Managing Complexity 2 – levels

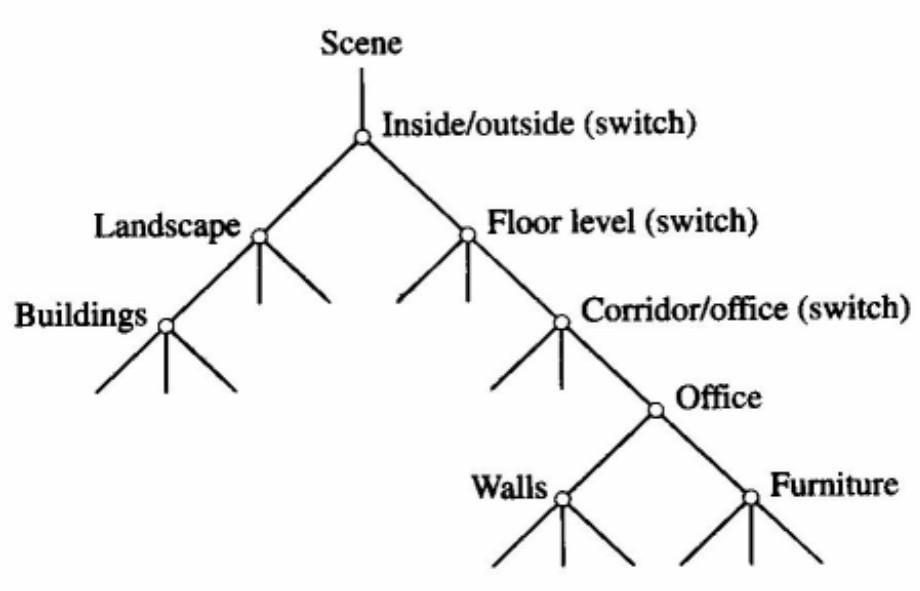
Visibility processing and early culling

LECTURE 5 Visibility processing of complex scenes

- Current games (particularly first person shooters) have complex static levels plus dynamic objects of varying complexity plus animated camera. Visibility processing based on structures set up as part of the build process was one of the techniques in early games that enabled complex levels to be navigated. (The other foundation technique was pre-calculated lighting.)
- In real time we need:
  - Fast visibility processing of static level
  - Fast processing of dynamic objects and lights
  - Fast collision detection

Visibility processing of static levels

- Use secondary data structures (built in a pre-processing step) strategies are usually:
  - Subdivide a scene into partitions
  - Organise the scene into a hierarchy



Trees as secondary data structures

- **Octrees** axis aligned partitioning. Divide a cubic space into 8, then each of these partitions into 8 etc.
- **B**(inary)**S**(space)**P**(partitioning) trees – usually non-axis aligned – divide the space into 2 partitions, then each of these into 2 etc

BSP trees: John Carmack – DOOM (early 90s)

- **Bounding volume** hierarchies

Partitions trees are hierarchical secondary data structures where each leaf of a tree points into an object data structure, or into a group of polygons that comprises that part of the object which occupies the space defined by the leaf.

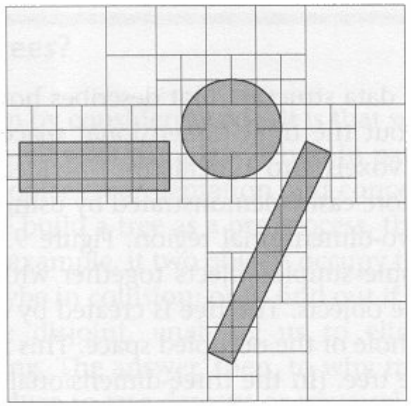
Standard problem is: insert the coordinates of the view point position at the top of the tree and descend to the leaf node to find a possible object ordering in terms of visibility.

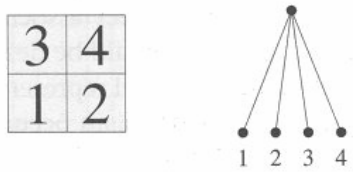
Therefore must have efficient partitioning to build a tree of low average depth.

First step in a hierarchy of different tests to reveal visibility (those parts of the scene that are contained within the view frustum)

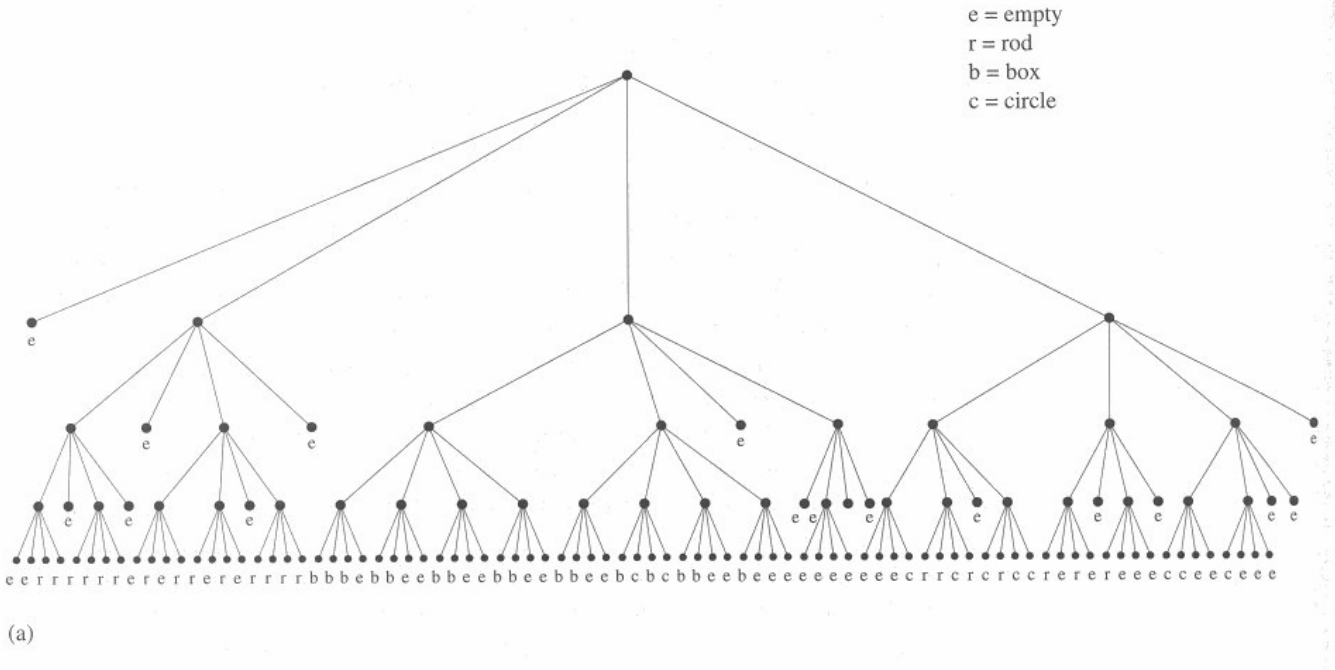
Octrees (quadtrees in 2D) – partitioning to the finest cell level

Figure 9.4
(a) Quadtree representation of a two-dimensional scene at the pixel level. A similar method is used to represent a three-dimensional scene by an octree.
(b) Ordering scheme for child nodes in quadtree illustrations.



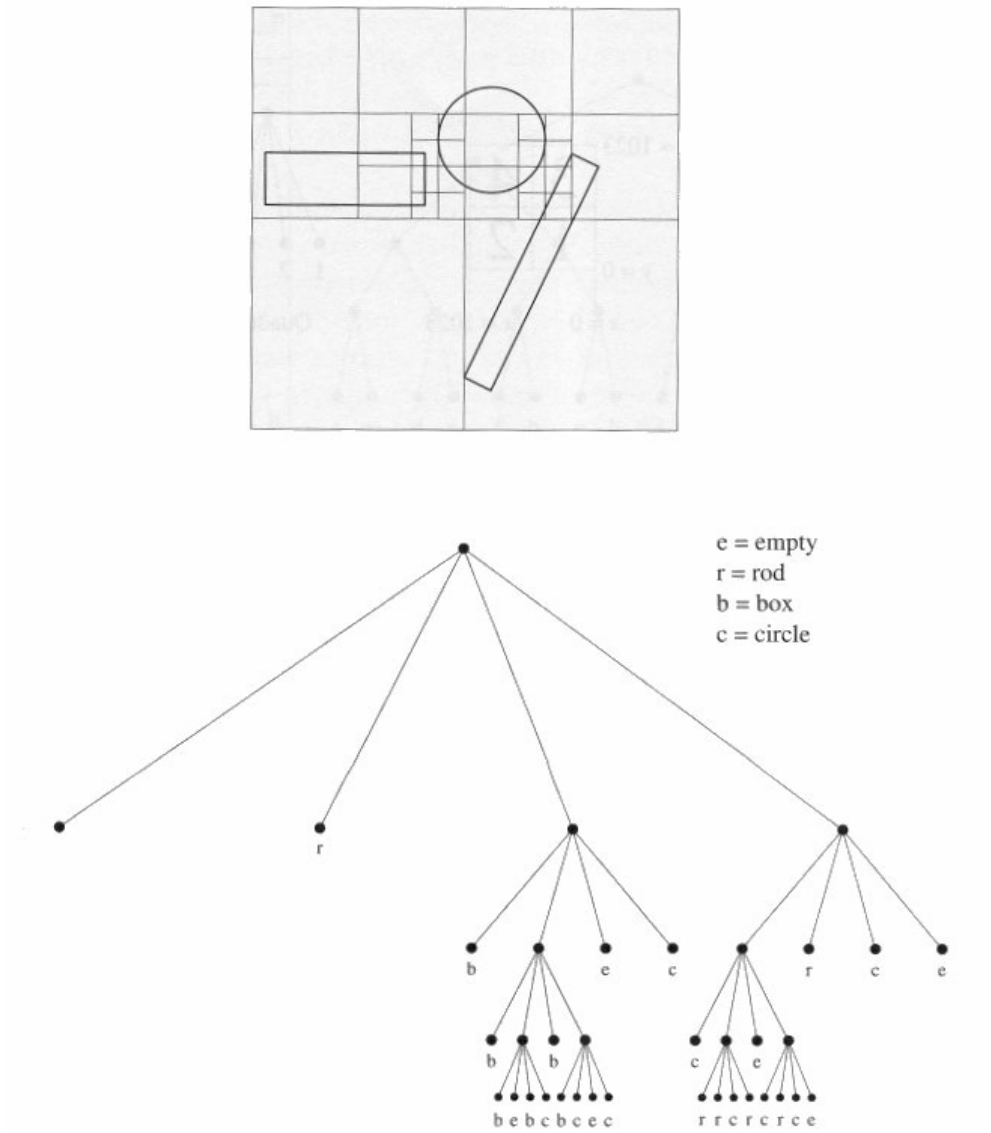


(b)



