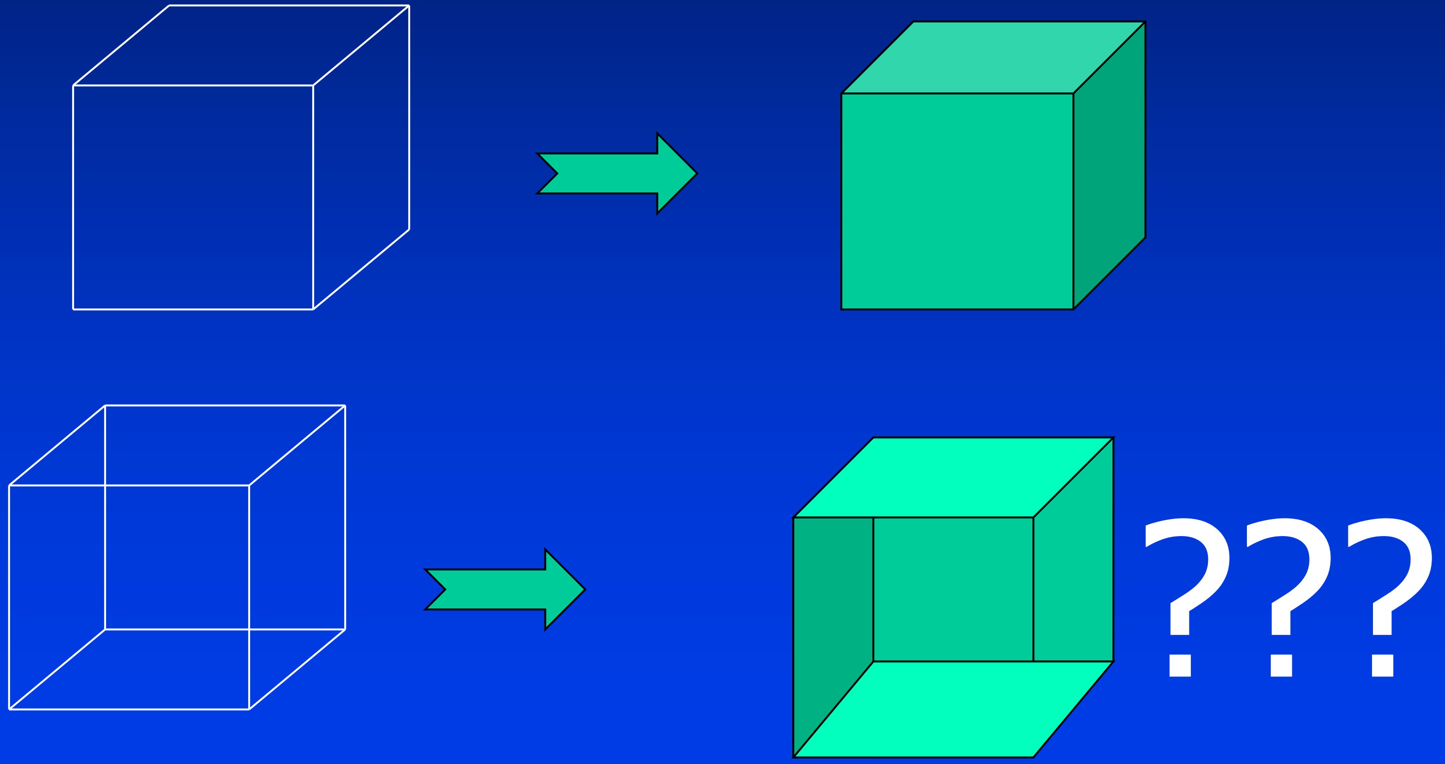
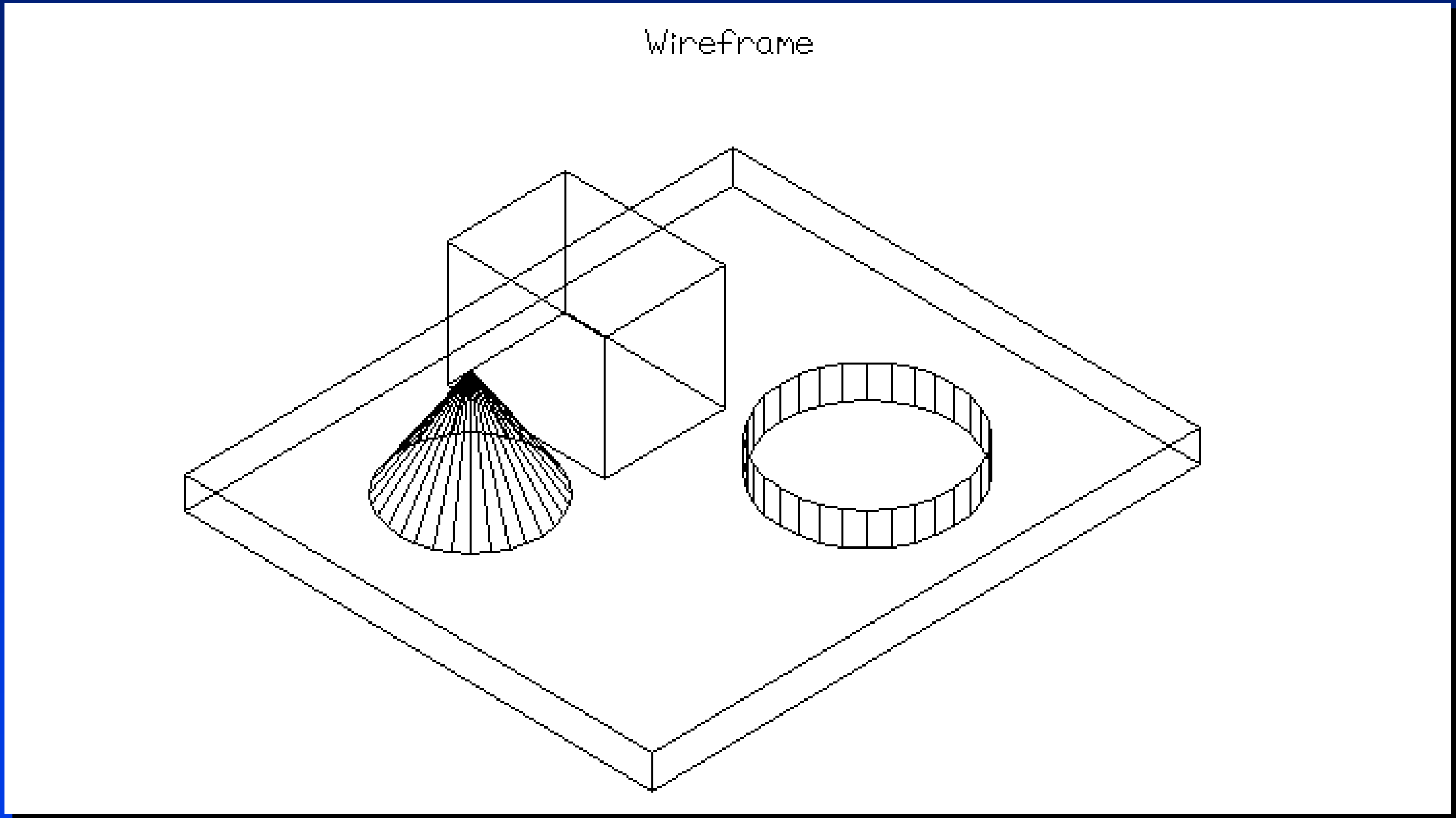
CSE328 Fundamentals of Computer Graphics: Concepts, Theory, Algorithms, and Applications

Hong Qin
Department of Computer Science
Stony Brook University (SUNY at Stony Brook)
Stony Brook, New York 11794-4400
Tel: (631)632-8450; Fax: (631)632-8334
qin@cs.sunysb.edu
http://www.cs.sunysb.edu/~qin

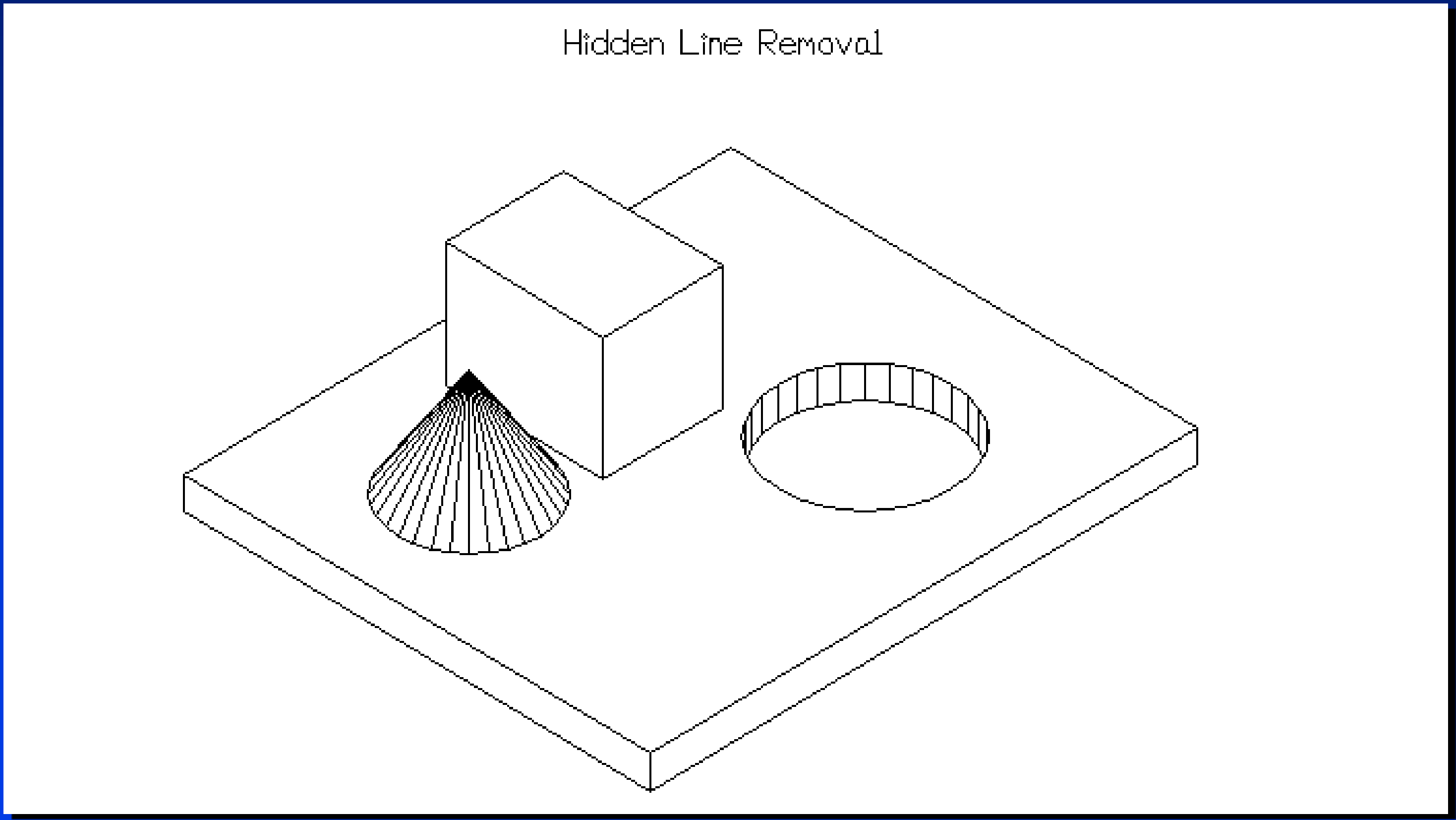
Hidden Surface Removal



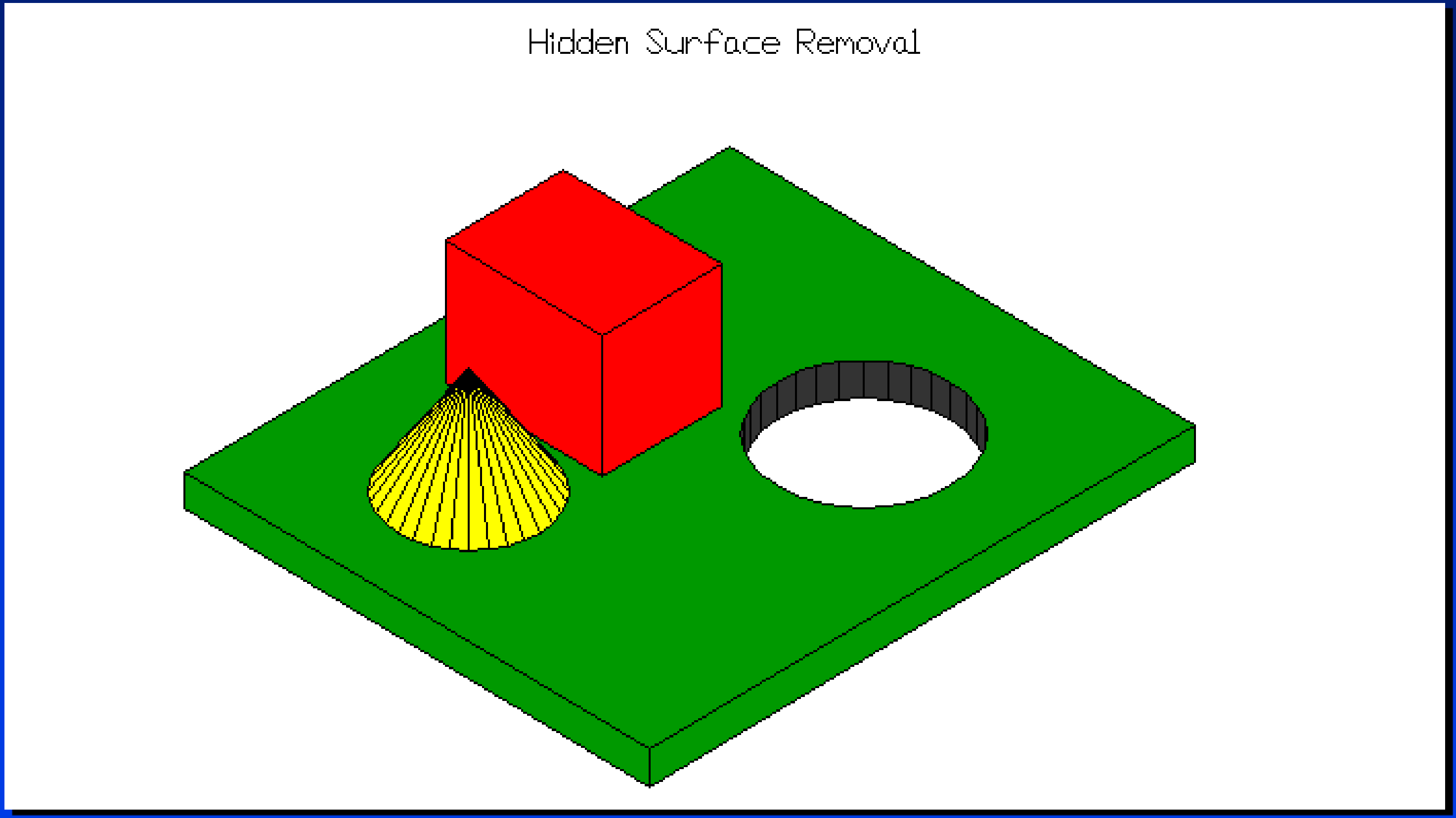
No Lines Removed



Hidden Lines Removed



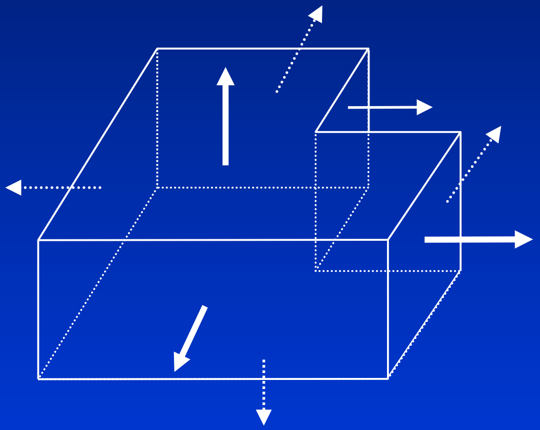
Hidden Surfaces Removed



Hidden Surface Removal (a.k.a. Visibility)

Motivation

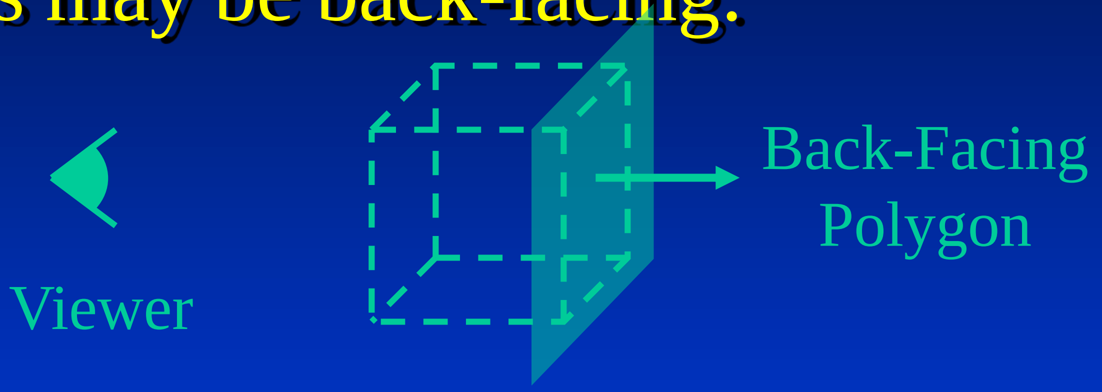
- Suppose that we have the polyhedron which has 3 totally visible surfaces, 4 totally invisible/hidden surfaces, and 1 partially visible/hidden surface.



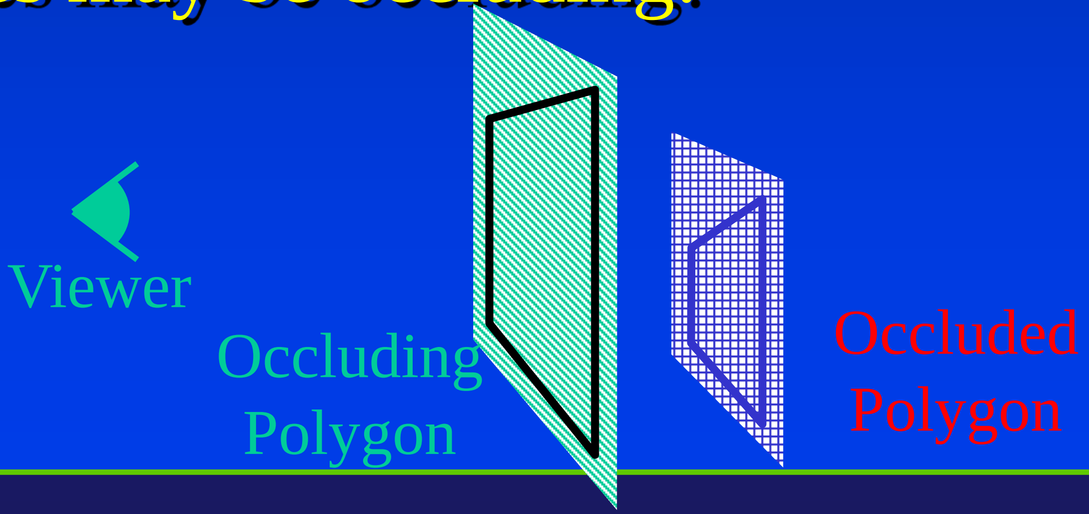
- Obviously, invisible/hidden surfaces do not contribute to the final image during graphics production.
- The procedure that distinguishes between visible surfaces from invisible/hidden surfaces is called *visible-surface determination*, which is often called *hidden-surface removal*.

Motivation

- Surfaces may be back-facing:

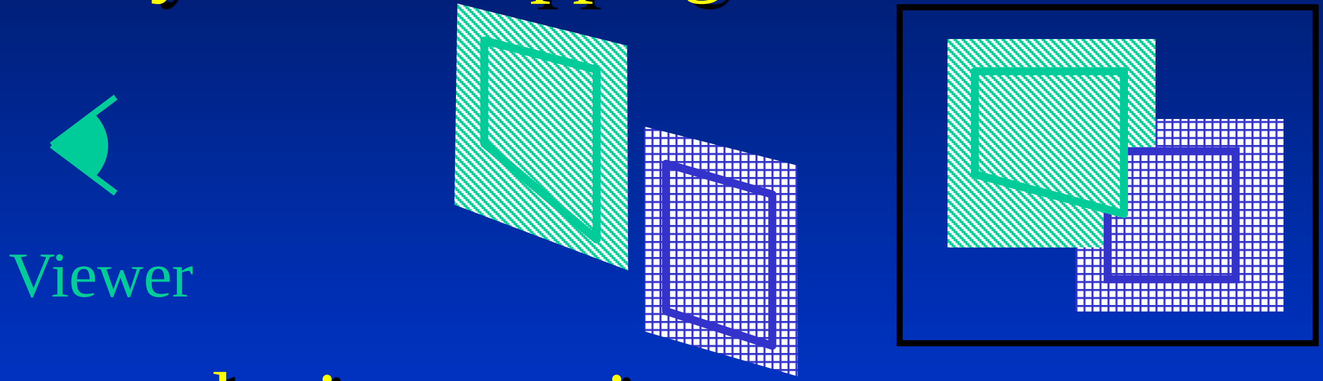


- Surfaces may be occluding:

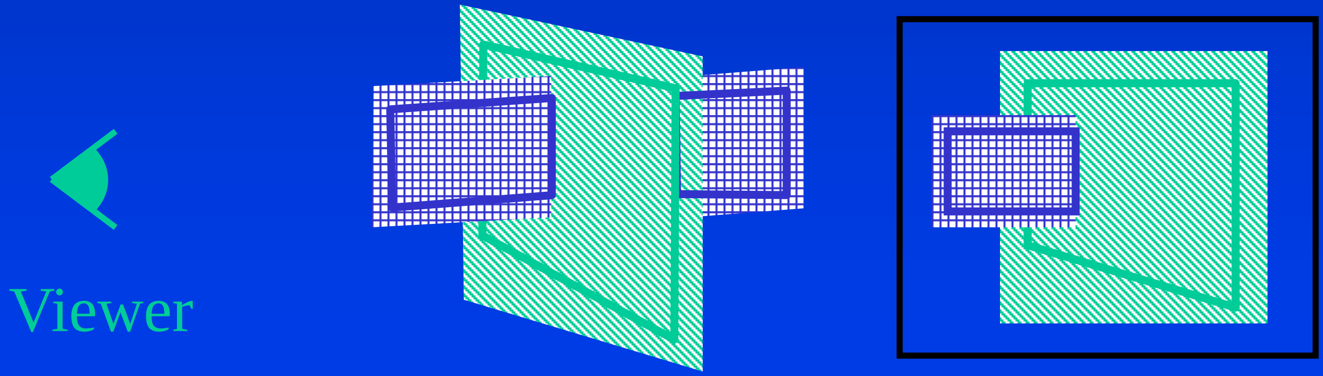


Motivation

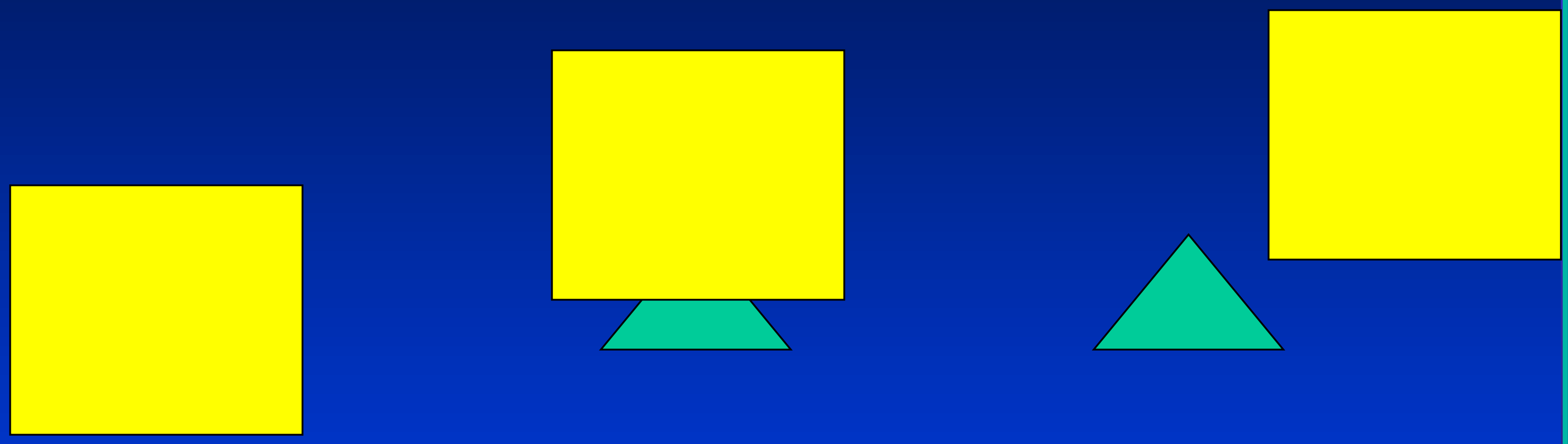
- Surfaces may be overlapping:



- Surfaces may be intersecting:



Occlusion: Full, Partial, None



Full **Partial** **None**

- The rectangle is closer than the triangle
- Should appear in front of the triangle

Hidden Surface Removal

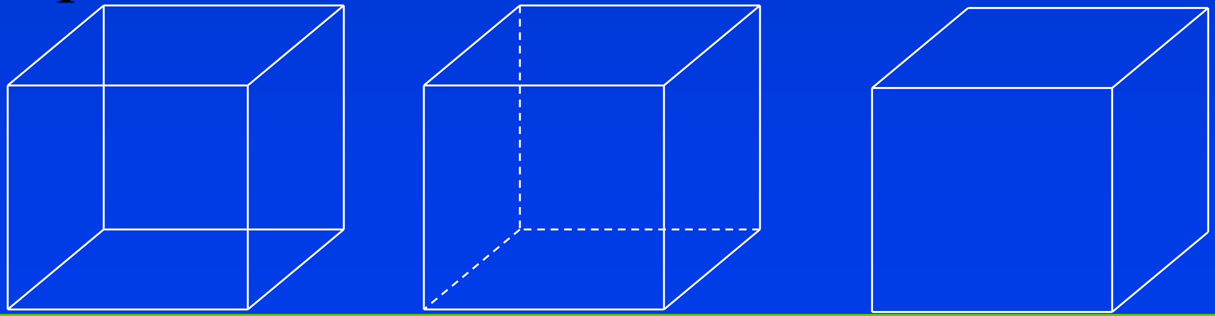
- Motivation
- Algorithms for Hidden Surface Removal
  - Back-face detection
  - Painter's algorithm
  - Ray casting
  - Scan-line
  - Z-buffer
  - Area subdivision
- Tradeoffs when comparing different techniques

Image or Object Space Algorithms

- Ideally an object space method converts the 3D scene into a list of 2D areas to be painted
- Image space decides for each pixel which surface to paint

Hidden Surface Removal

- In 3D we must be concerned with whether or not objects are obscured by other objects
- Most objects are opaque so they should obscure/block things behind them
- Also Known As: visible surface detection methods or hidden surface elimination methods
- Related problem : Hidden Line Removal

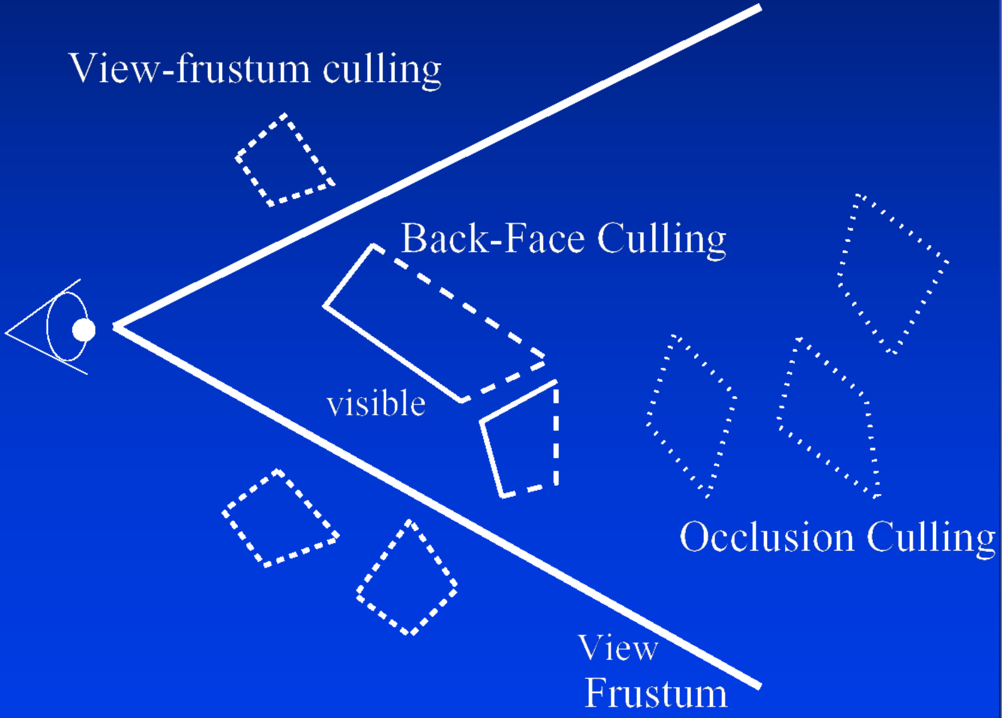


Visibility

- Basic assumption: All polygons are **opaque**
- What polygons are visible with respect to your view frustum?
  - Outside: **View Frustum Clipping**
    - Remove polygons outside of the view volume
    - For example, Liang-Barsky 3D Clipping
  - Inside: **Hidden Surface Removal**
    - Backface culling
      - Polygons facing away from the viewer
    - Occlusion
      - Polygons farther away are obscured by closer polygons
      - Full or partially occluded portions
- Why should we remove these polygons?
    - Avoid unnecessary expensive operations on these polygons later

Visibility Culling

- Culling techniques
  - *View-frustum culling*
    - Reject geometry outside the viewing volume
  - *Back-face culling*
    - Reject geometry facing away from the observer
  - *Occlusion culling*
    - Reject objects occluded by others

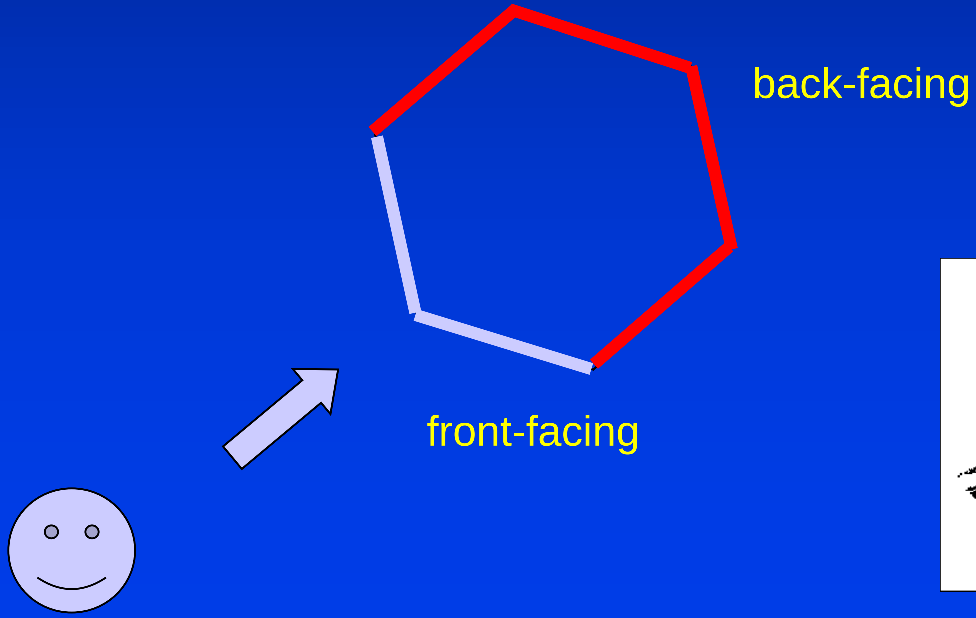


Back Face Culling

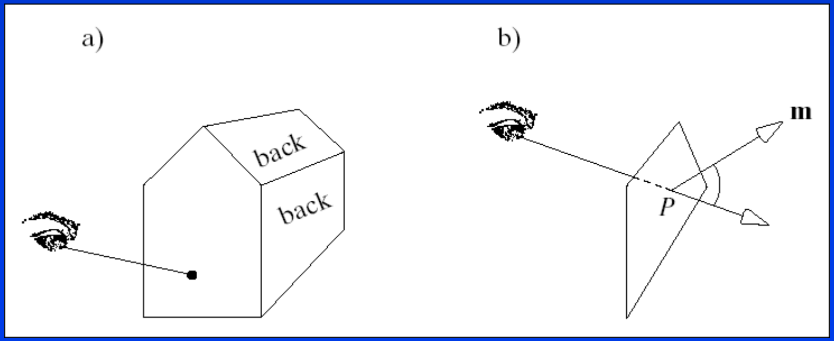
- Performance goals in real-time rendering
  - frame rate: 60-72 frames/sec?
  - resolution: 1600x1200?
  - photorealism (if necessary): undistinguishable from real scene!
- Unfortunately, there is no real upper limit on the scene complexity.
- We should *cull* those portions of the scene that are not considered to contribute to the final image as early as possible, and process only the rest of the scene.
- The simplest is *back-face culling*, which distinguishes between front faces (faces towards the viewer) and back faces (faces away from the viewer).

Backface Culling

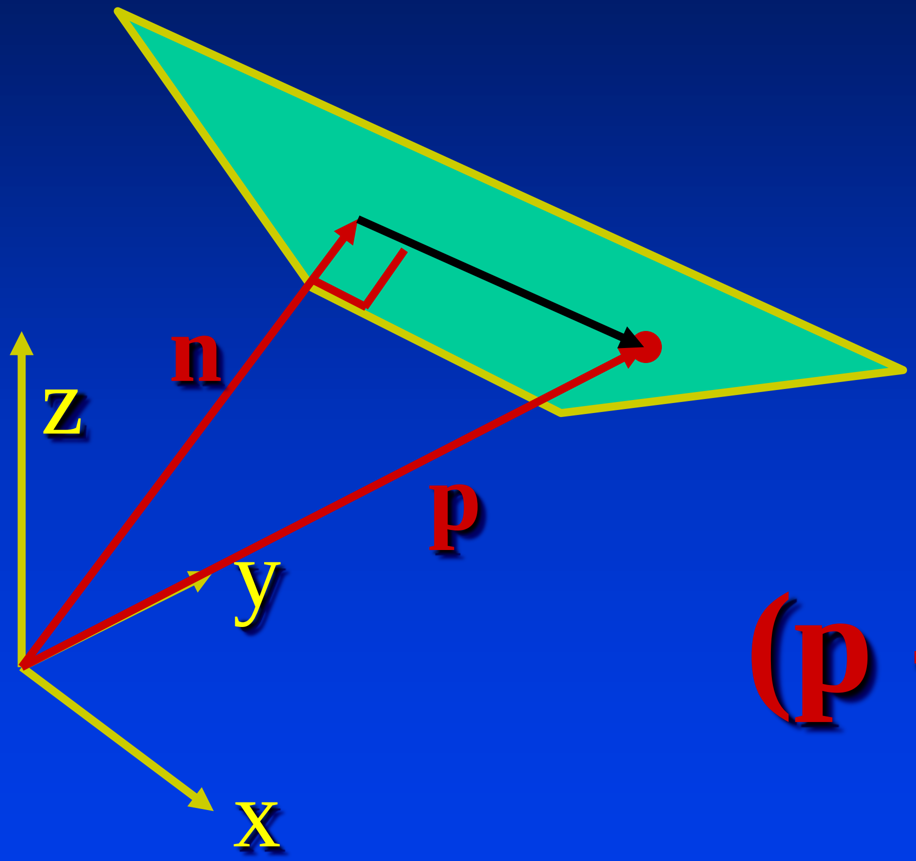
- Avoid drawing polygons facing away from the viewer
  - Front-facing polygons occlude these polygons in a closed polyhedron
- Test if a polygon is front- or back-facing?



Ideas?



Which Side of a Plane?

