Spatial Data Structures and Speed-Up Techniques

Ulf Assarsson

Department of Computer Science and Engineering

Chalmers University of Technology

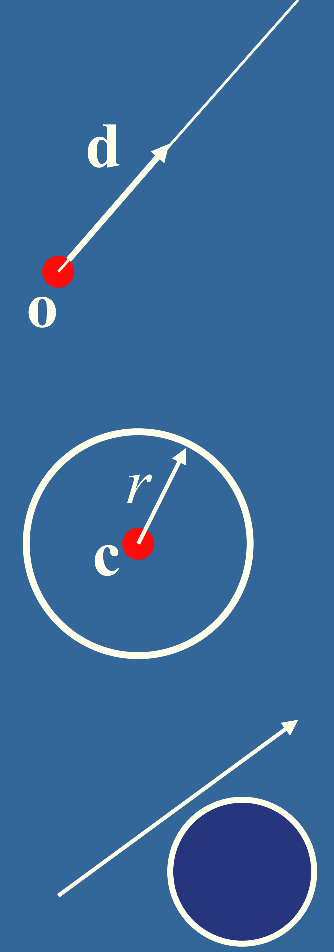
Have you done your homework ;-) ?

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

…e.g., the 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:



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

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

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

```
Bool raySphereIntersect(vec3f o, d, c, float r, Vec3f &hitPt) {
    float a = d.dot(d);
    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*a) - 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
  - Including 3 old exams
- There is an Advanced Computer Graphics Seminar Course in sp 4, 7.5p
  - One seminar every week
    - Advanced CG techniques
  - Do a project of your choice.
  - Register to the course