(a)

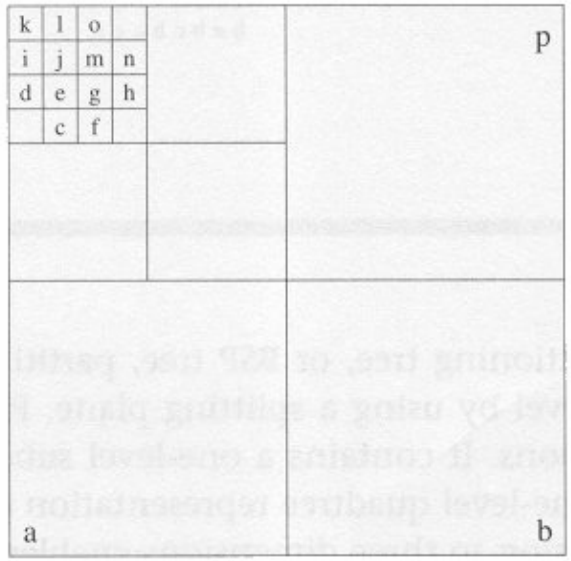
Octrees (quadtrees in 2D) partitioning to levels where a cell contains at most a single object



Partial scene octree in 3D – produced for a CAAD walk-through (same req. as many games) CAAD - Computer Aided Architecture Design

2 Binary Space Partitioning (BSP) trees

2.1 BSP trees – cubic cells



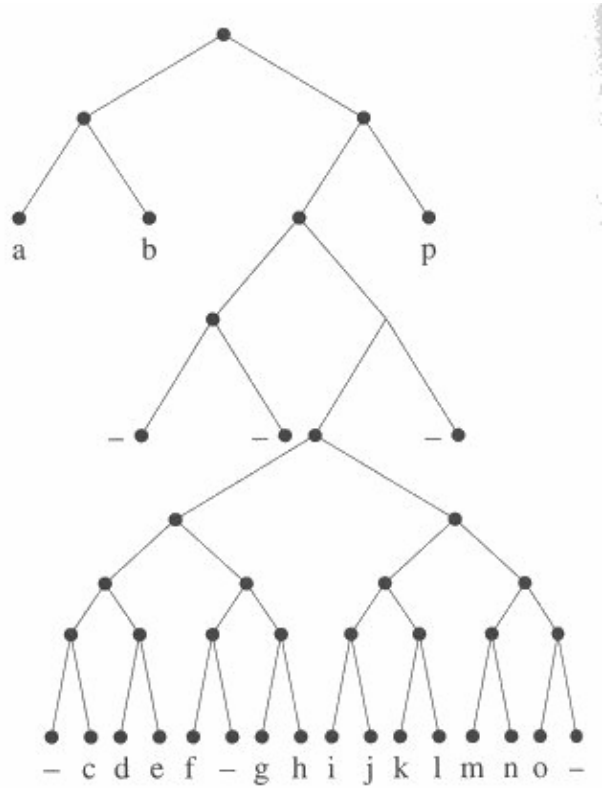
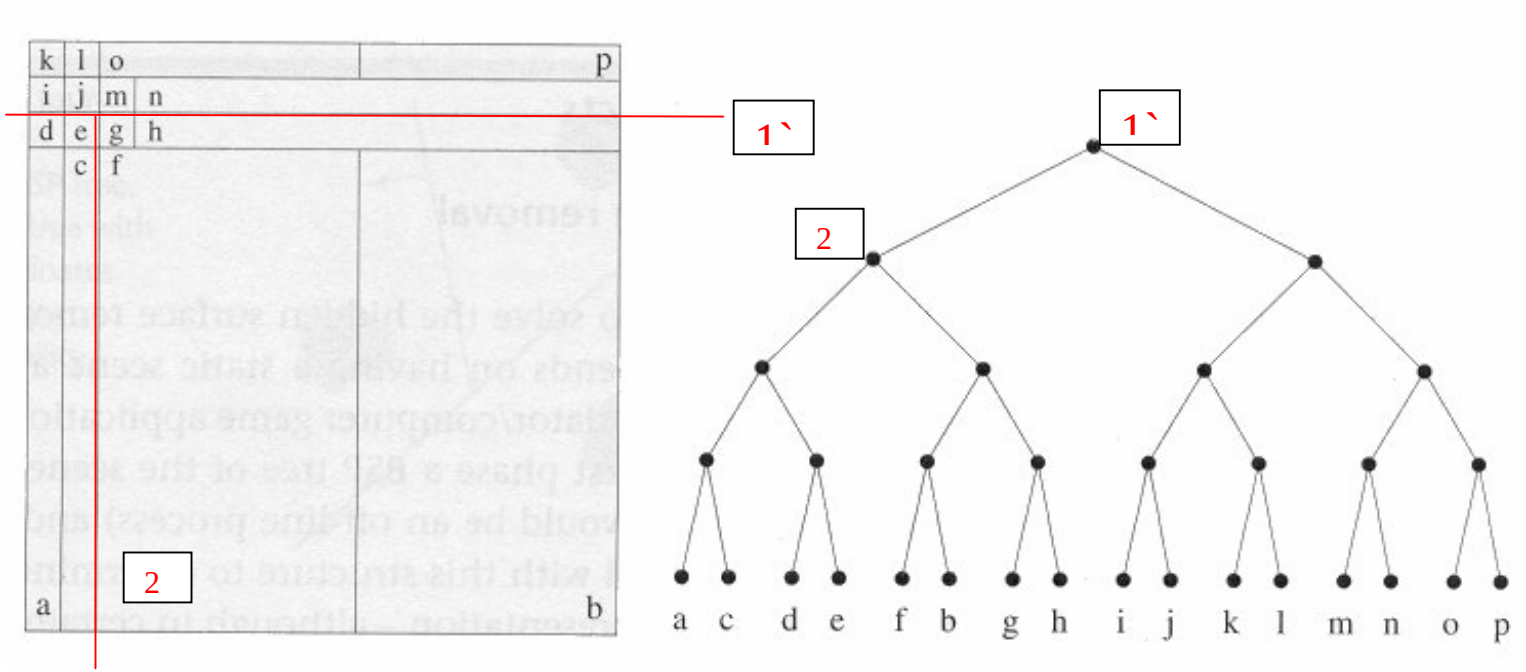


Figure 9.7
Straightforward subdivision of a two-dimensional scene containing 'objects' a–p unevenly distributed throughout the scene. The search path length in the tree for most objects is 8.

unsatisfactory - results in a maximum depth of 8 in this example.

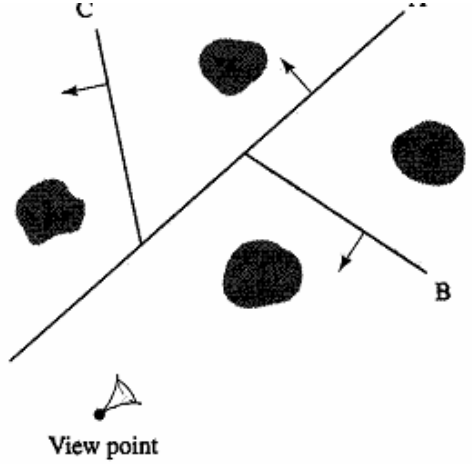
2.2 BSP trees adaptive



C.f. previous example the maximum depth of this tree is 4. At each node a partition line has been chosen that divides the current region into parts with an equal number of objects on either side.

In games we use adaptive BSP trees. In addition separating planes are not constrained to be parallel to the coordinate space axes.

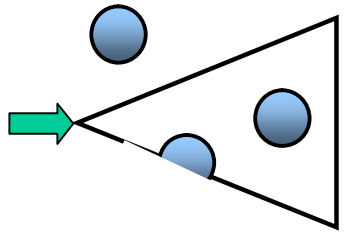
2.3 BSP Trees are just part of a Visibility hierarchy



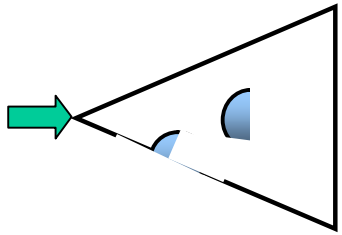
BSP and PVS (Potential Visible Sets)

- Calculated off-line
- Viewpoint is compared with structure to determine potl. visibility in real-time/frame
- Programmed

- VFC + Occlusion culling (View Frustum Culling)
  - Evaluated/frame
  - Programmed

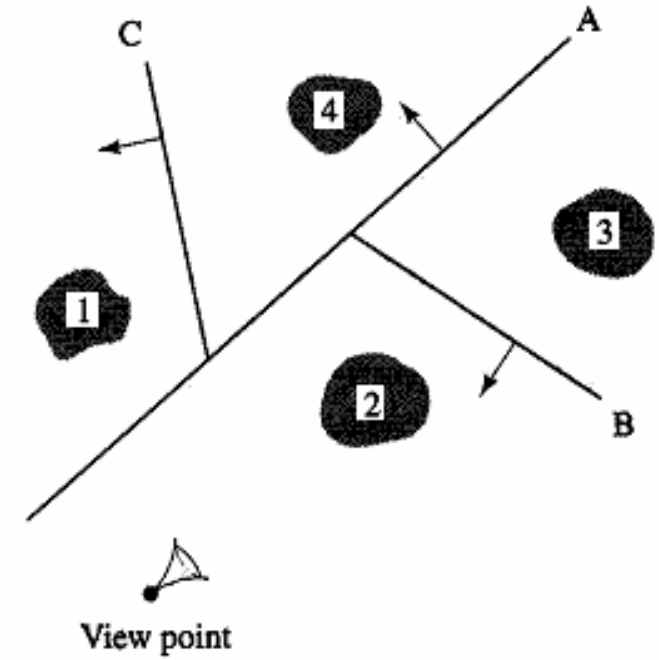


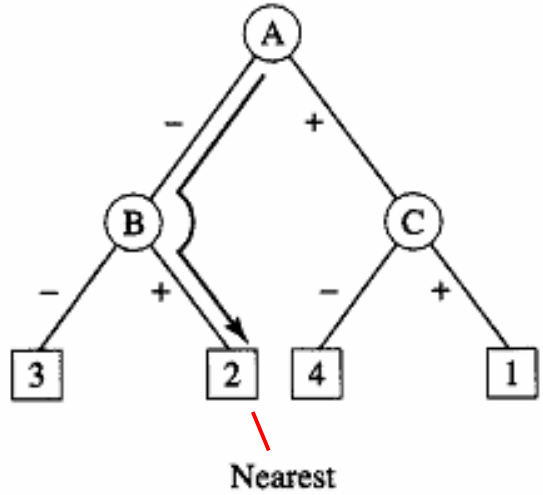
- Z-buffer
  - Evaluated/frame
  - Hardware – programmer can set options

