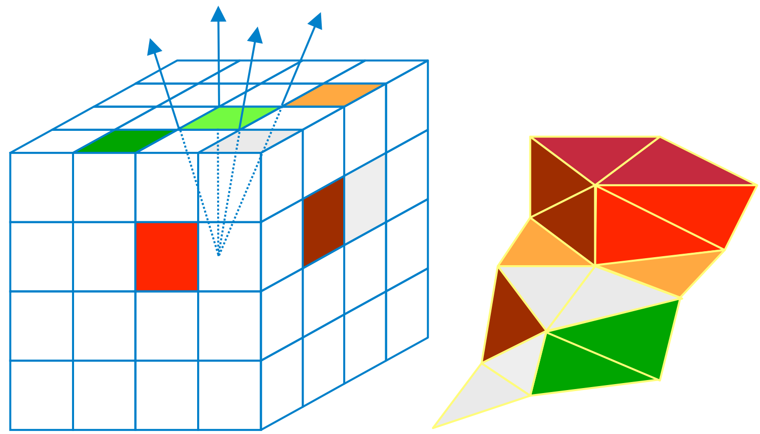
Demo


backface_culling
Rohr
[rohr.x]: 80 vertices, 80 faces
Faces rendered: 80/80 (100%)
FPS: 61.5
Scale: 70%
Fill Faces
BackFaces
Wireframe
Face Normals
EOF Render
High Quality
Clear Object
Refresh
Rotate view:
normal mode: move Mouse
OEF render: drag Mouse
Scale (Zoom): Mouse-Wheel
Load Meshes by selecting an item from the ComboBox above.
Fill Faces: Draw filled faces
Back faces: Draw faces viewed from behind
Wireframe: Draw lines around all Faces
Face Normals: Draw small lines for each face showing in the face's direction
OEF Render: Render "onEnterFrame"
High Quality: Flash Player's output quality

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 (aka. 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 $\triangleq$ bit string of length $6{\cdot}N^2$
  - A normal $\triangleq$ 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 256×8 = 2048 Bits

```
typedef struct PolygonNormalMask
{
   Byte offset, bitMask;
};
```

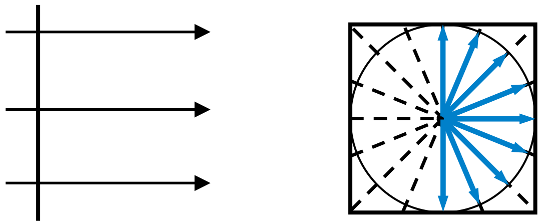
0....00$\boxed{00010000}$00.....0

offset
(in Bytes)

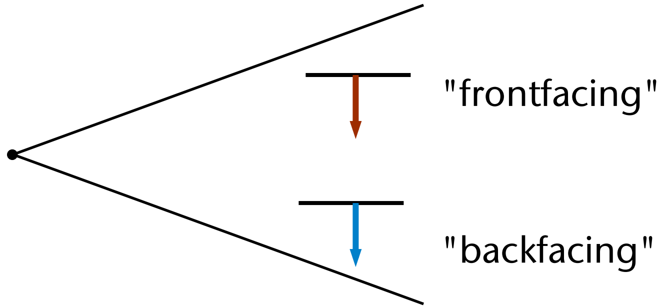
bitMask

  - 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

- Culling (initialization):
  - Identify all those normal classes whose normals are all backfacing
  - With orthographic projection:

