Graphics & Visualization

Chapter 5

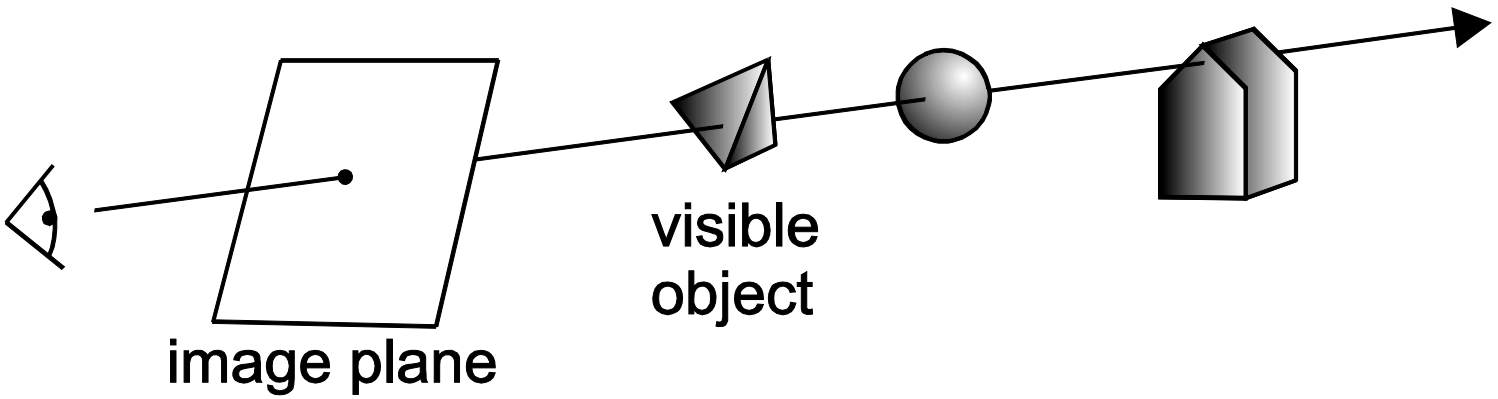
CULLING AND HIDDEN SURFACE ELIMINATION ALGORITHMS

Introduction

- Due to restrictions pertaining to our field of view as well as occlusions among the objects, we can only see a tiny portion of the objects that make up our world
- *Culling* algorithms remove primitives that are not relevant to the rendering of a specific frame because:
  - They are outside the field of view (*frustum culling*)
  - They are occluded by other objects (*occlusion culling*)
  - They are occluded by front-facing primitives of the same object (*back-face culling*)
- Frustum culling:
  - Removes primitives that are outside the field of view
  - Implemented by 3D clipping algorithms

Introduction (2)

- Back-face culling:
  - Removes primitives that are hidden by front-facing primitives of the same object
  - Uses normal vectors
- The occlusion problem:
  - *Determination of the visible object in every part of the image*
  - Can be solved by computing the first object intersected by each relevant ray emanating from the viewpoint
  - For correct rendering we must solve the occlusion problem



Introduction (3)

- Numerous Hidden Surface Elimination (HSE) algorithms have been proposed to solve the occlusion problem
  - Their purpose is to eliminate hidden surfaces of the objects → deal with the occlusion problem
- HSE algorithms involve sorting of the primitives:
  - Sorting in the Z (depth) dimension is essential as visibility depends on depth order
  - Sorting in the X, Y dimensions can accelerate the task of Z sorting, as primitives which do not overlap in X, Y can not possibly occlude each other
- HSE algorithms are classified according to their working space:
  - Object space
  - Image space

Introduction (4)

- General form of object space HSE:

```
for each primitive
{
   find visible part //(compare against all other primitives)
   render visible part
}
```

- Complexity is $O(P^2)$, where P is the number of primitives
- General form of image space HSE:

```
for each pixel
{
   find closest primitive
   render pixel with color of closest primitive
}
```

- Complexity is O(pP), where p is the number of image pixels

Introduction (5)

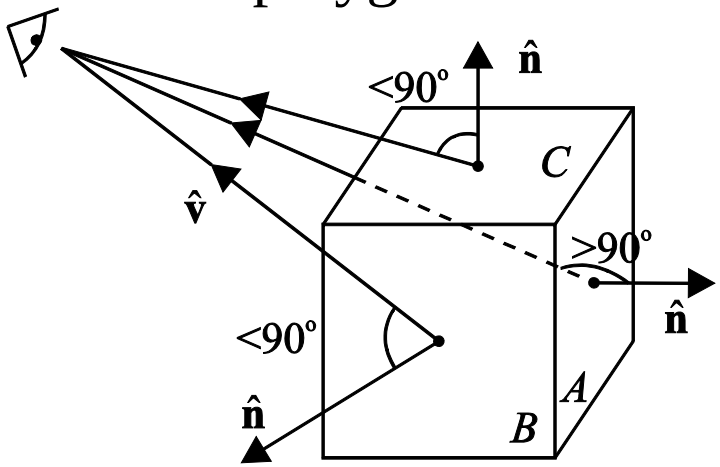
- Computational cost of HSE algorithms is overwhelming despite hardware implementations and parallel processing
- Large numbers of primitives can easily be discarded without HSE algorithms
- Back-face culling eliminates approximately half of the primitives (back-faces) by a single test at a cost of O(P)
- Frustum culling removes those remaining primitives that fall outside the field of view at a cost of O(Pv), where v is the average number of vertices per primitive
- Occlusion culling also costs O(P) in the usual case
- CONCLUSION: the performance bottleneck are the HSE algorithms which cost $O(P^2)$ or O(pP) depending on their working space

Back-Face Culling

- The visible polygons of an object, are those that lie on the hemisphere facing the viewer
- If models are constructed in such a way that the back sides of polygons are never visible, then we can cull such polygons. The requirements for this are:
  - Object surfaces have no boundary (closed)
  - They are 2D manifolds
  - They are opaque
- Convexity is not a constraint
- Back faces can be detected by computing the angle formed by a polygon's normal vector $\hat{\mathbf{n}}$ (pointing outwards from the opaque solid) and the view vector $\hat{\mathbf{v}}$

Back-Face Culling (2)

- If angle > 90º then then the polygon is a back face



- The back-face test uses the inner product of the vectors :

$$\hat{\mathbf{v}} \cdot \hat{\mathbf{n}} < 0$$

- Back face culling is extremely effective as it eliminates about 50% of the polygons
- Since the back face test and the computation of the normal and view vectors for each polygon take constant time, the complexity is O(P), where P is the number of polygons

Frustum Culling

- The viewing transformation defines the field of view of the observer
- The field is restricted by a minimum and maximum depth value, defining a 3D solid called *view volume* or *view frustum*
- The form of the frustum varies:
  - Truncated pyramid {perspective projection}
  - Rectangular parallelepiped {orthographic or parallel projection}
- Objective is to eliminate primitives outside the view volume
- Takes place after the transformation from ECS to CSS, before the division by *w* (for perspective projection)
- Frustum culling must be performed in 3D space using 3D clipping

Frustum Culling (2)

- Objects to be clipped:
  - Points
  - Line segments
  - polygons
- Point clipping is trivial
- Line segment and polygon clipping reduce to the computation of a line segment with the planes of the clipping object
- In 3D, the interior of the clipping object is defined as:

$$x_{min} \leq x \leq x_{max}$$
$$y_{min} \leq y \leq y_{max} \qquad \textbf{(1)}$$
$$z_{min} \leq z \leq z_{max}$$

Frustum Culling (3)

- In orthographic or parallel projection we use $\mathbf{M}_{\mathbf{ECS}\rightarrow\mathbf{CSS}}^{\mathbf{ORTHO}}$ matrix which maps the clipping planes to -1 and 1 so that:

$$x_{min} = y_{min} = z_{min} = -1$$

$$x_{max} = y_{max} = z_{max} = 1$$

- In perspective projection, the $\mathbf{M}_{\mathbf{ECS}\rightarrow\mathbf{CSS}}^{\mathbf{PERSP}}$ matrix (before the division by *w*) maps the clipping planes to –*w* and *w* so that:

$$x_{min} = y_{min} = z_{min} = -w$$

$$x_{max} = y_{max} = z_{max} = w$$

- The value of *w* is not constant (equal to a point's z_e)
- Clipping against *w* is called *homogeneous clipping*

Frustum Culling (4)