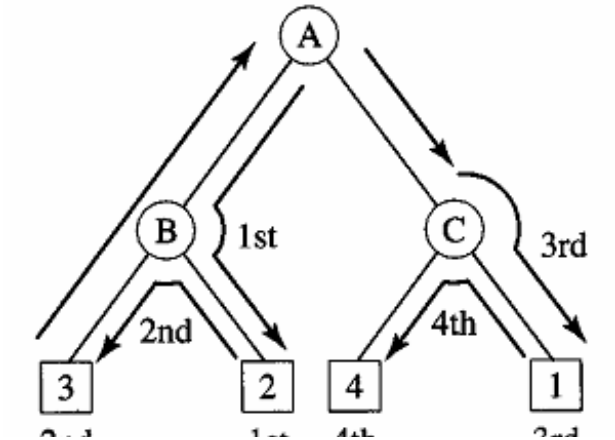


2.4 BSP Visibility ordering

- Descending the BSP tree with viewpoint co-ordinate gives nearest object
- At each node: viewpoint in front or behind of partition plane?
- Descend far side first and output polygons
- Far to near is inefficient
- Near to far with a write mask better
- This is a form of occlusion culling

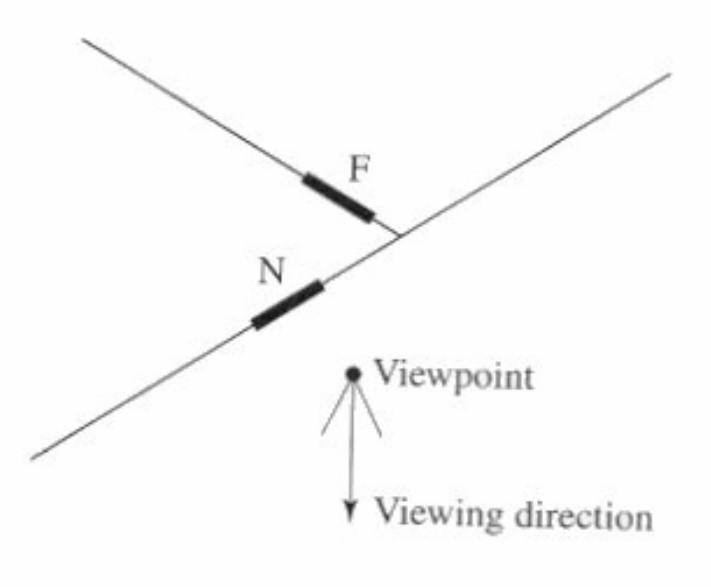




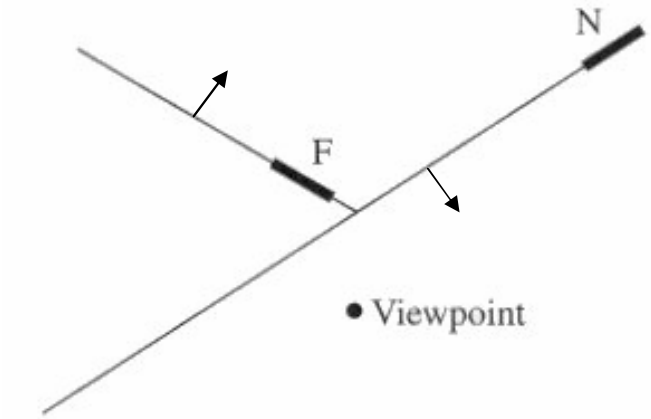


- Visibility ordering with BSPs is *not* directly related to the distance of the polygon from the viewpoint, nor is it related to viewing direction.

Thick line represent polys. Thin line represent partition planes. Planes and polygons are coincident (later).



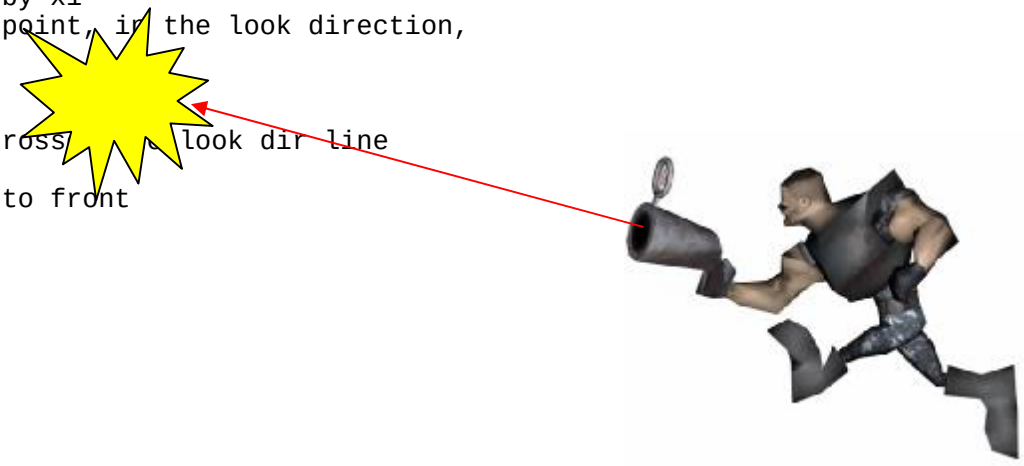
F and N still appear in visibility ordering, but cannot be seen from the viewpoint



Visibility ordering does not depend on distance from viewpoint

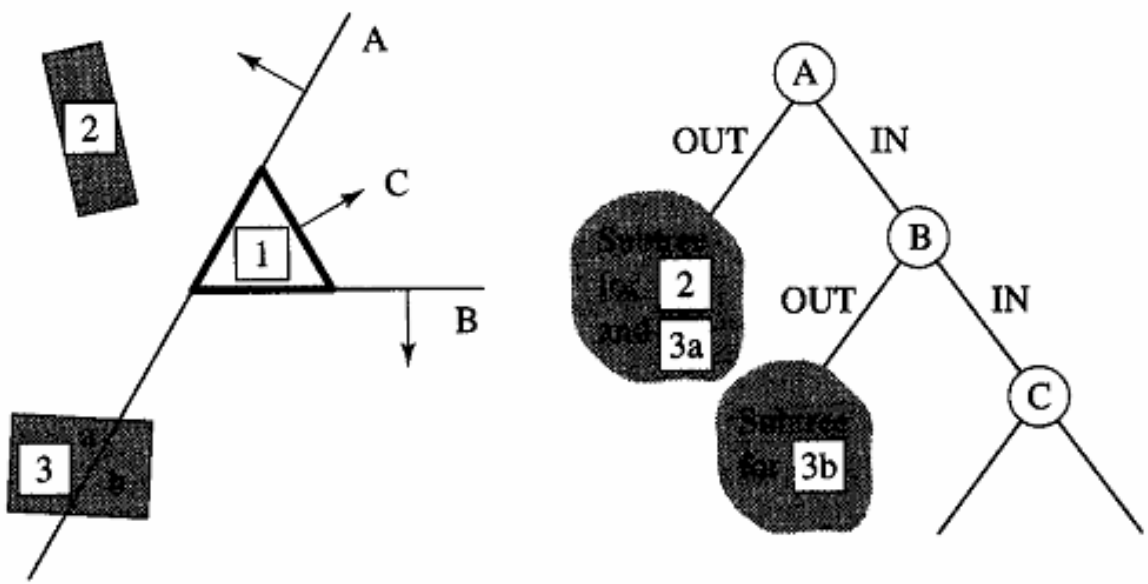
A simple routine that implements a tree traversal for far to near ordering. For near to far ordering the recursive calls to *draw_bsp* are reversed.

```
void draw_bsp(bsp_node *n, camera *c)
{
    // x1 is dot product of plane normal and camera look dir
    x1=VecDot(n->normal,c->Z);

    // x2 is the perpendicular distance from the camera point to the plane
    // it is the smallest distance from the camera point to the plane
    x2=VecDot(n->normal,c->pos)+n->d0;

    // y is the perpendicular distance divided by x1
    // this gives the distance from the camera point, in the look direction,
    // to the plane
    y=x2/x1;

    if(y>0 || fabs(x1)<0.7071)    // if plane cross
     {
      if (x2>0)  // draw the two children back to front
       {
       draw_bsp(n->child[1],c);
       draw_bsp(n->child[0],c);
       }
      else
       {
       draw_bsp(n->child[0],c);
       draw_bsp(n->child[1],c);
       }
     }
  else
    {
     if (x2>0)  // draws only one node
       draw_bsp(n->child[0],c);
     else
       draw_bsp(n->child[1],c);
    }
}
```



This solves the transparent polygon/Z-buffer problem but is *not* the primary use of a BSP database. BSP is used to quickly determine the region containing the viewpoint, to enable VFC and for fast ray intersects