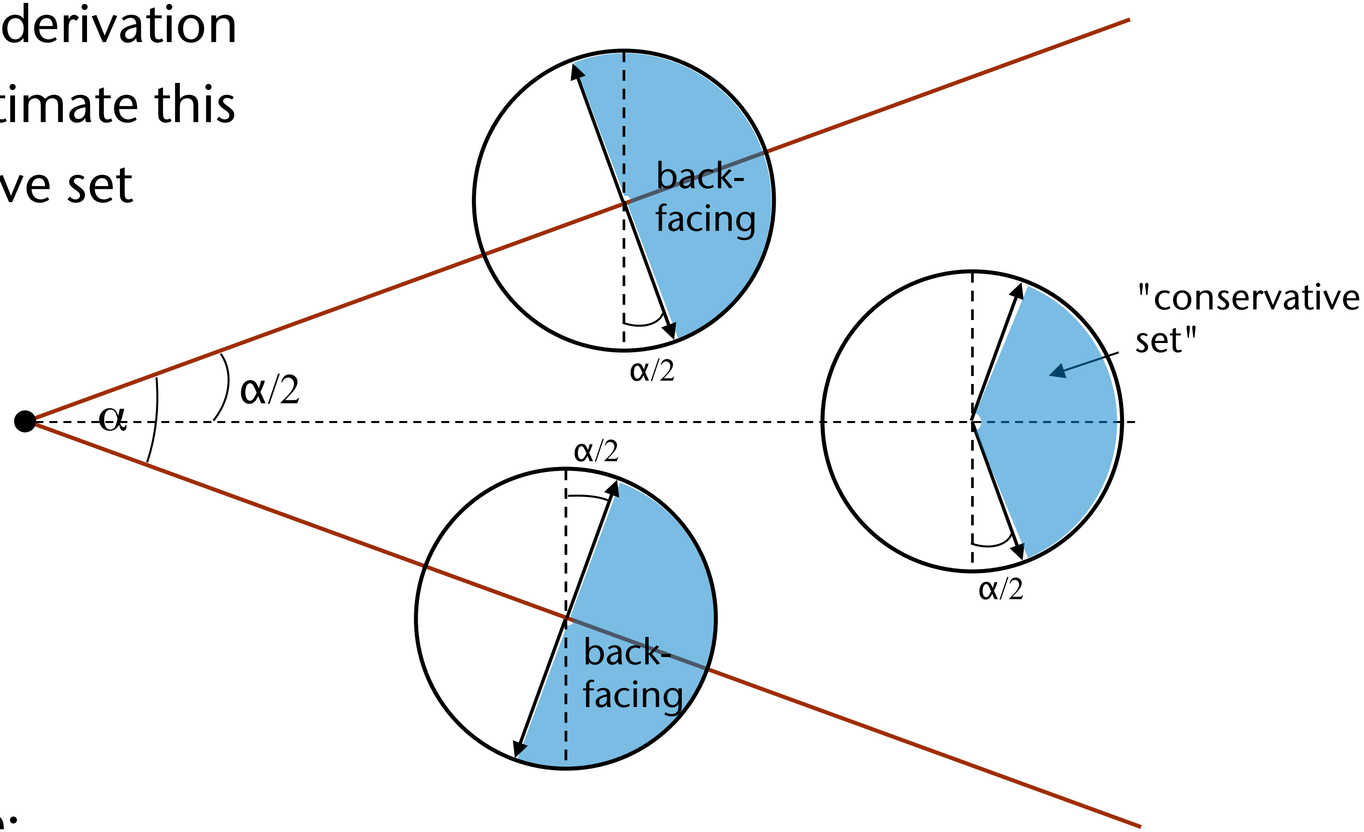


  - With perspective projection: which normals are backfacing depends on normal direction and position of the polygon!



  - Therefore: determine a "conservative" set of classes which are backfacing – regardless of the location of the polygon

- Graphical derivation how to estimate this conservative set of classes:



- 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:

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

- Culling (runtime): test for each polygon

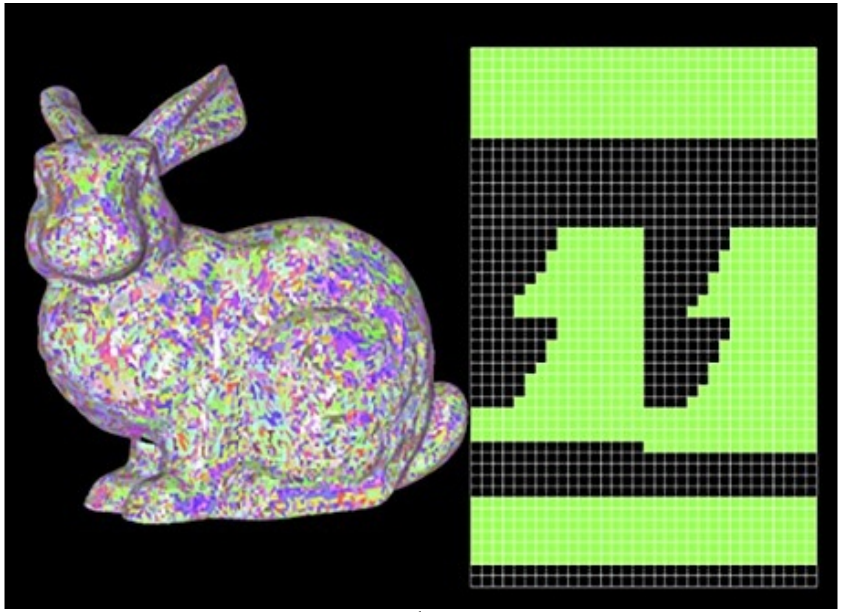
```
if ( (BackMask[byteOffset] & polygon.bitMask) == 1 )
    render polygon
```

