

# Spatial Data Structures and Speed-Up Techniques

Ulf Assarsson

Department of Computer Science and  
Engineering

Chalmers University of Technology

# Have you done your homework ;-)

## Exercises

- Create a function (by writing code on paper) that tests for intersection between:
  - two spheres
  - a ray and a sphere
  - view frustum and a sphere
  - Ray and triangle (e.g. use formulas from last lecture)
- Make sure you understand matrices:
  - Give a scaling matrix, translation matrix, rotation matrix and simple orthogonal projection matrix

# Ray/sphere test

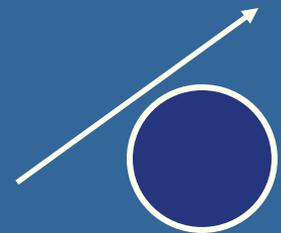
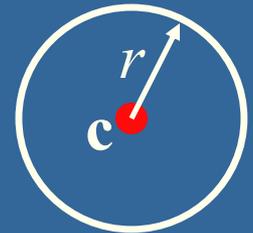
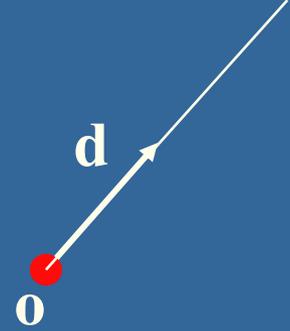
- Ray:  $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$
- Sphere center:  $\mathbf{c}$ , and radius  $r$
- Sphere formula:  $\|\mathbf{p} - \mathbf{c}\| = r$
- Replace  $\mathbf{p}$  by  $\mathbf{r}(t)$ , and square it:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) - r^2 = 0$$

$$t^2 + 2((\mathbf{o} - \mathbf{c}) \cdot \mathbf{d})t + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 = 0$$

$$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b}{2a} \pm \sqrt{\left(\frac{b}{2a}\right)^2 - \frac{c}{a}}$$

```
Bool raySphereIntersect(vec3f o, d, c, float r, Vec3f &hitPt) {  
    float b = 2.0f*((o-c).dot(d)); // dot is implemented in class Vec3f  
    float c = (o-c).dot(o-c);  
    if(b*b/4.0f < c) return false;  
    float t = -b/(2.0f) - sqrt(b*b/4.0f - c); // intersection for smallest t  
    if (t < 0) t = -b/(2.0f*a) + sqrt(b*b/4.0f - c); // larger t  
    if (t < 0) return false; else hitPt = o + d*t; // where * is an operator for vec mul  
    return true;  
}
```



# Misc

- Half Time wrapup slides are available in “Schedule” on home page
- There is an Advanced Computer Graphics Seminar Course in sp 3+4, 7.5p
  - One seminar every week
    - Advanced CG techniques
  - Do a project of your choice.
  - Register to the course

# Spatial data structures

- What is it?
  - Data structure that organizes geometry in 2D or 3D or higher
  - The goal is faster processing
  - Needed for most "speed-up techniques"
    - Faster real-time rendering
    - Faster intersection testing
    - Faster collision detection
    - Faster ray tracing and global illumination
- Games use them extensively
- Movie production rendering tools always use them too
- (You may read "Designing a PC Game Engine". Link available on website)

- week 6: room HC1
- week 7: room HC3

**NOTE 2: The follow-up course, [DAT205 Advanced Computer Graphics](#), will run in study period 3+4 as usual, despite what studentportalen says.**

**Home page is continuously being updated**

#### COURSE-PM

Course start: (sp2, week 1). Lectures each Wednesday 10-12, and Friday 9-12.  
 7,5 Högskolepoäng  
 Grades: U (failed), 3, 4, 5  
 Educational Level: Advanced  
 Institution: 37 - DATA- OCH INFORMATIONSTEKNIK  
 Teaching language: English

**Teacher and Examiner:** Ulf Assarsson, intern phone 1775 (031-7721775)  
 room 4115, floor 4, the corridor along Rännvägen, ED-huset E-mail: see above.

**Course assistants:** Erik Sintorn (erik dot sintorn at chalmers dot se), Ola Olsson (ola dot olsson at chalmers dot se), Markus Billeter (billeter at chalmers dot se)

**Course webpage:** <http://www.cse.chalmers.se/edu/course/TDA361/>  
[Course plan](#)

#### Links:

- [Link to home page at Studieportalen](#)
- [Seminar Course in Advanced Computer Graphics](#)
- Links to related previous courses, now obsolete:
  - [TDA361 Computer Graphics: 2010, 2009, 2008, 2007](#)
  - [TDA360 Datorgrafik 2006](#)
  - [Avancerad Datorgrafik 2006](#)

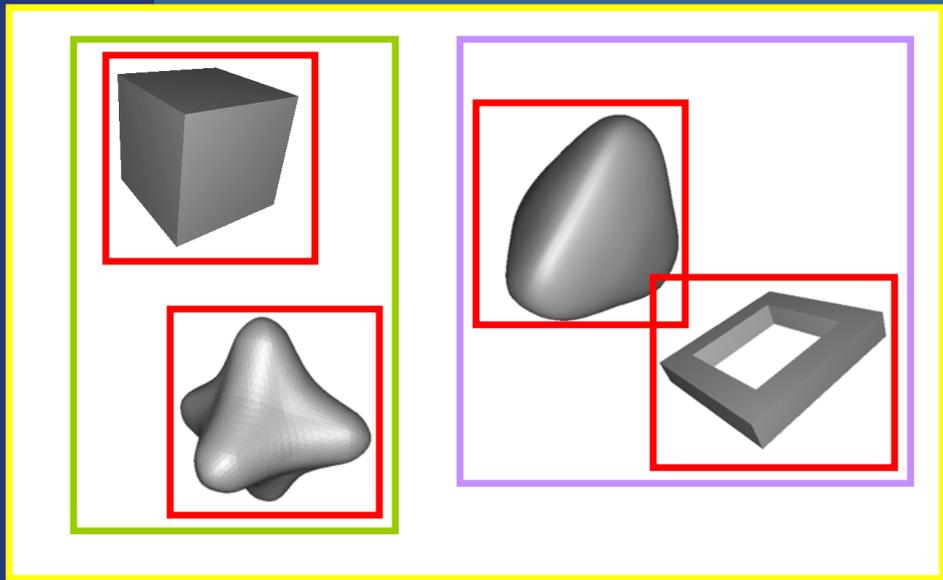
#### More Links:

- [OpenGL Quick Reference Card.pdf](#)
- [OpenGL Reference Manual 3.0](#)
- [GLSL specification 1.30](#)
- [NVIDIA G80 OpenGL Programming.](#)
- [Sample textures for download](#)
- [Bump mapping using GLSL](#)
- [Real-Time Rendering website](#)
- [OpenGL Reference Manual - The Bluebook](#)
- [The OpenGL Programming Guide - The Redbook \(html\) \(pdf\)](#)
- [All OpenGL Manuals, including release 2.0.](#)
- [GLU Reference Manual, release 1.3.](#)
- [GLUT Reference Manual, release 3. How to open a window etc.](#)
- [OpenGL.org](#)
- [GLSL manual and quick reference guide](#) and good [GLSL Tutorial](#)
- [Efficiency Issues for Ray Tracing](#), paper with optimization tricks for ray tracing.
- [A Fast Voxel Traversal Algorithm for Ray Tracing](#), paper about grid traversal.
- [MilkShape 3D](#), a free 3D-modeling application.
- [Designing a PC Game Engine](#). A paper about "game engine design"
- [L3DS](#) open C++ code for loading and rendering 3ds-files.
- [Cross Roads Converter](#) between 3D formats.
- [Converters for image, sound and 3D-models.](#)
- [Some 3D models.](#)
- [3D models, converters etc.](#)
- [commandline-converter for images.](#)
- [3D Planet](#) - some 3D models.
- [More 3D models 1](#)
- [More 3D models 2](#)
- [More 3D models 3](#)

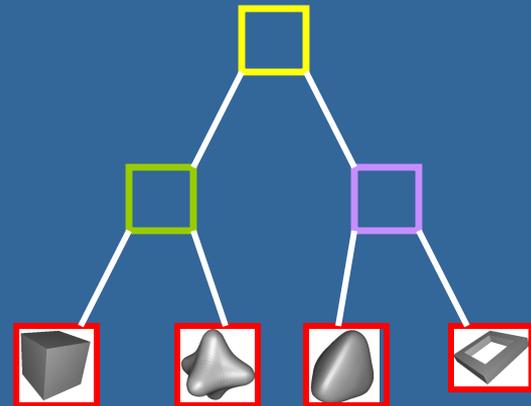
# How?

- Organizes geometry in some hierarchy

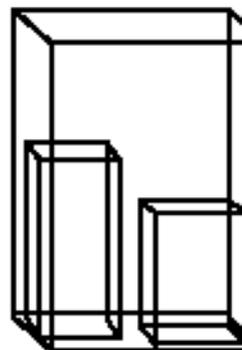
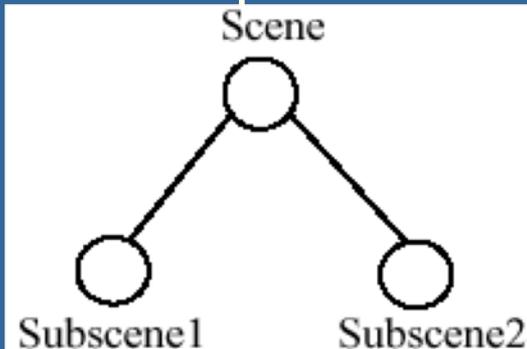
In 2D space



Data structure



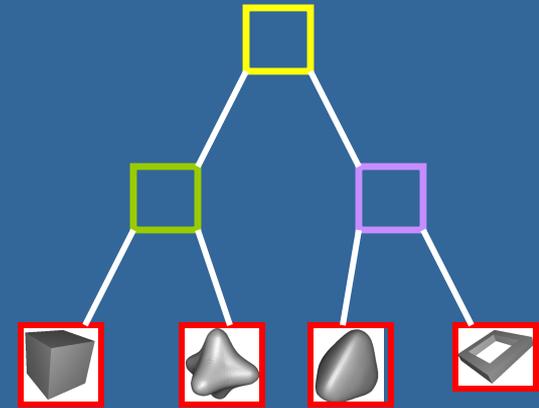
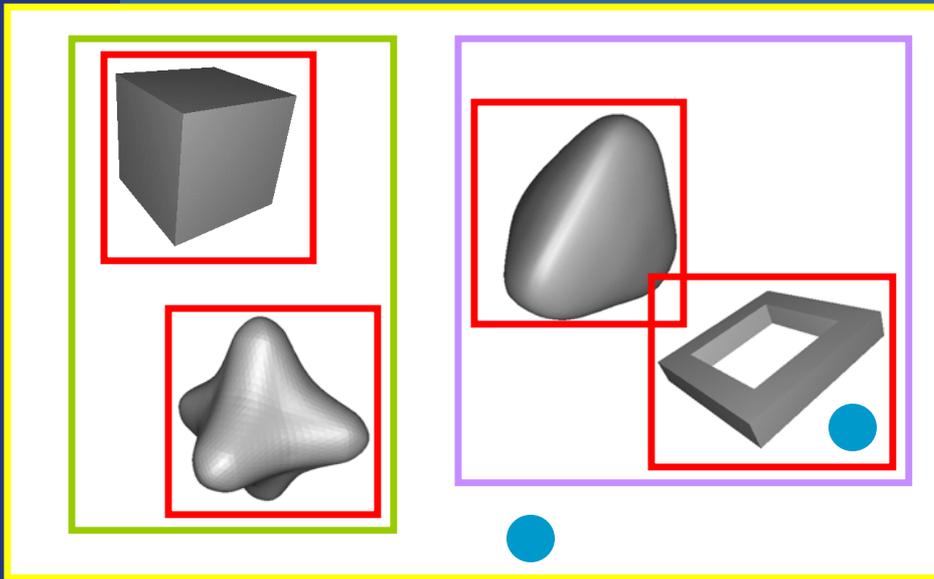
In 3D space:



# What's the point?

## An example

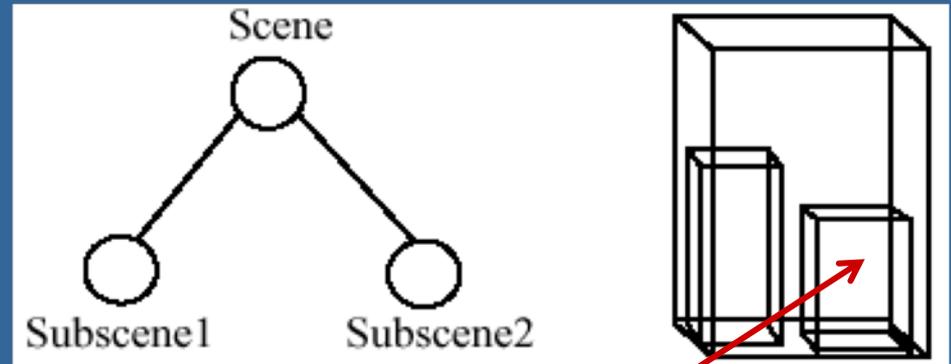
- Assume we click on screen, and want to find which object we clicked on



click!