- For a parametric line segment:

$$\mathbf{l}(t) = (1-t)\mathbf{p_1} + t\mathbf{p_2}$$

from $\mathbf{p_1} = [x_1, y_1, z_1, w_1]^T$ to $\mathbf{p_2} = [x_2, y_2, z_2, w_2]^T$,

the value of w can be interpolated as :

$$(1-t)w_1 + tw_2$$

then the inequalities **(1)** can be used to define the part of the line segment within the clipping object:

$$-((1-t)w_1 + tw_2) \le (1-t)x_1 + tx_2 \le (1-t)w_1 + tw_2$$
$$-((1-t)w_1 + tw_2) \le (1-t)y_1 + ty_2 \le (1-t)w_1 + tw_2 \quad \textbf{(2)}$$
$$-((1-t)w_1 + tw_2) \le (1-t)z_1 + tz_2 \le (1-t)w_1 + tw_2$$

Frustum Culling (5)

- Solving the 6 inequalities for t we get the 6 intersection points of the line segment with the clipping object planes:

$$left: t = \frac{x_1 + w_1}{(x_1 - x_2) + (w_1 - w_2)} \qquad right: t = \frac{x_1 - w_1}{(x_1 - x_2) + (w_2 - w_1)}$$

$$bottom: t = \frac{y_1 + w_1}{(y_1 - y_2) + (w_1 - w_2)} \qquad top: t = \frac{y_1 - w_1}{(y_1 - y_2) + (w_2 - w_1)} \qquad \textbf{(3)}$$

$$near: t = \frac{z_1 + w_1}{(z_1 - z_2) + (w_1 - w_2)} \qquad far: t = \frac{z_1 - w_1}{(z_1 - z_2) + (w_2 - w_1)}$$

3D Clipping Algorithms

- Most 2D clipping algorithms extend easily to 3D space by addressing:
  - The intersection computation
  - The inside / outside test

3D Cohen – Sutherland Line Clipping

- In 3D, 6 bits are used to code the 27 partitions of 3D space defined by the view frustum planes:

First bit: Set to 1 for $z > z_{max}$, else set to 0

Second bit: Set to 1 for $z < z_{min}$, else set to 0

Third bit: Set to 1 for $y > y_{max}$, else set to 0

Fourth bit: Set to 1 for $y < y_{min}$, else set to 0

Fifth bit: Set to 1 for $x > x_{max}$, else set to 0

Sixth bit: Set to 1 for $x < x_{min}$, else set to 0.

3D Clipping Algorithms (2)

- A 6-bit code can be assigned to a 3D point according to which of the 27 partitions of 3D space it lies in
- Let c_1, c_2 be the 6-bit codes of the endpoints $\mathbf{p_1}$, $\mathbf{p_2}$ of a line segment:
  - If $c1 \vee c2 = 000000$ accept line segment
  - If $c1 \wedge c2 \neq 000000$ reject line segment
- 3D CS pseudocode:

```
CS_Clip_3D ( vertex p1, p2 )
{
   int c1, c2;          vertex I;          plane R;
   c1 = mkcode (p1);  c2 = mkcode (p2);
   if ((c1 | c2) == 0) /* p1p2 is inside */
   else if ((c1 & c2) != 0) /* p1p2 is outside */
   else { R = /* frustum plane with (c1 bit != c2 bit) */
         i = intersect_plane_line (R, (p1,p2));
         if outside (R, p1) CS_Clip_3D(i, p2);  else CS_Clip_3D(p1, i);}
}
```

3D Clipping Algorithms (3)

- Differs from the 2D algorithm in the:
  - intersection computation
  - outside test
- Intersection computation:
  - A 3D plane-line intersection computation is used instead of a 2D line-line intersection computation
  - The clipping limits are not given in the pseudocode:
    - In orthographic or parallel projection, these are constant planes and plane-line intersection algorithms are used
    - In perspective projection and homogeneous coordinates, the plane-line intersections of equations **(3)** are used
- Outside test:
  - Can be implemented by a sign test on the evaluation of the plane equation R with the coordinates of $\mathbf{p}_1$

3D Clipping Algorithms (4)

3D Liang – Barsky Line Clipping

- The line segment to be clipped is represented by its starting and ending points $\mathbf{p_1}$ and $\mathbf{p_2}$ (as before)
- In orthogonal or parallel projection, the clipping object is a cube and the LB computations extend directly to 3D by adding a z-coordinate inequality: $z_{min} \le z_1 + t\Delta z \le z_{max}$
- For perspective projection and homogeneous coordinates, we use the inequalities **(2)** which define the part of the a parametric line segment within the clipping object:

$$-(w_1 + t\Delta w) \le x_1 + t\Delta x \le w_1 + t\Delta w$$
$$-(w_1 + t\Delta w) \le y_1 + t\Delta y \le w_1 + t\Delta w$$
$$-(w_1 + t\Delta w) \le z_1 + t\Delta z \le w_1 + t\Delta w$$

where $\Delta x = x_2 - x_1, \Delta y = y_2 - y_1, \Delta z = z_2 - z_1$

3D Clipping Algorithms (5)

- These inequalities have the common form $tp_i \leq q_i$ for i= 1,2,…,6 where:

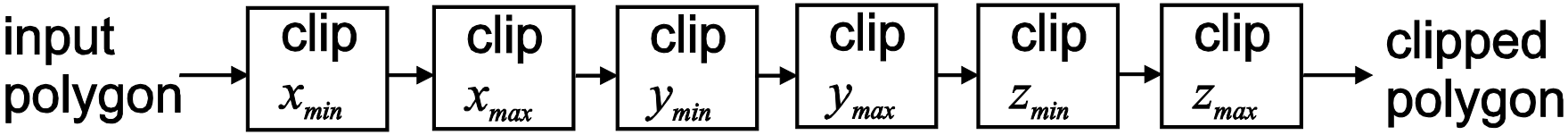
$$p_1 = -\Delta x - \Delta w \qquad q_1 = x_1 + w_1$$
$$p_2 = \Delta x - \Delta w \qquad q_2 = w_1 - x_1$$
$$p_3 = -\Delta y - \Delta w \qquad q_3 = y_1 + w_1$$
$$p_4 = \Delta y - \Delta w \qquad q_4 = w_1 - y_1$$
$$p_5 = -\Delta z - \Delta w \qquad q_5 = z_1 + w_1$$
$$p_6 = \Delta z - \Delta w \qquad q_6 = w_1 - z_1.$$

- Notice that the ratios (q_i / p_i) correspond to the parametric intersection values of the line segment with clipping plane *i*
- The rest of the LB algorithm remains as in 2D

3D Clipping Algorithms (6)

3D Sutherland – Hodgman Polygon Clipping

- In 3D the clipping object is a convex volume, the view frustum, instead of a convex polygon
- The algorithm consists of 6 pipelined stages, one for each face of the view frustum

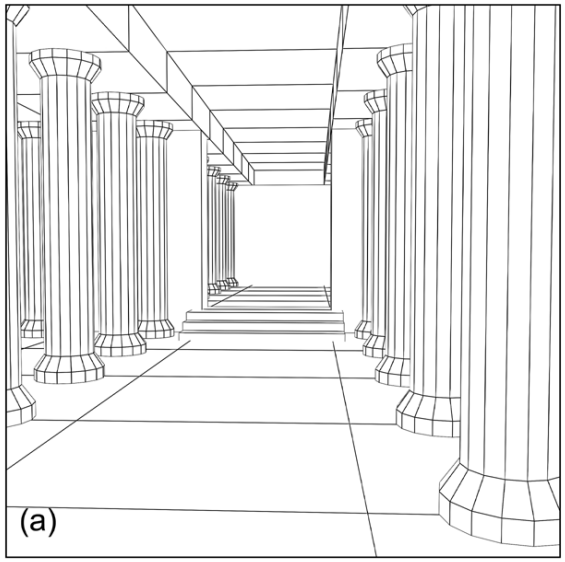


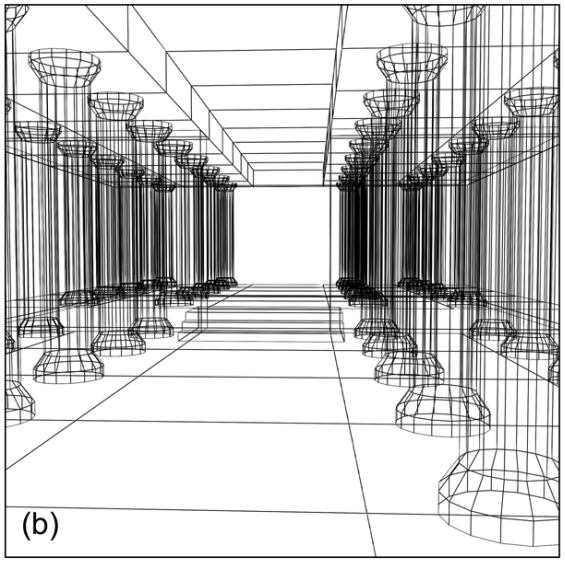
- Main differences against the 2D algorithm are:
  - The `inside_test` subroutine must be altered so that it tests whether a point is on the inside half-space of a plane → equivalent to testing the sign of the plane equation for the coordinates of the point
  - The `intersect_lines` subroutine must be replaced by `intersect_plane_line` to compute the intersection of a polygon edge against a plane of the clipping volume → equations **(3)** are used for perspective projection

