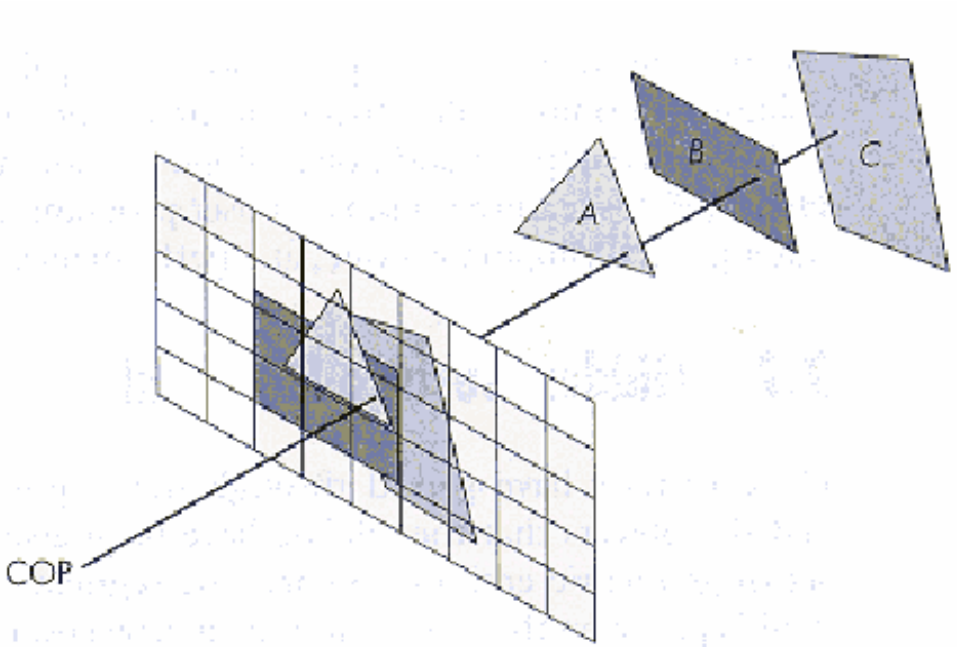
Zakrite ploskve



Problem outline

- Given a set of 3D objects and a viewing specification, we wish to determine which lines or surfaces are visible, so that we do not needlessly calculate and draw surfces, which will not ultimately be seen by the viewer, or which might confuse the viewer.

Approaches

- There are 2 fundamental approaches to the problem.
  - Object space
  - Image space

Object space

- Object space algorithms do their work on the objects themselves before they are converted to pixels in the frame buffer. The resolution of the display device is irrelevant here as this calculation is done at the mathematical level of the objects

- Pseudo code…
  - `for each object a in the scene`
    - `determine which parts of object a are visible`
    - `draw these parts in the appropriate colour`

- Involves comparing the polygons in object a to other polygons in a and to polygons in every other object in the scene.

Image space

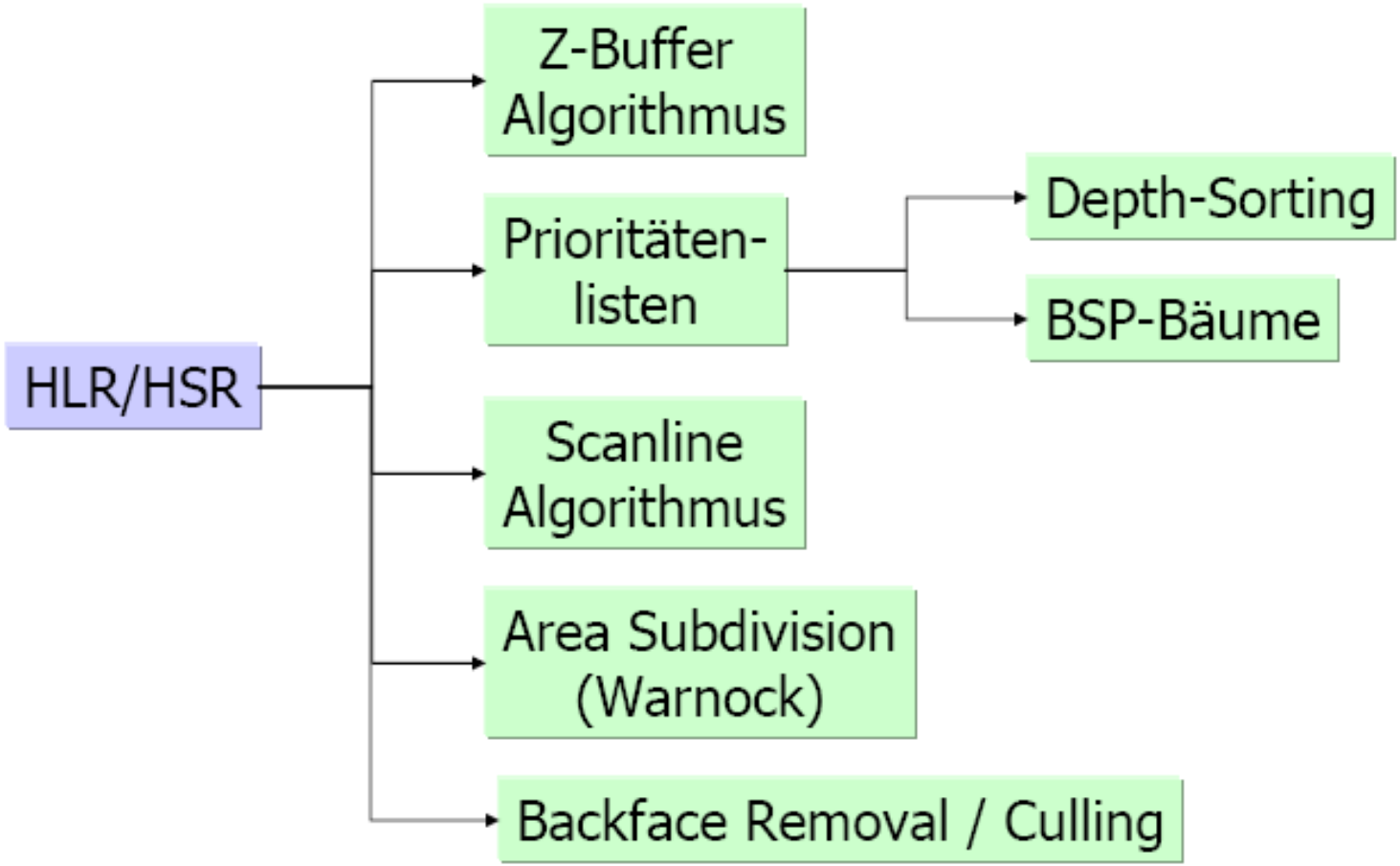
- Image space algorithms do their work as the objects are being converted to pixels in the frame buffer. The resolution of the display device is important here as this is done on a pixel by pixel basis.

- Pseudo code…
  - `for each pixel in the frame buffer`
    - `determine which polygon is closest to the viewer at that pixel location`
    - `colour the pixel with the colour of that polygon at that location`

Algoritmi določanja vidnosti

- Object space
  - Back-face culling
- Image space
  - z-buffer algorithm
- Hybrid
  - Depth-sort

Postopki ugotavljanja vidnosti



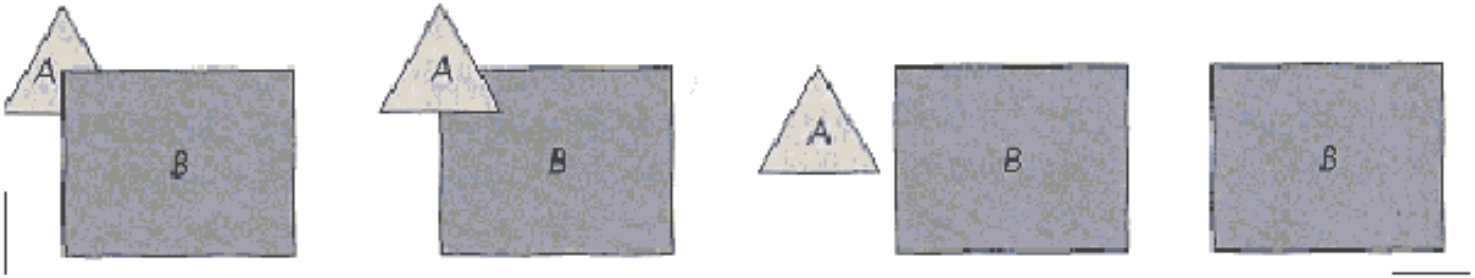
- Generic object precision algorithm

for (each object in the world) {

determine those parts of the object whose view is unobstructed by other parts of of it or any other object

draw those parts in the appropriate color

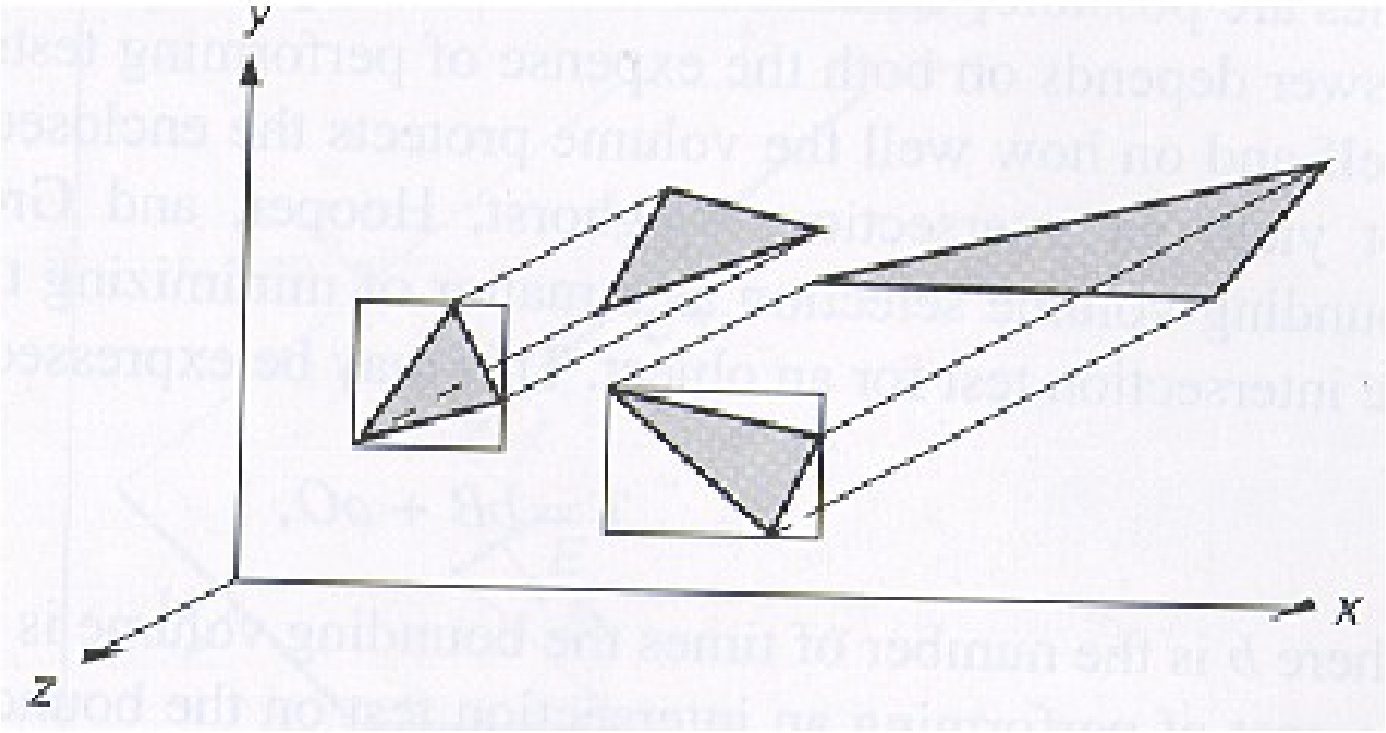
}



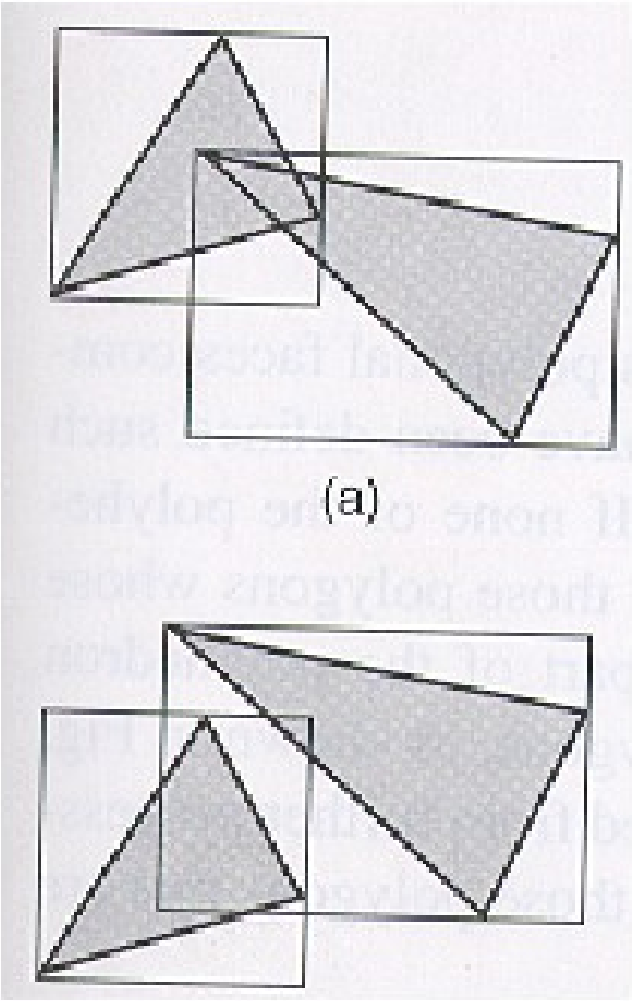
- Cost: $(n-1)+(n-2)+\ldots+1=n^2$, (n: number of objects)

