Rasterization and Visibility Algorithms

Gonzalo Besuievsky

IMAE - UdG

Universitat de Girona

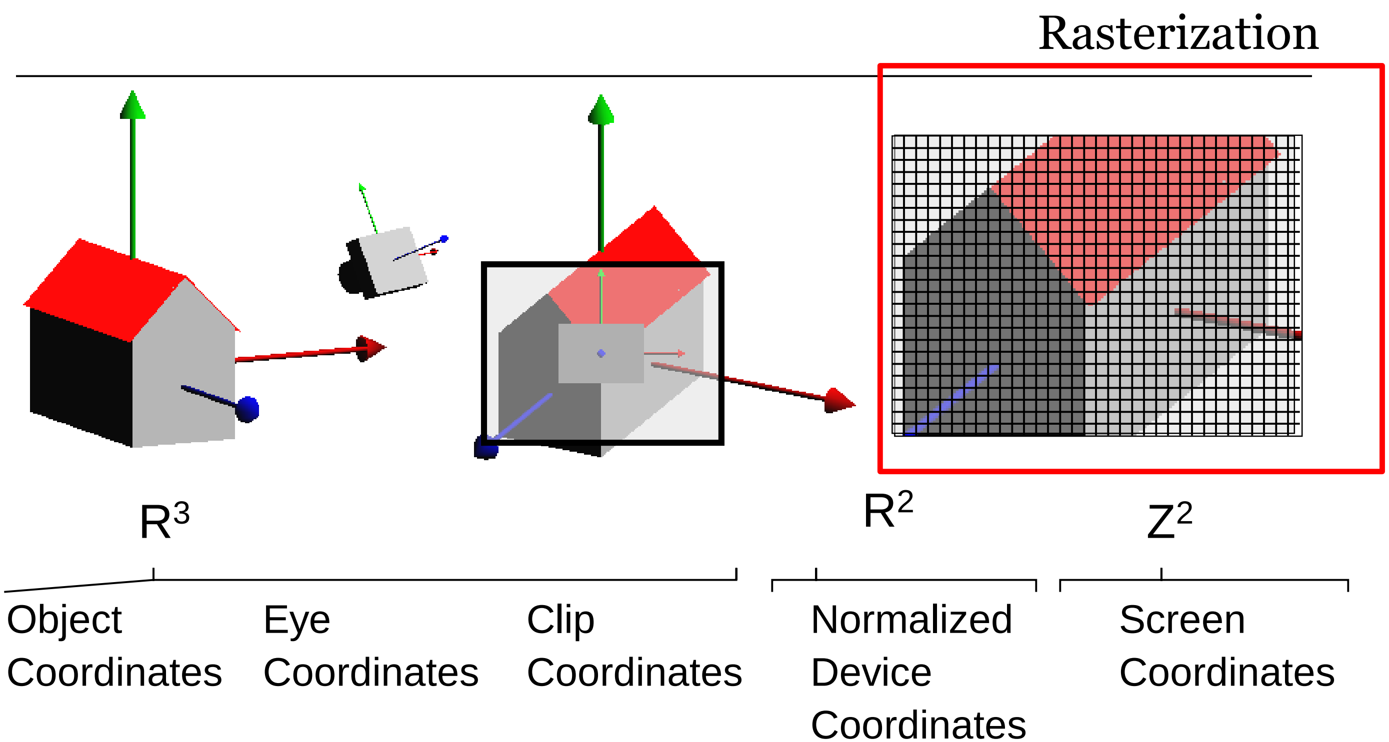
Contents

- Rasterization
  - Basic Concepts
  - Bresenham Algorithms
- Visibility Algorithms
  - Basic Problem
  - Painter Algorithm
  - BSP
  - Ray Casting
- Visibility Optimization

RASTERIZATION

Universitat de Girona

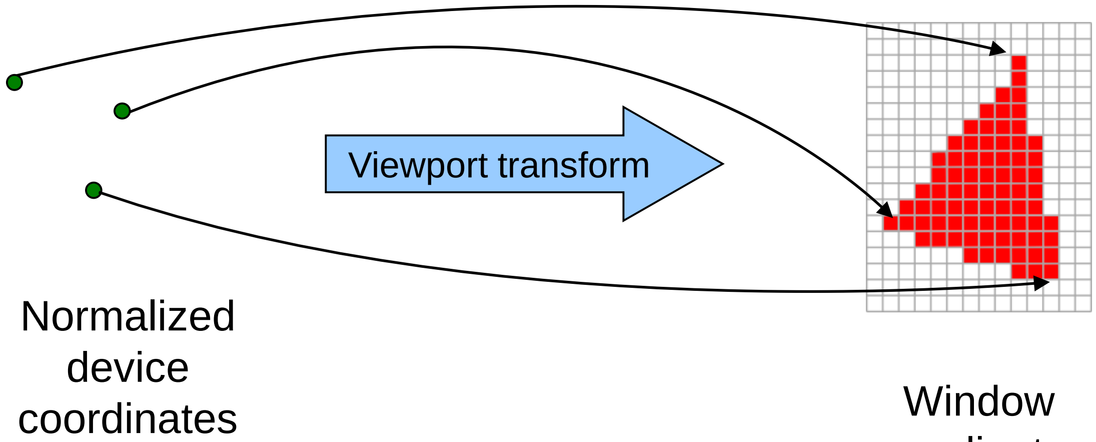
Stages



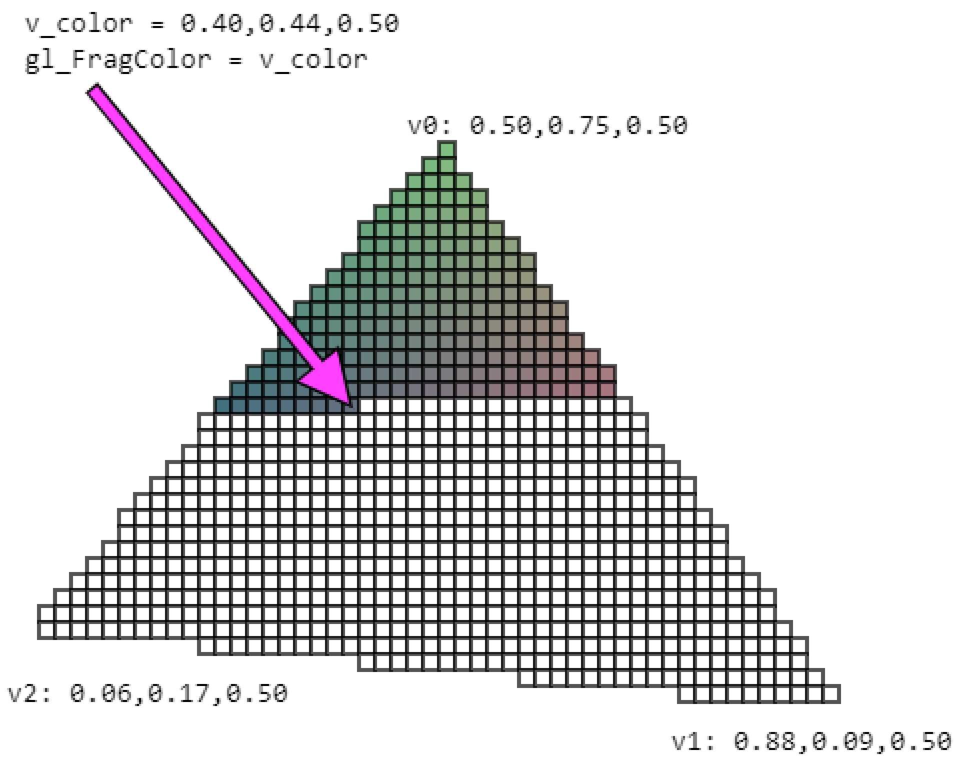
Rasterization

- Convert geometry into a set of fragments
- A fragment is an operation performed on a pixel
- A fragment can modify:
  - Color buffer
  - Depth buffer
  - … any buffer

Viewport tranform



(x,y,z) ➔ (x’,y’,z’)
x in [-1,1], y in [-1,1]
x’ in [0,width], y’ in [0,height]



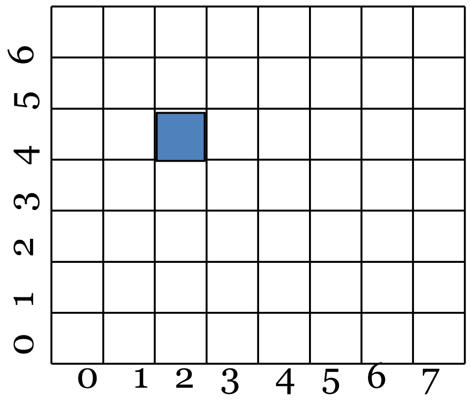
Animated example:
https://webgl2fundamentals.org/webgl/lessons/webgl-how-it-works.html

Rasterization: Point

- Direct approach

```
acció Rasteritzar_punt(punt p)
 setPixel(p.x,p.y)
facció
```

- punt (2,4)



What fragments to generate?

- Bressenham's algorithm:

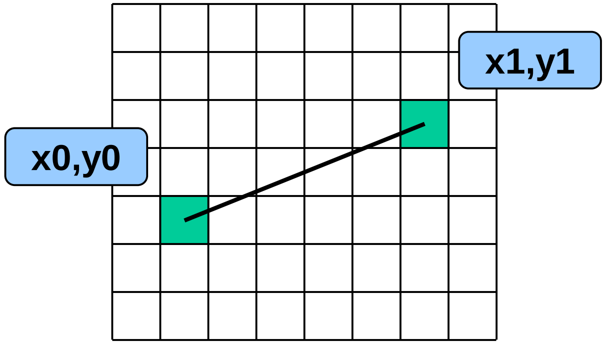
Line : (x0,y0) - (x1,y1)

Goals:

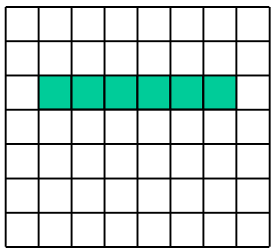
- •Continuous
- •Constant thickness

Trivial cases:

- •Horitzontal
- •Vertical
- •Diagonal (45º)

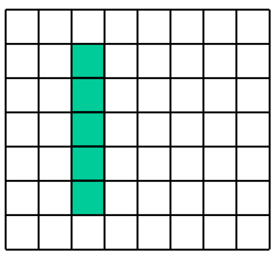


Bressenham's trivial cases



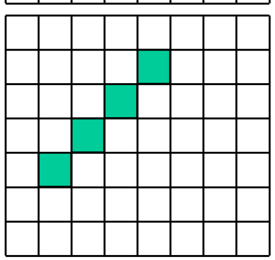
y0 = y1

```
per x=a.in.x fins a.fi.x fer
  setPixel(x,a.in.y)
fiper
```



x0 = x1

```
per y=a.in.y fins a.fi.y fer
  setPixel(a.in.x,y)
fiper
```

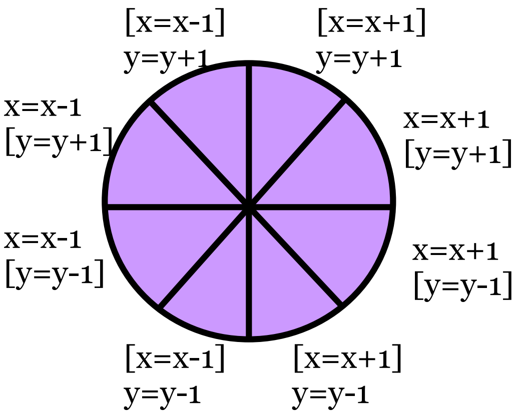


x1-x0 = y1-y0

```
per x=a.in.x fins a.fi.x fer
  setPixel(x,a.in.y+x-a.in.x)
fiper
```

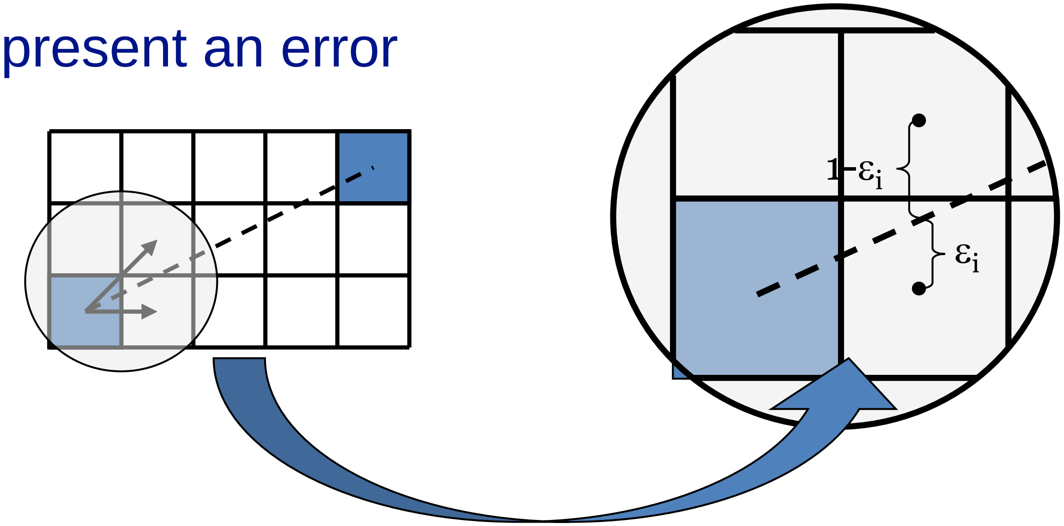
Rasterization: Bresenham Algorithm

- Most used algorithm
- For axis between 0..45º : x=x+1 i [y=y+1]
- Other cases by symetry