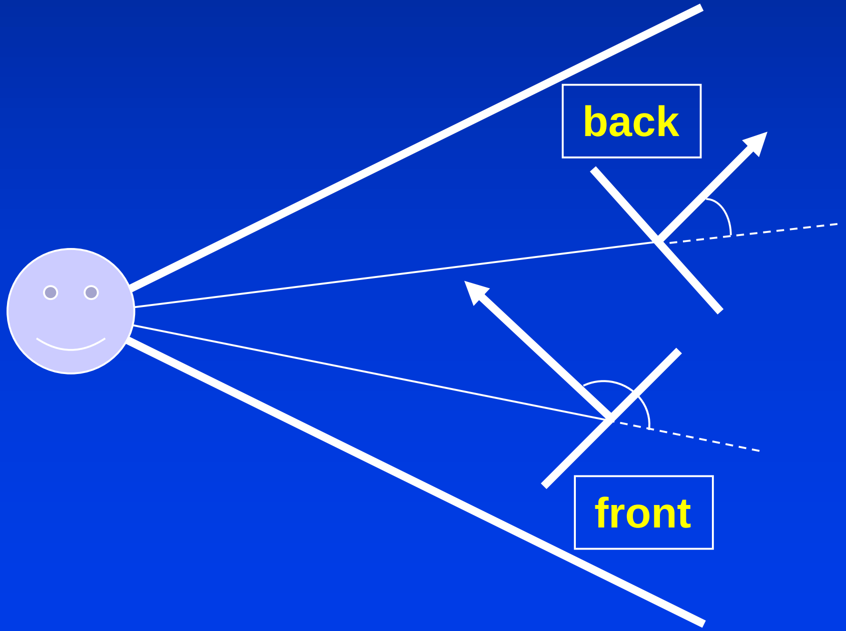


$$(\mathbf{p} - \mathbf{n}).\mathbf{n} = 0$$

Detecting Back-face Polygons

- The polygon normal of a …
  - ➢ front-facing polygon points **towards** the viewer
  - ➢ back-facing polygon points **away** from the viewer

If $(\mathbf{n} \bullet v) > 0 \Rightarrow$ "back-face"
If $(\mathbf{n} \bullet v) \leq 0 \Rightarrow$ "front-face"
v = view vector



- Eye-space test …. EASY!
  - ➢ "back-face" if $\mathbf{n}_z < 0$

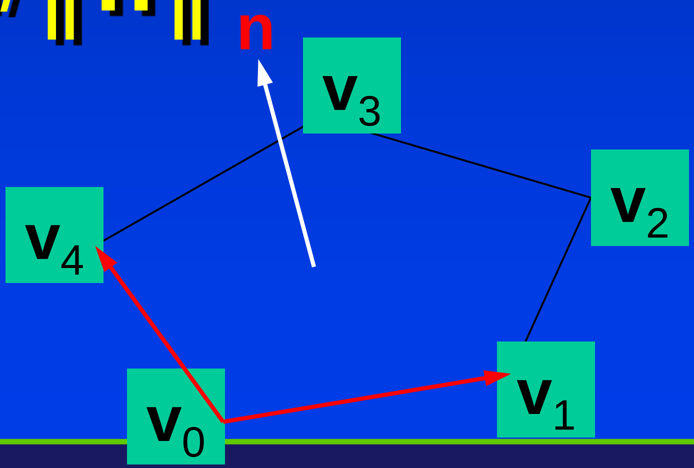
- **glCullFace**(GL_BACK)

Polygon Normals

- Let polygon vertices $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$,…, $\mathbf{v}_{n-1}$ be in counterclockwise order and co-planar
- Calculate normal with cross product:

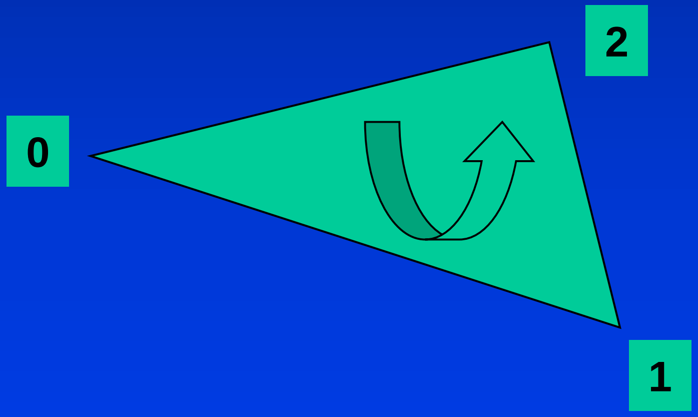
$$\mathbf{n} = (\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_{n-1} - \mathbf{v}_0)$$

- Normalize to unit vector with $\mathbf{n}/\|\mathbf{n}\|$

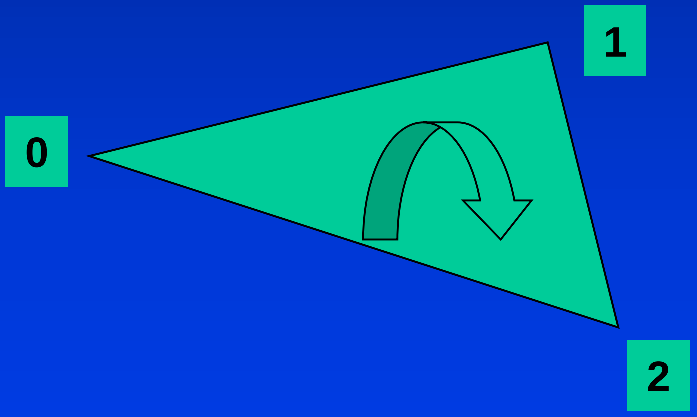


Normal Direction

- Vertices counterclockwise $\Rightarrow$ Front-facing
- Vertices clockwise $\Rightarrow$ Back-facing



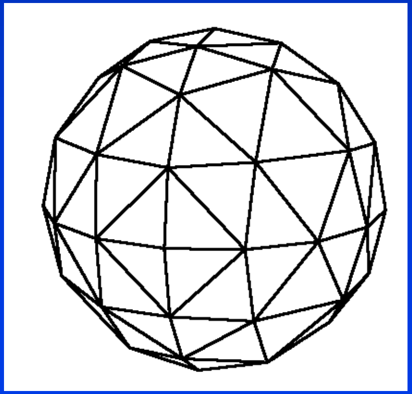
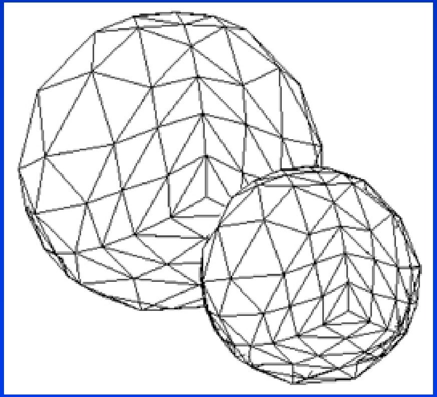
Front facing



Back facing

Back Face Culling

- Back-face culling works as a preprocessing step for hidden surface removal, and is very powerful in that almost half of polygons of an object are discarded as back faces.
- Especially, for a single convex polyhedron, the back-face culling does the entire job of hidden-surface removal.
- Hidden-surface removal is applied to only the remaining front faces.

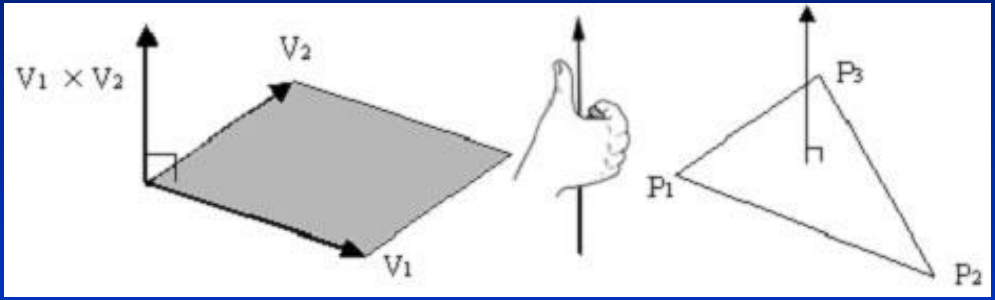



single convex polyhedron

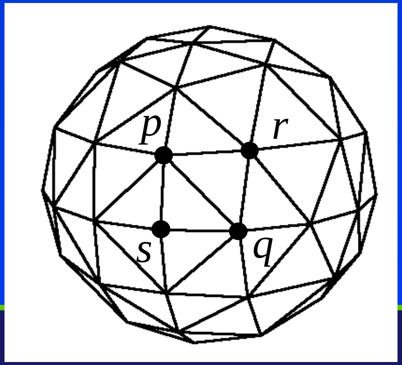
convex, but not a single polyhedron

Face Normal

- The normal of a triangle $<p_1, p_2, p_3>$ is computed as v_1 X v_2 where v_1 is the vector connecting p_1 and p_2, and v_2 connects p_1 and p_3.



- If the vertices of every polygon are ordered consistently (CCW or CW), we can make every polygon's normal point out of the polyhedron. All mesh data used in this class have triangles of CCW vertex ordering.



$t_1 = (p, q, r)$

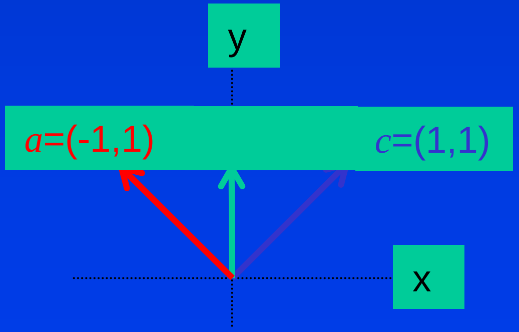
$t_2 = (s, q, p)$

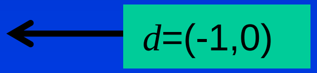
t_1's normal = $(q-p)$X$(r-p)$

t_2's normal = $(q-s)$X$(p-s)$

Dot Product (revisited)

- Recall the dot product of two vectors v_i & v_j .
  - If $v_i \bullet v_j = 0$, v_i & v_j are orthogonal/perpendicular.
  - If $v_i \bullet v_j > 0$, *angle < 90*
  - If $v_i \bullet v_j < 0$, *angle > 90*
- The same applies to 3D.

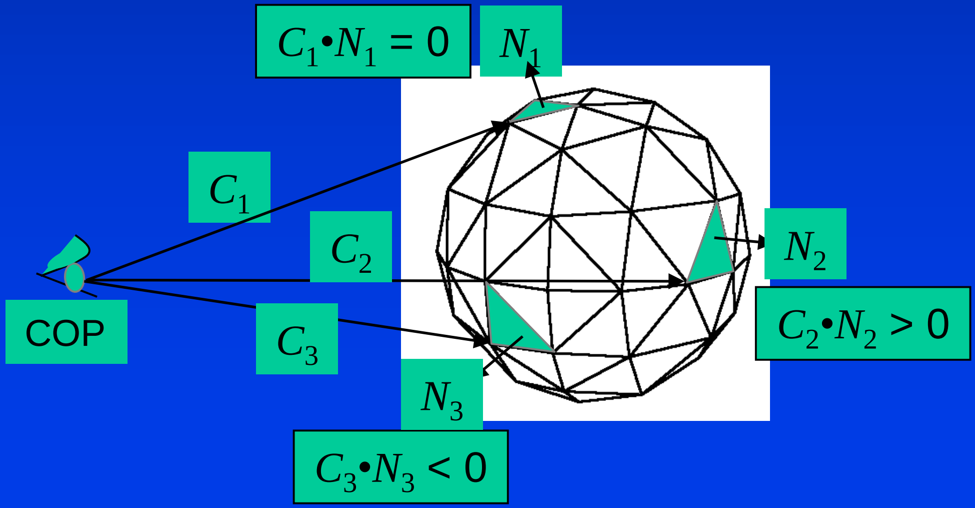




$a \cdot d = (-1,1) \cdot (-1,0) = 1 > 0$
$b \cdot d = (0,1) \cdot (-1,0) = 0$
$c \cdot d = (1,1) \cdot (-1,0) = -1 < 0$

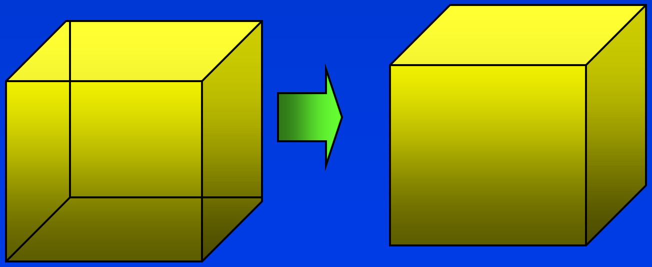
Dot Product for Back Face Culling

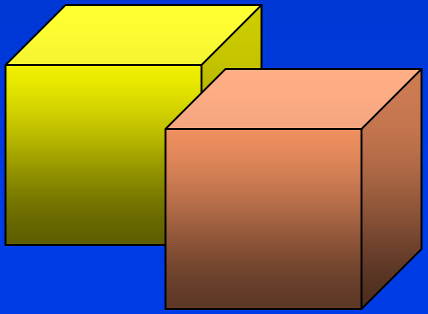
- Determining if a polygon is a front face or a back face
  - Generate a vector C connecting COP and a vertex of the polygon.
  - Take the dot product $C \bullet N$ of the vector C and the polygon's normal N.
  - If $C \bullet N > 0$, it's a back face.
  - If $C \bullet N < 0$, it's a front face.
  - If $C \bullet N = 0$, it's a face that is projected as an edge/silhouette on the screen.



Back Face Removal (Culling)

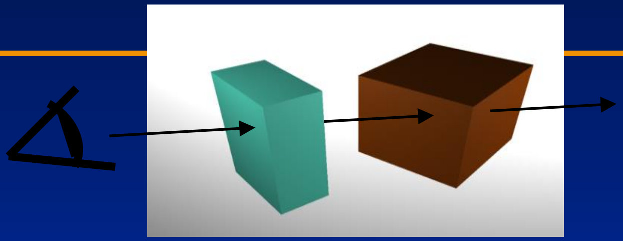
- Used to remove unseen polygons from convex, closed polyhedron (Cube, Sphere)
- Does not completely solve hidden surface problem since one polyhedron may obscure another

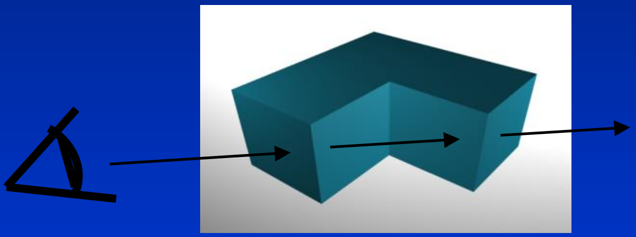




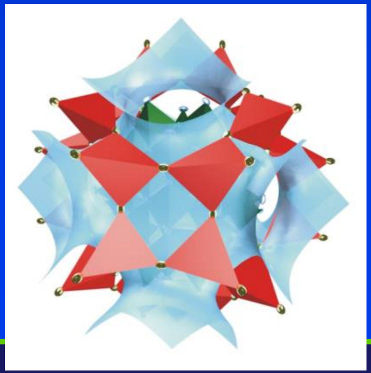
Backface Culling

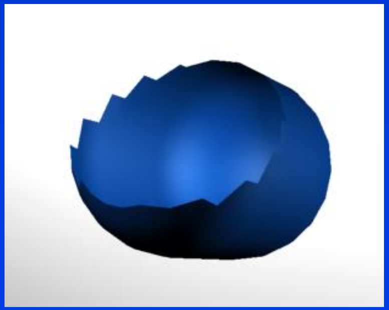
- For all polygons $\mathbf{P}_i$
  - Find Polygon Normal **n**
  - Find Viewer Direction **v**
  - IF $(\mathbf{n} \bullet \mathbf{v} \leq 0)$
    - Then CULL $\mathbf{P}_i$





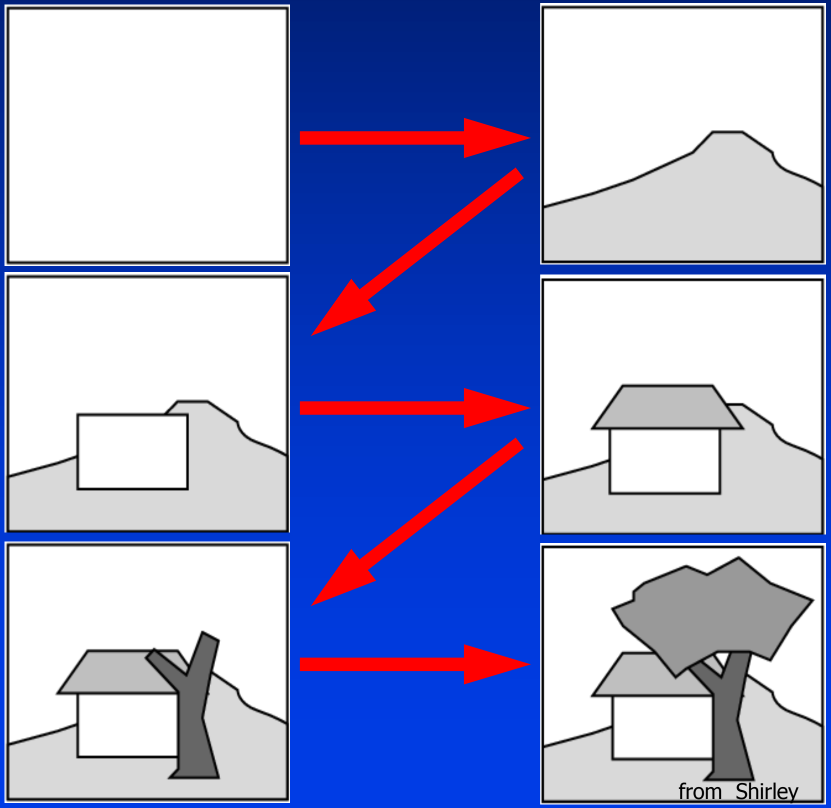
- Does not work well for:
  - Overlapping front faces due to
    - Multiple objects
    - Concave objects
  - Non-polygonal models
  - Non-closed Objects





Painter’s Algorithm

- Basic Idea
  - A painter creates a picture by drawing background scene elements before foreground ones
- Requirements
  - Draw polygons in back-to-front order
  - Need to **sort** the polygons by depth order to get a correct image



Painter’s Algorithm

1. Sort all objects’ z_{min} and z_{max}

Painter's Algorithm

1. Sort all objects' z_{min} and z_{max}
2. If an object is uninterrupted (its z_{min} and z_{max} are adjacent in the sorted list), it is fine

Painter’s Algorithm

1. Sort all objects’ z_{min} and z_{max}
2. If an object is uninterrupted (its z_{min} and z_{max} are adjacent in the sorted list), it is fine
3. If 2 objects DO overlap

   3.1 Check if they overlap in x

   - If not, they are fine

   3.2 Check if they overlap in y

   - If not, they are fine
   - If yes, need to split one

Painter’s Algorithm

- The splitting step is the tough one
  - Need to find a plane to split one polygon so that each new polygon is entirely in front of or entirely behind the other
  - Polygons may actually intersect, so then need to split each polygon by the other

Painter’s Algorithm

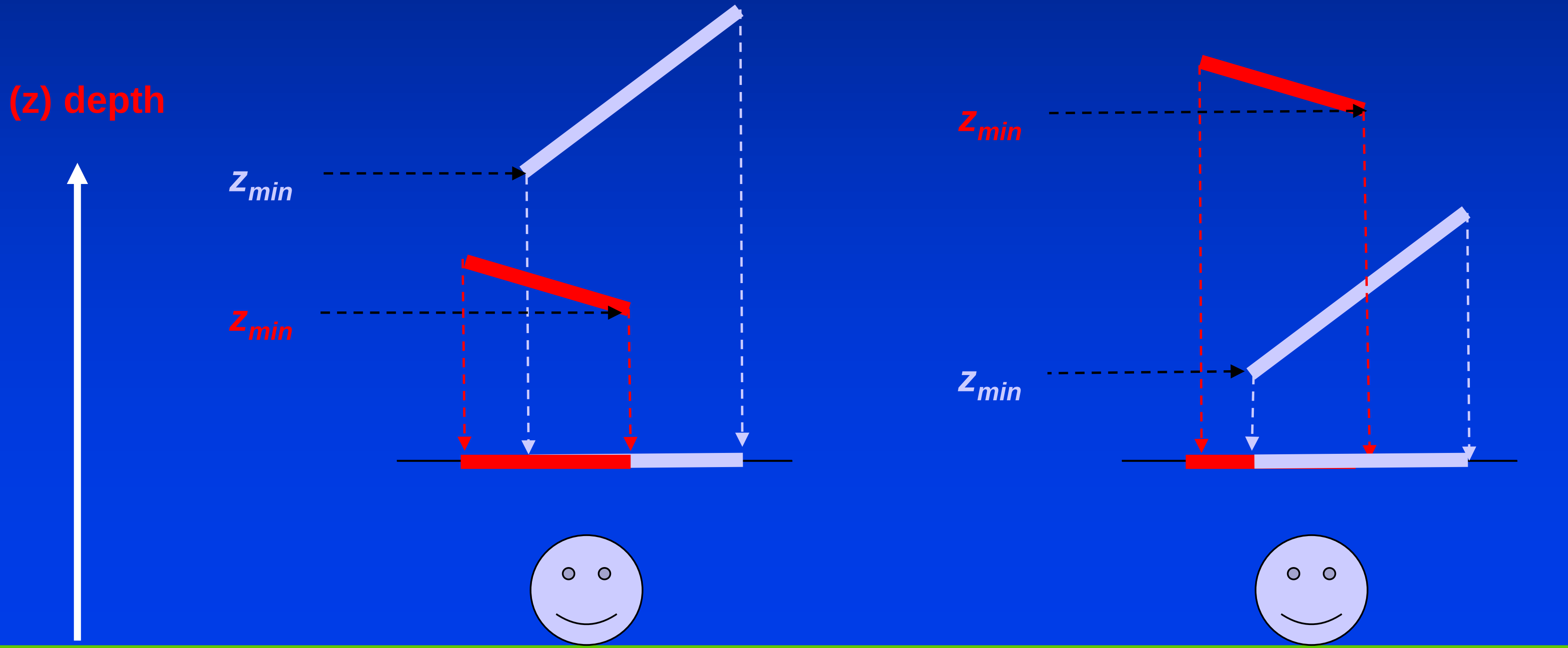
- The splitting step is the tough one
  - Need to find a plane to split one polygon so that each new polygon is entirely in front of or entirely behind the other
  - Polygons may actually intersect, so then need to split each polygon by the other
- After splitting, you can resort the list and should be fine

Painter’s Algorithm-Summary

- Advantages
  - Simple algorithm for ordering polygons
- Disadvantages
  - Sorting criteria difficult to produce
  - Redraws same pixel many times
  - Sorting can also be expensive

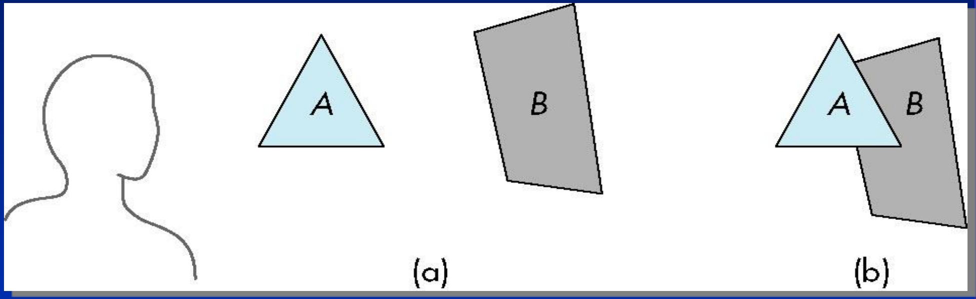
Painter's Algorithm

- Compute z_{min} ranges for each polygon
- Project polygons with furthest z_{min} first



Depth Sorting

- The *painter's algorithm:* draw from back to front



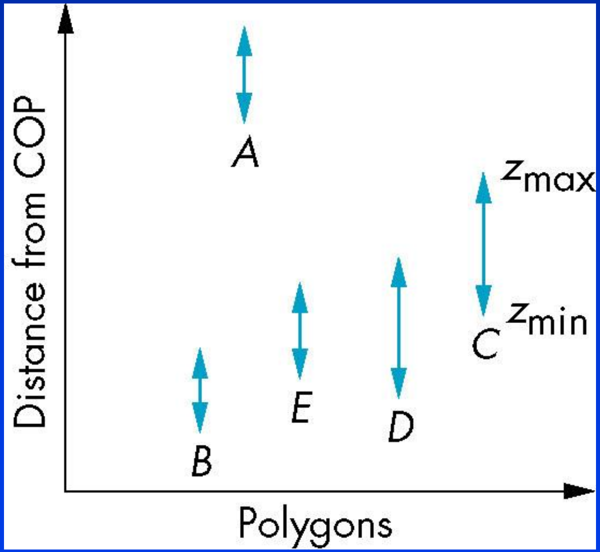
- Depth-sort hidden surface removal:
  - sort display list by z-coordinate from back to front
  - render/display
- Drawbacks
  - it takes some time (especially with bubble sort!)
  - it doesn't work

Depth Sorting

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

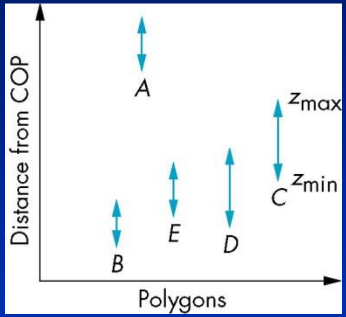
- Order polygons and deal with easy cases first, harder later

Polygons sorted by distance from COP

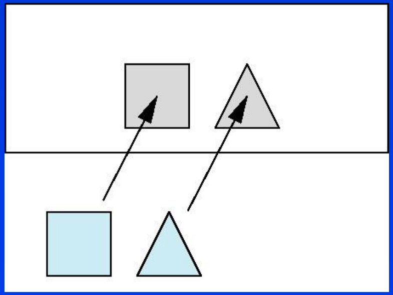


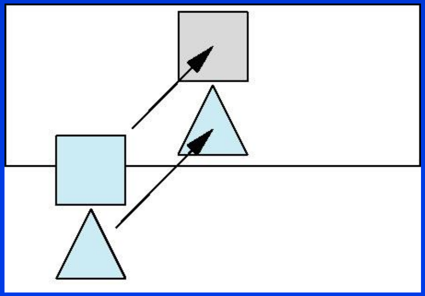
Easy Cases

- A lies behind all other polygons
  - Can render



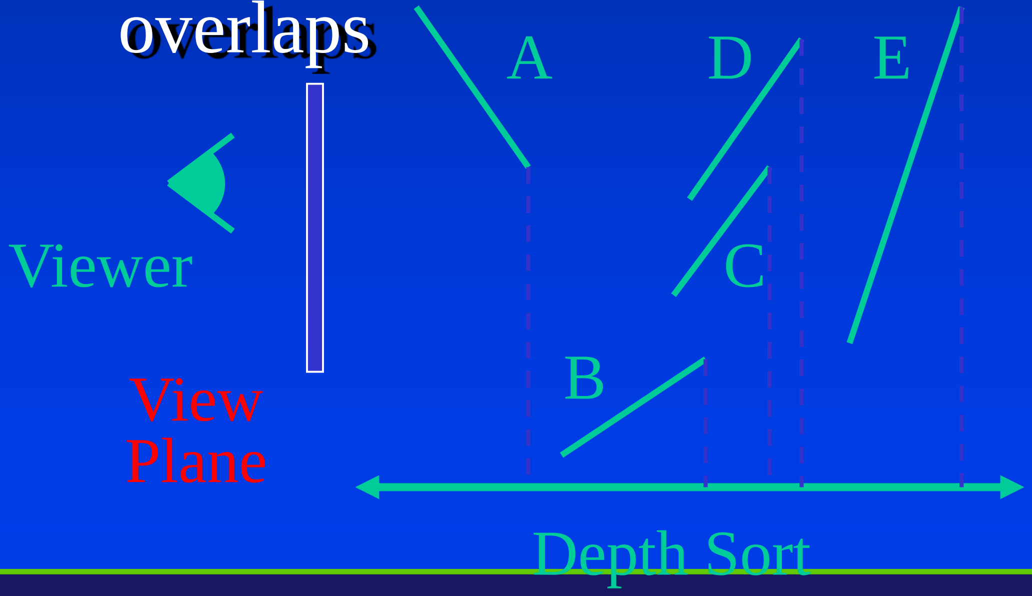
- Polygons overlap in z but not in either x or y
  - Can render independently



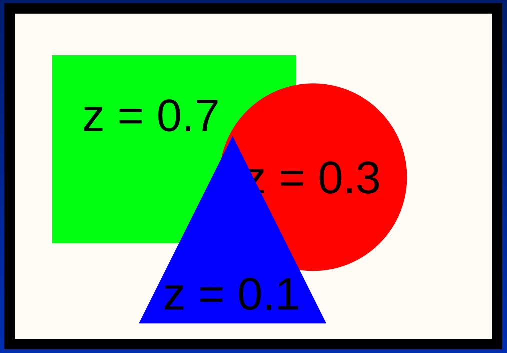


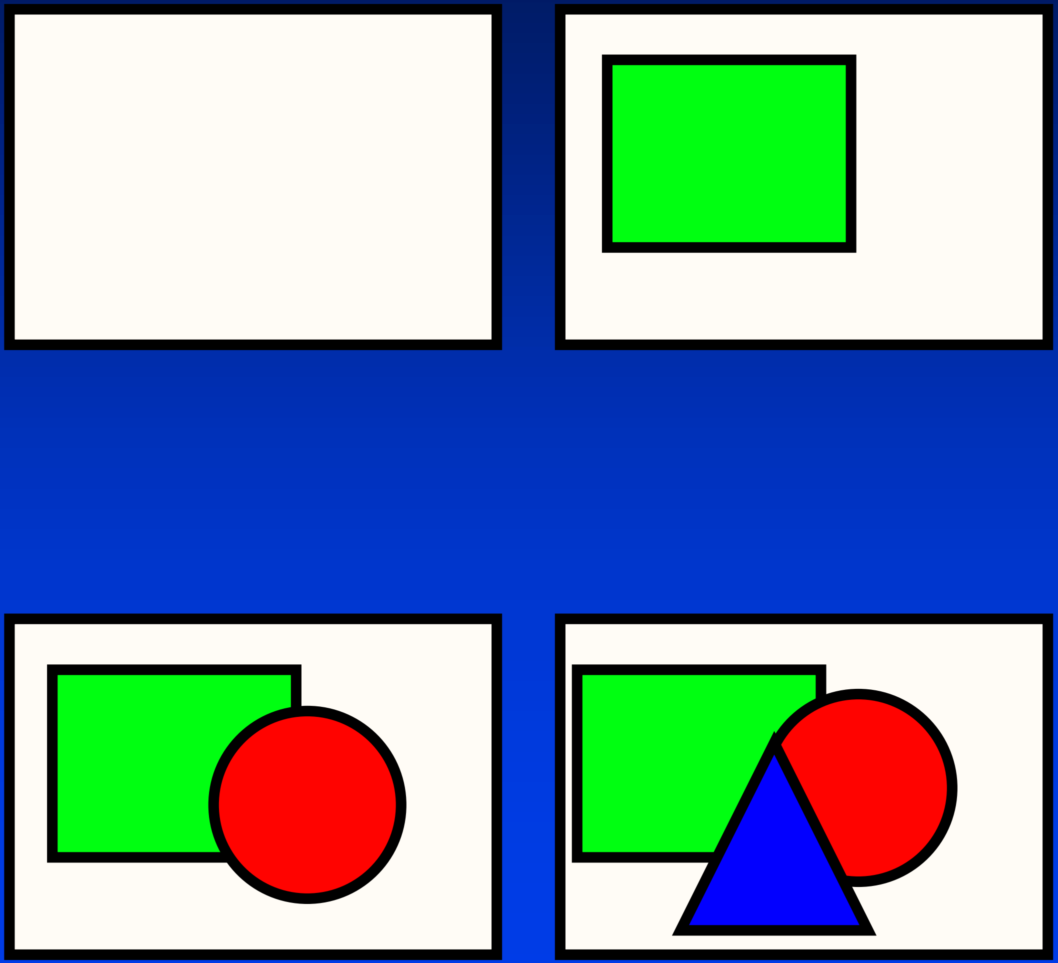
Depth Sorting Example

- Painter’s Algorithm:
  - Sort surfaces in order of decreasing maximum depth
  - Scan convert surfaces in order starting with ones of greatest depth, reordering as necessary based on overlaps



Painter's Example





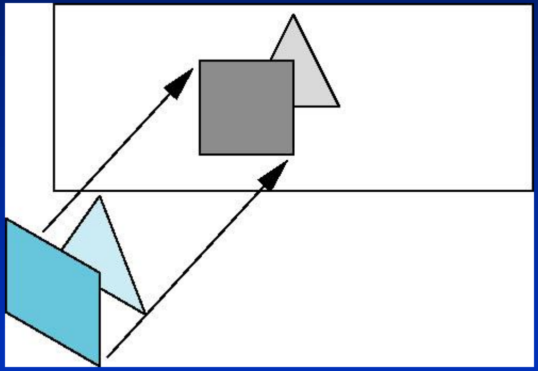
Sort by depth:

Green rect

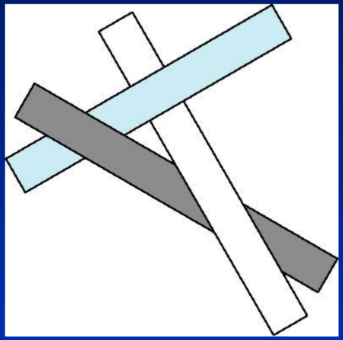
Red circle

Blue tri

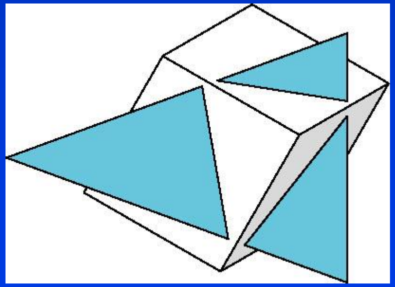
Hard Cases



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



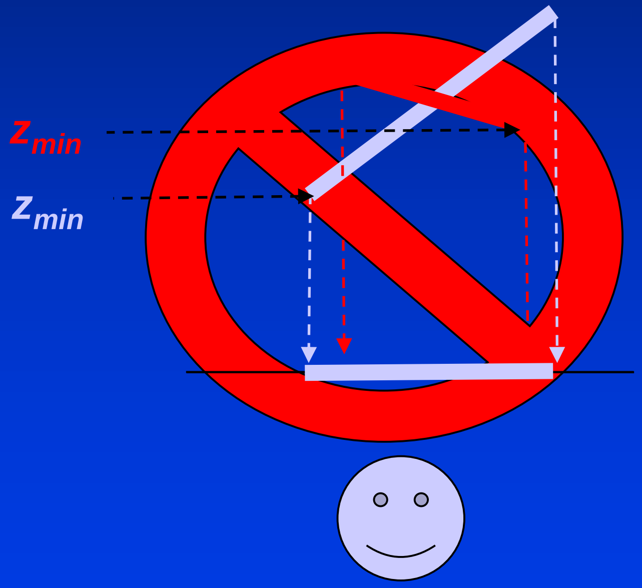
cyclic overlap

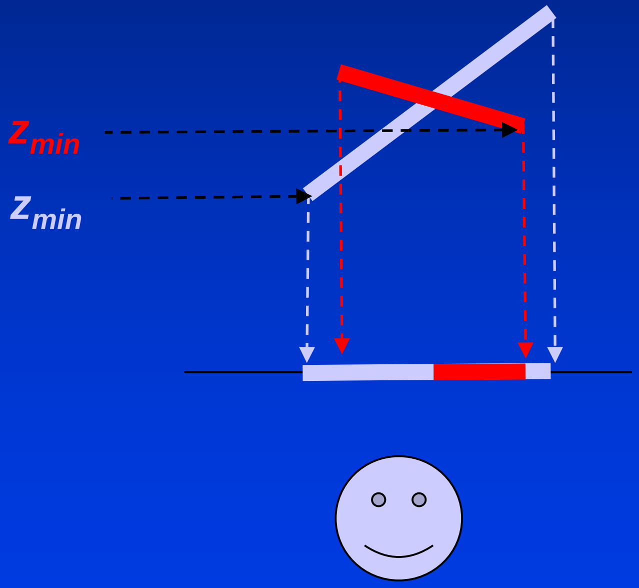


penetration

Painter’s Algorithm

- Problem: Can you get a total sorting?