- Extents

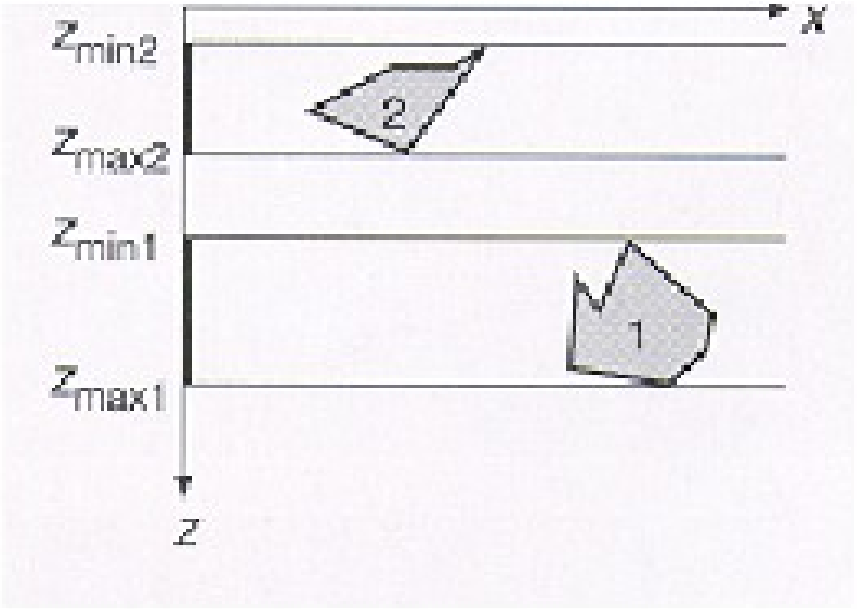
* If the extents of the projections do not overalp, no comparisons need to be made



* If the extents overlap, more testing is needed



- Minmax testing: bounding each dimension seperately (*minmax testing*)

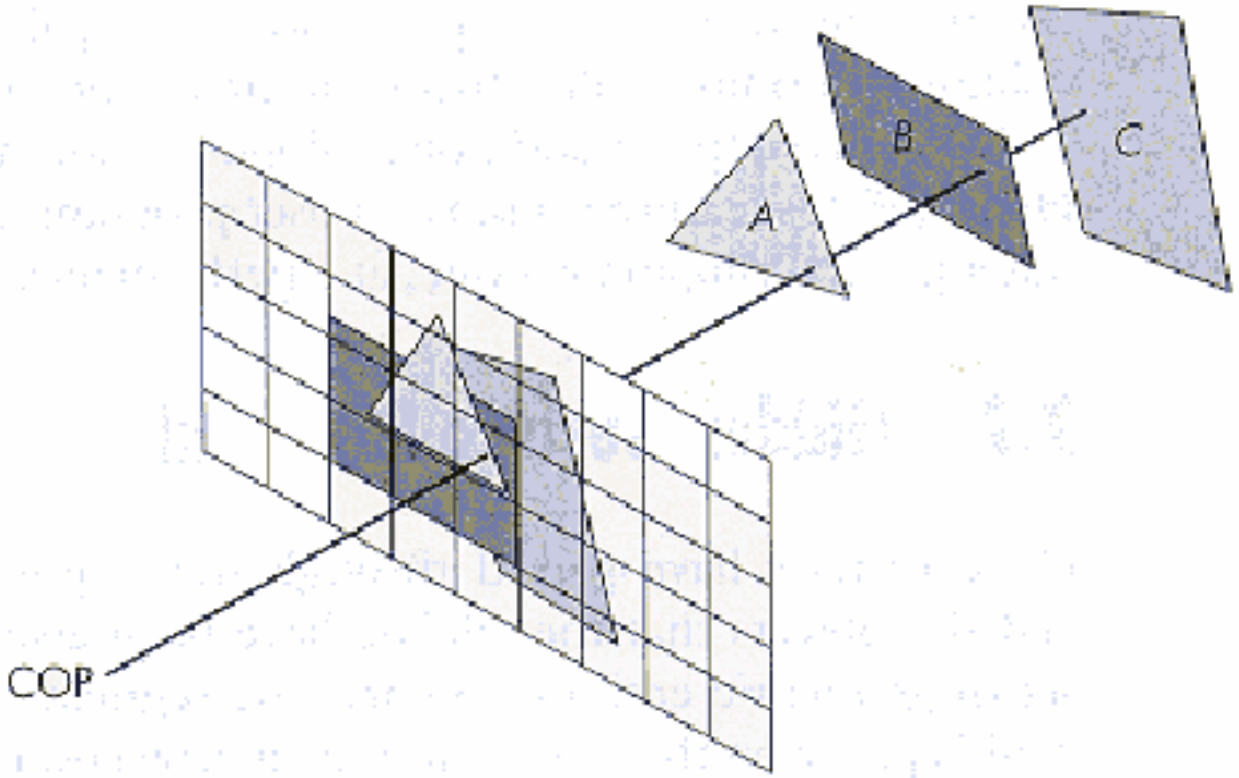


* No overlap if: $z_{max2} < z_{min1}$ or $z_{max1} < z_{min2}$

Metode v prostoru slike (image precision methods)

- Use projected images of objects and are performed at the resolution of the display device

- Idea: visibility is determined pixel by pixel on the projection plane



- Generic image precision algorithm

```
for (each pixel in the image) {

  determine the object closest to the viewer that is pierced by
  the projector through the pixel

  draw the pixel in the appropriate color
}
```

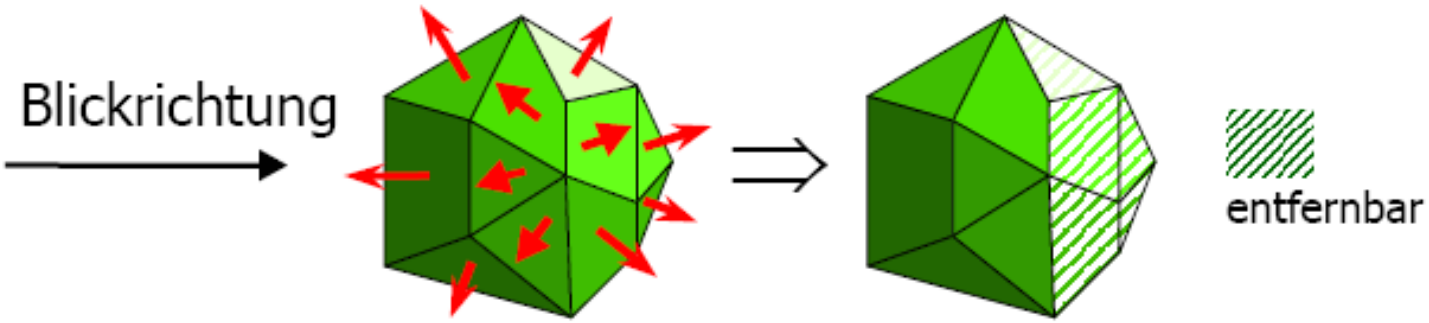
- Cost: np, (n: number of objects, p: number of pixels, $p >> n$), worst-case

Comparison between object and image precision algorithms

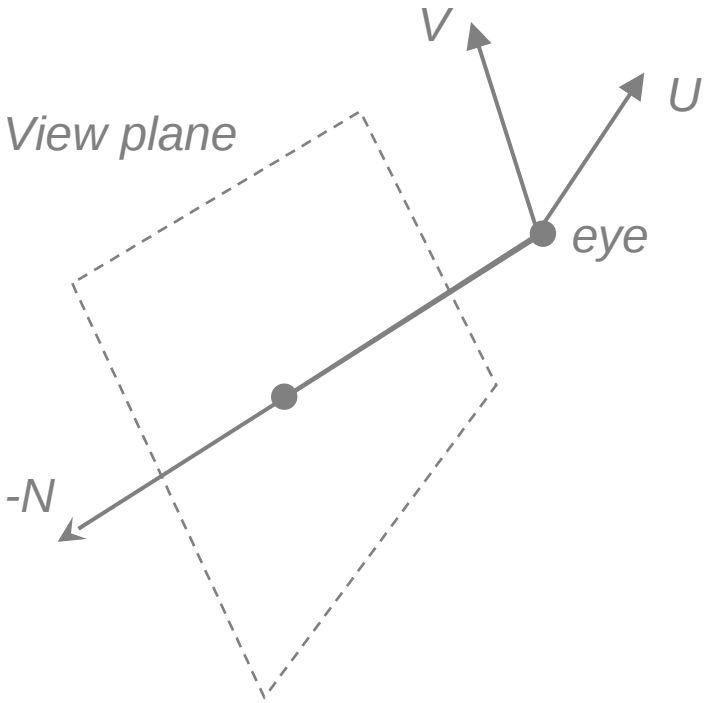
Image Precision	*Object Precision*
performed at the resolution of the display device	performed at the precision with which each object is defined
if the size of the image changes, visible-surface calculations have to be repeated	if the size of the image changes, visible-surface calculations do not have to be repeated
subject to aliasing (jaggies)	no aliasing problems
initially written for raster devices	initially written for vector devices

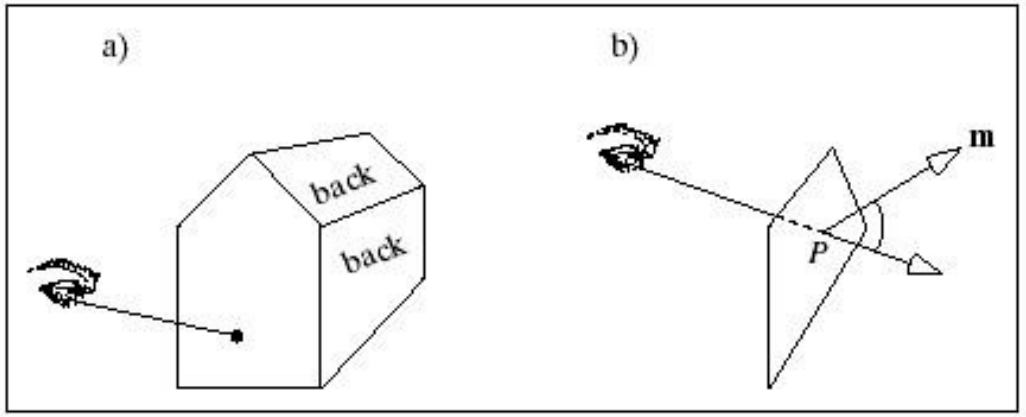
Izločanje zadnjih ploskev (backface culling)

- Polygone, die der Anwender nicht sieht, werden nicht dargstellt
- Wir unterscheiden zwei Methoden:
  - Bildschirmkoordinaten
  - Augkoordinaten



Izločanje zadnjih ploskev (backface culling)





F is *back face* if

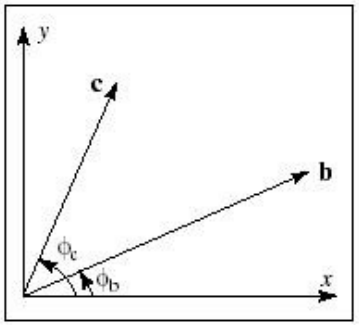
$$\left(P - eye\right)\ dot\ m_F > 0$$

Skalarni produkt (Dot Product)

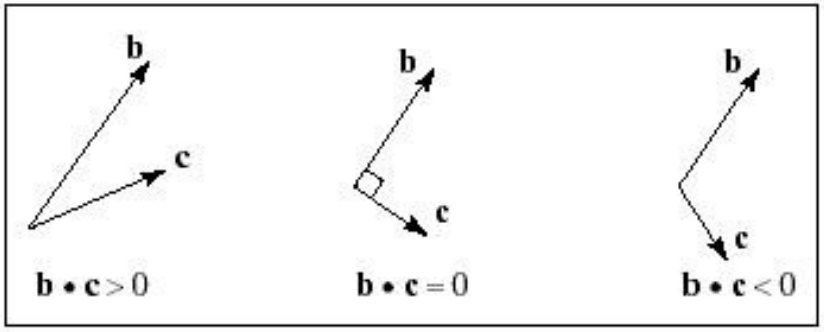
$$d = v \cdot w = \sum_{i=1}^{n} v_i w_i$$