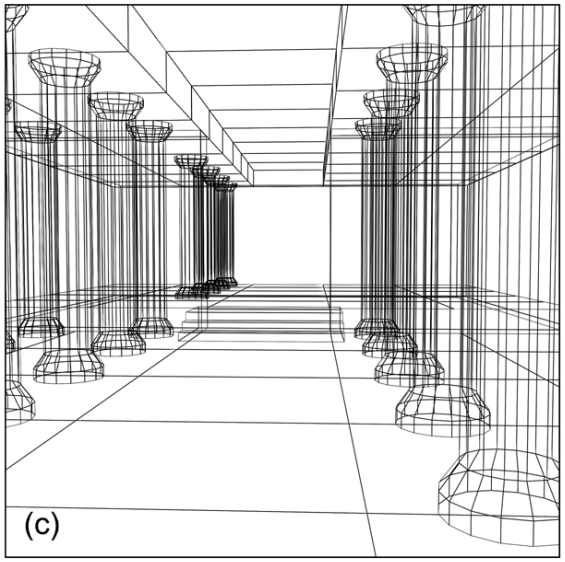
Occlusion Culling

- Occlusion culling discards primitives hidden by other primitives nearer to the observer
- Aims at efficiently discarding a large number of primitives before computationally expensive HSE algorithms are applied
- *Visible set* is the subset of primitives that are rendered on at least one pixel of the final image
- Occlusion culling algorithms compute a tight superset of the visible set so that the rest of the primitives can be discarded
- This superset is called *potentially visible set* (PVS)
- Occlusion culling does not expend time in determining *exactly* which parts of the primitives are hidden → HSE algorithms do
- Instead it determines which primitives are entirely NOT visible and quickly discards those, computing the PVS

Occlusion Culling (2)







(a) visible set

(b) all primitives

(c) PVS

Occlusion Culling (3)

- PVS is then passed to HSE algorithms
- Occlusion culling costs O(P), where P is # of primitives
- Two main categories of occlusion culling:
  - From-point occlusion culling:
    - Solve the occlusion problem for a single viewpoint
    - Suitable for outdoor scenes
  - From-region occlusion culling:
    - Solve the occlusion problem for an entire region of space
    - Suitable for dense indoor scenes
    - Suitable for static scenes because of the pre-computation required

From-Region Occlusion Culling

- A number of applications consists of a set of convex regions (*cells*), connected by transparent *portals*
  - Simplest form: scene represented by 2D floor plan and cells and portals are parallel to either x or y
- Primitives are only visible between cells via portals
  - Cell c_a may be visible from cell c_b via cell c_m, if appropriate sightlines exist that connect their portals
- Algorithm requires a pre-processing step
  - Its cost is only paid once assuming the cells and portals to be static
- At pre-processing, a PVS matrix and a BSP tree are constructed
- PVS matrix:
  - Gives the PVS for every cell that the viewer may be in
  - Visibility is symmetric → PVS matrix is symmetric

From-Region Occlusion Culling (2)

- It is constructed, starting from each cell *c* and recursively visiting all cells reachable from the cell adjacency graph, while sightlines exist that allow visibility from *c*
- Thus the *stab tree* of *c*, which defines the PVS of *c*, is constructed

- BSP tree:
  - Uses separating planes, which may be cell boundaries, to recursively partition the scene
  - Leafs represent cells
  - A balanced BSP tree can be used to quickly locate the cell that a point lies in, in $O(\log_2 n_c)$ time, where n_c is the # of cells
- At runtime, the steps that lead to the rendering of the PVS for a viewpoint **v** are:
  - 1. Determine cell *c* of **v** using the BSP tree
  - 2. Determine PVS of cell *c* using PVS matrix
  - 3. Render PVS

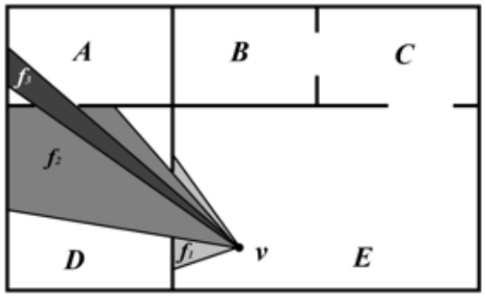
From-Region Occlusion Culling (3)

(a) Scene modeled as cells and portals

(b) Stab trees of the cells

(c) PVS matrix

(d) BSP tree

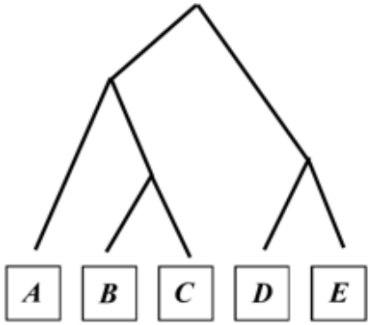


(a)

(b)

	A	B	C	D	E
A	1			1	1
B		1	1		1
C		1	1	1	1
D	1		1	1	1
E	1	1	1	1	1

(c)



(d)

- PVS does not change while **v** remains in the same cell
- The first 2 steps are only executed when **v** crosses a cell boundary
- At runtime, only the BSP tree and PVS matrix are used

From-Region Occlusion Culling (4)

- Occlusion culling can be optimized by combining it with frustum and back-face culling
  - Rendering can be restricted to primitives that are both within the view frustum and the PVS
  - View frustum must be recursively constricted from cell to cell on the stab tree
- Pseudocode:

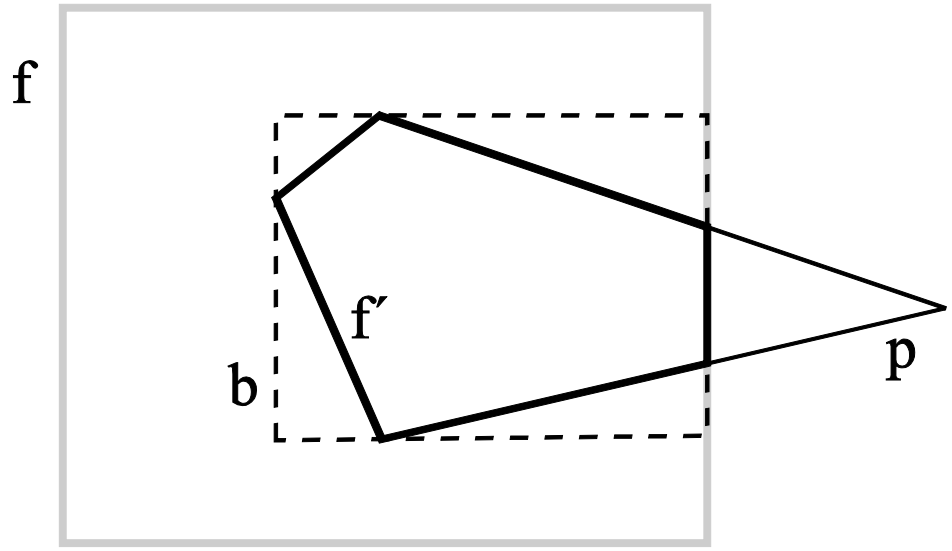
```
main()
{
   determine cell c of viewpoint using BSP tree;
   determine PVS of cell c using PVS matrix;
   f = original view frustum;
   portal render(c, f, PVS);
}
```

From-Region Occlusion Culling (5)

```
portal render(cell c, frustum f, list PVS);
{
    for each polygon R in c {
        if ((R is portal) & (c' in PVS)) {
                /* portal R leads to cell c' */
                /* compute new frustum f' */
                f' = clip frustum(f, R);
                if (f' <> empty) portal render(c', f', PVS);
        }
        else if (R is portal) {}
        else {  /* R is not portal */
                /* apply back-face cull */
                if (!back face(R)) {
                        /* apply frustum cull */
                        R' = clip poly(f, R);
                        if (R' <> empty) render(R');
                }
        }
    }
}
```

From-Region Occlusion Culling (6)

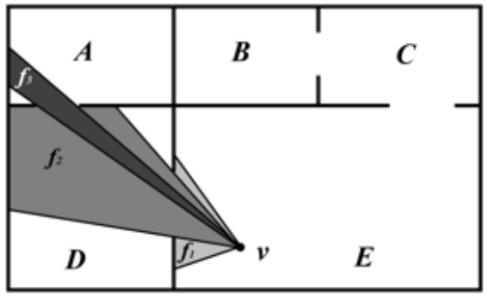
- `f' = clip frustum(f, R)` command computes the intersection of the current frustum *f* and the volume formed by the viewpoint and the portal polygon *R*
  - If this produces odd convex shapes, we may lose the ability to use hardware support
  - A solution is to replace *f'* by its bounding box