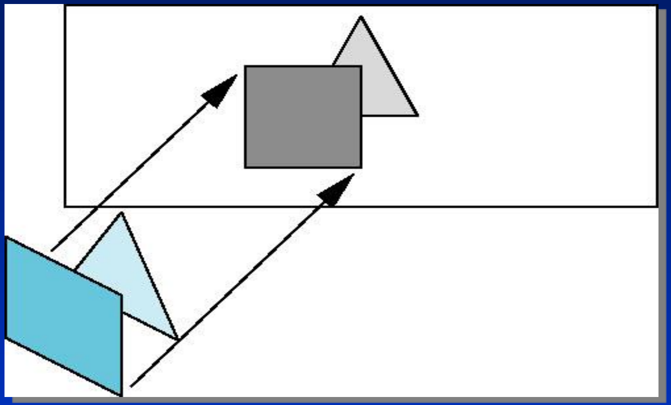


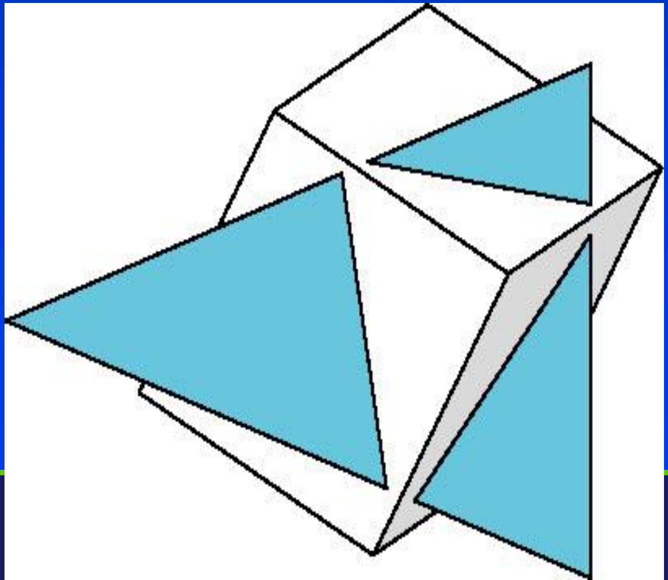


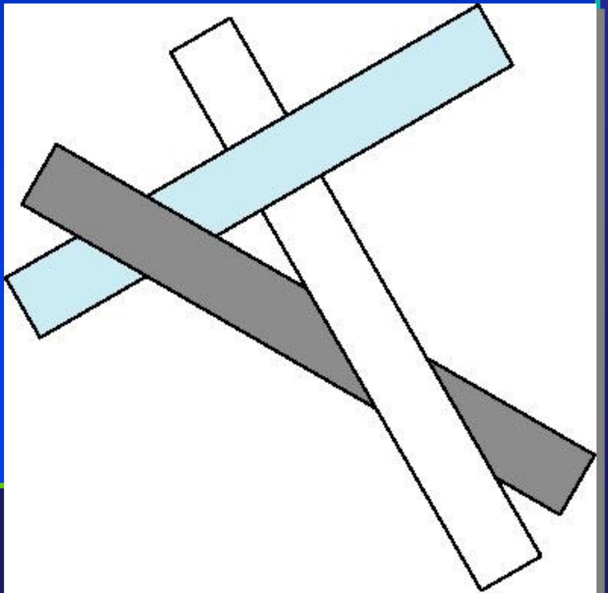
Correct?

Depth-sorting Difficulties

- Polygons with overlapping projections
- Cyclic overlap
- Interpenetrating polygons
- What to do?

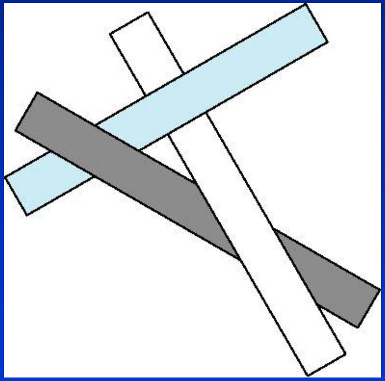






Painter’s Algorithm

- Cyclic Overlap
  - ➢ How do we sort these three polygons?



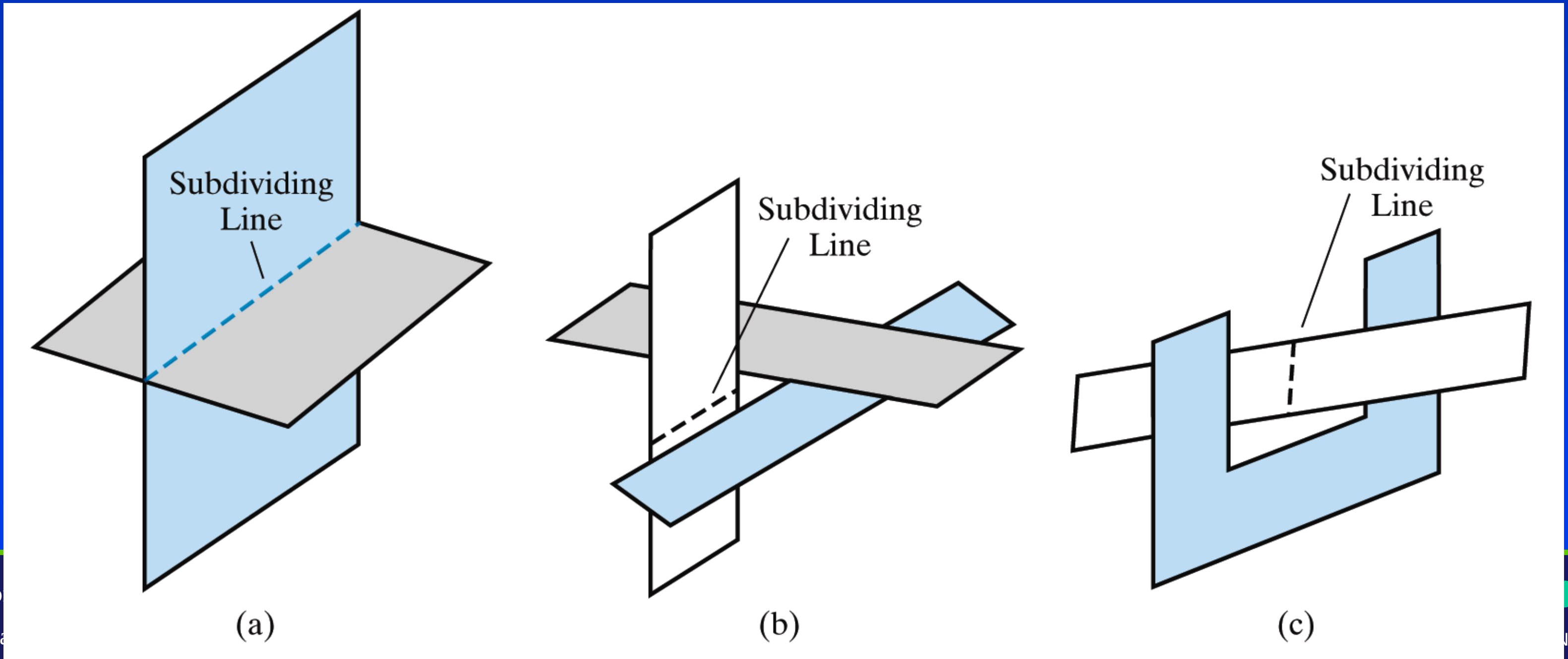
- Sorting is nontrivial
  - ➢ Split polygons in order to get a total ordering
  - ➢ Not easy to do in general

Depth Sorting

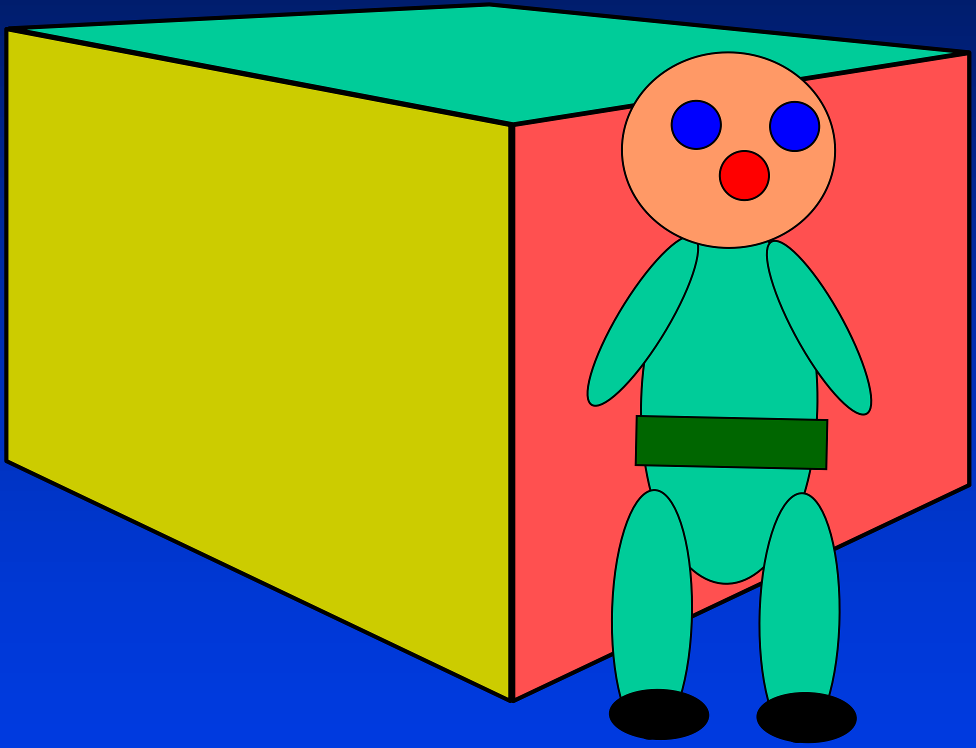
- Completely in front – put in front
- Not overlapping in x, y – either
- Intersecting – divide along intersection
- overlapping – divide along plane of one polygon.

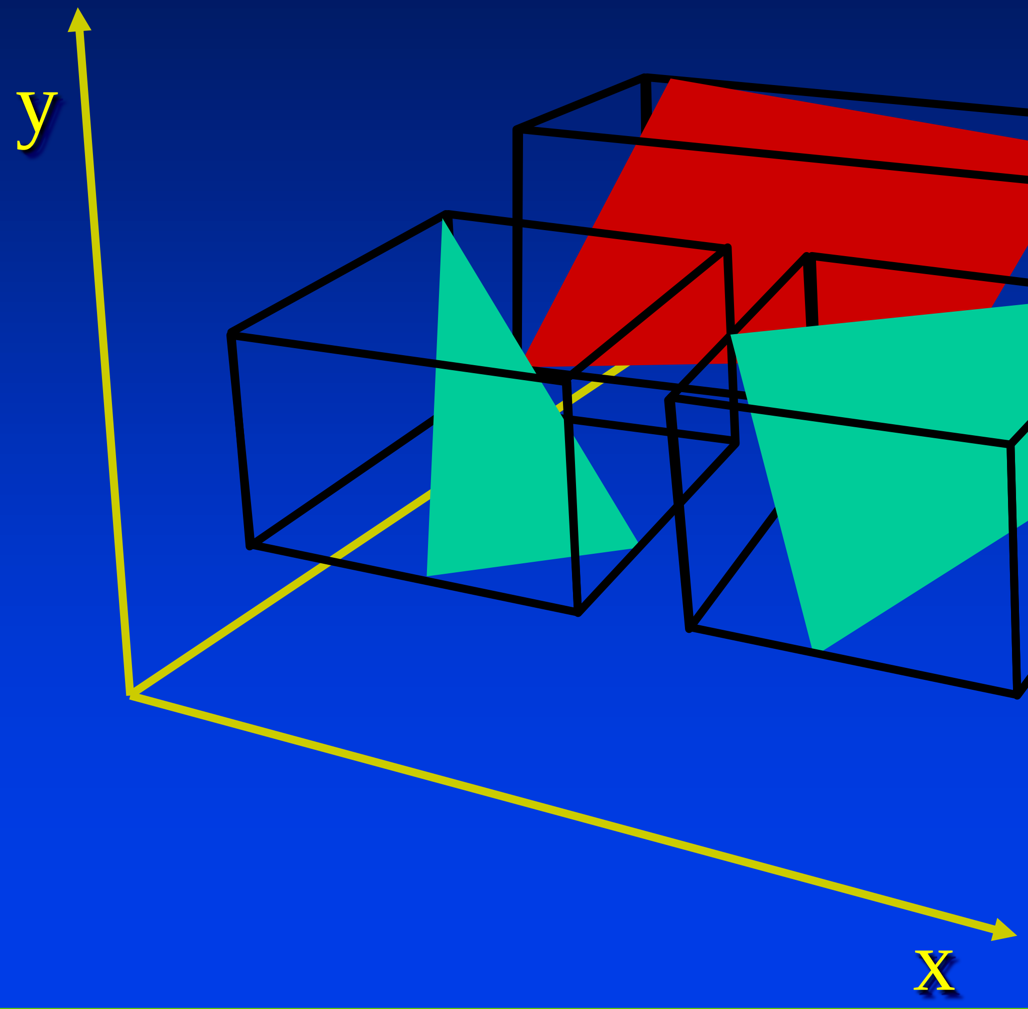
Depth Sorting

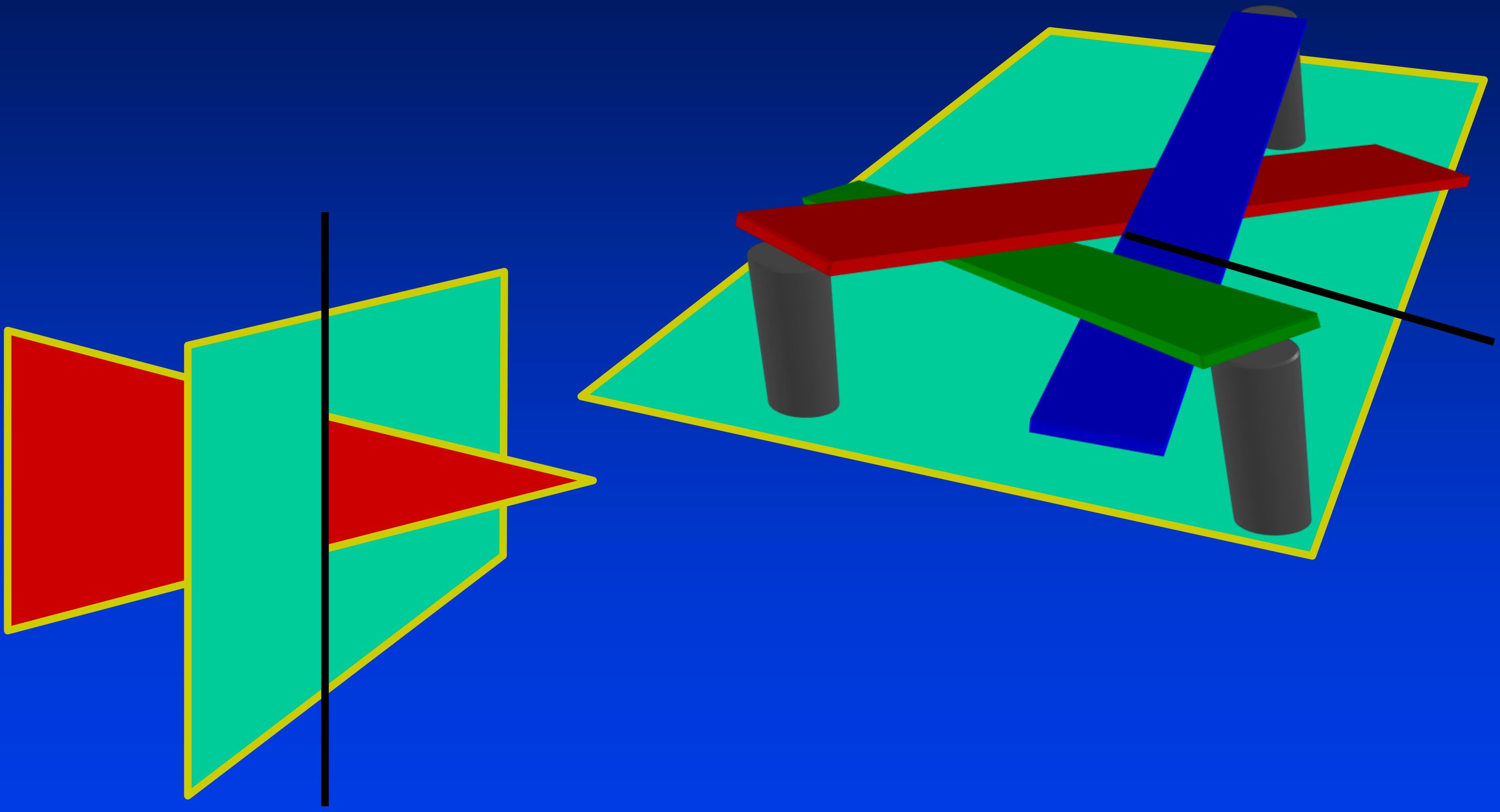
- Cyclically overlapping surfaces that alternately obscure one another
- We can divide the surfaces to eliminate the cyclic overlaps



Painter’s Algorithm




y
x



The Painters Algorithm

- Sort polygons according to their z values and render from back to front
- Ends up drawing over the polygons in the back (more or less)
- Problems arise when polygons overlap or are allowed to pierce one another
- *Heedless Painter's Algorithm*: sort by "farthest" point and draw in order
- *Depth sort* improves on this by splitting up overlapping polygons into less ambiguous pieces

Visibility

- How do we ensure that closer polygons overwrite further ones in general?

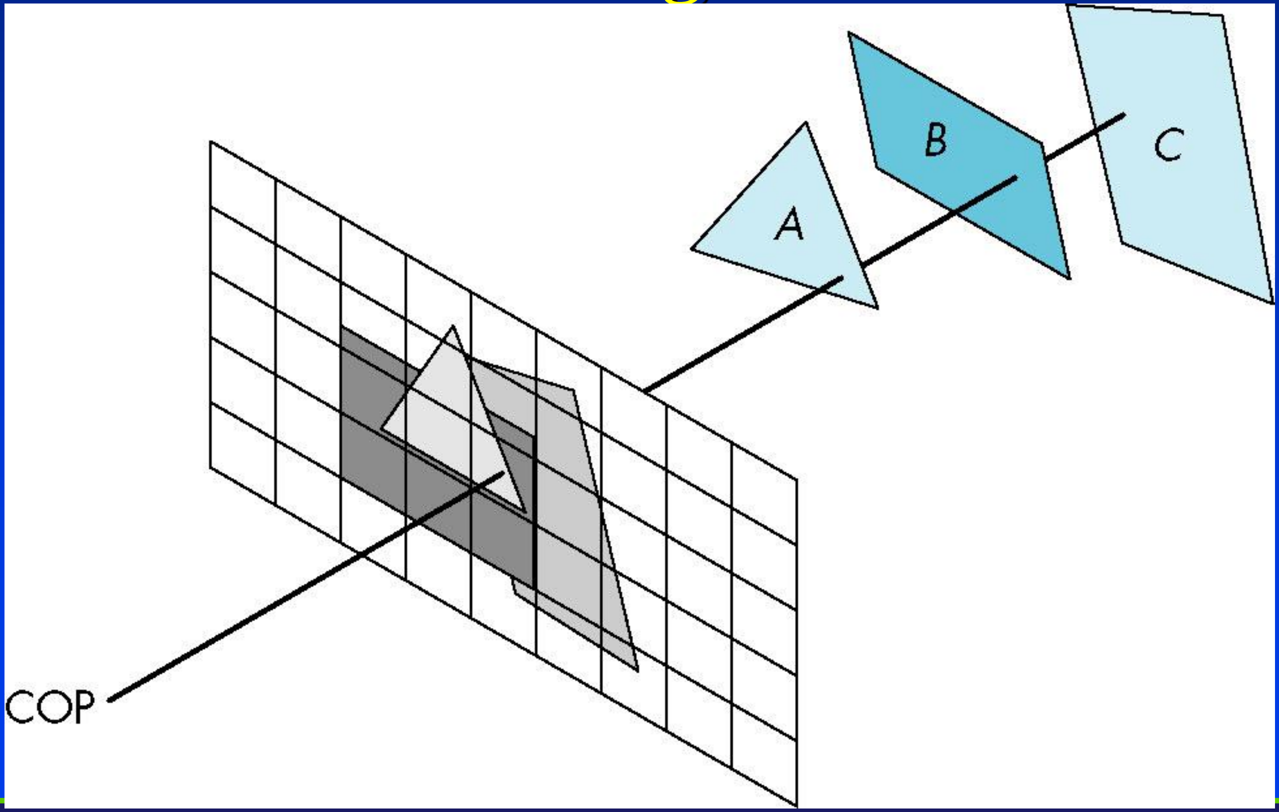
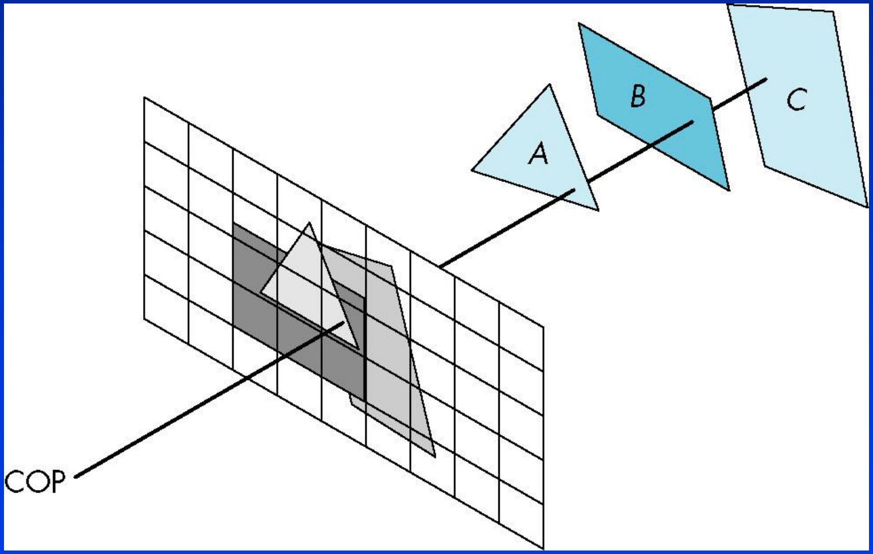


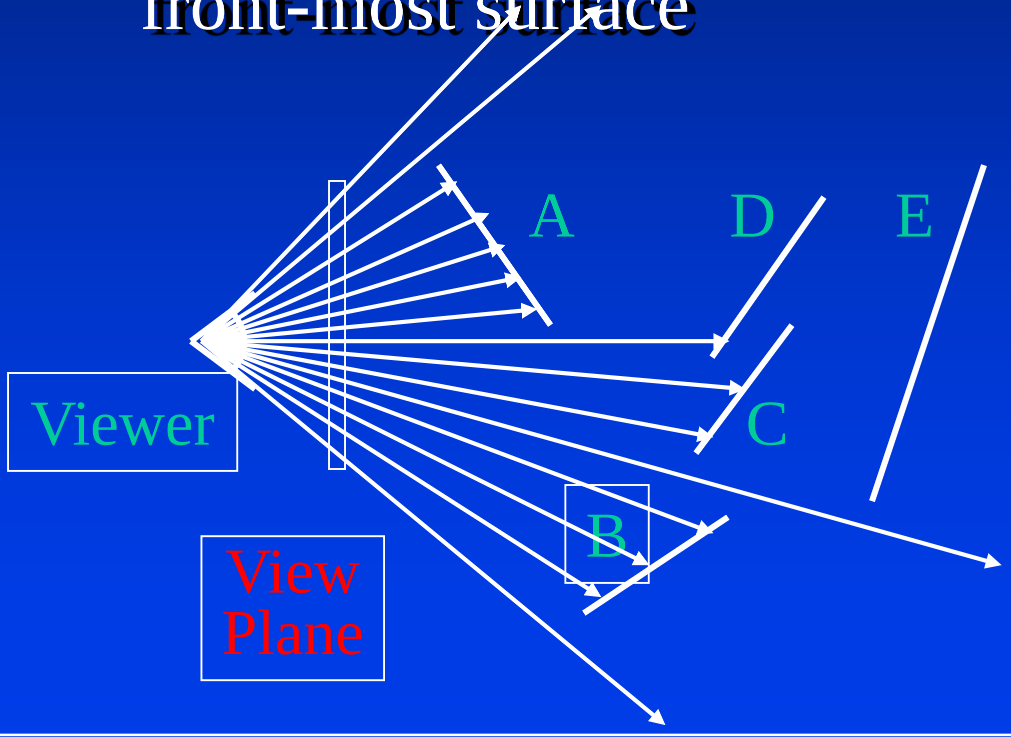
Image Space Approach

- Look at each projection (nm for an n x m frame buffer) and find closest of k polygons
- Complexity O(nmk)
- Ray tracing
- z-buffer



Ray Casting

- Algorithm:
  - Cast ray from viewpoint through each pixel to find front-most surface



The Z-buffer Algorithm

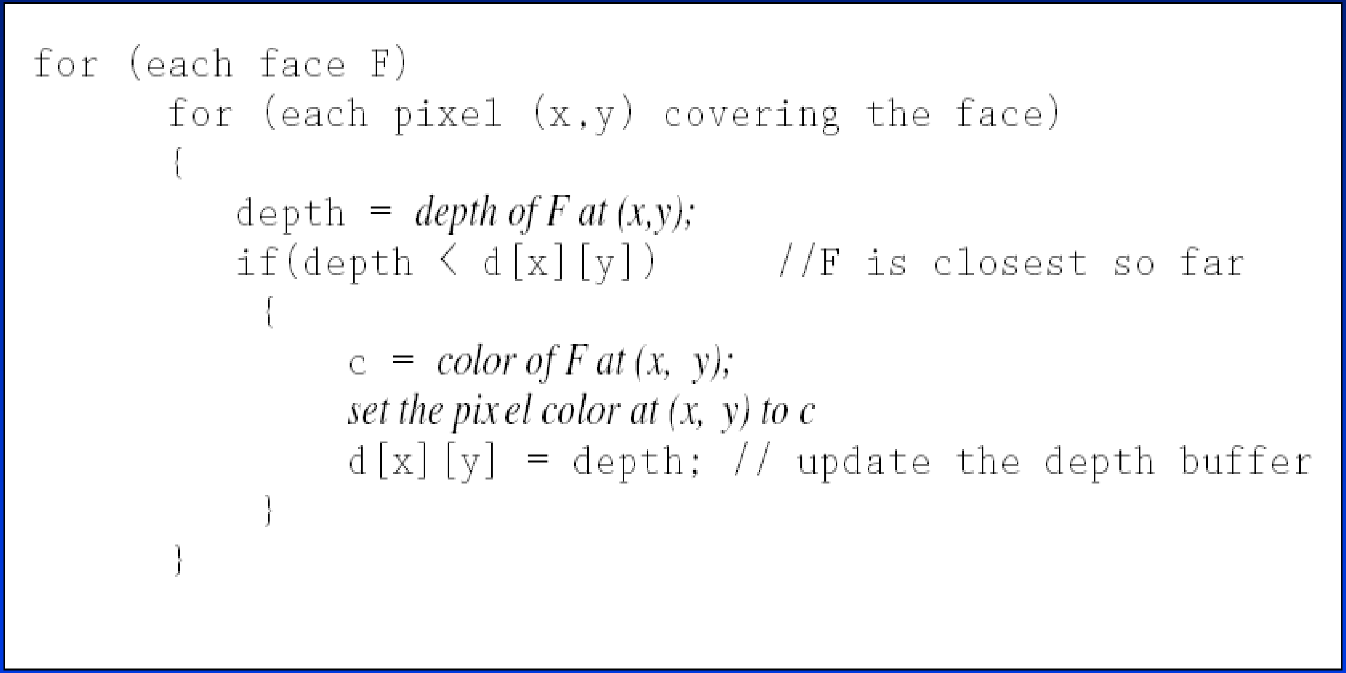
- The most widely-used hidden surface removal algorithm
- Relatively easy to implement in hardware or software
- An image-space algorithm which traverses scene and operates per polygon rather than per pixels
- We rasterize polygon by polygon and determine which (parts of) polygons get drawn on the screen
- Relies on a Secondary Buffer called the z-buffer or depth buffer
- Depth buffer has same width and height as the frame-buffer
- Each cell contains the z-value (distance from viewer) of the object at that pixel position

Z-Buffer

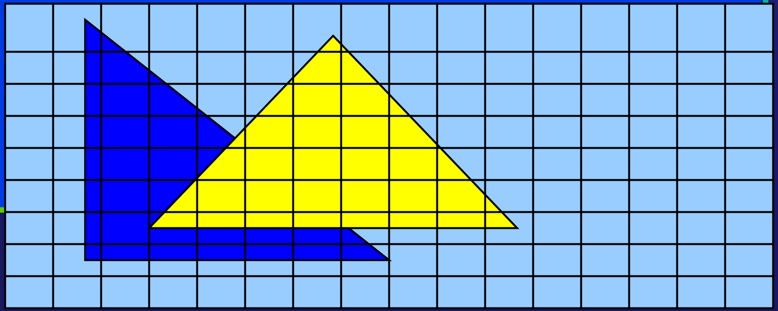
- Depth buffer (Z-Buffer)
  - ➢ A secondary image buffer that holds depth values
  - ➢ Same pixel resolution as the color buffer
  - ➢ Why is it called a **Z-Buffer**?
    - ➢ After eye space, depth is simply the *z*-coordinate
- Sorting is done at the pixel level
  - ➢ **Rule**: Only draw a polygon at a pixel if it is closer than a polygon that has already been drawn to this pixel

Z-Buffer Algorithm

- Visibility testing is done during rasterization



```
for (each face F)
     for (each pixel (x,y) covering the face)
     {
        depth = depth of F at (x,y);
        if(depth < d[x][y])      //F is closest so far
         {
            c = color of F at (x, y);
            set the pixel color at (x, y) to c
            d[x][y] = depth; // update the depth buffer
         }
     }
```

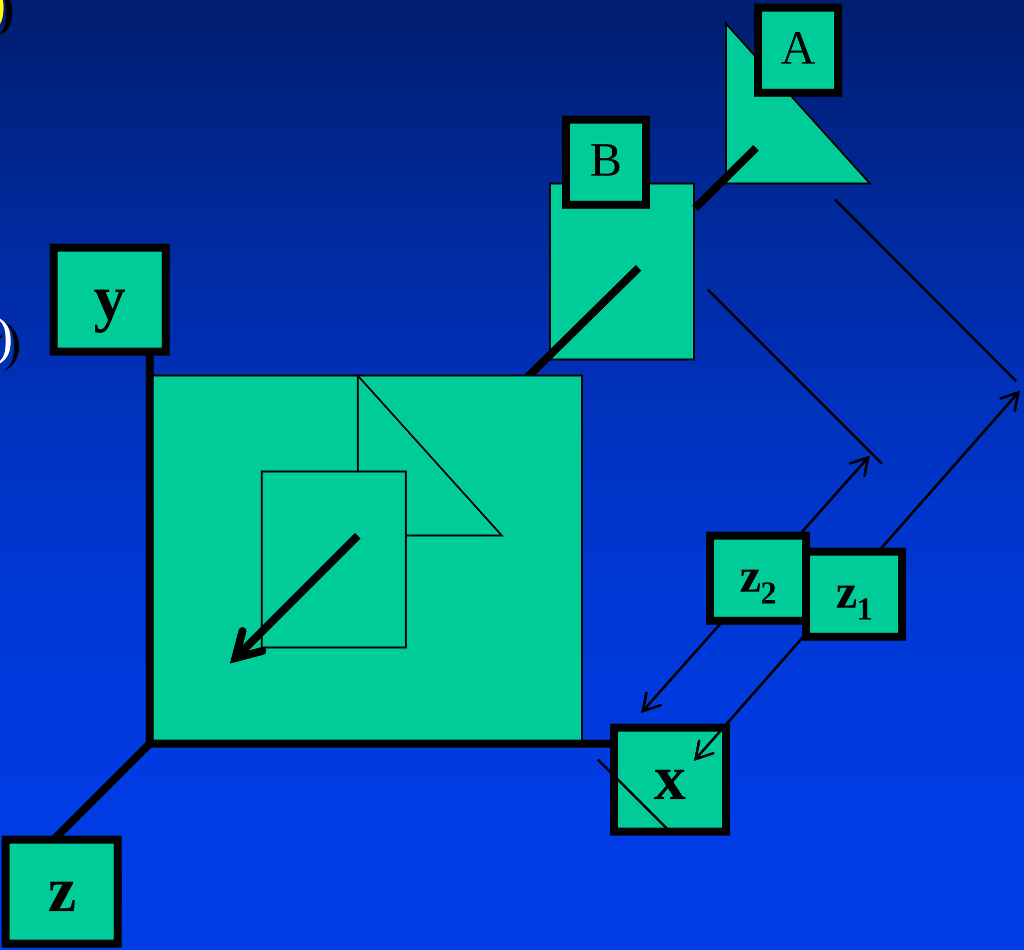


The Z-buffer Algorithm

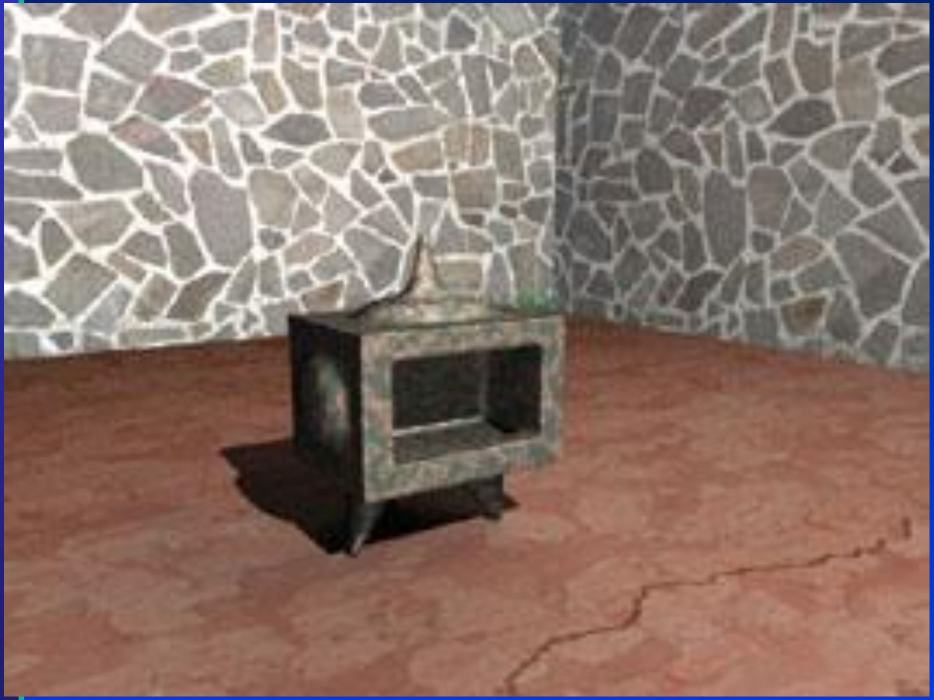
1. Initialize all *depth(x,y)* to 0 and *refresh(x,y)* to background color
2. For each pixel

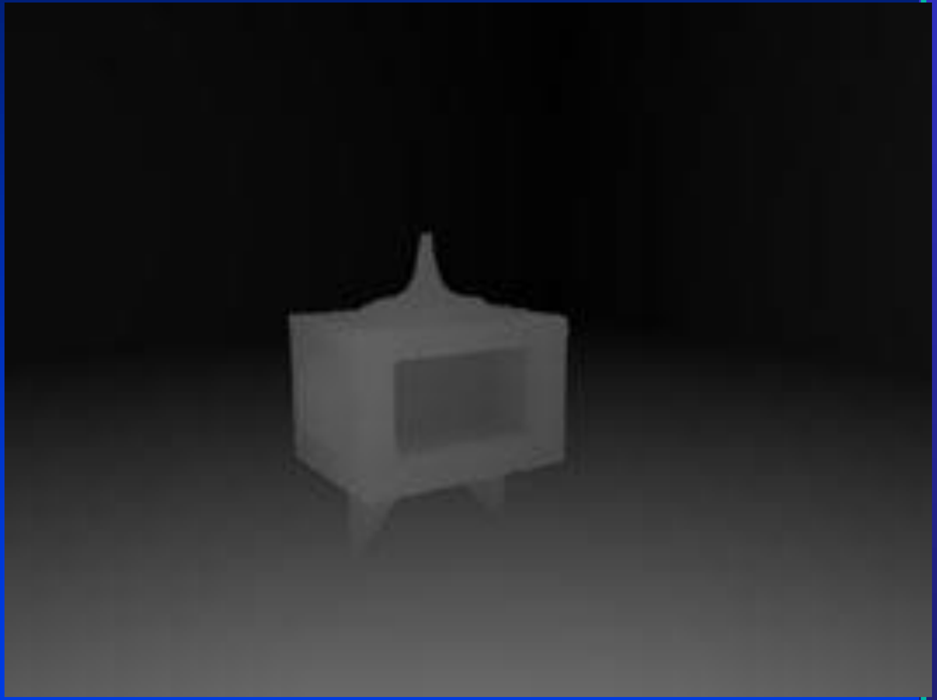
```
    1. Get current value depth(x,y)
    2. Evaluate depth value z
    if z > depth(x,y)
    then
    {
        depth(x,y) = z
        refresh(x,y) = $I_s$ (x,y)
    }
```

Calculate this using shading algorithm/illumination/fill color/texture



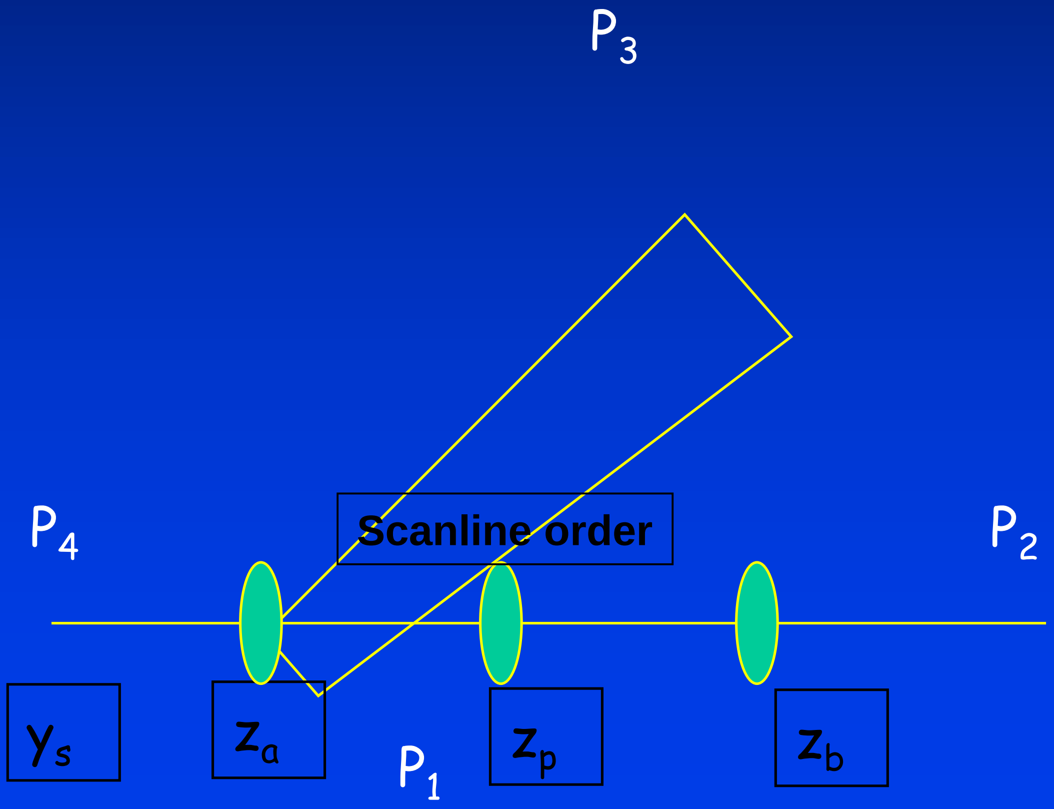
Z-buffer: A Secondary Buffer


Color buffer


Depth buffer

Z-Buffer

- How do we calculate the depth values on the polygon interior?