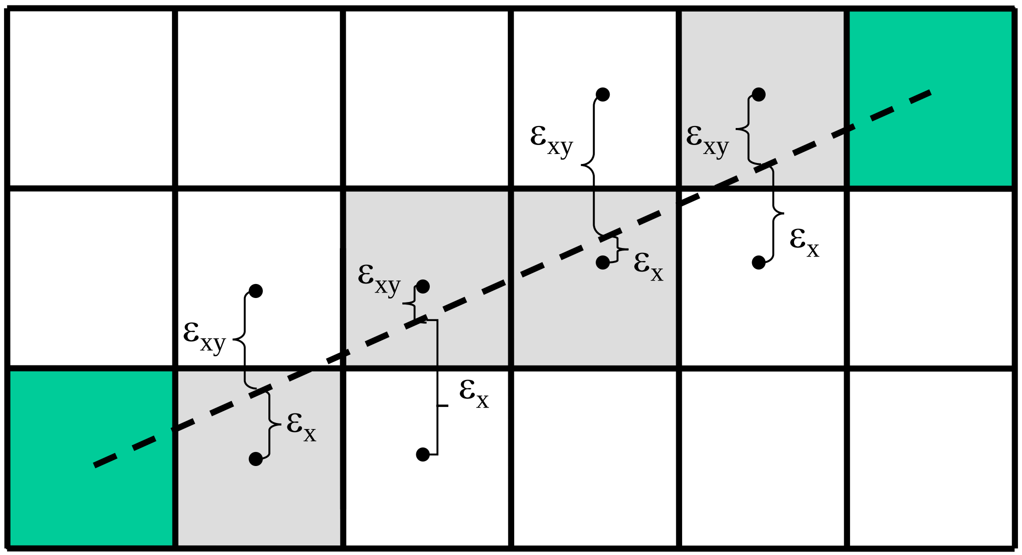
Rasterization: Bresenham Algorithm 0-45º

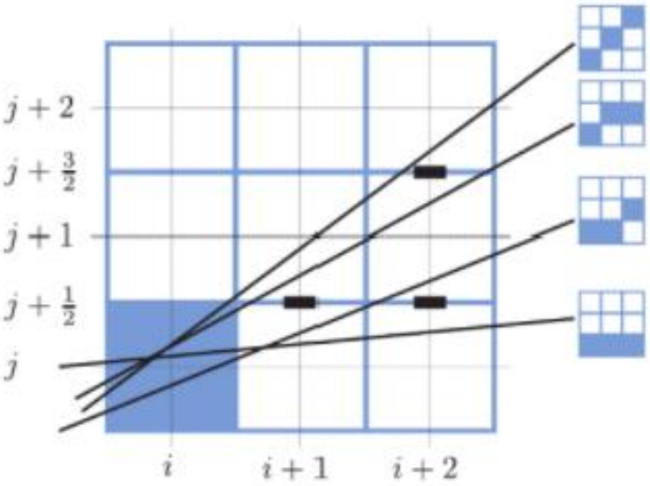
- Each option represent an error
  - Move forward
    - x or (x and y)



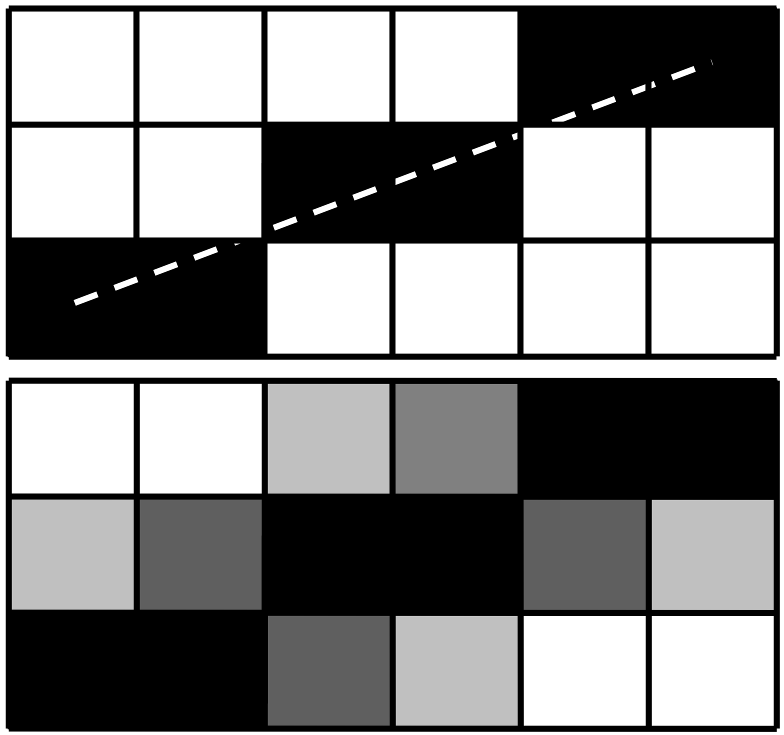
- Minimize error:
  - $\varepsilon_{i+1}=\varepsilon_i$ + slope
  - If $\varepsilon_{i+1}<1/2 \rightarrow x=x+1$
  - If $\varepsilon_{i+1}>1/2 \rightarrow x=x+1;\ y=y+1;\ \varepsilon_{i+1}=1-\varepsilon_{i+1};$

Bressenham's general case



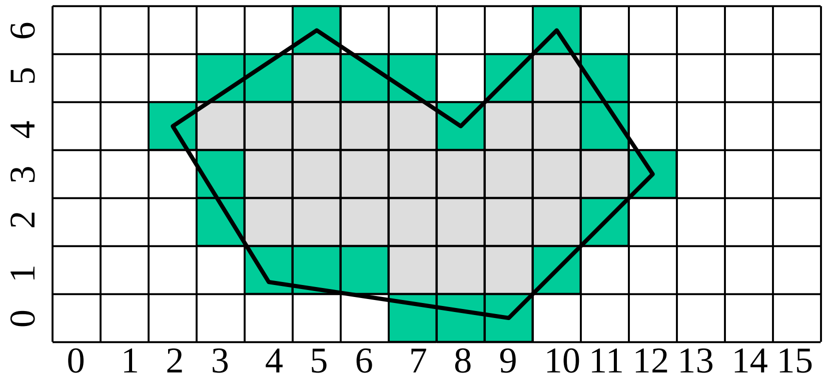


Anti-aliasing



Polygon rasterization

Scan-line Algorithm



- Més utilitzat
- Per tota línia horitzontal → busquem interseccions
  pintem entre interseccions

VISIBILITY

Visibility problem

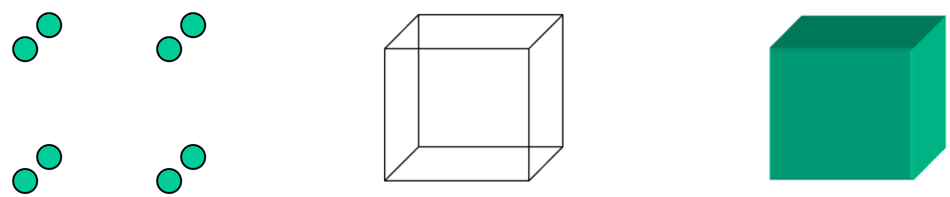
- We should convert simple primitives to pixels/fragments
- How do we know which primitives (or which parts of primitives) should be visible?
  - **Hidden Surface removal problem**

- Solution:
  - Visibility algorithms: one of the bottleneck of the rendering pipeline

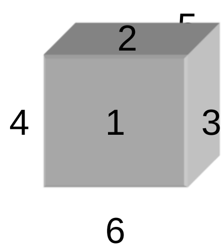
- Overview:
  - Analyze main proposal

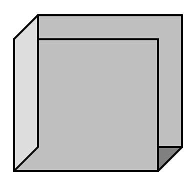
Definition

- Can we paint primitives in any order ?

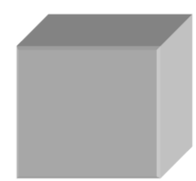


- We have to paint the near ones in front





2,3,5,6,1,4



[5,6,4],2,3,1

Visibility algorithms

- **Object space**: decide order of the primitives

–Visibility is determined in “world” coordinates before conversion to pixels.

–Resolution of the device is irrelevant

  – Painter
  – Binary trees

- **Image space**: which object is visible at each pixel

–Visibility is computed when objects “are converted” onto pixels.

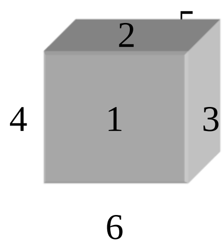
–Resolution of the device fixes the precision of the calculations.

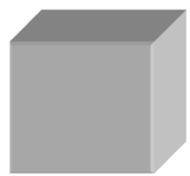
  – Z-buffer
  – Ray casting

Universitat de Girona

Object Space Approaches

- Order faces





•5,6,4,2,3,1

•4,5,6,1,2,3

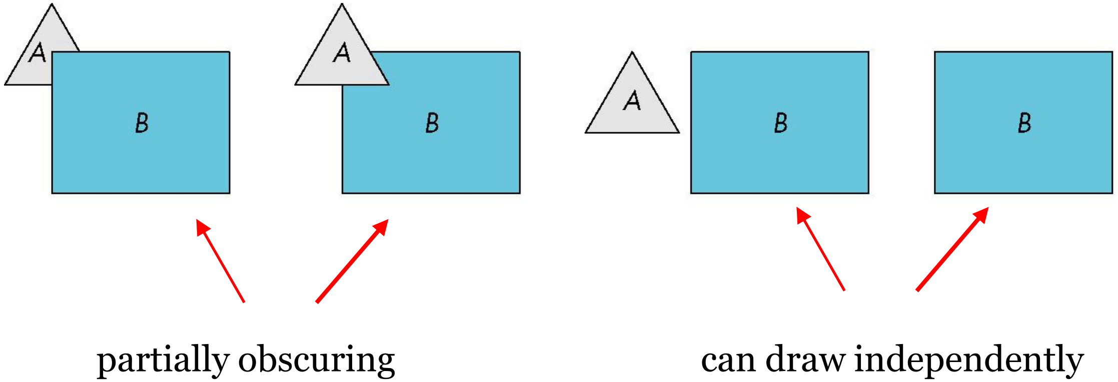
•6,4,5,2,1,3

•...

- Different orders !
- We order in object space
  - •Painter Algorithm
  - •Binary Trees

Hidden Surface Removal

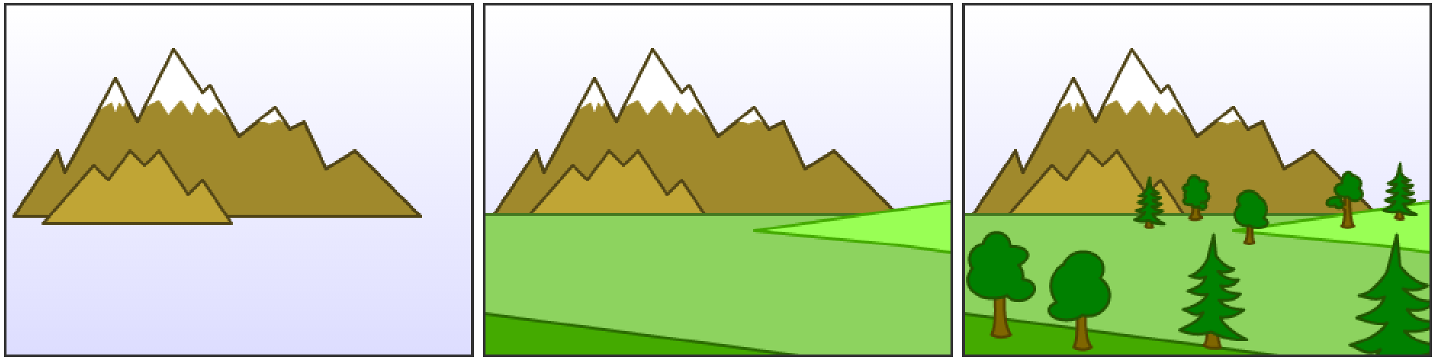
- Object-space approach: use pairwise testing between polygons (objects)