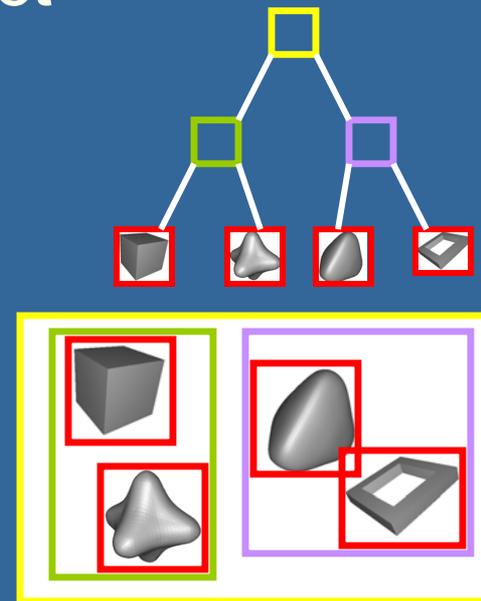
- 1) Test the root first
  - 2) Descend recursively as needed
  - 3) Terminate traversal when possible
- In general: get  $O(\log n)$  instead of  $O(n)$

# 3D example



# Bounding Volume Hierarchy (BVH)

- Most common bounding volumes (BVs):
  - Sphere
  - Boxes (AABB and OBB)
- The BV does not contribute to the rendered image -- rather, encloses an object
- The data structure is a  $k$ -ary tree
  - Leaves hold geometry
  - Internal nodes have at most  $k$  children
  - Internal nodes hold BVs that enclose all geometry in its subtree

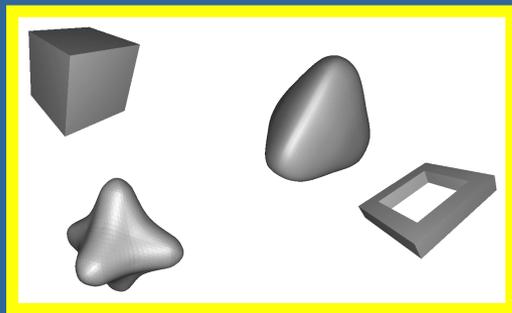


# Some facts about trees

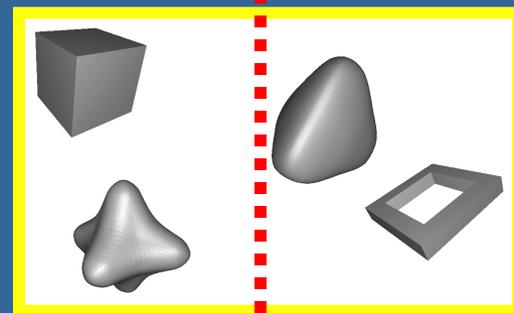
- *Height of tree,  $h$* , is longest path from root to leaf
- *A balanced tree* is full except for possibly missing leaves at level  $h$
- Height of balanced tree with  $n$  nodes:  
 $\text{floor}(\log_k(n))$
- Binary tree ( $k=2$ ) is the simplest
  - $k=4$  and  $k=8$  is quite common for computer graphics as well

# How to create a BVH? Example: BV=AABB

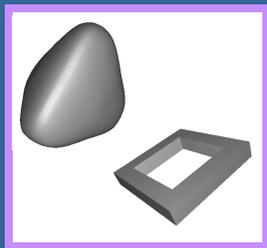
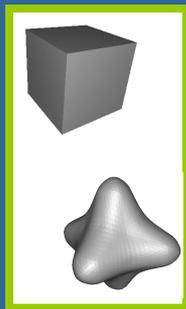
- Find minimal box, then split along longest axis



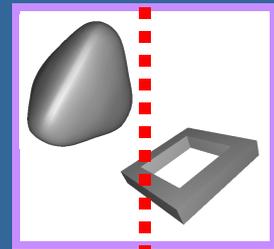
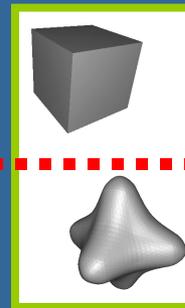
x is longest



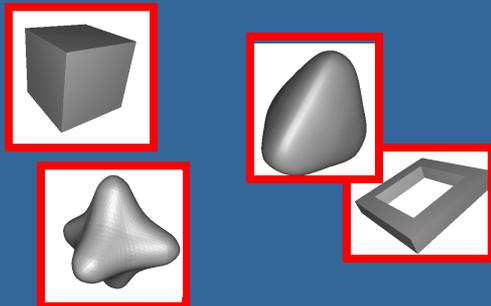
Find minimal boxes



Split along longest axis



Find minimal boxes



Called TOP-DOWN method  
Works similarly for other BVs

# Stopping criteria for Top-Down creation

- Need to stop recursion some time...
  - Either when BV is empty
  - Or when only one primitive (e.g. triangle) is inside BV
  - Or when  $<n$  primitives is inside BV
  - Or when recursion level  $l$  has been reached
- Similar criteria for BSP trees and octrees

# Example

Killzone (2004-PS2) used kd-tree / AABB-tree based system for the collision detection



Kd-tree = Axis Aligned BSP tree

# Binary Space Partitioning (BSP) Trees

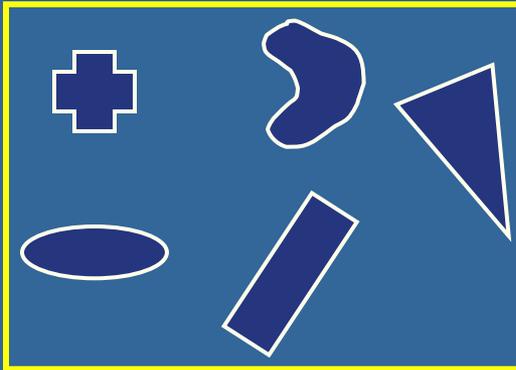
- Two different types:
  - Axis-aligned
  - Polygon-aligned
- General idea:
  - Split space with a plane
  - Divide geometry into the space it belongs
  - Done recursively
- If traversed in a certain way, we can get the geometry sorted back-to-front or front-to-back w.r.t. a camera position
  - Exact for polygon-aligned
  - Approximately for axis-aligned

- Split space with a plane
- Divide geometry into the space it belongs
- Done recursively

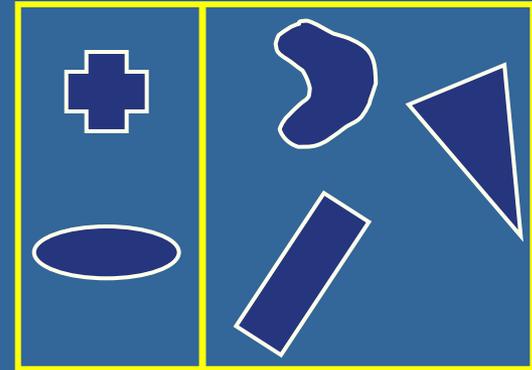
# Axis-Aligned BSP tree (1)

- Can only make a splitting plane along x,y, or z

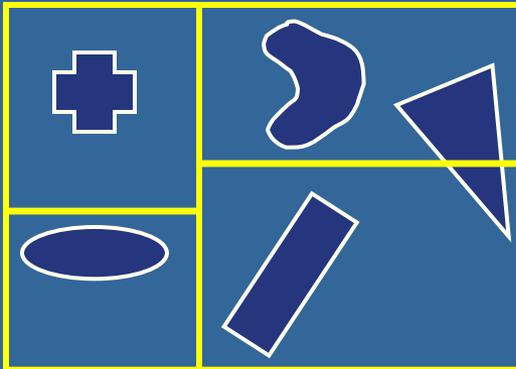
Minimal box  
→



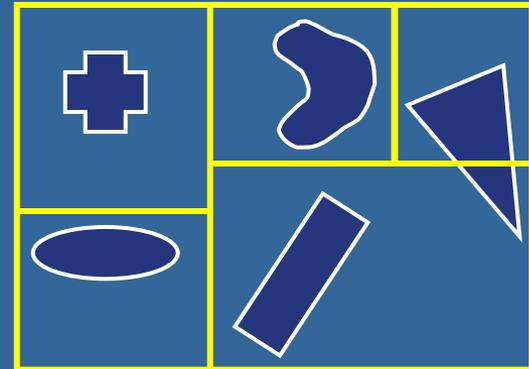
Split along plane  
→



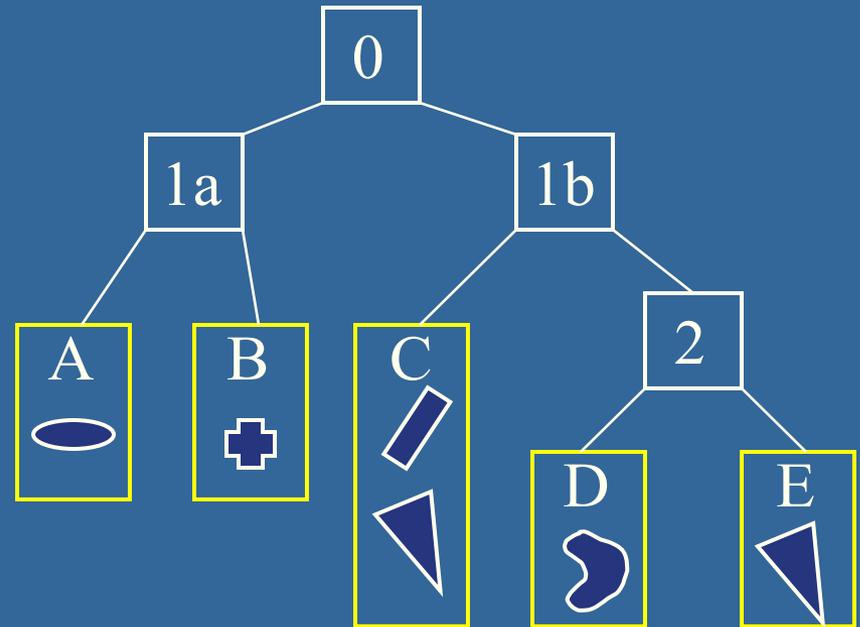
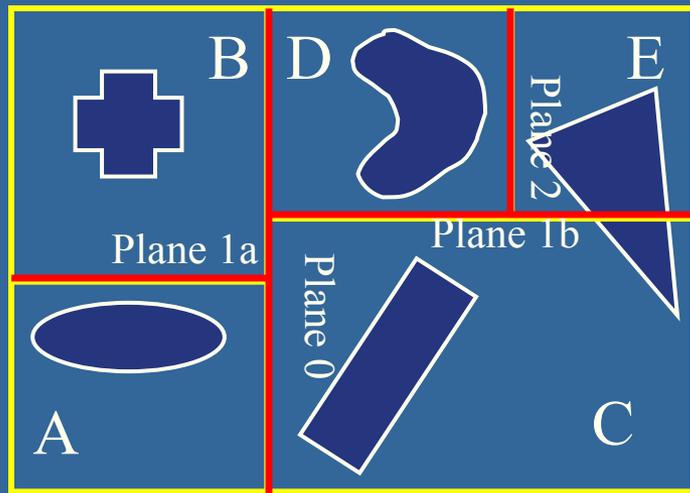
Split along plane  
→



Split along plane  
→



# Axis-Aligned BSP tree (2)

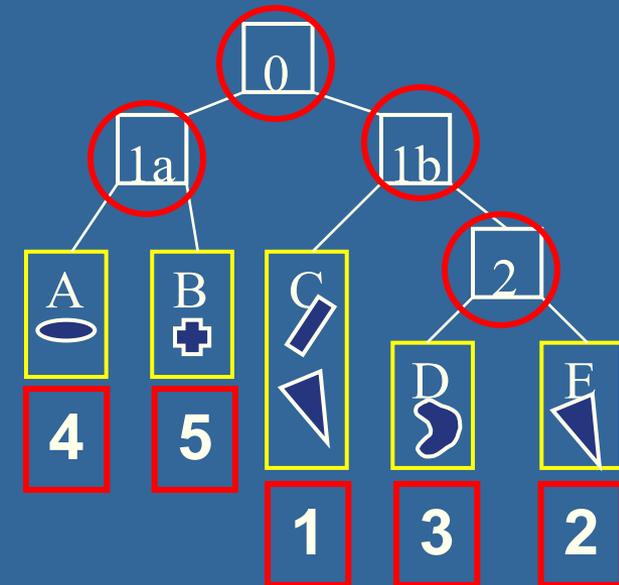
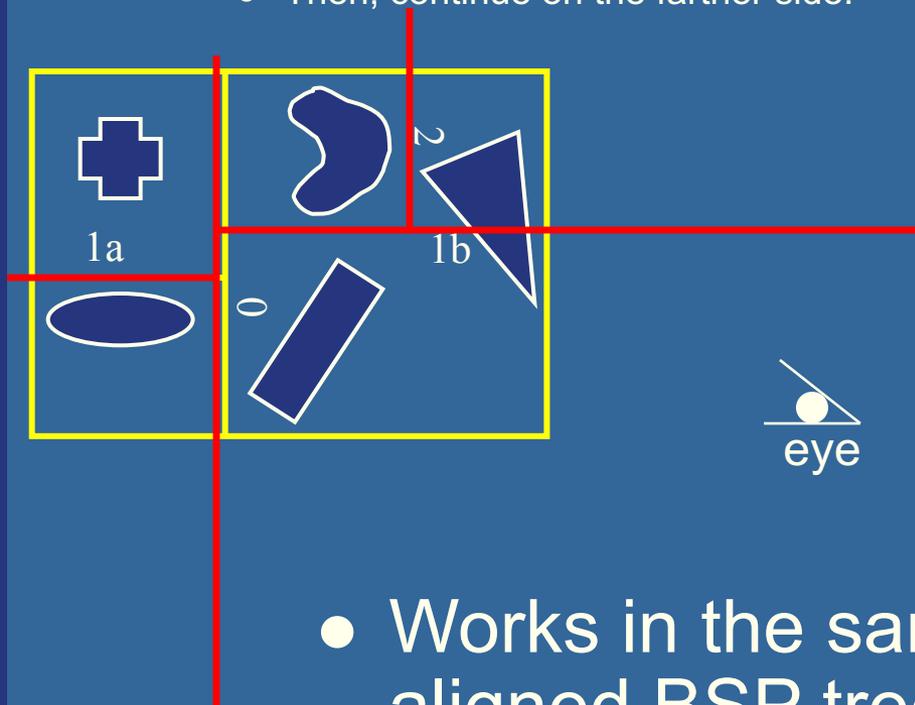


- Each internal node holds a divider plane
- Leaves hold geometry
- Differences compared to BVH
  - BSP tree encloses entire space and provides sorting
  - The BV hierarchy can have spatially overlapping nodes(no sort)
  - BVHs can use any desirable type of BV

# Axis-aligned BSP tree

## Rough sorting

- Test the planes, recursively from root, against the point of view. For each traversed node:
  - If node is leaf, draw the node's geometry
  - else
    - Continue traversal on the "hither" side with respect to the eye (to sort front to back)
    - Then, continue on the farther side.