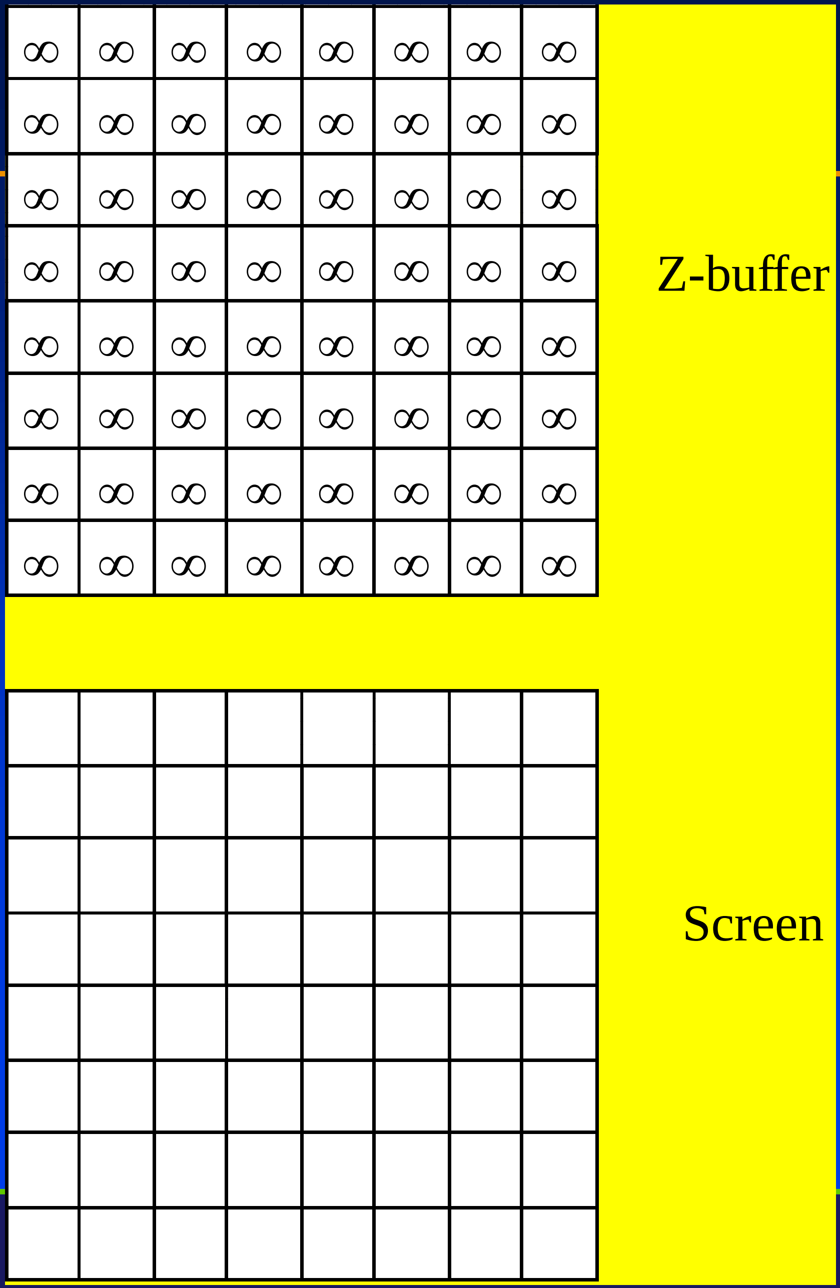
$$z_a = z_1 + (z_4 - z_1)\frac{(y_1 - y_s)}{(y_1 - y_4)}$$

$$z_b = z_1 + (z_2 - z_1)\frac{(y_1 - y_s)}{(y_1 - y_2)}$$

$$z_p = z_a + (z_b - z_a)\frac{(x_a - x_p)}{(x_a - x_b)}$$

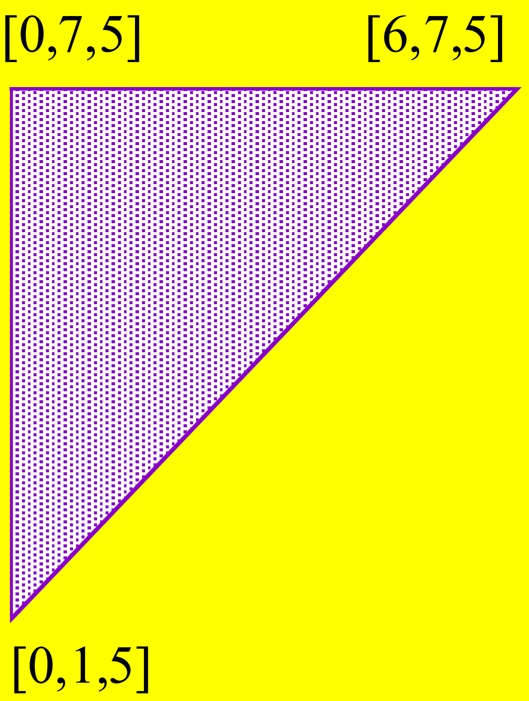
Bilinear Interpolation

Example



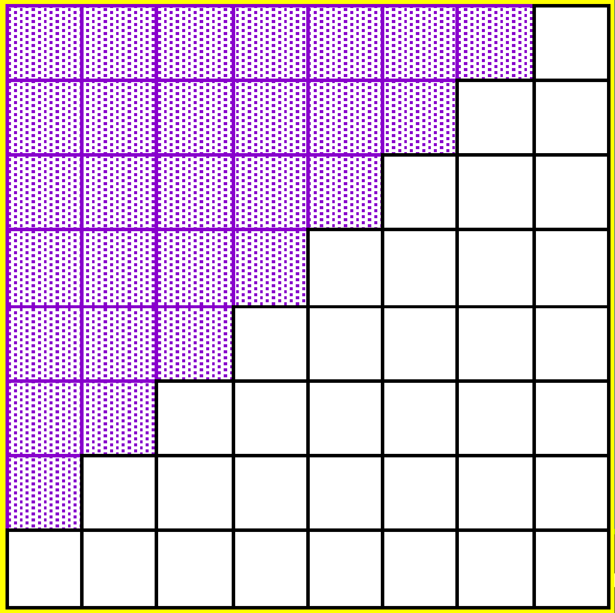
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

Parallel with the image plane



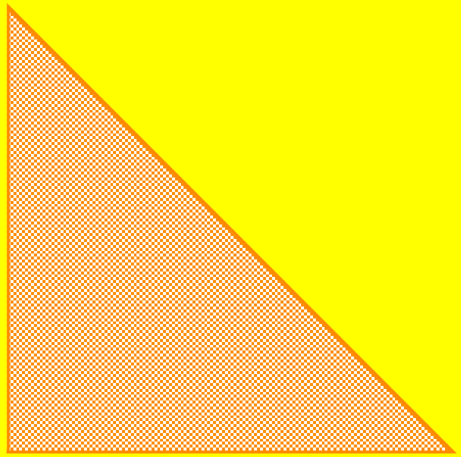
5	5	5	5	5	5	5
5	5	5	5	5	5	
5	5	5	5	5		
5	5	5	5			
5	5	5				
5	5					
5						

5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
5	5	5	∞	∞	∞	∞	∞
5	5	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞



Not Parallel

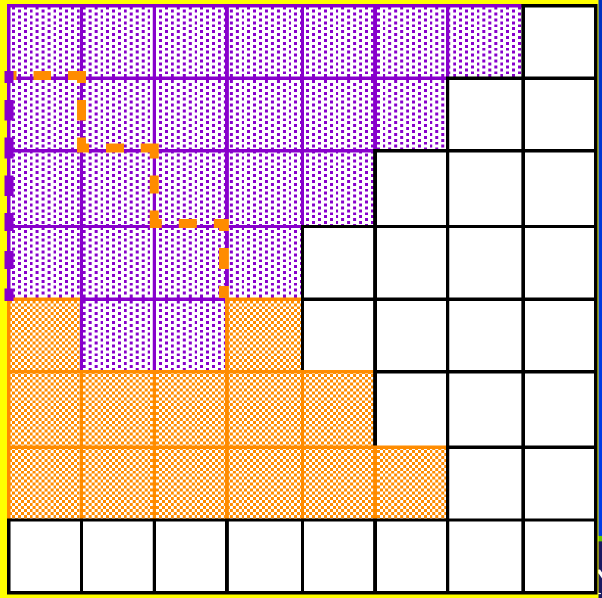
[0,6,7]



7					
6	7				
5	6	7			
4	5	6	7		
3	4	5	6	7	
2	3	4	5	6	7

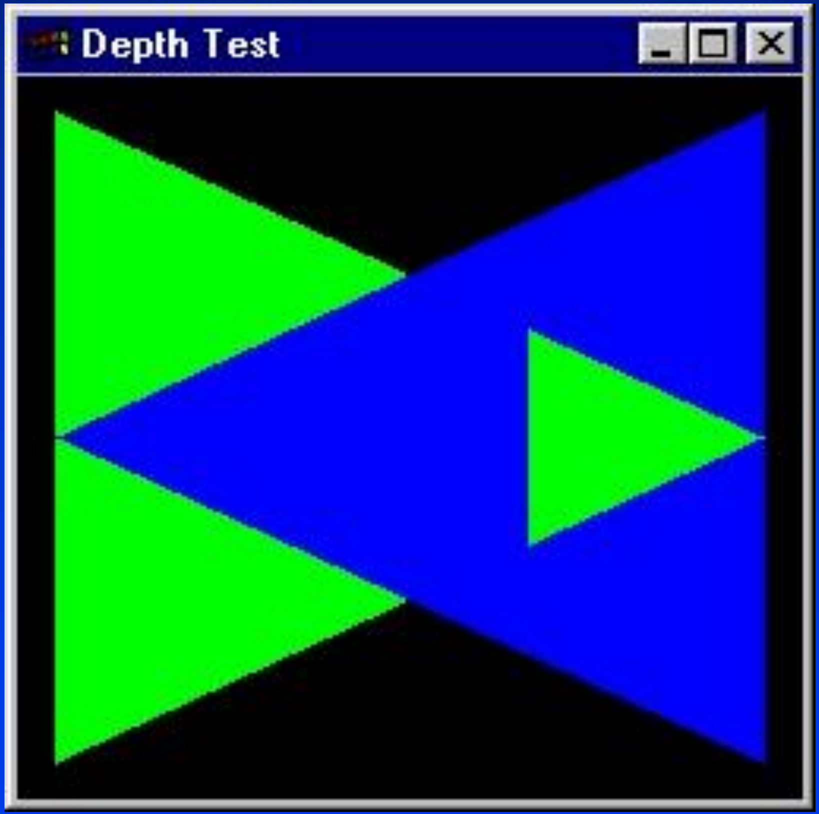
[0,1,2] [5,1,7]

5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
4	5	5	7	∞	∞	∞	∞
3	4	5	6	7	∞	∞	∞
2	3	4	5	6	7	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞



Z-Buffer Algorithm

- Algorithm easily handles this case

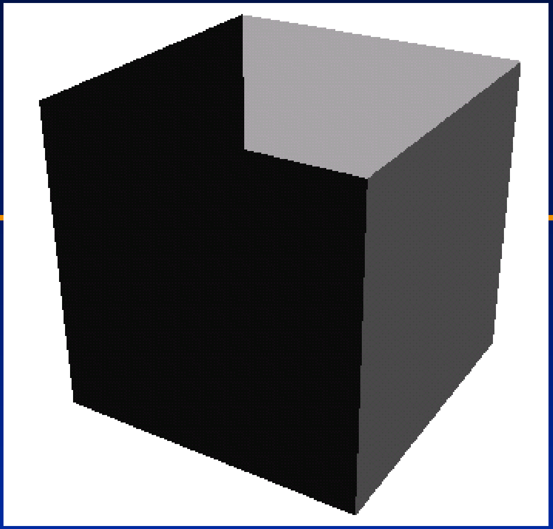


Z-buffering in OpenGL

- Create depth buffer by setting **`GLUT_DEPTH`** flag in **`glutInitDisplayMode()or the appropriate flag in the PIXELFORMATDESCRIPTOR`**

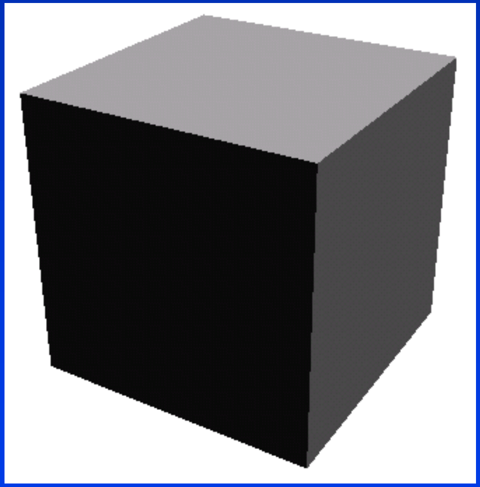
- Enable per-pixel depth testing with **`glEnable(GL_DEPTH_TEST)`**

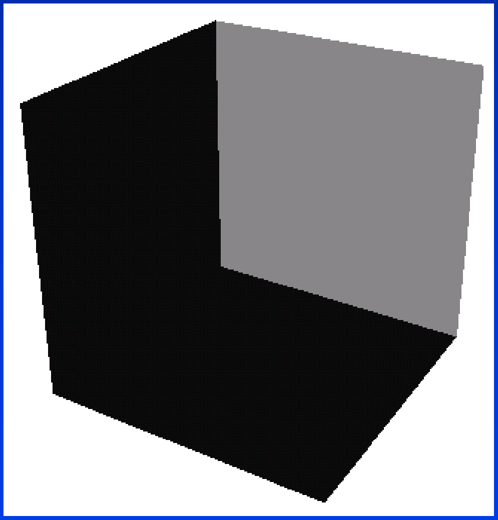
- Clear depth buffer by setting **`GL_DEPTH_BUFFER_BIT`** in **`glClear()`**



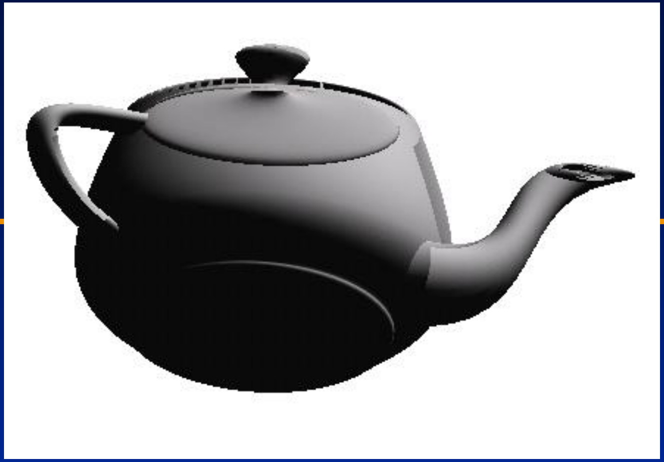
Without Hidden surface removal the wrong polygons can be drawn over

With Backface culling

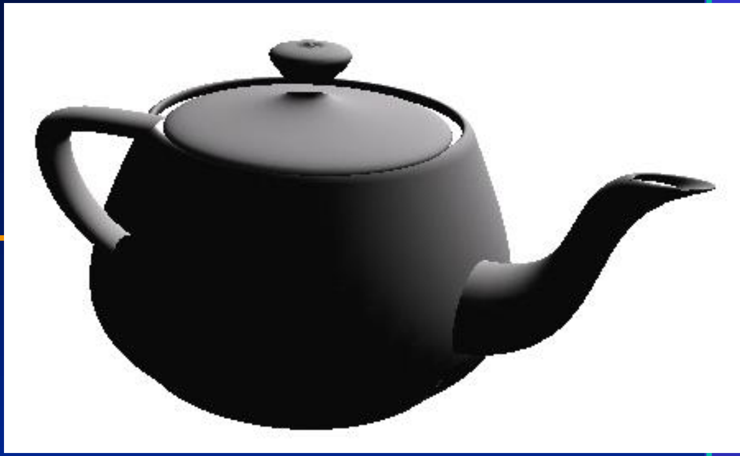




Alternatively we could cull the front faces to see inside the solid

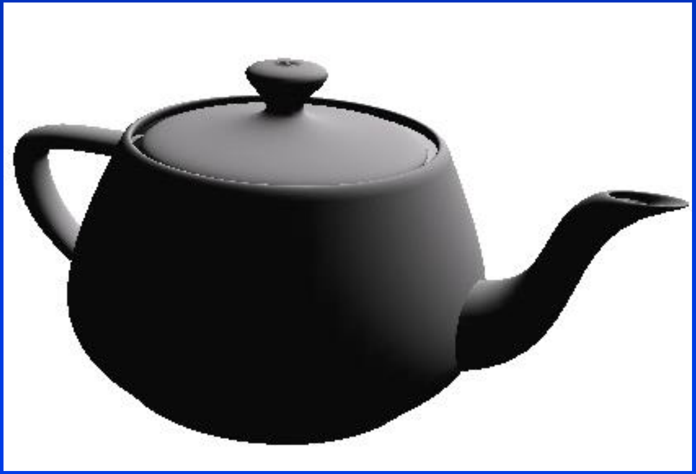


No hidden surface removal

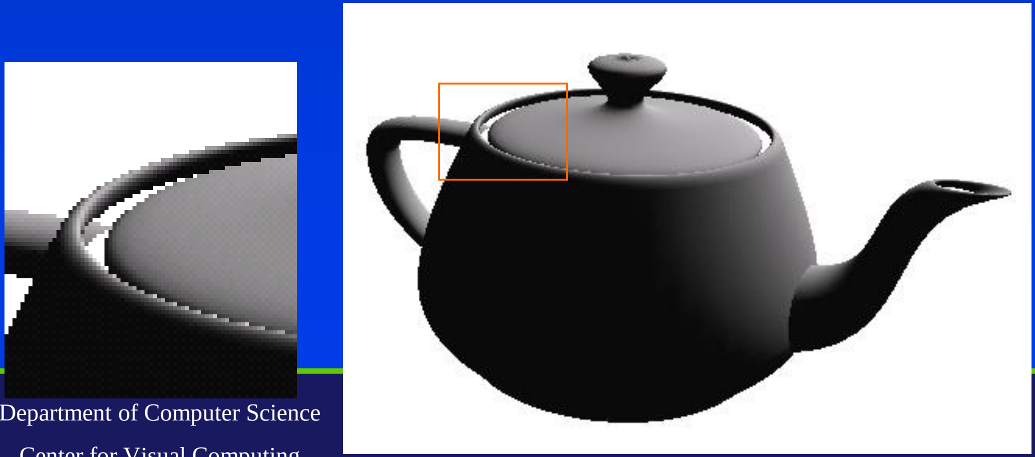


Backface Culling only: correct in some places but not adequate for objects which have holes, are non convex or multiple objects

Culling can reduce workload for depth testing but we need to ensure that objects are proper **solids**. This teapot is not quite a proper solid and as a result the image is incorrect. However, combining backface culling with more expensive depth-testing is usually a good practice.



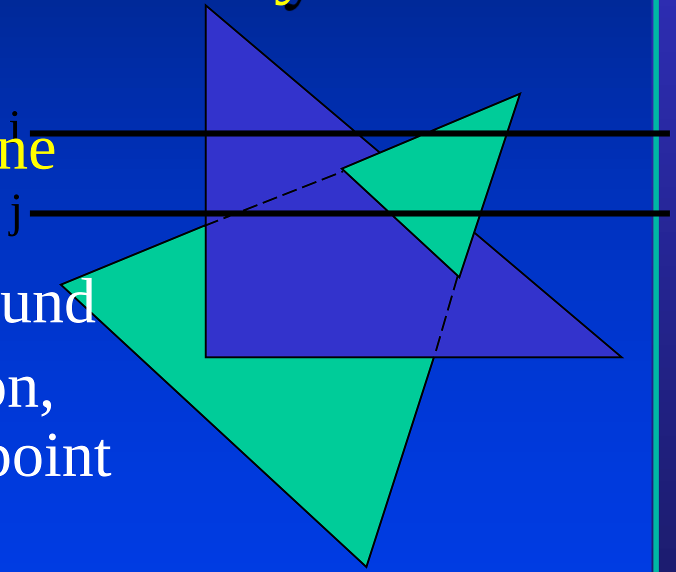
Depth Testing Only



Scan Line Algorithm

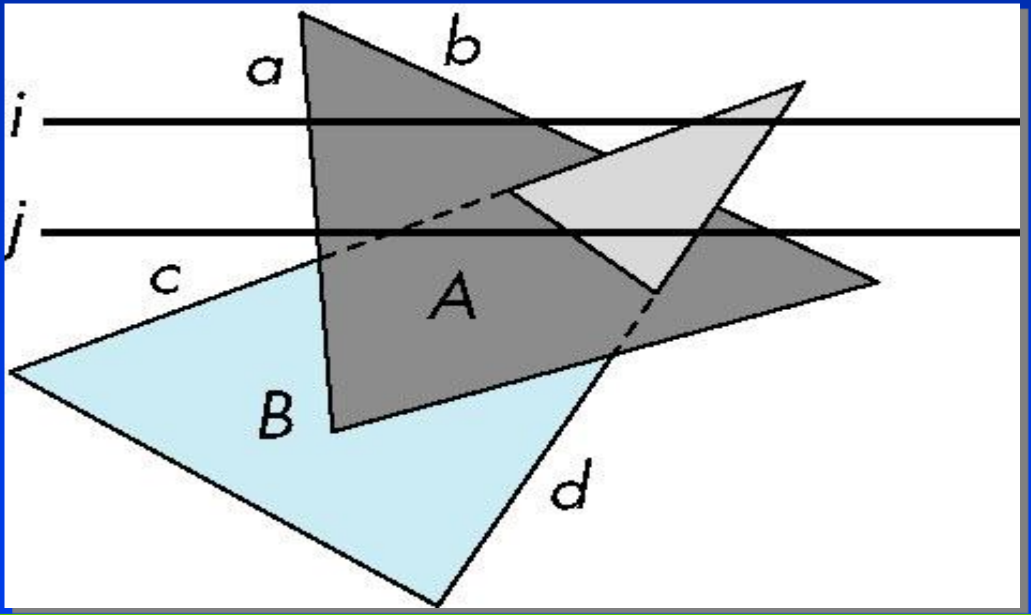
- Similar in some respects to the z-buffer method but handles the image scan-line by scan-line
- Due to coherency in data, this can be relatively efficient.

1. Rasterize all polygon boundaries (edges)
2. Scanning across each scan line we determine the color of each pixel
   a. By default color everything as background
   b. If we encounter the edge of one polygon, start evaluating polygon color at each point and shade the scanline it accordingly
   c. For multiple edges do depth evaluation to see which polygon "wins"



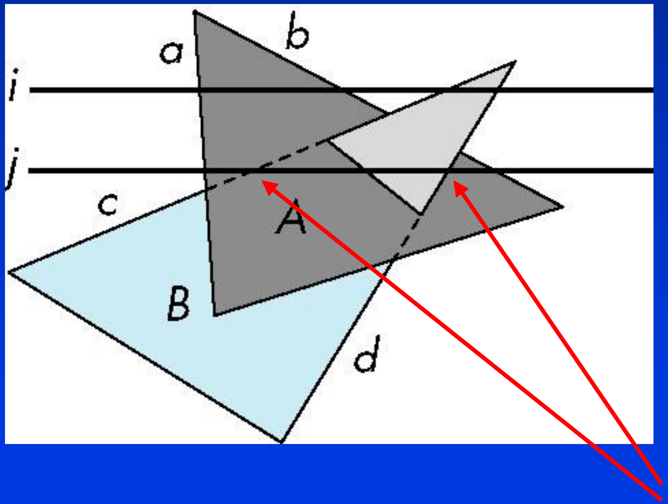
Scan Line Algorithm

- Work one scan line at a time
- Compute intersections of faces along scanline
- Keep track of all "open segments" and draw the closest



Scan Line Algorithm

- Can combine shading and hsr through scan line algorithm



scan line i: no need for depth information, can only be in no or one polygon

scan line j: need depth information only when in more than one polygon

Scan Conversion

- At this point in the pipeline, we have only polygons and line segments. Render!
- To render, convert to pixels (“fragments”) with integer screen coordinates (ix, iy), depth, and color
- Send fragments into fragment-processing pipeline

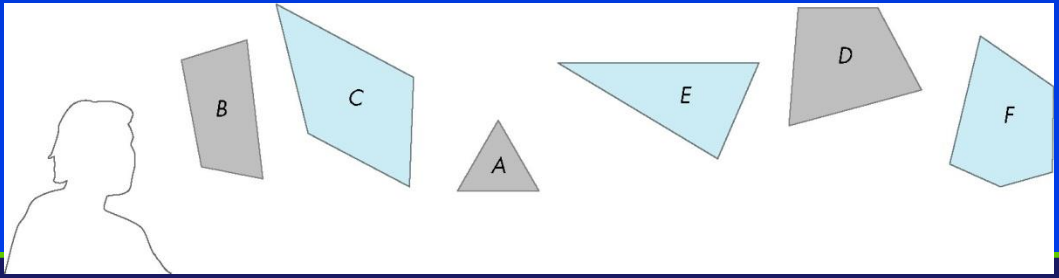
Hidden Surface Removal

- Object-space vs. Image space
- The main image-space algorithm: z-buffer
- Drawbacks
  - Aliasing
  - Rendering invisible objects

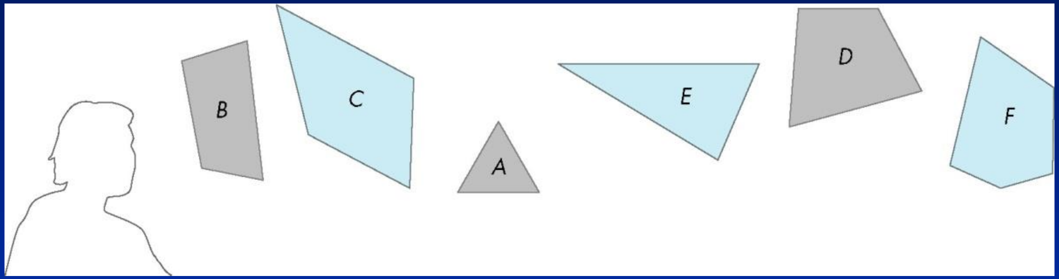
- How would *object-space hidden surface removal* work?

Visibility Testing

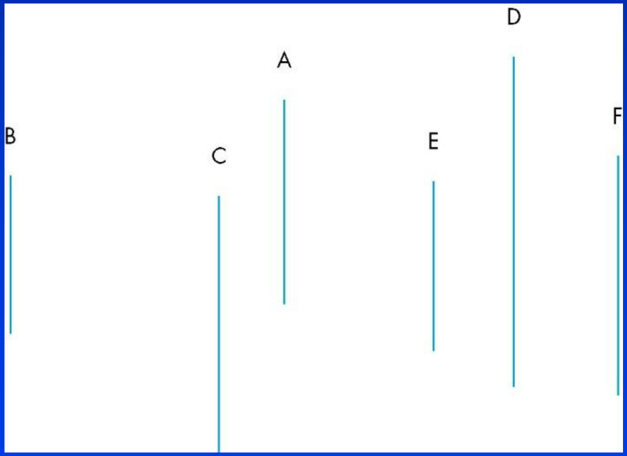
- In many real-time applications, such as games, we want to eliminate as many objects as possible within the application
  - –Reduce burden on pipeline
  - –Reduce traffic on bus
- Partition space with Binary Spatial Partition (BSP) Tree



Simple Example



consider 6 parallel polygons



top view

The plane of A separates B and C from D, E and F

Painter’s Algorithm with BSP Trees

- Building the tree
  - May need to split some polygons
  - Slow, but done only once
- Traverse back-to-front or front-to-back
  - Order is viewer-direction dependent
  - However, the tree is viewer-independent
  - What is front and what is back of each line changes
  - Determine order on the fly

Divide Scene with a Plane

- Everything on the same side of that plane as the eye is in front of everything else.
- Divide front and back with more planes.
- If necessary split polygons by planes.

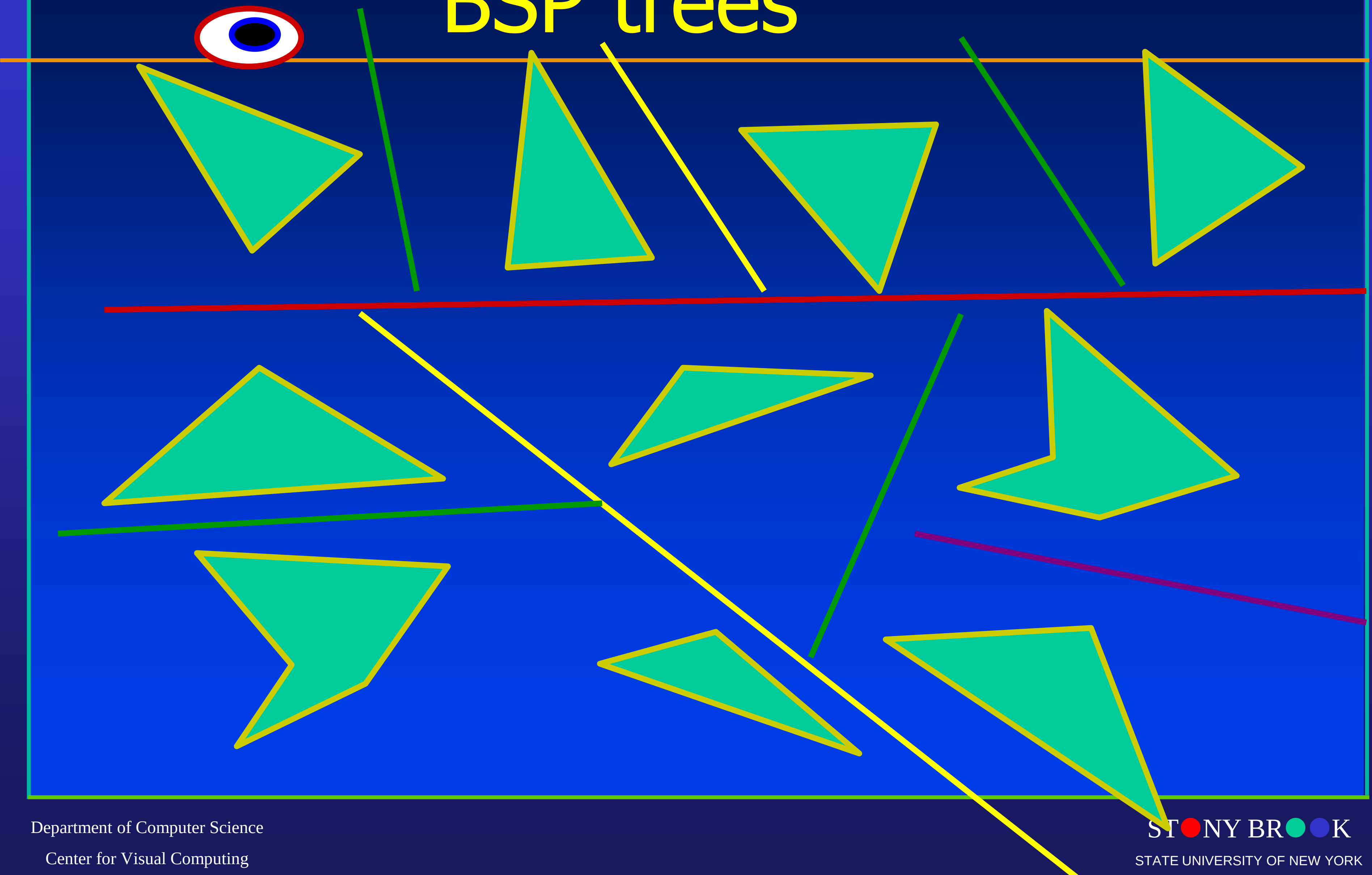
Details of Painter’s Algorithm

- Each face has form Ax + By + Cz + D = 0
- Plug in coordinates and determine
  - Positive: front side
  - Zero: on plane
  - Negative: back side
- **Back-to-front**: inorder traversal, farther child first
- **Front-to-back**: inorder traversal, near child first
- Do backface culling with same sign test
- Clip against visible portion of space (portals)

Binary Space Partition Trees(1979)

- BSP tree: organize all of space (hence *partition*) into a binary tree
  - *Preprocess*: overlay a binary tree on objects in the scene
  - *Runtime*: correctly traversing this tree enumerates objects from back to front
  - Idea: divide space recursively into half-spaces by choosing *splitting planes*
    - Splitting planes can be arbitrarily oriented

BSP trees

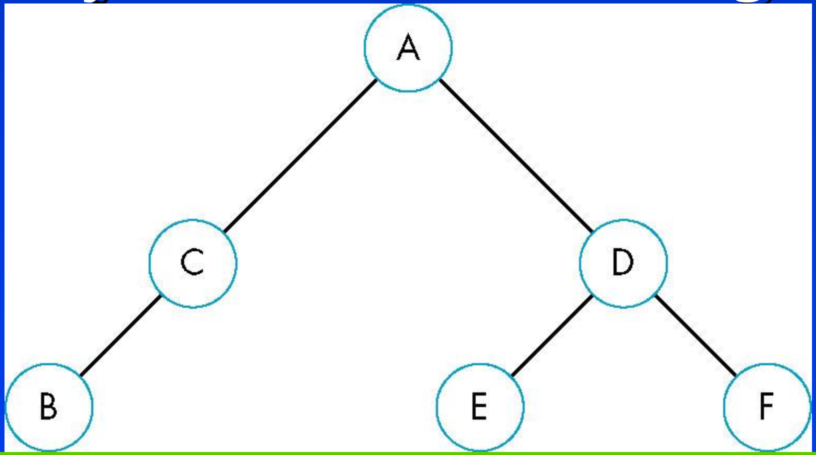


BSP Tree

- Split space with any line (2D) or plane (3D)
- Applications
  - Painters algorithm for hidden surface removal
  - Ray casting
  - Solid modeling
- Inherent spatial ordering given viewpoint
  - Left subtree: in front
  - right subtree: behind
- Problem: finding good space partitions
  - Proper ordering for the tree
  - Balance tree

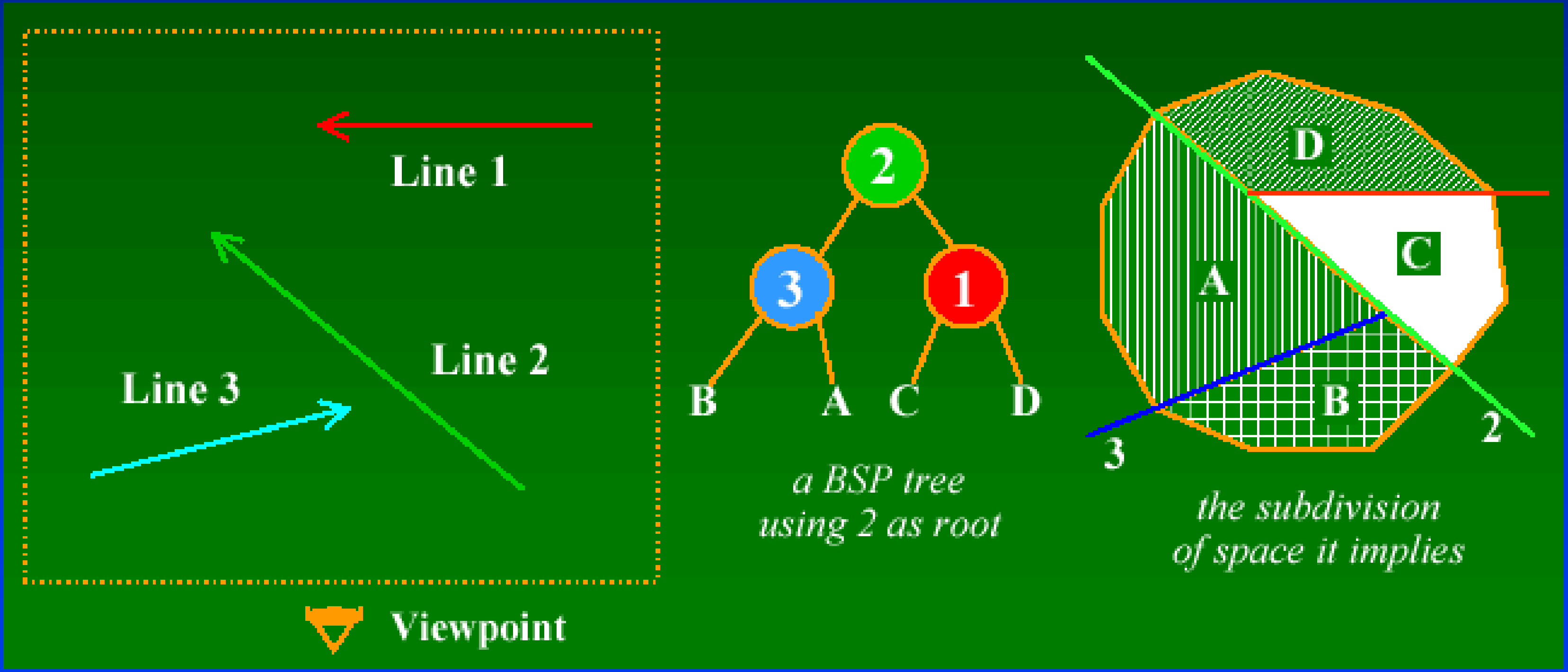
BSP Tree

- Can continue recursively
  - –Plane of C separates B from A
  - –Plane of D separates E and F
- Can put this information in a BSP tree
  - –Use for visibility and occlusion testing