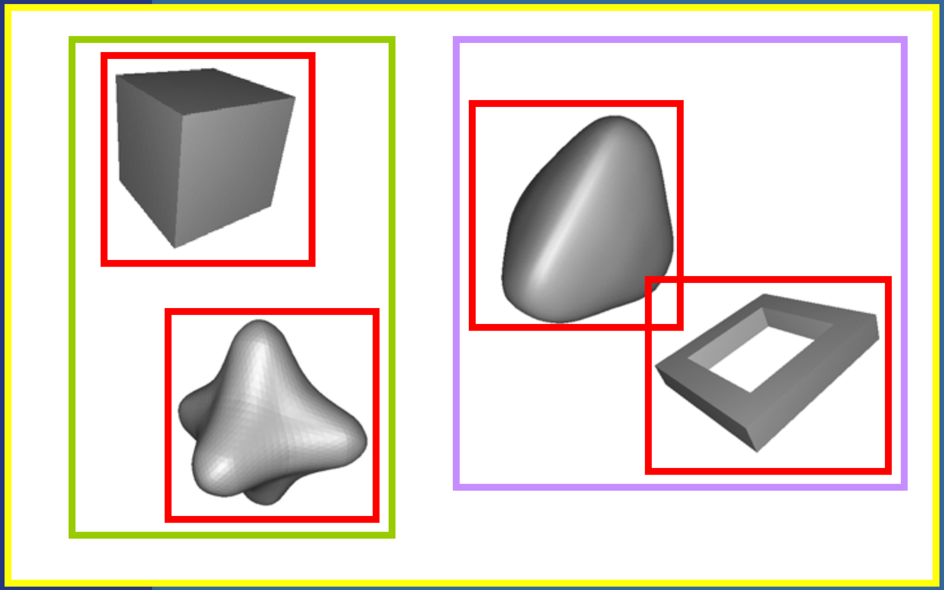
Spatial data structures

- 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

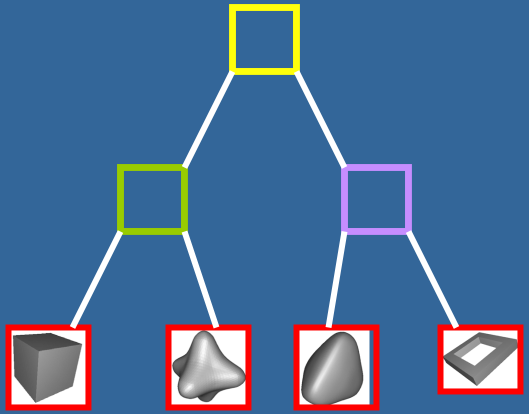
How?

- Organizes geometry in some hierarchy

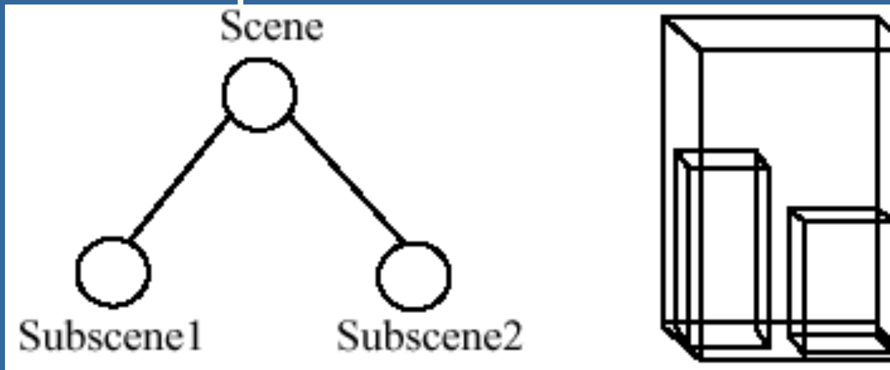
In 2D space



Data structure

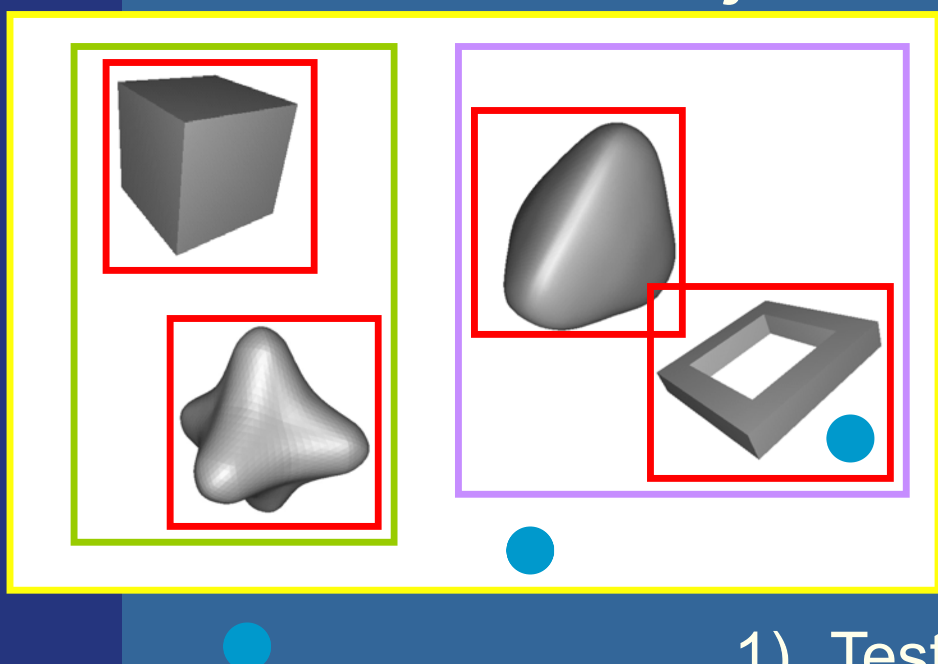


In 3D space:

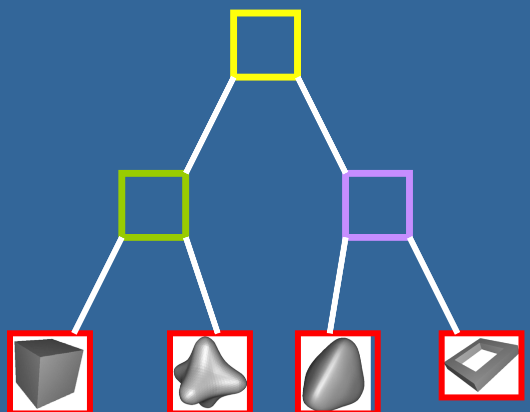


What’s the point?
An example

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



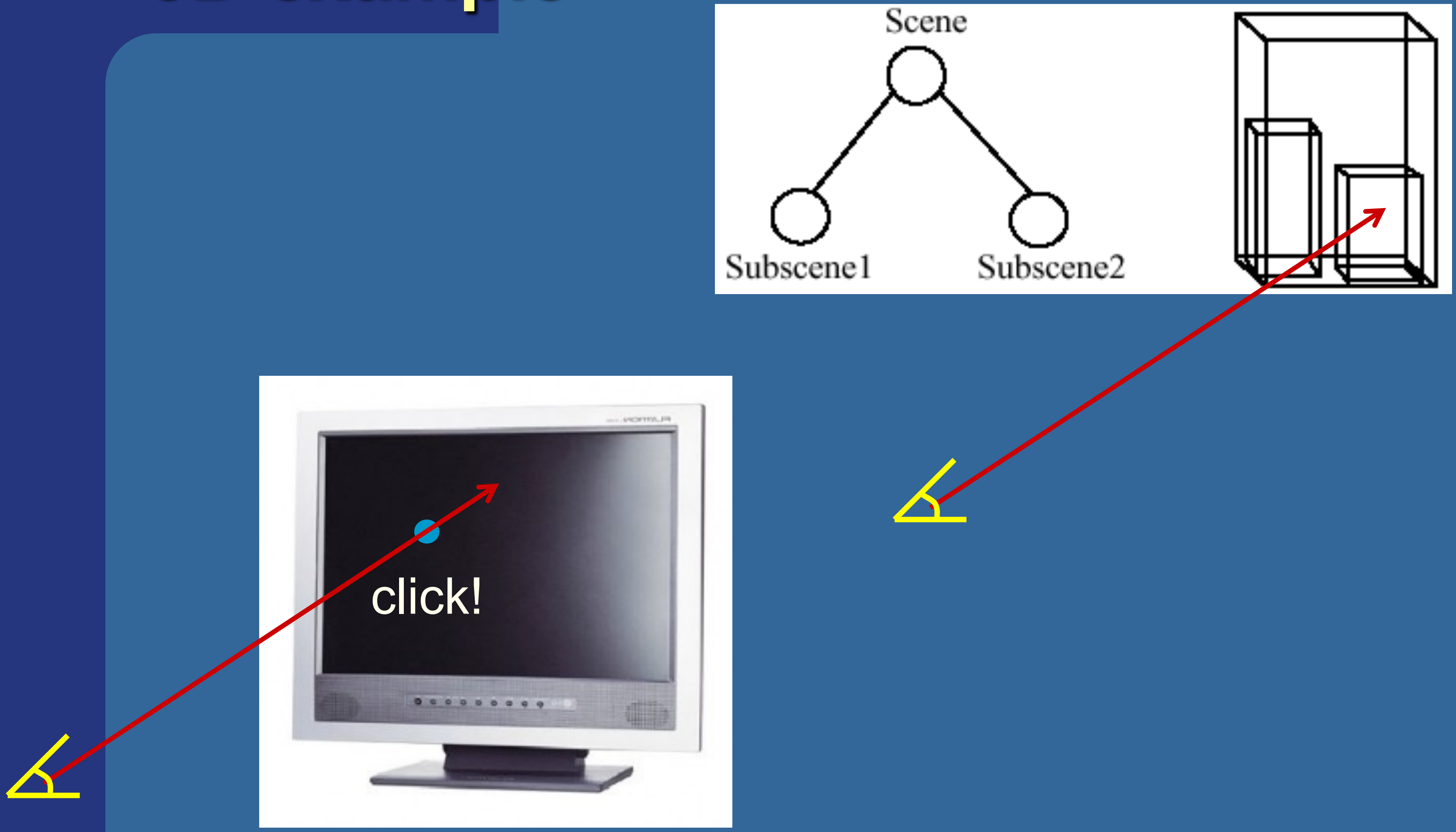
click!



1) Test the root first
2) Descend recursively as needed
3) Terminate traversal when possible

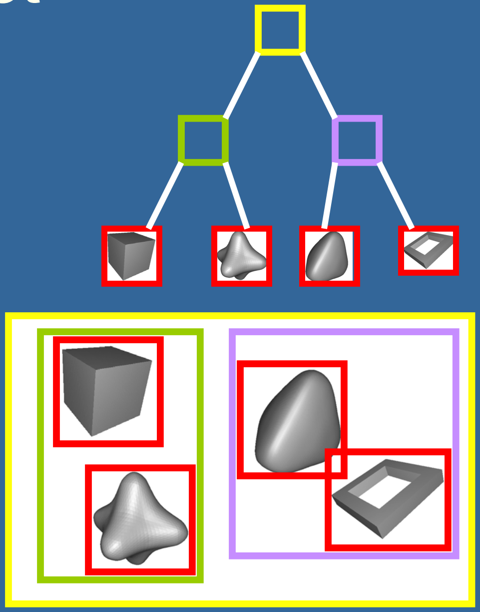
In general: get O(log n) instead of O(n)

3D example



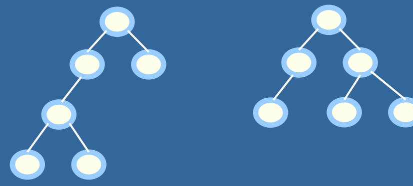
Bounding Volume Hierarchy (BVH)

- Most common bounding volumes (BVs):
  - Sphere
  - Boxes (AABB and OBB)
- The BV does not contibute 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