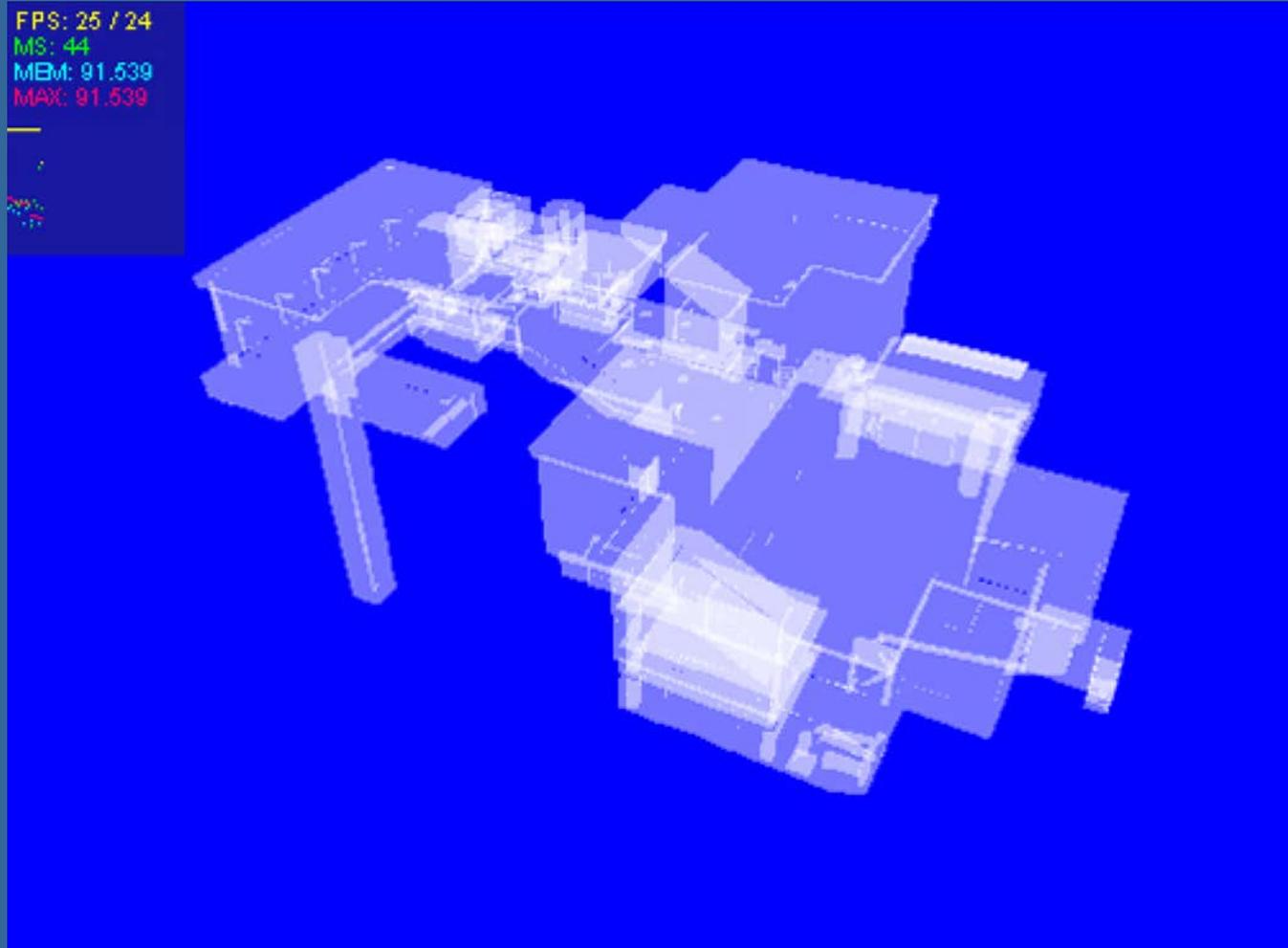


- Works in the same way for polygon-aligned BSP trees --- but that gives exact sorting

# Polygon Aligned BSP tree – Quake 2

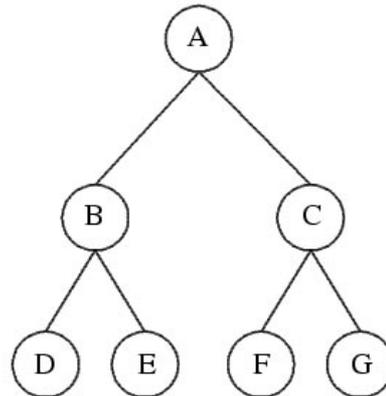
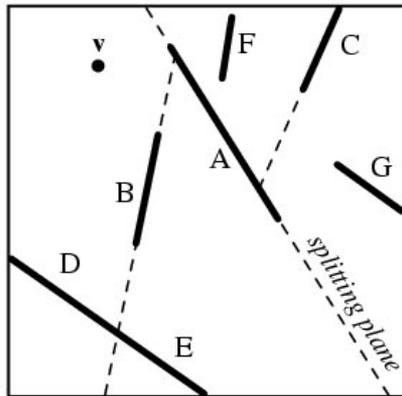


# Example – Quake 2



# Polygon-aligned BSP tree

- Allows exact sorting
- Very similar to axis-aligned BSP tree
  - But the splitting plane are now located in the planes of the triangles



```
Drawing Back-to-Front {  
    recurse on farther side of P;  
    Draw P;  
    Recurse on hither side of P;  
}  
//Where hither and  
farther are with respect  
to viewpoint  $v$ 
```

# Algorithm for BSP trees

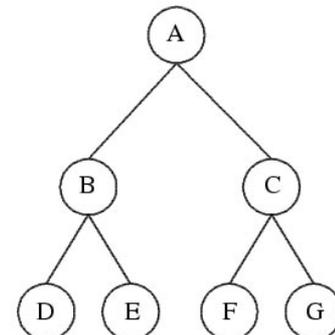
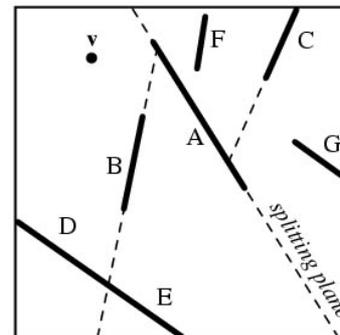
```
class BSPtree:  
    Polygon P;  
    BSPtree behindP;  
    BSPtree frontOfP;
```

```
Tree CreateBSP(PolygonList L) {  
    If L empty, return empty tree;  
    Else:  
        T->P = arbitrary polygon in L.  
        T->behindP = CreateBSP(polygons behind P)  
        T->frontOfP = CreateBSP(polygons in front of P)  
    Return T.  
}
```

Drawing Back-to-Front:

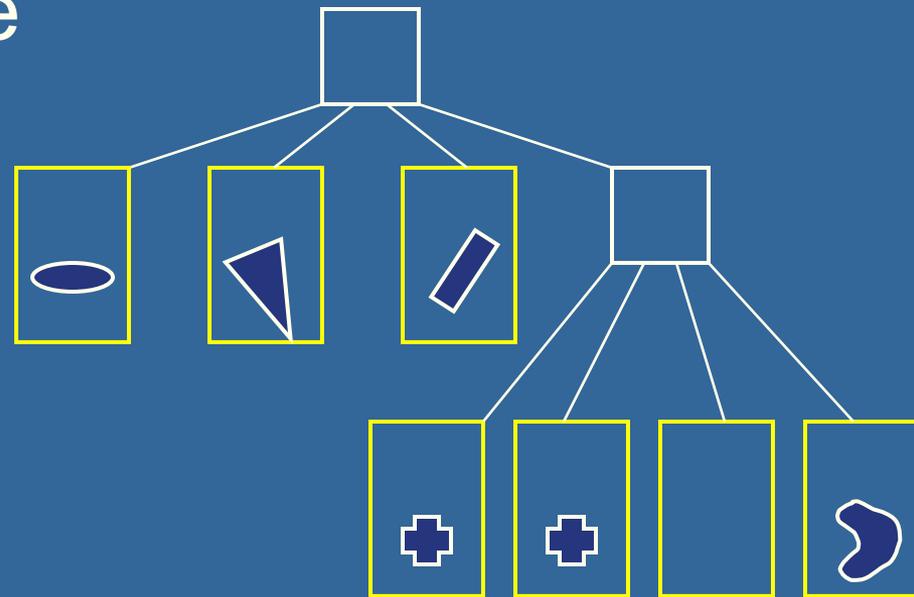
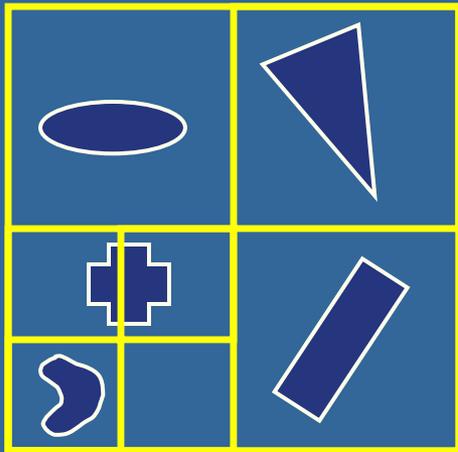
```
void DrawBSP(Tree t) {  
    If (t==NULL) return;  
    If eye front of polygon t->P:  
        DrawBSP(t->behindP);  
        Draw P;  
        DrawBSP(t->frontOfP);  
    Else:  
        DrawBSP(t->frontOfP);  
        Draw P;  
        DrawBSP(t->behindP);  
}
```

```
Drawing Back-to-Front {  
    recurse on farther side of P;  
    Draw P;  
    Recurse on hither side of P;  
}
```



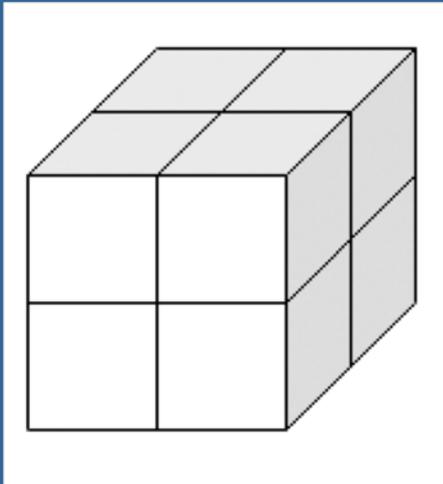
# Octrees (1)

- A bit similar to axis-aligned BSP trees
- Will explain the quadtree, which is the 2D variant of an octree

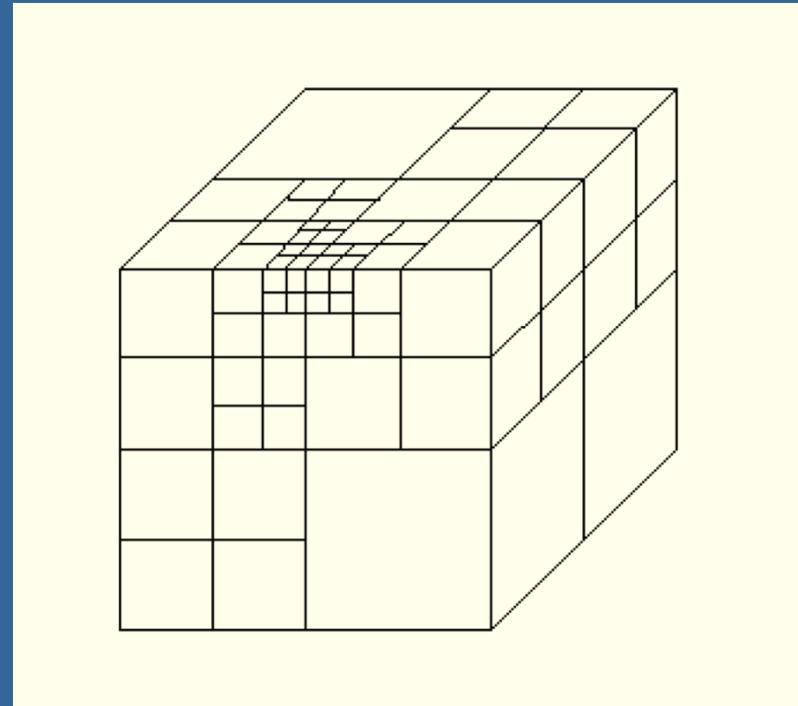


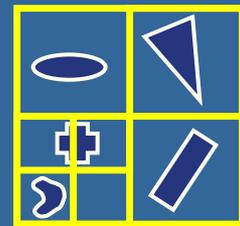
- In 3D, each square (or rectangle) becomes a box, and 8 children

# Example of Octree

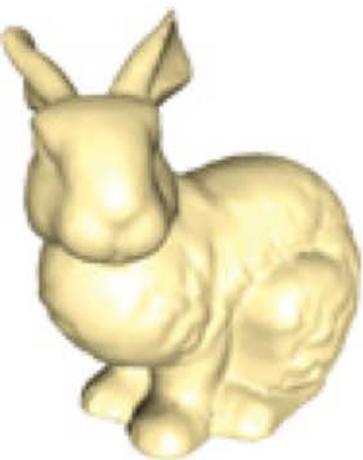


Recursively split space  
in eight parts – equally  
along x,y,z dimension  
simultaneously for each  
level