2.5 Building a BSP tree using object polygons (off-line)



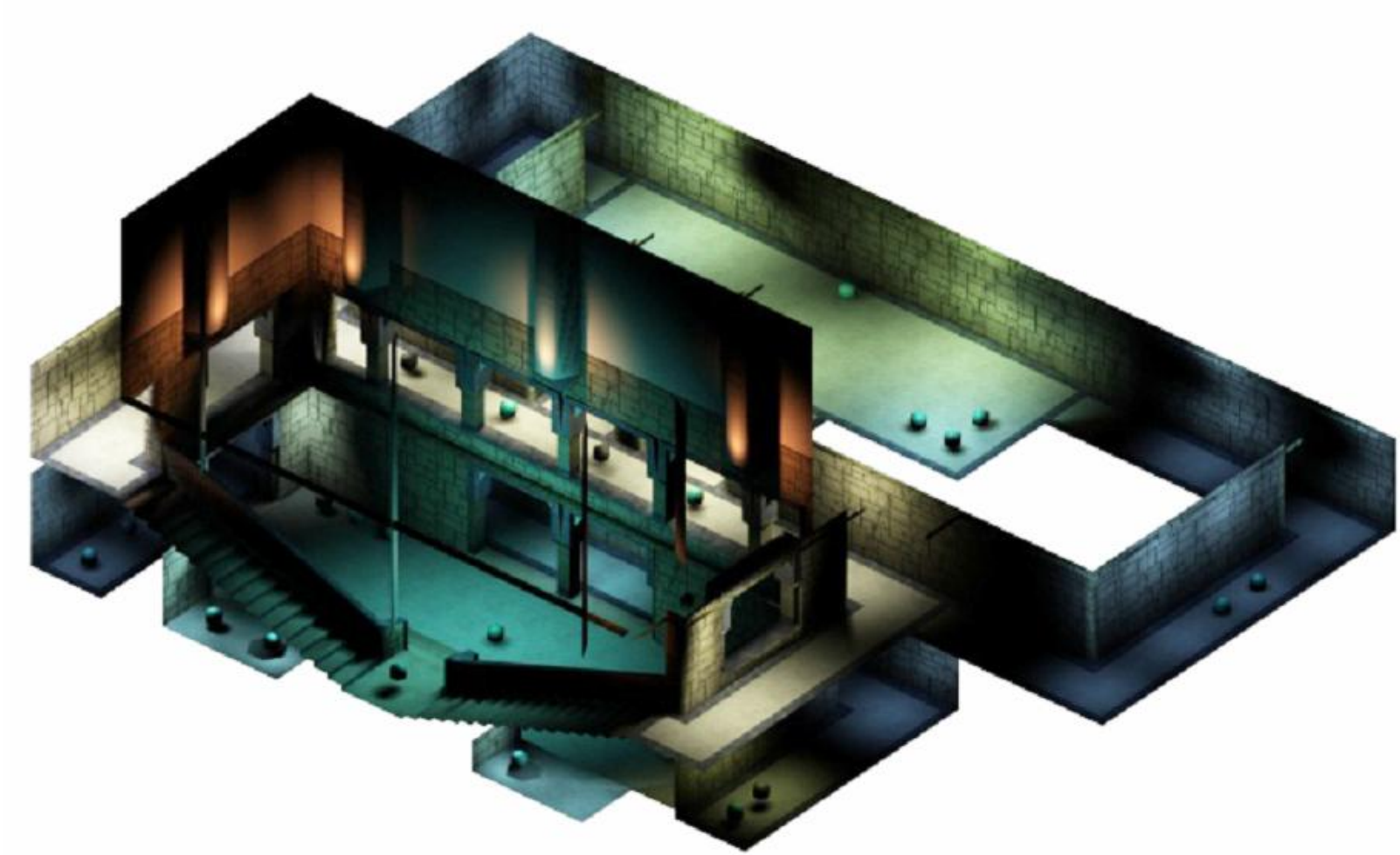
- Use object polygons as partitioning planes:
  - Determine root plane from first polygon. All other polygons lie on one side or the other or are split.
  - Recurse until a maximum depth or threshold number of polygons for a leaf.
- Minimise polygon splitting – look ahead to see number of splits a particular polygon produces. Determine choice accordingly.

2.6 Building BSPs in games and CAAD (Computer Aided Architecture Design)

- A good way to build a BSP tree is to categorize the level/scene geometry to distinguish between **structural** faces and **detail** faces.
- Structural faces are large polygons, like those which represent walls, and which can be used as splitting planes in the BSP tree.
- Such faces should form themselves into concave volumes – the entire level/scene
- Detail faces belong to small objects that are contained by such a volume.
- The overall aim of the strategy is to split a scene into the minimum number of convex volumes.
- Another practical point is that polygon splitting can be ignored and the whole polygon can be sent to both sides of a branch node.

Structural elements in a CAAD application used as BSP splitting planes

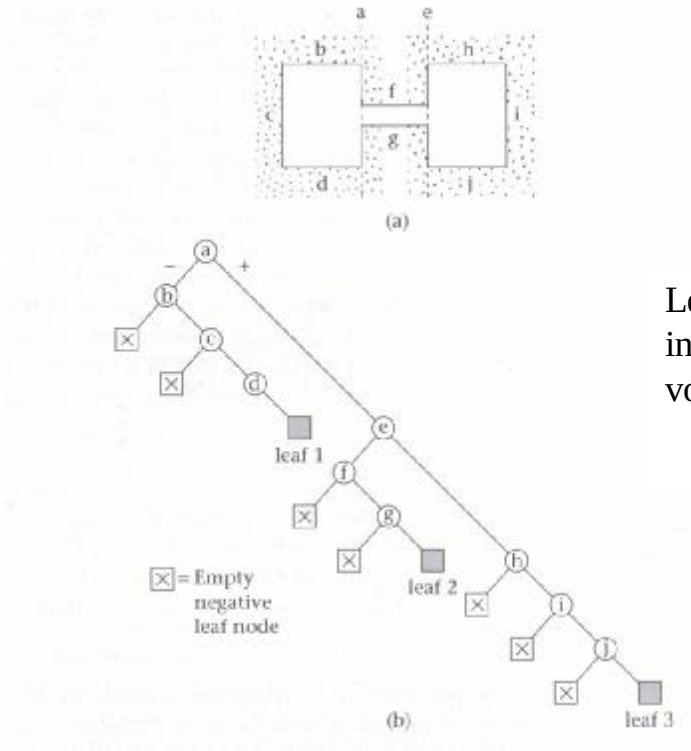
Detail elements in a CAAD application



A view of an entire games level

2.7 BSPs and convex/concave volumes

Construct levels, or convert levels into sets of ***convex*** volumes. Example is the plan of a simple level (one concave volume divided into 3 convex volumes). This consists of 12 faces for the walls plus 3 faces each for the floor and the ceiling. The method then splits this into 3 convex volumes as shown which form the leaf nodes. The structure consists of 18 convex polygons (quads) and 10 planes. Consider the tree which demonstrates the recursive build process for this level considering the walls only. The root node chooses plane *a*. Subsequent child nodes chooses planes *b*, *c* and *d*. Each of these generate children with empty negative nodes and the recursion continues to generate leaf 1. Leaf 1 is then the convex volume to the left of plane *a*. This volume contains all the faces belonging to this convex volume. The process continues in a similar manner to generate two more leaf nodes.