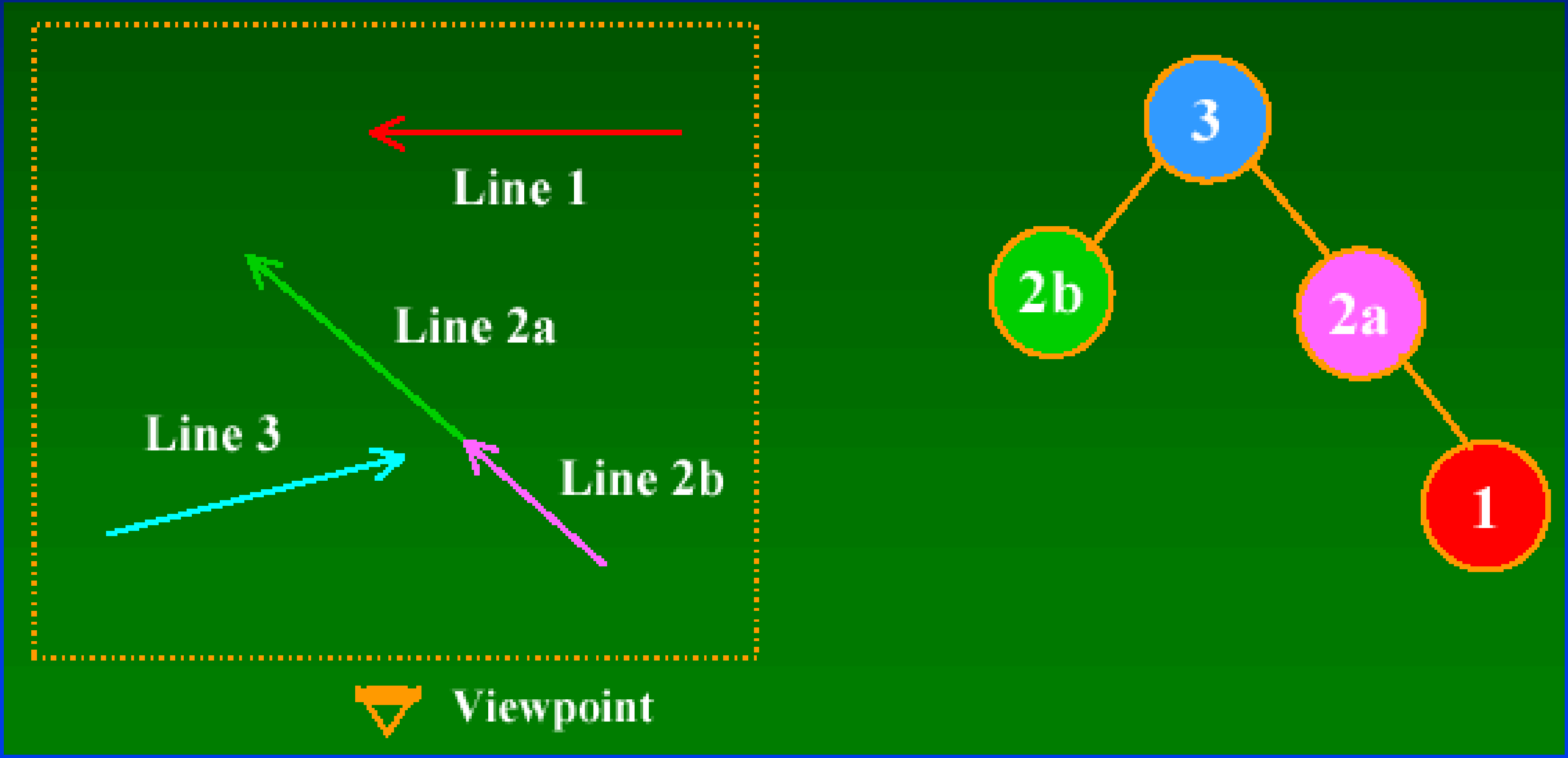
Building BSP Trees

- Use hidden surface removal as intuition
- Using line 1 or line 2 as root is easy


Line 1
Line 2
Line 3
Viewpoint
2
3
1
B
A
C
D
a BSP tree using 2 as root
D
C
A
B
2
3
the subdivision of space it implies

Building BSP Trees

- Using line 3 as root requires splitting

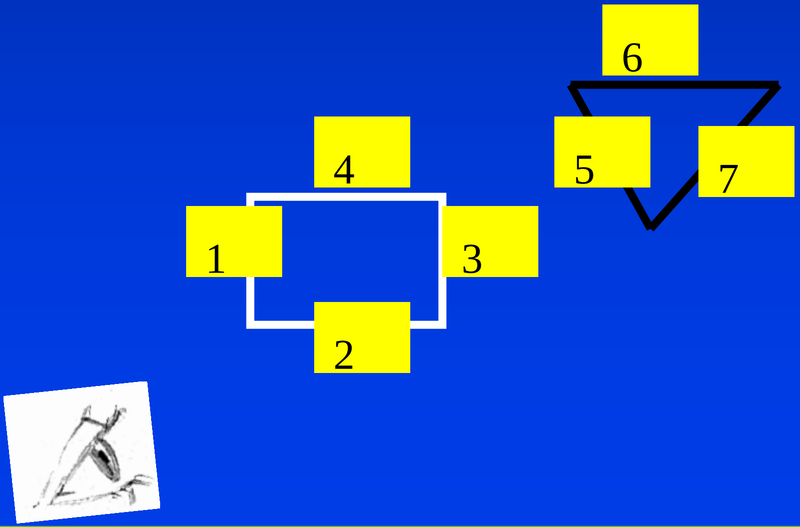


Building a Good Tree

- Naive partitioning of n polygons yields O(n ^{3}) polygons (in 3D)
- Algorithms with O(n 2) increase exist
  - Try all, use polygon with fewest splits
  - Do not need to split exactly along polygon planes
- Should balance tree
  - More splits allow easier balancing
  - Rebalancing?

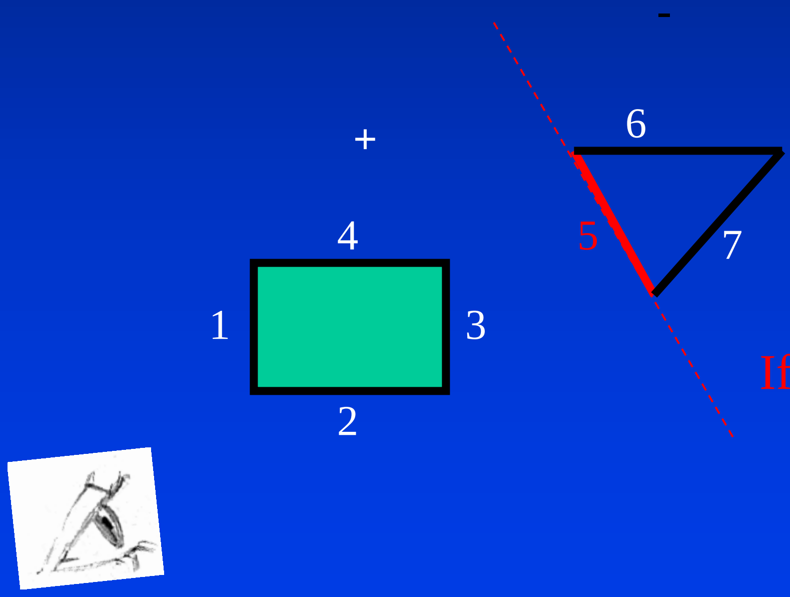
Binary Space Partitioning Trees

- Basic idea: Objects in the half space opposite of the viewpoint do not obscure objects in the half space containing the viewpoint; thus, one can safely render them without covering foreground objects



Binary Space Partitioning Trees

- Basic Idea: Objects in the half space opposite of the viewpoint do not obscure objects in the half space containing the viewpoint; thus, one can safely render them without covering foreground objects



If we want to draw 5 correctly

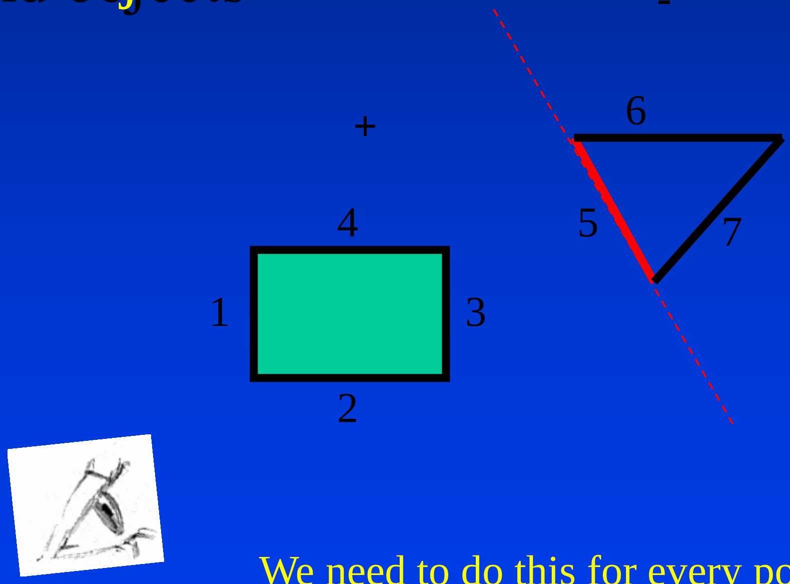
- we need draw 6 and 7 first,

- then draw 5,

- then draw 1,2,3,4

Binary Space Partitioning Trees

- Basic principle: Objects in the half space opposite of the viewpoint do not obscure objects in the half space containing the viewpoint; thus, one can safely render them without covering foreground objects



If we want to draw 5 correctly

- we need draw 6 and 7 first,
- then draw 5,
- then draw 1,2,3,4

We need to do this for every polygon

Can we do this efficiently?

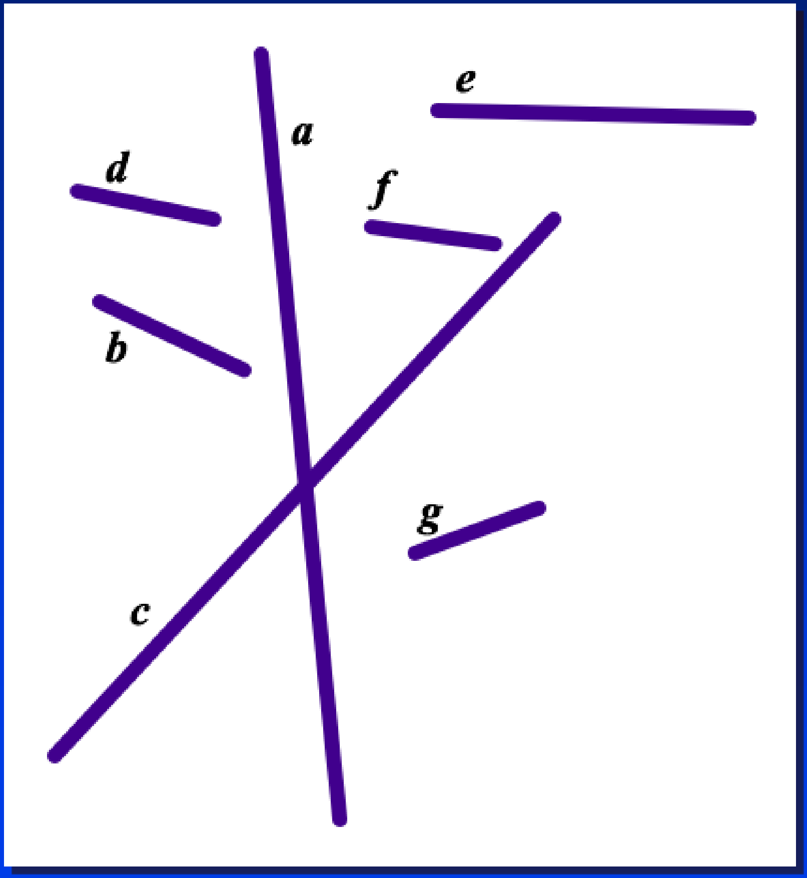
Binary Space Partition Trees

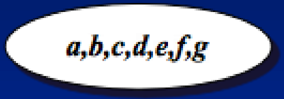
- BSP tree: organize all of space (hence *partition*) into a binary tree
  - *Preprocess*: overlay a binary tree on objects in the scene
  - *Runtime*: correctly traversing this tree enumerates objects from back to front
  - Idea: divide space recursively into half-spaces by choosing *splitting planes*
    - Splitting planes can be arbitrarily oriented

BSP-Trees

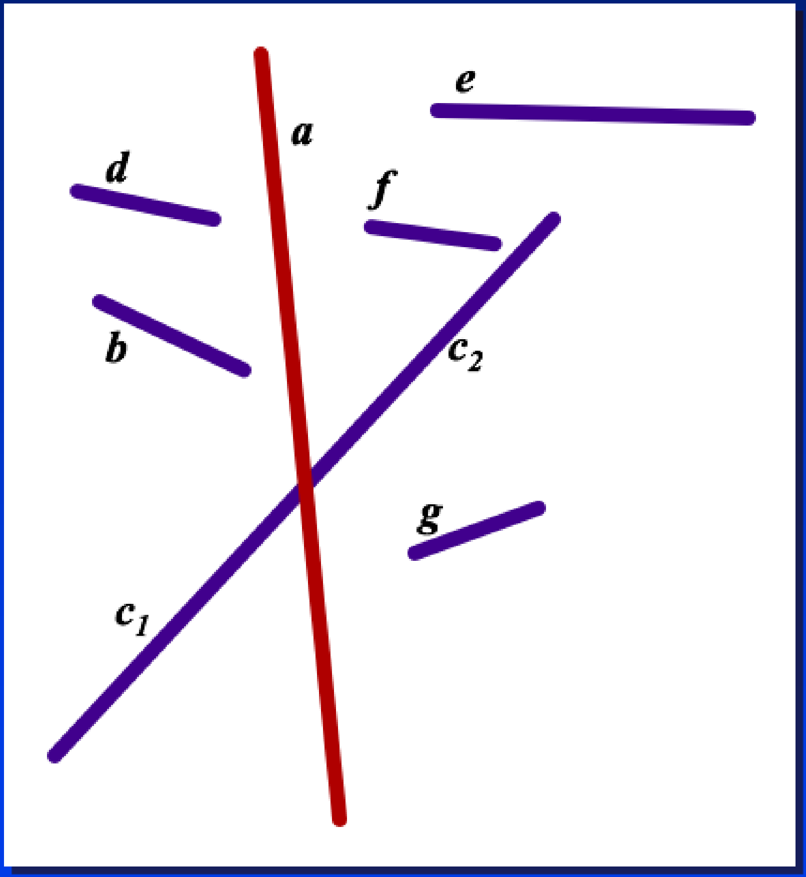
- Binary Space Partition Trees
  - Split space along planes
  - Allows fast queries of some spatial relations
- Simple construction algorithm
  - Select a plane as sub-tree root
  - Everything on one side to one child
  - Everything on the other side to other child
  - Use random polygon for splitting plane

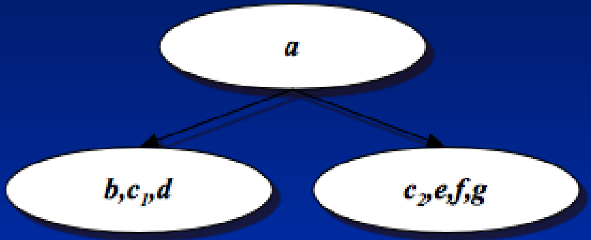
BSP-Trees



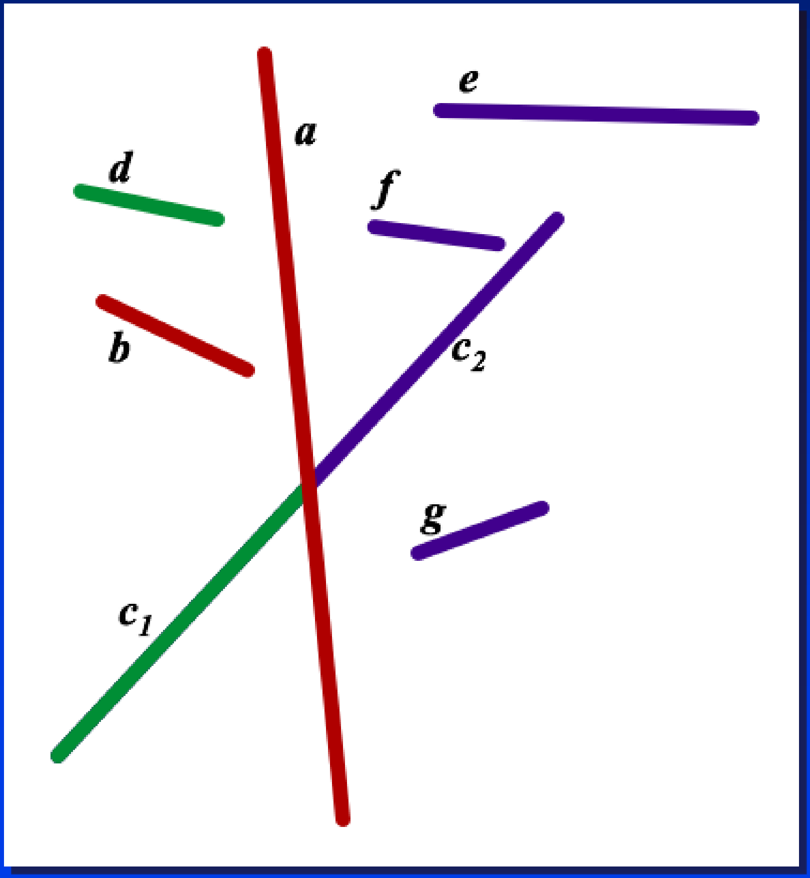


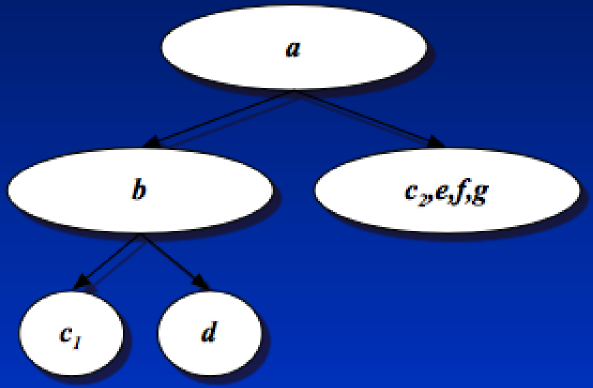
BSP-Trees



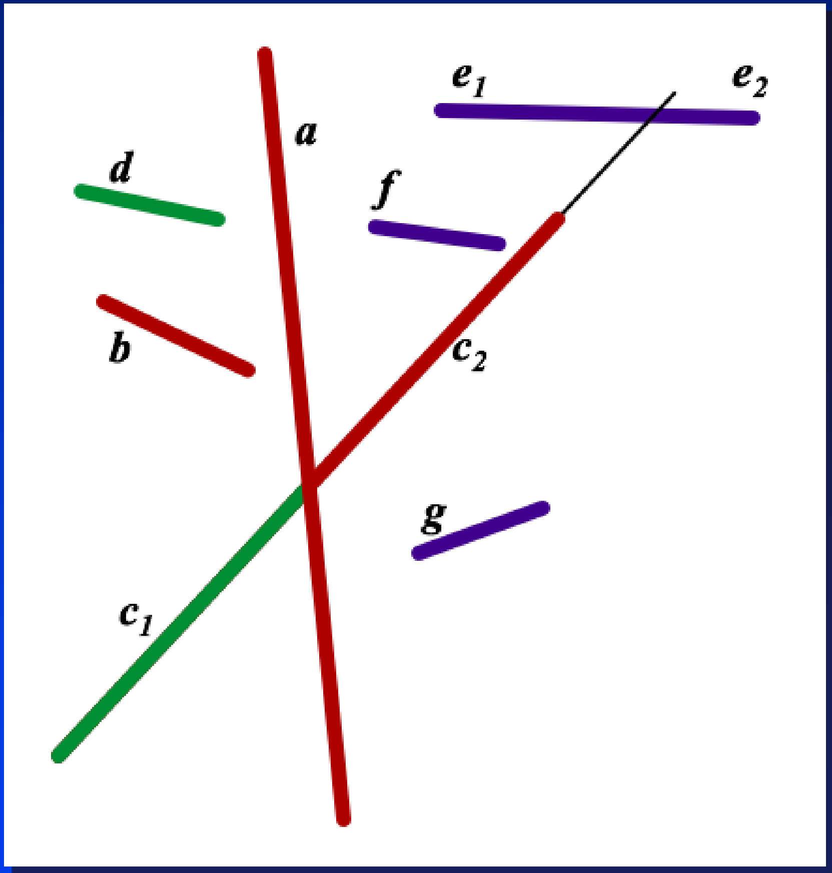


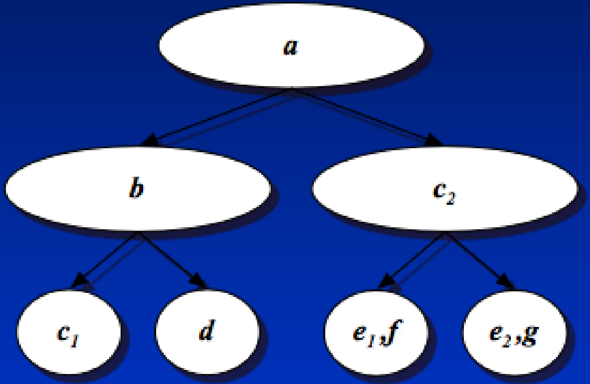
BSP-Trees



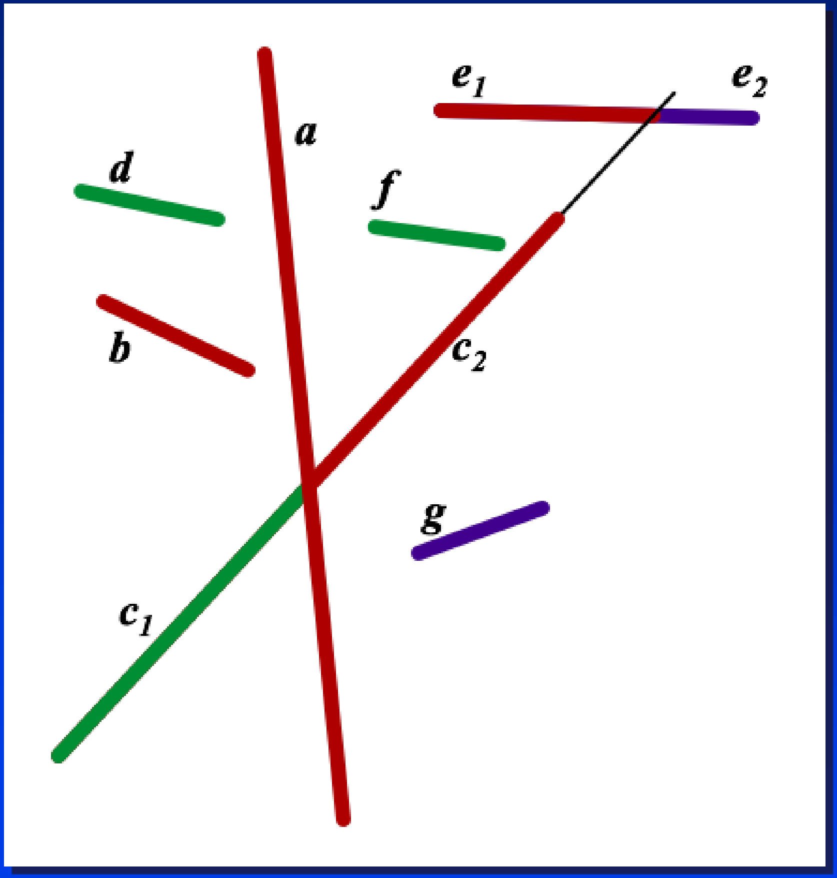

$$\begin{tikzcd}
 & a \arrow[dl] \arrow[dr] & \\
b \arrow[d] \arrow[dr] & & c_2,e,f,g \\
c_1 & d &
\end{tikzcd}$$

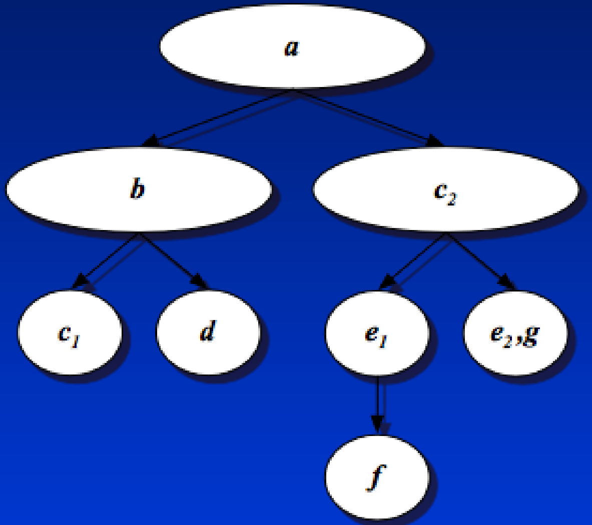
BSP-Trees



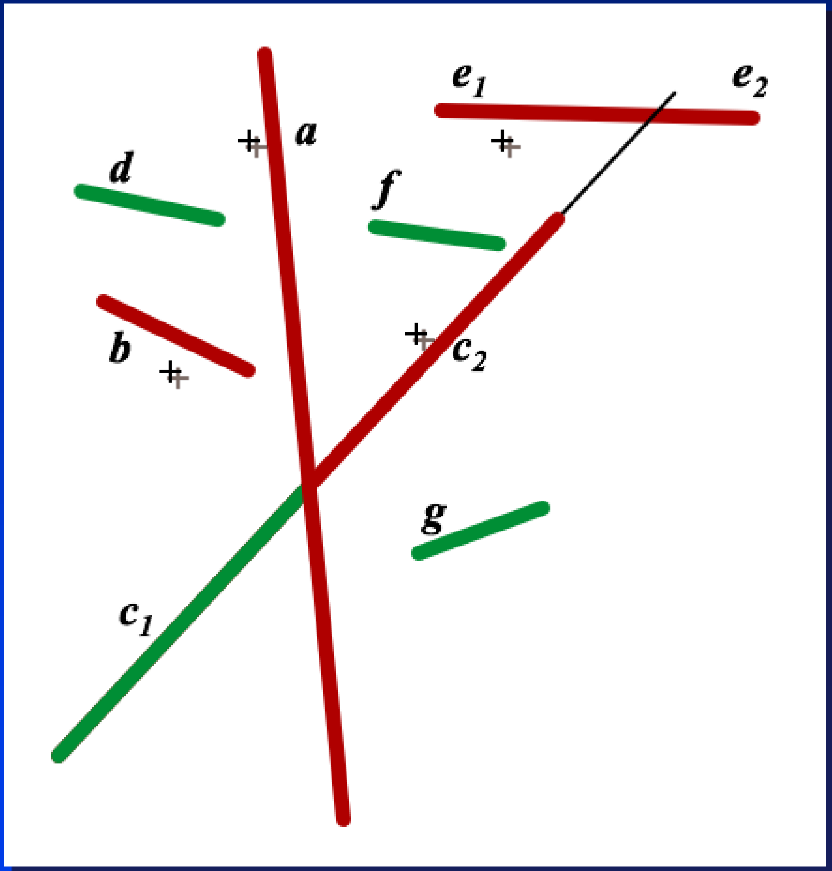

$$\begin{tikzcd}[column sep=small]
 & & & a \arrow[dll] \arrow[drr] & & & \\
 & b \arrow[dl] \arrow[dr] & & & & c_2 \arrow[dl] \arrow[dr] & \\
c_1 & & d & & e_1,f & & e_2,g
\end{tikzcd}$$

BSP-Trees




$$\begin{tikzcd}[column sep=small]
 & & & a \arrow[dll] \arrow[drr] & & & \\
 & b \arrow[dl] \arrow[dr] & & & & c_2 \arrow[dl] \arrow[dr] & \\
c_1 & & d & & e_1 \arrow[d] & & e_2,g \\
 & & & & f & &
\end{tikzcd}$$

BSP-Trees



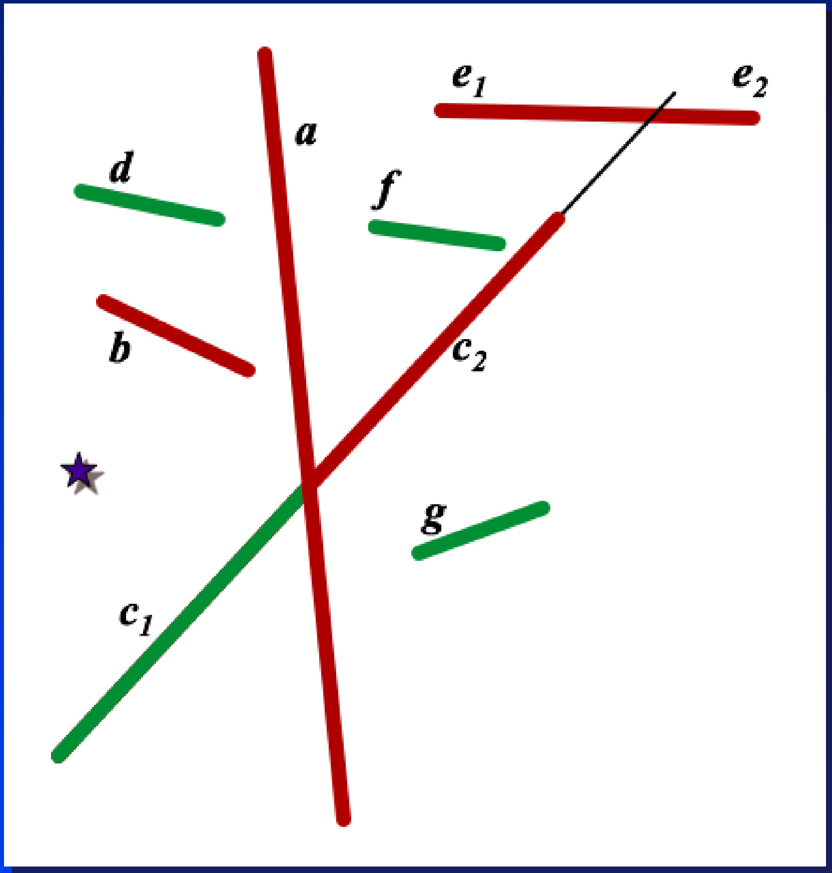
$$\begin{tikzcd}
 & & & a \arrow[dll, "+"'] \arrow[drr, "-"] & & & \\
 & b \arrow[dl, "+"'] \arrow[dr, "-"] & & & & c_2 \arrow[dl, "+"'] \arrow[dr, "-"] & \\
c_1 & & d & & e_1 \arrow[d, "+"'] & & e_2 \arrow[d, "+"'] \\
 & & & & f & & g
\end{tikzcd}$$

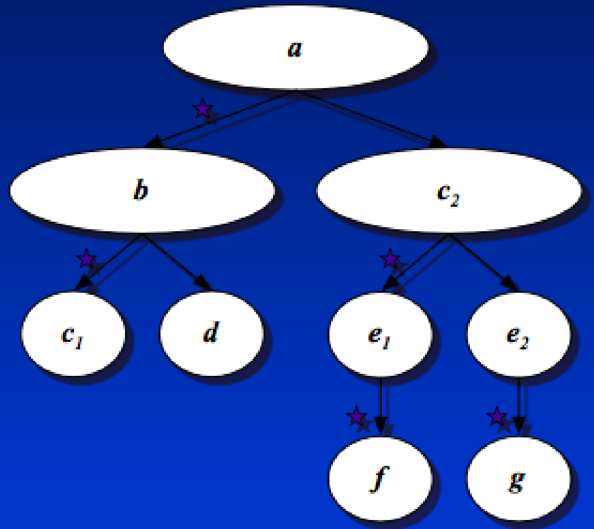
How Do We Traverse BSP Tree?