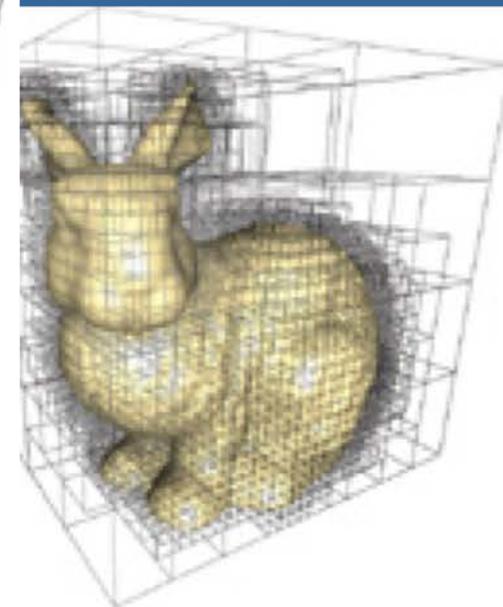
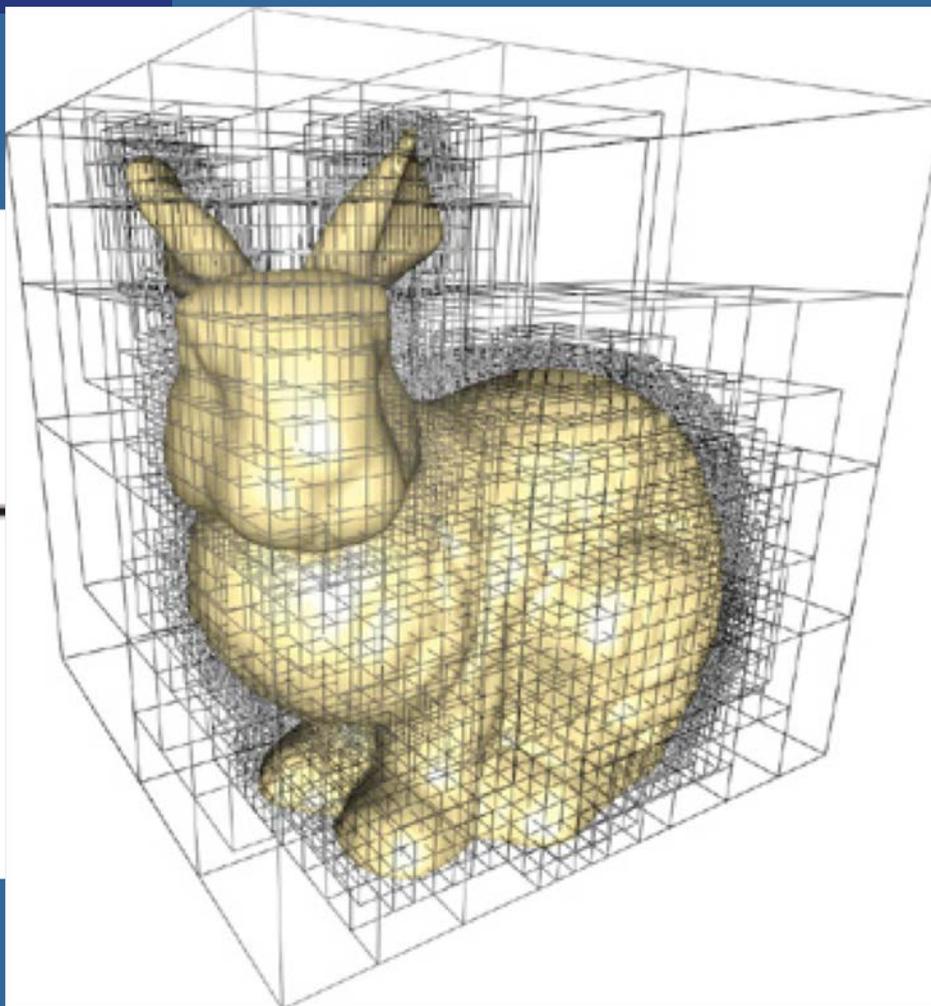




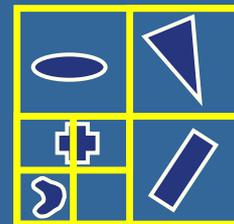
# Example of octree



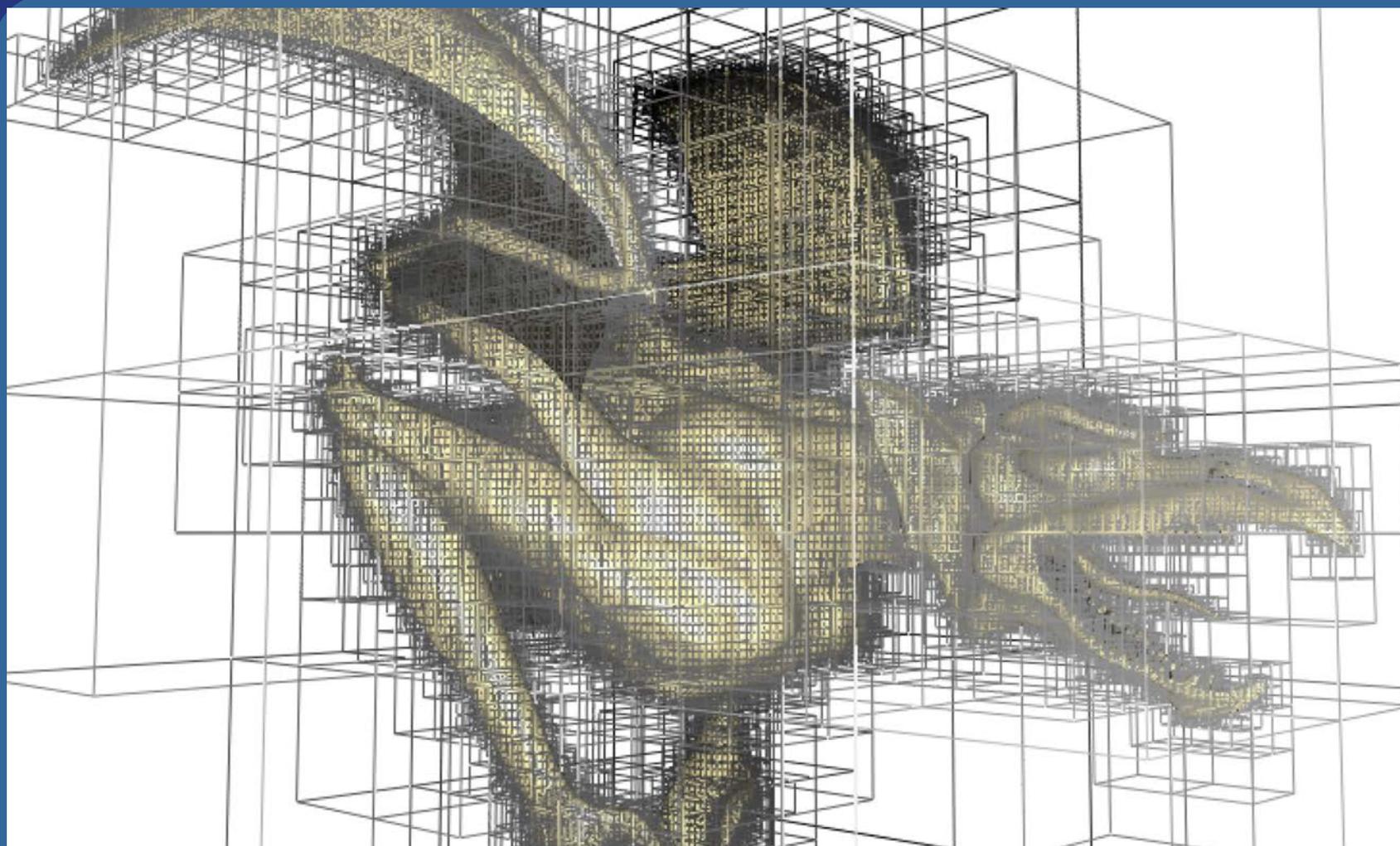
(a)



(c)



# Example of octree

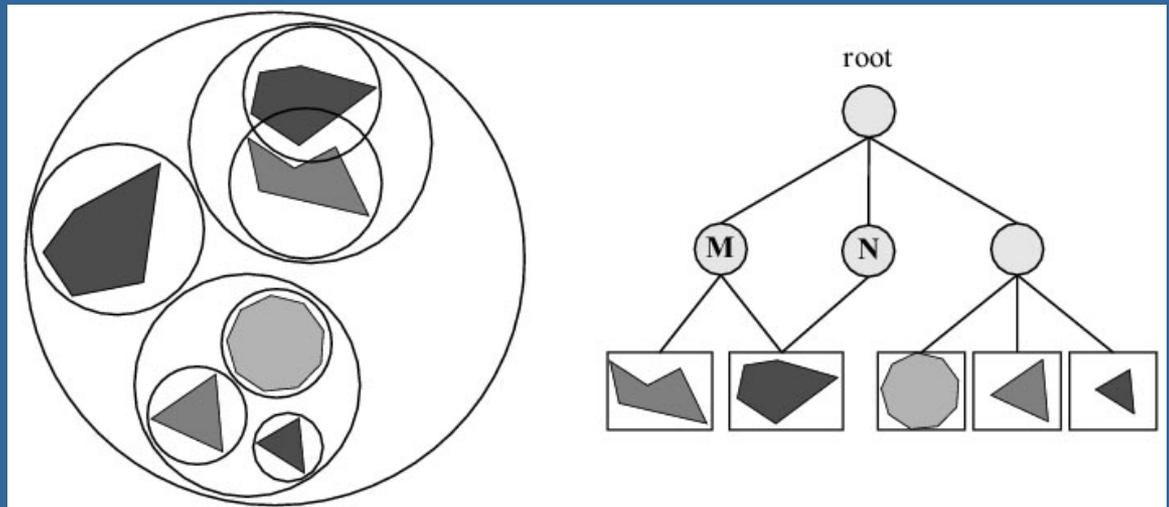


## Octrees (2)

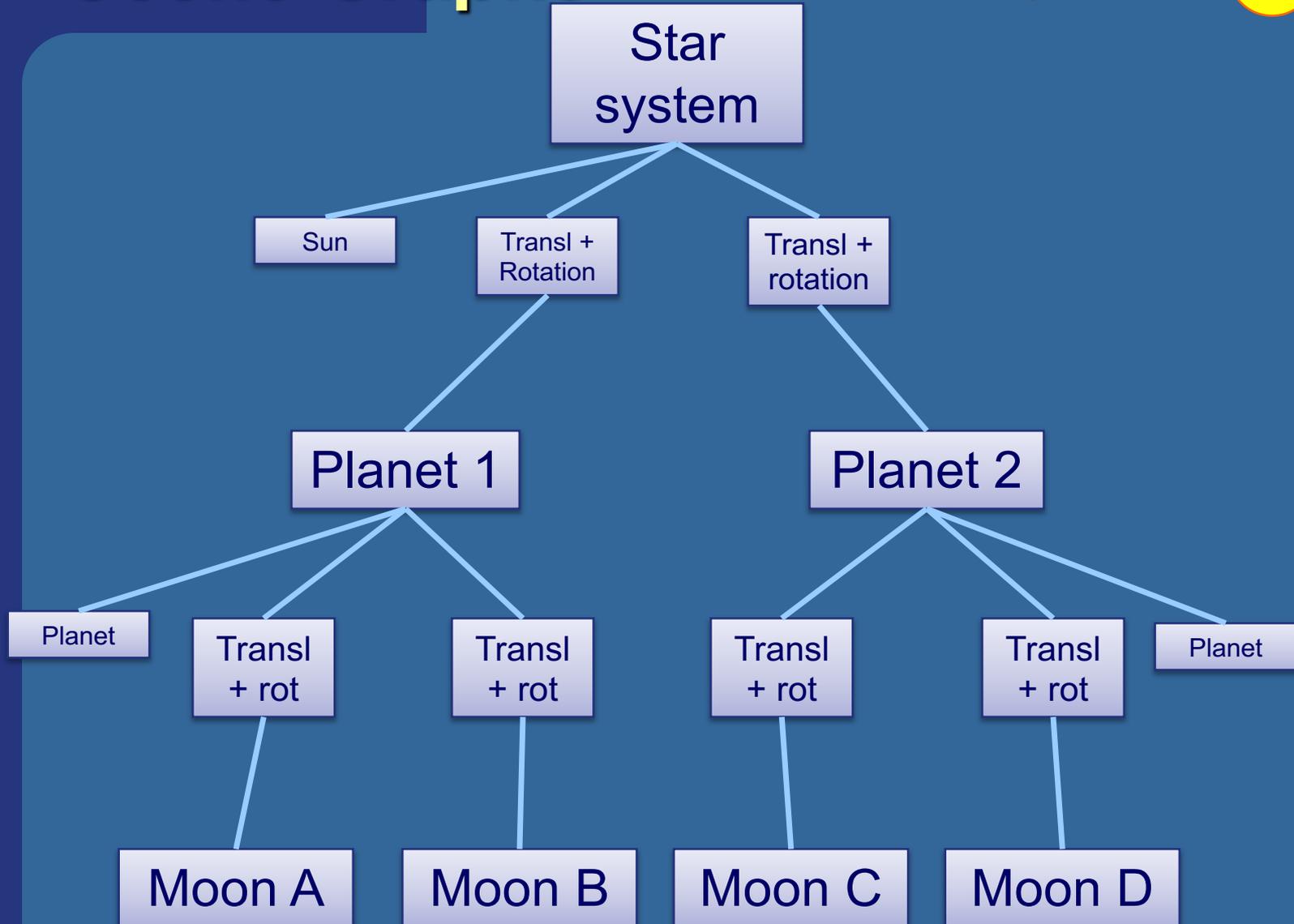
- Expensive to rebuild (BSPs are too)
- Octrees can be used to
  - Speed up ray tracing
  - Faster picking
  - Culling techniques
  - Are not used that often in real-time contexts
    - ~~An exception is loose octrees~~