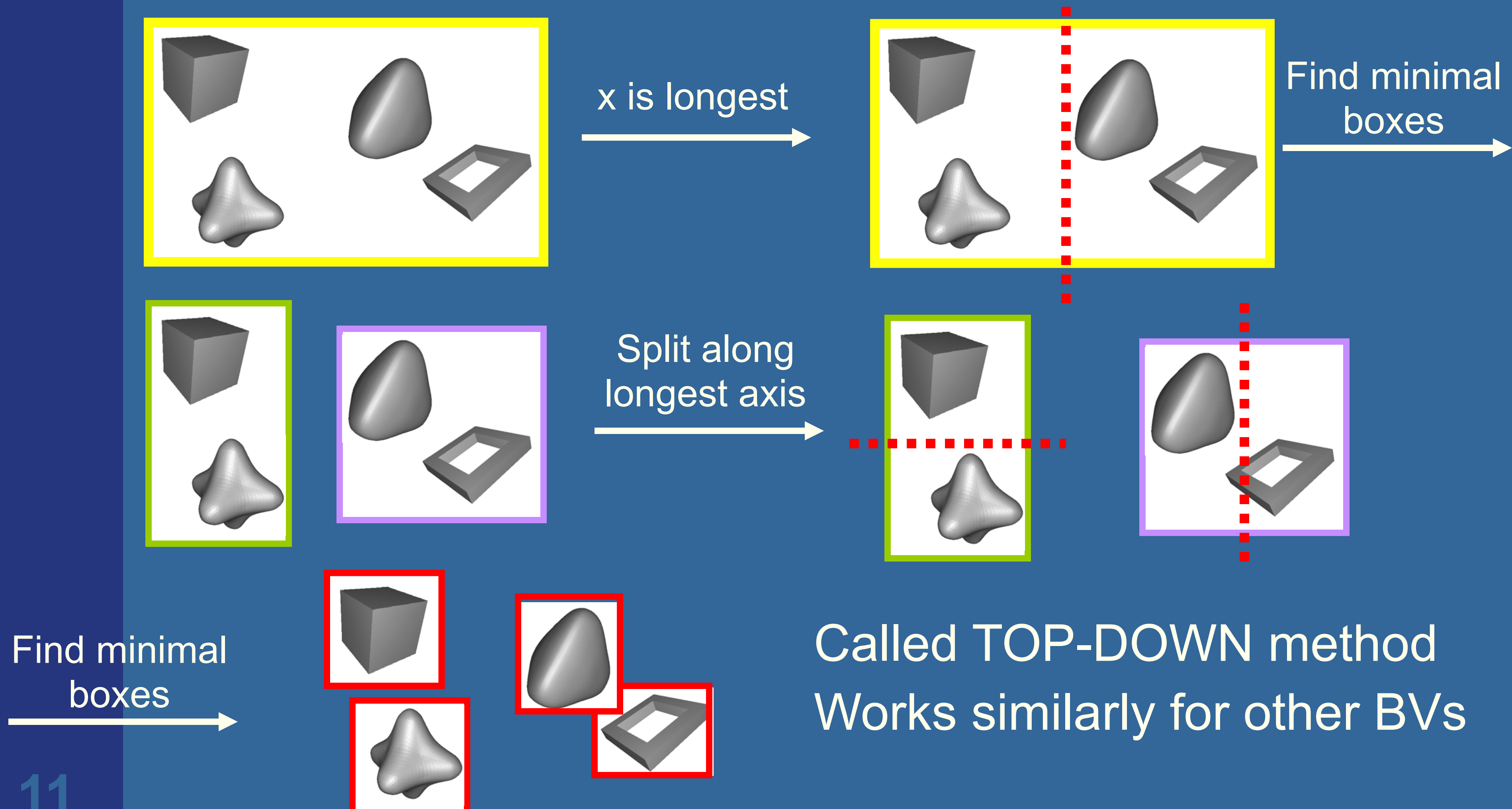
Some facts about trees

- *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 <u>balanced</u> tree with n nodes: $\text{floor}(\log_k(n)) + 1$
- 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



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 some max recursion level l has been reached

- Similar critera for BSP trees and octrees

State-of-the-Art BVH builders:

- *A Survey on Bounding Volume Hierarchies for Ray Tracing*. Meister et al. 2021.
- *Ploc++ parallel locally-ordered clustering for bounding volume hierarchy construction revisited*. Benthin et al. 2022

Example

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

Kd-tree = Axis Aligned BSP tree

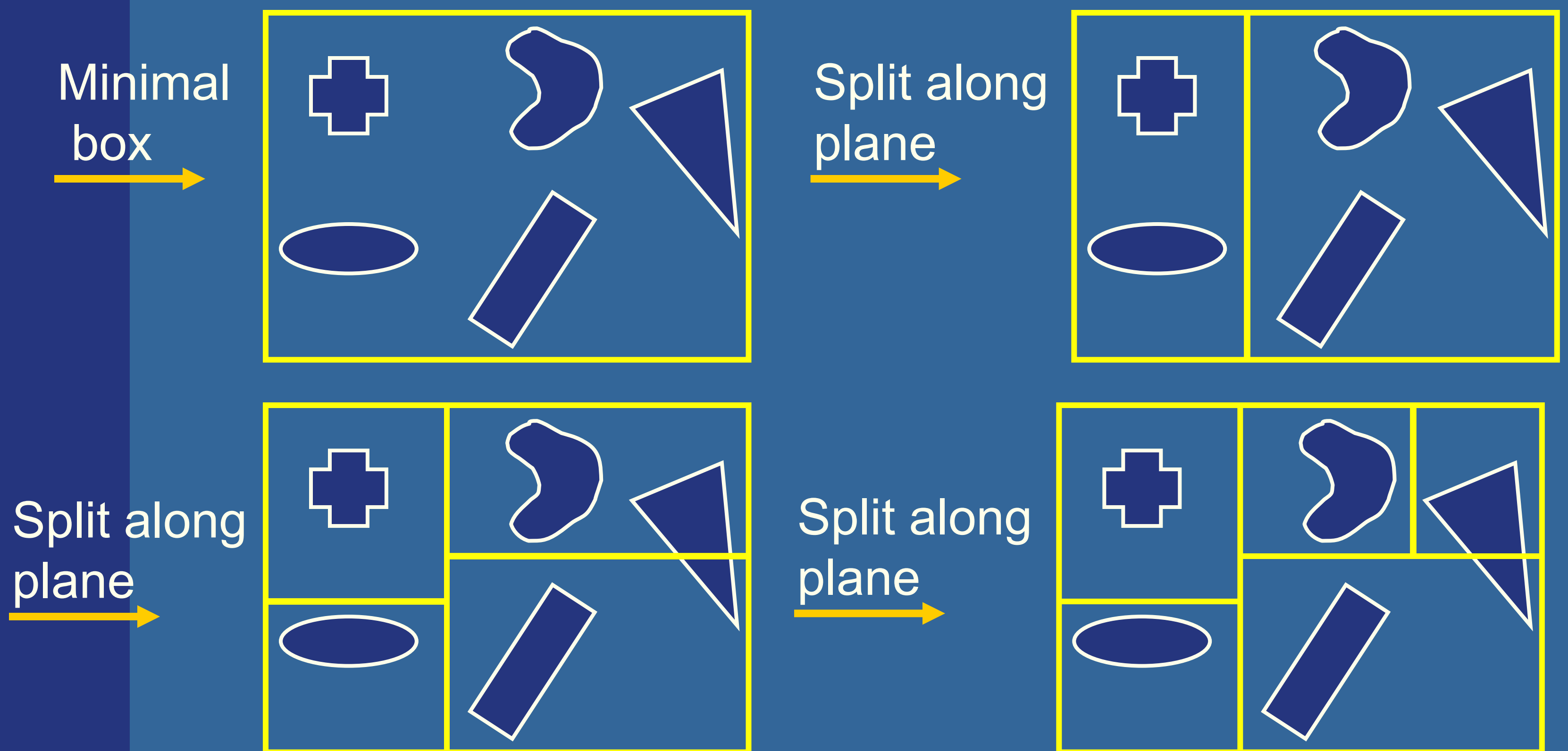
Binary Space Partitioning (BSP) Trees

- 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

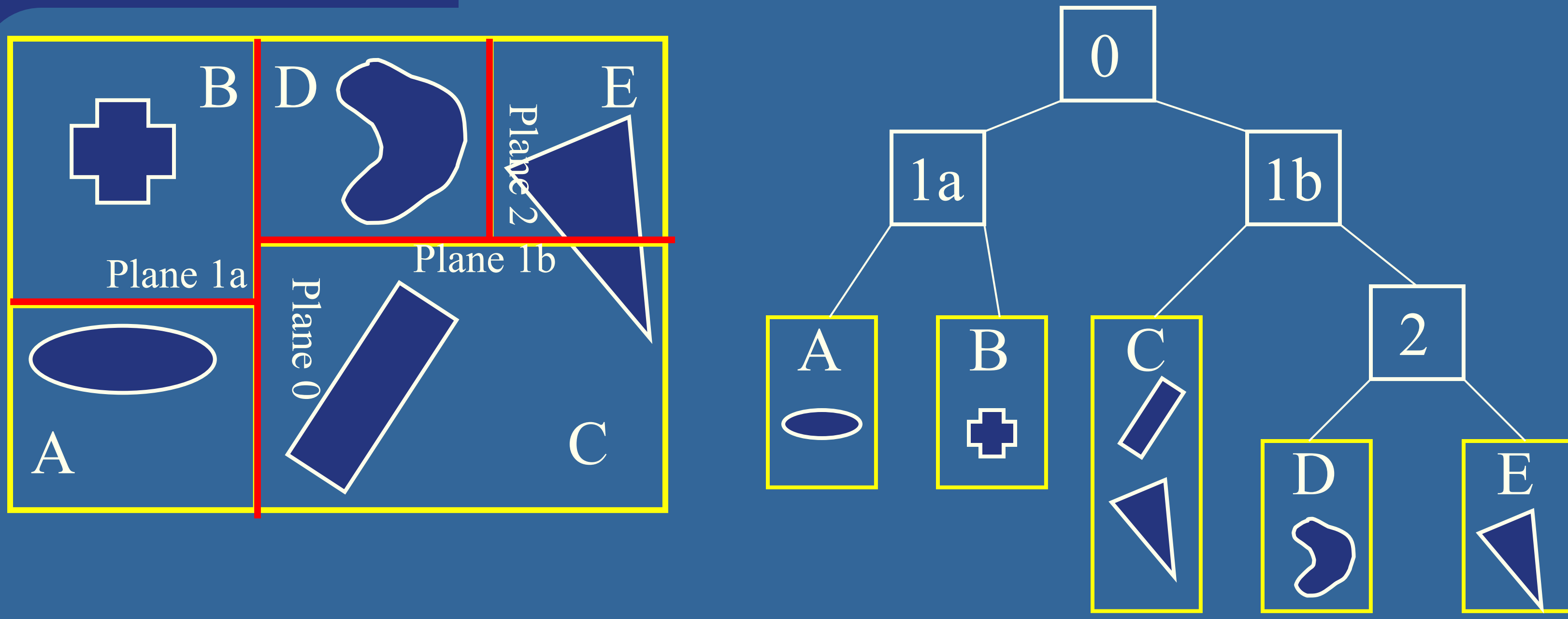