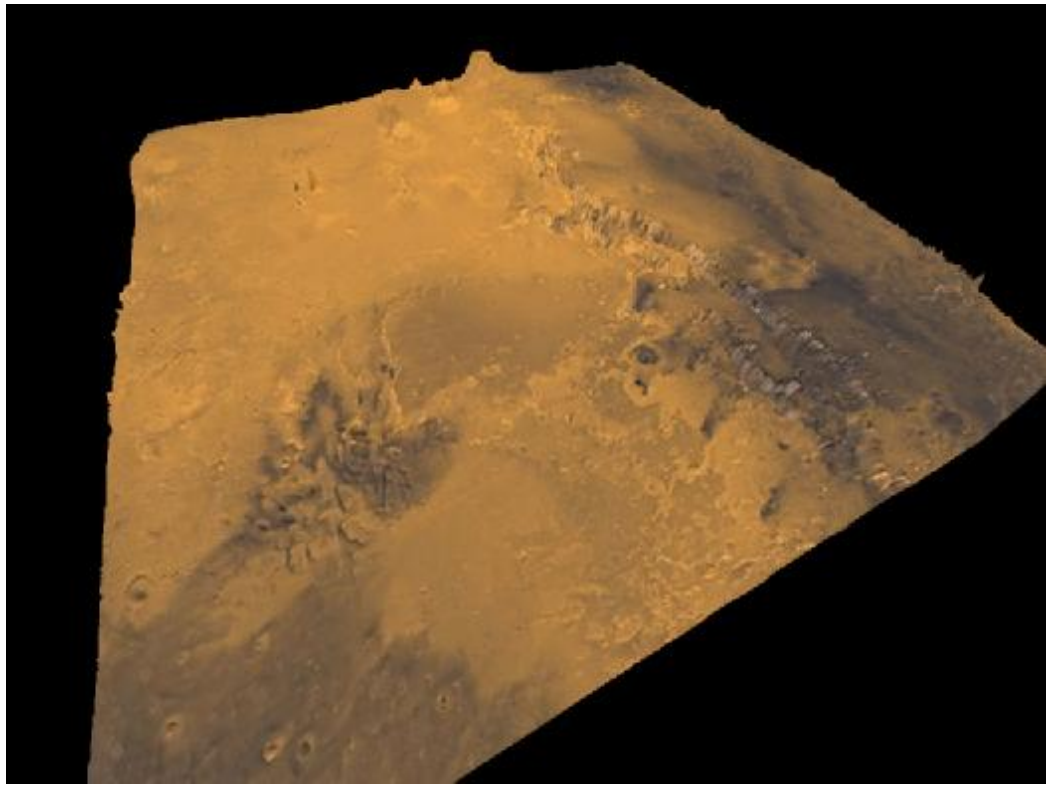


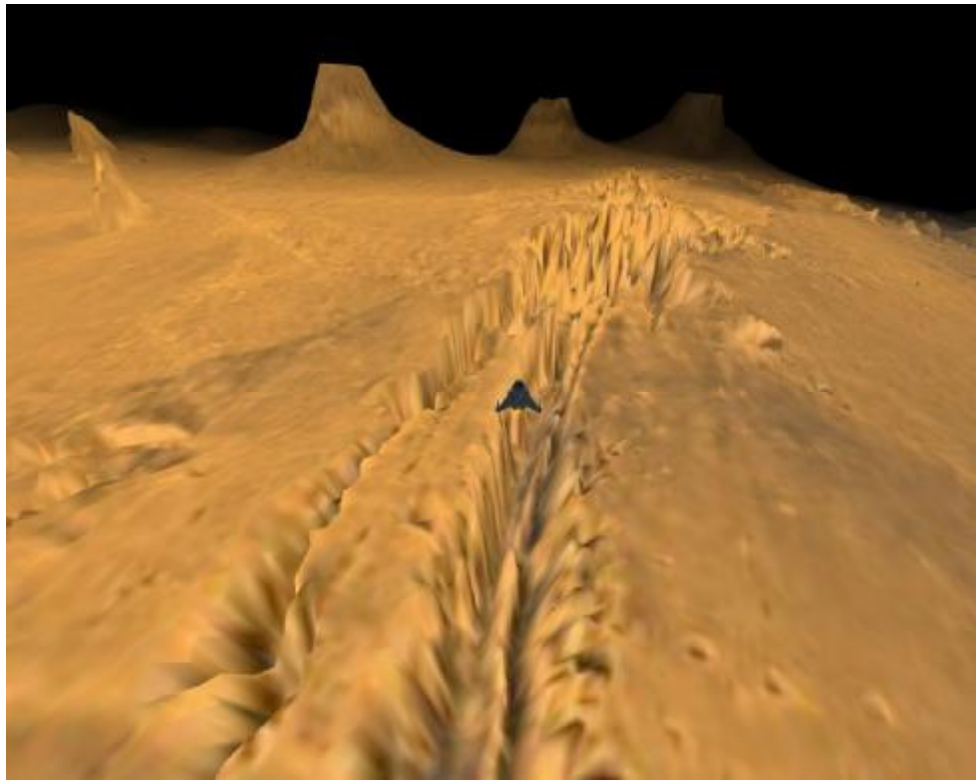
Leaves are the 3 convex volumes in the level. Can determine which volume contains the viewpoint

(note that there are NO negative leaf nodes – this is useful in path planning)

2.8 BSP and landscape

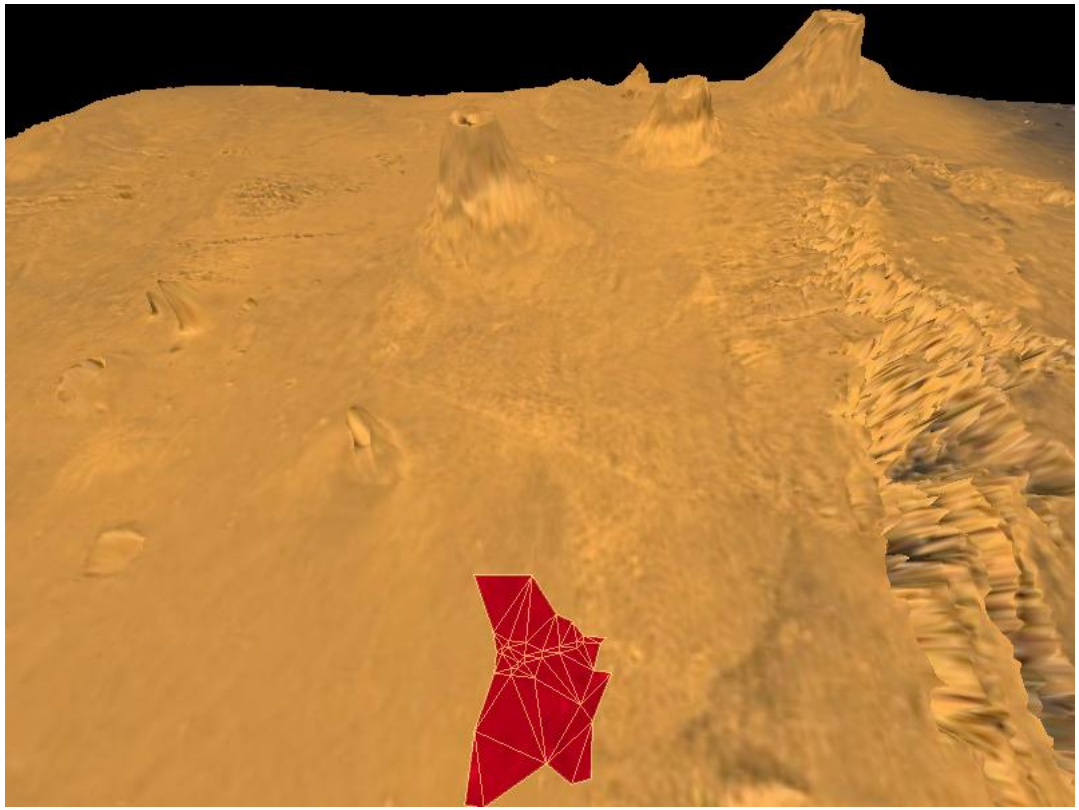
- 133,000 polygons – this is part of the surface of Mars and is rendered from NASA data.





BSPs and landscape

- Landscapes contain no elements that can be used as structural planes
- Define a cellular grid and use Axis Aligned planes as splitters choosing an X plane or a Y plane at each node depending according to occupancy
- Apply BSP splitter that uses landscape triangles within each cell

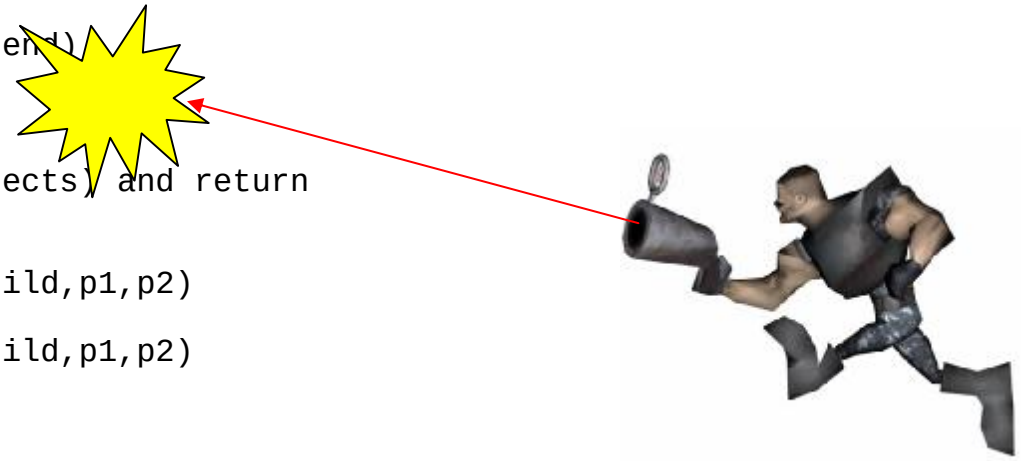


2.9 BSPs and ray intersects

A BSP partitioned scene can be used to quickly find the first object intersected by a ray defined by *start,end.* This test can be used collision detection. Where an object, or light beam, travelling in a straight line from a source hits a target The efficiency of this important utility derives from the BSP partitioning.

A ray intersect test with a BSP tree and has low code complexity as the following example demonstrates.

```
call bsp_intersect(bsp root node,start,end)

bsp_intersect(node,p1,p2)
{
if (node is leaf) rayintersect(leaf objects) and return
classify p1 (+/-)
classify p2 (+/-)
if (p1 and p2 are +) bsp_intersect(+ child,p1,p2)
else
if (p1 and p2 are -) bsp_intersect(- child,p1,p2)
else
{
bsp_intersect(- child,p1,p2);
bsp_intersect(+ child,p1,p2);
}
}
```



This returns all intersects along the ray.

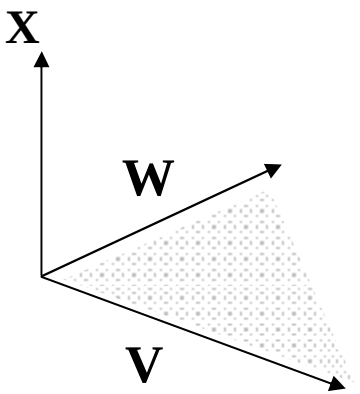
3. BSP maths

3.1 Determining a plane from a polygon

- A plane has the equation:

  $Ax + By + Cz + D = 0$

  where A, B, C are the components of its normal vector (calculated from any three non-collinear vertices).



- The cross product of two vectors **V** and **W** is defined as:

  $\mathbf{X} = \mathbf{V} \times \mathbf{W} = (v_2w_3 - v_3w_2)\mathbf{i} + (v_3w_1 - v_1w_3)\mathbf{j} + (v_1w_2 - v_2w_1)\mathbf{k}$

  where i, j and k are the standard unit vectors. A, B and C are thus:

  $A = v_2w_3 - v_3w_2$
  $B = v_3w_1 - v_1w_3$
  $C = v_1w_2 - v_2w_1$

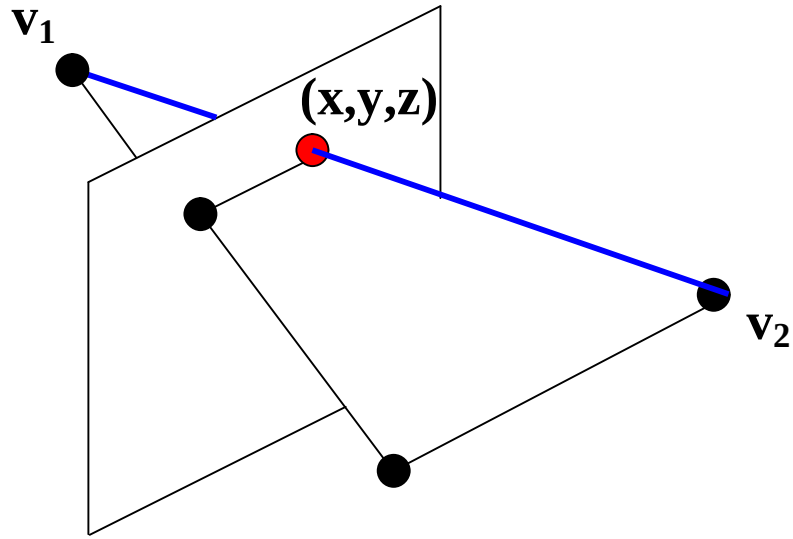
- D is obtained by substituting a point known to lie on the plane (in other words a vertex) into the plane equation.