# Scene graphs

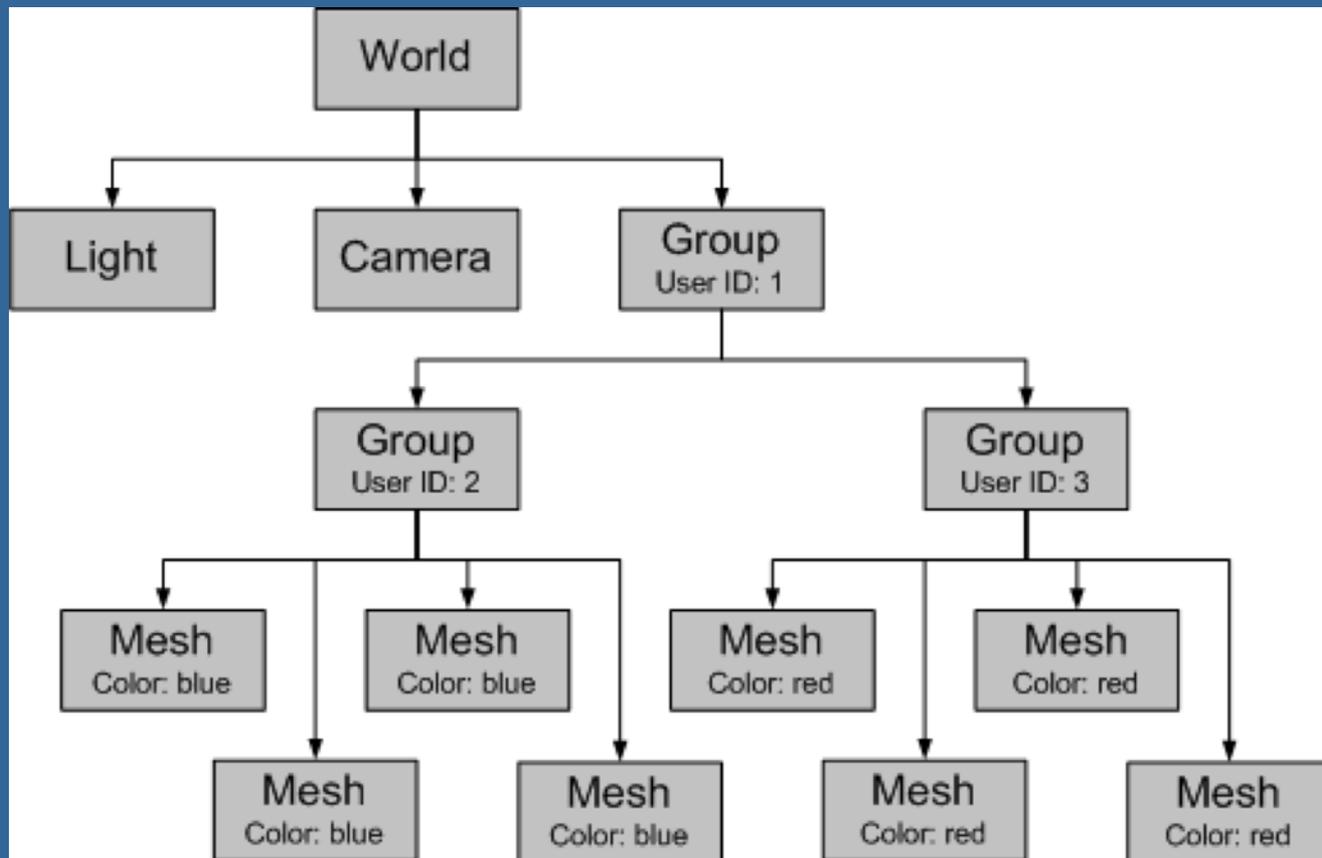
- BVH is the data structure that is used most often
  - Simple to understand
  - Simple code
- However, it stores just geometry
  - Rendering is more than geometry
- The scene graph is an extended BVH with:
  - Lights
  - Materials
  - Transforms
  - And more
  - Typically the logical structure



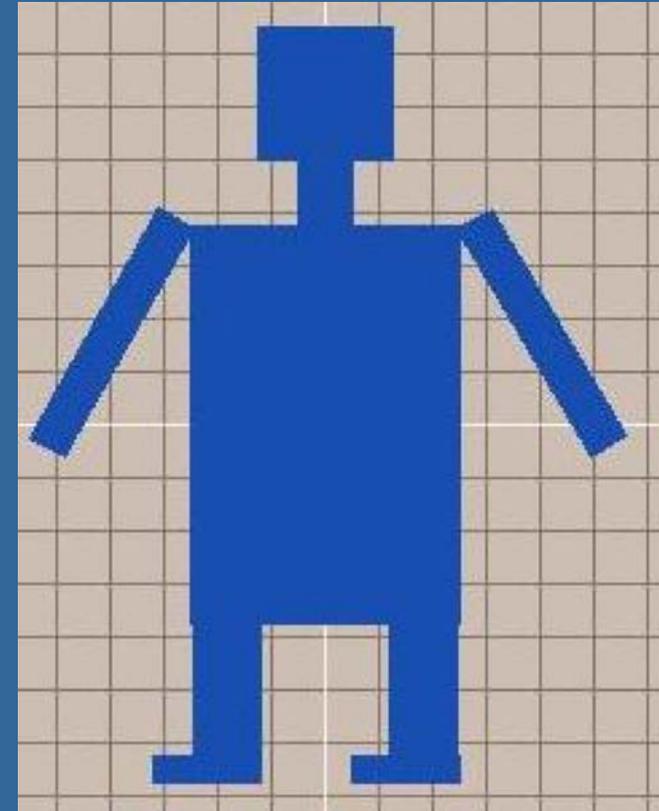
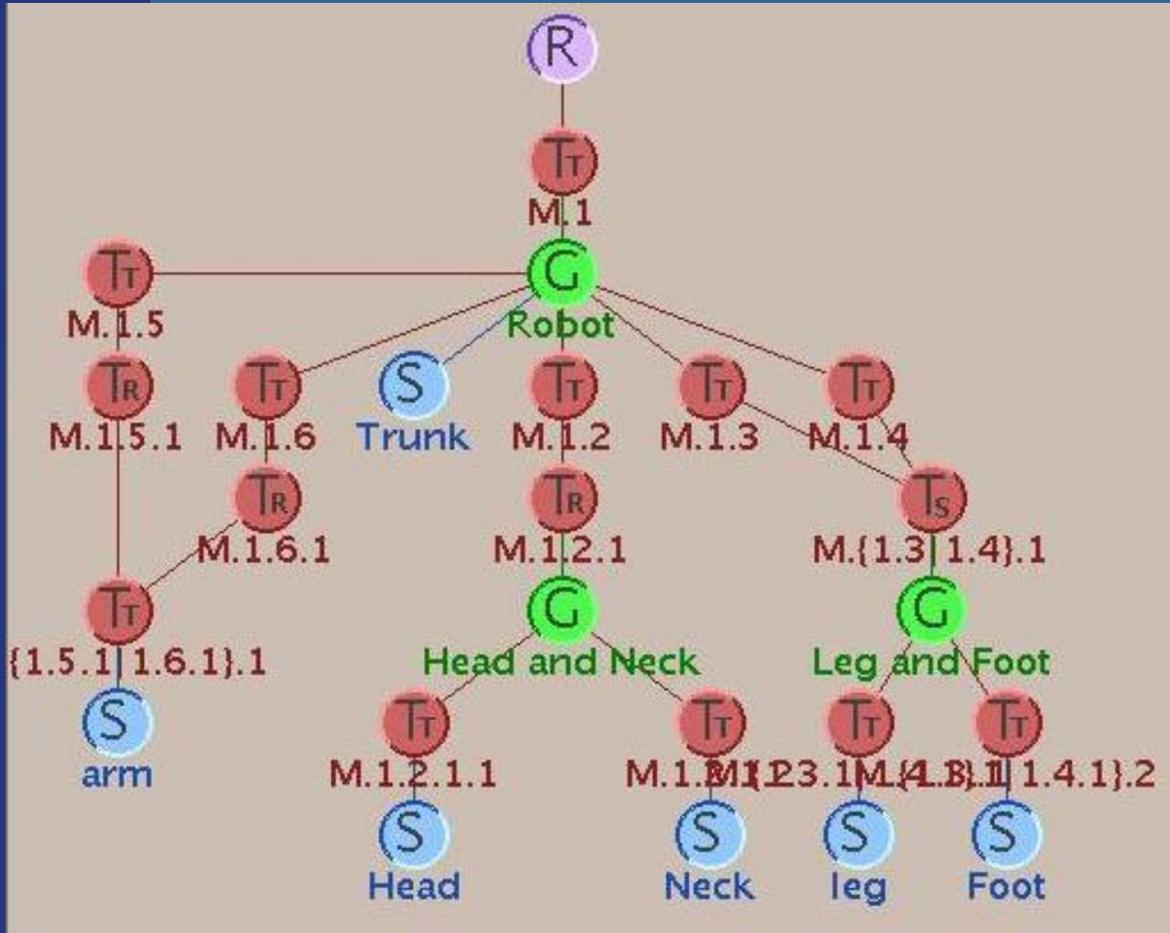
# Scene Graphs



# Scene Graphs



# Scene Graphs



# Speed-Up Techniques

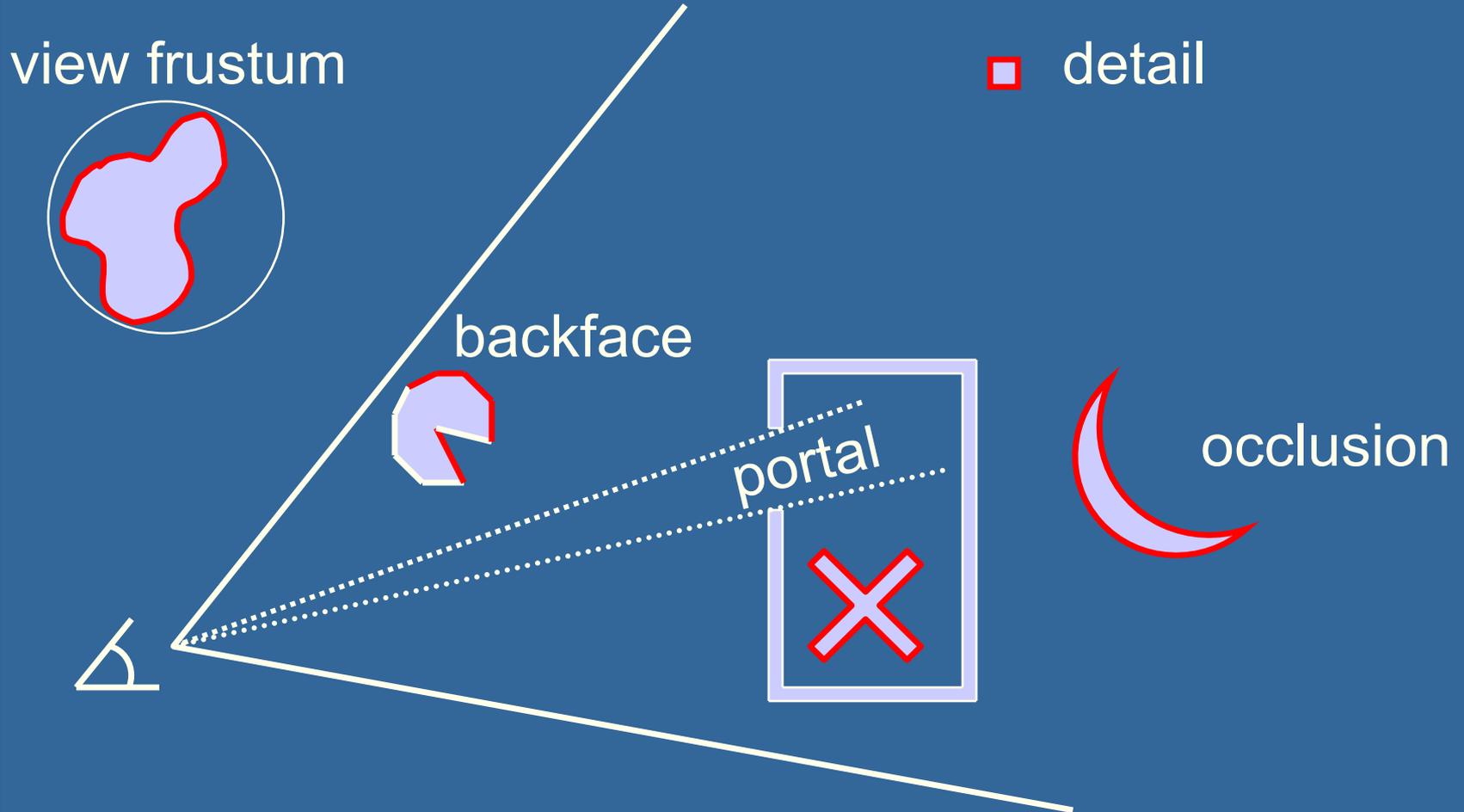
- Spatial data structures are used to speed up rendering and different queries
- Why more speed?
- Graphics hardware 2x faster in 6-12 months!
- Wait... then it will be fast enough!
- NOT!
- We will never be satisfied
  - Screen resolution: angular resolution in “gula fläcken”  
~0.001 degree (eye sweeps scene)
    - Apple’s retina screen: 2880 x 1800
  - Realism: global illumination
  - Geometrical complexity: no upper limit!

# What we'll treat now

- Culling techniques
- Level-of-detail rendering (LODs)
- “To cull” means “to select from group”
  - “Sort out”, “remove”, “cut away”, something picked out and put aside as inferior.
- In graphics context: do not process data that will not contribute to the final image

# Different culling techniques

(red objects are skipped)

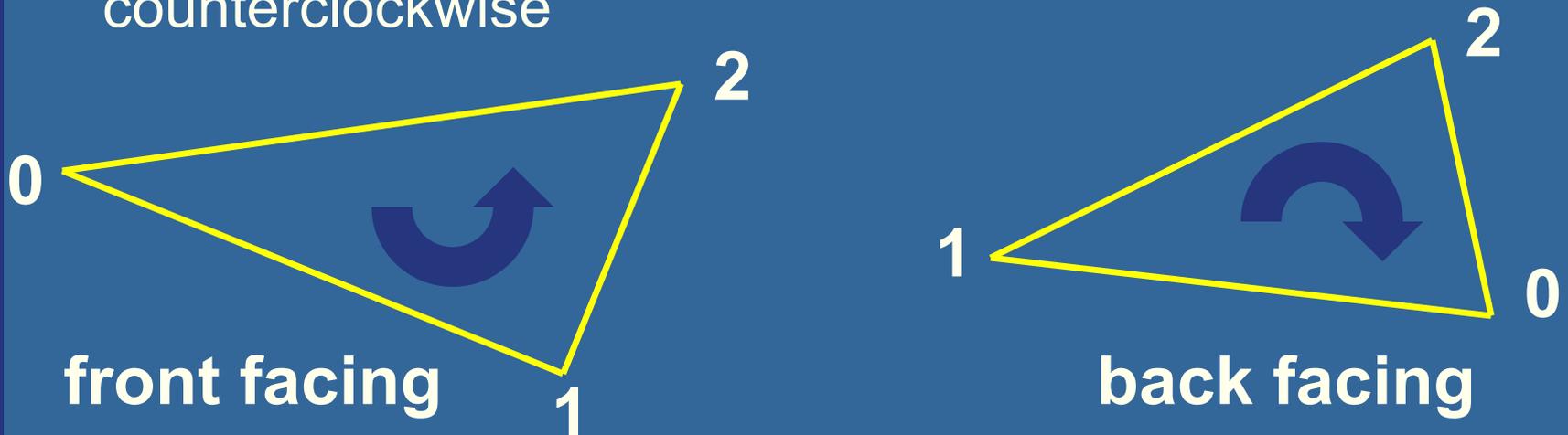


# Backface Culling

- Simple technique to discard polygons that faces away from the viewer
- Can be used for:
  - closed surface (example: sphere)
  - or whenever we know that the backfaces never should be seen (example: walls in a room)
- Two methods (screen space, eye space)
- Which stages benefits?
  - Rasterizer stage