- Split space with a plane
- Divide geometry into the space it belongs
- Done recursively

Axis-Aligned BSP tree (1)

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



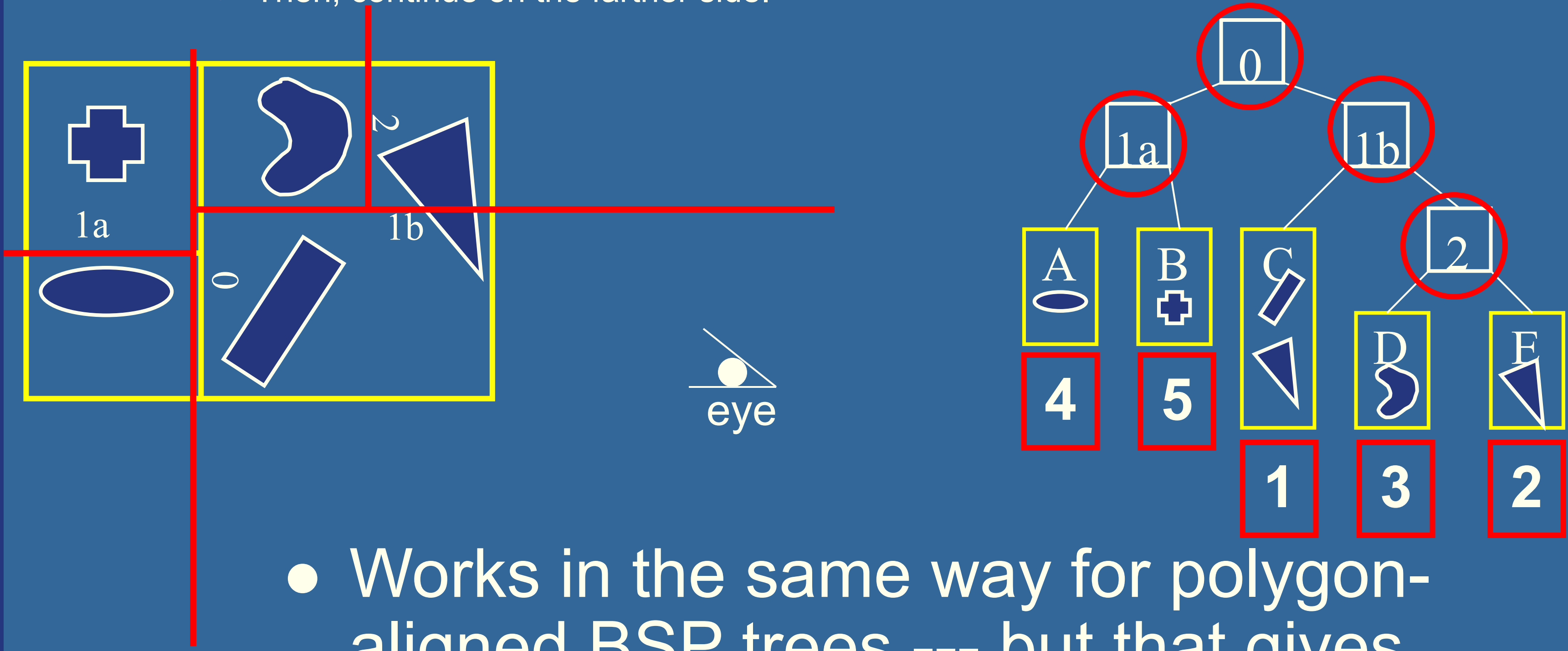
Axis-Aligned BSP tree (2)



- 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

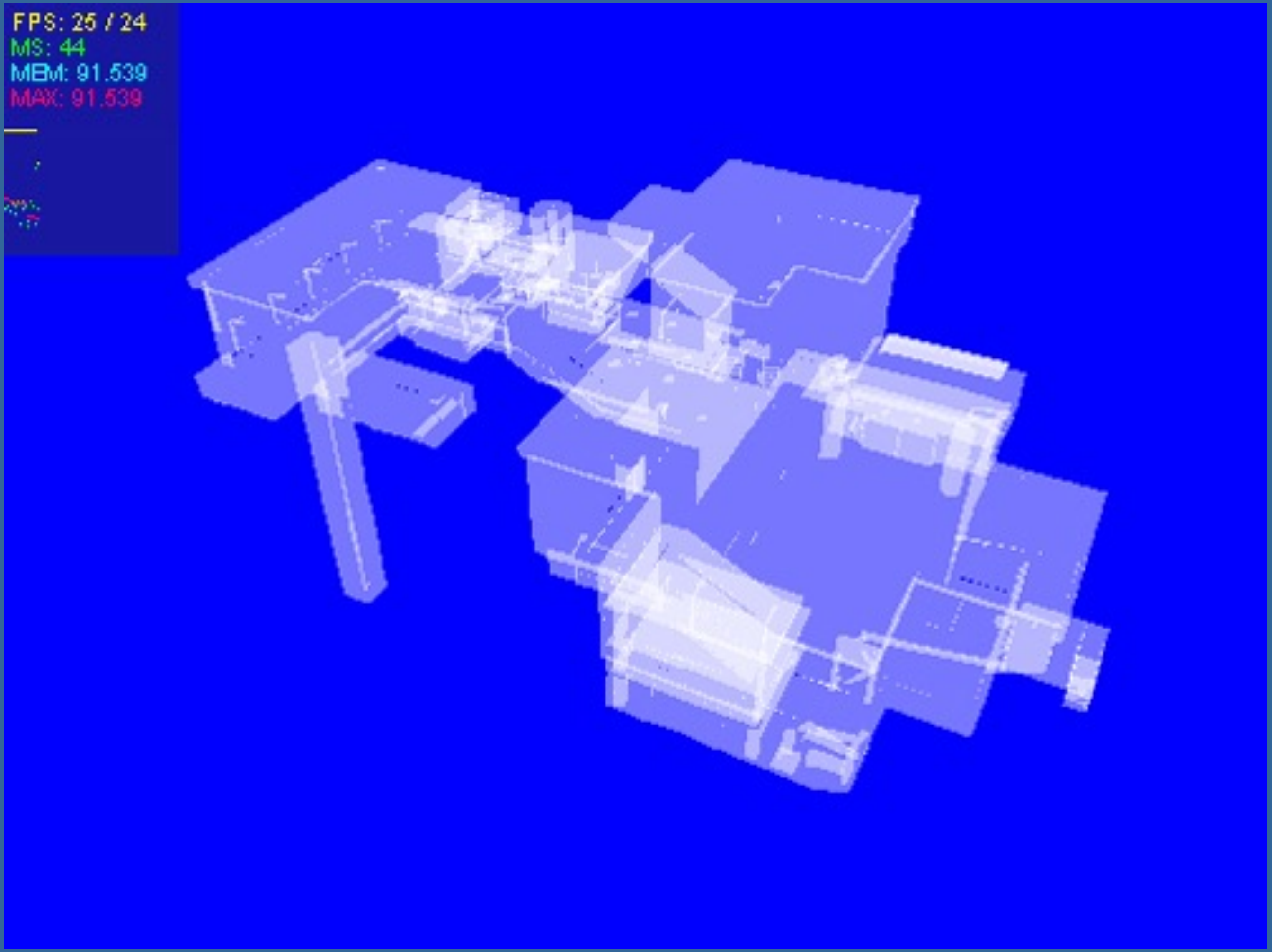
- Test the planes, recursively from root, against the point of view. For each traversed node:
  - If node is leaf, draw the node's geometry
  - else
    - Continue traversal on the "hither" side with respect to the eye (to sort front to back)
    - Then, continue on the farther side.



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

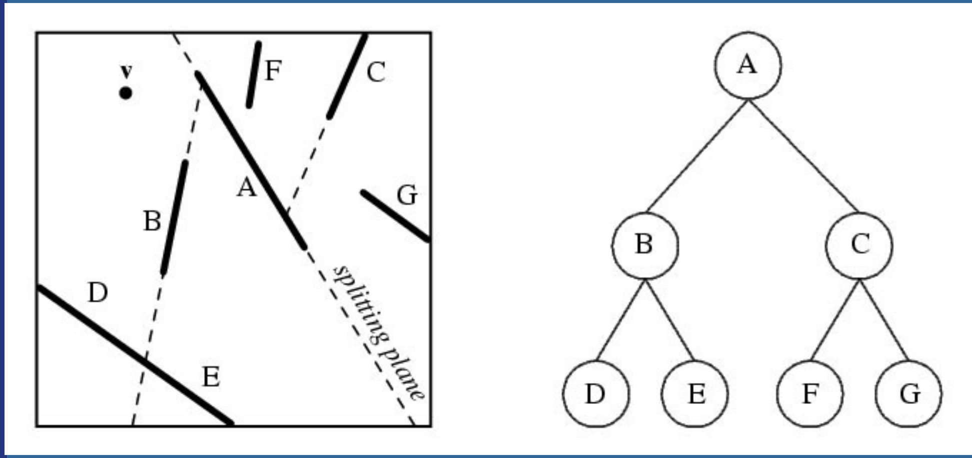
Polygon Aligned BSP tree – Quake 2

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

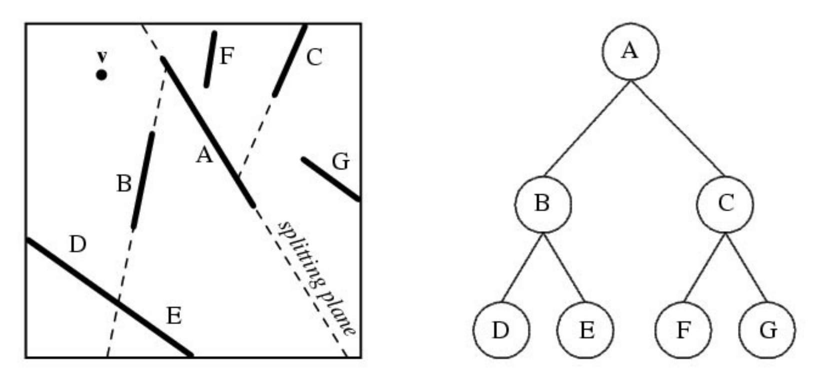
class BSPtree:
    Polygon P;
    BSPtree behindP;
    BSPtree frontOfP;