Angle between Two Vectors

$$\cos(\theta) = \hat{b} \cdot \hat{c}$$

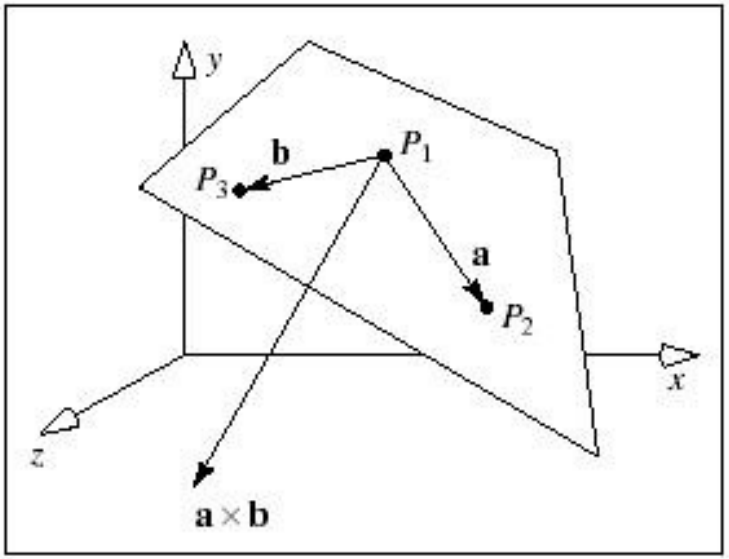


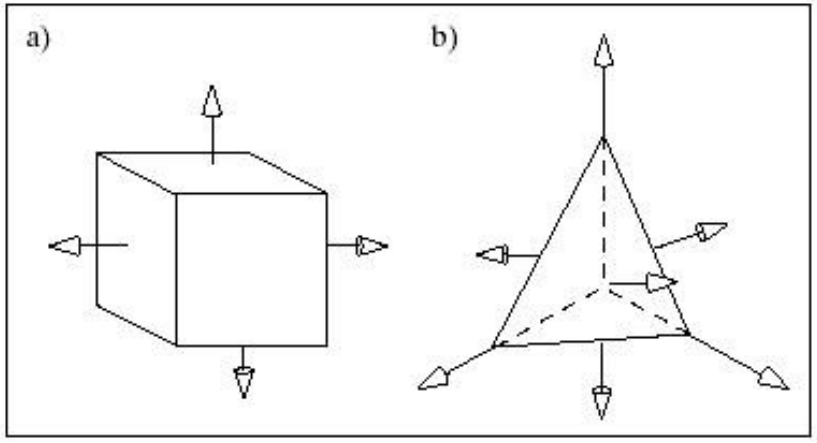
Sign and Perpendicularity

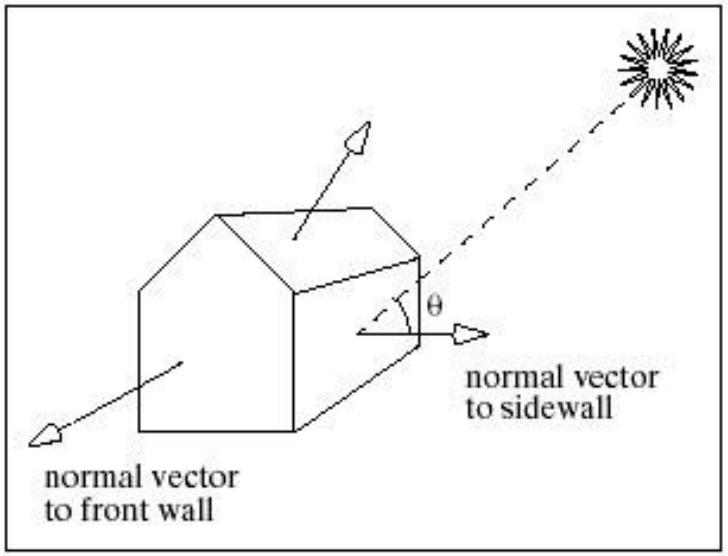


"perpendicular"
orthogonal
normal

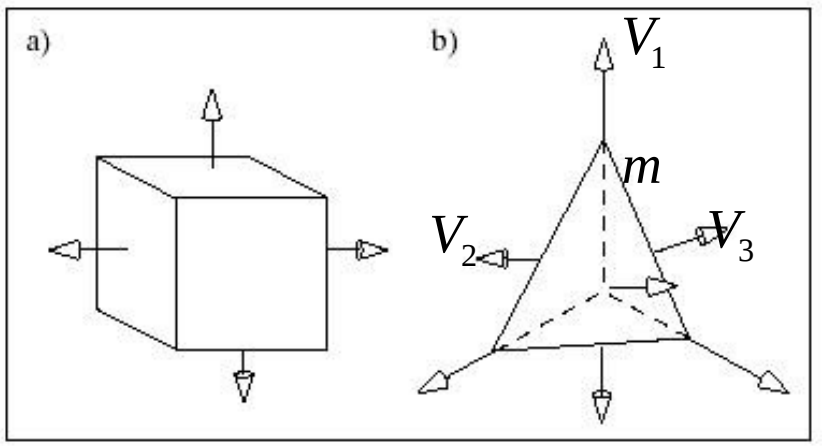
Normala na ravnino







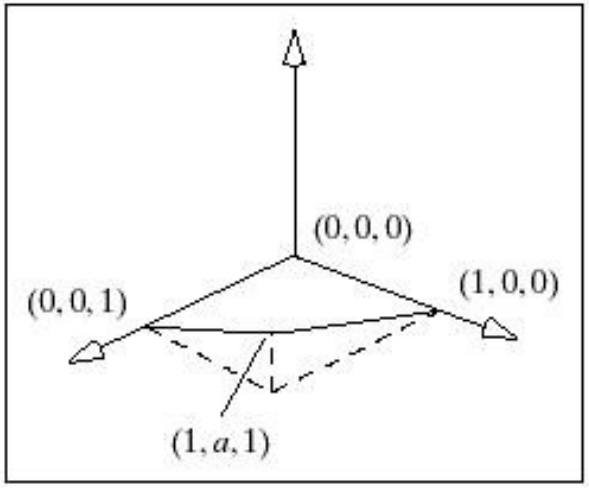
Kako dobimo vektorje normal



Flat Face:

$$m = (V_1 - V_2) \times (V_3 - V_2)$$

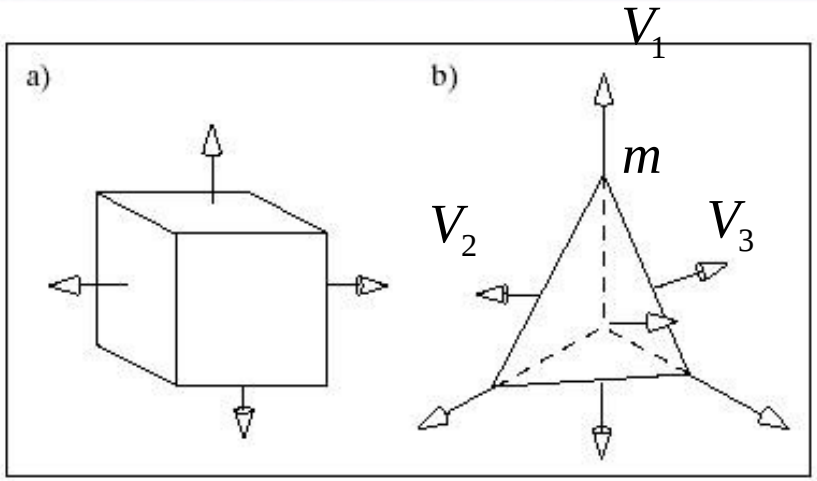
Two Problems:



Vectors nearly parallel, the cross product will be very small, Numerical inaccuracies may result

Polygon is not perfect planar

Finding the Normal Vectors



Robust Method (Newell):

$$m_x = \sum_{i=0}^{N-1} \left(y_i - y_{next(i)} \right) \left(z_i + z_{next(i)} \right)$$

$$m_y = \sum_{i=0}^{N-1} \left(z_i - z_{next(i)} \right) \left(x_i + x_{next(i)} \right)$$

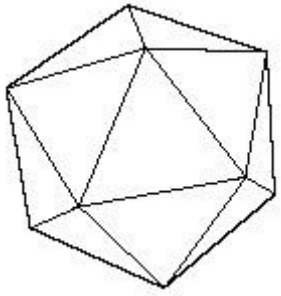
$$m_z = \sum_{i=0}^{N-1} \left(x_i - x_{next(i)} \right) \left(y_i + y_{next(i)} \right)$$

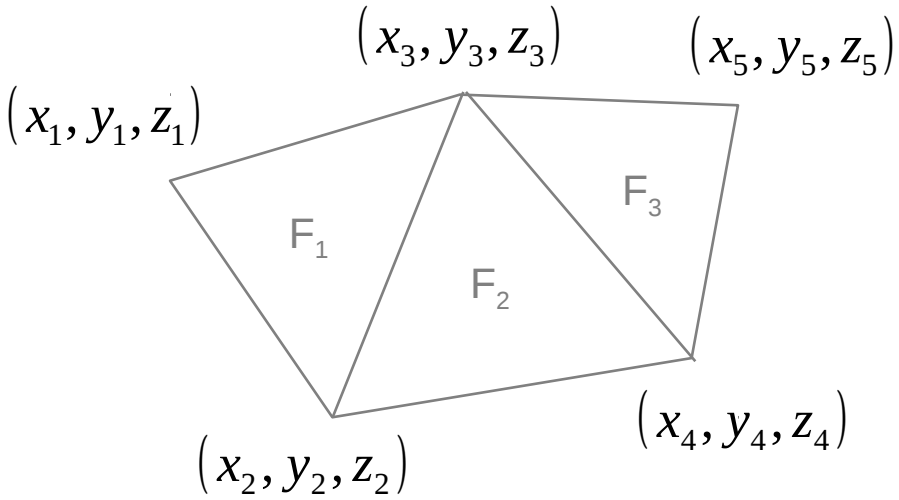
Tabela verteksov in ploskev (Vertex and Face Table)

Each face lists vertex references

Shared vertices

Still no topology information





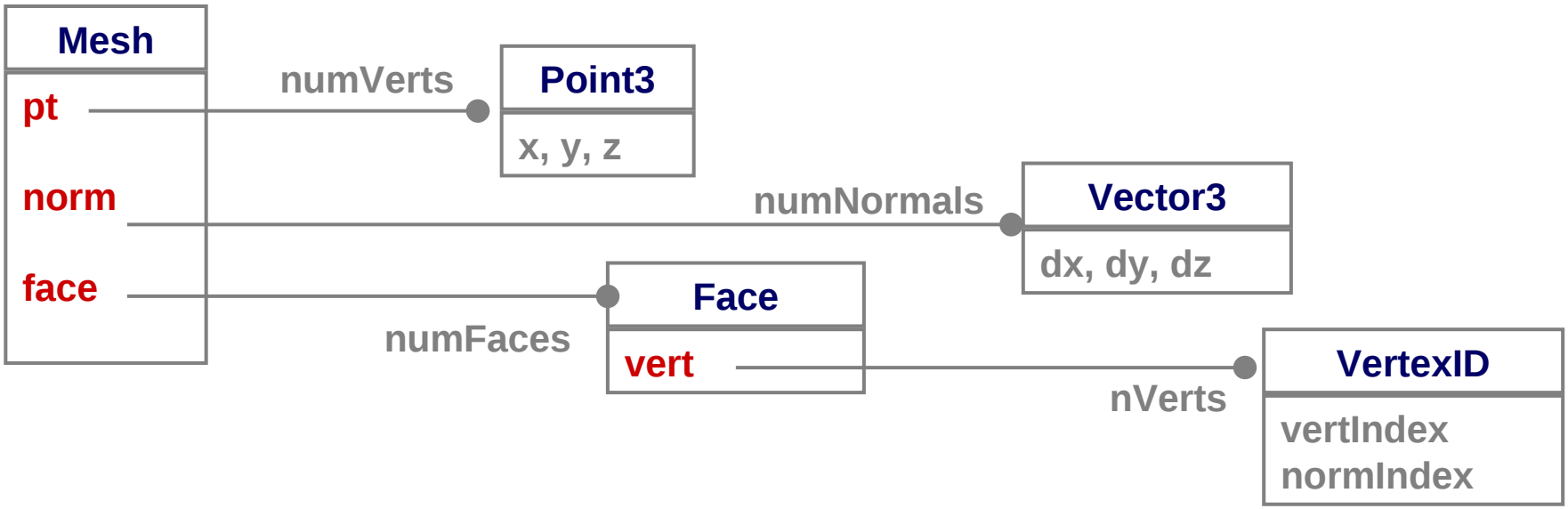
VERTEX TABLE

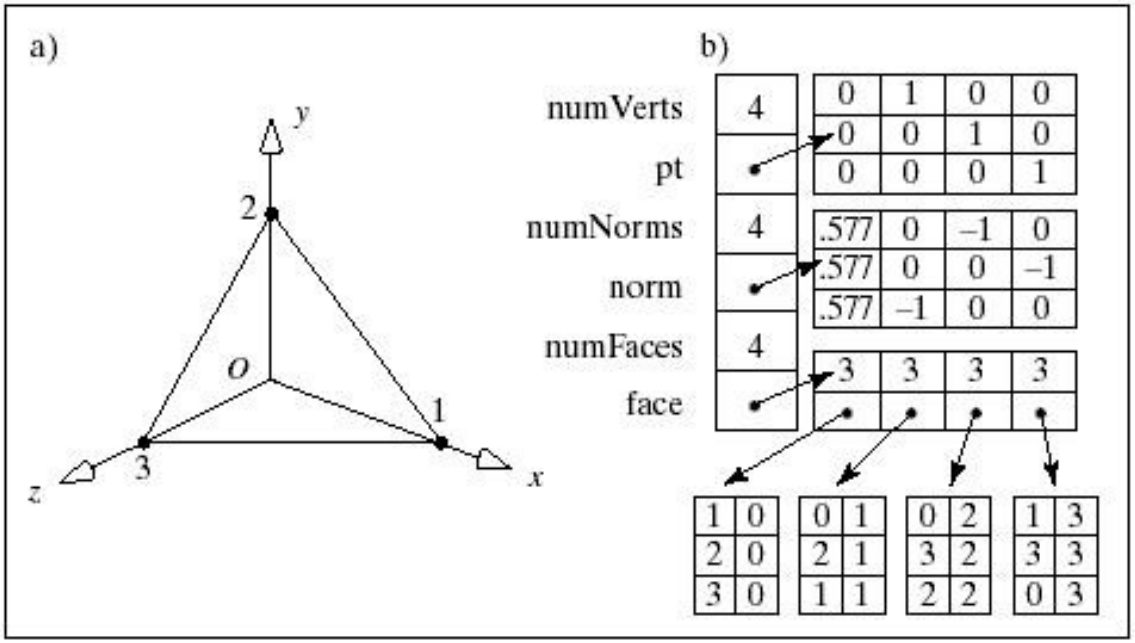
V_1	(x_1, y_1, z_1)
V_2	(x_2, y_2, z_2)
V_3	(x_3, y_3, z_3)
V_4	(x_4, y_4, z_4)
V_5	(x_5, y_5, z_5)