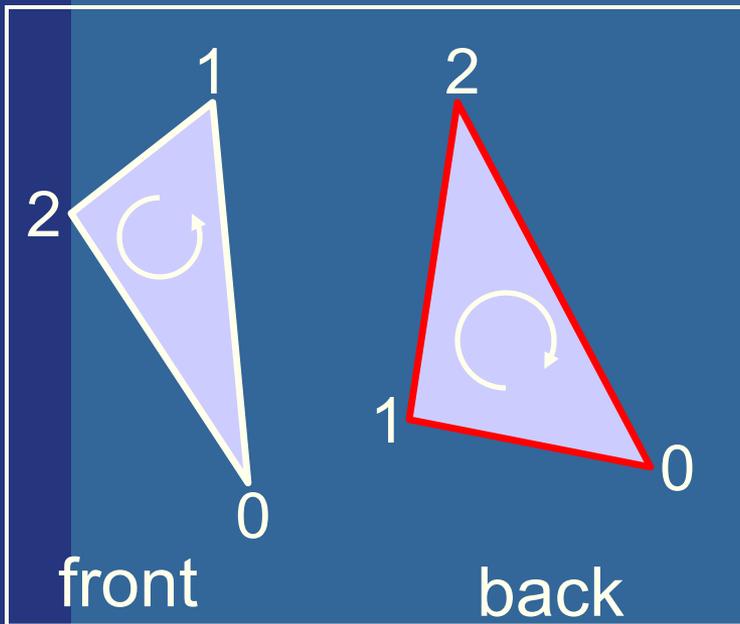
# Backface culling (cont' d)

- Often implemented for you in the API
- OpenGL:
  - `glCullFace(GL_BACK)` ;
  - `glEnable(GL_CULL_FACE)` ;
- How to determine what faces away?
- First, must have consistently oriented polygons, e.g., counterclockwise

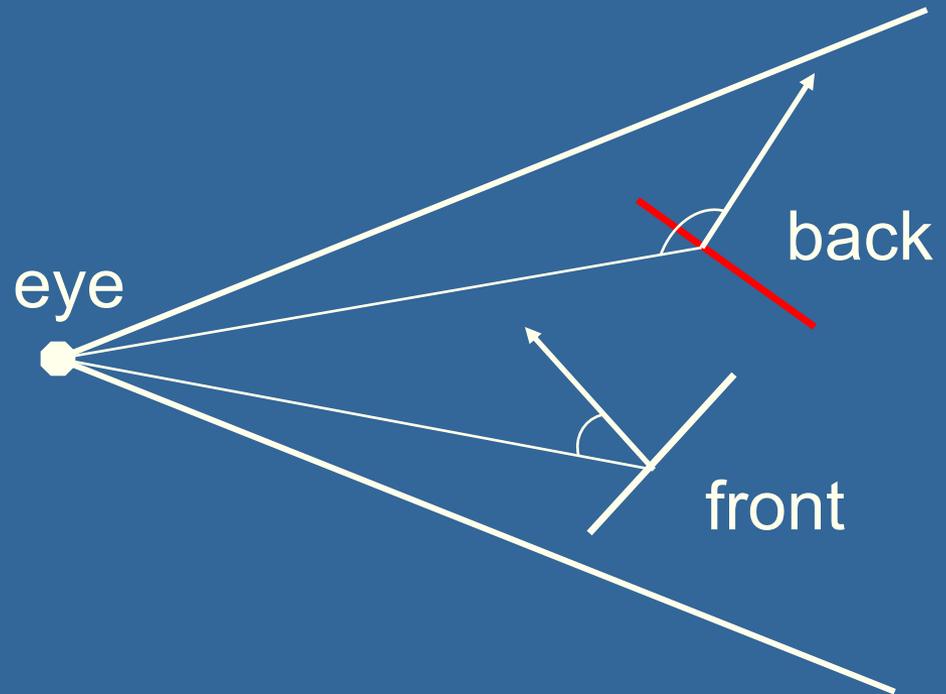


# How to cull backfaces

- Two ways in different spaces:



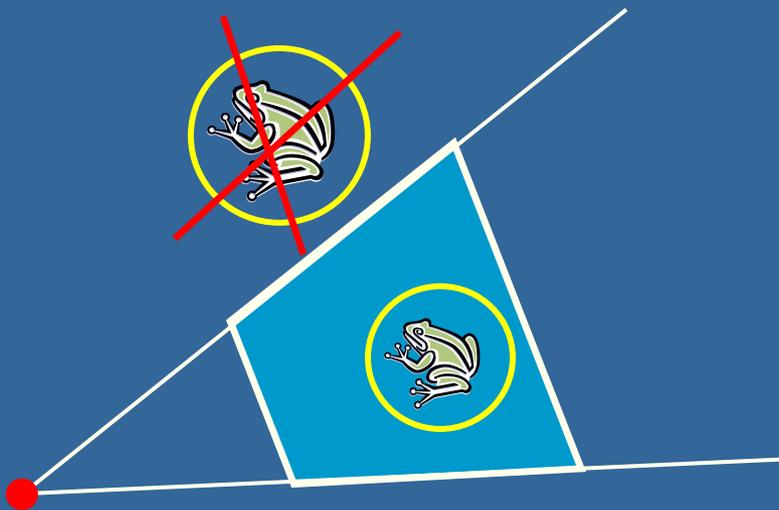
screen space



eye space

# View-Frustum Culling

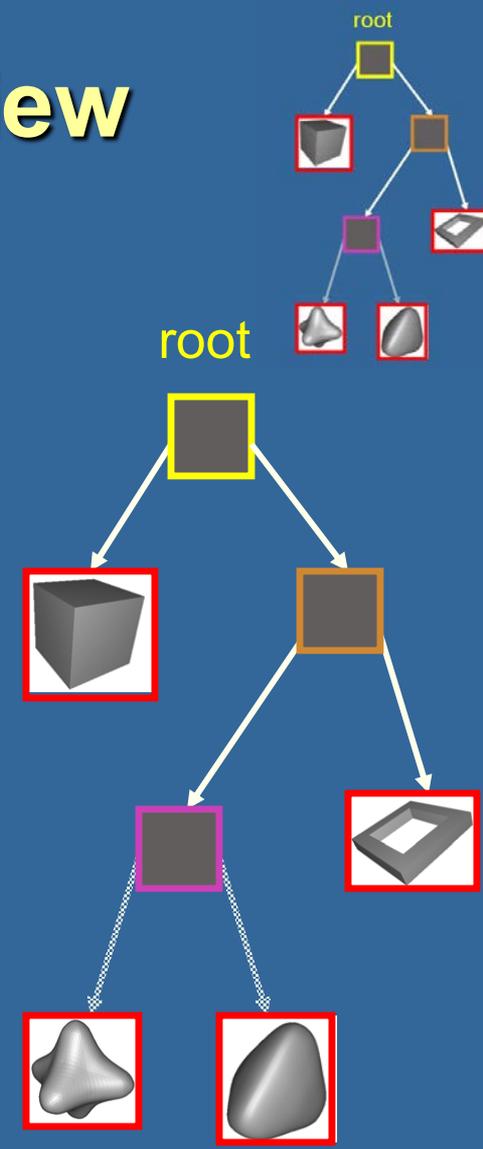
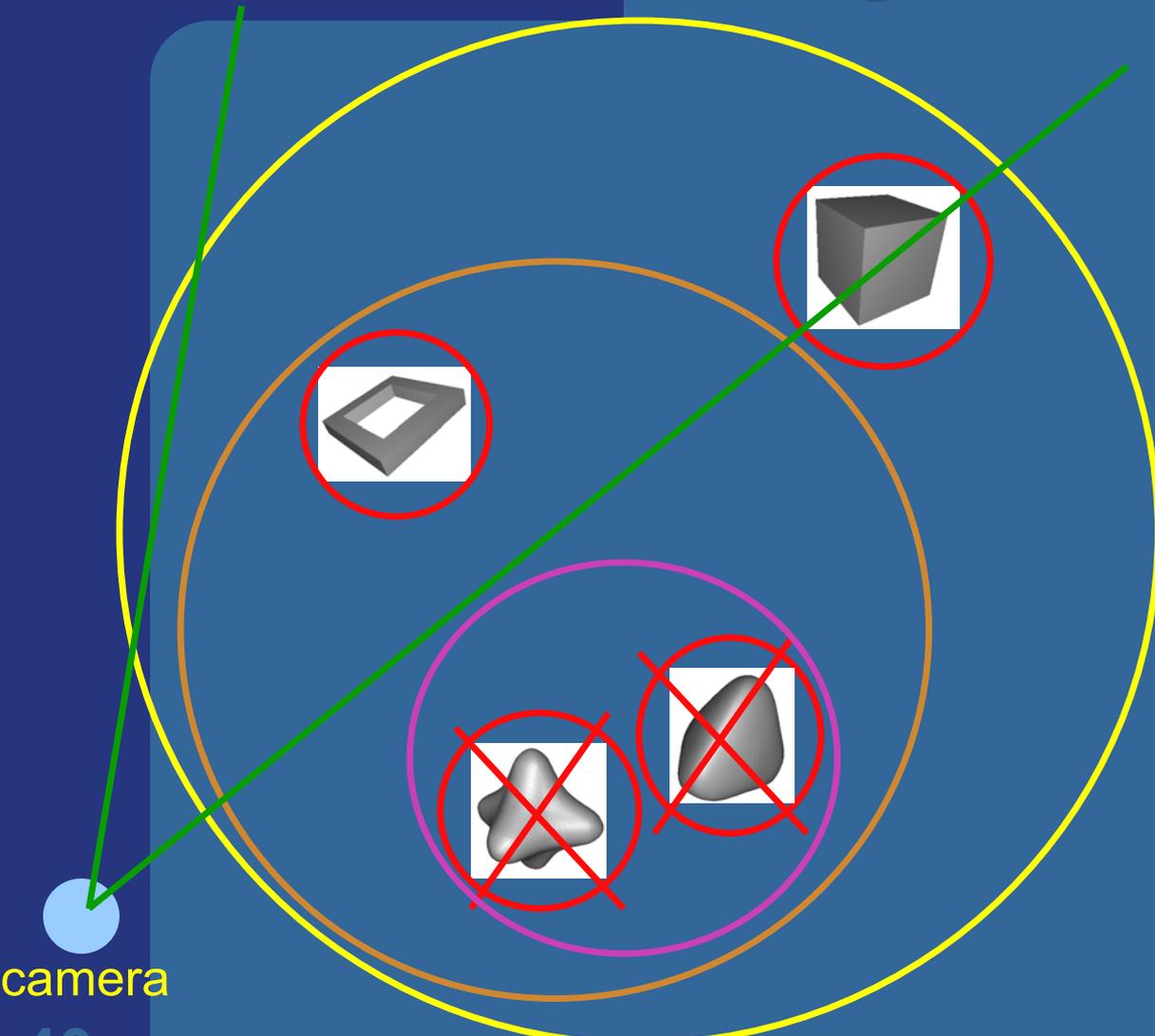
- Bound every “natural” group of primitives by a simple volume (e.g., sphere, box)
- If a bounding volume (BV) is outside the view frustum, then the entire contents of that BV is also outside (not visible)



# Can we accelerate view frustum culling further?

- Do what we always do in graphics...
- Use a hierarchical approach, e.g., a spatial data structure (BVH, BSP)
- Which stages benefits?
  - Geometry and Rasterizer
  - Possibly also bus between CPU and Geometry

# Example of Hierarchical View Frustum Culling



Refined view frustum culling:  
frustum gets smaller for each door

# Portal Culling

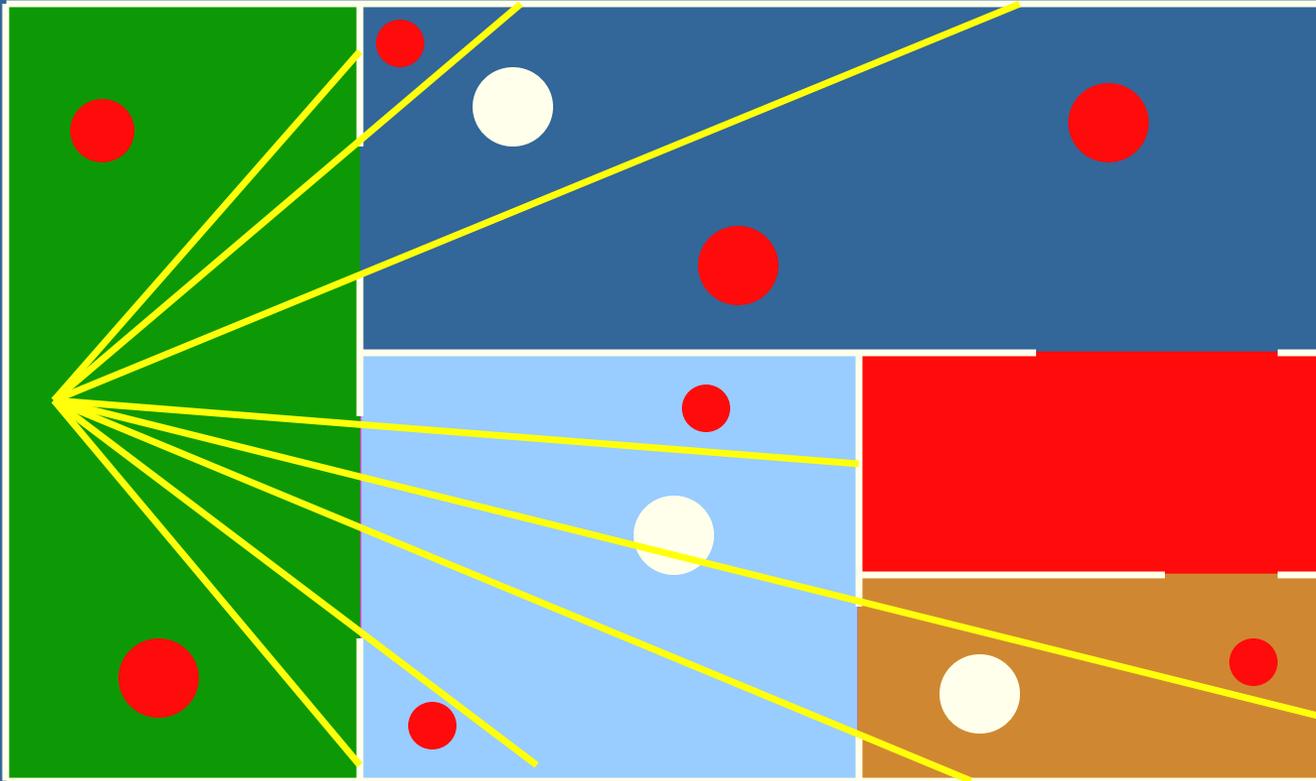
Images courtesy of David P. Luebke and Chris Georges