- Visibility traversal based on BSP tree
  - Variation of in-order-traversal
    - »Child one
    - »Sub-tree root
    - »Child two
- Select "child one" based on location of viewpoint

– Child one on same side of sub-tree root as viewpoint

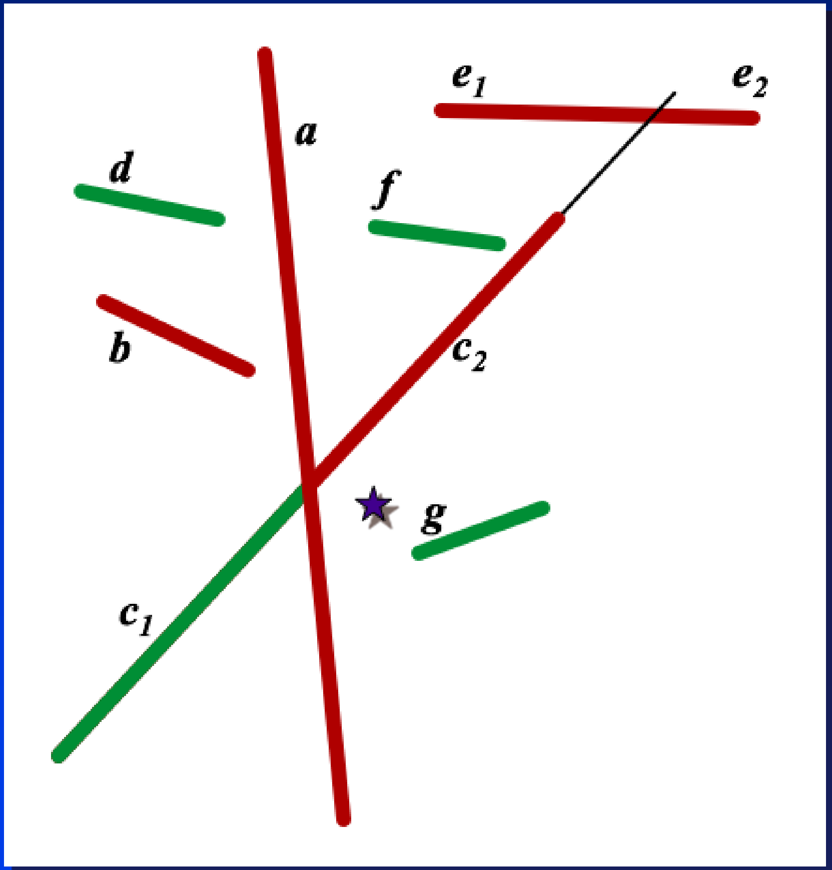
BSP-Trees

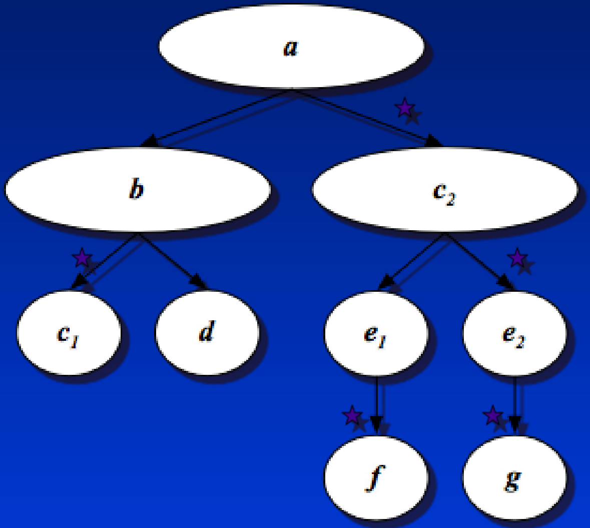




$c_1{:}b{:}d{:}a{:}f{:}e_1{:}c_2{:}g{:}e_2$

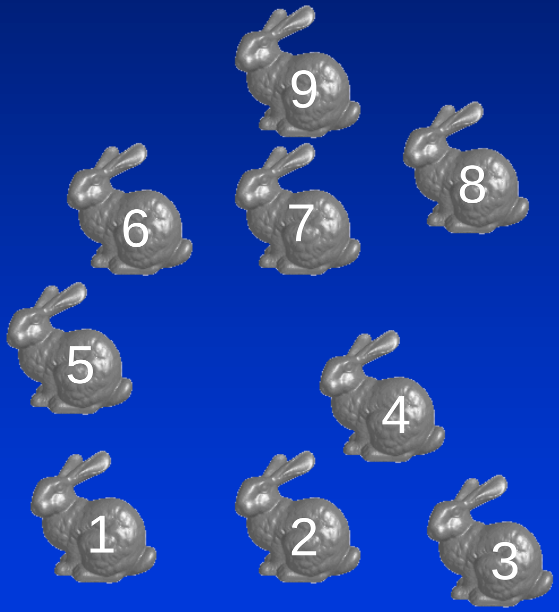
BSP-Trees



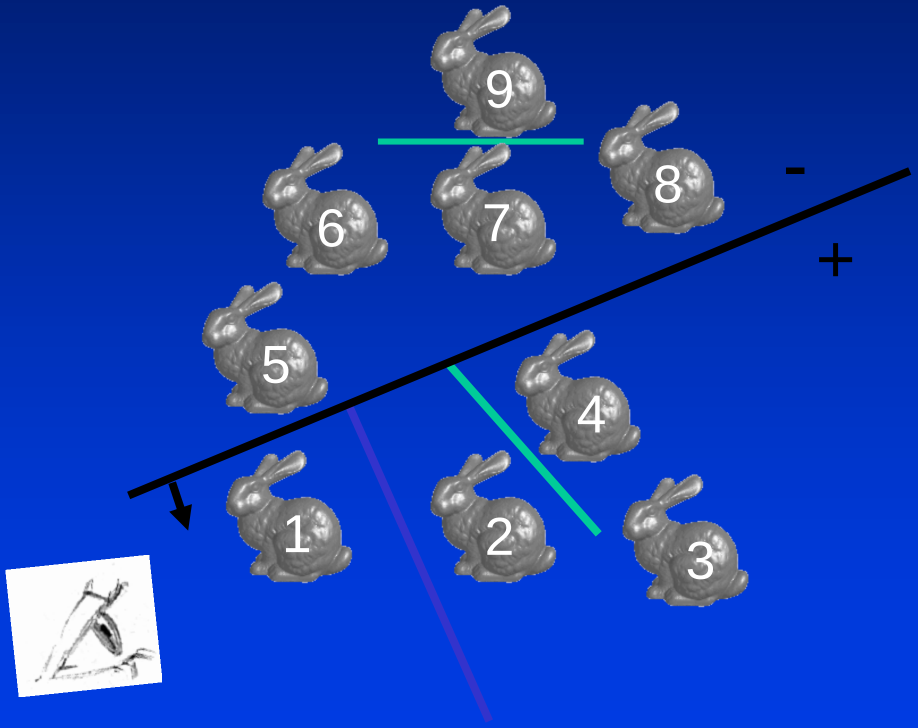


$g{:}e_2{:}c_2{:}f{:}e_1{:}a{:}c_1{:}b{:}d$

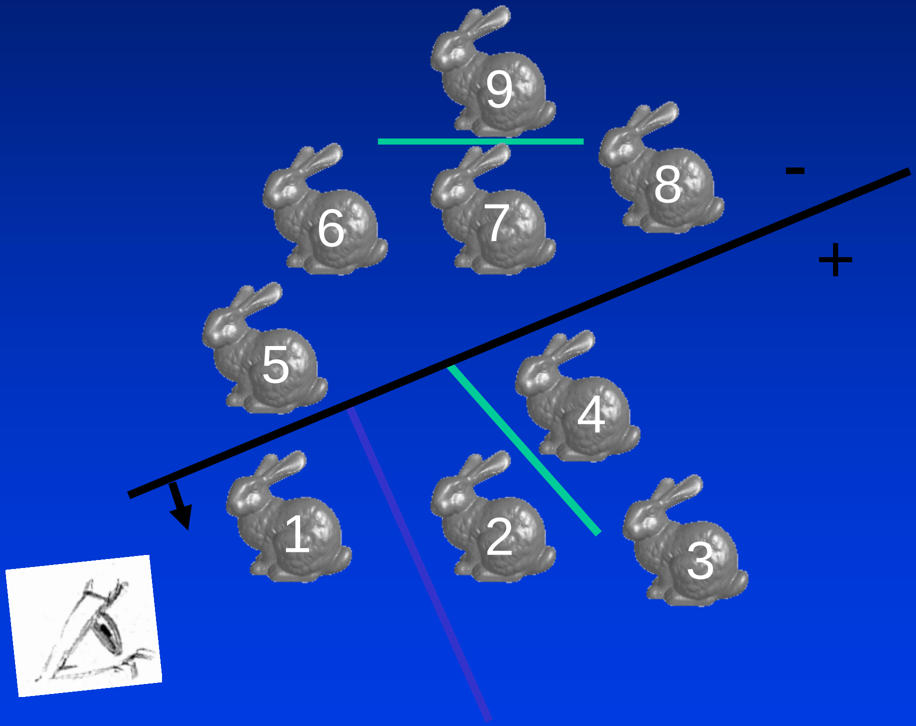
BSP Trees: Another Example



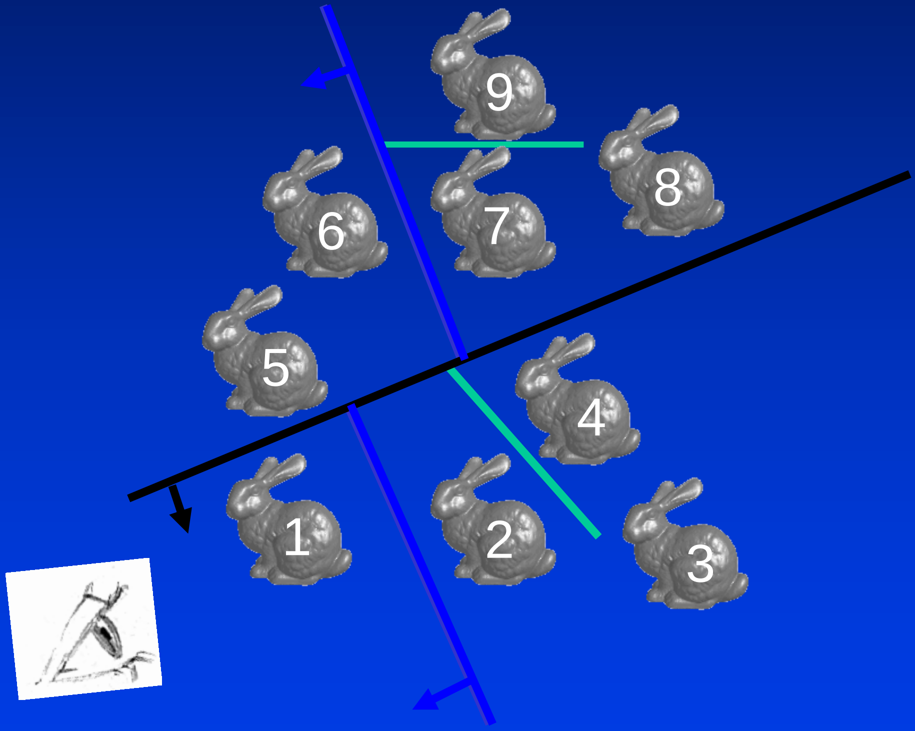
BSP Trees: Objects



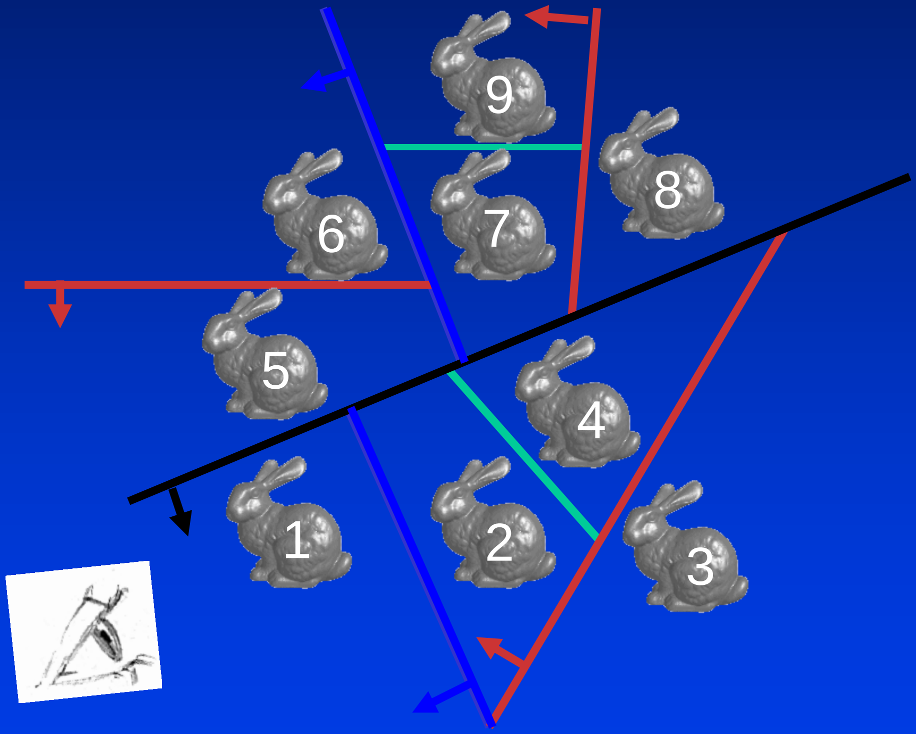
BSP Trees: Objects



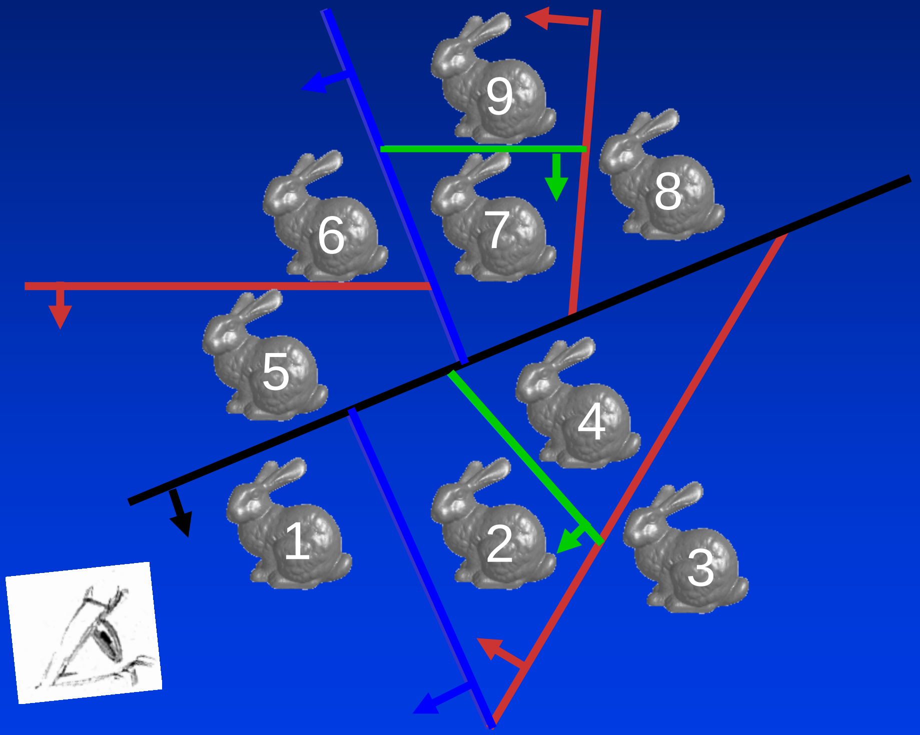
BSP Trees: Objects



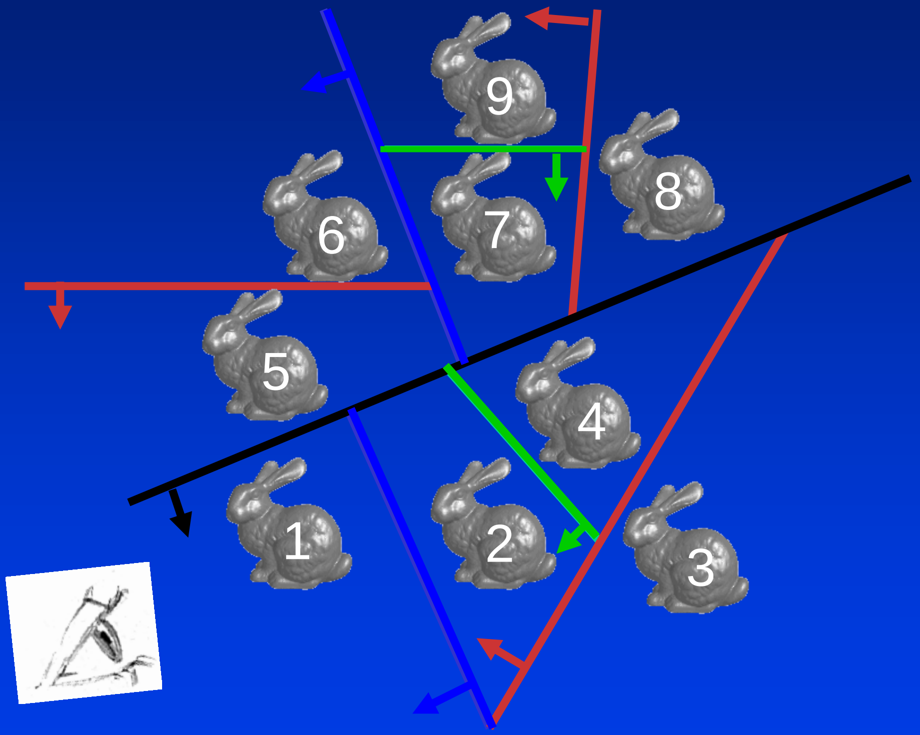
BSP Trees: Objects



BSP Trees: Objects



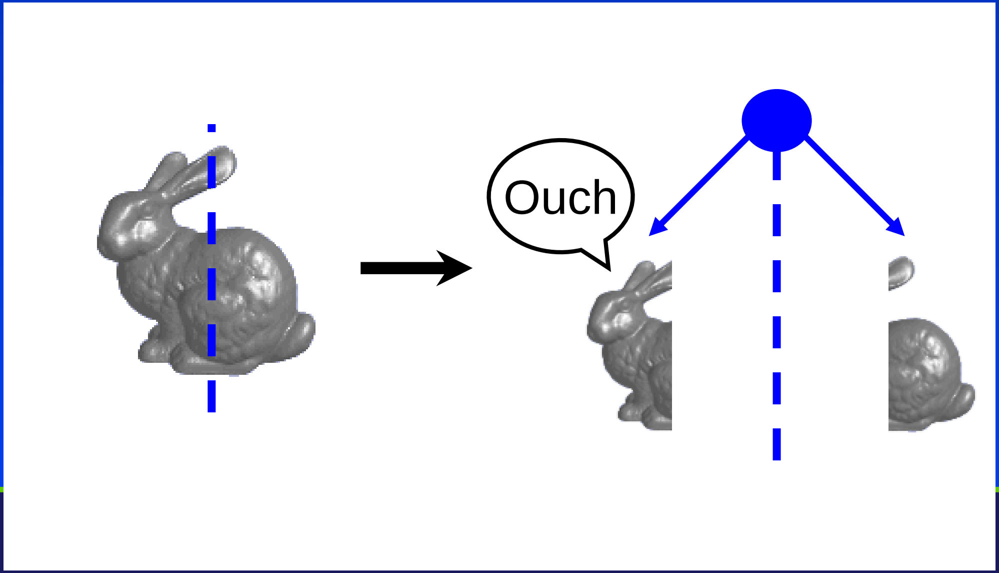
BSP Trees: Objects



When to stop the recursion?

Object Splitting

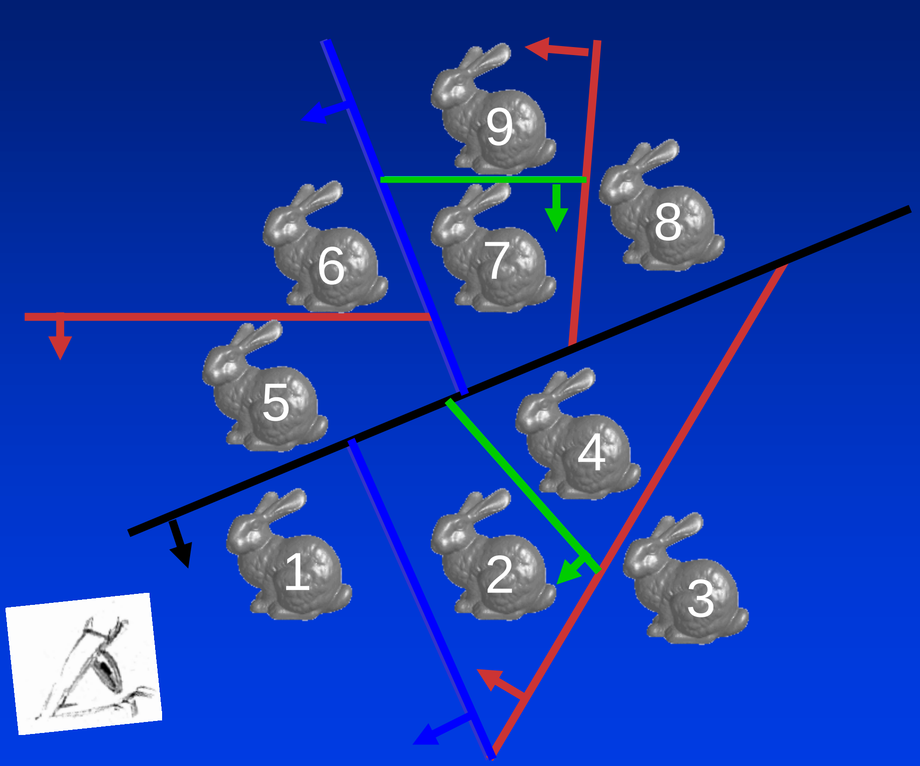
- No bunnies were harmed in my example
- But what if a splitting plane passes through an object?
  - Split the object; give half to each node:
  - Worst case: can create up to $O(n^3)$ objects!

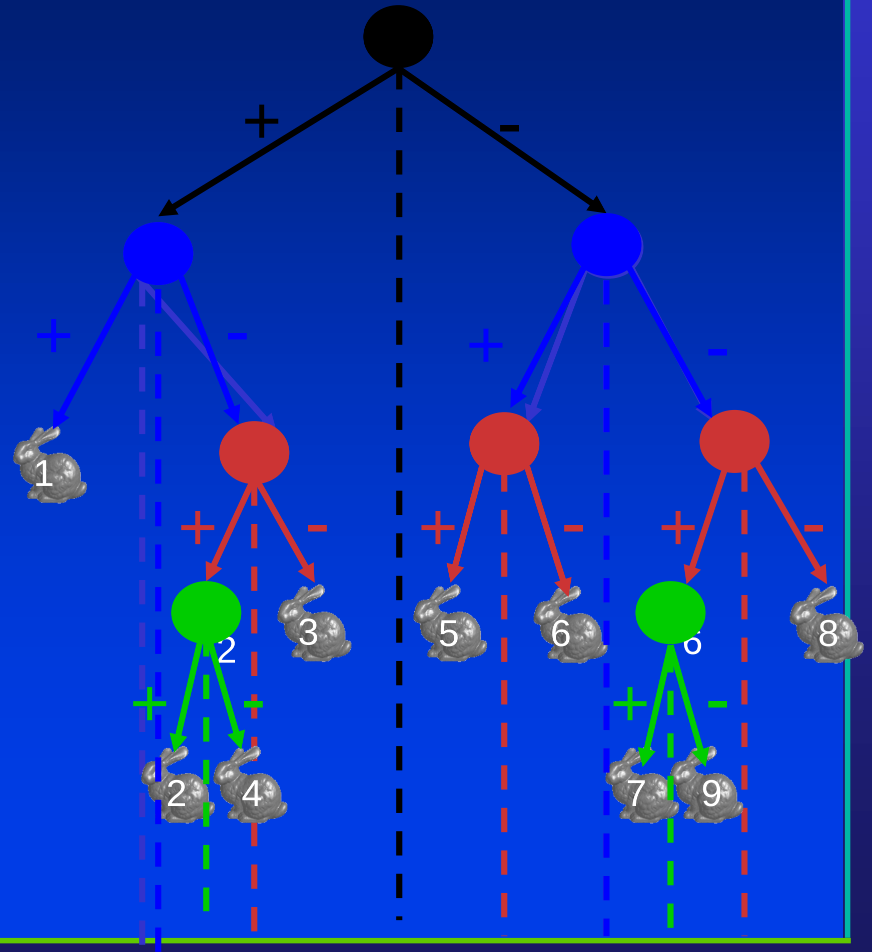


Polygons: BSP Tree Construction

- Split along the plane containing any polygon
- Classify all polygons into positive or negative half-space of the plane
  - If a polygon intersects plane, split it into two
- Recursion down the negative half-space
- Recursion down the positive half-space

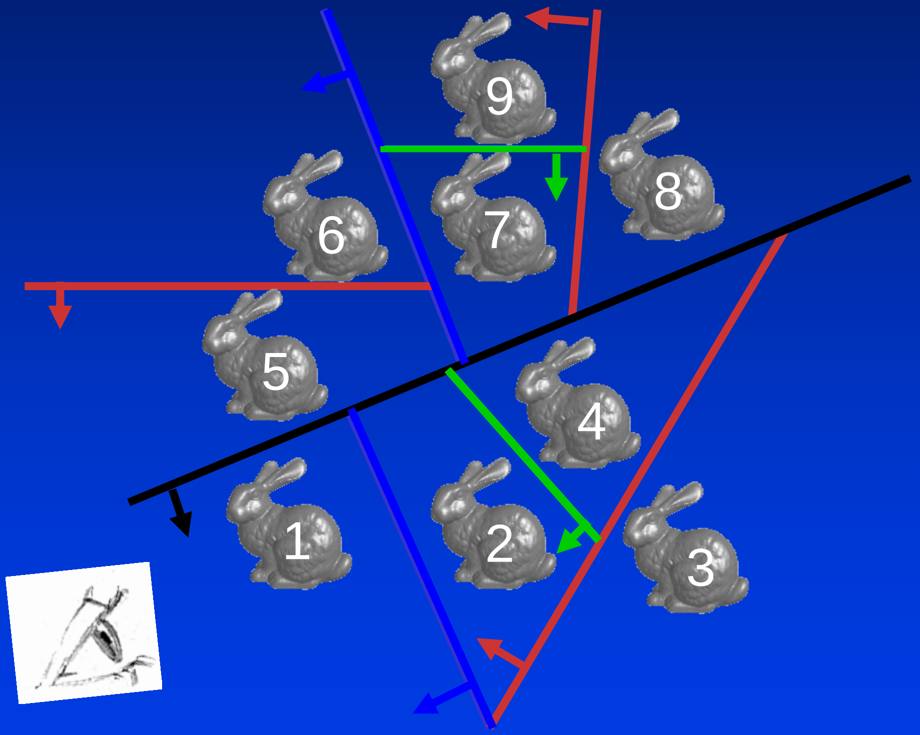
BSP Trees: Objects

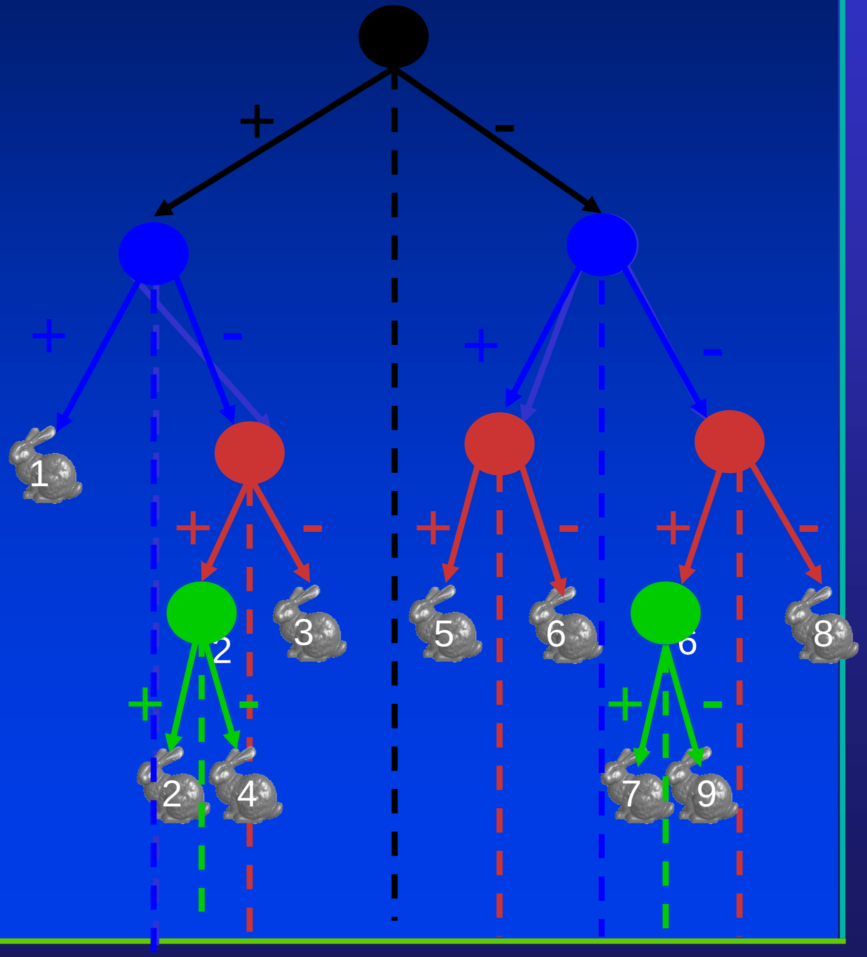




Traversal order?

BSP Trees: Objects



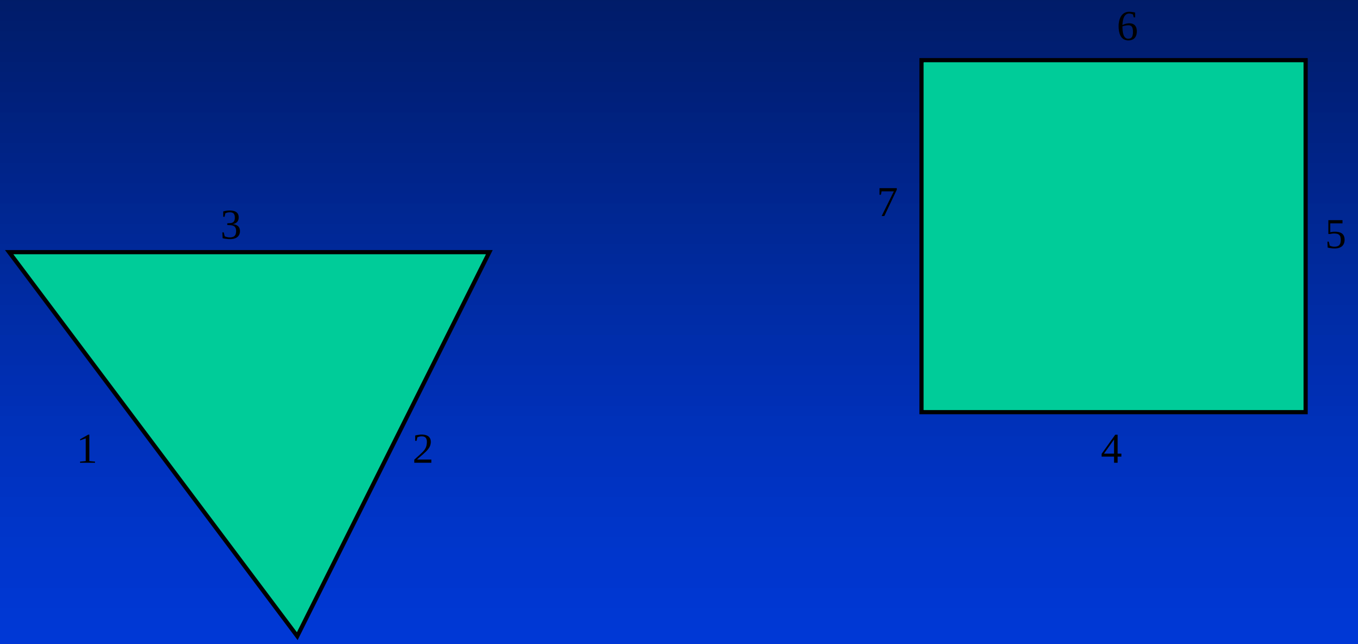


Traversal order:
8->9->7->6->5->3->4->2->1

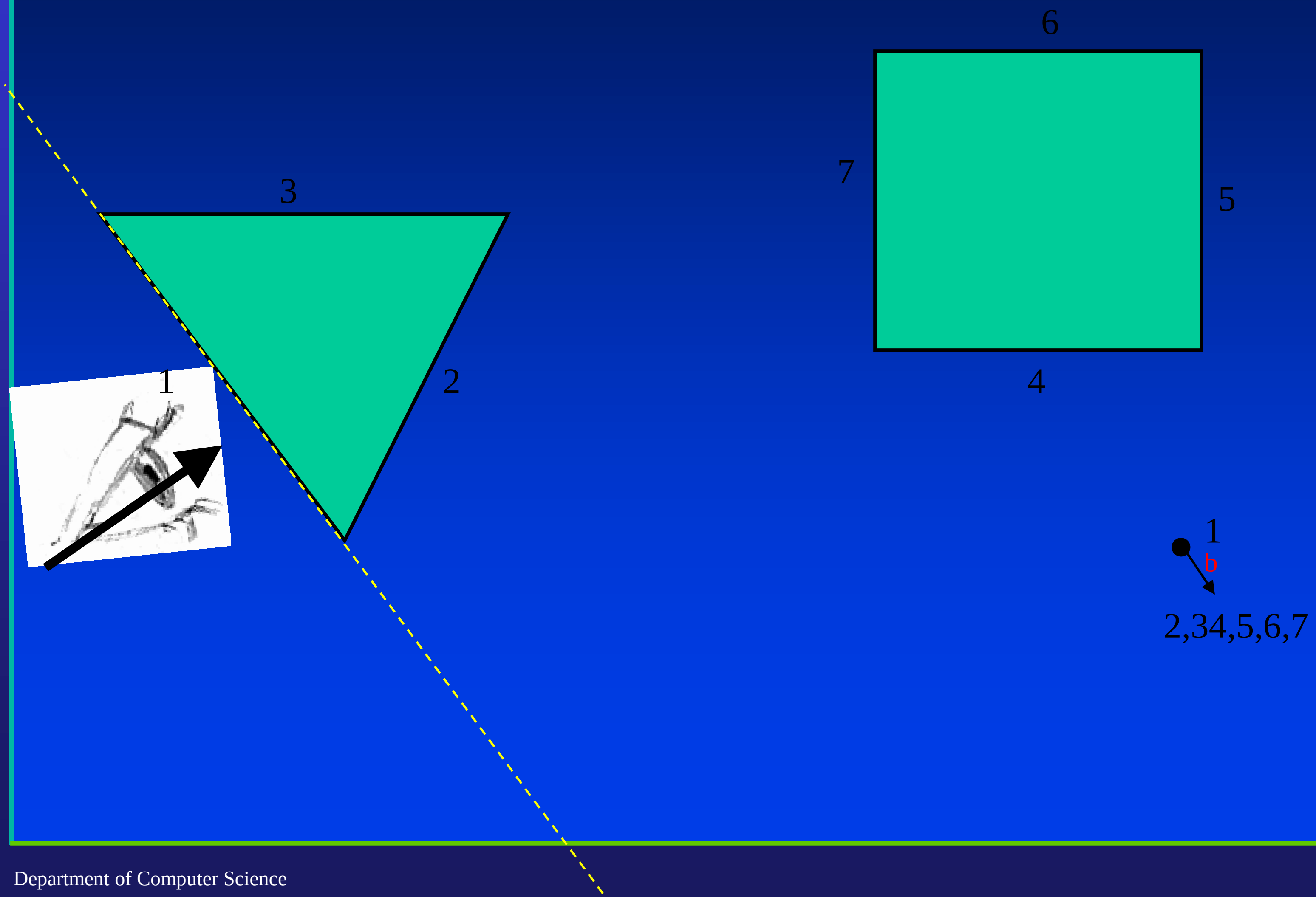
Building a BSP Tree for Polygons

- Choose a splitting polygon
- Sort all other polygons as
  - Front
  - Behind
  - Crossing
  - On
- Add “front” polygons to front child, “behind” to back child
- Split “crossing” polygons with infinite plane
- Add “on” polygons to root
- Recursion