- Worst case complexity $O(n^2)$ for n polygons

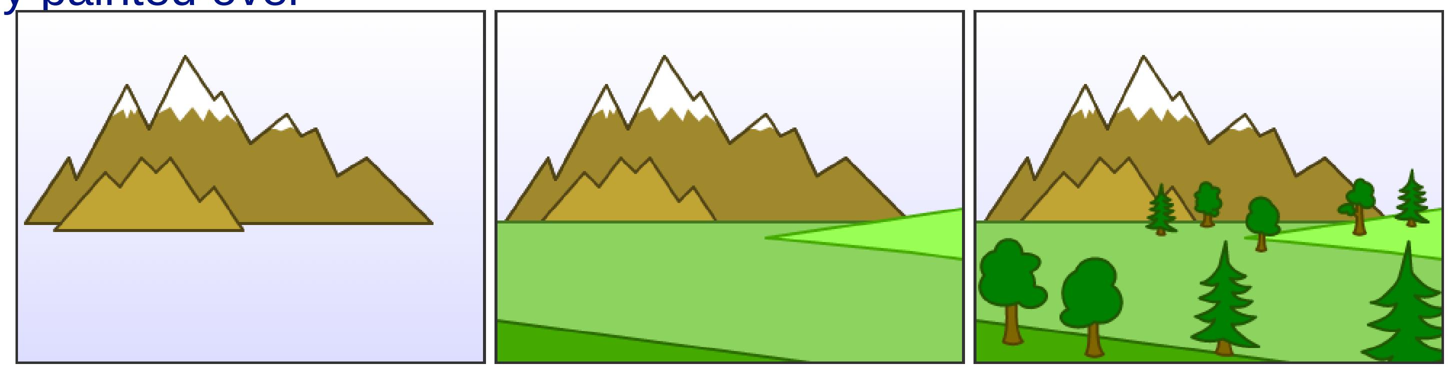
Painter's Algorithm

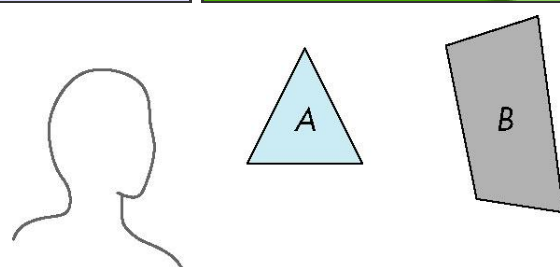
- Render polygons a back to front order so that polygons behind others are simply painted over



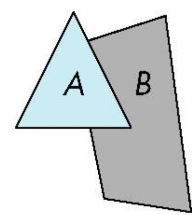
Painter’s Algorithm

•Render polygons a back to front order so that polygons behind others are simply painted over





B behind A as seen by viewer



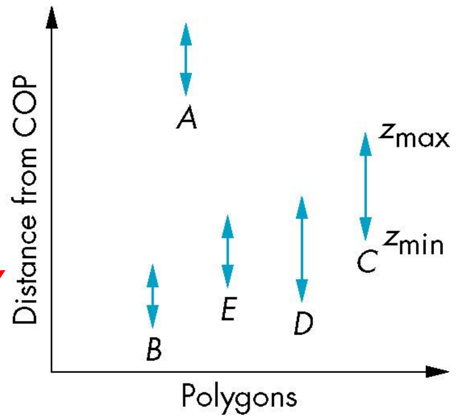
Fill B then A

Universitat de Girona

Depth Sort

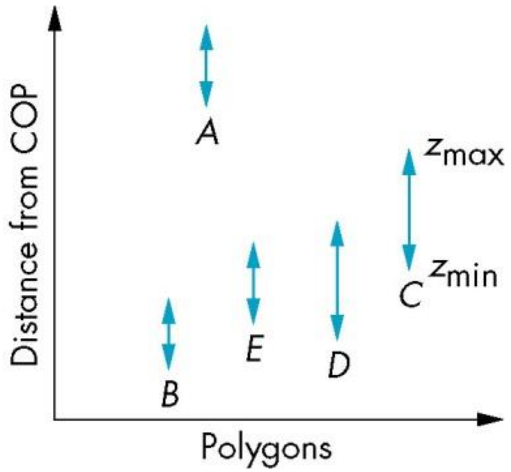
- Requires ordering of polygons first
  - O(n log n) calculation for ordering
  - Not every polygon is either in front or behind all other polygons

- Order polygons and deal with easy cases first, harder later



Polygons sorted by distance from COP

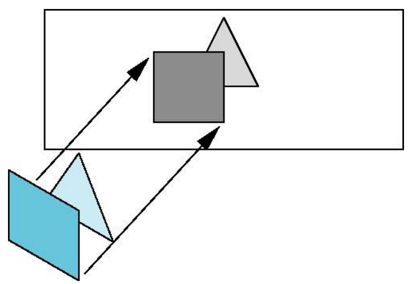
Easy Cases



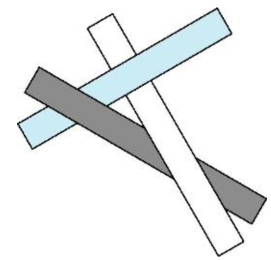
- A lies behind all other polygons
  - Can render

- Polygons overlap in z but not in either x or y
  - Can render independently

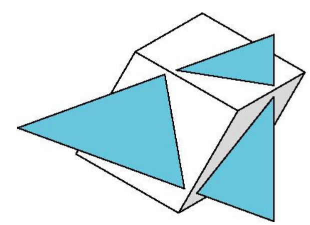
Hard Cases



Overlap in all directions
but one is fully on
one side of the other



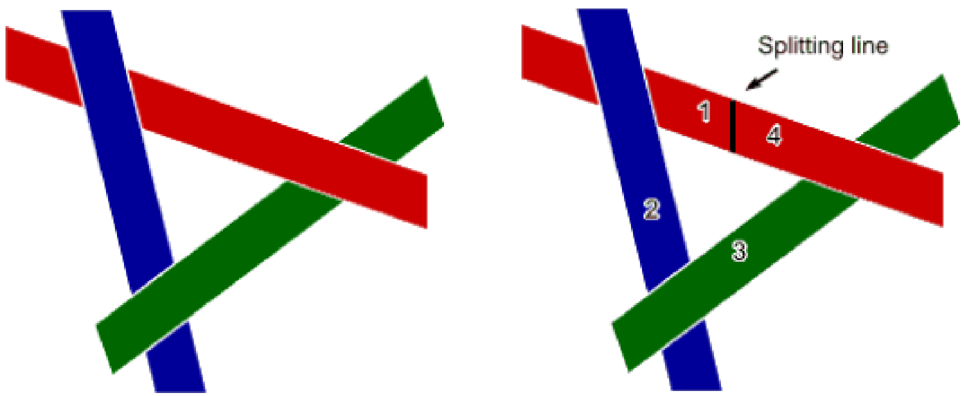
cyclic overlap



penetration

Painter's Algorithm: Cycles

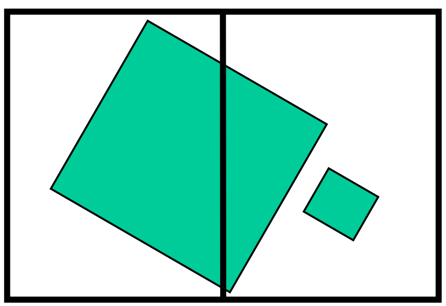
•Which to scan first?

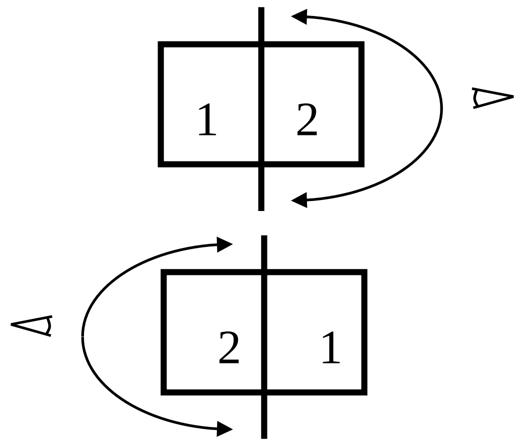


- Split along line, then scan 1,2,3,4 (or split another polygon and scan accordingly)

Binary Space Partition

- Order independent of viewer

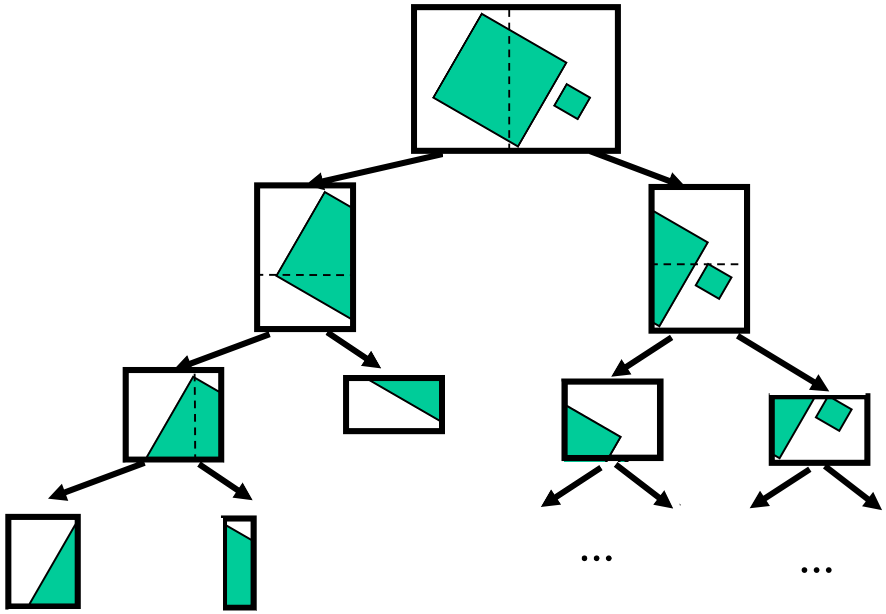




- Binary tree
  - Leaves are single primitives
  - Different subdivision criteria

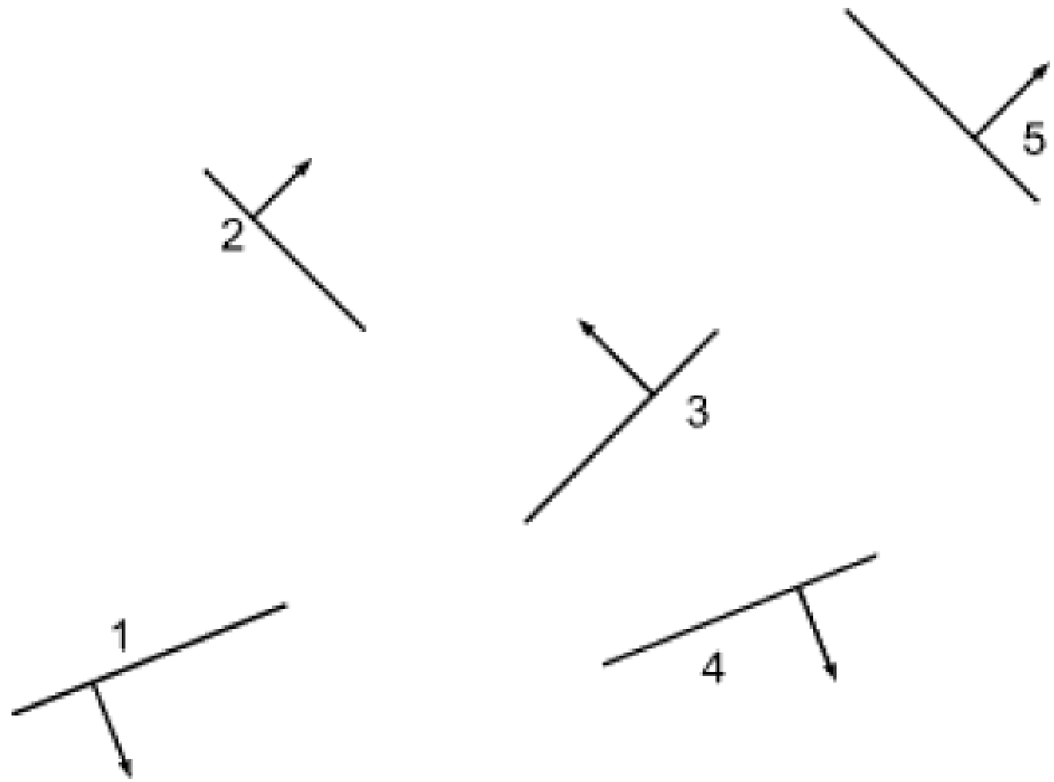
Binary Space Partition

- Example:



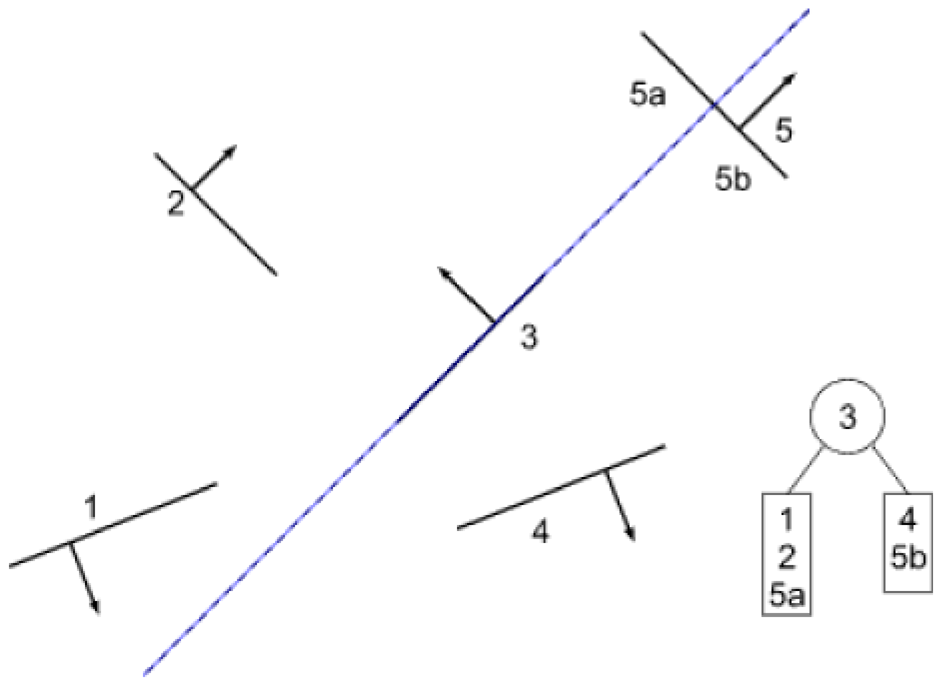
Leave: poligon
Node: cutting plane

Binary Space Partitioning



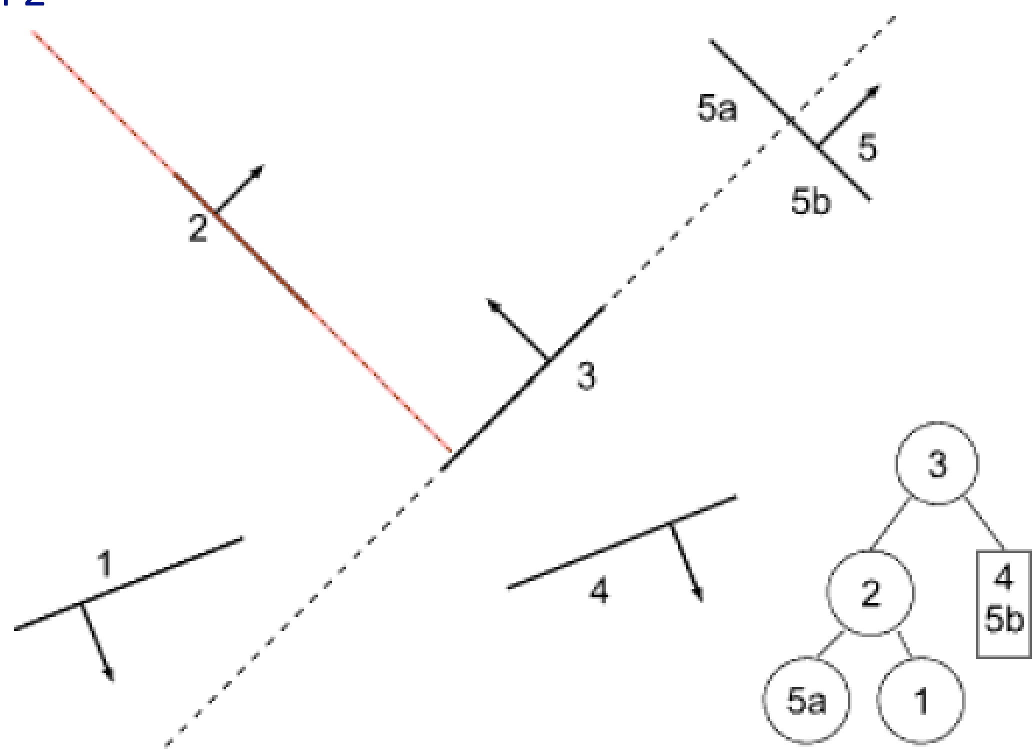
Building a BSP Tree

- Use poligon 3 as root, split on its plane
- Pgon 5 split into 5a and 5b



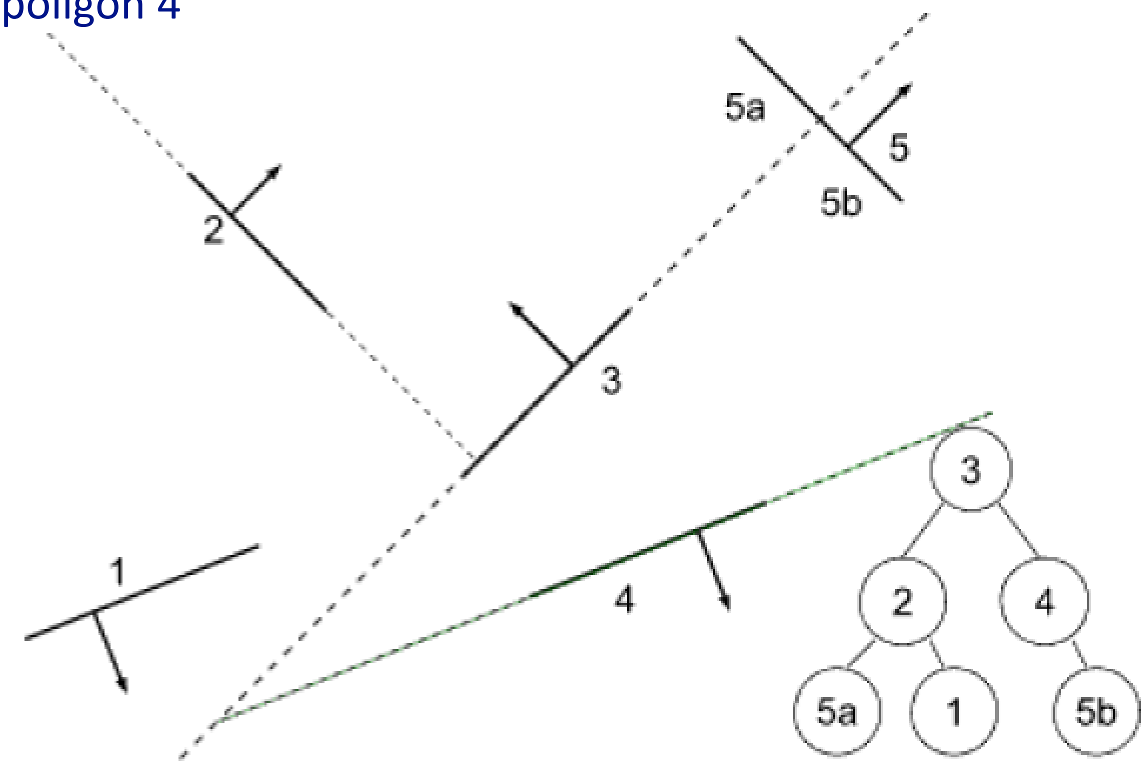
Building a BSP Tree

- Split left subtree at poligon 2



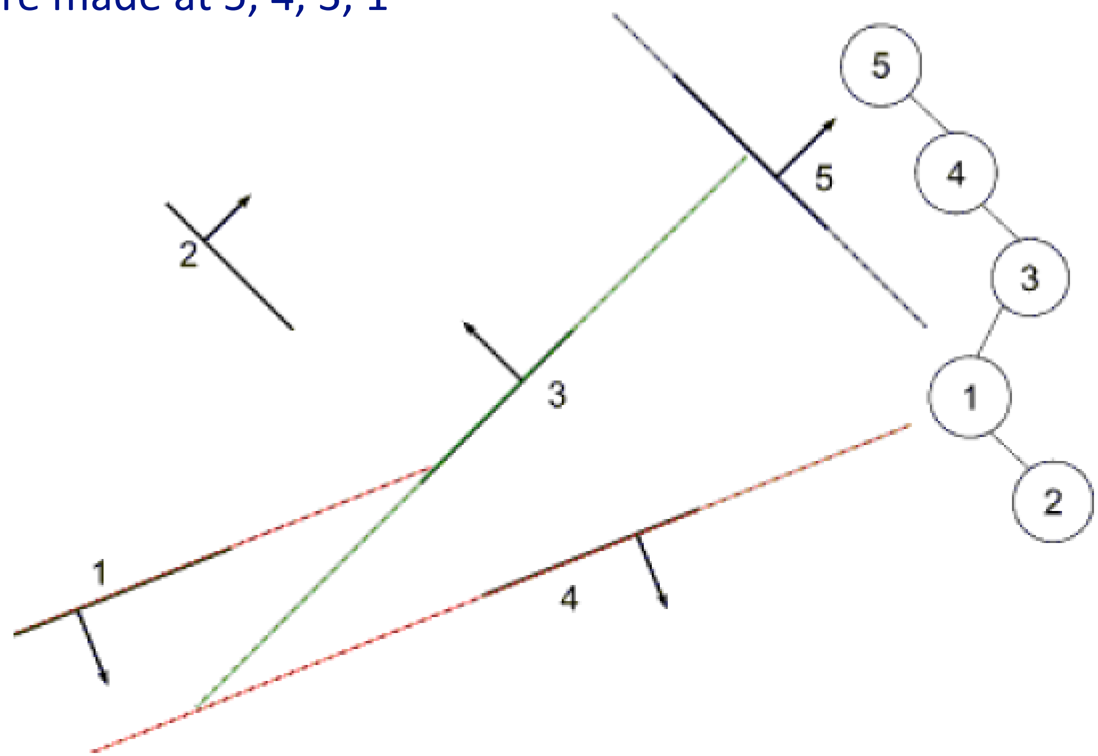
Building a BSP Tree

- Split right subtree at poligon 4



Building a BSP Tree

- Alternate tree if splits are made at 5, 4, 3, 1



Example 2: Traversing the Tree

Traversing:
1. Back
2. Root
3. Front

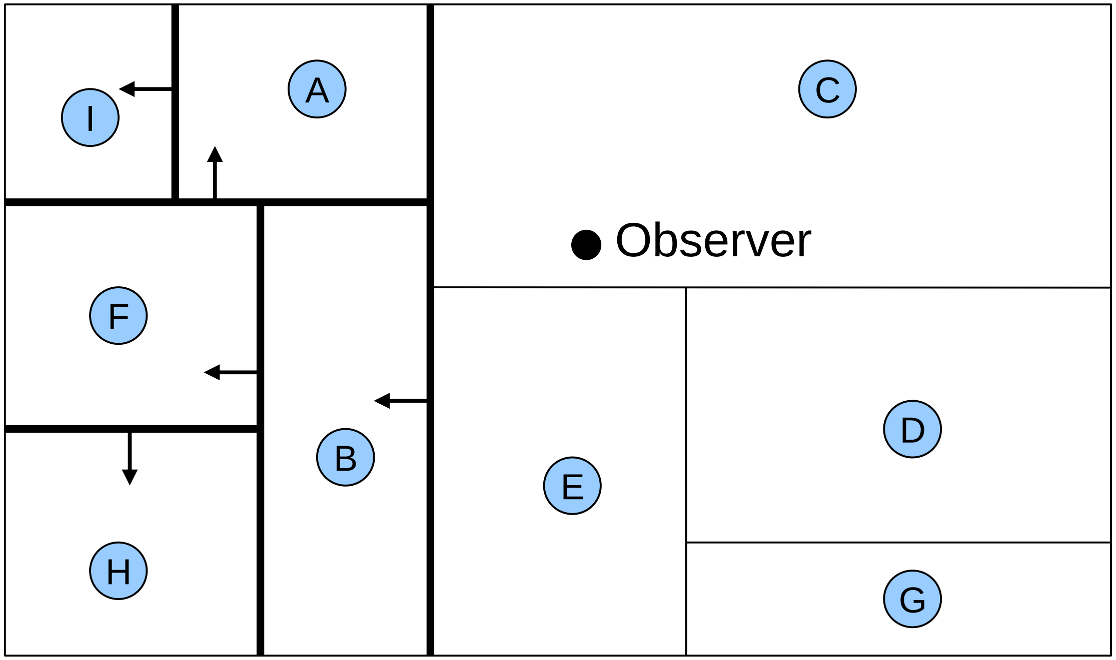
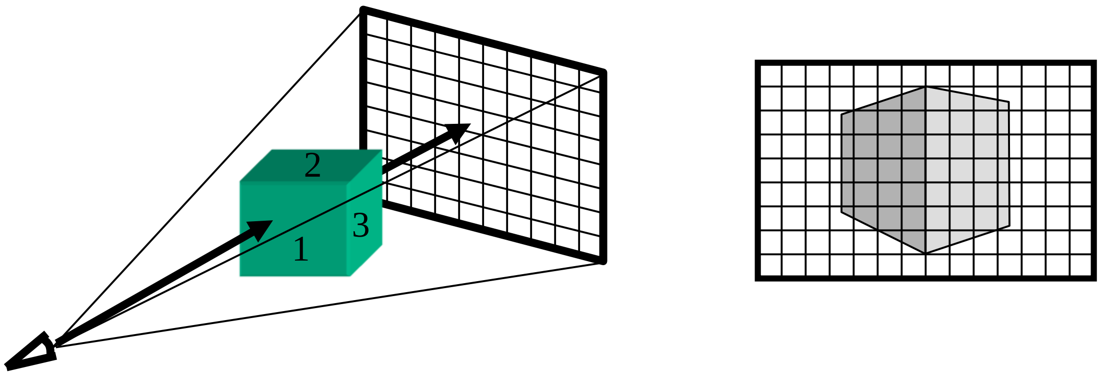