- Average: culled 20-50% of the polys in view
- Speedup: from slightly better to 10 times

# Portal culling example

- In a building from above
- Circles are objects to be rendered



# Portal Culling Algorithm (1)

- "Recursively do VFC through visible portals (i.e., doors & mirrors)"

## Algorithm:

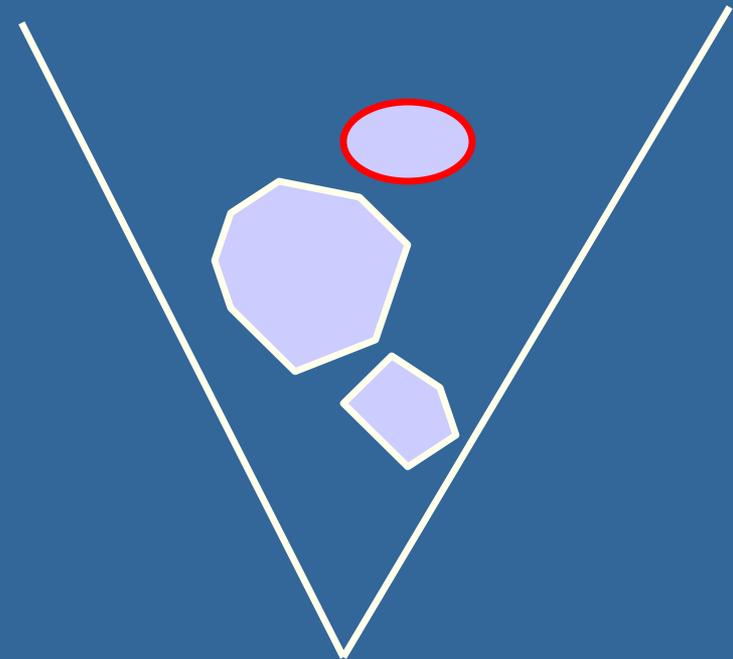
- Divide into cells with portals (build graph)
- For each frame:
  - Locate cell of viewer and init 2D AABB to whole screen
  - \* Render current cell with View Frustum culling w.r.t. AABB
  - Traverse to closest cells (through portals)
  - Intersection of AABB & AABB of traversed portal
  - Goto \*

# Portal Culling Algorithm (2)

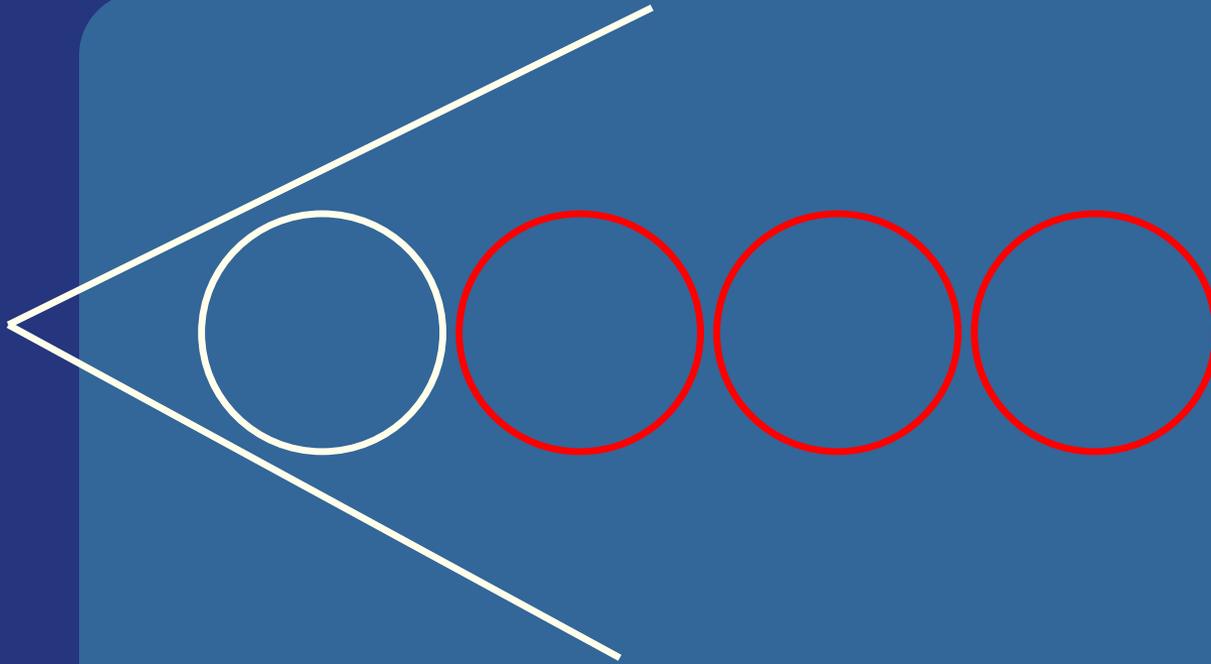
- When to exit:
  - When the current AABB is empty
  - When we do not have enough time to render a cell (“far away” from the viewer)
- Also: mark rendered objects

# Occlusion Culling

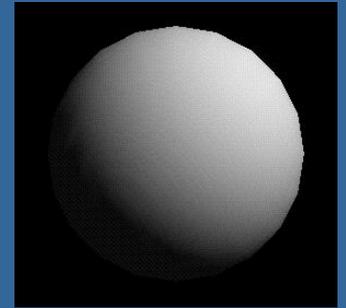
- Main idea: Objects that lies completely “behind” another set of objects can be culled
- Hard problem to solve efficiently
- Has been lots of research in this area
  - OpenGL: “Occlusion Queries”



# Example



final image



- Note that “Portal Culling” is type of occlusion culling

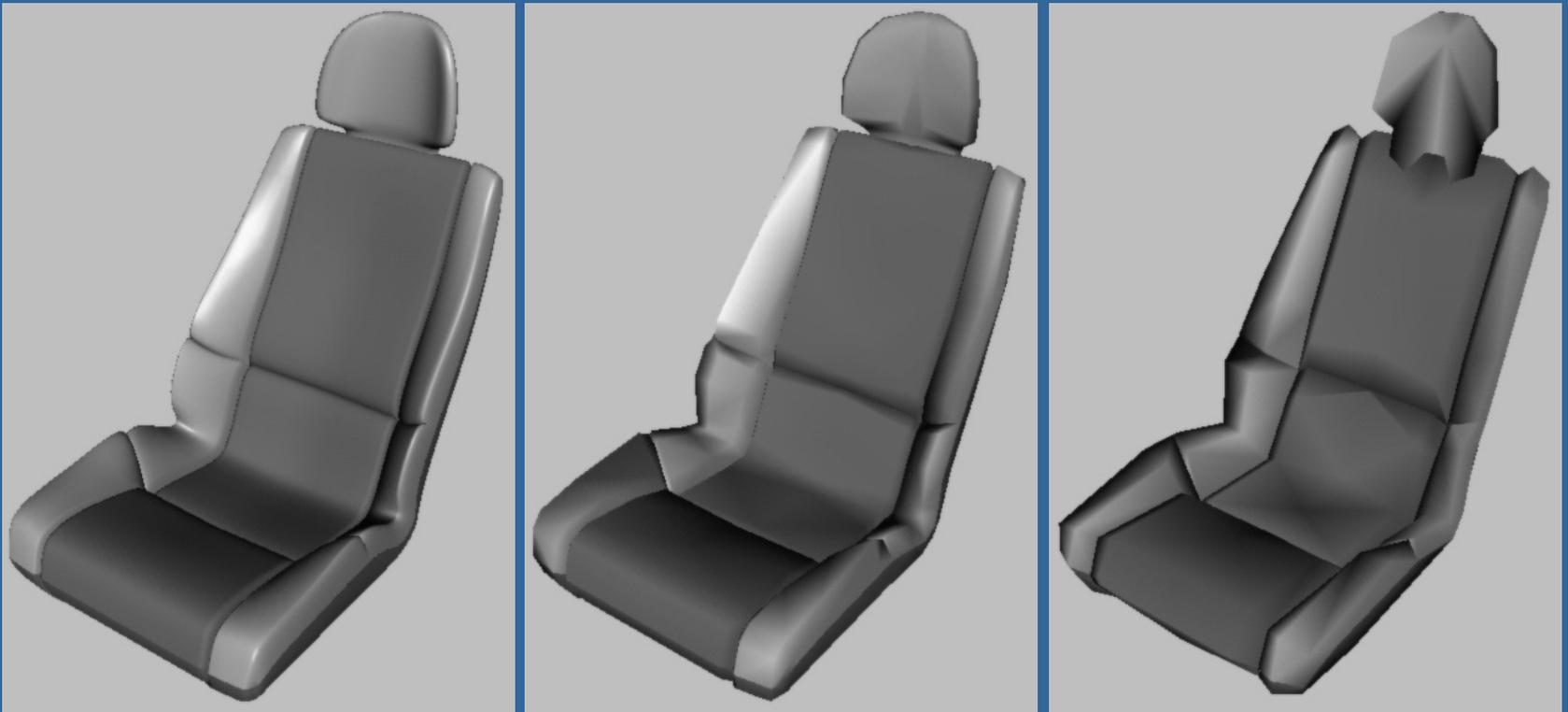
# Occlusion culling algorithm

Use some kind of occlusion representation  $O_R$

```
for each object  $g$  do:  
    if( not Occluded( $O_R, g$ ))  
        render( $g$ );  
        update( $O_R, g$ );  
    end;  
end;
```

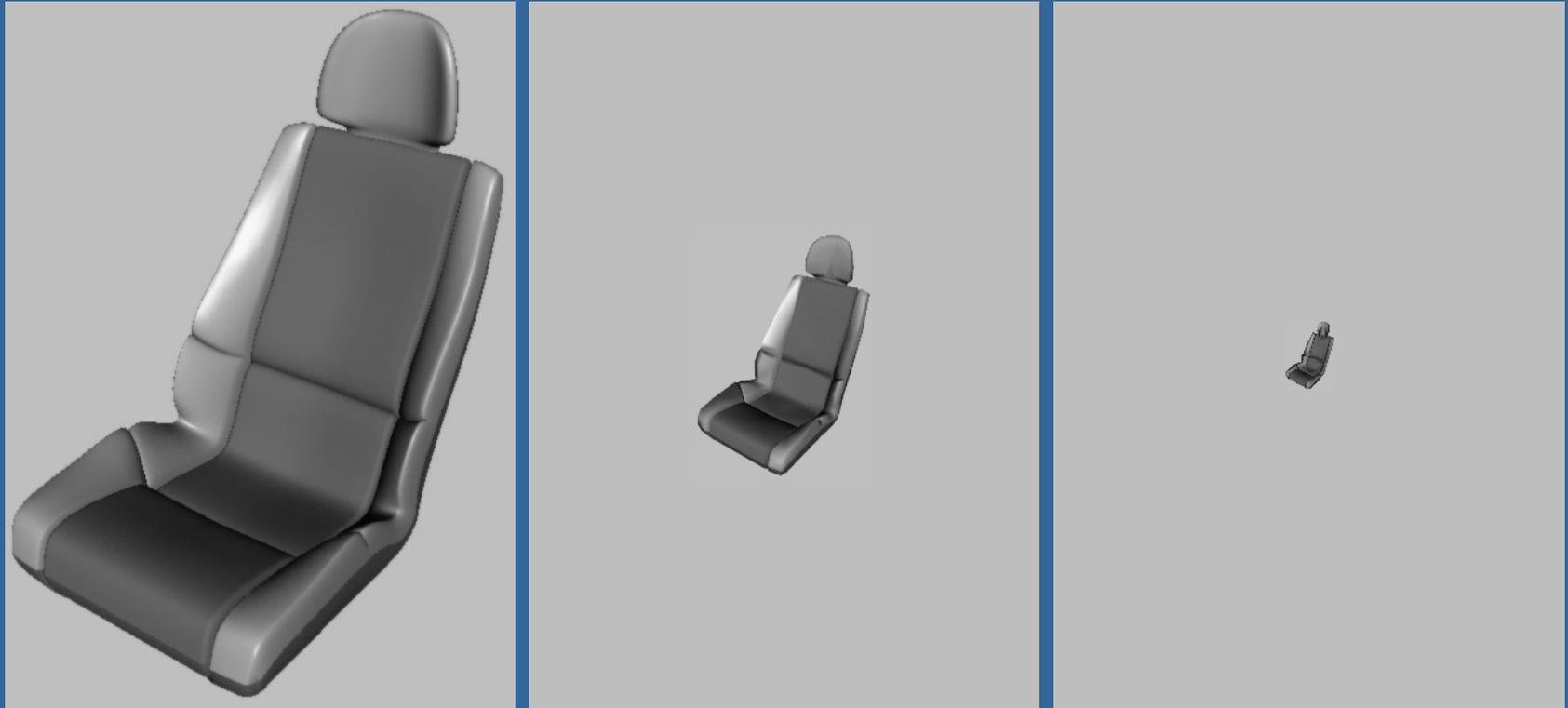
# Level-of-Detail Rendering

- Use different levels of detail at different distances from the viewer
- More triangles closer to the viewer