- Further acceleration:
  - Divide view frustum into sectors
  - Render the scene "sector by sector"
  - Thus, the angle $\alpha/2$ in each sector is smaller
  - For each sector, compute its own BackMask**[]**

Example

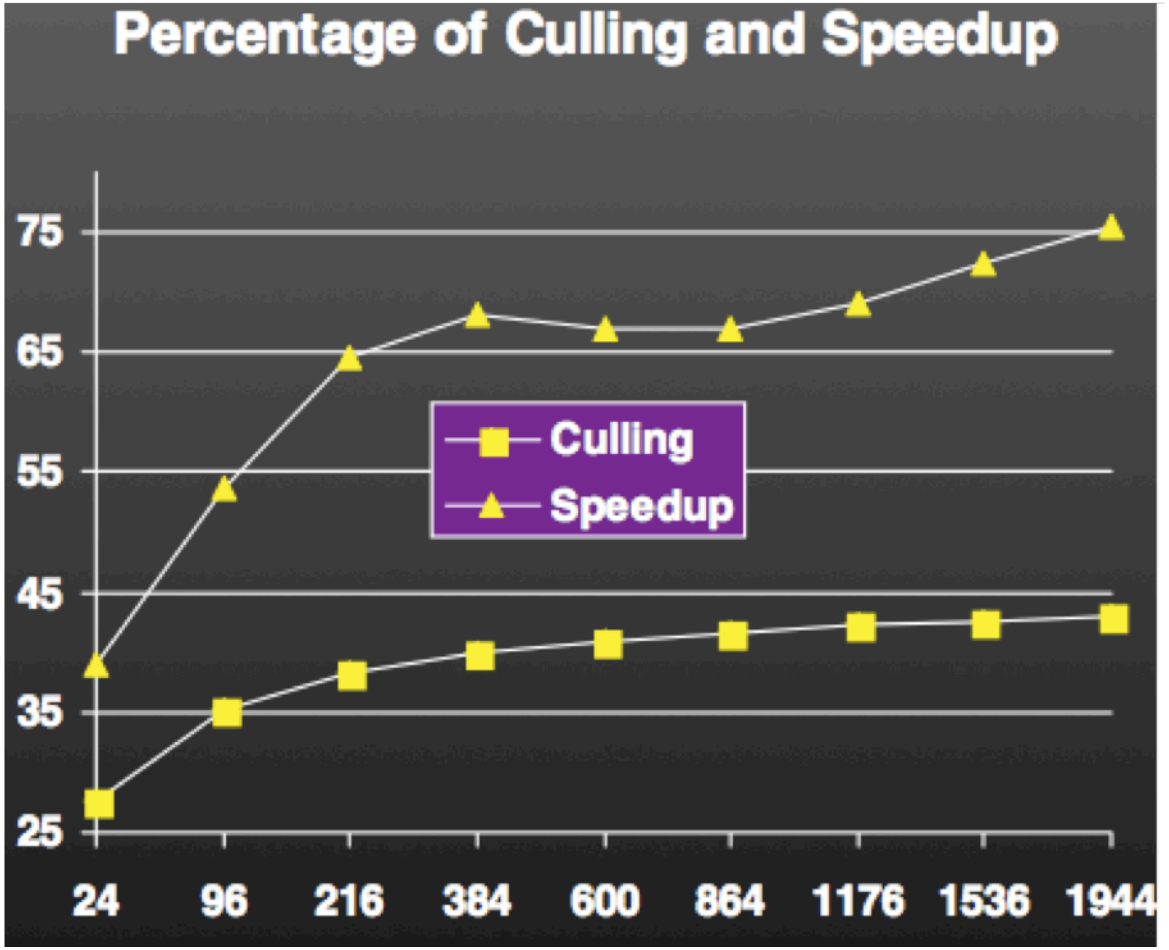
216 classes ("clusters")

1536 classes ("clusters")