Image Space Approaches

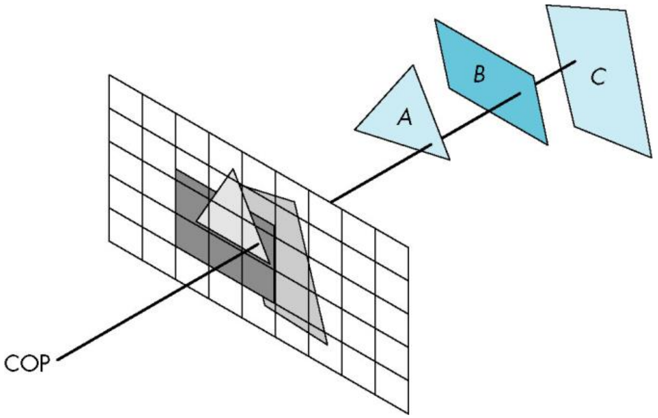
- Determine visibility per pixel



- Algorithms:
  - Ray-casting
  - Z-Buffer

Image Space Approach

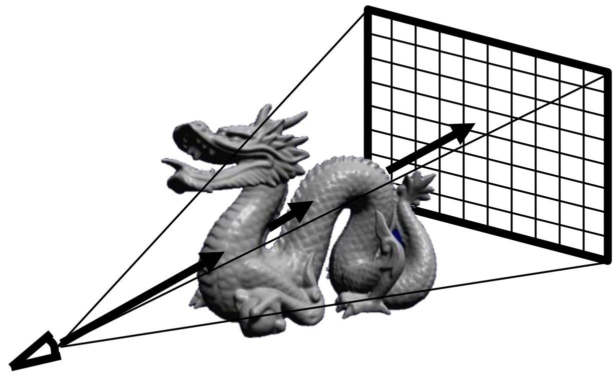
- Look at each projector (nm for an $n \times m$ frame buffer) and find closest of k polygons
- Complexity $O(nmk)$



Universitat de Girona

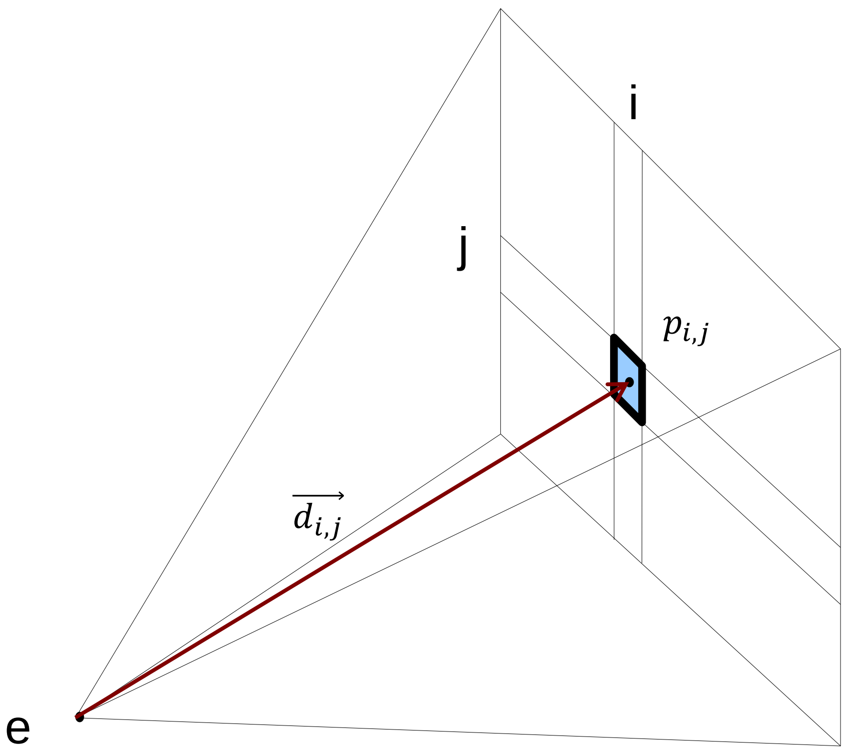
Ray Casting

- Compute intersections observer-pixel-model



- 1 ray per pixel → N intersections
- We get the closest one

Ray

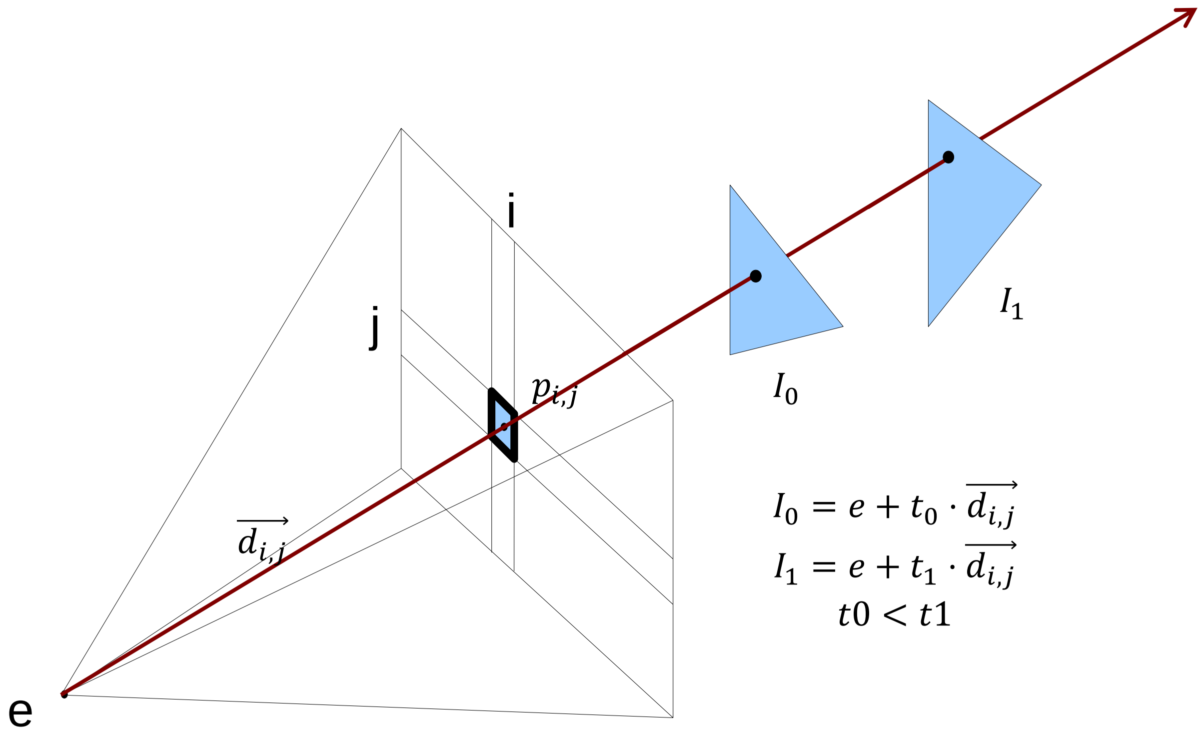


$$p = e + t \cdot \overrightarrow{d_{i,j}}$$

$$\overrightarrow{d_{i,j}} = \frac{p_{i,j} - e}{\| p_{i,j} - e \|}$$

Universitat de Girona

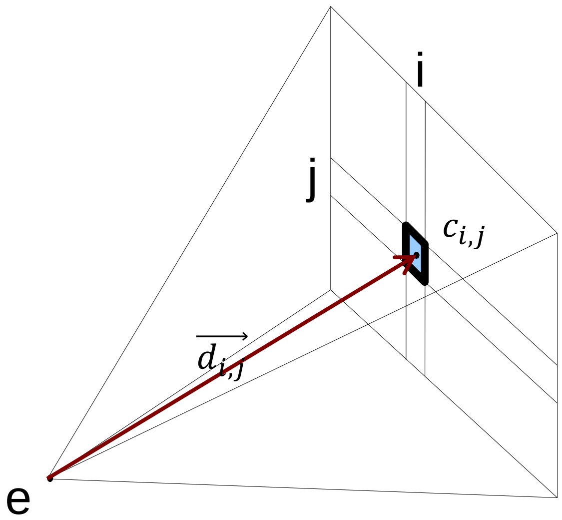
Intersections



$$I_0 = e + t_0 \cdot \overrightarrow{d_{i,j}}$$
$$I_1 = e + t_1 \cdot \overrightarrow{d_{i,j}}$$
$$t0 < t1$$

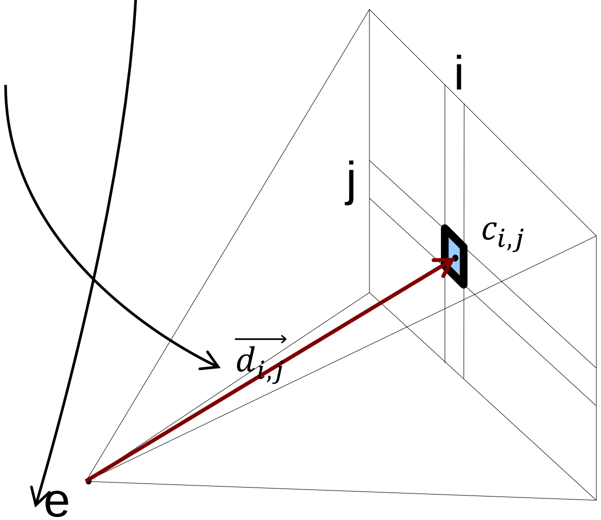
Algorithm

```
action RayCasting(scene, camera)
   for each Pixel px in camera do
      r=defineRay(e,px,camera);
      color=intersectScene(scene,r);
      setPixel(px.i, px.j, color);
   end for
end action
```



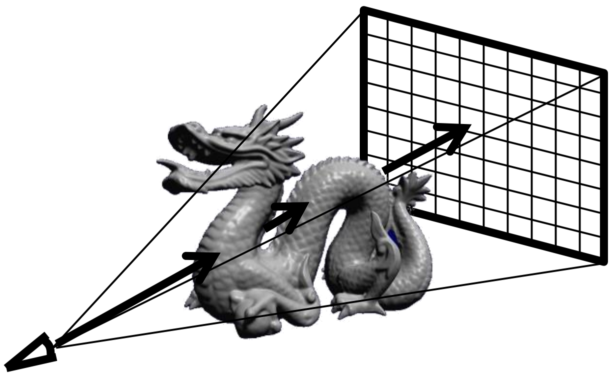
Define ray

```
function defineRay(e,px,camera):Ray
   var r:Ray;
   r.o=e;
   r.d=computeRayDirection(camera,px);
   return r;
end function
```



Scene intersection

```
action RayCasting(scene, camera)
   for each Pixel px in camera do
      r=defineRay(e,px,camera);
      color=intersectScene(scene,r);
      setPixel(px.i, px.j, color);
   end for
end action
```



Algorithm

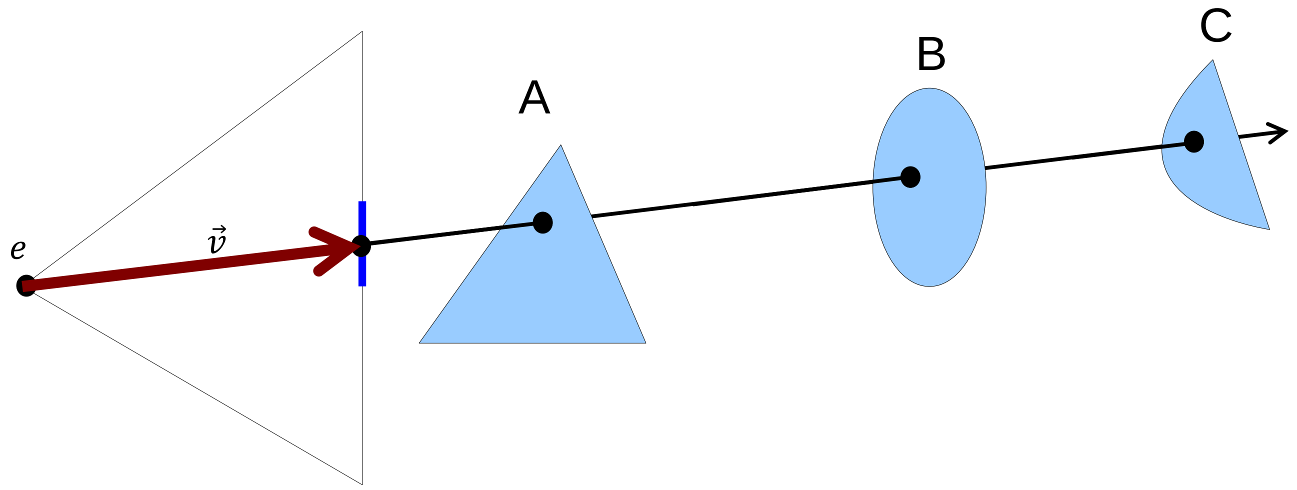
```
function intersectScene(scene,r): Color
  hit=computeFirstHit(scene,r);
  if interaction(hit)
    return computeColor(scene, hit);
  end if
  return BACKGROUND_COLOR;
end function
```

hit stores all the information about the intersection: point, normal, surface id

Algorithm

```
function computeFirstHit(scene,r): Hit
  Hit h;
  for each Primitive p in scene
   Hit h2 = p.intersect(r);
      if h2.t < h.t
          h = h2
      end if
  end for
  return h
end function
```

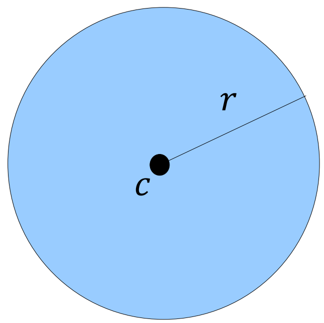
Example



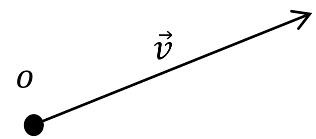
Types of primitives

- Any that can be intersected with a ray:
  - Any polygon
  - Cone
  - Sphere
  - Cilindre
  - Splines
  - NURBS
  - Subdivision surfaces
  - ...