3.2 Splitting a polygon

- A polygon can be classified by testing its vertices with respect to the split plane by substituting vertices into the above equation.
- For a single point, the result will be:
  - positive if on side consistent with direction of surface normal,
  - negative if on other side,
  - or zero if lies in the plane.
- Find vertex pairs intersected by the plane.
- Find points of intersection and generate two polygons. For the *blue* edge:

$$t = \frac{Ax_1 + By_1 + Cz_1 + D}{A(x_2 - x_1) + B(y_2 - y_1) + C(z_2 - z_1)}$$

$$x = x_1 + (x_2 - x_1)t$$
$$y = y_1 + (y_2 - y_1)t$$
$$z = z_1 + (z_2 - z_1)t$$



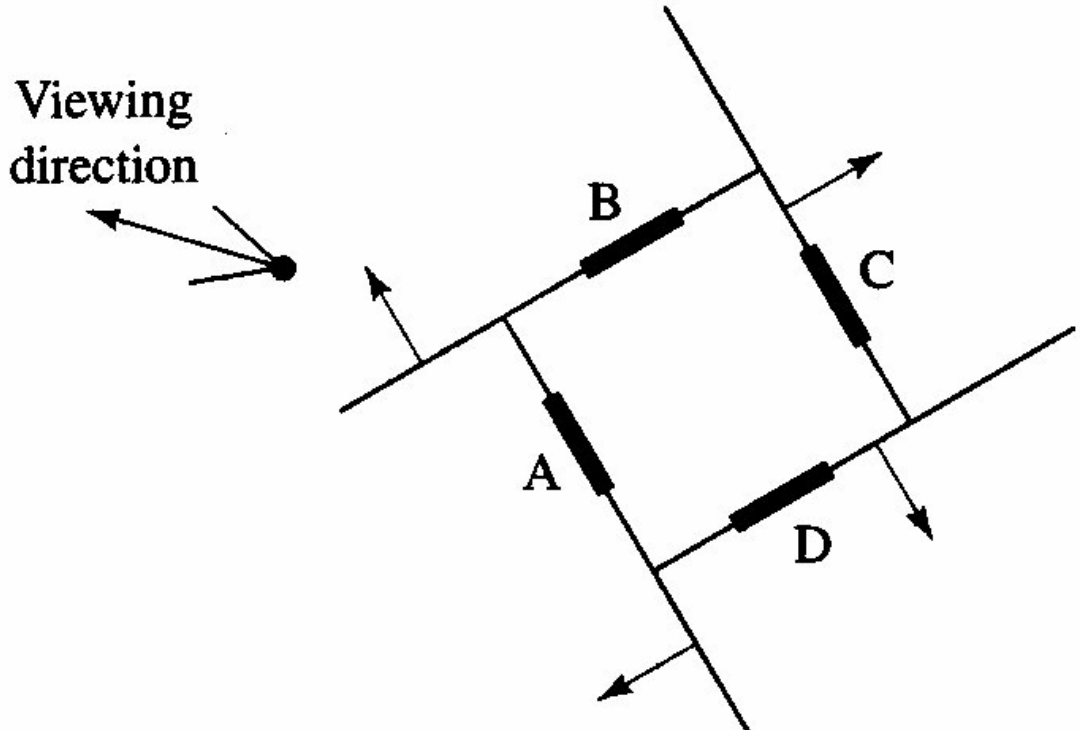
where (x_1,y_1,z_1) and (x_2,y_2,z_2) are the two vertices of the edge that crosses the plane with coefficients A,B,C,D at intersection point (x,y,z).

4. BSPs and Culling

Culling means removing entities that cannot be seen as early as possible. Rendering order (Painter's algorithm) is not an important aspect of BSPs. and is not generally used. The real benefits of BSPs are efficiently determining the region containing the viewpoint and culling .

4.1 Back face culling

- No-cost operation carried out during tree traversal to find a visibility ordering
- If the viewpoint is in the –ve (IN) half space of the plane of a polygon, that polygon will be (automatically) culled as part of the tree traversal.
- This operation is independent of the direction of the viewer.
- Culling with a BSP tree does not eliminate all the polygons that would be culled by a test involving a line of sight vector (conventional back face culling)



Only C and D are culled

BSP and culling

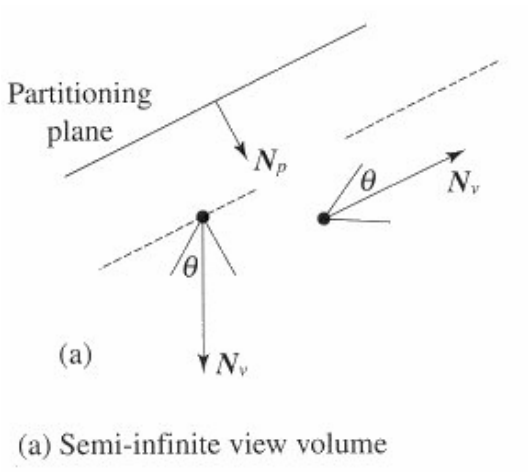
4.2 View frustum culling (VFC)

- At a node the view frustum is compared with the partitioning plane and if the appropriate culling condition is fulfilled the entire subtree associated with that node can be eliminated.
- (a) Semi-infinite view volume – compare view vector and θ with normal of partition plane

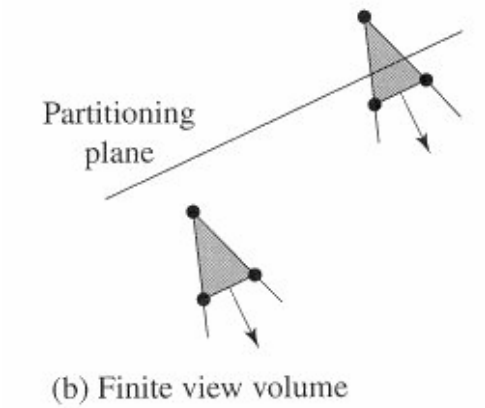
  Culling condition for positive side: $\mathbf{N_v.N_p} \geq \cos^{-1}(\pi/2 - \theta)$
- (b) Finite view volume

  Check (5) vertices of view frustum against partition plane.

  If entire frustum is on one side or the other – cull at current node



(a) Semi-infinite view volume



(b) Finite view volume

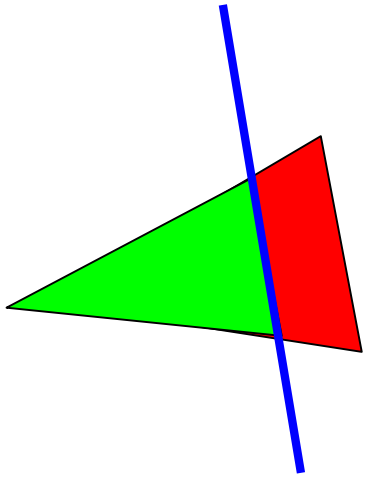
With finite view frustum far plane can become visible:

Finite view volume

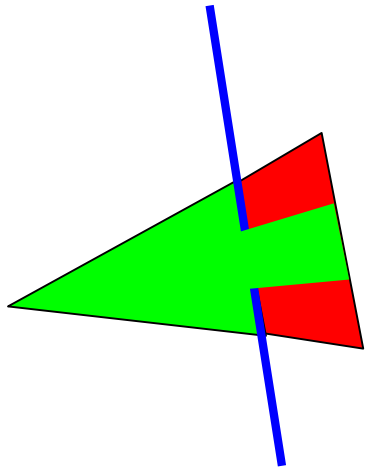
Finite view volume + fog

5 VFC problems and Potentially Visible Sets (PVS)

Culling using a BSP tree is still too inefficient – must improve



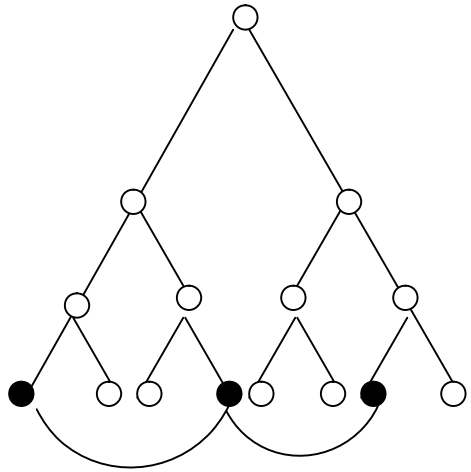
View frustum straddles a splitting plane which is a wall or floor – entities which should be invisible (red) are not culled.



View frustum straddles a wall containing a portal – must render information visible through portal

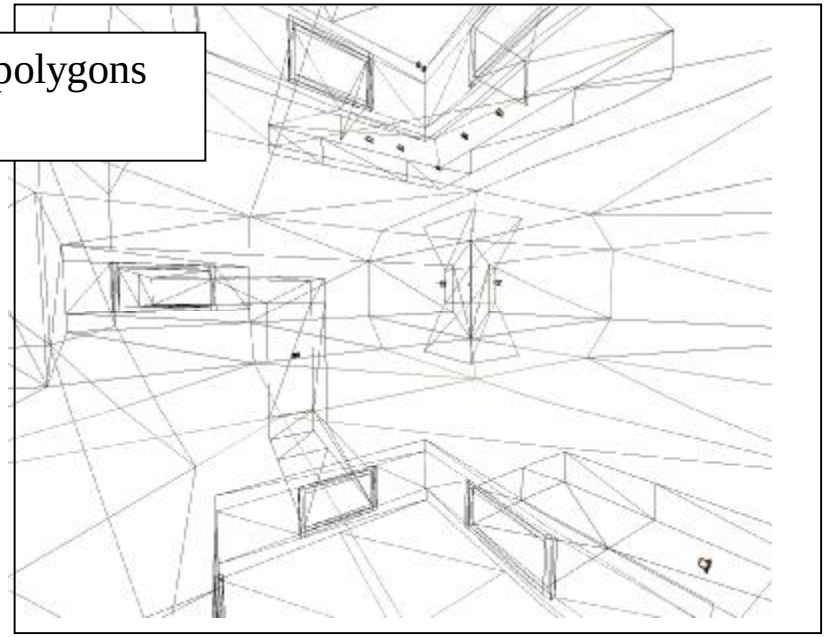
Both these problems can be solved by using PVS