BackMask for the current viewpoint
(green = backfacing)

Speedup



Number of normals classes

Result: speedup factor ~1.5 compared to OpenGL backface culling

Clustered Backface Culling

- Reminder: some simple rules for min/max

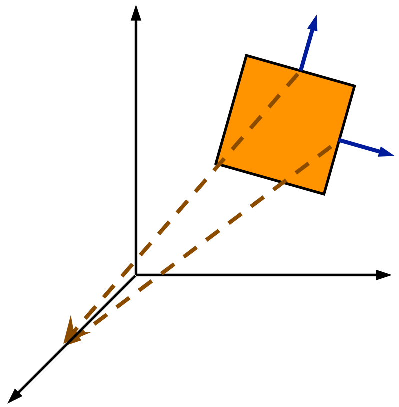
$$\max_i \{x_i + y_i\} \le \max_i \{x_i\} + \max_i \{y_i\}$$

$$\max_i \{x_i - y_i\} \le \max_i \{x_i\} - \min_i \{y_i\}$$

$$\max_i \{kx_i\} = \begin{cases} k \max_i \{x_i\} & , \ k \ge 0 \\ k \min_i \{x_i\} & , \ k < 0 \end{cases}$$

- In the following, $\mathbf{n}^i$ and $\mathbf{p}^i$ are the normal and a vertex of a polygon from a cluster (a set) of polygons; let $\mathbf{e}$ be the viewpoint
- Attention: in the following, we use the "inverted" definition for backfacing!

$$\mathbf{n} \cdot (\mathbf{e} - \mathbf{p}) \le 0$$



- Assumption: cluster (= set) of polygons is given
- All polygons in cluster are backfacing if and only if

$$\forall i : \mathbf{n}^i \left(\mathbf{e} - \mathbf{p}^i\right) \leq 0 \quad \Leftrightarrow$$

$$\max \left\{ \mathbf{n}^i \left(\mathbf{e} - \mathbf{p}^i\right) \right\} \leq 0 \tag{1}$$

- Upper bound for (1) is

$$\max \left\{ \mathbf{n}^i \left(\mathbf{e} - \mathbf{p}^i\right) \right\} \leq \max \left\{ \mathbf{e}\mathbf{n}^i \right\} - \min \left\{ \mathbf{n}^i \mathbf{p}^i \right\} \tag{2}$$

- Set $d := \min\{ \boldsymbol{n}^i \cdot \boldsymbol{p}^i \}$ (pre-computation)
- Write (2) as

$$\begin{aligned} \max \left\{ \mathbf{n}^i \left(\mathbf{e} - \mathbf{p}^i\right) \right\} &\leq \max \left\{ e_x n^i_x + e_y n^i_y + e_z n^i_z \right\} - d \\ &\leq \max \left\{ e_x n^i_x \right\} + \max \left\{ e_y n^i_y \right\} + \max \left\{ e_z n^i_z \right\} - d \end{aligned} \tag{3}$$

- Assumption: **e** is located in the positive octant, i.e., $e_x, e_y, e_z \geq 0$; then we can give an upper bound of (3):

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

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

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

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

$$\max\left\{ \mathbf{n}^i \left(\mathbf{e} - \mathbf{p}^i \right) \right\} \leq \mathbf{m}' \cdot \mathbf{e} - d \; , \quad \text{with} \quad \mathbf{m}' = \begin{pmatrix} \min\{n_x^i\} \\ \min\{n_y^i\} \\ \min\{n_z^i\} \end{pmatrix}$$

- For all other octants, combine min and max appropriately
- We can write this with kind of a "combination" operator on vectors

$$\text{comb}(\mathbf{u}, \mathbf{v}; \mathbf{e}) := \mathbf{w} \quad \text{with} \quad w_\alpha = \begin{cases} u_\alpha & , e_\alpha \leq 0 \\ v_\alpha & , e_\alpha > 0 \end{cases} \quad , \; \alpha \in \{x, y, z\}$$

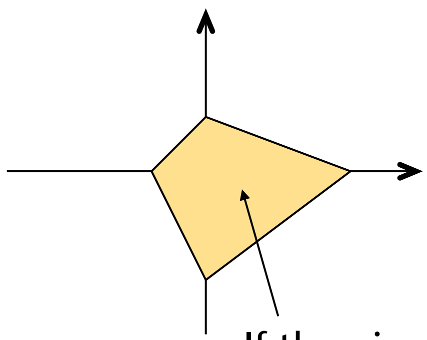
- This allows us write the (conservative) test as:

$$\text{comb}(\mathbf{m}', \mathbf{m}; \mathbf{e}) \cdot \mathbf{e} - d \leq 0 \quad \Rightarrow \quad \text{cluster is backfacing} \tag{4}$$