Ray-Sphere intersection



$$\| p - c \| = r$$



$$p = o + t \cdot \vec{v}$$
$$\| \vec{v} \| = 1$$

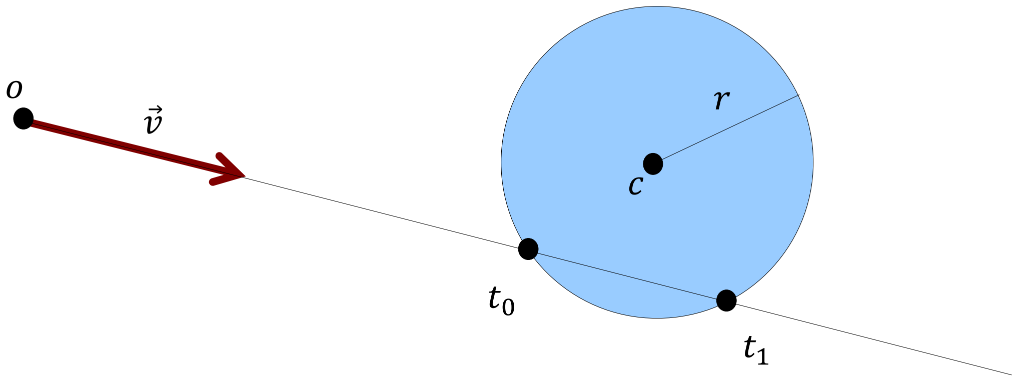
$$\| o + t \cdot \vec{v} - c \| = r \Rightarrow \| o + t \cdot \vec{v} - c \|^2 = r^2$$
$$\Rightarrow (o_x + t \cdot v_x - c_x)^2 + \left(o_y + t \cdot v_y - c_y\right)^2 + (o_z + t \cdot v_z - c_z)^2 = r^2$$
$$\Rightarrow t^2 \cdot \left(v_x^2 + v_y^2 + v_z^2\right)$$
$$+t \cdot \left(2 \cdot (o_x - c_x) \cdot v_x + 2 \cdot \left(o_y - c_y\right) \cdot v_y + 2 \cdot (o_z - c_z) \cdot v_z\right)$$
$$+(o_x - c_x)^2 + \left(o_y - c_y\right)^2 + (o_z - c_z)^2 = r^2$$

Universitat de Girona

Ray-Sphere intersection

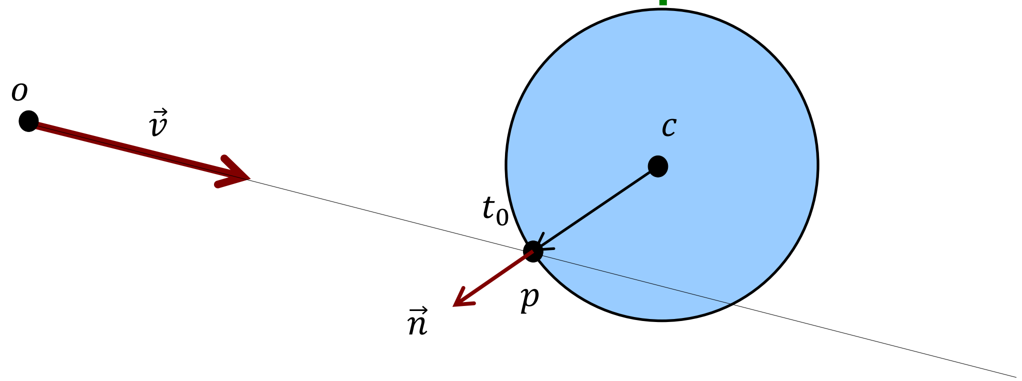
- Second degree equation:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \Rightarrow \begin{cases} a = 1 \\ b = 2(o - c) \cdot \vec{v} \\ c = (o - c) \cdot (o - c) - r^2 \end{cases}$$



Ray-Sphere intersection

- Normal at intersection point:



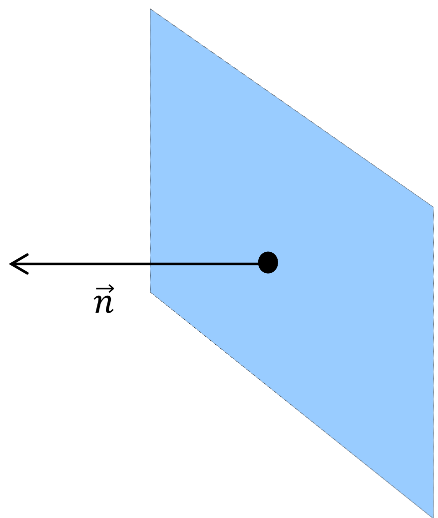
$$p = o + t_0 \cdot \vec{v}$$
$$\vec{n} = \frac{p - c}{\parallel p - c \parallel} \Rightarrow \frac{p - c}{r}$$

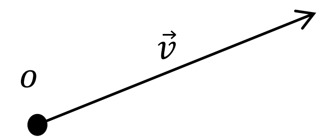
Ray-plane intersection

$$Ax + By + Cz + D = 0$$
$$\vec{n} = (A, B, C)$$

$$\vec{n} \cdot P + D = 0$$

Però també:
$$\vec{n} \cdot (P - P0) = 0$$
On P0 és un punt del pla



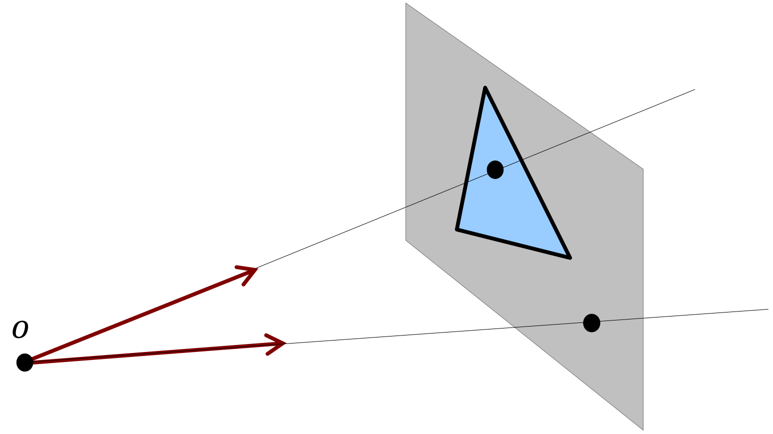


$$p = o + t \cdot \vec{v}$$
$$\| \vec{v} \| = 1$$

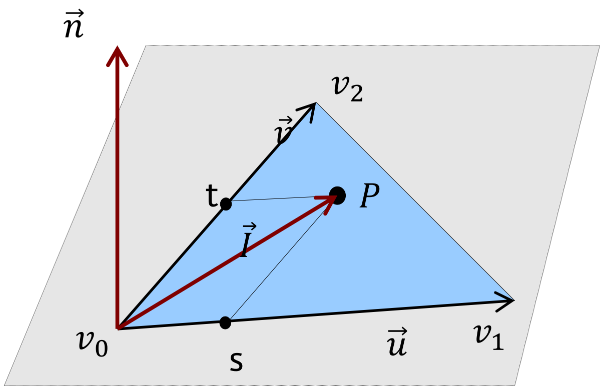
$$Ax + By + Cz + D = 0 \Rightarrow \vec{n} \cdot P + D = 0 \Rightarrow \vec{n} \cdot (o + t \cdot \vec{v}) + D = 0$$
$$\Rightarrow t = \frac{-D - \vec{n} \cdot o}{\vec{n} \cdot \vec{v}}$$

Ray-triangle intersection

- First compute intersection with triangle plane
- Then check if intersection point is inside the triangle



Ray-triangle intersection



$$\vec{u} = v_1 - v_0$$
$$\vec{v} = v_2 - v_0$$
$$P = v_0 + \mathrm{s} \cdot \vec{u} + \mathrm{t} \cdot \vec{v}$$

Work with v_0 as origin:

$$\vec{I} = P - v_0 \Rightarrow \vec{I} = \mathrm{s} \cdot \vec{u} + \mathrm{t} \cdot \vec{v}$$

$$\vec{I} = \mathrm{s} \cdot \vec{u} + \mathrm{t} \cdot \vec{v} \Rightarrow \begin{cases} I_x = \mathrm{s}u_x + tv_x \\ I_y = \mathrm{s}u_y + tv_y \\ I_z = \mathrm{s}u_z + tv_z \end{cases}$$

$$s = \frac{(\vec{u} \cdot \vec{v})(\vec{I} \cdot \vec{v}) - (\vec{v} \cdot \vec{v})(\vec{I} \cdot \vec{u})}{(\vec{u} \cdot \vec{v})^2 - (\vec{u} \cdot \vec{u})(\vec{v} \cdot \vec{v})}$$

$$t = \frac{(\vec{u} \cdot \vec{v})(\vec{I} \cdot \vec{u}) - (\vec{u} \cdot \vec{u})(\vec{I} \cdot \vec{v})}{(\vec{u} \cdot \vec{v})^2 - (\vec{u} \cdot \vec{u})(\vec{v} \cdot \vec{v})}$$

Intersection if 0 <= s+t <= 1

Universitat de Girona

Ray-triangle intersection precomputations

$$s = \frac{(\vec{u}\cdot\vec{v})(\vec{I}\cdot\vec{v}) - (\vec{v}\cdot\vec{v})(\vec{I}\cdot\vec{u})}{(\vec{u}\cdot\vec{v})^2 - (\vec{u}\cdot\vec{u})(\vec{v}\cdot\vec{v})}$$

$$t = \frac{(\vec{u}\cdot\vec{v})(\vec{I}\cdot\vec{u}) - (\vec{u}\cdot\vec{u})(\vec{I}\cdot\vec{v})}{(\vec{u}\cdot\vec{v})^2 - (\vec{u}\cdot\vec{u})(\vec{v}\cdot\vec{v})}$$

$$s = \frac{UV(\vec{I}\cdot\vec{v}) - VV(\vec{I}\cdot\vec{u})}{UV2 - UU\cdot VV} \qquad t = \frac{UV(\vec{I}\cdot\vec{u}) - UU(\vec{I}\cdot\vec{v})}{UV2 - UU\cdot VV}$$

$$\Rightarrow$$

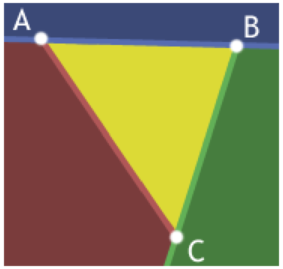
$$s = \frac{UV(\vec{I}\cdot\vec{v}) - VV(\vec{I}\cdot\vec{u})}{D} \qquad t = \frac{UV(\vec{I}\cdot\vec{u}) - UU(\vec{I}\cdot\vec{v})}{D}$$

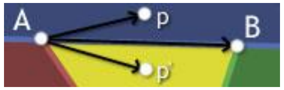
Only four floats precomputed: UV, VV, UU, D

Another method

- Check if the point lays in the same side against the segment of the triangle
  - Use crossProduct
  - From https://blackpawn.com/texts/pointinpoly/default.html

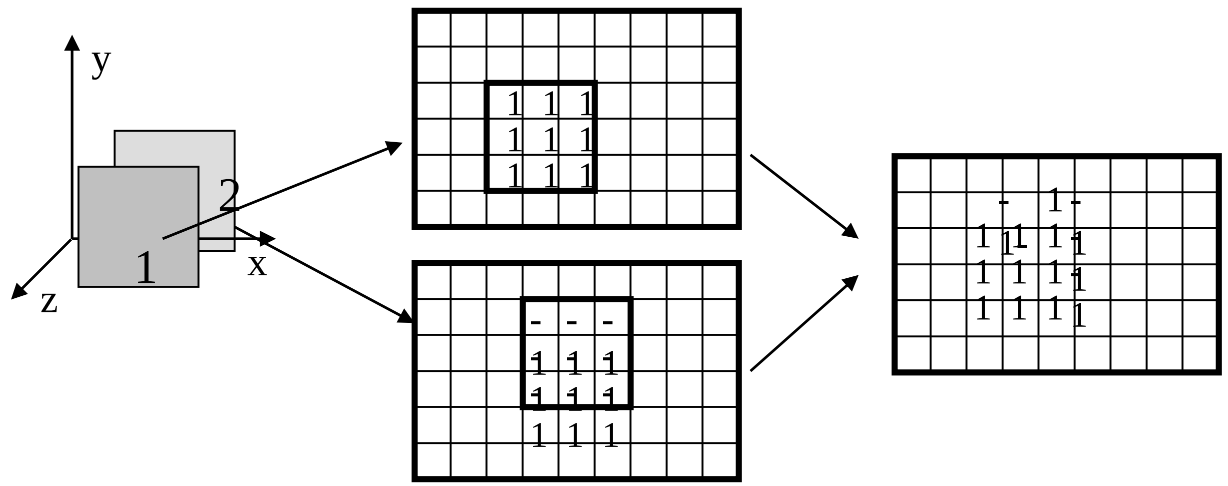
```
function SameSide(p1,p2, a,b)
    cp1 = CrossProduct(b-a, p1-a)
    cp2 = CrossProduct(b-a, p2-a)
    if DotProduct(cp1, cp2) >= 0 then return true
    else return false

function PointInTriangle(p, a,b,c)
    if SameSide(p,a, b,c) and SameSide(p,b, a,c)
        and SameSide(p,c, a,b) then return true
    else return false
```





Z-Buffer

- Rasterization with depth (normalized device coordinates)



- We render the closest one: the one with the lower z

Z-Buffer

•Where to Store depth?