FACE TABLE

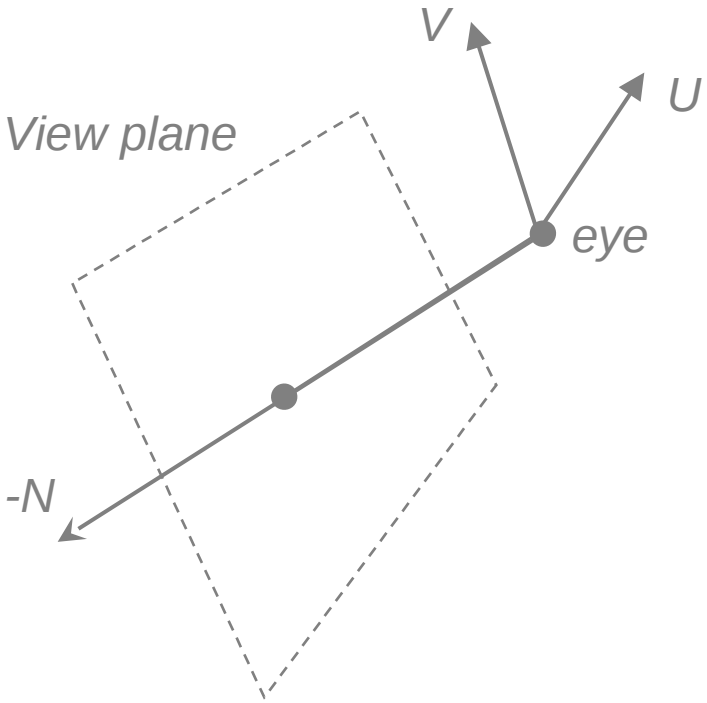
F_1	V_1	V_2	V_3
F_2	V_3	V_2	V_4
F_2	V_3	V_4	V_5

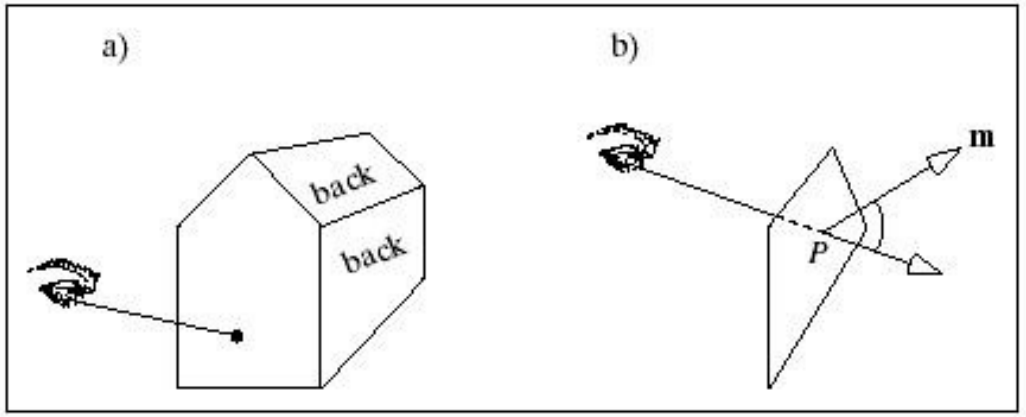
Podatkovne strukture





Izločanje zadnjih ploskev (backface culling)

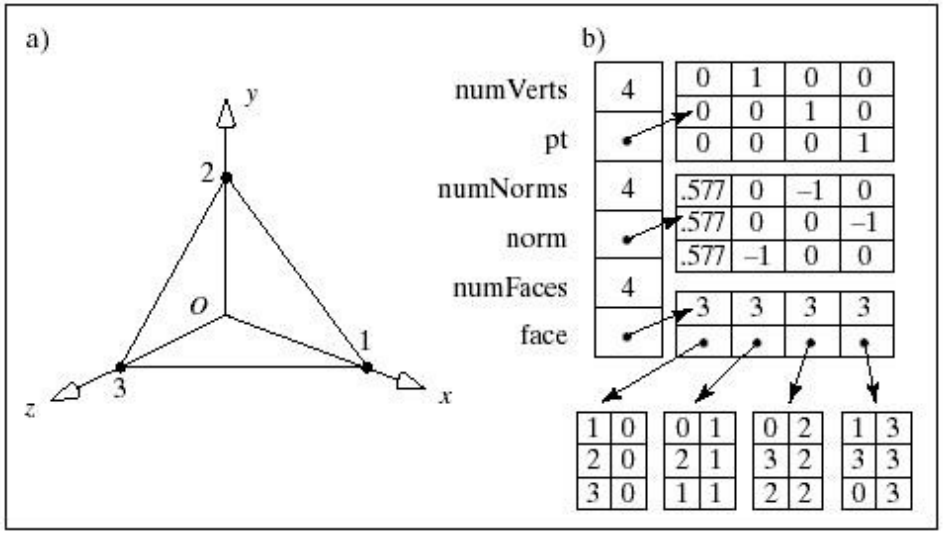




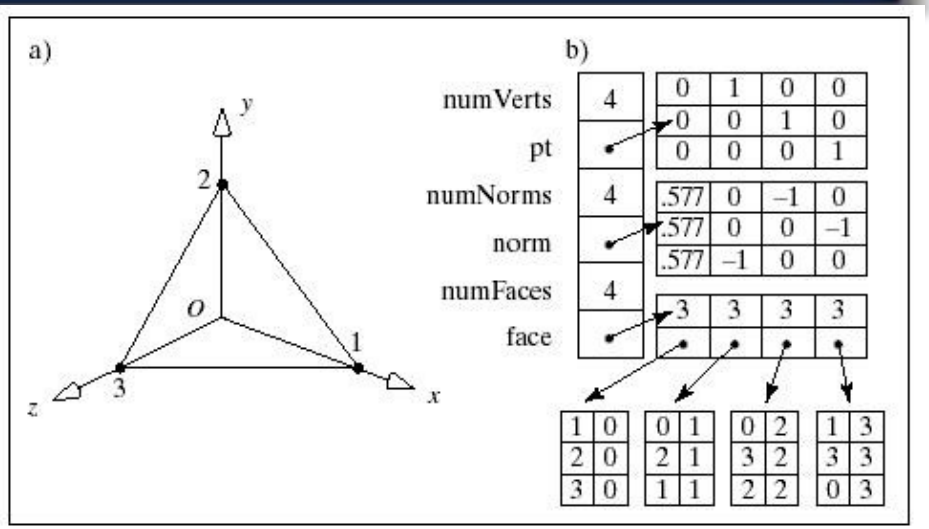
F is *back face* if

$$\left(P - eye\right)\ dot\ m_F > 0$$

Algoritem




```
void Mesh :: draw ()
{
    for (int f = 0; f < numFaces; f++)
    {
        glBegin (GL_POLYGON)
        for (int v = 0; v < face[f].nVerts; v++)
        {
            int iv = face[f].vert[v].vertIndex;
            glVertex3f(pt[iv].x, pt[iv].y, pt[iv].z);
        }
        glEnd();
    }
}
```




```
void Mesh :: draw ()
{
    for (int f = 0; f < numFaces; f++)
    {
        If ( isBackFace ( f, eye ) ) continue;
        glBegin (GL_POLYGON)
        for (int v = 0; v < face[f].nVerts; v++)
        {
            int iv = face[f].vert[v].vertIndex;
            glVertex3f(pt[iv].x, pt[iv].y, pt[iv].z);
        }
        glEnd();
    }
}
```

Z - buffer Algorithm

Advantages

- easy to implement
- no presorting
- no object-object comparisons
- saves rendering time
- can be hardware implemented

Disadvantages

- memory requirements
- time wasted overwriting pixels

Z-buffer algorithm

- The z-buffer or depth-buffer is one of the simplest visible-surface algorithms.
- Z-buffering is an image-space algorithm, an involves the use of a (surprisingly) z-buffer, to store the depth, or z-value of each pixel.
- All of the elements of the z-buffer are initially set to be 'very far away.' Whenever a pixel colour is to be changed the depth of this new colour is compared to the current depth in the z-buffer. If this colour is 'closer' than the previous colour the pixel is given the new colour, and the z-buffer entry for that pixel is updated as well. Otherwise the pixel retains the old colour, and the z-buffer retains its old value.

Z-buffer algorithm

- Pseudo code…

```
for each polygon for each pixel p in the polygon's projection
{
    pz = polygon's normalized z-value at (x, y);
    //z ranges from -1 to 0

    if (pz > zBuffer[x, y]) // closer to the camera
      { zBuffer[x, y] = pz;
      framebuffer[x, y] = colour of pixel p;
    }
}
```

Z-Buffer Algorithm

```
for( each pixel(i, j) )        // clear Z-buffer and frame buffer
{
    z_buffer[i][j] = far_plane_z;
    color_buffer[i][j] = background_color;
}

for( each face A)
    for( each pixel(i, j) in the projection of A)
    {
        Compute depth z and color c of A at (i,j);
        if( z > z_buffer[i][j] )
        {
            z_buffer[i][j] = z;
            color_buffer[i][j] = c;
        }
    }
```

Uporaba prednostnih seznamov

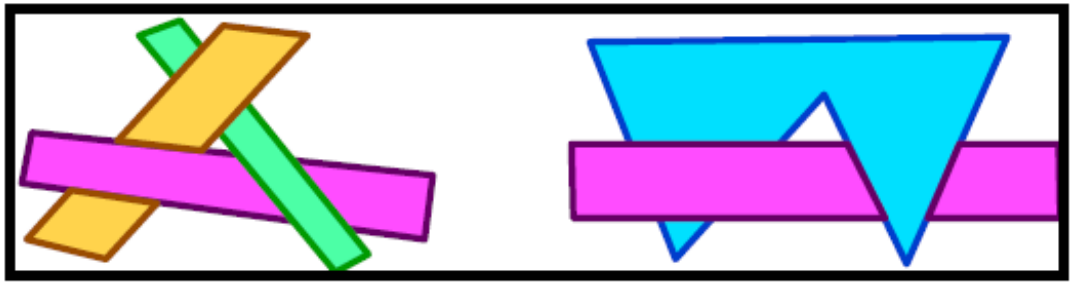
Idee: Reihenfolge der Polygone entsprechend ihrer Tiefe sortieren, sodass die Darstellung geordnet durchgeführt werden kann.

Verfahren:

- Depth-Sort Algorithmus (Painters Algorithm)
- BSP-Algorithmus

Algoritem razvrščanja po globini (depth sort)

Idee: Polygone werden von hinten nach vorne sortiert und dann in dieser Reihenfolge gezeichnet.



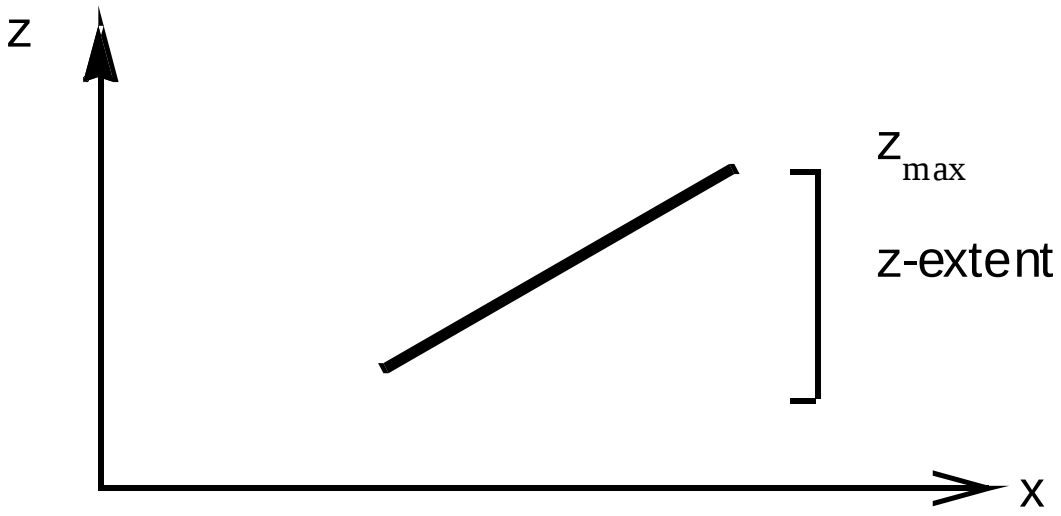
1. Vorsortierung aller Polygone bzgl. ihrer Entfernung vom Bildschirm (z-Wert)
2. Auflösen aller Unregelmäßigkeiten, wenn nötig Polygone zerschneiden.
3. Rasterisierung aller Polgone (von hinten nach vorne)

Depth-sort algorithm

- **a.k.a. The Painter's Algorithm**
- The idea here is to go back to front drawing all the objects into the frame buffer with nearer objects being drawn over top of objects that are further away.
- Simple algorithm:
  - Sort all polygons based on their farthest z coordinate
  - Resolve ambiguities
  - draw the polygons in order from back to front
- This algorithm would be very simple if the z coordinates of the polygons were guaranteed never to overlap. Unfortunately that is usually not the case, which means that step 2 can be somewhat complex.