- Pre-computation: for each cluster determine **m**, **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 octants with $\boldsymbol{e}_x \geq 0$
  - All 4 planes have normals of the form $\boldsymbol{n} = (m_x, \cdot\,, \cdot\,)$
  - So, they all intersect the x-axis at the point $\left(\frac{d}{m_x}, 0, 0\right)$.
- Those 8 planes form a closed volume, the so-called culling volume



If the viewpoint is in the culling volume, then the cluster is completely backfacing

Further Optimization: Test 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 $\longrightarrow$ the test is never positive
- Solution: run the test in a *local coordinate system* by
- Move the local origin $\mathbf{c}$ such that

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

  is as large (and positive) as possible
- Wanted is the optimal $\mathbf{c}$
  - Question: Will rotation achieve something?
  - In practice: Try the center and corner of the BBox of the cluster as $\mathbf{c}$
- Save $\mathbf{c}$ with the cluster, then test

$$\text{comb}(\mathbf{m}', \mathbf{m}; \mathbf{e} - \mathbf{c}) \cdot (\mathbf{e} - \mathbf{c}) - d \leq 0$$

Hierarchical Clustered Backface Culling

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

$$\hat{\mathbf{m}}' = \begin{pmatrix} \min(m'^1_x, m'^2_x) \\ \min(m'^1_y, m'^2_y) \\ \min(m'^1_z, m'^2_z) \end{pmatrix} \qquad \hat{\mathbf{m}} = \begin{pmatrix} \max(m^1_x, m^2_x) \\ \max(m^1_y, m^2_y) \\ \max(m^1_z, m^2_z) \end{pmatrix}$$

$$\hat{d} = \min(d_1, d_2)$$

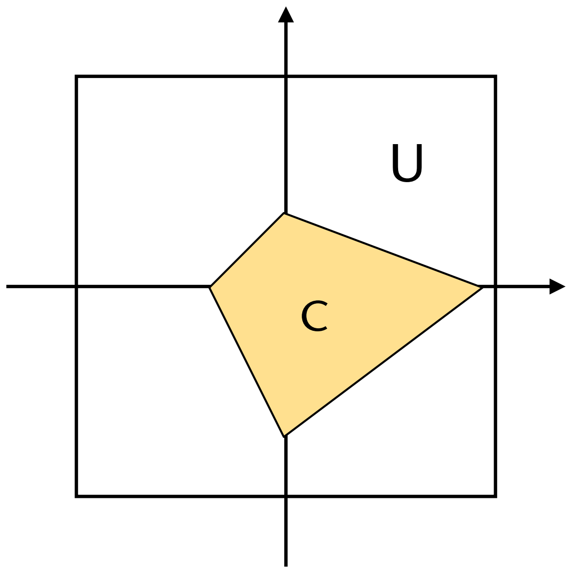
  - These two vectors and $\hat{d}$ provide a conservative estimate
  - I.e.: if the joint cluster is invisible, then the two original clusters are guaranteed to be invisible, too → cluster hierarchy
- If a hierarchy of clusters is created, define a front-facing test, analogously to the backfacing test:
  - 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 will be culled
- Use a heuristic to calculate P :

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

  - 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 A and B separately,

  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*) = probability, that cluster *A* gets culled,

  R(*A*) = time to render *A* (without further tests), and

  t = time for backface test of a cluster

- So, combining clusters *A* and *B* is worth it, if and only if

$$T(C) < T(A) + T(B) \quad \Leftrightarrow$$

$$t + (1 - P(C))\,R(C) < 2t + (1 - P(A))\,R(A) + (1 - P(B))\,R(B) \quad \Leftrightarrow$$

$$P(C) > \frac{-t + P(A)R(A) + P(B)R(B)}{R(A) + R(B)} \quad \Leftrightarrow$$

$$P(C) > \frac{P(A)n_A + P(B)n_B - \frac{t}{r}}{n_A + n_B}$$

Assumption:
$R(A) = n_A \cdot r$,
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, ...)