```
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 {
```
    recurse on farther side of P;
    Draw P;
    Recurse on hither side of P;
}
```

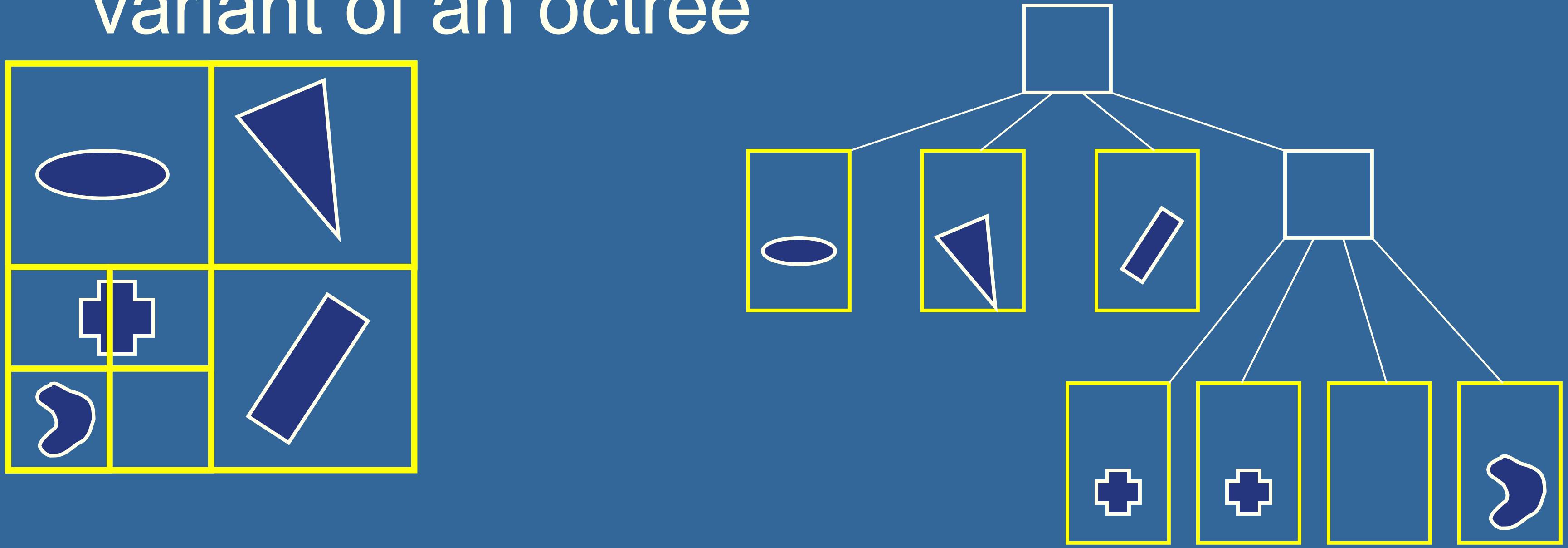
Drawing Back-to-Front:

```
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);
}
```