From-Region Occlusion Culling (7)

EXAMPLE:

- Cell *E*, that the viewer **v** lies in, is first determined
- Objects in that cell are culled against original frustum f_1
- The first portal leading to PVS cell *D* constricts the frustum to f_2
- Objects within cell *D* are culled against the new frustum
- The second portal leading to cell *A* reduces to frustum f_3

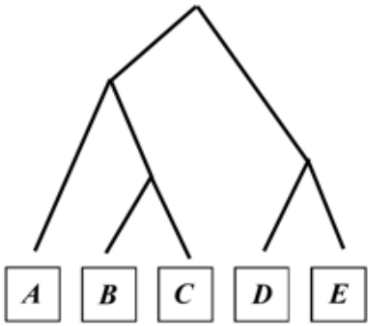


(a)

(b)

	A	B	C	D	E
A	1			1	1
B		1	1		1
C		1	1	1	1
D	1		1	1	1
E	1	1	1	1	1

(c)



(d)

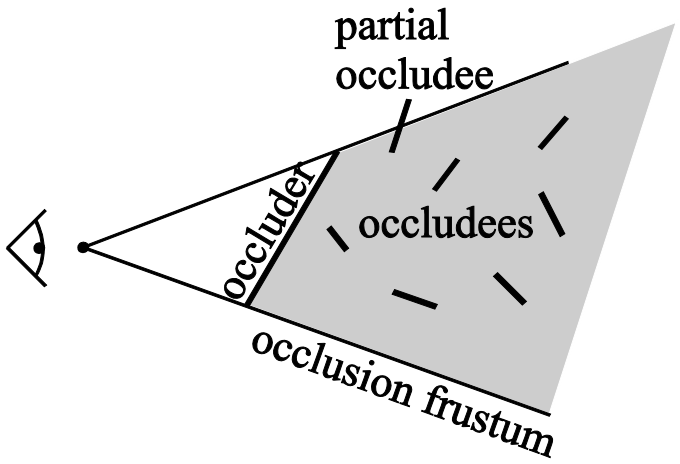
- Objects within cell *A* are culled against f_3 frustum
- Recursive process stops here as there are no new portal polygons within f_3 frustum

From-Point Occlusion Culling

- In outdoor scenes it cannot be assumed that the scene consists of cells and portals
- Partitioned regions of such scenes would not be coherent, with regard to their occlusion properties
- From-point occlusion culling solves the problem for a single viewpoint → does not require as much pre-processing as from-region occlusion culling, since PVS is not computed
- Main idea behind from-point techniques is the *occluder*:
  - Occluder is a primitive, or combination of primitives, that occludes a large number of other primitives (called *occludees*) with respect to a certain viewpoint
  - Region of space defined by the viewpoint and the occluder is the *occlusion frustum*

From-Point Occlusion Culling (2)

- Primitives that lie entirely within the occlusion frustum can be culled
- Partial occludees must be referred to the HSE algorithms



- Two main steps are required to perform occlusion culling for a viewpoint **v**:
  - Create a small set of good occluders for **v**
  - Perform occlusion culling using these occluders

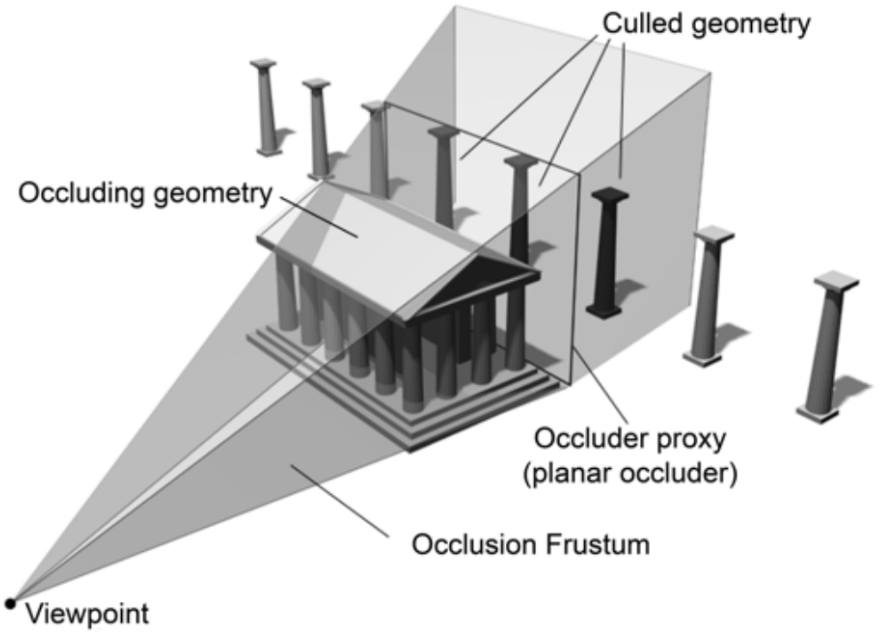
From-Point Occlusion Culling (3)

- Planar occluders ranked according to the area of screen space projections were used by Coorg and Teller
  - Their ranking function is:

$$f_{planar} = \frac{-A(\hat{\mathbf{n}} \cdot \hat{\mathbf{v}})}{\left|\vec{\mathbf{v}}\right|^2}$$

    where A is the area of a planar occluder, $\hat{\mathbf{n}}$ is its unit normal vector and $\hat{\mathbf{v}}$ is the vector from the viewpoint to the center of the plane occluder
  - A usual way of computing a planar occluder is as the proxy for a primitive or object
  - The proxy is a convex polygon perpendicular to the view direction inscribed within the occlusion frustum of the occluder object or primitive

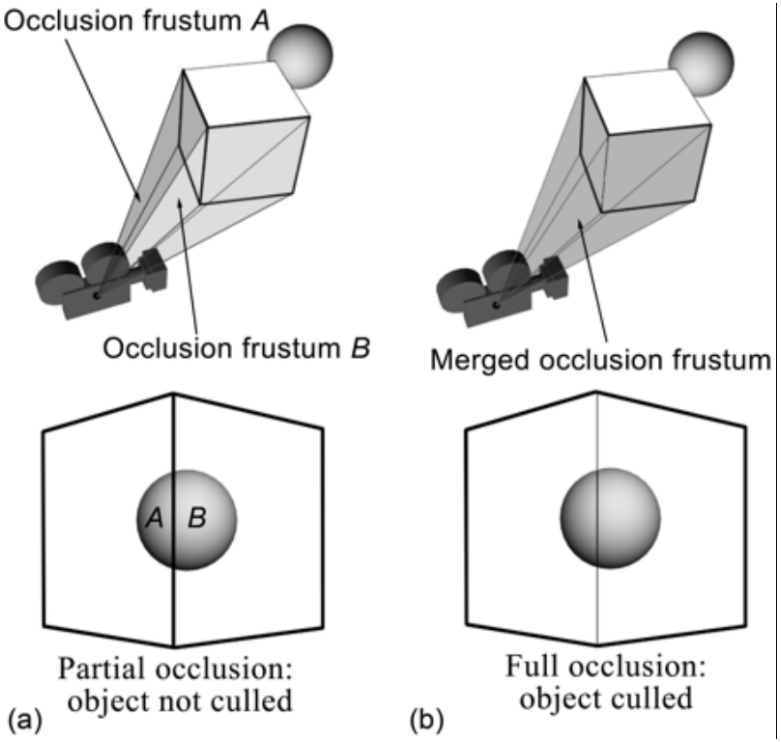
From-Point Occlusion Culling (4)



- Occlusion culling step can be made more efficient by keeping a hierarchical bounding volume description of the scene
- Starting at the top level, a bounding volume that is entirely inside/outside an occlusion frustum is rejected/rendered
- A bounding volume that is partially inside and partially outside is split into the next level of bounding volumes

From-Point Occlusion Culling (5)

- Simple occlusion culling suffers from the problem of partial occlusion:
  - An object may not lie in the occlusion frustum of any individual primitive → cannot be culled, although it may lie in the occlusion frustum of a combination of adjacent primitives
- For this reason algorithms that merge primitives or their occlusion frusta have been developed:
  - Papaioanou proposed an extension to the basic planar occluder method, called solid occluders, to address the partial occlusion problem, by dynamically producing a planar occluder for the entire volume of an object