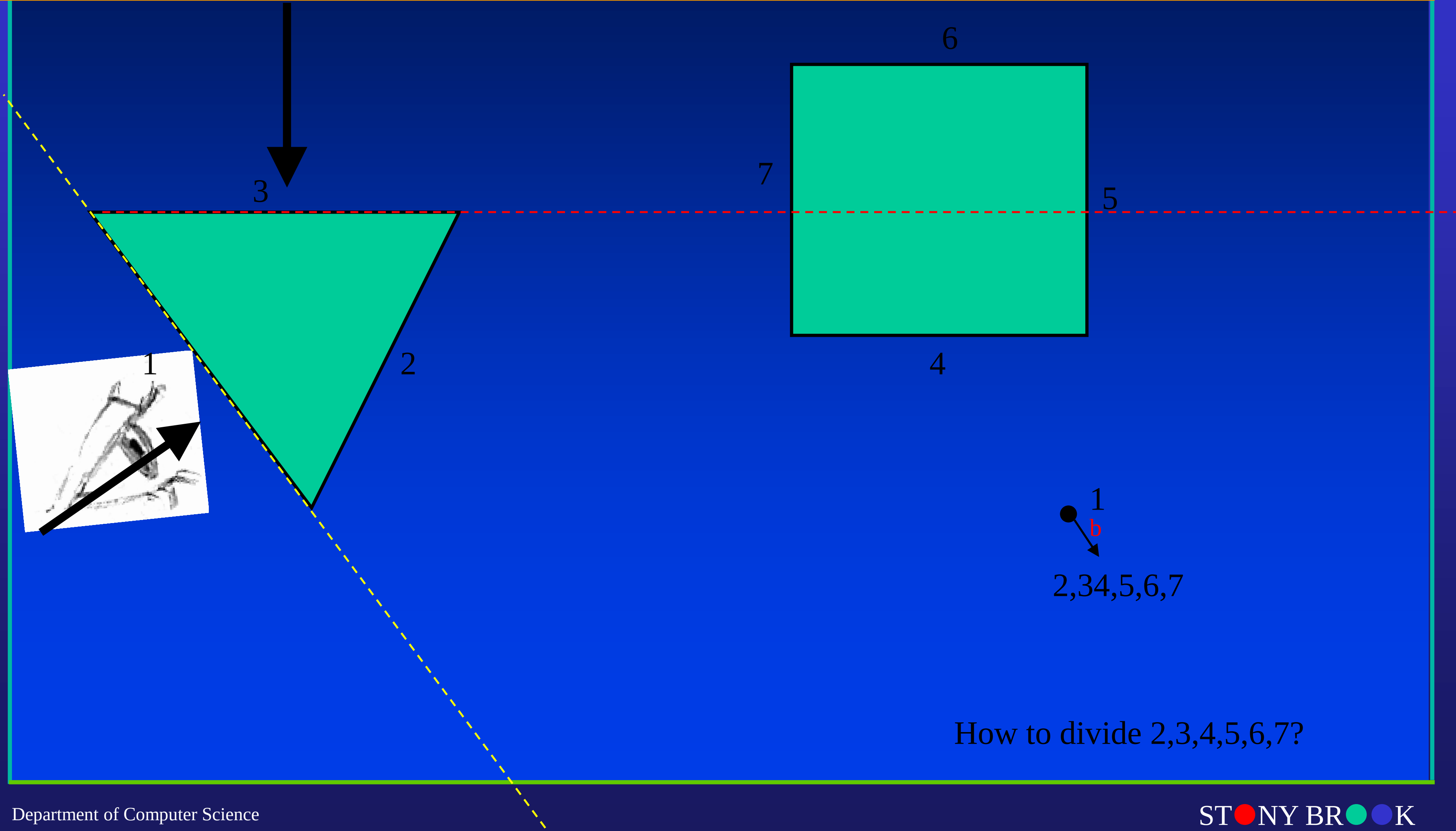
Building a BSP Tree



Building a BSP Tree

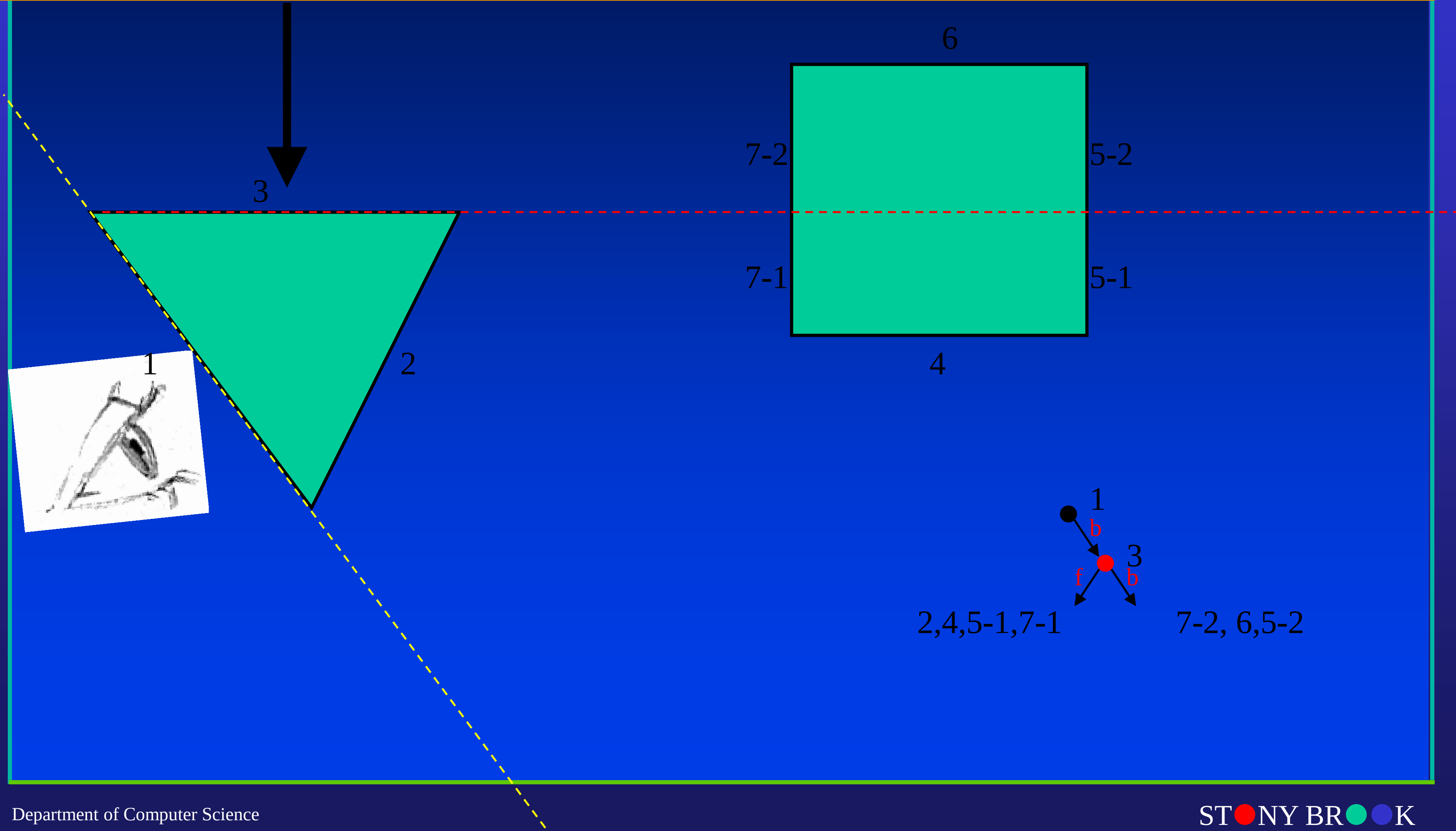


Building a BSP Tree

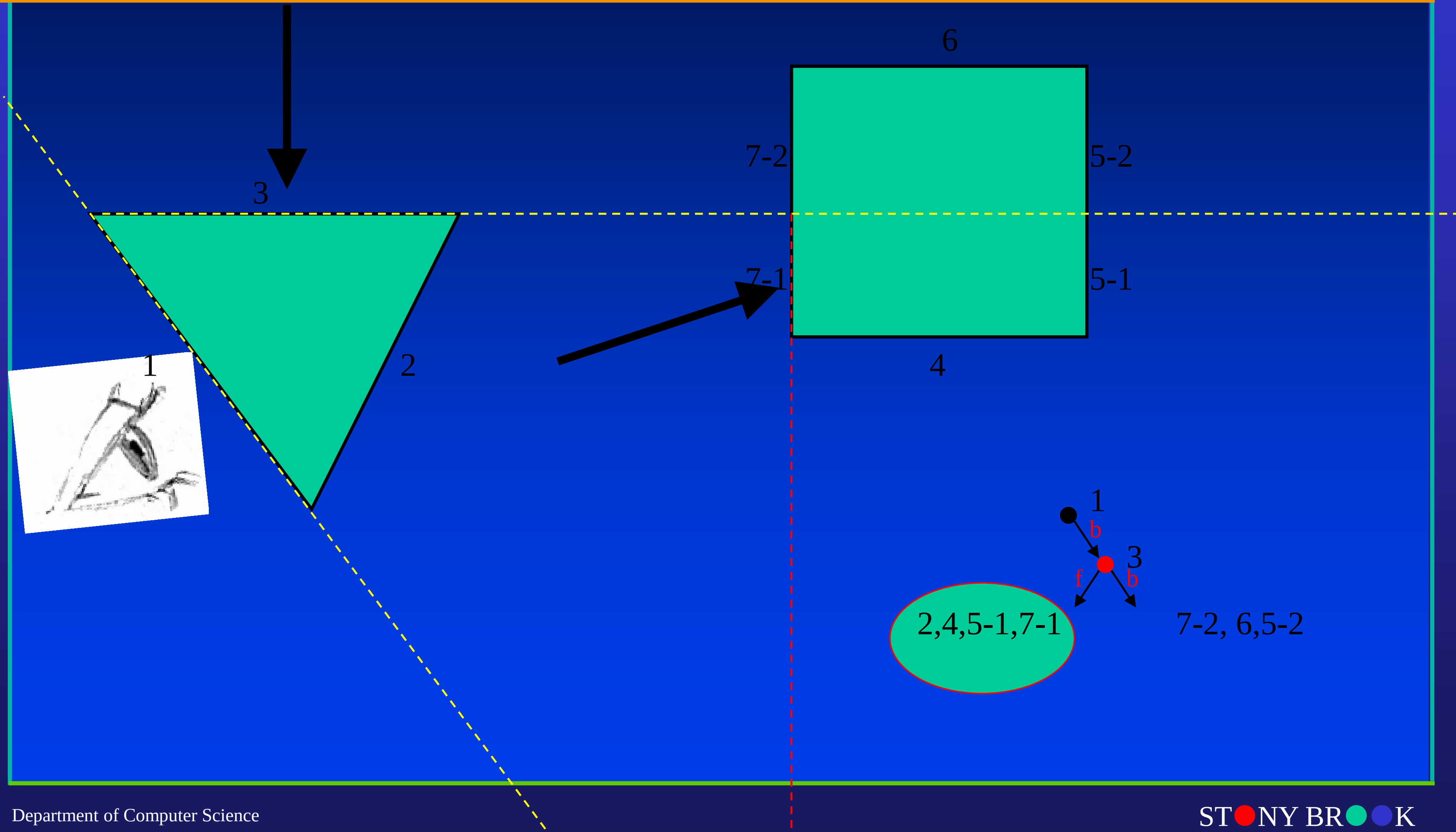


How to divide 2,3,4,5,6,7?

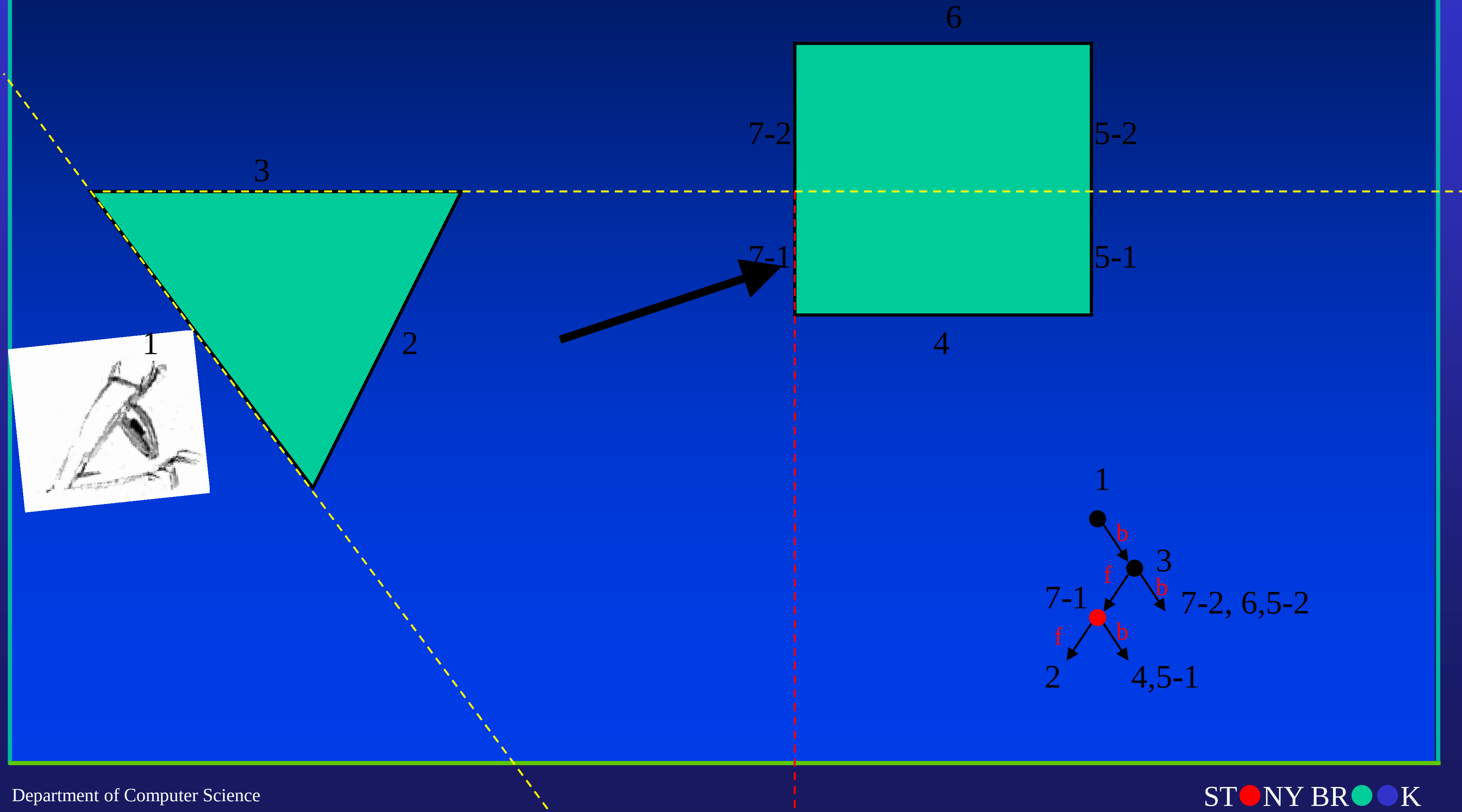
Building a BSP Tree



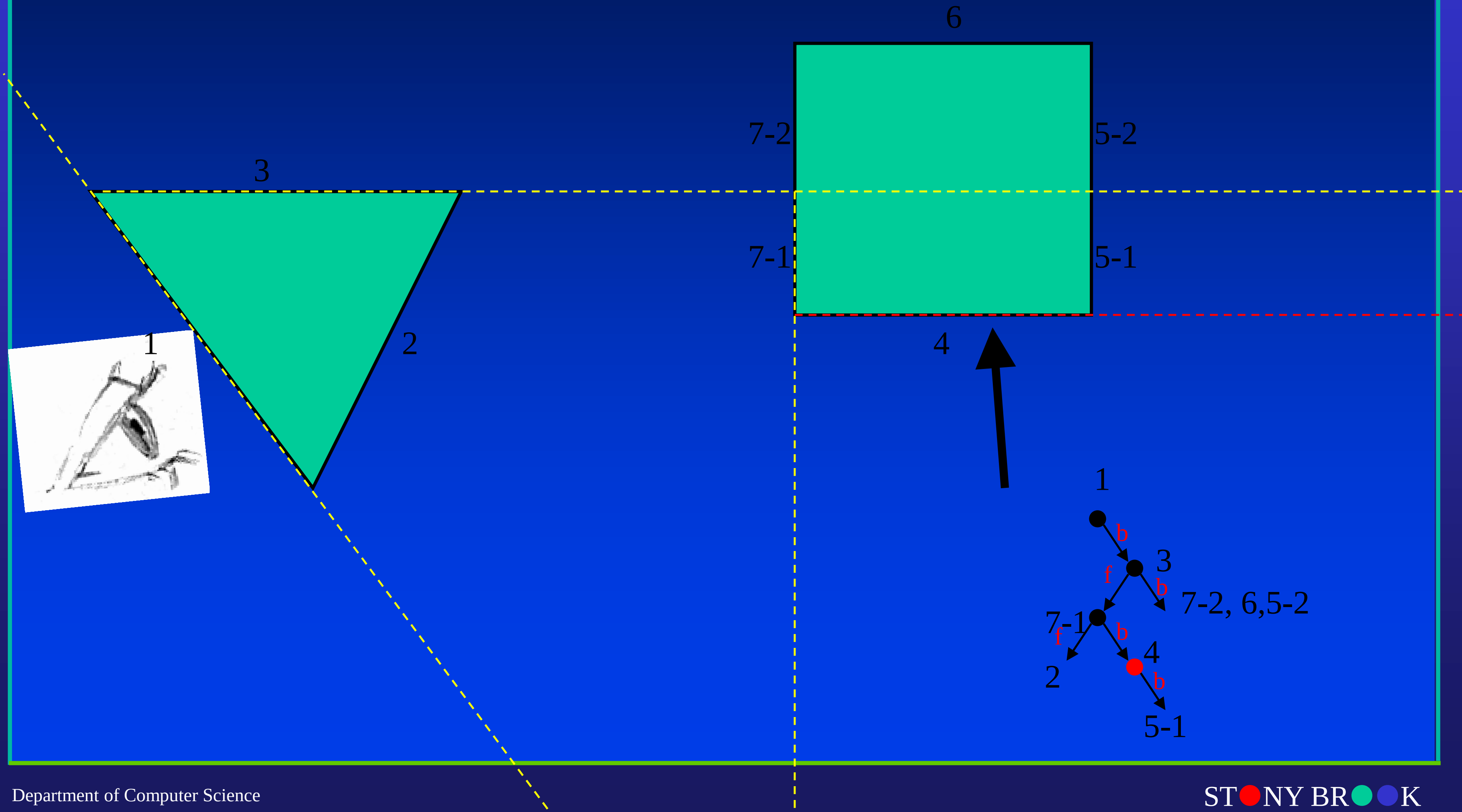
Building a BSP Tree



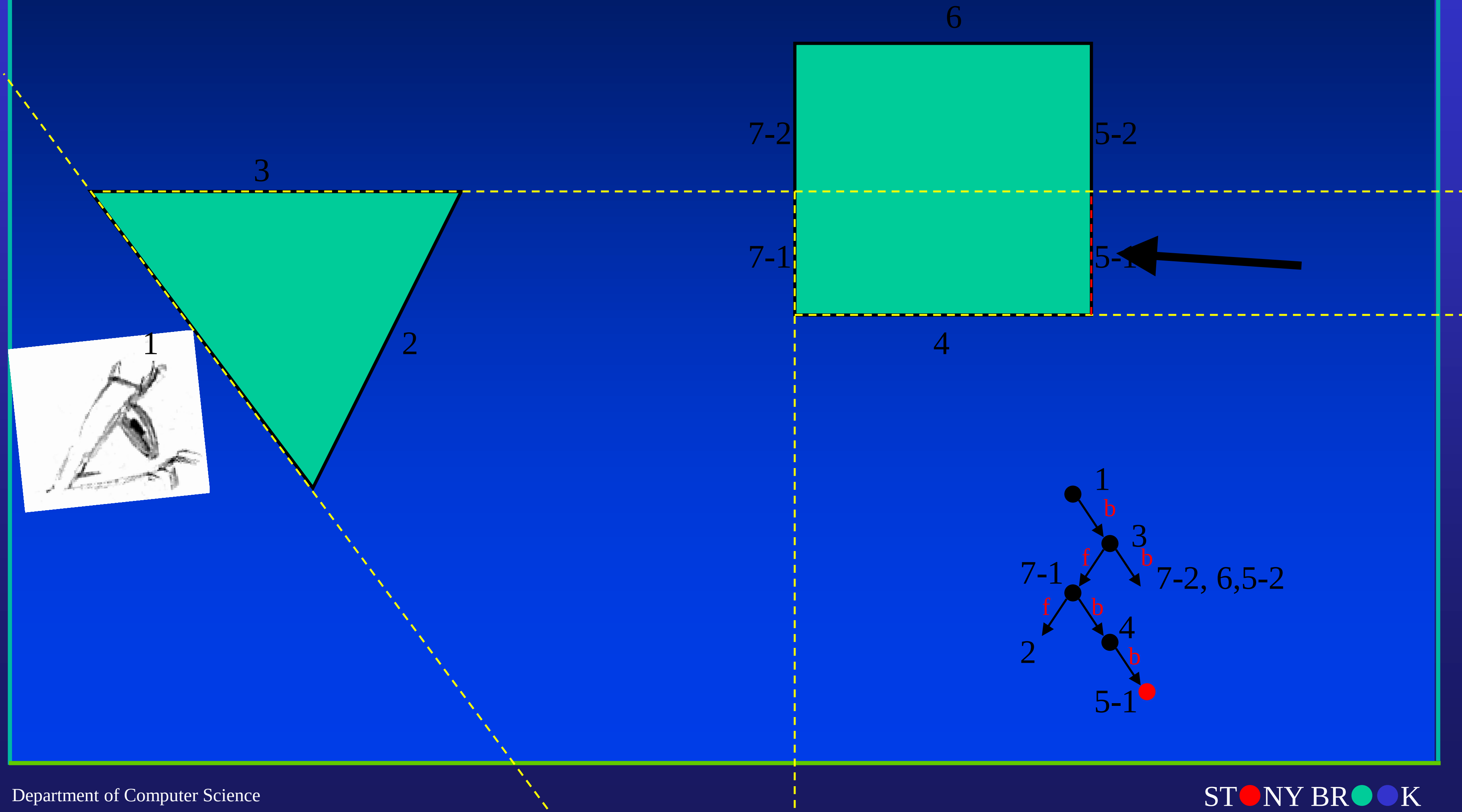
Building a BSP Tree



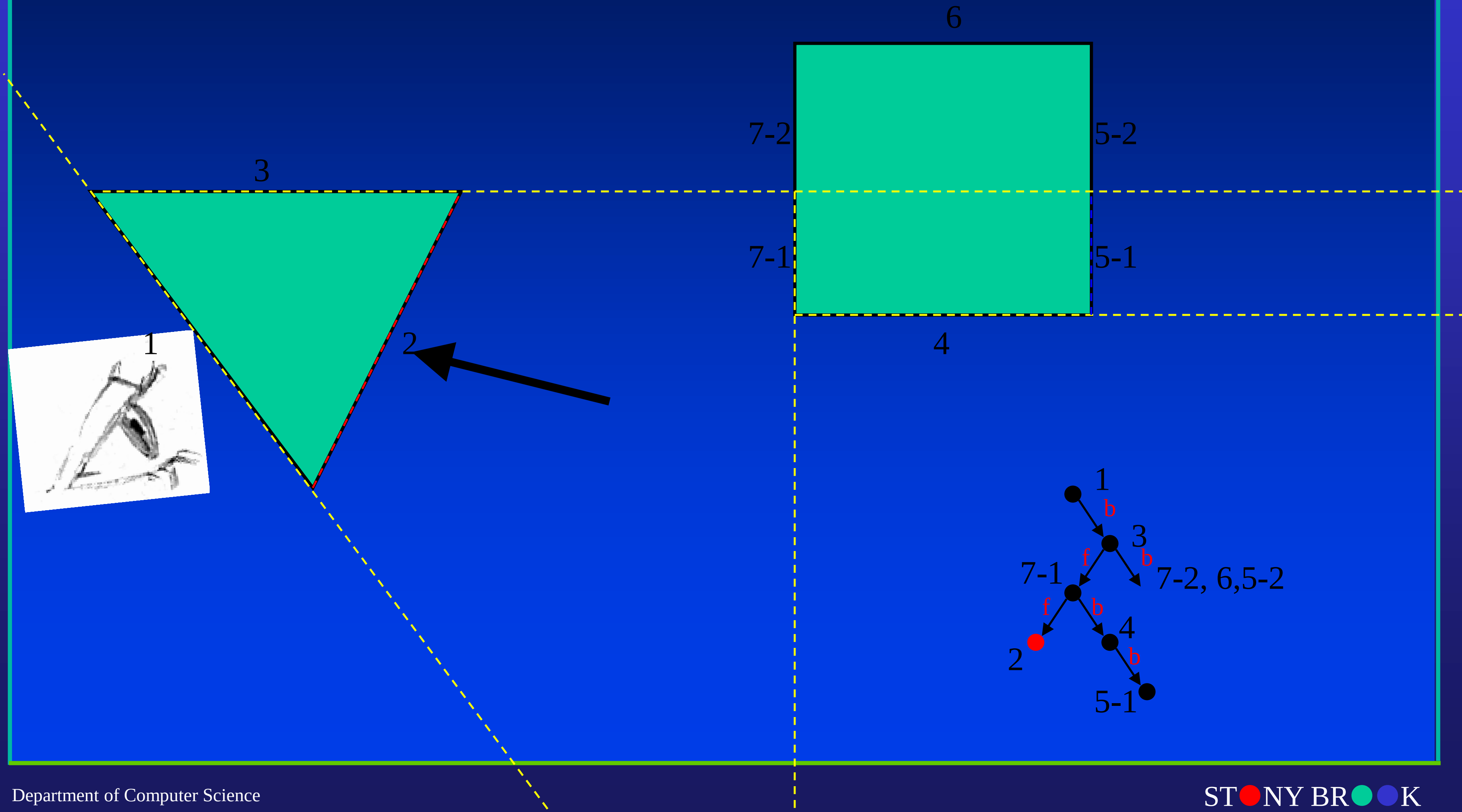
Building a BSP Tree



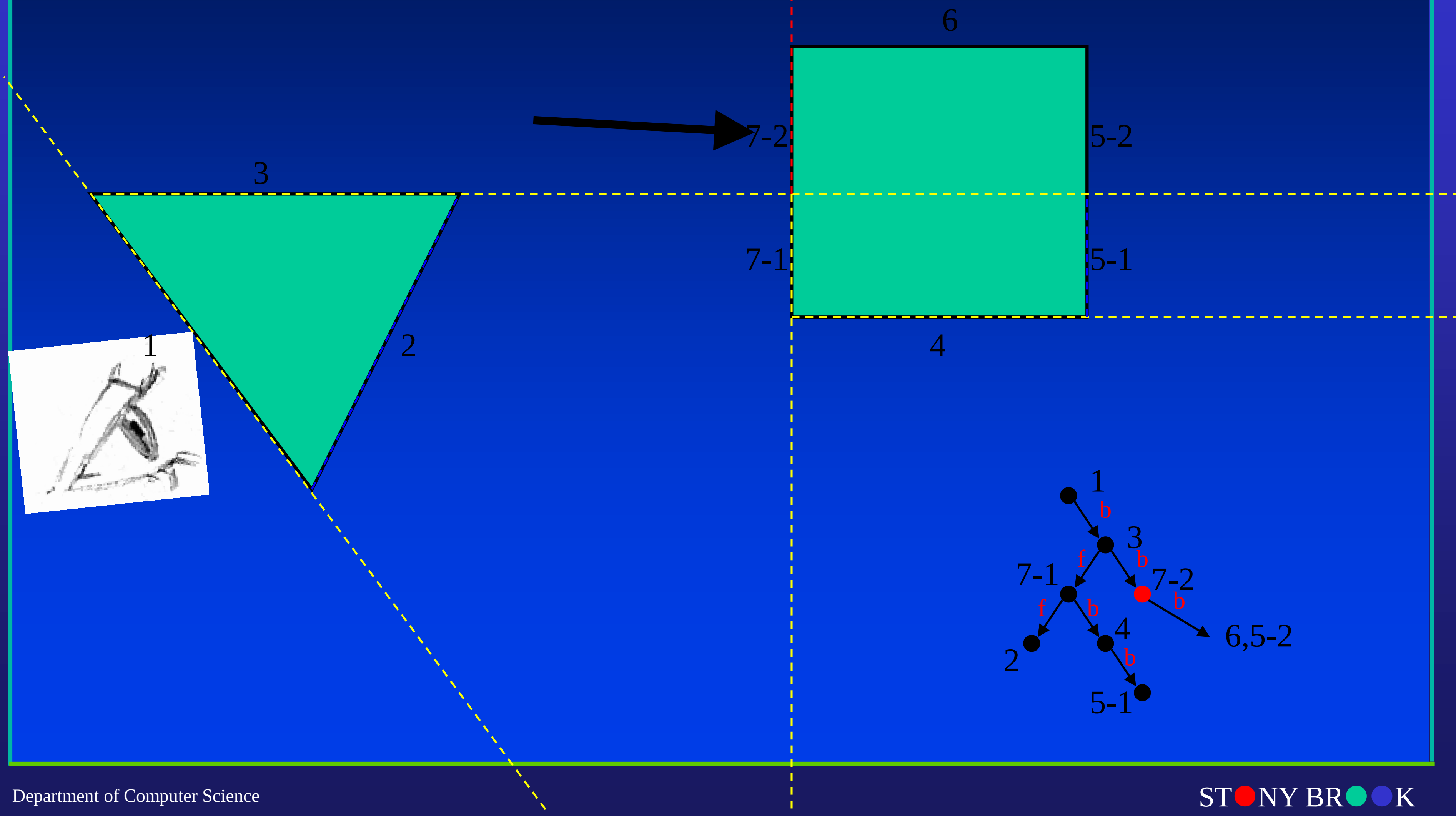
Building a BSP Tree



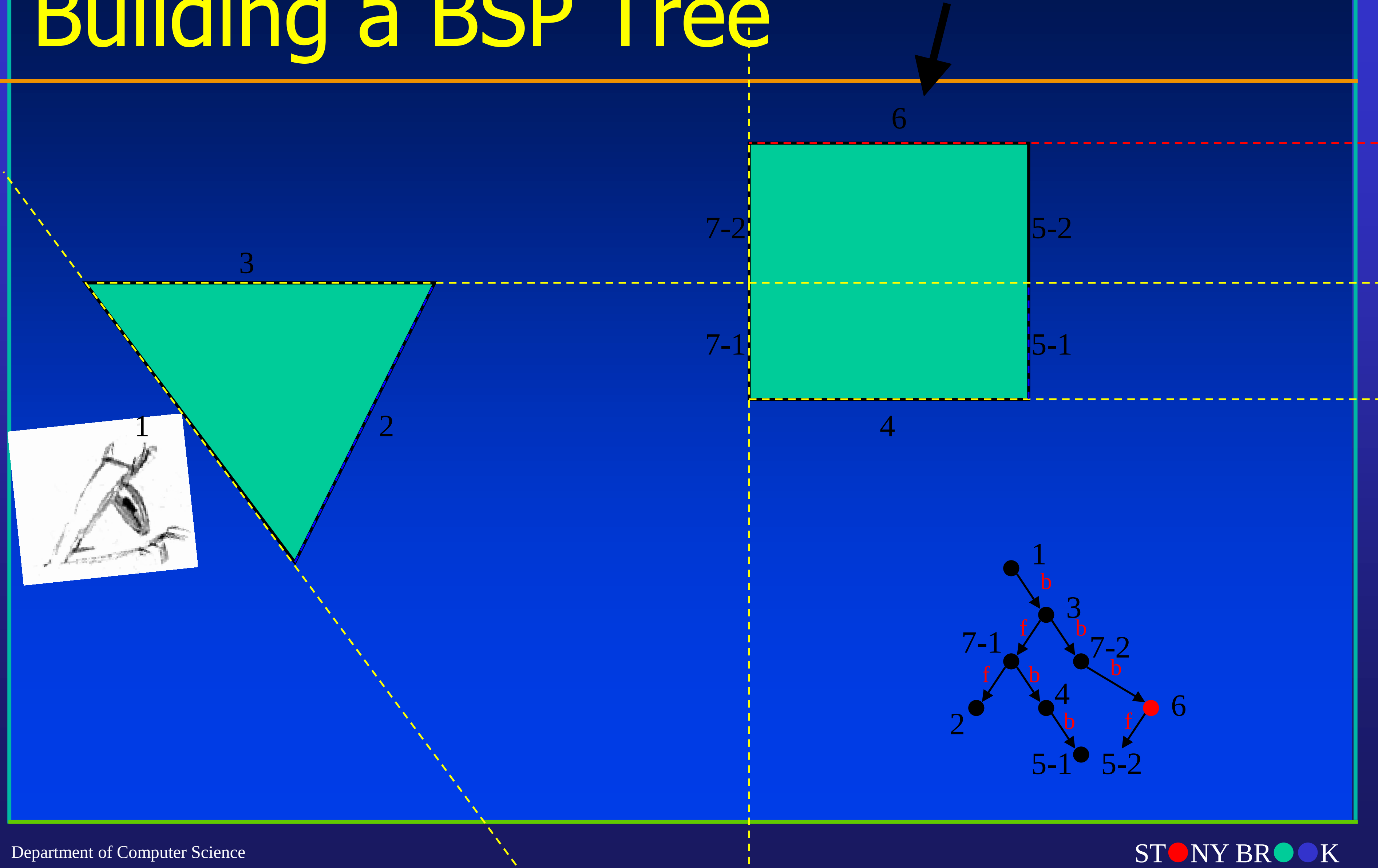
Building a BSP Tree



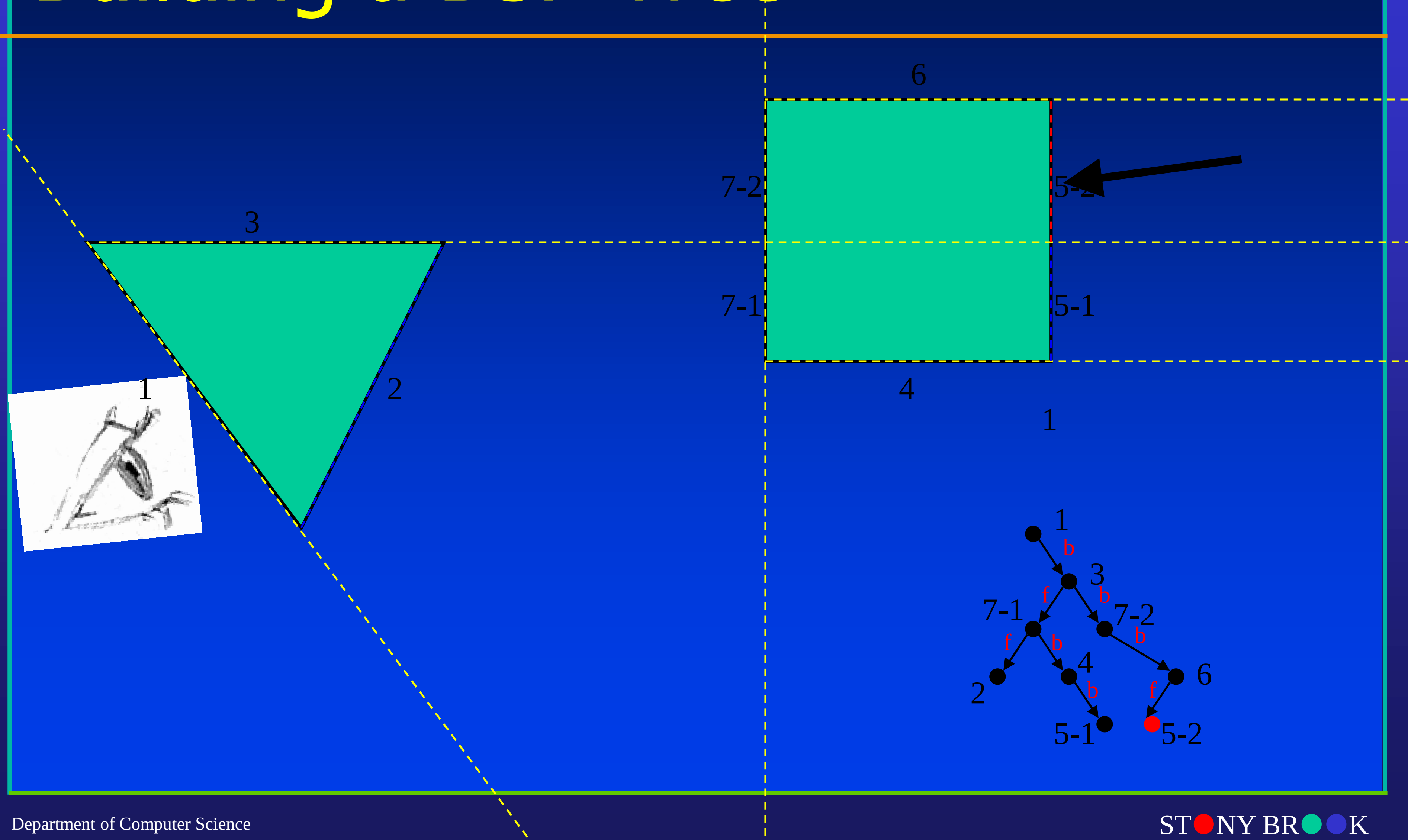
Building a BSP Tree



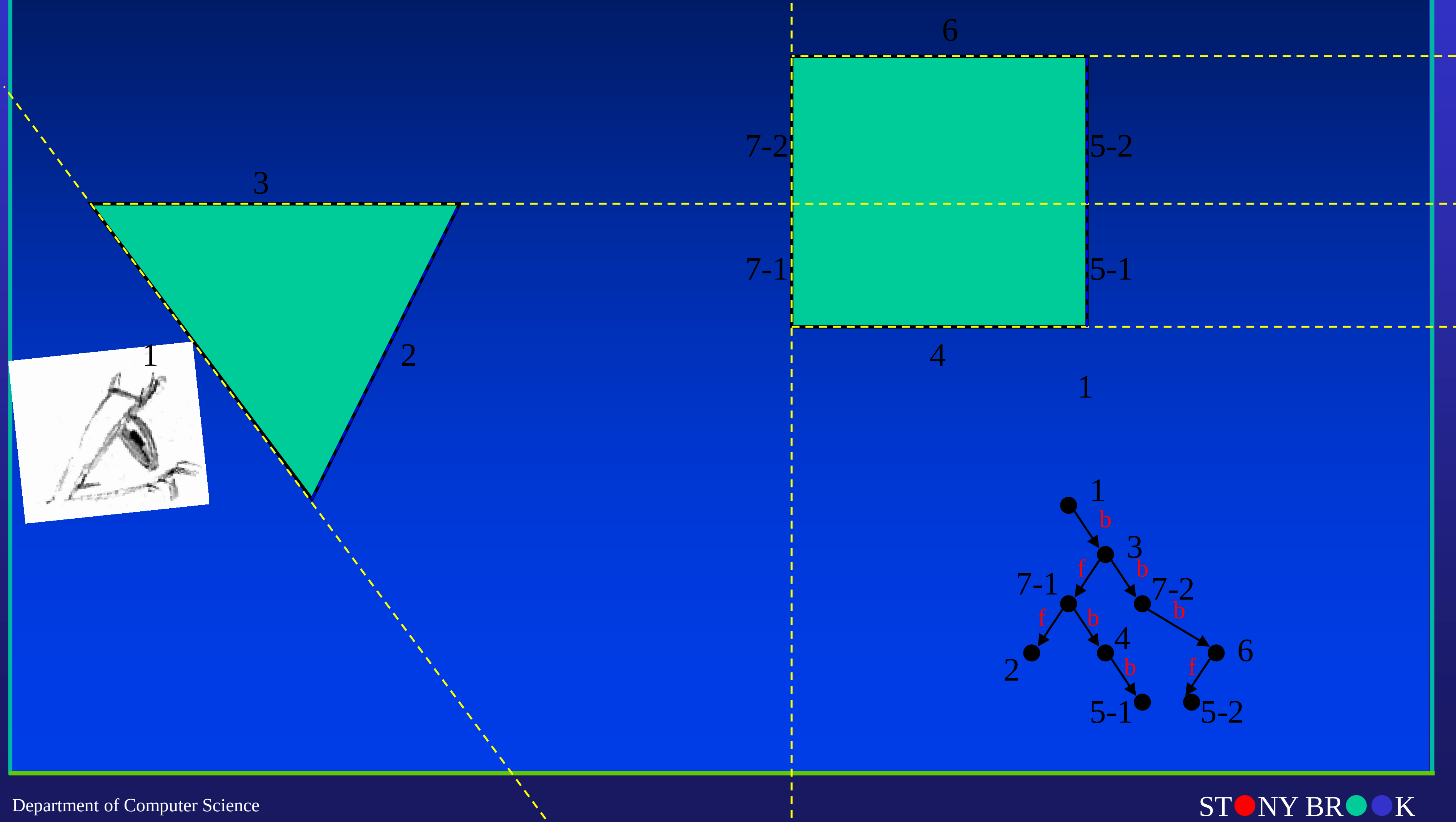
Building a BSP Tree



Building a BSP Tree



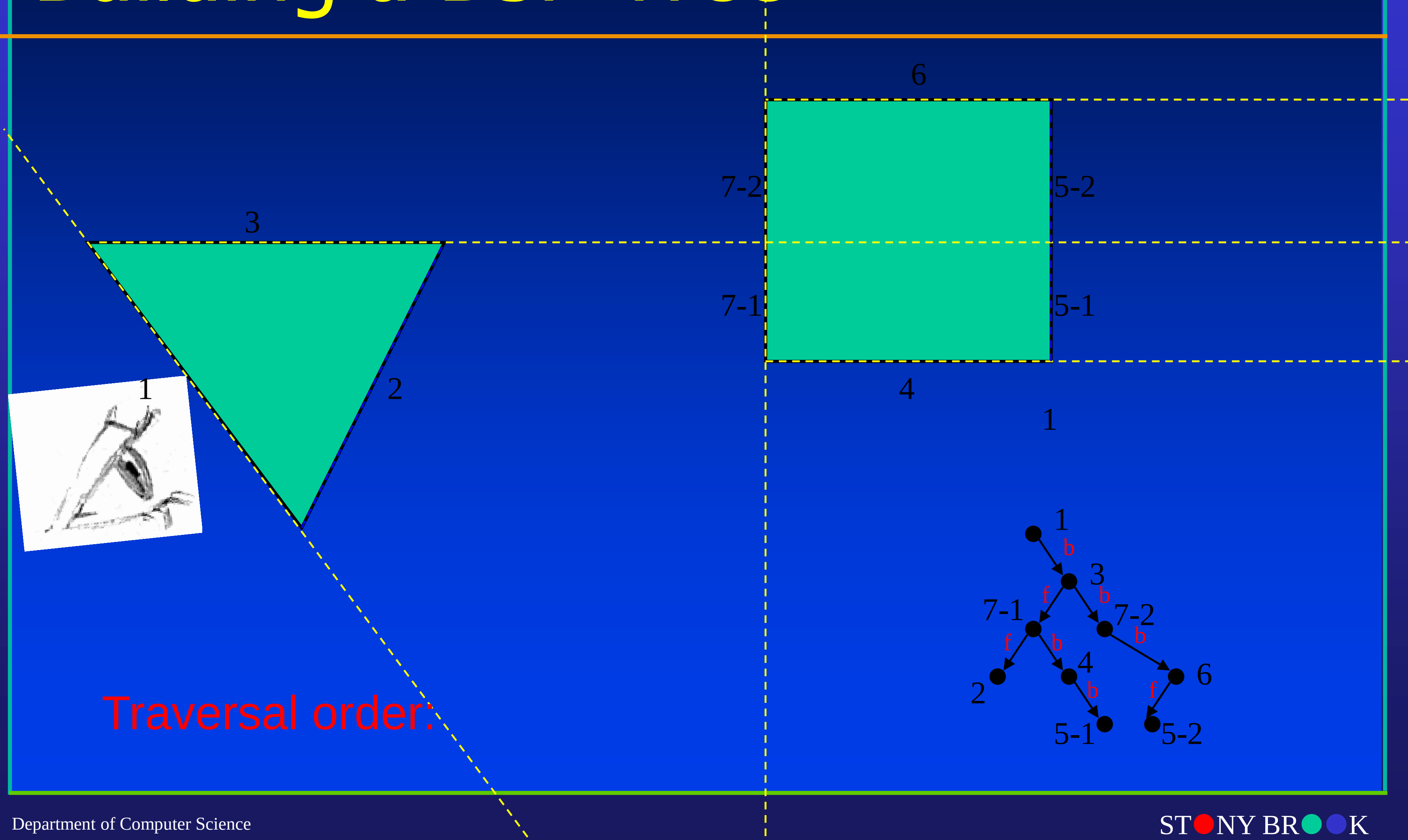
Building a BSP Tree



Rendering with a BSP Tree

- If eye is in front of plane
  - Draw “back” polygons
  - Draw “on” polygons
  - Draw “front” polygons
- If eye is behind plane
  - Draw “front” polygons
  - Draw “on” polygons
  - Draw “back” polygons
- Else eye is on plane
  - Draw “front” polygons
  - Draw “back” polygons