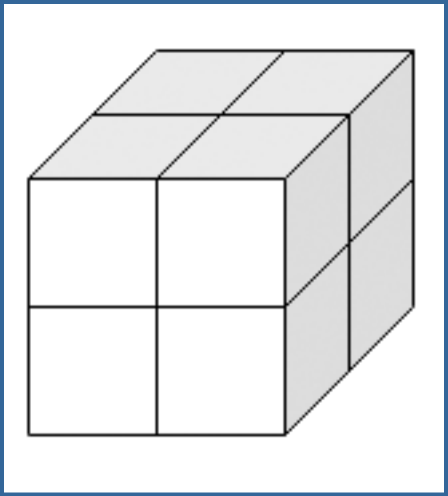
Octrees (1)

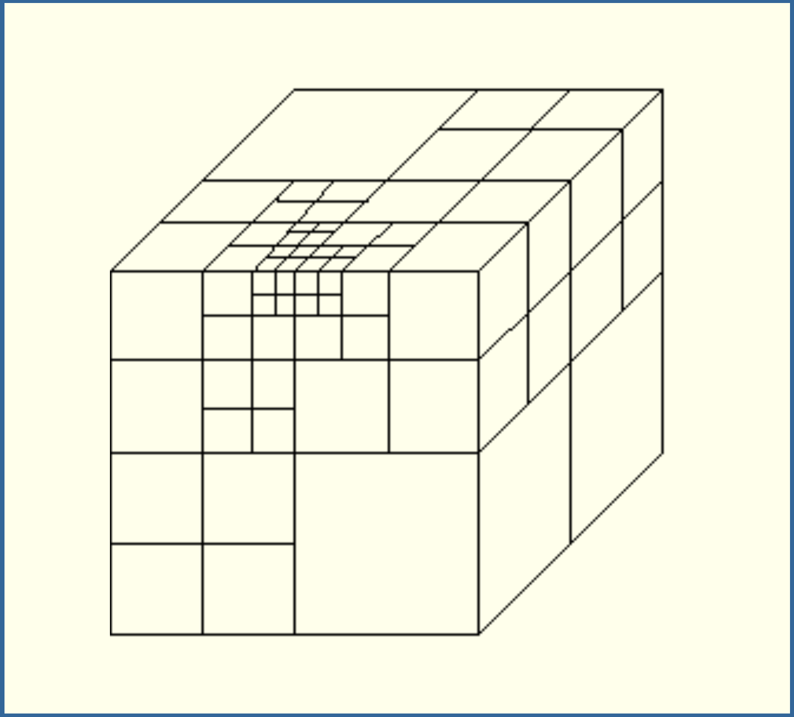
- A bit similar to axis-aligned BSP trees
- Will explain the quadtree, which is the 2D variant of an octree