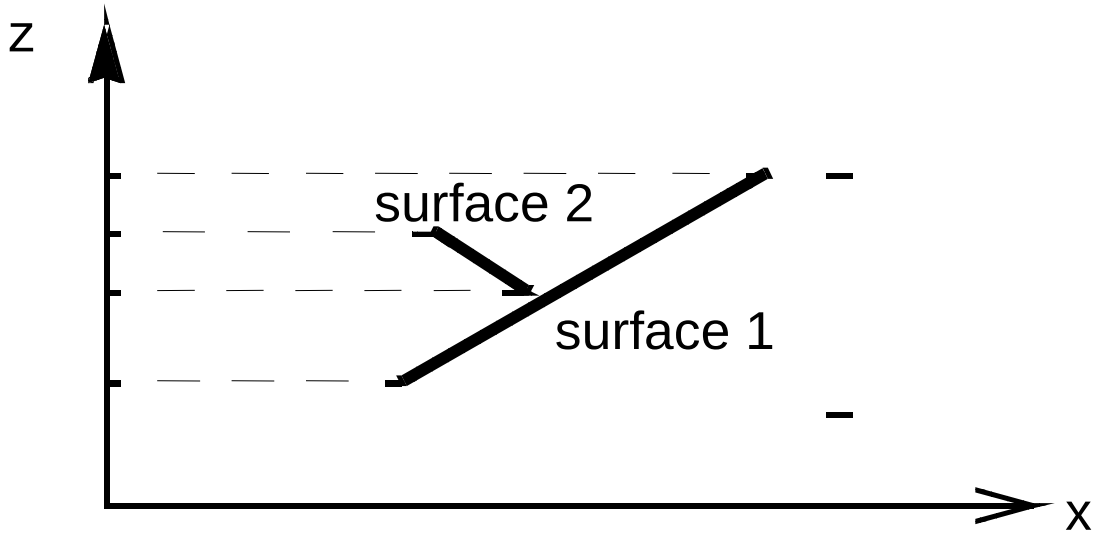
Depth-sort algorithm

- First must determine z-extent for each polygon

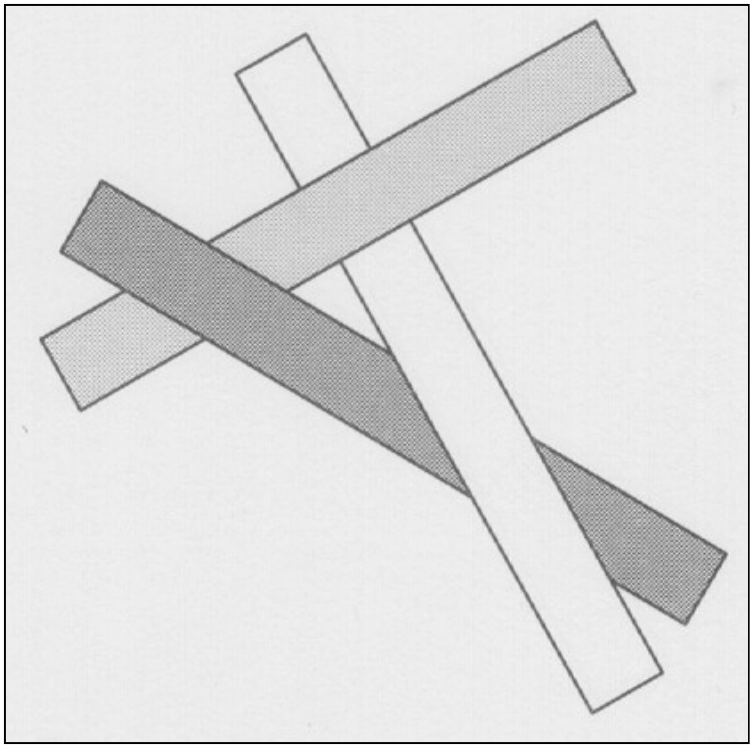


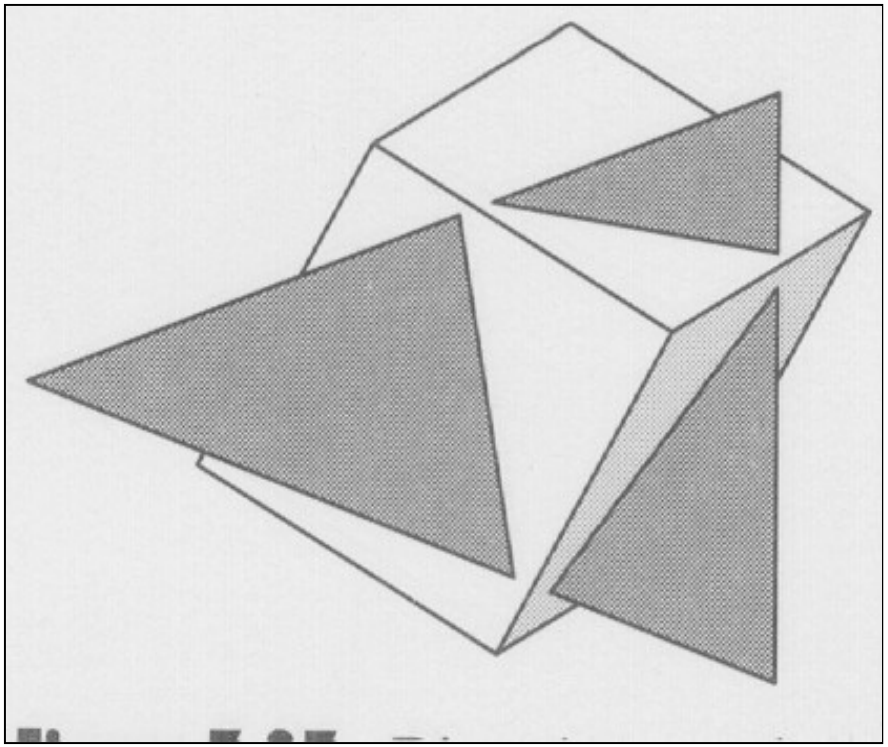
Depth-sort algorithm

- Ambiguities arise when the z-extents of two surfaces overlap.



Problem Cases: Cyclic and Intersecting Objects





Depth-sort algorithm

- Any polygons whose z extents overlap must be tested against each other.
- We start with the furthest polygon and call it P. Polygon P must be compared with every polygon Q whose z extent overlaps P's z extent. 5 comparisons are made. If any comparison is true then P can be written before Q. If at least one comparison is true for each of the Qs then P is drawn and the next polygon from the back is chosen as the new P.

Depth-sort algorithm

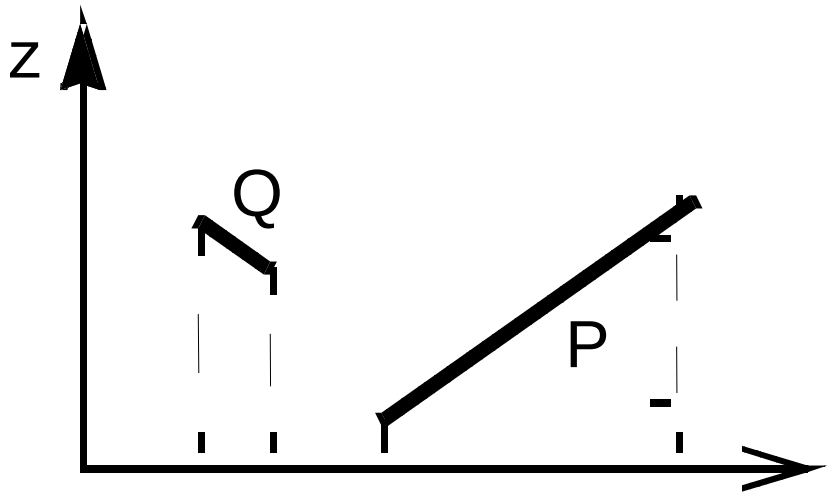
1. do P and Q's x-extents not overlap.
2. do P and Q's y-extents not overlap.
3. is P entirely on the opposite side of Q's plane from the viewport.
4. is Q entirely on the same side of P's plane as the viewport.
5. do the projections of P and Q onto the (x,y) plane not overlap.

- If all 5 tests fail we quickly check to see if switching P and Q will work. Tests 1, 2, and 5 do not differentiate between P and Q but 3 and 4 do. So we rewrite 3 and 4

3’. is Q entirely on the opposite side of P's plane from the viewport.
4’. is P entirely on the same side of Q's plane as the viewport.

Depth-sort algorithm

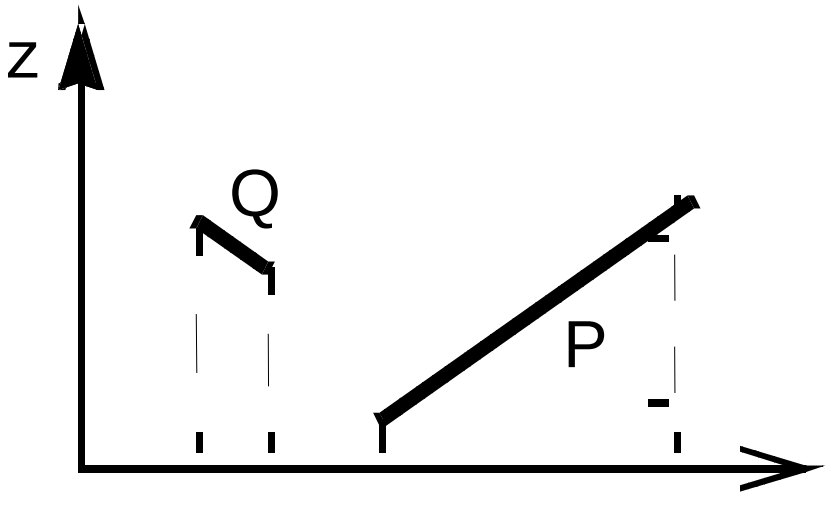
x - extents not overlap?



if they do, test fails

Depth-sort algorithm

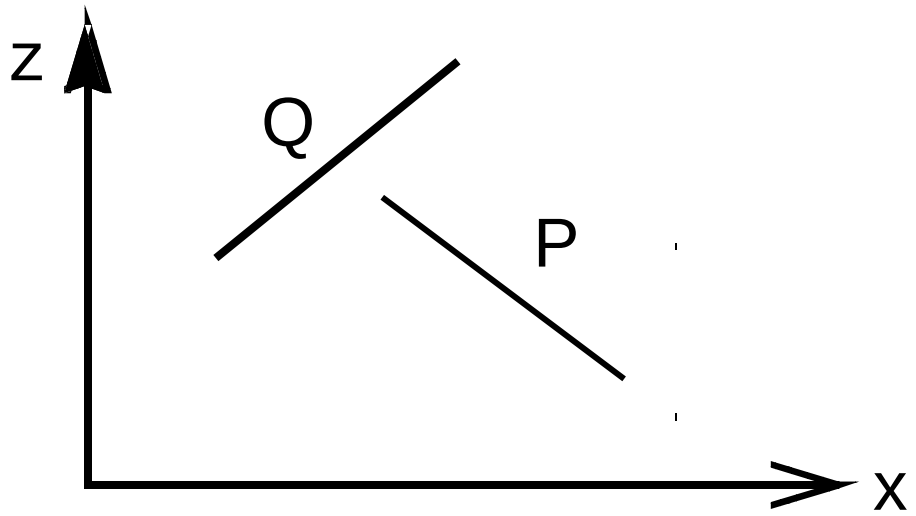
y - extents not overlap?



if they do, test fails

Depth-sort algorithm

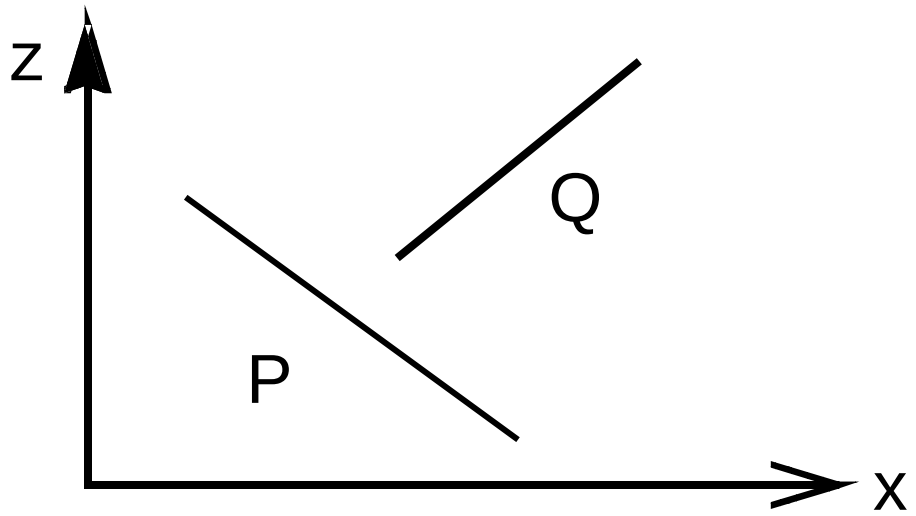
is P entirely on the opposite side of Q's plane from the viewport.