Hidden Surface Elimination

- HSE algorithms must provide a complete solution to the occlusion problem
- Primitives or parts of primitives that are visible must be rendered directly
- HSE algorithms sort the primitives intersected by the projection rays
- For occlusion this reduces to the comparison of 2 points:

$$\mathbf{p_1}=[x_1, y_1, z_1]^T \text{ and } \mathbf{p_2}=[x_2, y_2, z_2]^T$$

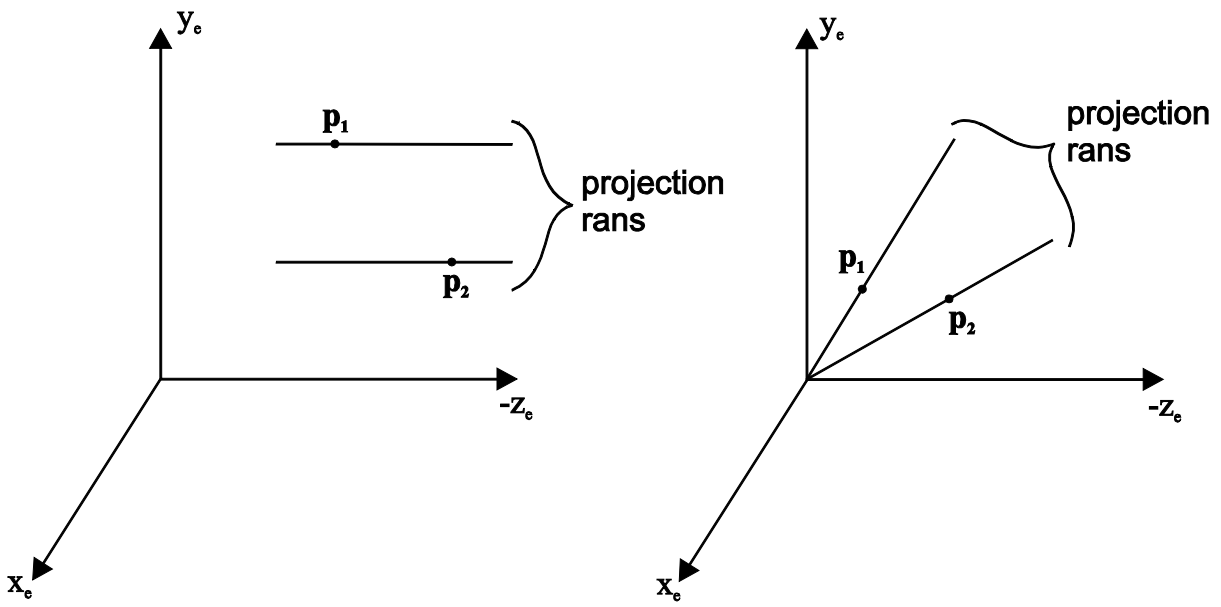
  - If they are on the same ray then they form an *occluding pair* → the nearer one will occlude the other
- There are 2 cases:
  - Orthogonal projection: assuming that projection rays are parallel to the z-axis, 2 points form an occluding pair if:

$$(x_1 = x_2) \text{ and } (y_1 = y_2)$$

Hidden Surface Elimination (2)

- Perspective projection: perspective division must be performed to determine if 2 points form an occluding pair. The condition now is:

$$(x_1 / z_1 = x_2 / z_2) \text{ and } (y_1 / z_1 = y_2 / z_2)$$



- In perspective projection, the costly perspective division is performed anyway within the ECS to CSS part of the viewing transformation

Hidden Surface Elimination (3)

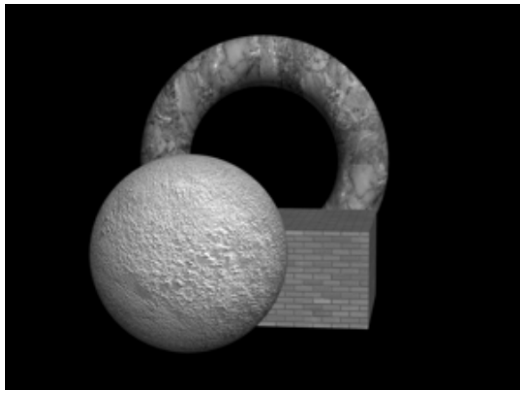
- It transforms the perspective view volume into a rectangular parallelepiped → make direct comparisons of x and y coordinates for the determination of occluding pairs
- For this reason HSE takes place after the viewing transformation into CSS
- Z coordinates are maintained for the purpose of HSE during the viewing transformations
- Most HSE algorithms take advantage of *coherence* → intersection calculations are replaced by incremental computation:
  - Surface coherence
  - Object coherence
  - Scanline coherence
  - Edge coherence
  - Frame coherence

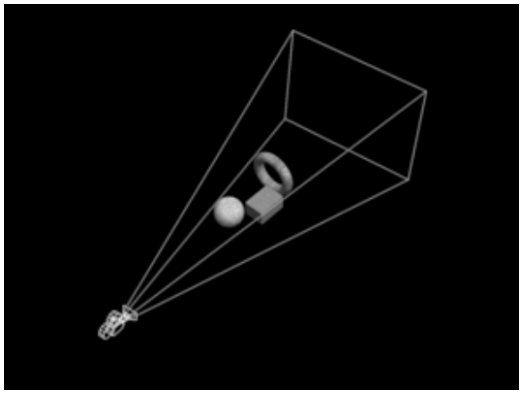
Z – Buffer Algorithm

- Initially dismissed because of its high memory requirements
- Today a hardware implementation of the Z-buffer algorithm can be found on every graphics accelerator
- Algorithm maintains a 2D memory of depth values (Z-buffer), with the same spatial resolution as the frame buffer
- 1:1 correspondence between frame and Z–buffer elements
- Every element of the Z-buffer maintains the minimum-depth-so-far for the corresponding pixel of the frame buffer
- Z-buffer is initialized to a maximum, usually the far clipping plane
- For each primitive, during rendering, we compute (z_p, c_p) at a pixel $p = (x_p, y_p)$, where z_p is the depth of the primitive at p, and c_p its color at p

Z – Buffer Algorithm (2)

frame – buffer Z – buffer 3D scene





- Assuming that depth values decrease as we move away from the view point, the main Z-buffer test is:

```
if (z-buffer[xp, yp] < zp)
{
   f-buffer[xp, yp] = cp; /* update frame buffer */
   z-buffer[xp, yp] = zp; /* update depth buffer */
}
```

Z – Buffer Algorithm (3)

- Computation of the depth z_p at each pixel that a primitive covers, is an efficiency issue of the Z-buffer algorithm:
  - Computing the intersection of the ray defined by the viewpoint and the pixel with the primitive is expensive
  - We take advantage of surface coherence. Let the plane equation of the primitive be:

$$F(x, y, z) = ax + by + cz + d = 0$$

Solving for the depth (z) we get:

$$F'(x, y) = z = - d/c - (a/c)x - (b/c)y$$

F' is incrementally computed from pixel (x, y) to pixel (x+1, y) since:

$$F'(x+1, y) – F'(x, y) = - a/c$$

- By adding the constant first forward difference of F' in x or y, we compute the depth value from pixel to pixel at a cost of one addition

Z – Buffer Algorithm (4)

- In practice, depth values at vertices of the planar primitive are interpolated across its edges and then across the scanlines
- Same argument applies to the color vector
- Complexity of Z-buffer algorithm is O(P*s*), where P # of primitives and *s* average # of pixels covered by a primitive
- In practice, as P increases, *s* decreases proportionally → O(p), where p # of pixels
- Advantages of the algorithm:
  - Simplicity
  - Constant performance
- Weaknesses of the algorithm:
  - Difficulty to handle some special effects (transparency)

Z – Buffer Algorithm (5)

- Result has fixed resolution, inherited from its image space nature
- Z-fighting: arithmetic depth sorting inaccuracies for wide clipping ranges

- Z-buffer computed during a rendering, can be kept and used in various ways:
  - Allows depth merging of 2 or more images. Suppose (f_a, z_a) and (f_b, z_b) represent the frame- and Z-buffers of 2 parts of a scene. These can be merged by selecting the part with the nearest depth value at each pixel:

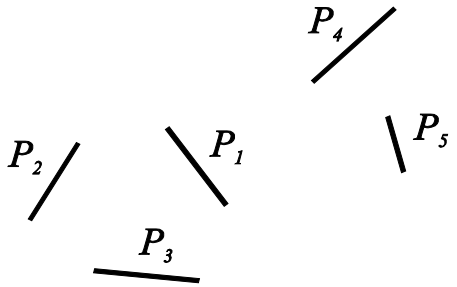
```
for (x=0; x<XRES; x++){
        for (y=0; y<YRES; y++) {
        Fc[x, y] = (Za[x, y]>Zb[x, y])? Fa[x, y]:Fb[x, y];
        Zc[x, y] = (Za[x, y]>Zb[x, y])? Za[x, y]:Zb[x, y];
        }
}
```

Z – Buffer Algorithm (6)