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

Example of Octree





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

Example of octree

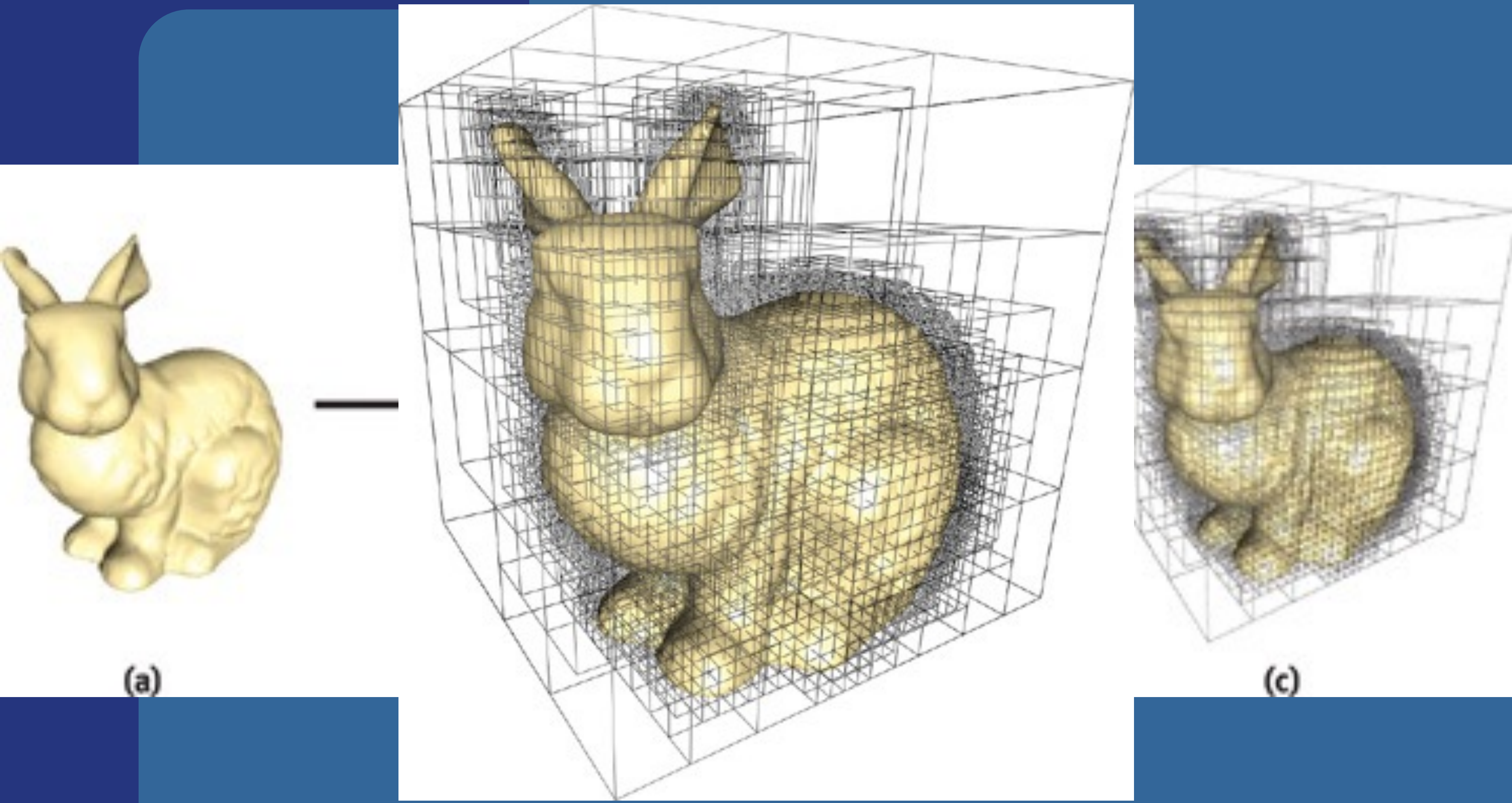


Image from Lefebvre et al.