Test is true…

Depth-sort algorithm

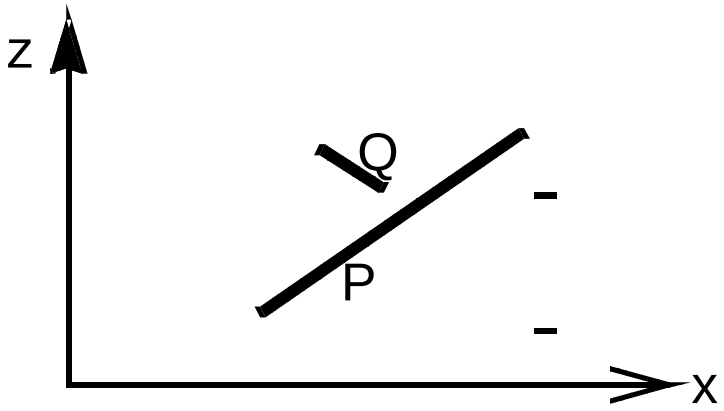
is Q entirely on the same side of P's plane as the viewport.

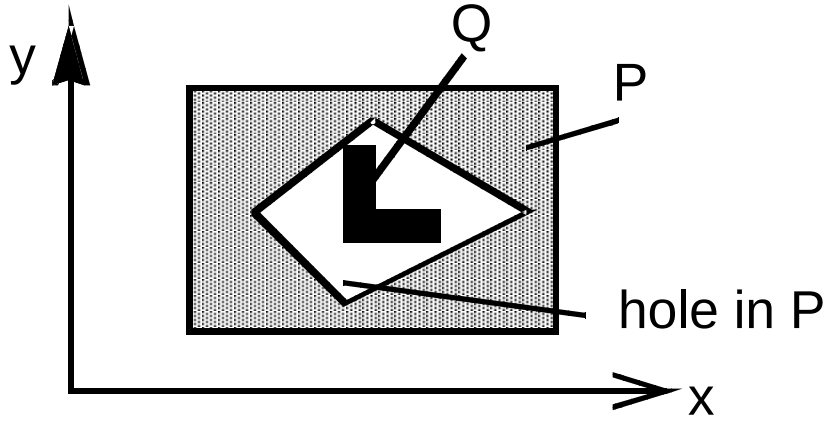


Test is true…

Depth-sort algorithm

do the projections of P and Q onto the (x,y) plane not overlap.





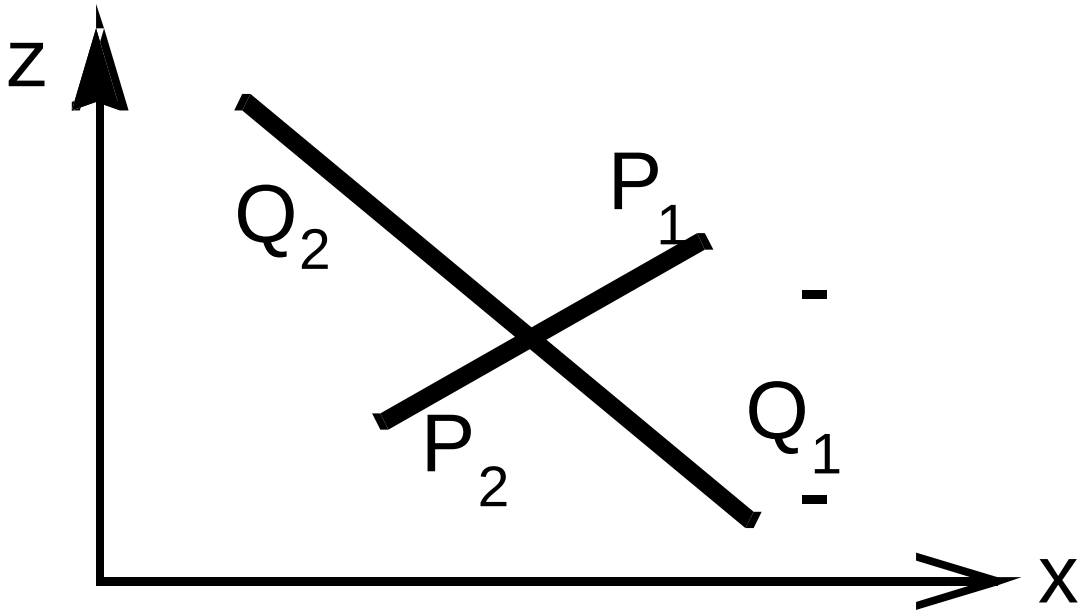
Test is true…

Depth-sort algorithm

- If all tests fail…
  - … then reverse P and Q in the list of surfaces sorted by Z_{max}
  - set a flag to say that the test has been performed once.
  - If the tests fail a second time, then it is necessary to split the surfaces and repeat the algorithm on the 4 surfaces

Depth-sort algorithm

- End up drawing Q2,P1,P2,Q1



BSP Trees

- Idea

  Preprocess the relative depth information of the scene in a tree for later display

- Observation

  The polygons can be painted correctly if for each polygon F:

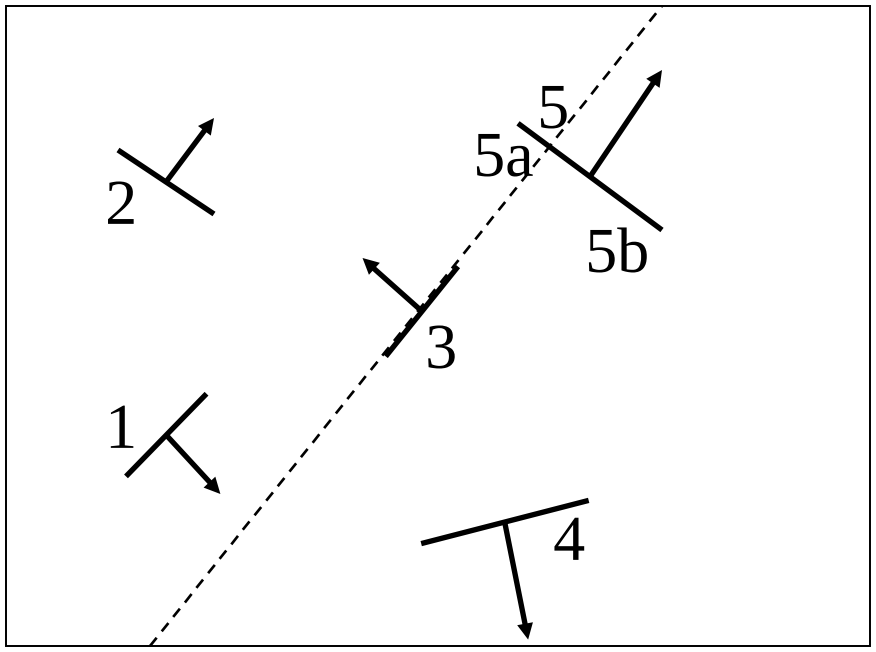
  – Polygons on the other side of F from the viewer are painted before F

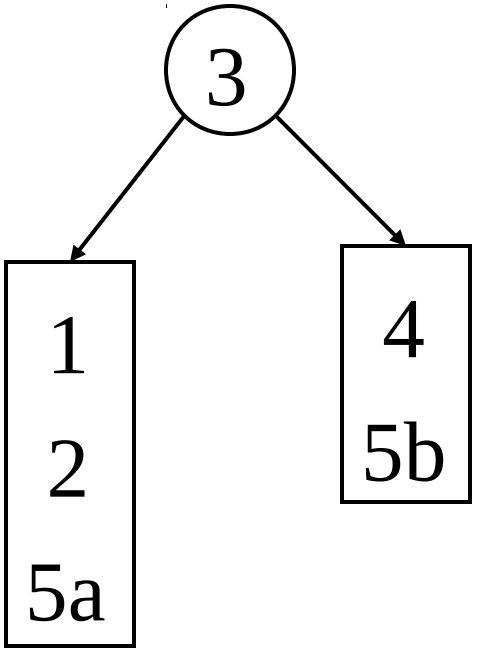
  – Polygons on the same side of F as the viewer are painted after F

Building a BSP Tree

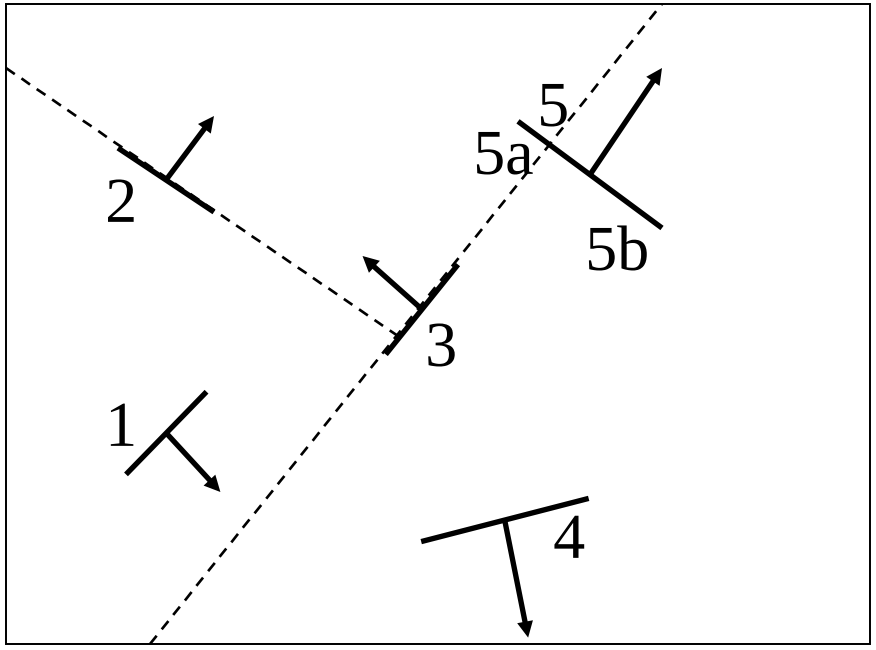
```
Typedef struct {
    polygon root;
    BSP_tree *backChild, *frontChild;
} BSP_tree;

BSP_tree  *makeBSP(polygon *list)
{
    if( list = NULL) return NULL;

    Choose polygon F from list;
    Split all polygons in list according to F;

    BSP_tree* node = new BSP_tree;
    node->root = F;
    node->backChild = makeBSP( polygons on front side of F );
    node->frontChild = makeBSP( polygons on back side of F );
    return node;
}
```

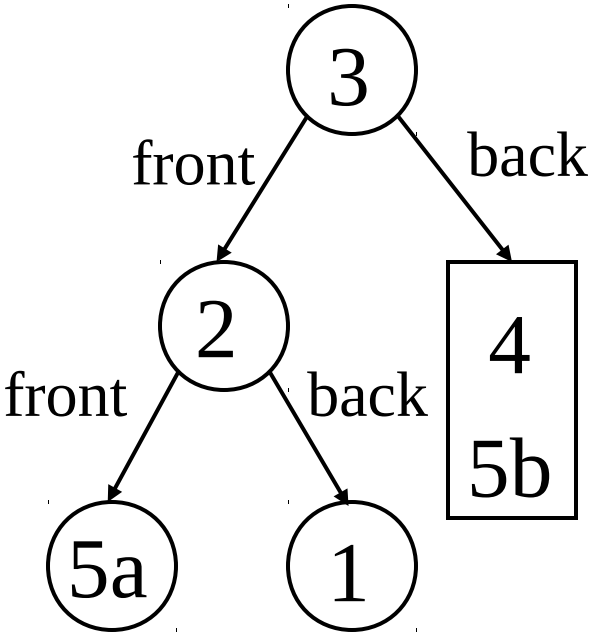
Building a BSP Tree (2D)