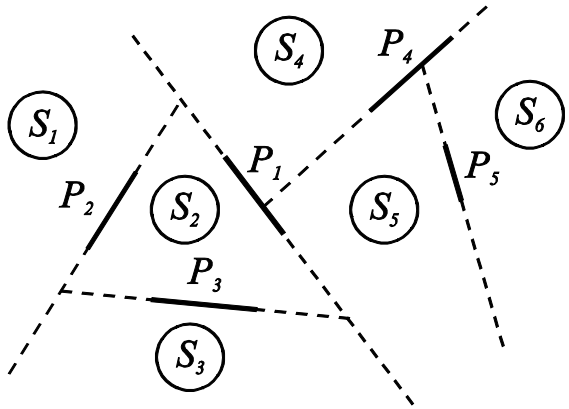
- Many more computations can be performed using Z-buffer:
  - Shadow determination
  - Voxelization
  - Voronoi computation
  - Object reconstruction
  - Symmetry detection
  - Object retrieval

BSP Algorithm

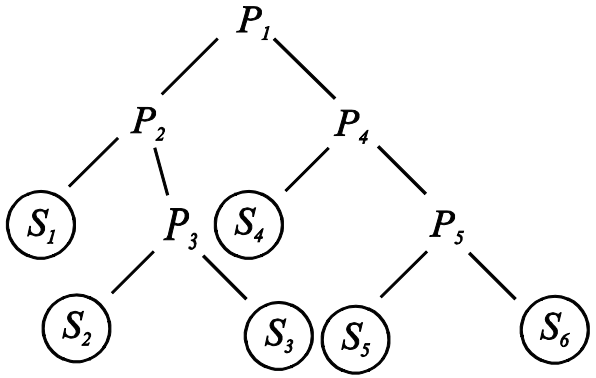
- Binary Space Partitioning (BSP) is an object space algorithm that uses a binary tree that recursively subdivides space
- The node data represent polygons of the scene
- Internal nodes split space by the plane of their polygon, so that children on the left subtree are on one side of the plane and children on the right subtree are on the other



(a)



(b)



(c)

BSP Algorithm (2)

- To construct a BSP tree the following algorithm is used:

```
BuildBSP(BSPnode, polygonDB);
{
   Select a polygon (plane) Pi from polygonDB;
   Assign Pi to BSPnode;
   /* Partition scene polygons into those that lie on either side
   of plane Pi, splitting polygons that intersect Pi */
   Partition(Pi, polygonDB, polygonDBL, polygonDBR);
   if (polygonDBL != empty) BuildBSP(BSPnode->Left,polygonDBL);
   if (polygonDBR != empty) BuildBSP(BSPnode->Right,polygonDBR);
}
```

- The selection of the partitioning plane P_i is critical since we would like to create a balanced BSP tree
- A plane is therefore selected, that divides the scene in 2 parts of roughly equal cardinality

BSP Algorithm (3)

- During partitioning, polygons that intersect the partitioning plane must be split to enforce the partitioning → can be achieved by extending a clipping algorithm to deliver both the "inside" and the "outside" parts of a clipped polygon
- BSP trees can be used to display the scene with the hidden surfaces removed
- For a viewpoint **v** and a BSP node, all polygons that lie in the same partition as **v** cannot possibly be hidden by polygons of the other partition → polygons of the other partition should be displayed first

BSP Algorithm (4)

- Pseudocode:

```
DisplayBSP(BSPnode, v);
{
    if IsLeaf(BSPnode) Render(BSPnode->Polygon)
    else if (v in 'left' subspace of BSPnode->Polygon) {
          DisplayBSP(BSPnode->Right, v);
          Render(BSPnode->Polygon);
          DisplayBSP(BSPnode->Left, v);
    }
    else /* v in 'right' subspace of BSPnode->Polygon */ {
          DisplayBSP(BSPnode->Left, v);
          Render(BSPnode->Polygon);
          DisplayBSP(BSPnode->Right, v);
    }
}
```

BSP Algorithm (5)

- The `DisplayBSP` algorithm costs O(P)
- The `BuildBSP` algorithm costs $O(P^2)$
- The overall complexity of the BSP algorithm is $O(P^2)$
- For static scenes, `BuildBSP` is used once and then, for every new position of the viewpoint, only the `DisplayBSP` algorithm must run → BSP is suitable for static scenes, but NOT suitable for dynamic scenes

Depth Sort Algorithm

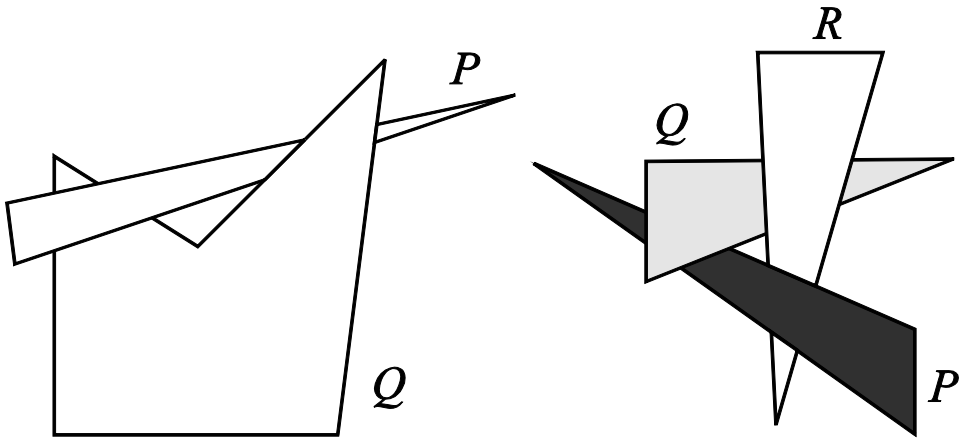
- This algorithm sorts polygons according to their distance from the observer and displays them in reverse order (back to front)
- Also called *painters' algorithm*
- Minimum depth value of polygons is used for the sorting:

```
DepthSort(polygonDB);
{
   /* Sort polygonDB according to minimum z */
   for each polygon in polygonDB find MINZ and MAXZ;
   sort polygonDB according to MINZ;
   resolve overlaps in z;
   display polygons in order of sorted list;
}
```

- Overlaps in z arise when the z extents of polygons overlap → the sorting becomes ambiguous as it is not clear which polygon obscures the other → cannot perform sorting in some cases

Depth Sort Algorithm (2)

- Not sortable polygons:



- When the *z* extents of two polygons *R* and *Q* overlap, a sequence of tests of increasing complexity are employed to resolve the ambiguity of their order in the display list
- A positive conclusion of one of the following tests (increasing complexity order) establishes that *Q* can not be occluded by *R:*
  1. The *x* extents of *R* and *Q* do not overlap
  2. The *y* extents of *R* and *Q* do not overlap

Depth Sort Algorithm (3)

3. R lies entirely in the half–space of Q which does not include the viewpoint **v**. This can be established by checking that the sign of the plane equation of Q is the same for all vertices of R and different to its sign for **v**:

$$\text{sign}(f_Q(\mathbf{r_i})) \neq \text{sign}(f_Q(\mathbf{v})), \quad \forall\, \mathbf{r_i} \in R$$

where

$$f_Q(x, y, z) = a_Q x + b_Q y + c_Q z + d_Q = 0$$

is the plane equation for polygon Q

4. Q lies entirely in the half–space of R which includes the viewpoint **v**. This can be established by checking that the sign of the plane equation of R is the same for all vertices of Q and for **v**:

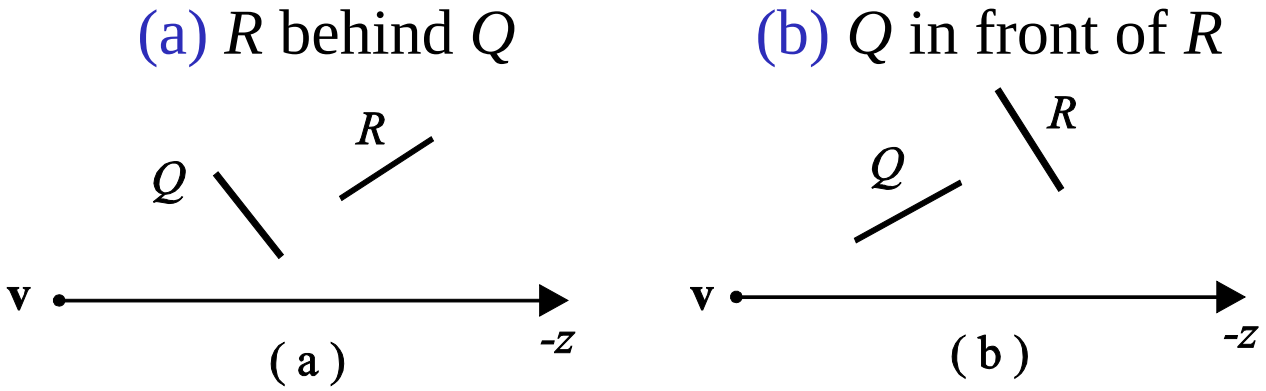
$$\text{sign}(f_Q(\mathbf{q_i})) = \text{sign}(f_Q(\mathbf{v})), \quad \forall\, \mathbf{q_i} \in Q$$

where

$$f_R(x, y, z) = a_R x + b_R y + c_R z + d_R = 0$$

is the plane equation for polygon R

Depth Sort Algorithm (4)