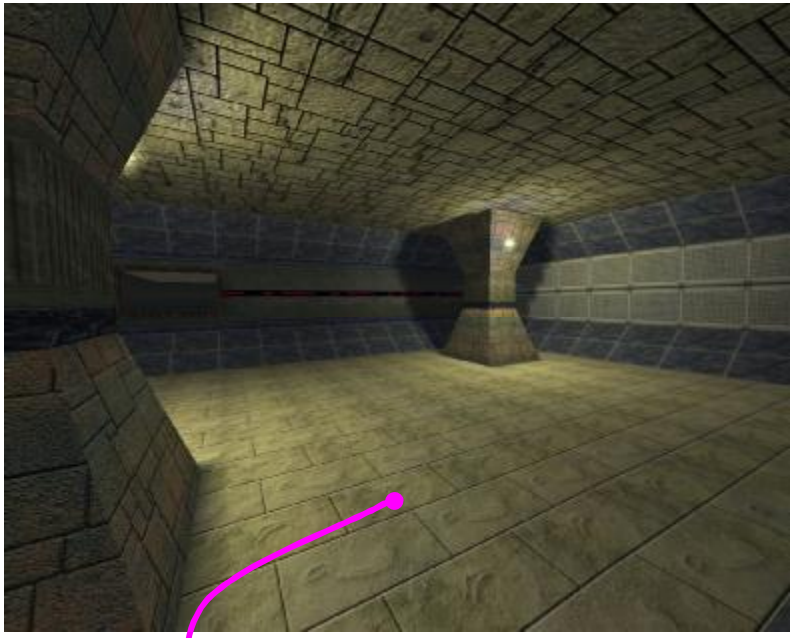
- Potentially Visible Set = the sub-set of all polygons that **may** be visible
- Render:
  - descend tree to leaf with viewpoint co-ordinates
  - pass contents of leaf for rendering
  - pass contents of **connected** leaf nodes for rendering
  - **BSP tree with connected leaf nodes is a PVS**



- Connected leaf nodes:
  - $Node_i$ is connected to $node_j$ iff $node_i$ can be seen from $node_j$

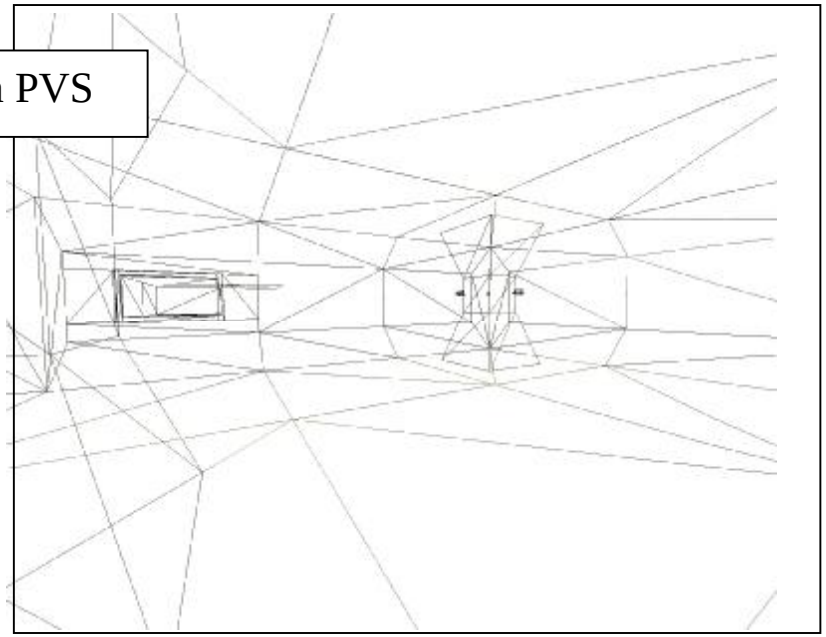
Without PVS – polygons rendered





Floor has been used as a splitter finite view frustum straddles floor implying many invisible polygons rendered

With PVS



Constructing PVS (non-deterministic)

1. Build BSP
2. Generate a random viewpoint
3. Determine leaf node for this viewpoint
4. ‘Render’ scene from that viewpoint in 6 mutually orthogonal directions using BSP for VFC
5. Find leaf nodes containing visible polygons and connect

---

6. Rendering in real-time – when a leaf node is reached its contents are only passed for rendering if it is connected to the node that contains the viewpoint.

- Watt (2001) claims that:

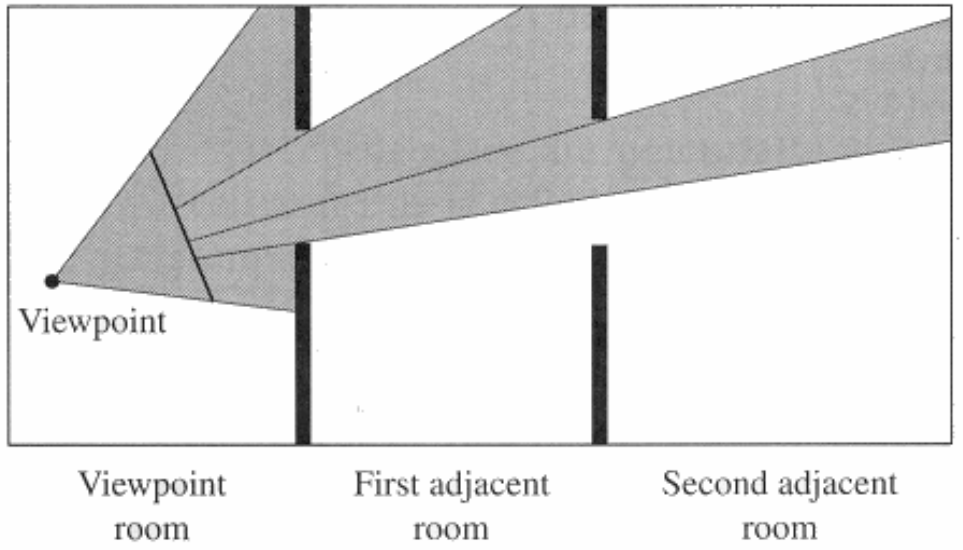
  “For a simple games environment of 9900 triangles and a BSP tree of 1500 leaves, 50000 sample viewpoints resulted in no errors in a substantial test run. The procedure reduced the number of leaf nodes entered in the BSP tree to around 5% of the number reached without this enhancement.”

6 BSPs –processing dynamic lights

Moving lights modulate static lighting. White light is static; red and yellow lights are moving. Consider an event such as an explosion at a point whose influence we decide to limit to a sphere/box centred on the reference point or the event centre. For example, we may only want to 'paint light' on polygons that are near to the explosion. This is easily implemented by descending the tree from a node only if the distance from the event centre to the partitioning plane is less than the radius of the sphere of influence (perhaps animated – increasing per frame - in the case of an explosion).

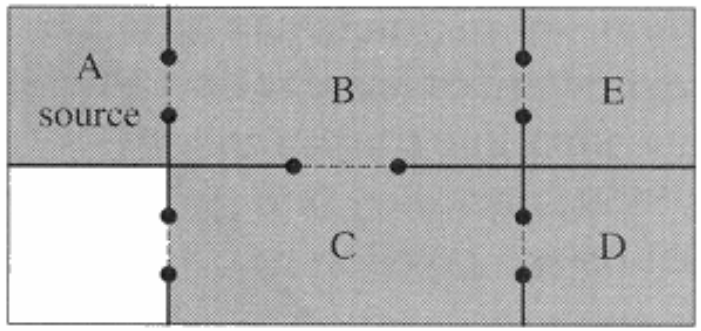
7 Cell/Cell visibility for ‘portal' environments (alternative to BSP+PVS)

- In its simplest form this does not resolve visibility exactly but is a method for culling (and clipping) objects that cannot possibly be seen from the room that contains the viewpoint.
- The model database is divided into rooms or cells.
- For the viewpoint room objects can be culled against the view volume as normal.
- If the primary view volume intersects any portal then a view volume for the adjacent room can be constructed from the viewpoint and the corner points of the portal.
- The process can be recursive if the adjacent room itself contains portals, intersected by this volume, into a third room and so on.
- Mirrors are simply tagged as portals except that we recurse into the same room and reflect the room contents (and the view frustum) about the plane of the mirror.



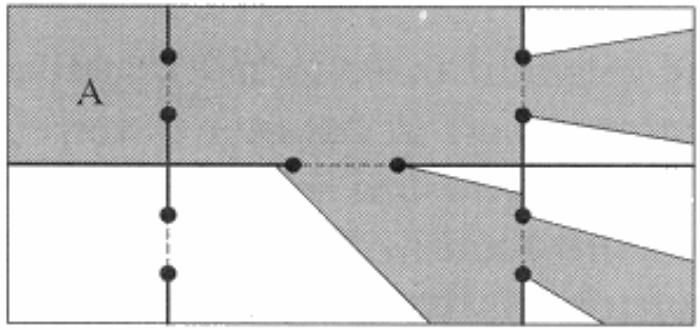
6.1 Generalisation of portals

- Main work is due to Teller and Sequin (1991) and Funkhouser (1996).
- a) to c) are pre-computed and d) is computed dynamically.
- (a) is an example of **cell to cell visibility**. This is the set of cells connected by portals to the source cell V which contains the viewpoint. It is the set of potentially visible cells.