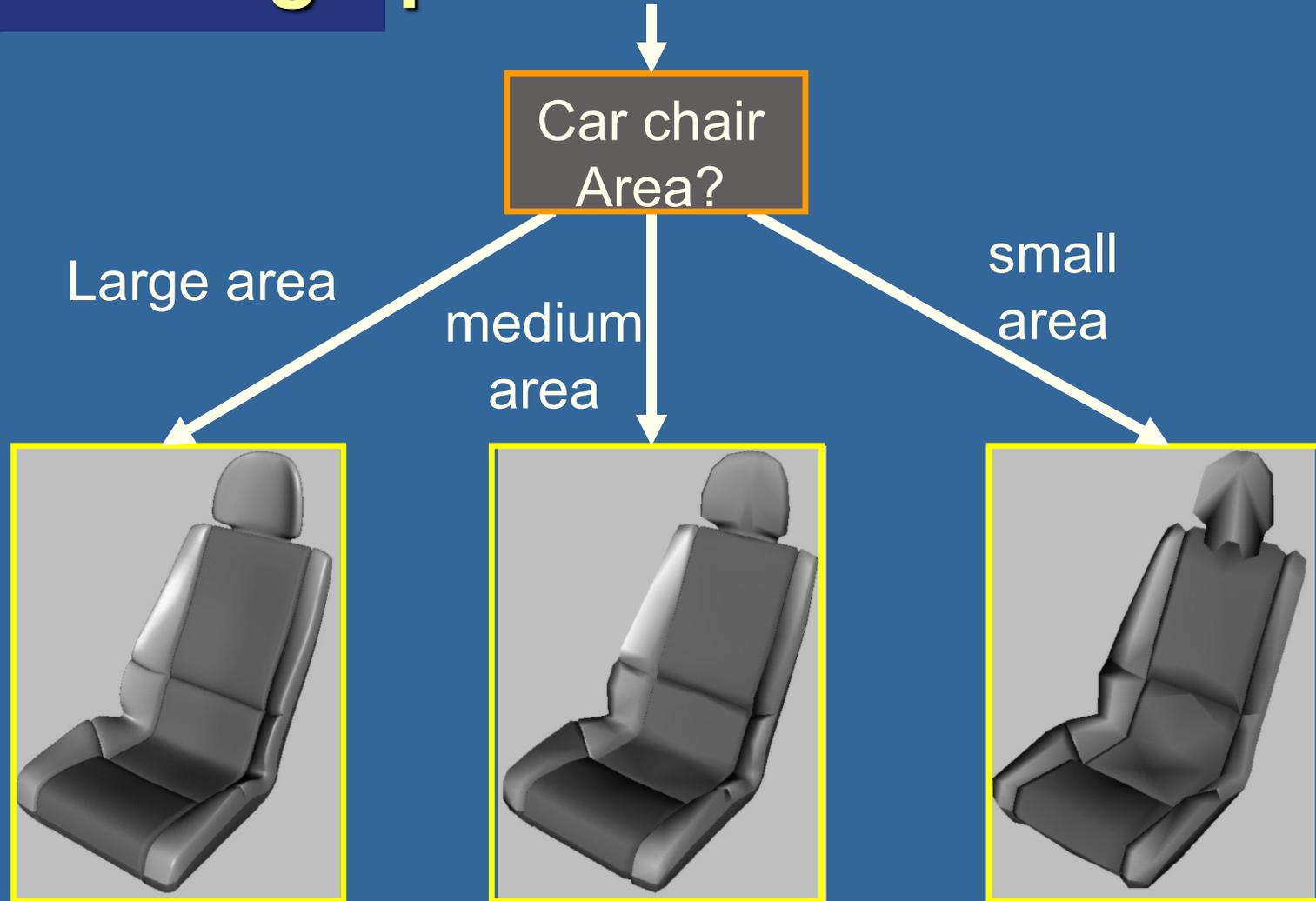
# LOD rendering

- Not much visual difference, but a lot faster



- Use area of projection of BV to select appropriate LOD

# Scene graph with LODs



# Far LOD rendering

- When the object is far away, replace with a quad of some color
- When the object is *really far away*, do not render it (called: detail culling)!
- Use projected area of BV to determine when to skip

# Misc

- Half Time wrapup slides will be available in “Schedule” on home page
- There is an Advanced Computer Graphics Seminar Course in sp 3+4, 7.5p
  - One seminar every week
    - Discussing advanced CG papers and techniques
  - Do a project of your choice.
  - Register to the course

**THE END**

# Exercise

- Create a function (by writing code on paper) that performs hierarchical view frustum culling
  - void hierarchicalVFC(node\* sceneGraphNode)

# What you need to know

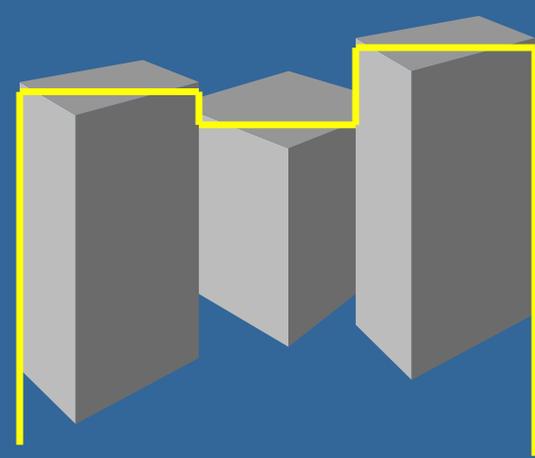
- Top-down construction of BVH, AABSP-tree,
- Construction + sorting with AABSP and Polygon-Aligned BSP
- Octree/quadtree (skip loose octrees)
- Scene Graphs (briefly)
- Culling – VFC, Portal, Detail, Backface, Occlusion
  - Backface culling – screenspace is robust, eyespace non-robust.
- What is LODs
- Describe how to build and use BVHs, AABSP-tree, Polygon aligned BSP-tree.
- Describe the octree/quadtree.

**THE END**

# BONUS MATERIAL

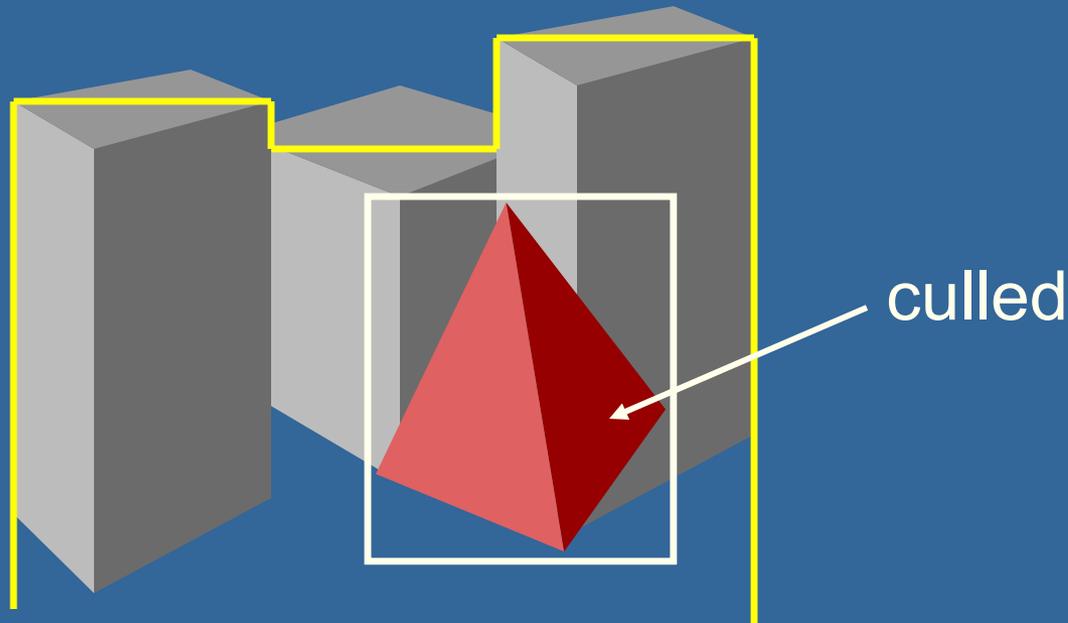
## Occlusion Horizon

- Target: urban scenery
  - dense occlusion
  - viewer is about 2 meters above ground
- Algorithm:
  - Process scene in front-to-back using a quad tree
  - Maintain a piecewise constant horizon
  - Cull objects against horizon
  - Add visible objects' occluding power to the horizon



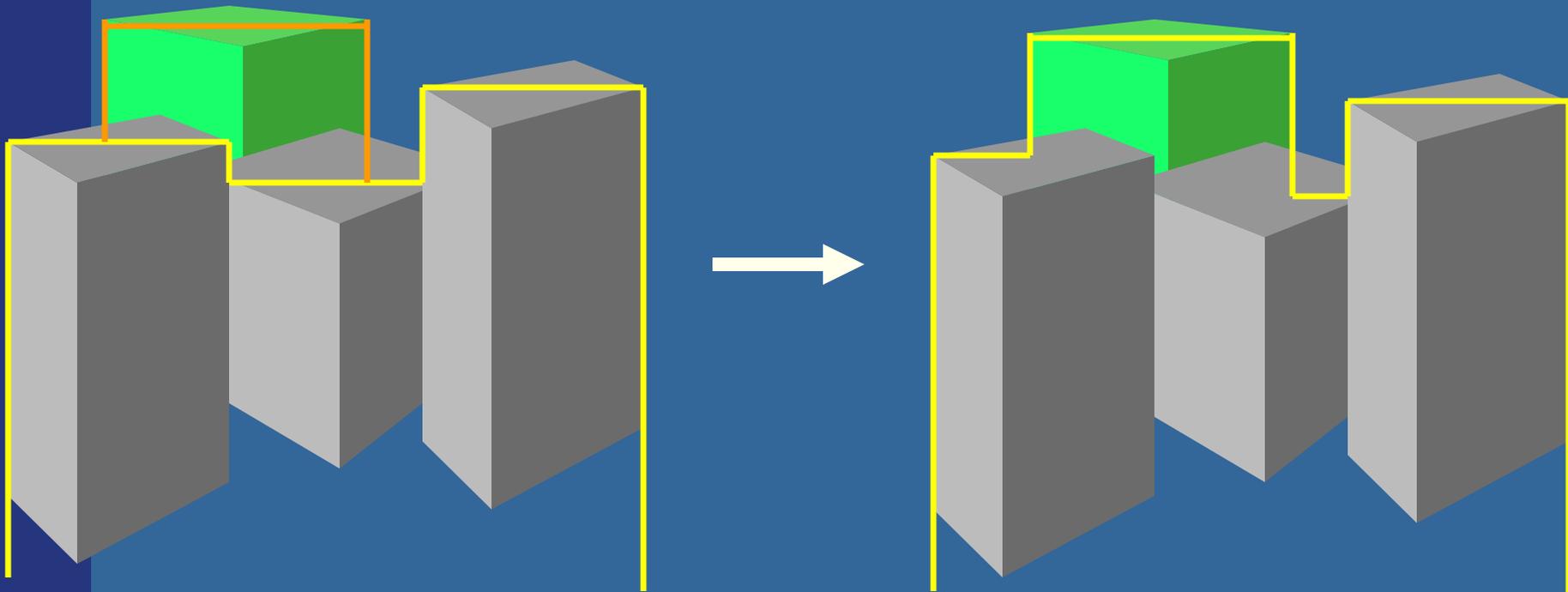
# Occlusion testing with occlusion horizons

- To process tetrahedron (which is behind grey objects):
  - find axis-aligned box of projection
  - compare against occlusion horizon

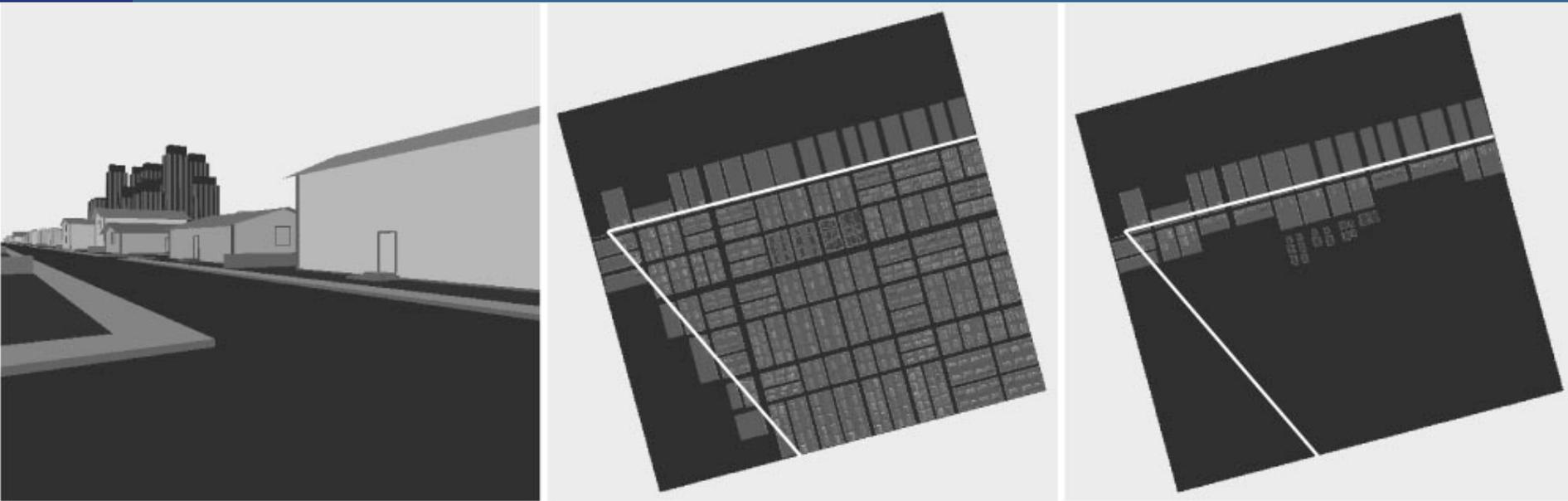


# Update horizon

- When an object is considered visible:
- Add its “occluding power” to the occlusion representation



# Example:



- Read about the details in paper on website (compulsory material!)