5. The projections of *R* and *Q* do not overlap

- If none of the above tests is positive, the roles of *R*, *Q* are swapped and tests 3, 4 are repeated
- Tests 1, 2, 5 are symmetric → no need to repeat
- If the order is still not resolved, then *R* is split into 2 polygons using the plane of *Q*, the new polygons replace *R* and the process is repeated

Depth Sort Algorithm (5)

- Depth sort is an object space algorithm except for the last step (image space)
- An optimization is to draw the polygons from front to back in the display step:
  - Succeeding polygons are only drawn on pixels that have not already been written to by nearer polygons
  - The display step can stop as soon as all image pixels have been written to at least once
- The cost of the sorting step is $O(P\log_2 P)$
- Resolution of *z* overlaps could cost $O(P^2)$ in the worst case
- Depth Sort is a slow algorithm in typical high complexity scenes
- Positive side:
  - Depth Sort algorithm can straightforwardly handle transparency

Ray – Casting Algorithm

- A ray is followed for every pixel p
- Ray is defined by the viewpoint **v** and the vector $\vec{\mathbf{p}} - \vec{\mathbf{v}}$
- Intersections with all scene primitives are computed and the intersection nearest to **v** defines the visible primitive
- Basic algorithm:

```
RayCasting(primitiveDB, v); {
    for each pixel p {
        minp = MAXINT;
        for each primitive R in primitiveDB {
            /* compute intersection of ray (v,p) with R */
            I = intersect_primitive_ray(R,v,p);  /* MAXINT if none */
            if (|i-v| < minp) {
                p -> nearest_primitive = R;
                minp = |i-v|
            }
        }
    }
}
```

Ray – Casting Algorithm (2)

- Ray casting takes no advantage of coherence → it is slow, O(pP)
- It is a very general method
- Can be sped up in a straightforward manner by distributing the rays among parallel processors
- Ray casting algorithm can be applied before or after perspective projection:
  - In the former case the rays are the projection rays
  - In the latter case the rays are parallel to each other and orthogonal to the projection plane
- Ray casting can be classified as either object or image space

Summary

- The complexities and application spaces of the presented HSE algorithms are given in the following table:

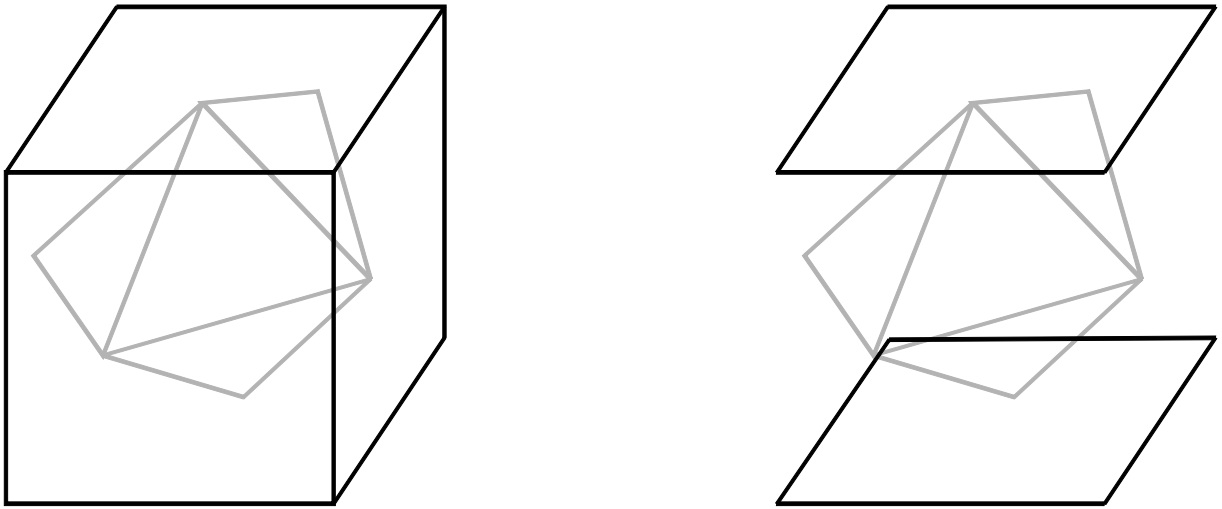
HSE algorithm	Complexity	Space
Z-Buffer	$O(Ps) \simeq O(p)$	Image
BSP	$O(P^2)$	Object
Depth Sort	$O(P^2)$	Object
Ray Casting	$O(pP)$	Image/Object

Efficiency Issues

- Culling, HSE, Ray Tracing and other algorithms, require intersection computations
- Intersection computations are expensive
- Techniques developed to accelerate intersection computations:
  - Bounding volumes
  - Space subdivision

Bounding Volumes

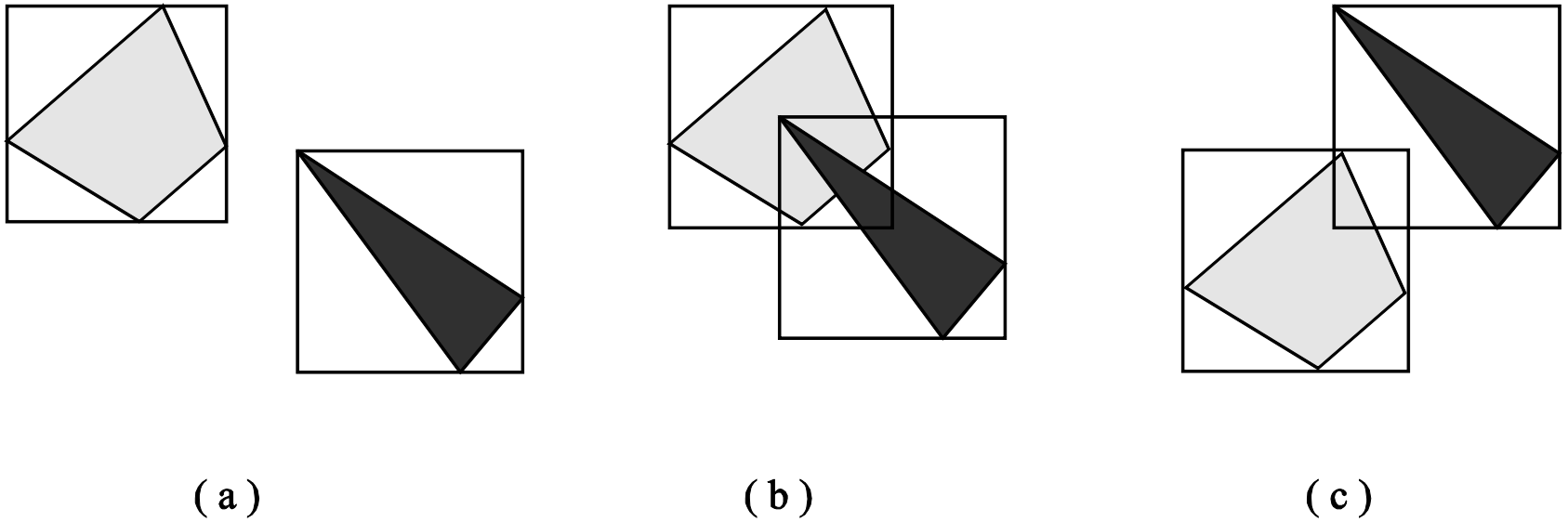
- A way to reduce the cost of computing intersections with a complex model, is to cluster its primitives in a bounding volume
- A bounding volume could be a rectangular parallelepiped, or a sphere
- A bounding volume need not be closed



(a) (b)

Bounding Volumes (2)

- Intersection with a bounding volume does not necessarily imply intersection with the model, as the volume includes *void space* between itself and the model → False alarm
- Non intersection with the bounding volume DOES imply no intersection with the model involved