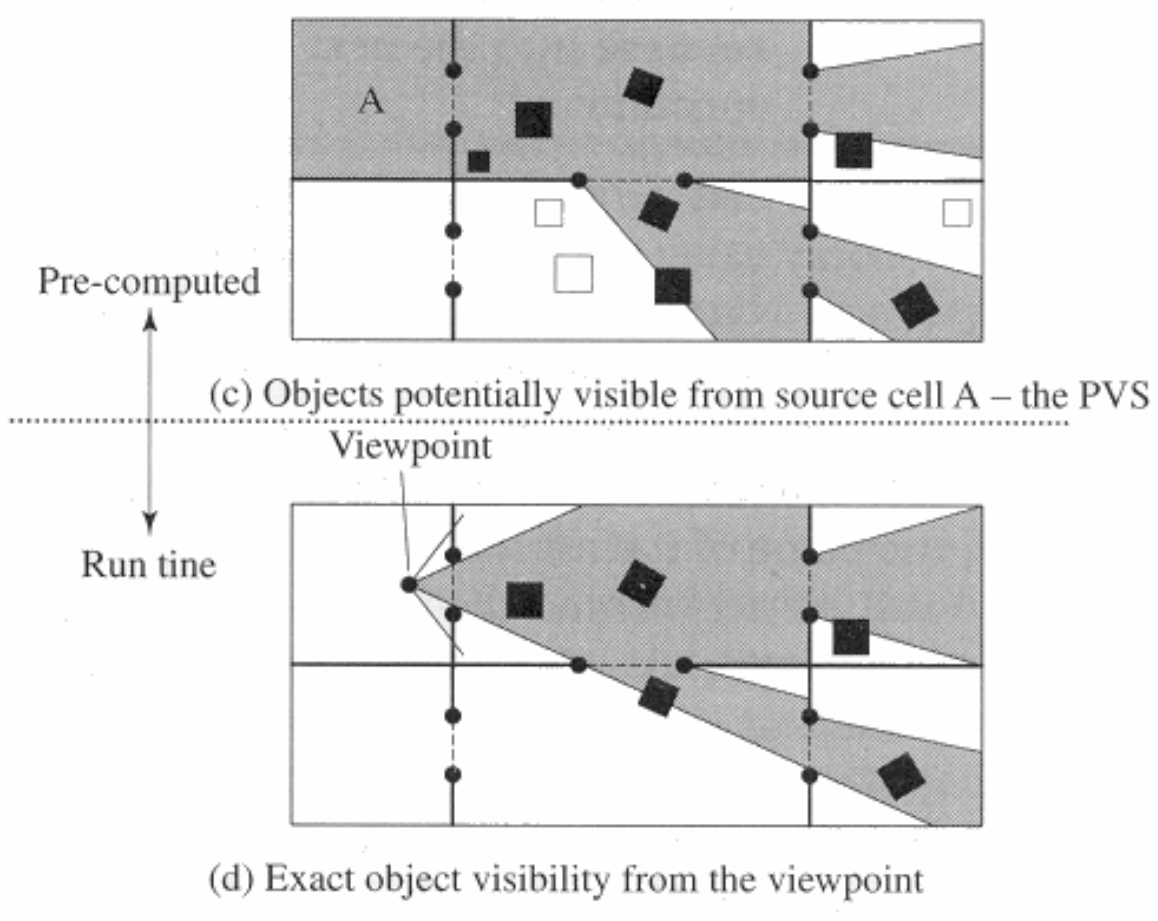
(a) Cells visible from source cell A – cell to cell visibility

- (b) is the space visible to a viewer, positioned anywhere in A, through the portals. This is called **cell to region visibility**. In general these are a set of wedges that emanate through each portal in the source cell which narrow as more and more portals are traversed.



(b) The region potentially visible to a viewer positioned in A – cell region visibility

- (c) all objects that intersect this space are potentially visible from anywhere within the source cell and this is called **cell to object visibility**. This is the extent of the pre-computation phase.
- (d) at run-time a viewpoint and frustum is determined and this defines the subset of c) that intersects with the view frustum. This then gives the **eye to object visibility** - those objects which can wholly or partially be see through the source cell's portal.

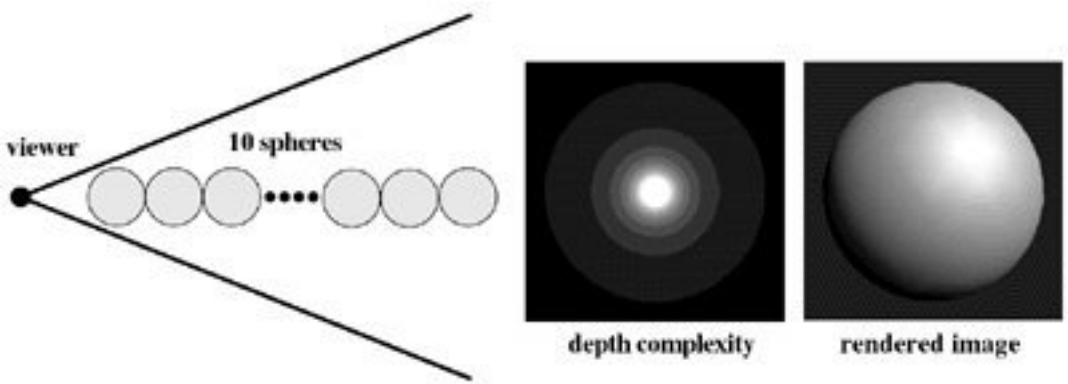


Portals v. BSP + PVS

BSP option is more efficient than portal based schemes as complexity increases. Also as we have seen BSP portioning offers a range of easily implementable applications. The build process for BSP+PVS is completely automatic. However, portal based schemes can incorporate dynamically changing scenes by causing the real time part of the rendering process to follow (b) in the hierarchy in the previous section.

Occlusion Culling (aka Z-occlusion)

VFC is an insufficient culling operation for scenes of high depth complexity. Must cull within frustum:



High depth complexity implies many objects behind one pixel (aka the overdraw burden). Modern hardware can execute occlusion queries and effectively cull occluded objects in the view frustum. This is also called Z-occlusion because it uses Z testing. Similar to early-Z rejection (depth test without render) but requires more programmer intervention.

(Like the Z-buffer which eliminated ‘geometric’ HSR algorithms, then so has Z-occlusion wiped out the need for complicated occlusion algorithms).

Principle - use bounding boxes – if bounding box is occluded then object is occluded. Uses occlusion query extension (NV_occlusion_query). We substitute the complex geometry with a BB ‘rendering’ it for depth testing only. Occlusion query returns how many pixels (from the BB) would end up on the screen. This enables a programmer to set a threshold – if an object is *almost totally* occluded then do not render. Best results if objects are rendered from front to back.

Hardware occlusion culling algorithm

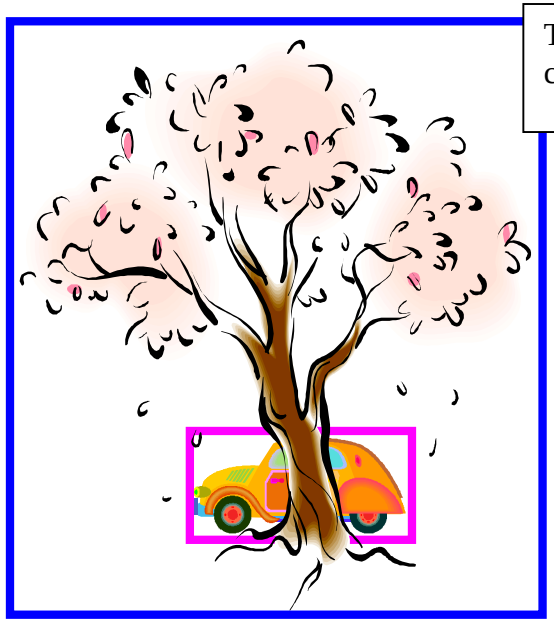
1. Create an occlusion query
2. Disable rendering to screen
3. Disable writing to Z-buffer
4. Issue occlusion query (resets visible pixel counter)
5. Render object’s BB (only testing depth)
6. End query
7. Enable rendering to screen
8. Enable depth writing
9. Get query result (no. of visible pixels
10. if result (visible pixels) > 0 *(or some threshold)* render object

Problems with Z occlusion culling

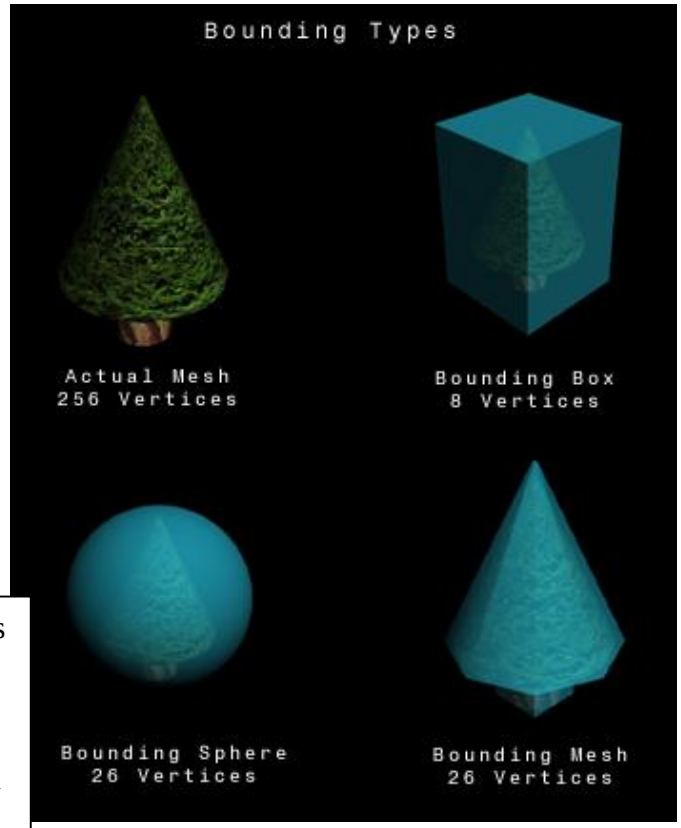
1) BBs cannot write to the Z-buffer.
So there is an obvious problem because of the BB 'fit'
to which there are 2 solutions:

a)Use tight bounding box. Bounding boxes are computed off-line and so there is no cost penalty. Best is bounding mesh.

b)Render very large objects before applying occlusion queries



Tree BB occludes the car BB erroneously



Bounding mesh – is a tight bounding volume – can generate using a tool to produce a low res. mesh then inflate.

Problems with occlusion culling

2) Even although BB is rendered only to Z-buffer there is still a
Fill rate (per pixel cost) penalty. Also for a ‘non tight’ BB many
more pixels are rendered than would be the case for rendering the object.
This problem is bad for a large complex objects
that use simple fragment shaders. Because may spend more time in
occlusion testing than rendering the object with early-Z rejection.