•In a depth buffer (Z-Buffer)

```
z_buffer=taula [1..W][1..H] de reals
```

Ex: 1024 x 768 x 4 bytes = 3M2

Universitat de Girona

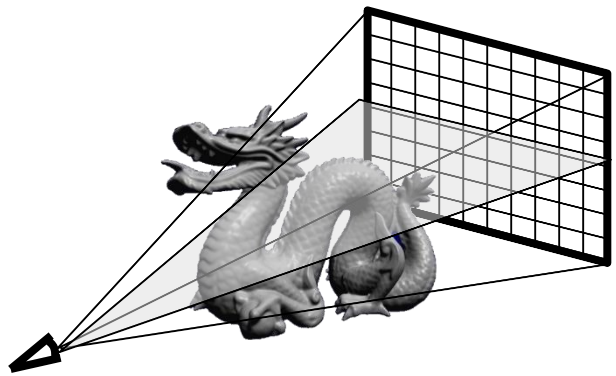
Z-Buffer

- Algorithm

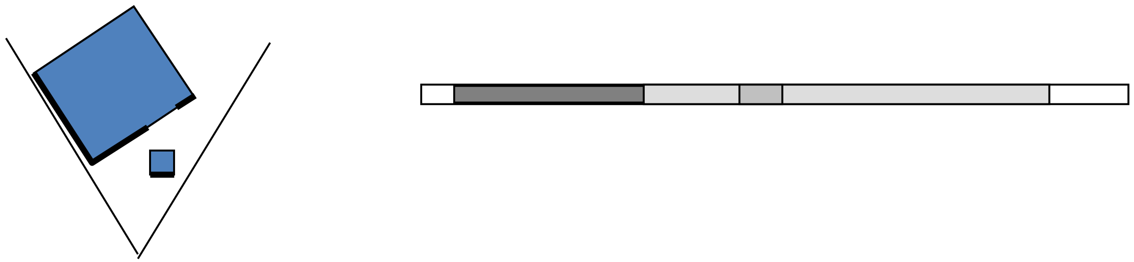
```
InicialitzaBuffer(color,profunditat);
per cada cara c del model fer
 per cada pixel p de c fer
  z=CàlculProf(p,c);
  si profunditat[p.x,p.y]<z
  llavors SetPixel(p.x,p.y);
          profunditat[p.x,p.y]=z;
  fsi;
 fiper
fiper
fialgorisme
```

Scan-Line

•Interseccions observador - línia de píxels – model



•Per cada línia → 1 pla → n segments intersecció



Universitat de Girona

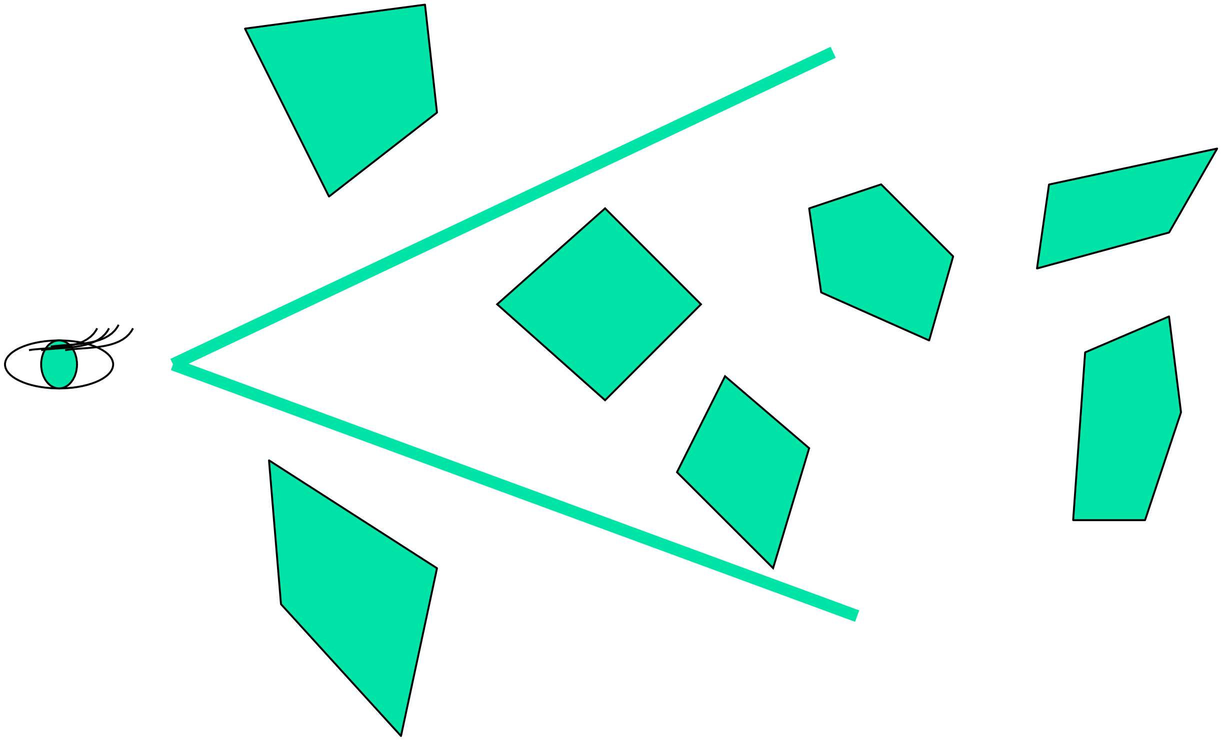
VISIBILITY OPTIMIZATION

Visibility culling

- What is it for?
  - Avoid processing polygons which do not contribute to the rendered image
- We have three different cases of non-visible objects:
  - those outside the view volume
    (***view frustum/volume culling***)
  - those which are facing away from the user
    (***back face culling***)
  - those occluded behind other visible objects
    (***occlusion culling***)