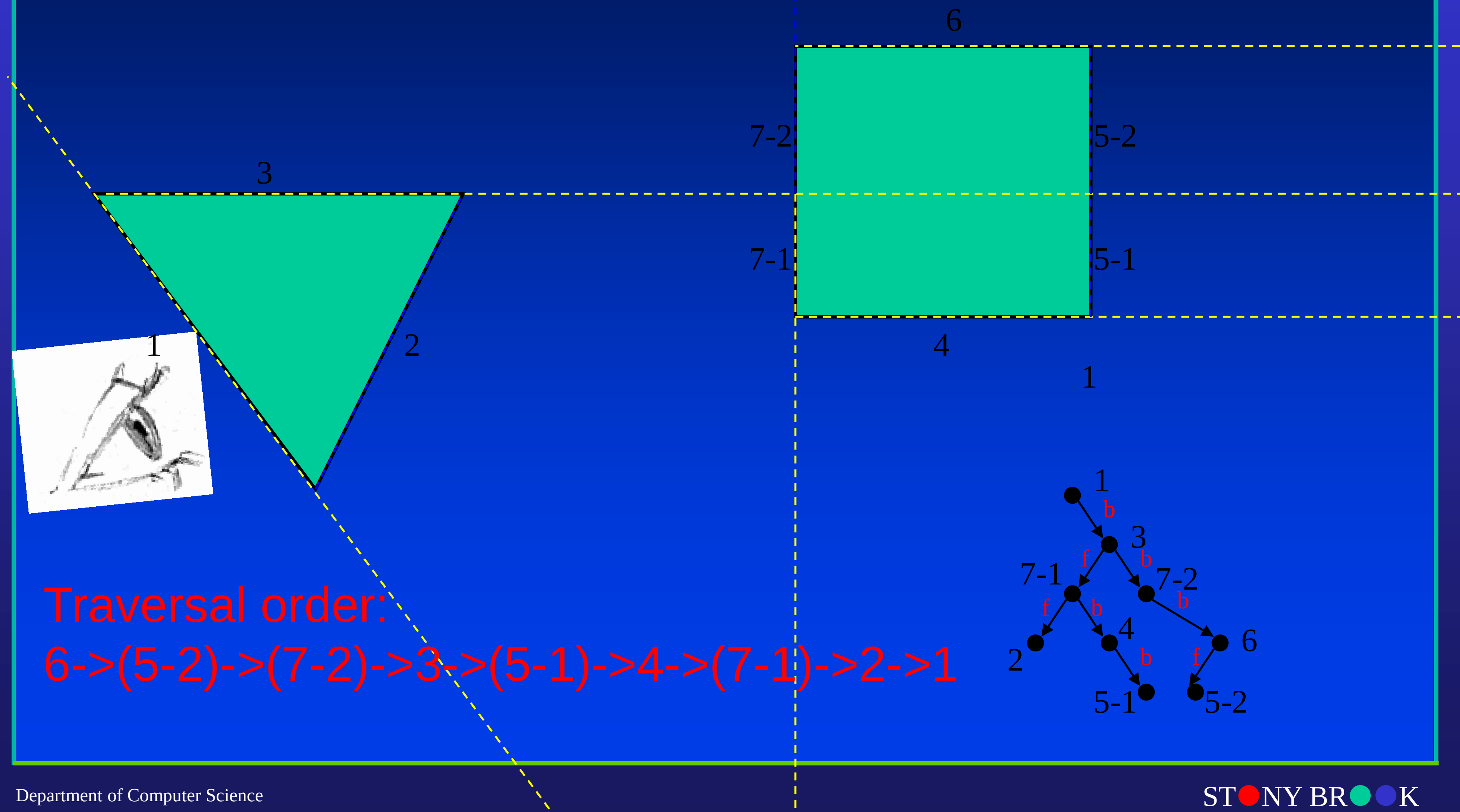
Building a BSP Tree



$$\begin{tikzcd}
 & 1 \arrow[dr, "b"] & & & & \\
 & & 3 \arrow[dl, "f"'] \arrow[dr, "b"] & & & \\
 & \text{7-1} \arrow[dl, "f"'] \arrow[dr, "b"] & & \text{7-2} \arrow[drr, "b"] & & \\
2 & & 4 \arrow[dr, "b"] & & & 6 \arrow[dl, "f"'] \\
 & & & \text{5-1} & \text{5-2} &
\end{tikzcd}$$

Traversal order:

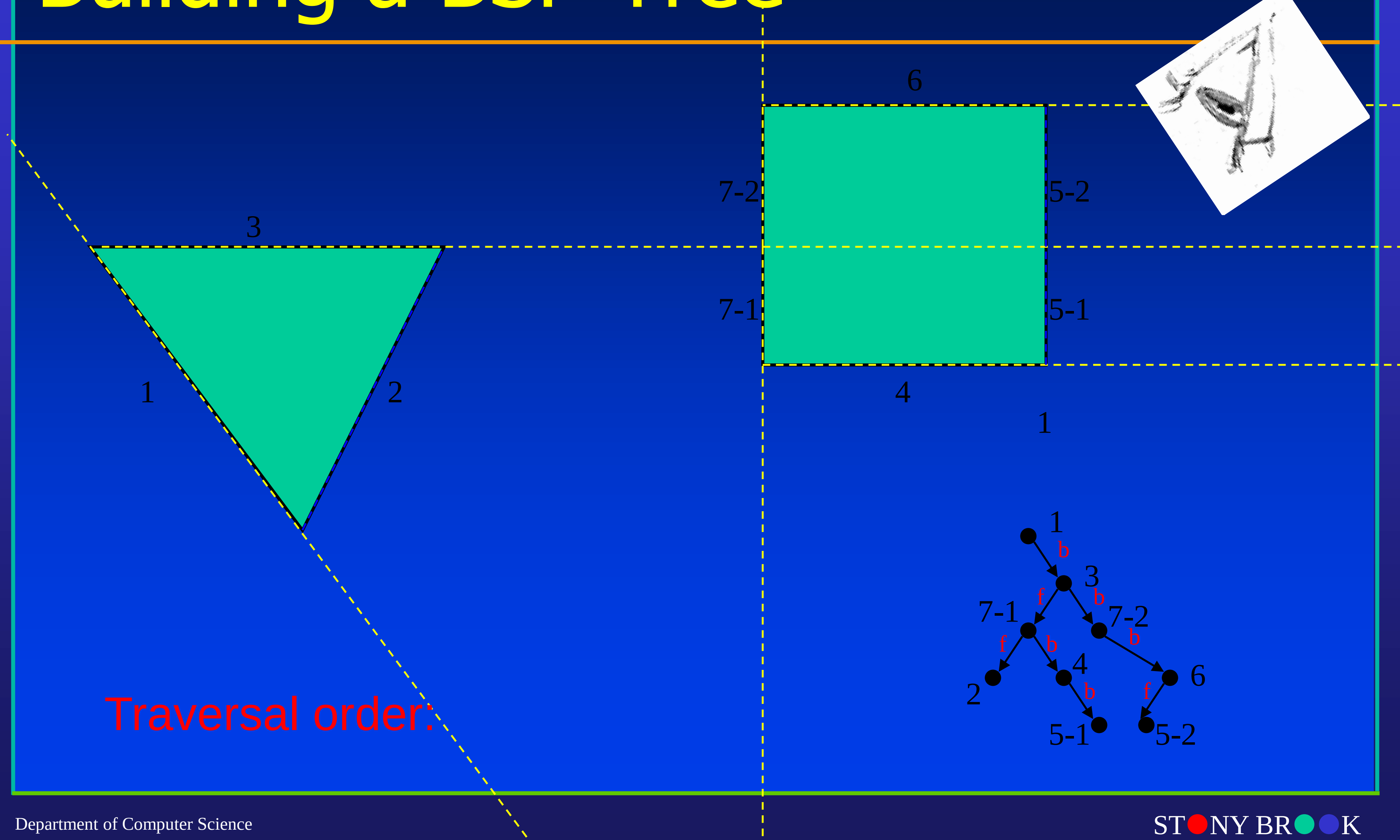
Building a BSP Tree

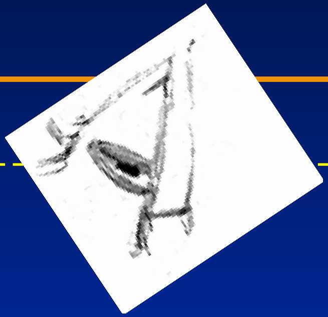


Traversal order:

6->(5-2)->(7-2)->3->(5-1)->4->(7-1)->2->1

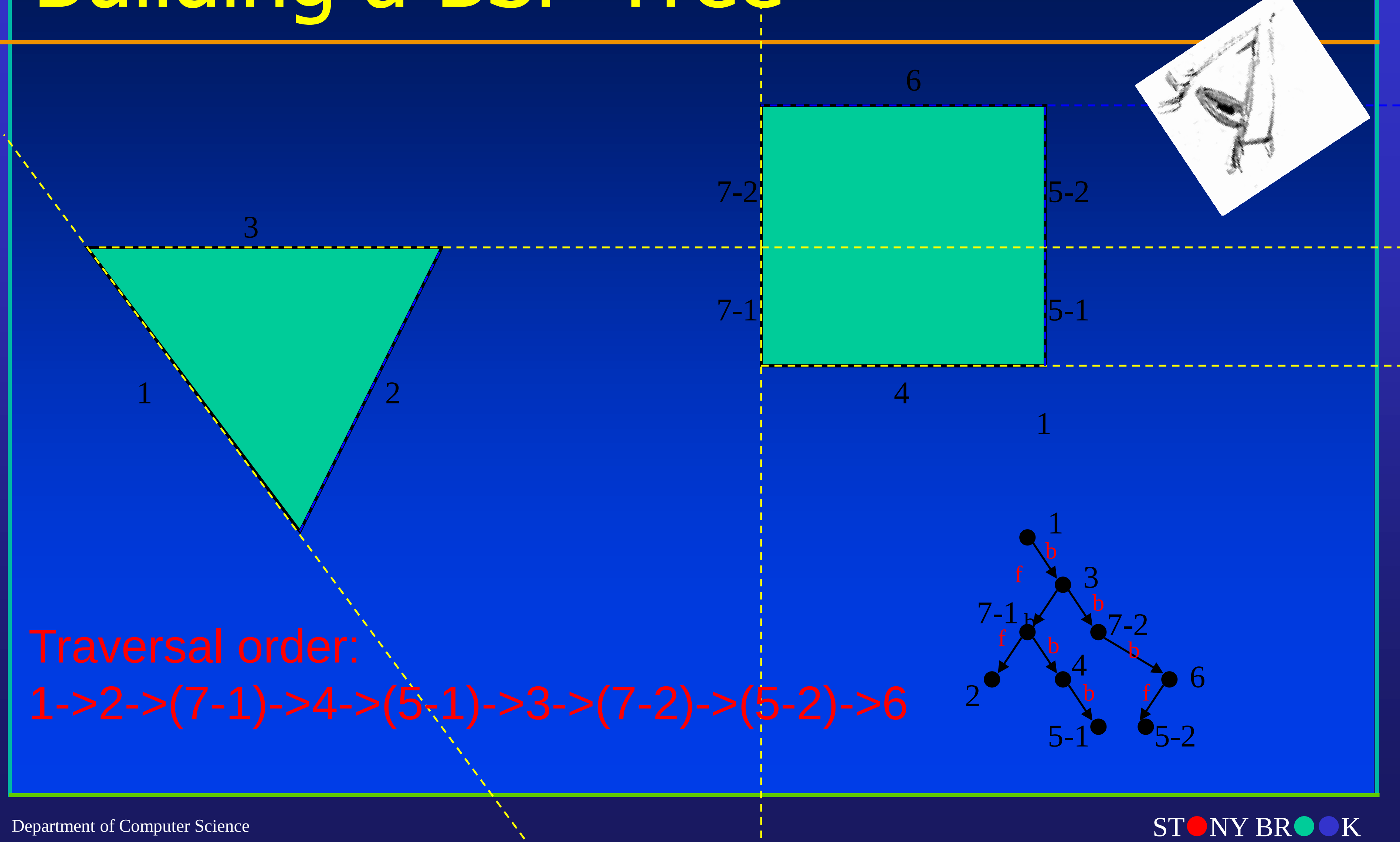
Building a BSP Tree





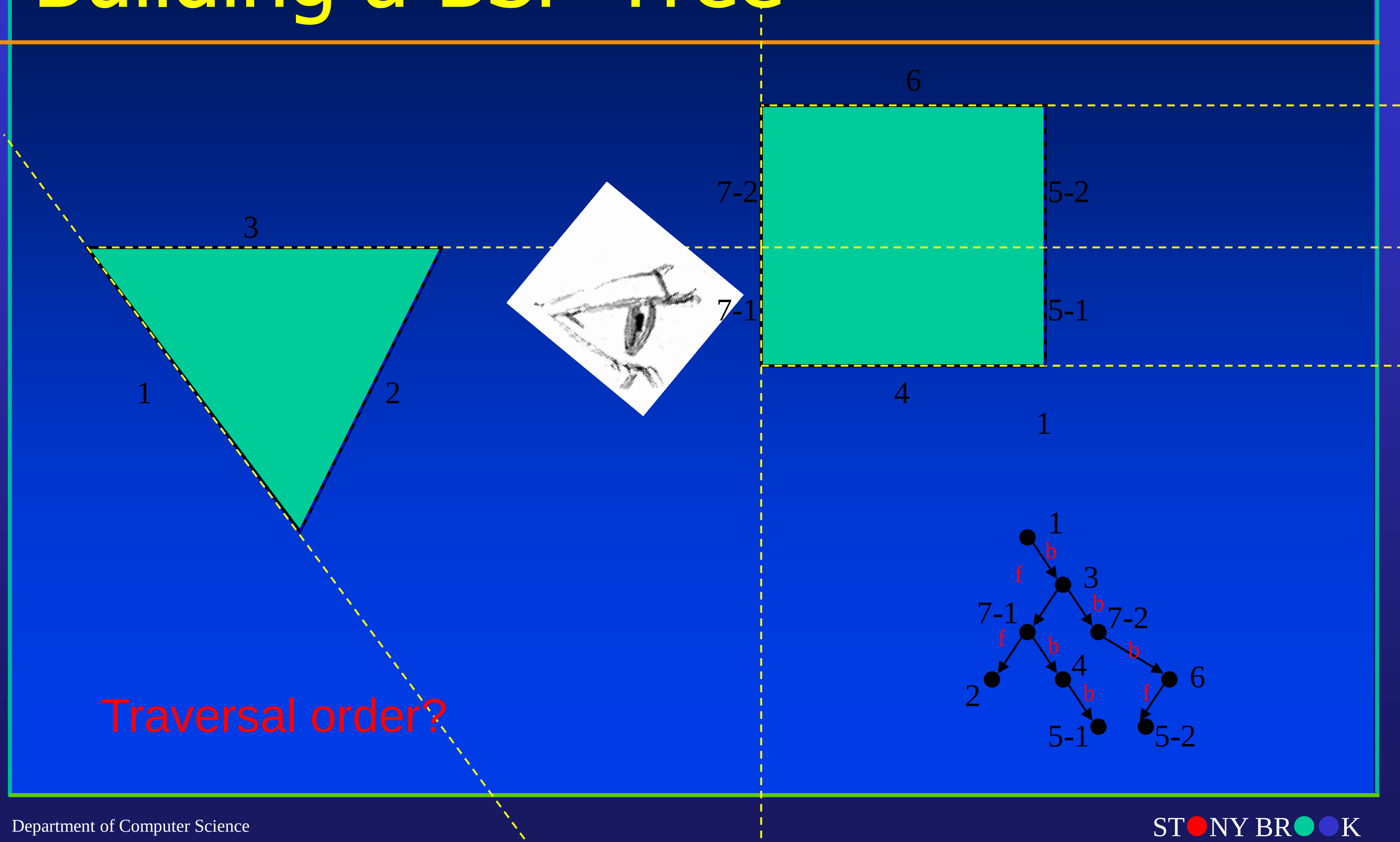
Traversal order:

Building a BSP Tree



Traversal order:
1->2->(7-1)->4->(5-1)->3->(7-2)->(5-2)->6

Building a BSP Tree



Traversal order?

Summary: BSP Trees

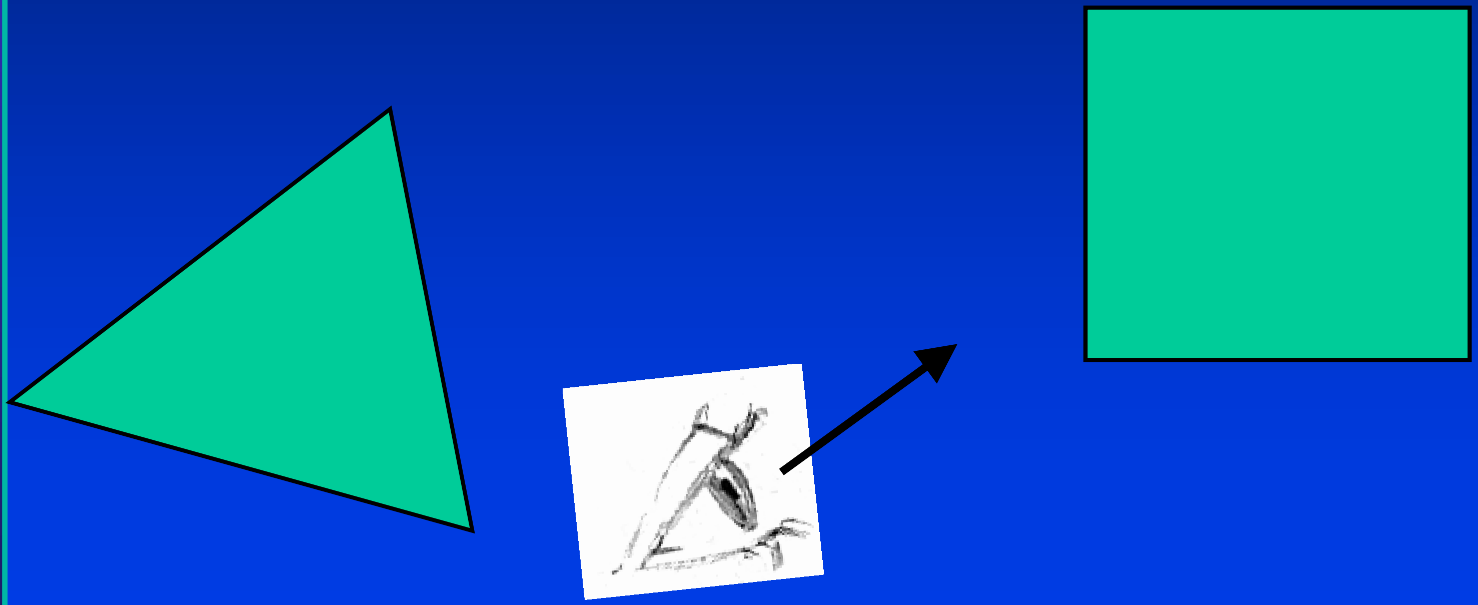
- Pros:
  - Simple, elegant scheme
  - No depth comparisons needed
  - Polygons split and ordered automatically
  - Only writes to framebuffer (i.e., painters algorithm)

Summary: BSP Trees

- Cons:
  - Computationally intense preprocess stage restricts algorithm to static scenes
  - Splitting increases polygon count
  - Redraws the same pixel many times
  - Choosing splitting plane not an exact science

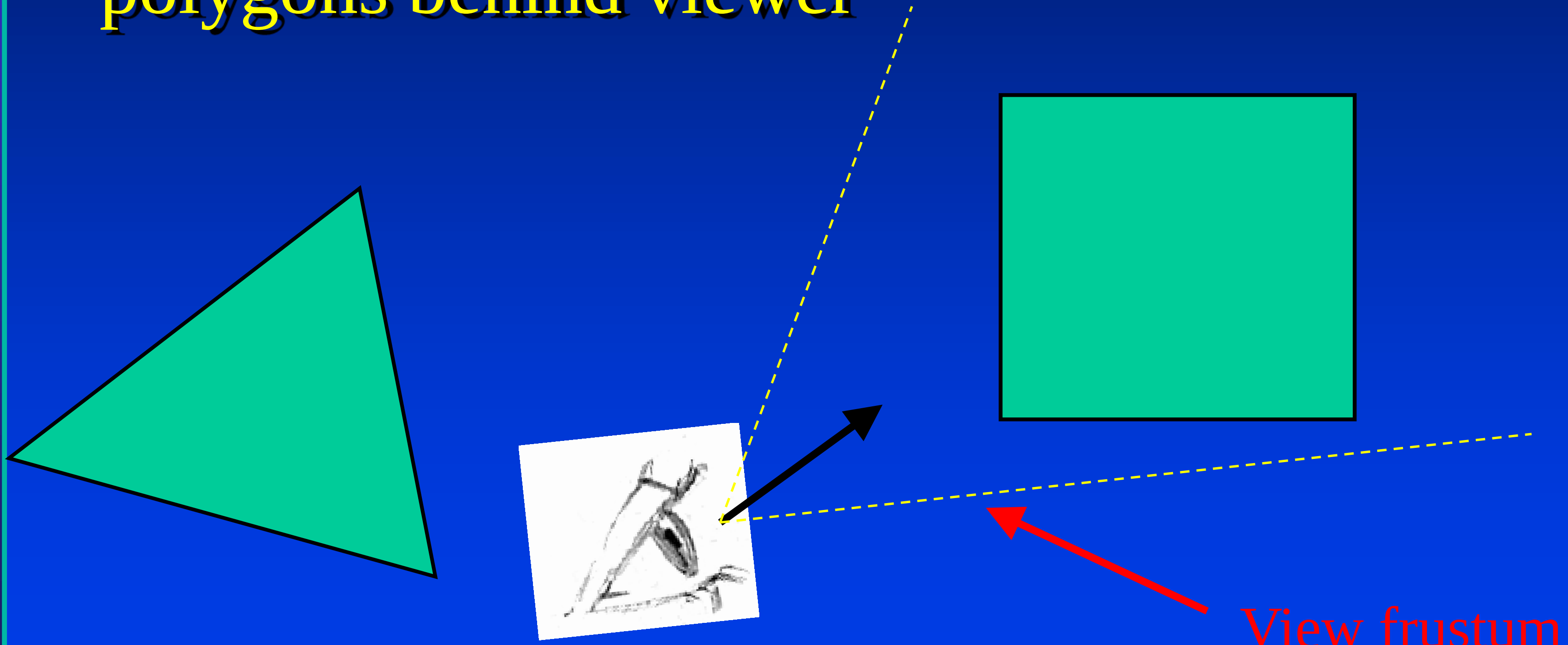
Improved BSP Rendering

- Take advantage of view direction to cull away polygons behind viewer



Improved BSP Rendering

- Take advantage of view direction to cull away polygons behind viewer



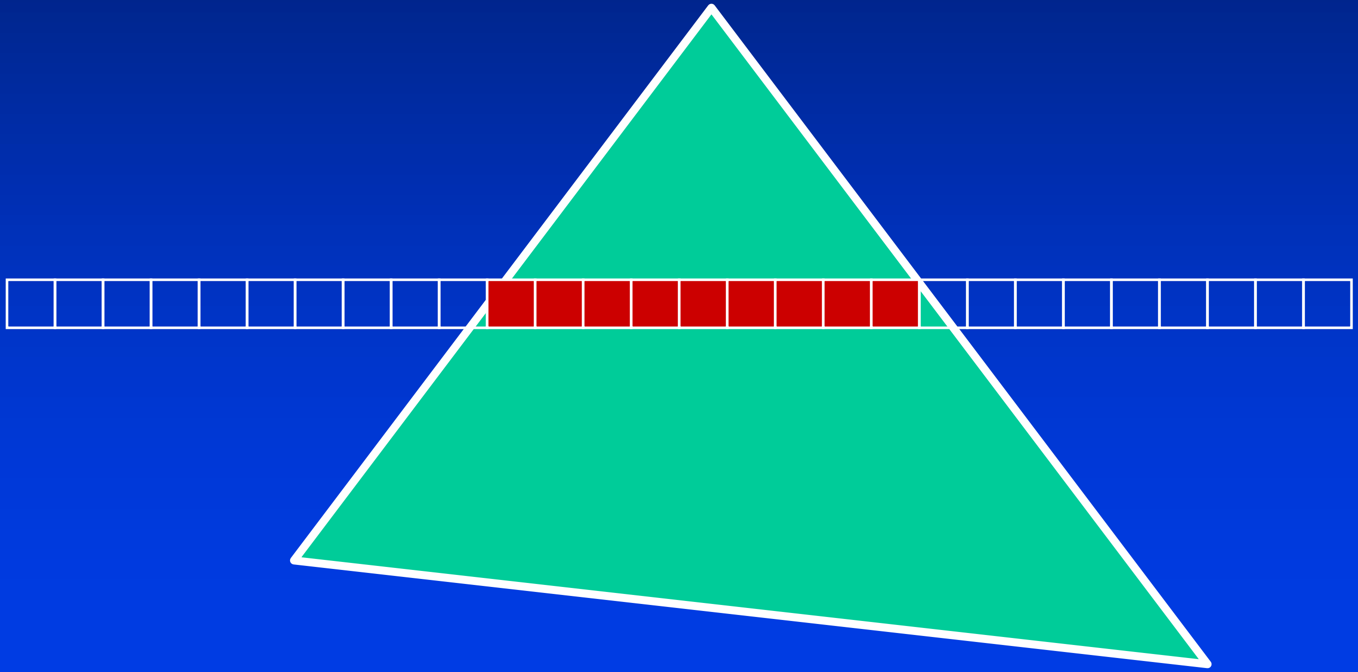
Efficiency

- BSP trees are order n log(n) in the number of polygons.
- Good for VR 'walkthrough' because you only re-compute when the eye crosses a separating plane.

Z-Buffer

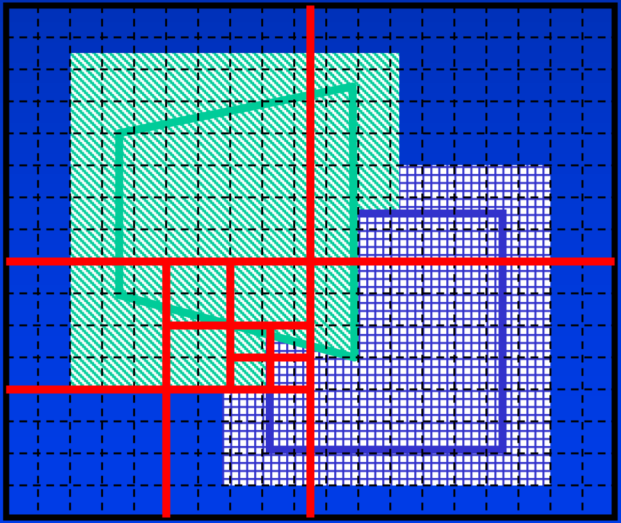
- Record r, g, b and z (depth) for each pixel.
- Process each polygon line by line and if closer replace r,g,b,z in the buffer.

Scan Line in Screen Space



Area Subdivision

- Fill area if:
  - All surfaces are outside
  - Only one surface intersects
  - One surface occludes other surfaces within area.
- Otherwise, subdivide

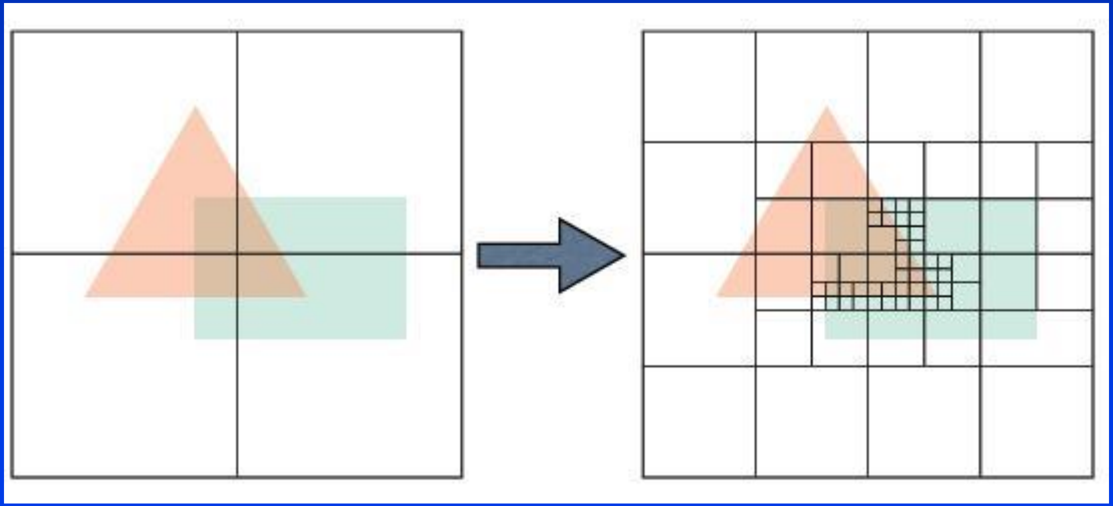


The Warnock's Algorithm

- The Warnock algorithm divides the screen into smaller areas and sorts triangles within these. If there is ambiguity (i.e., polygons overlap in depth extent within these areas), then further subdivision occurs. At the limit, subdivision may occur down to the pixel level.

Warnock Algorithm

- Takes advantage of *area coherence*: divide the display area into successively smaller rectangles until the entire rectangle can be filled with a single color

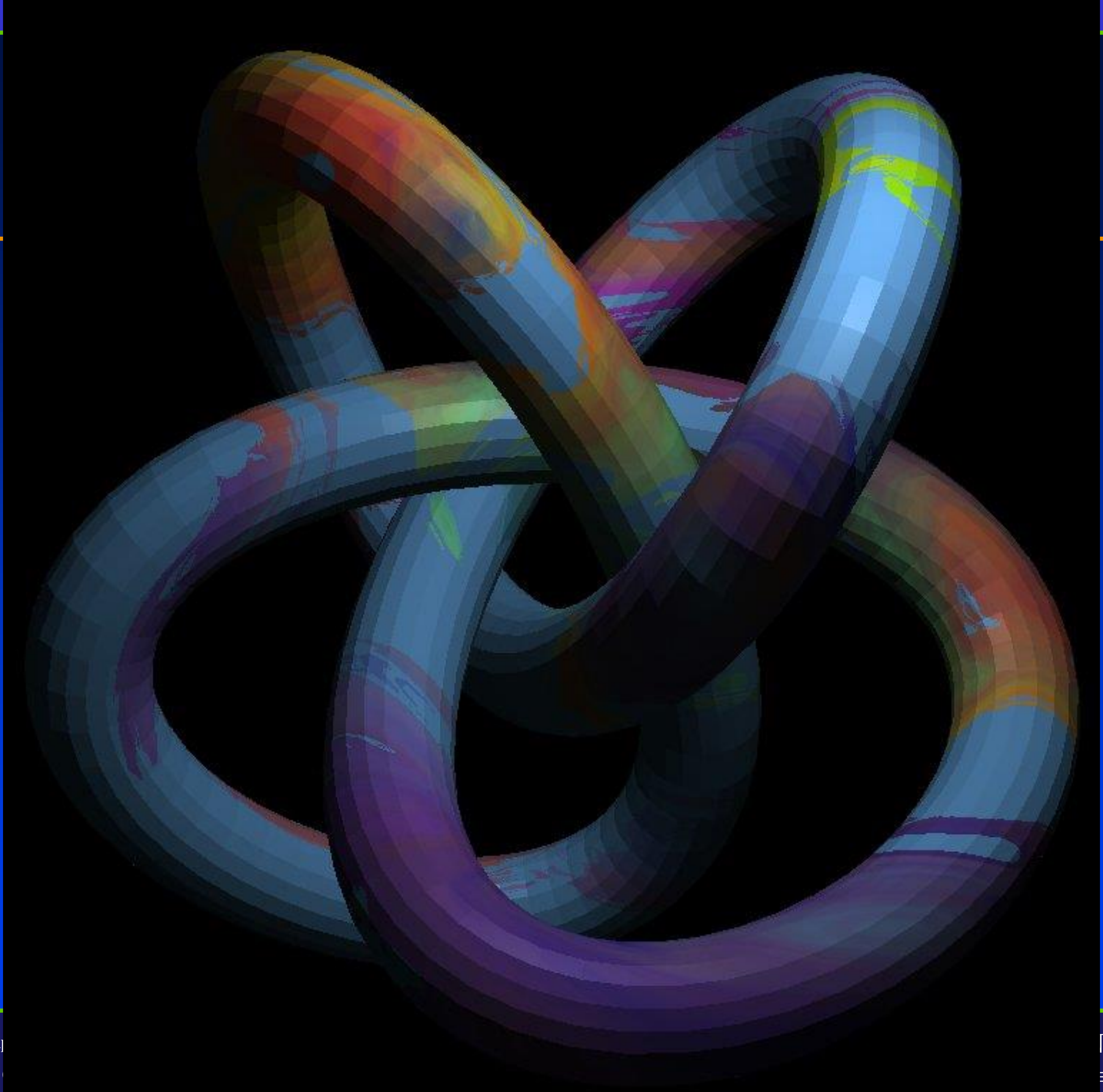


Finding the Depth

- Plane equation is $Ax + By + Cz + D = 0$
- $z = -(Ax + By + D)/C$
- replace x by x+1
- $z' = -(A(x+1) + By + D)/C$

□ $\Delta z = z' - z = -A/C$

- New z is found by adding a constant.



tenui