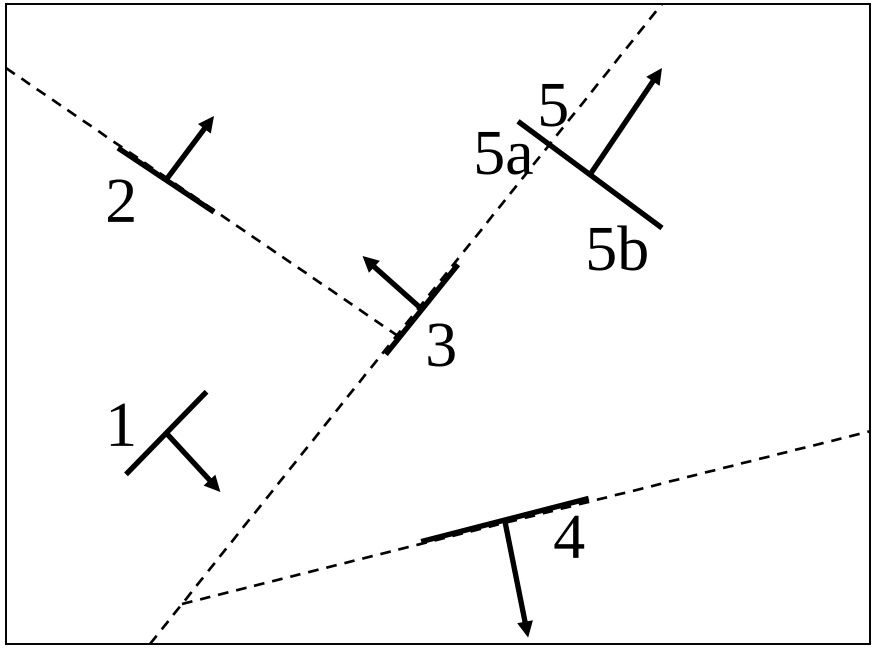


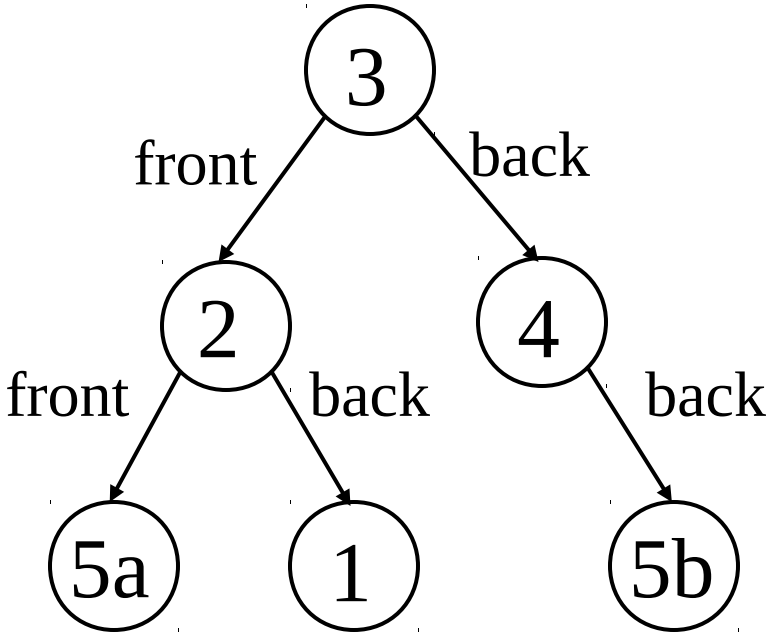
Building a BSP Tree (2D)



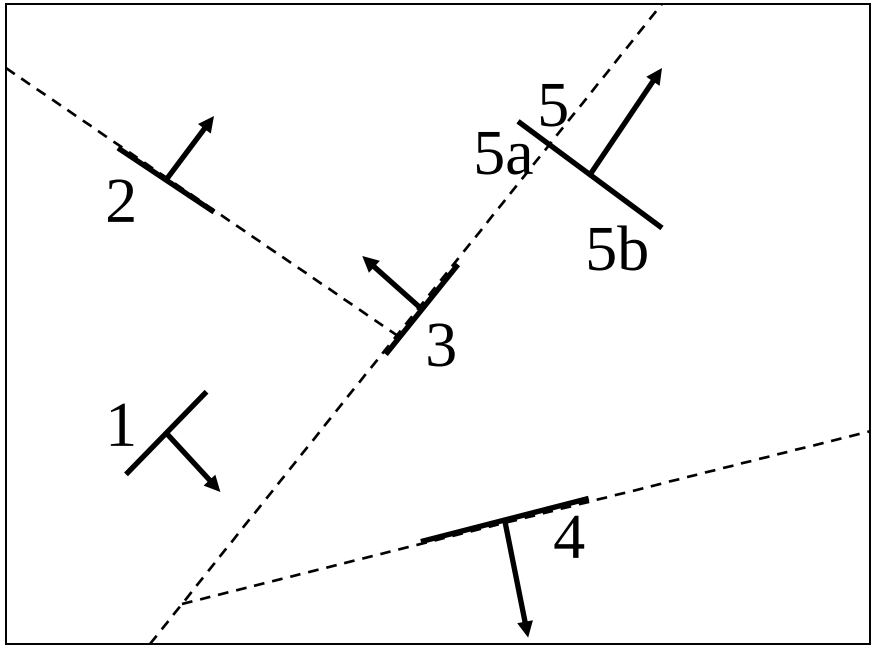


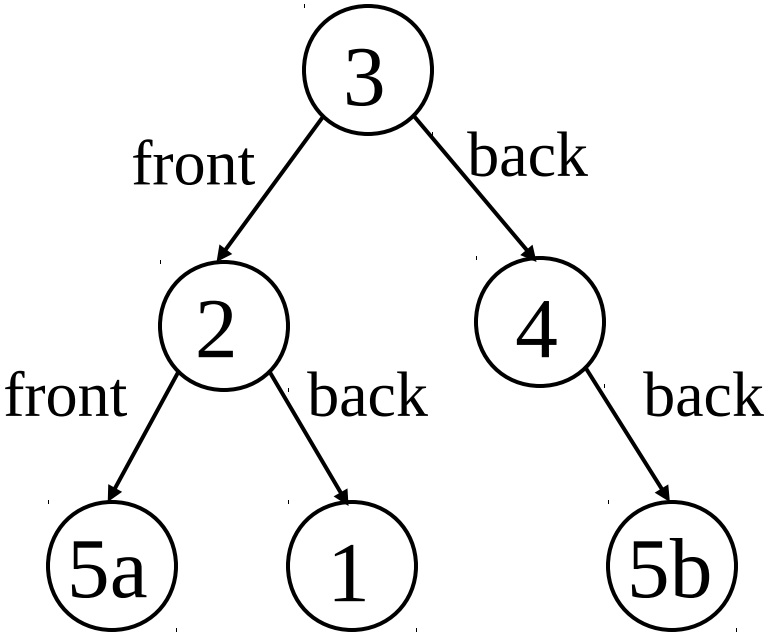
Building a BSP Tree (2D)





Displaying a BSP Tree





Display order: **4, 5b, 3, 5a, 2, 1 (only 3 is front facing)**

Displaying a BSP Tree

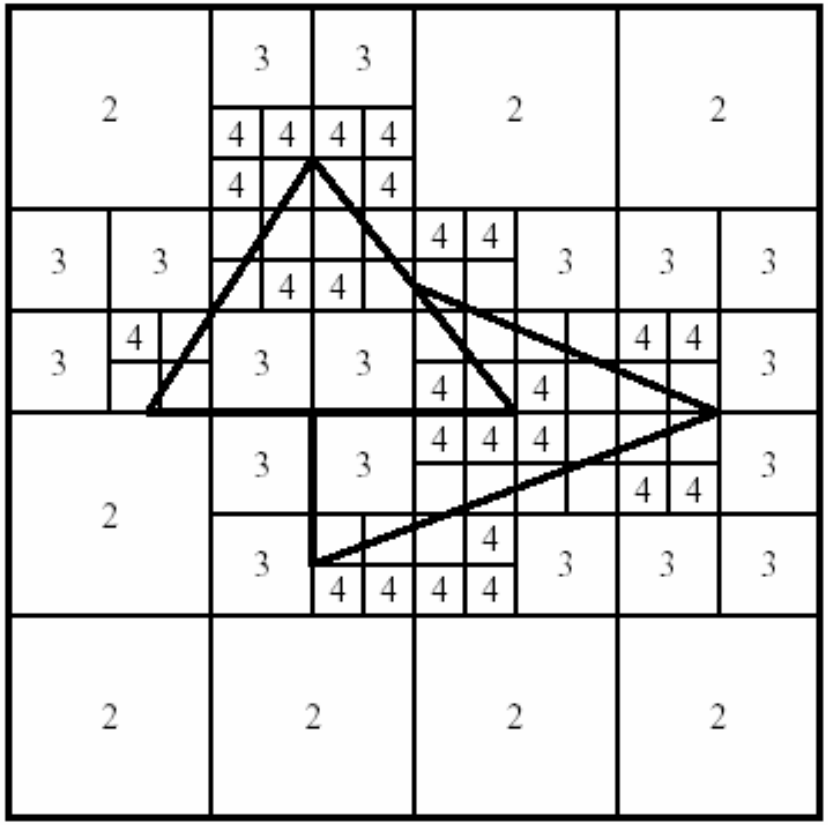
```
void  displayBSP ( BSP_tree *T )
{
      if ( T != NULL) {
            if ( viewer is in front of T->root ) {    // display backChild first
                  displayBSP ( T->backChild );
                  displayPolygon ( T->root );
                  displayBSP ( T->frontChild );
            }
            else {     // display frontChild first
                  displayBSP ( T->frontChild );
                  displayPolygon ( T->root );
                  displayBSP ( T->backChild );
            }
}
```

BSP Trees: Analysis

- Advantages
  - Efficient
  - View-independent
  - Easy transparency and antialiasing
- Disadvantages
  - Tree is hard to balance
  - Not efficient for small polygons

Warnockov algoritem

Recursively subdivide the screen until only one triangle is visible inside each rectangle.



Warnock PseudoCode

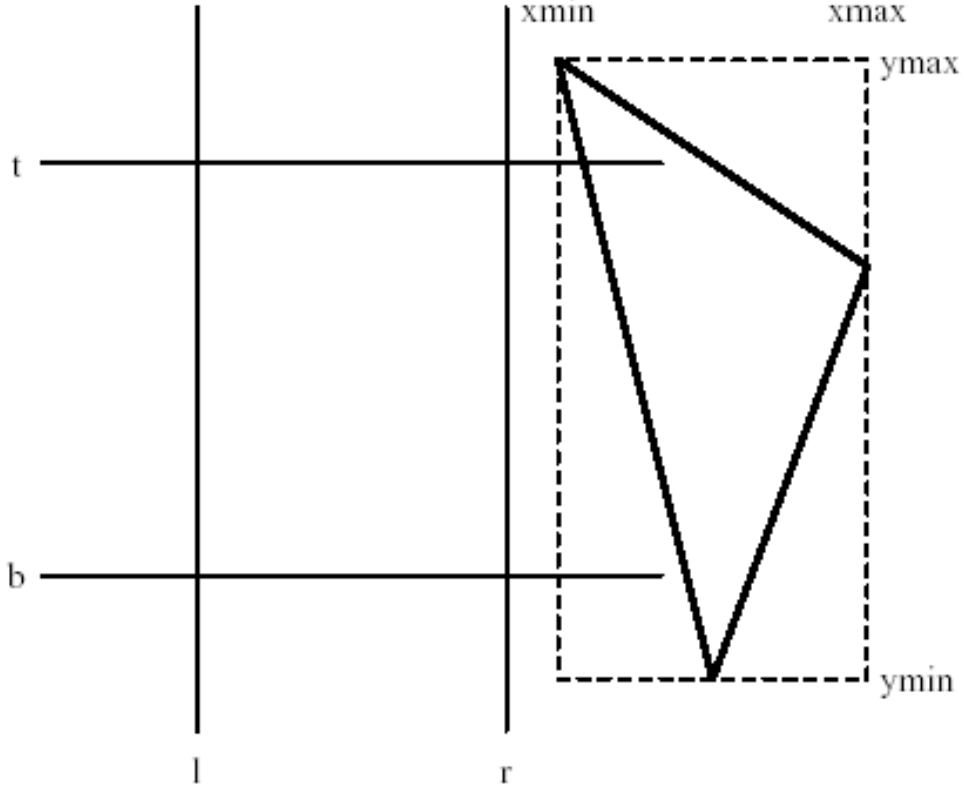
```
void warnock(rectangle r)
{
  if r is a pixel
      if r covers no triangles, color with background color
      else color according to nearest triangle
  if r is a non−pixel rectangle
      if r contains only background, color with background color
      if r is completely contained in a triangle
          and that triangle is nearer than any other overlapping triangles
          then color according to the nearest triangle.
  else
    subdivide into four rectangles: r1, r2, r3, r4
    warnock(r1); warnock(r2); warnock(r3); warnock(r4);
}
```

Two Main Problems

1. **Testing for overlap and/or containment between rectangle and triangles**
   (a) Intersection of bounding box of triangle with rectangle.
   (b) Intersection of triangle edges with rectangle.
   (c) Containment of rectangle inside triangle or vice versa.
2. **Determining the nearest enclosing triangle**
   (a) Vertex test.

It is ok to miss some termination cases—if they can be caught during subsequent subdivisions.