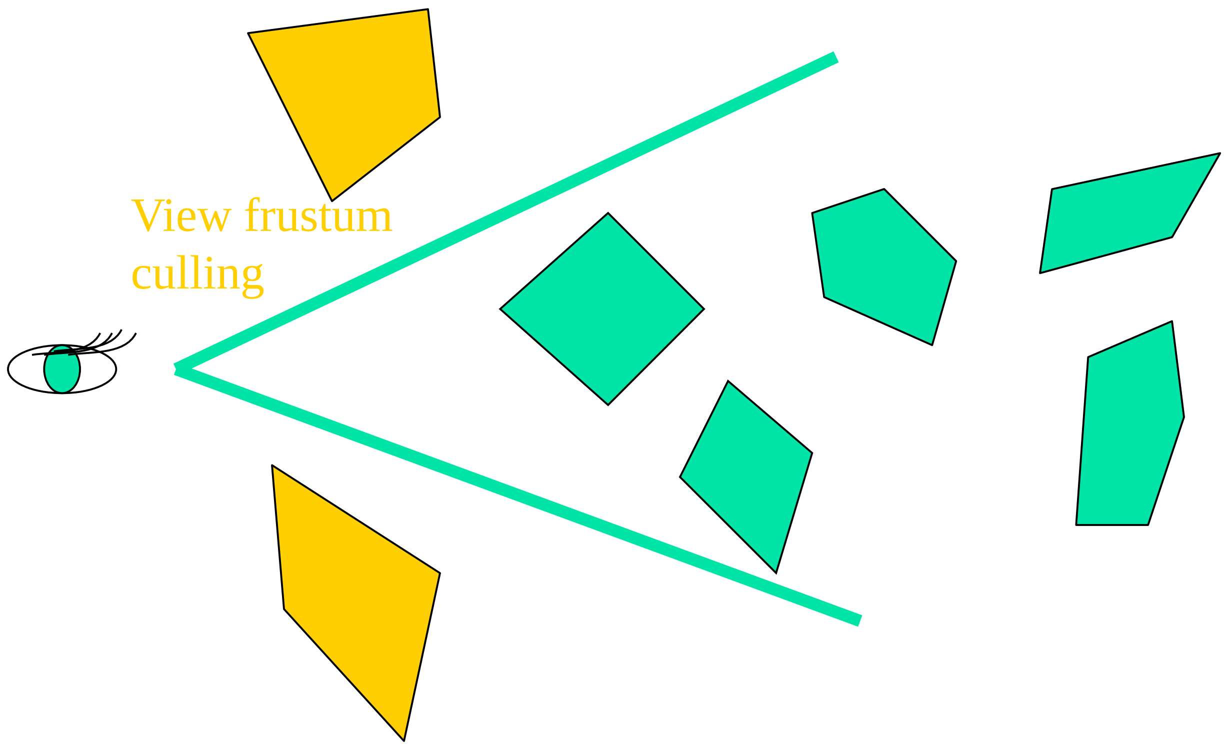
Universitat de Girona

Visibility culling

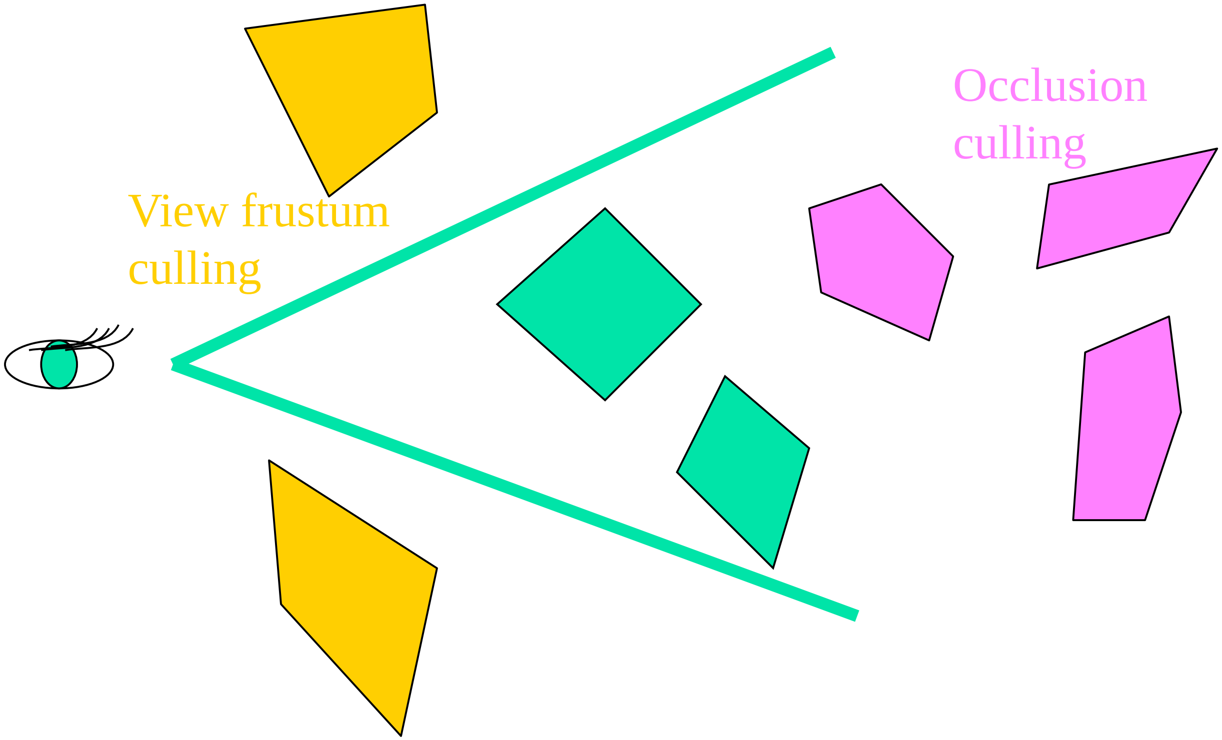


Universitat de Girona

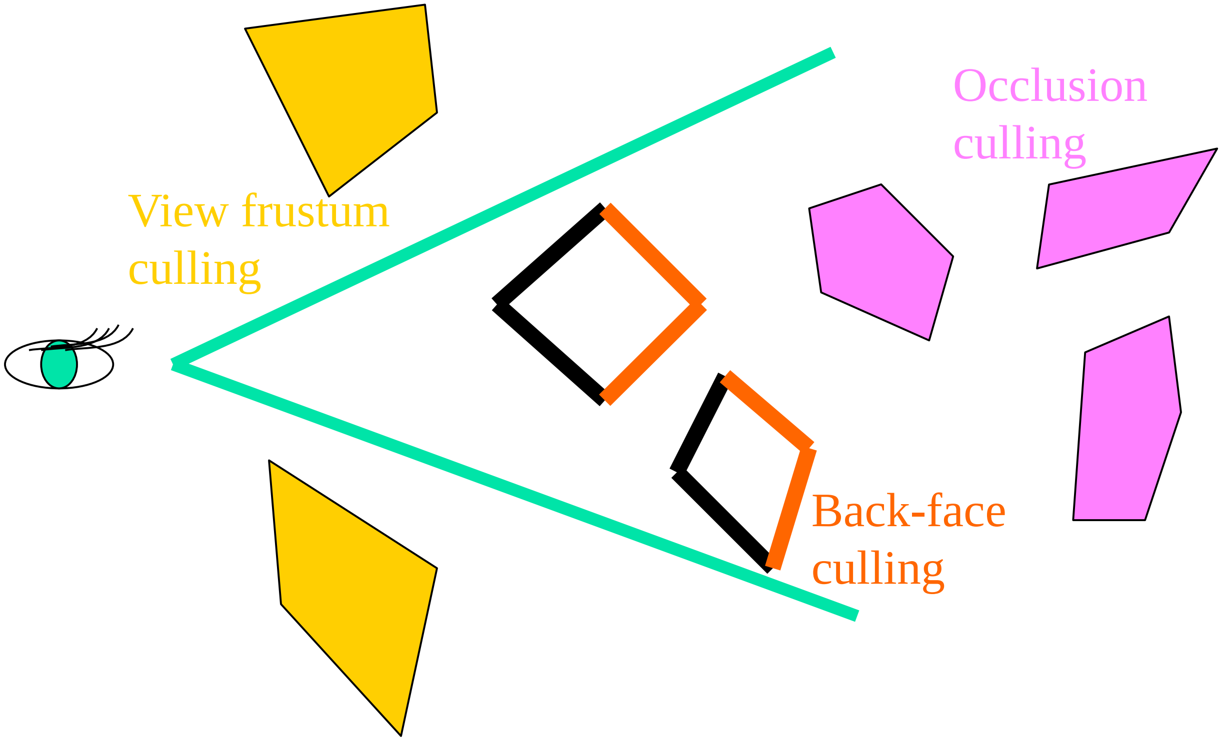
Visibility culling



Visibility culling

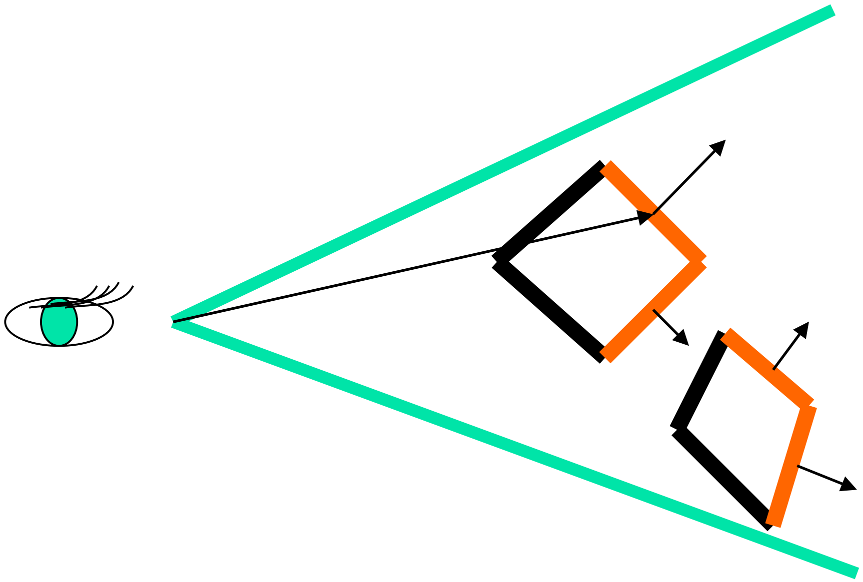


Visibility culling



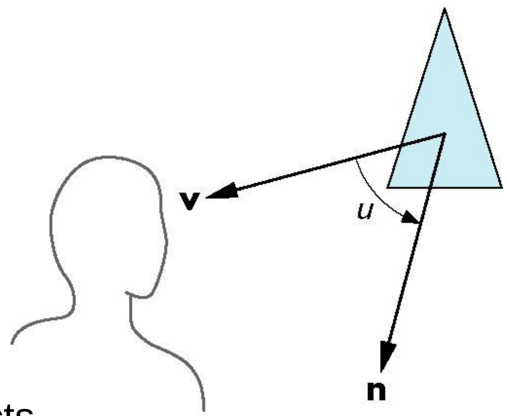
Back-face culling

- Simplest version is to do it per polygon
  - Just test the normal of each polygon against the camera-to-polygon vector (dot product)



Back-Face Culling

- face is visible iff $90 \geq \theta \geq -90$
  - equivalently $\cos\theta \geq 0$
  - or $\mathbf{v} \cdot \mathbf{n} \geq 0$
- plane of face has form
  - $ax + by + cz + d = 0$
- In WebGL we can simply enable culling
  - but may not work correctly if we have nonconvex objects

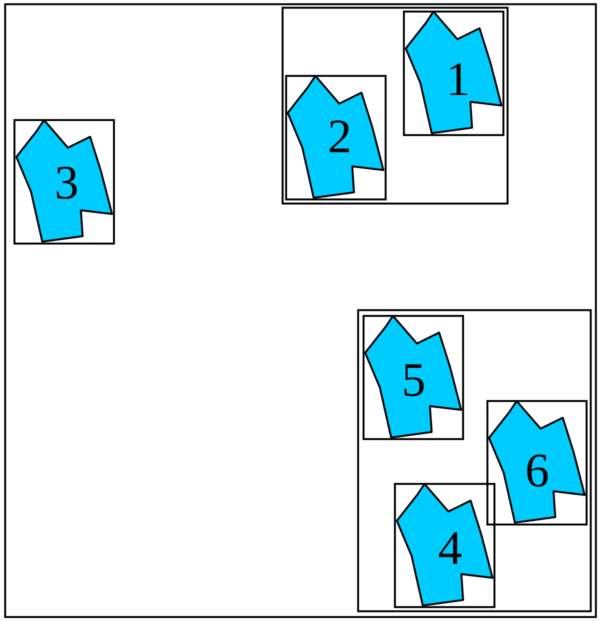


WebGL

- For z-buffer use a depth buffer:
  - `gl.enable(gl.DEPTH_TEST);`
- Also clear the color buffer together
  - `g.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);`

- For culling:
  - `gl.enable(gl.CULL_FACE);`
- Then select face
  - `gl.cullFace(face);`
  - Where face could be `gl.BACK` or `gl.FRONT`

Hierarchical representation of scenes: bounding volumes

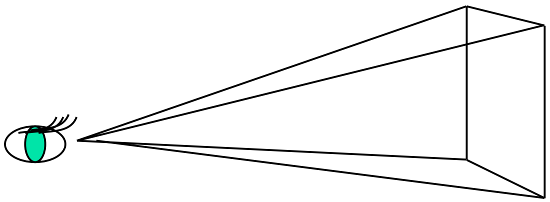


Universitat de Girona

Space partitioning

View frustum culling

- Purpose: cull the polygons that are not inside the pyramid defined by
  - The viewpoint
  - The view direction
  - The two angles defining the field of view
- Easiest way
  - Test bounding box of object against the view volume (planes)



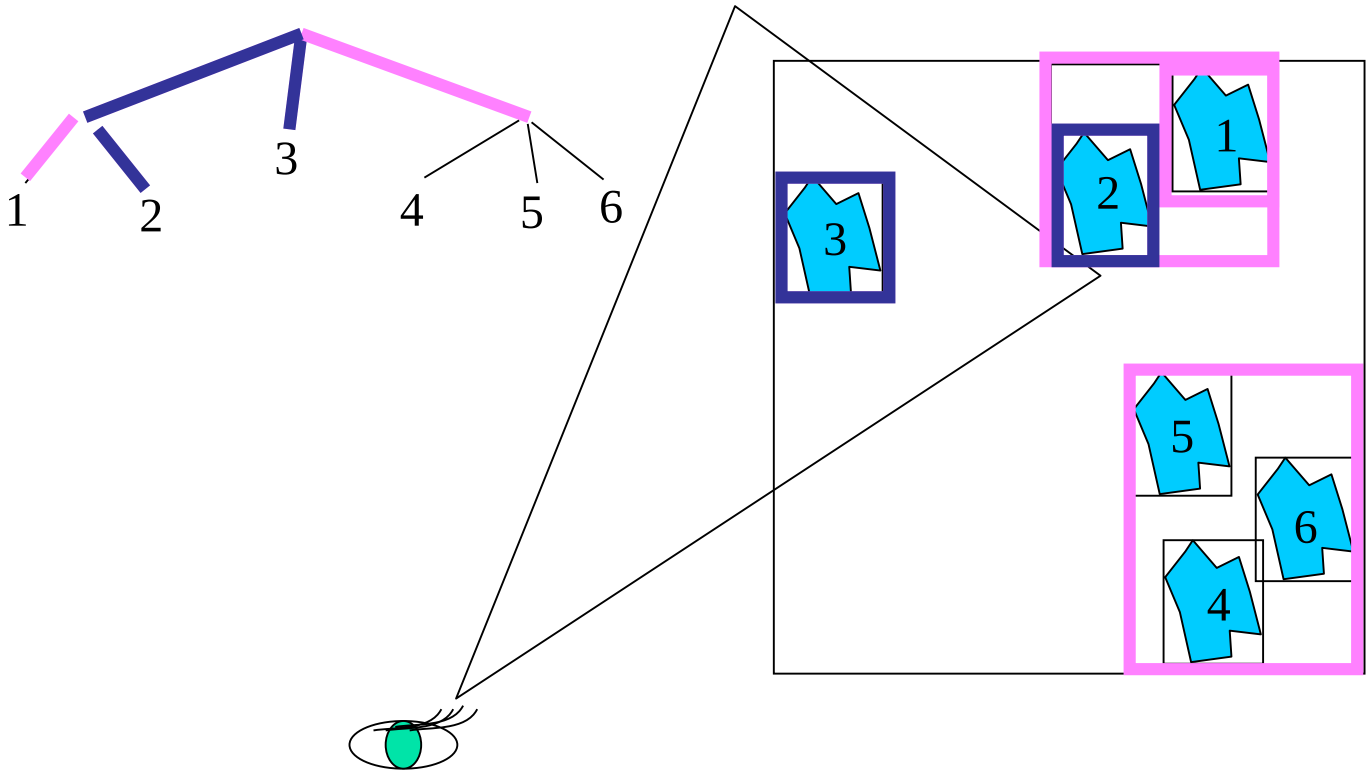
View frustum culling - algorithm

- Compare the scene hierarchically against the view volume:
  - Test the root node against the view volume
  - If node is outside then stop and discard everything below it
  - If node is fully inside then render without clipping
  - Otherwise,

•If leaf node render it,

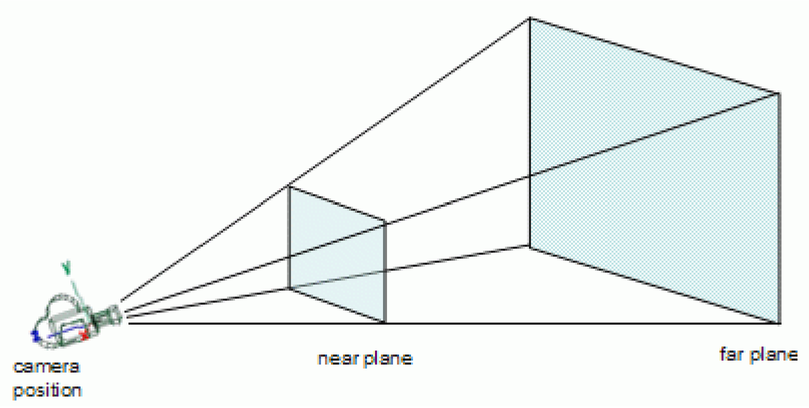
•Else recursively test each of its children

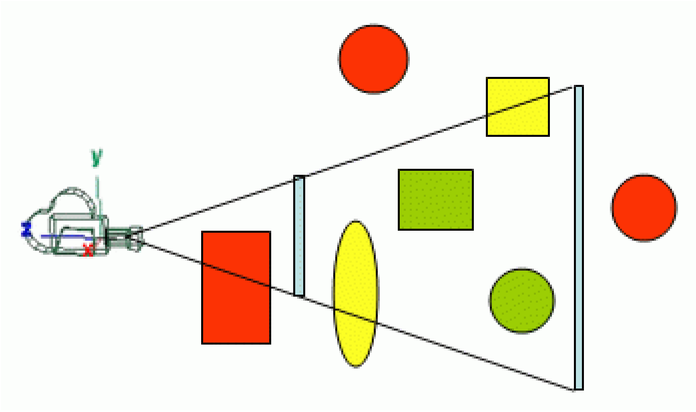
Example



View frustum culling

- Easy to implement
- A very fast computation
- Very effective result
- Therefore it is included in almost all current rendering systems





Next

- Color
- Shading
- Illumination
- Textures