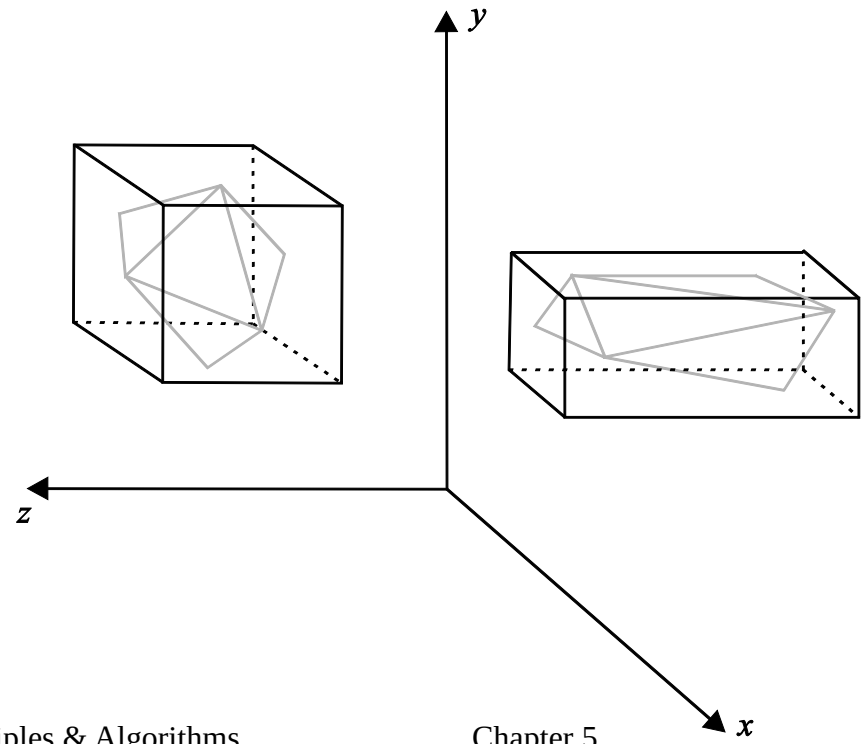


Bounding Volumes (3)

- Whenever the bounding volume is not intersected, no model intersection tests take place → save computational effort
- Bounding volumes must possess 2 qualities to be successful:
  - Be simple
  - Minimize void space
- The first quality makes the intersection tests against the bounding volume efficient
- The second quality ensures that as few false alarms as possible are generated
- The achievement of both qualities is contradictory:
  - A compromise has to be achieved

Bounding Volumes (4)

- Rectangular parallelepiped bounding volumes are created by taking the minima and the maxima of the model's vertex coordinates
- Defined by six planes perpendicular to the coordinate axis and thus called *axis-aligned bounding boxes* (AABBs)
  - They include a large amount of void space

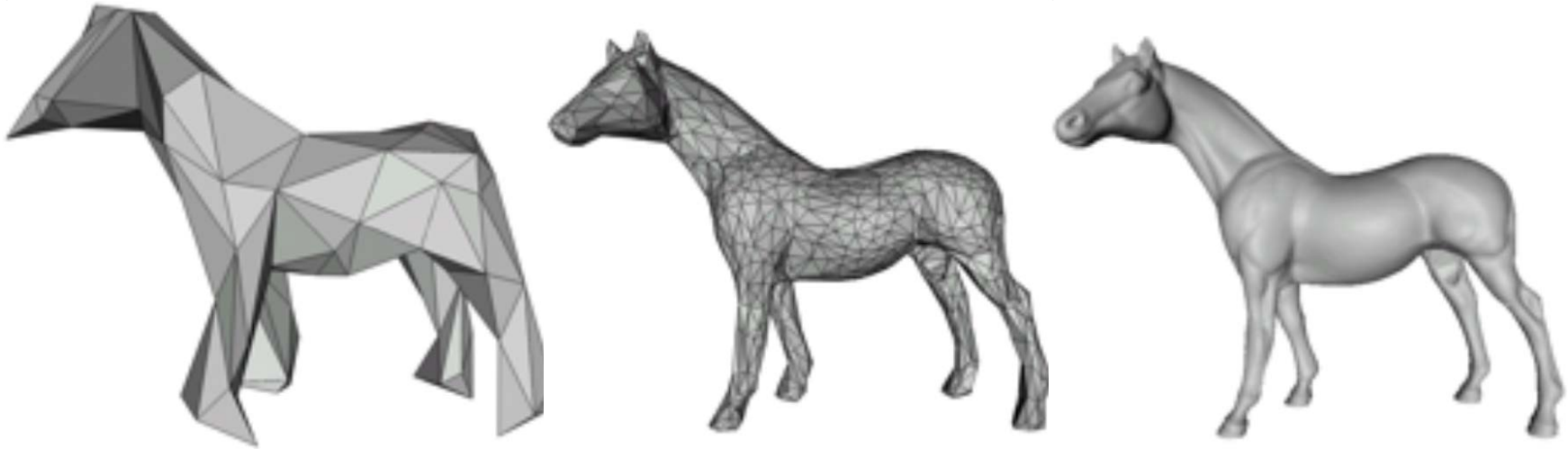


Bounding Volumes (5)

- *Oriented bounding boxes* (OBBs) are arbitrarily oriented rectangular parallelepipeds
  - OBBs include less void space than AABBs
- Hierarchical bounding volumes provide a better compromise between simplicity and void space
  - These include hierarchies of k-DOPs (polyhedra whose faces may only have predefined orientations) and hierarchies of OBBs
  - Trees of nested volumes are constructed
    - Root represents a bounding volume containing the whole model
    - Nodes represent smaller subvolumes
    - Leaves represent individual primitives
  - Tree structure restricts the area of potential intersections

Bounding Volumes (6)

- Another hierarchical method is *progressive hulls*
  - Succession of hulls that enclose the model more tightly
  - Each hull encloses all subsequent hulls
  - All the hulls enclose the model
  - Outer hulls are simpler but have more void space
  - Inner hulls are more complex but have less void space
  - Hulls are used starting from the outmost (simplest), while intersections are found



Bounding Volumes (7)

- Pseudocode for hierarchical intersection test of a model *M*

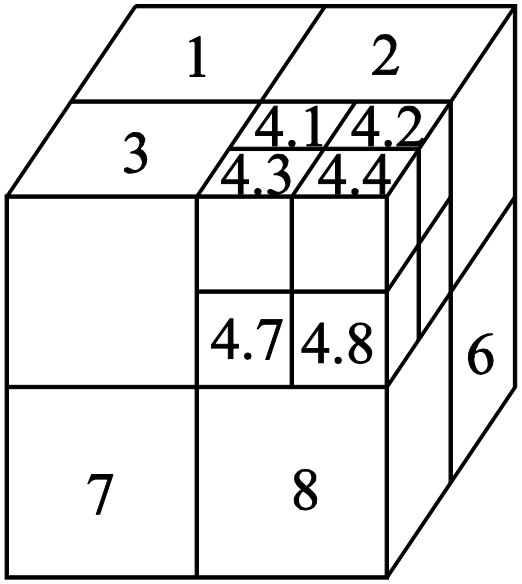
```
IntersectionTest(M) {
   if BottomLevel(M)    return(LLIntersectionTest(M))
   else if LLIntersectionTest(BoundingVolume(M)) {
        v = false;
        for each component M->C
                v = (v || IntersectionTest(M->C));
        return(v);
        }
   else return(false);
}
```

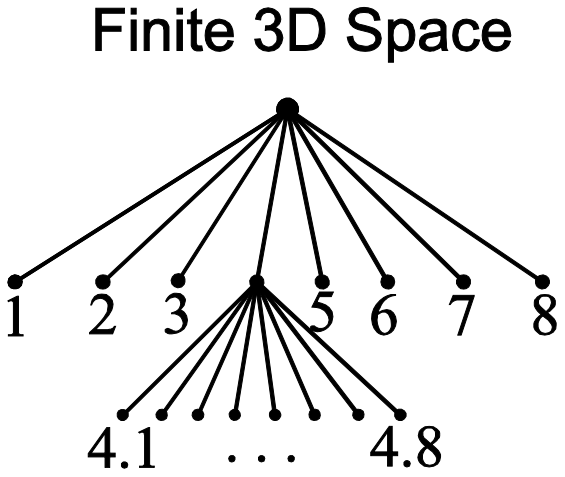
- `LLIntersectionTest`: performs an exhaustive intersection test with the primitives of its parameter
- `M->C`: represents a component one level below in the object hierarchy

Space Subdivision

- Divide space into an ordered set of cells
- Cells indirectly determine the spatial relationship between objects that occupy them
  - 2 objects potentially intersect if they occupy common cells
  - Ordering of the cells infers if an object potentially occludes another object
- Space subdivision techniques require special data structures and processing steps to assign objects to these structures
- *Octree* is a common hierarchical 3D space subdivision technique
  - Recursively subdivides an initial cell into 8 subcells
  - The subdivision stops:
    - When an elementary cell size (*voxel*) is reached
    - When the object complexity within a cell is below a certain limit

Space Subdivision (2)





- In a culling application, models that do not occupy the cells of interest are discarded:
  - In frustum culling only models that occupy cells common to the view frustum are considered
  - In occlusion culling only objects that occupy cells with the same X, Y coordinates are considered
- Octree has more levels where higher scene complexity exists