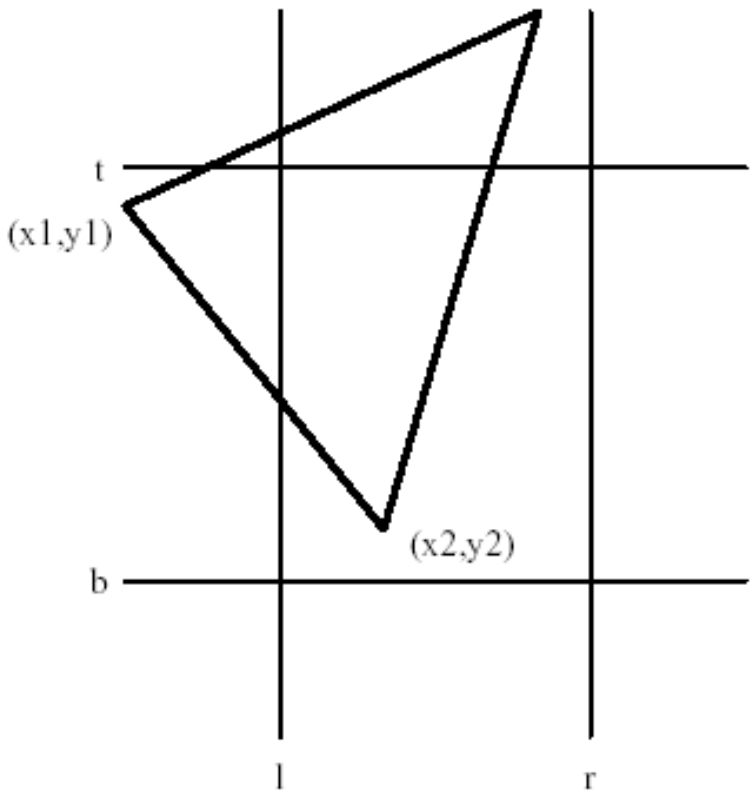
Overlap and Containment (1): Bounding Box Test



There is no overlap if

$$xmax < l \quad \text{or} \quad xmin > r \quad \text{or} \quad ymax < b \quad \text{or} \quad ymin > t$$

Overlap and Containment (2): Edge Intersection Tests

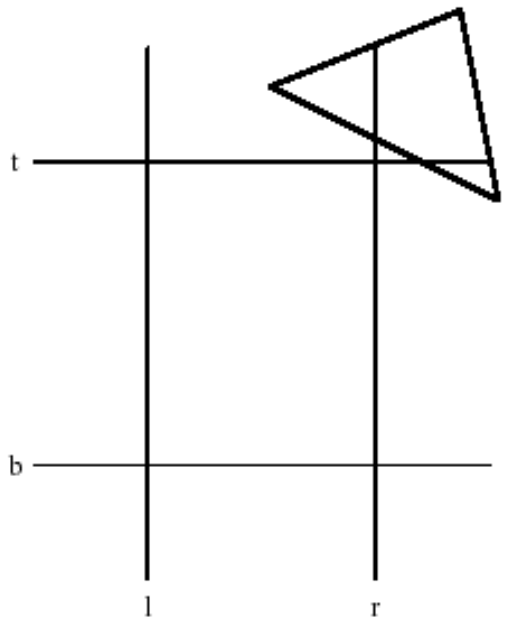


Compute

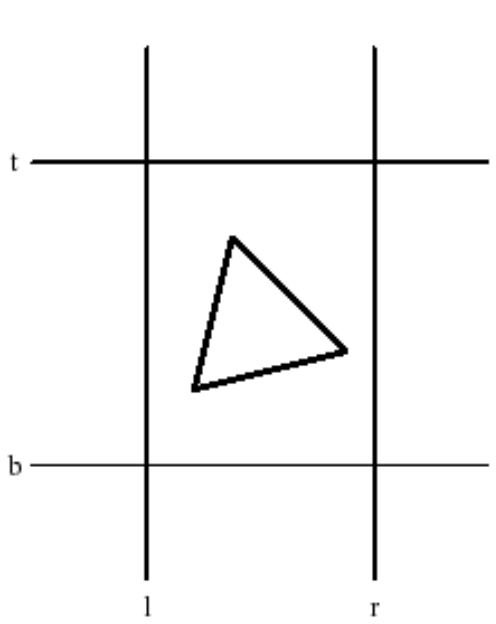
$$y = y_1 + \frac{l - x_1}{x_2 - x_1}(y_2 - y_1)$$

If $b < y < t$, then the edge from (x_1, y_1) to (x_2, y_2) intersects the left side.

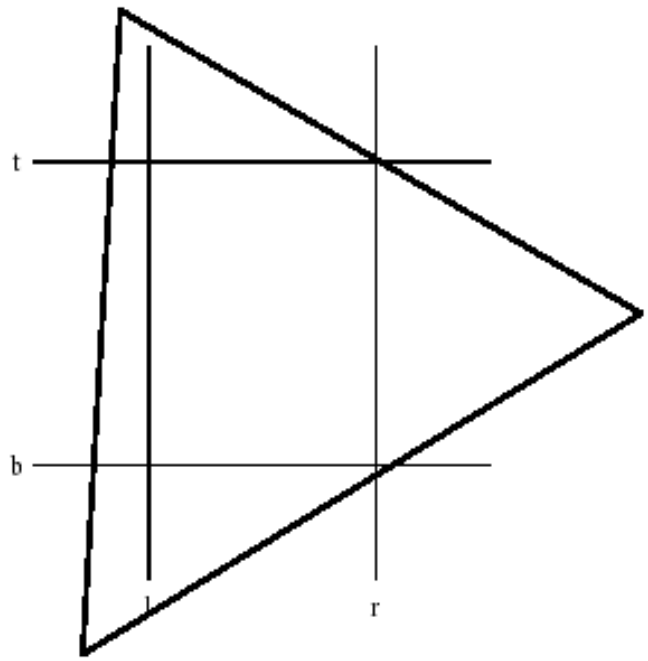
Overlap and Containment (3): Containment Tests



Case 1



Case 2



Case 3

If one vertex of triangle is inside rectangle: **Case 2.**

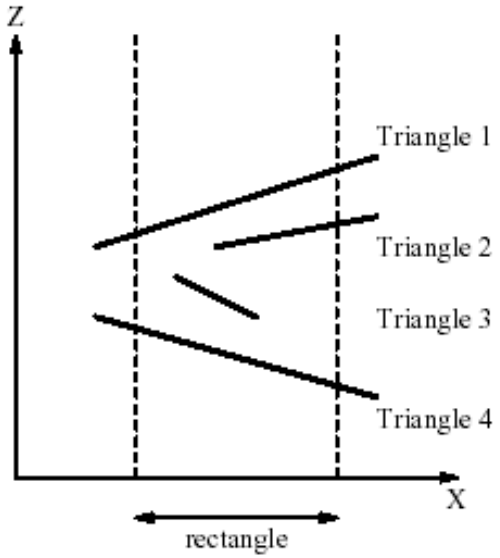
If one vertex of rectangle is inside triangle: **Case 3.**

Else: **Case 1.**

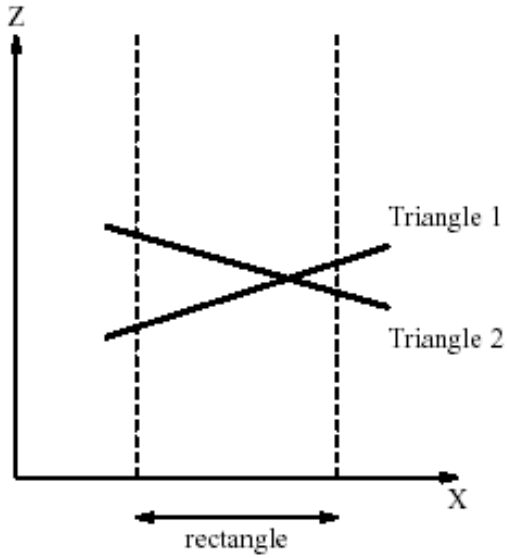
Based on these tests, we can classify every triangle as either **disjoint**, **intersecting** (including fully inside), or **enclosing**. If all triangles are disjoint, we fill with background color and return.

Nearest Enclosing Triangle

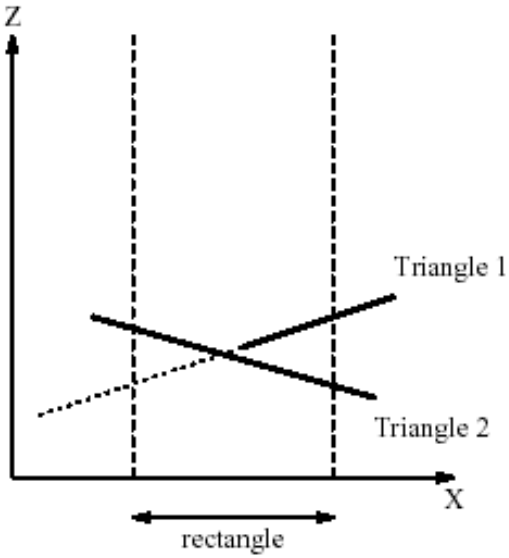
If there are any **enclosing** triangles, then we must check to see if one of them hides all of the other **intersecting** and **enclosing** triangles. At each vertex of the rectangle, compute the z depth of the plane of each overlapping triangle.



Case 1



Case 2



Case 3

If one triangle is nearest at all four vertexes (**Case 1**), we fill the rectangle with its color and return.

Otherwise, we subdivide and process the four subrectangles recursively. Note that we miss an opportunity in **Case 3**.

Efficient Classification of Triangles

Two key facts to exploit:

- **If a triangle is disjoint from a rectangle, then it is disjoint from all subdivisions of that rectangle.**
- **If a triangle encloses a rectangle, then it encloses all subdivisions of that rectangle.**

Maintain three linked lists:

`dlist` All triangles known to be disjoint with the current rectangle.

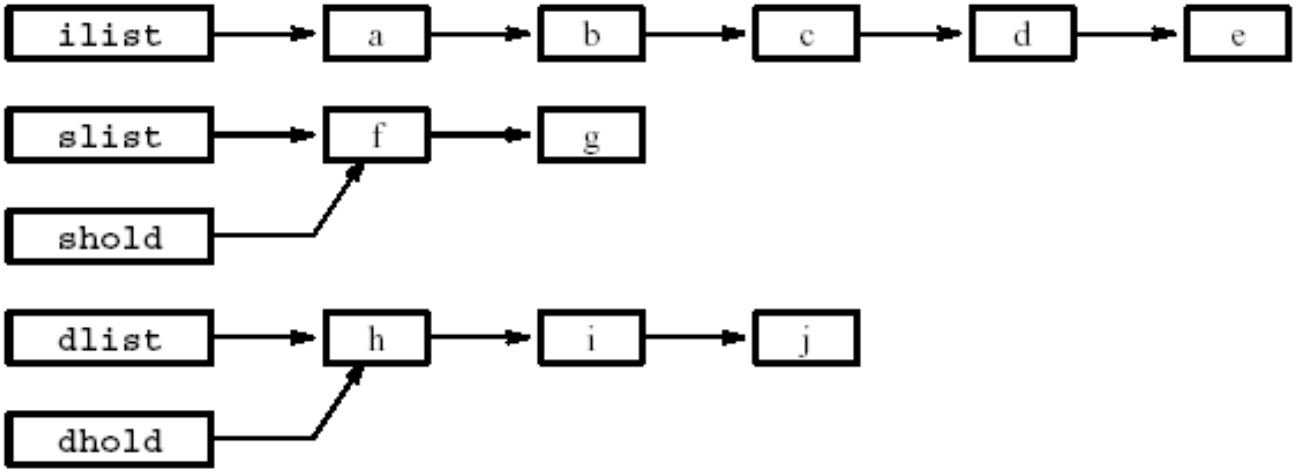
`slist` All triangles known to enclose the current rectangle ("surround" list).

`ilist` All other triangles (they are all potentially intersecting).

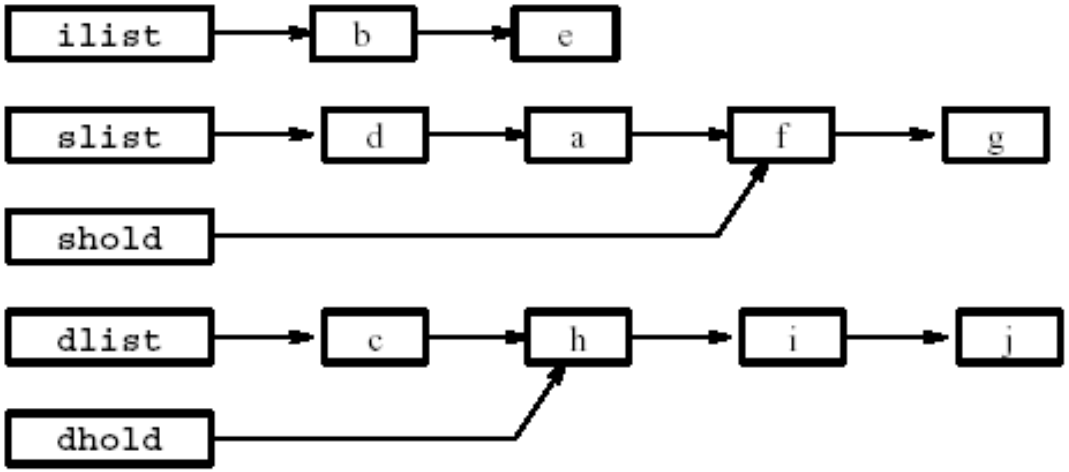
```
void warnock(rectangle r, triangle_t* dlist, triangle_t* slist, triangle_t* ilist)
{
  For each triangle on ilist
    perform overlap tests and move to dlist or slist if appropriate
  If slist is empty
    If ilist is empty
        Paint background color
        Restore ilist and return
    Else
        If there is a member of slist closer than all other elements
        of slist and ilist
            Paint with its color.
            Restore ilist and return
  Subdivide r into r1, r2, r3, r4
  warnock(r1,dlist,slist,ilist); warnock(r2,dlist,slist,ilist);
  warnock(r3,dlist,slist,ilist); warnock(r4,dlist,slist,ilist);
  Restore ilist and return.
}
```

A Fast Way to Restore `ilist`

Introduce pointers **shold** and **dhold**.



Distribute the rectangles from **ilist** onto **dlist** and **slist**:



Demonstracija – primerjava metod