Example of octree

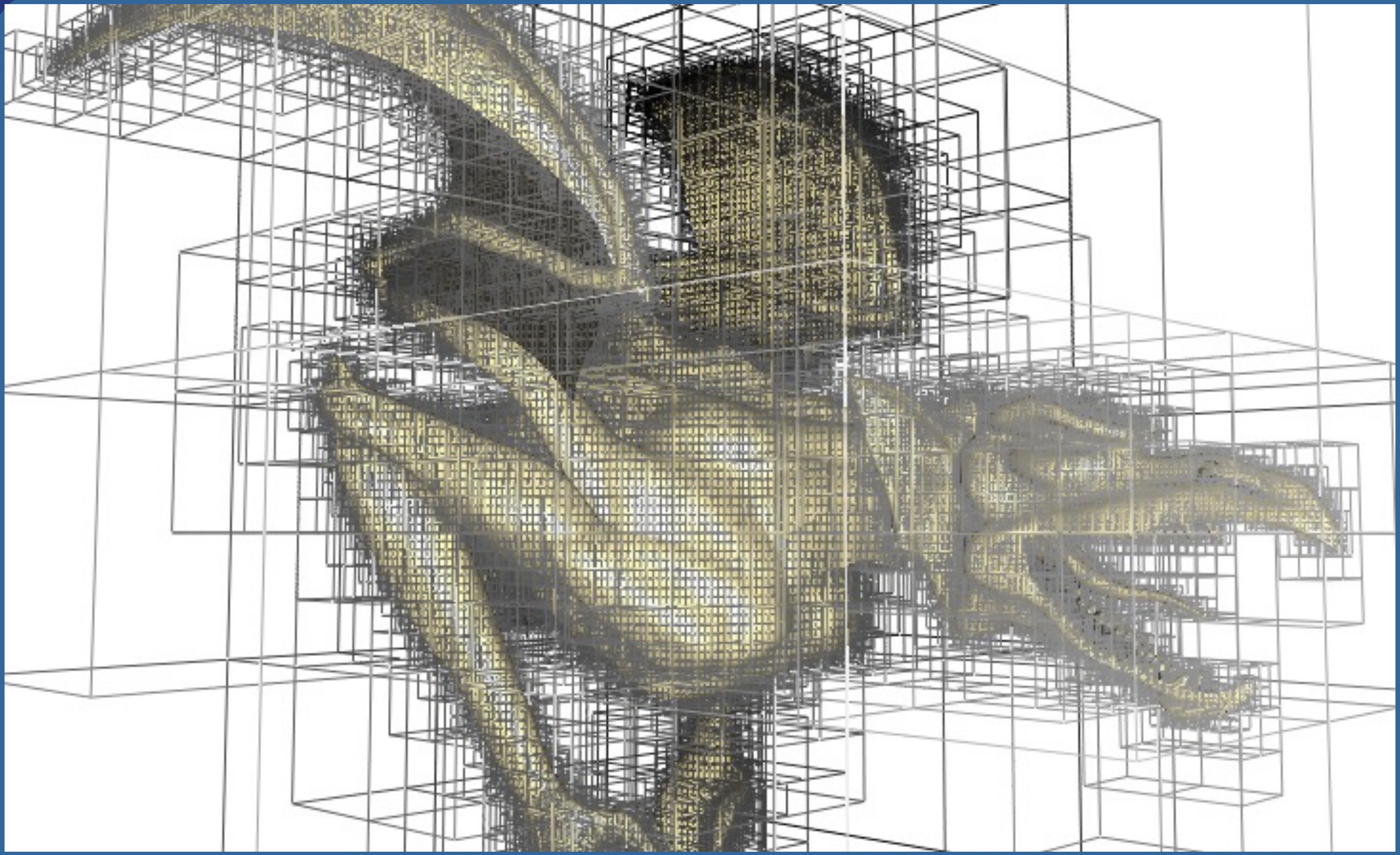


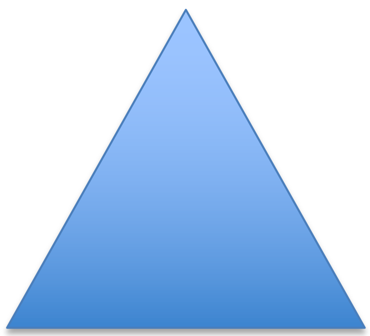
Image from Lefebvre et al.

Octrees (2)

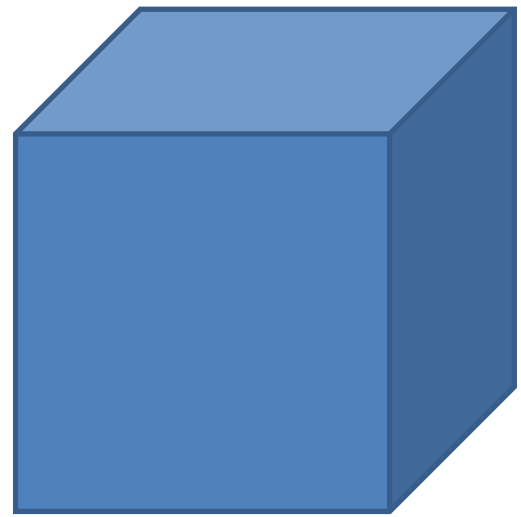
- Expensive to rebuild (BSPs are too)

- Octrees can be used to
  - Speed up ray tracing
  - Faster picking
  - Culling techniques
  - Are not used that often these days, except for Sparse Voxel Octrees (SVO:s)

Voxels



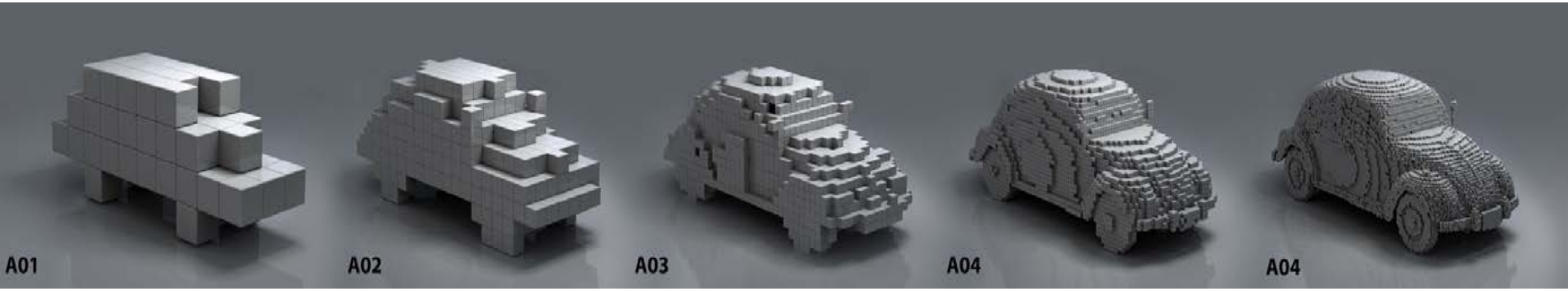
Triangle
36 bytes



Voxel
Volume – element
1 bit

Voxels

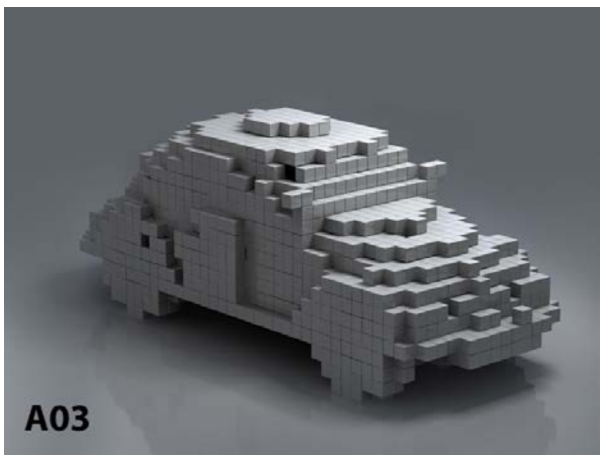
- Desirable to be able to use very high resolutions

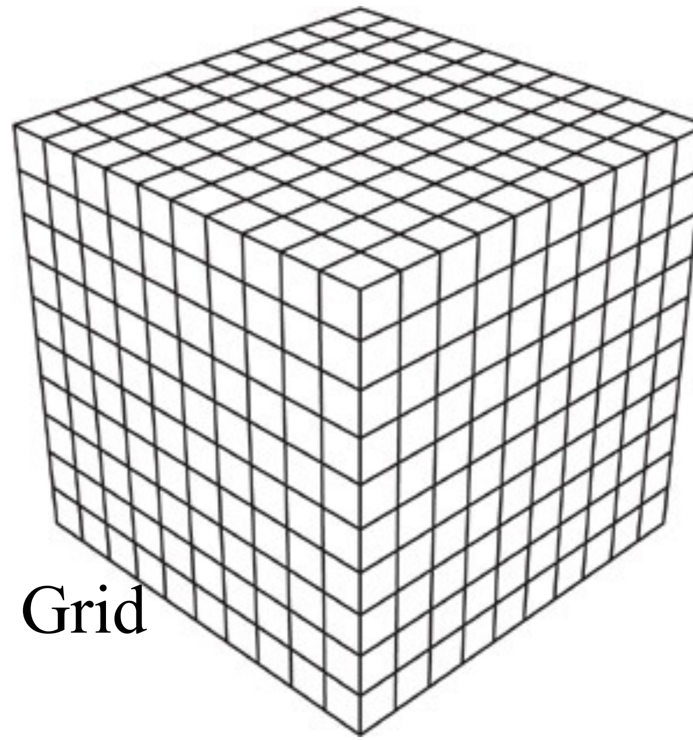


Voxels

One possible data structure:

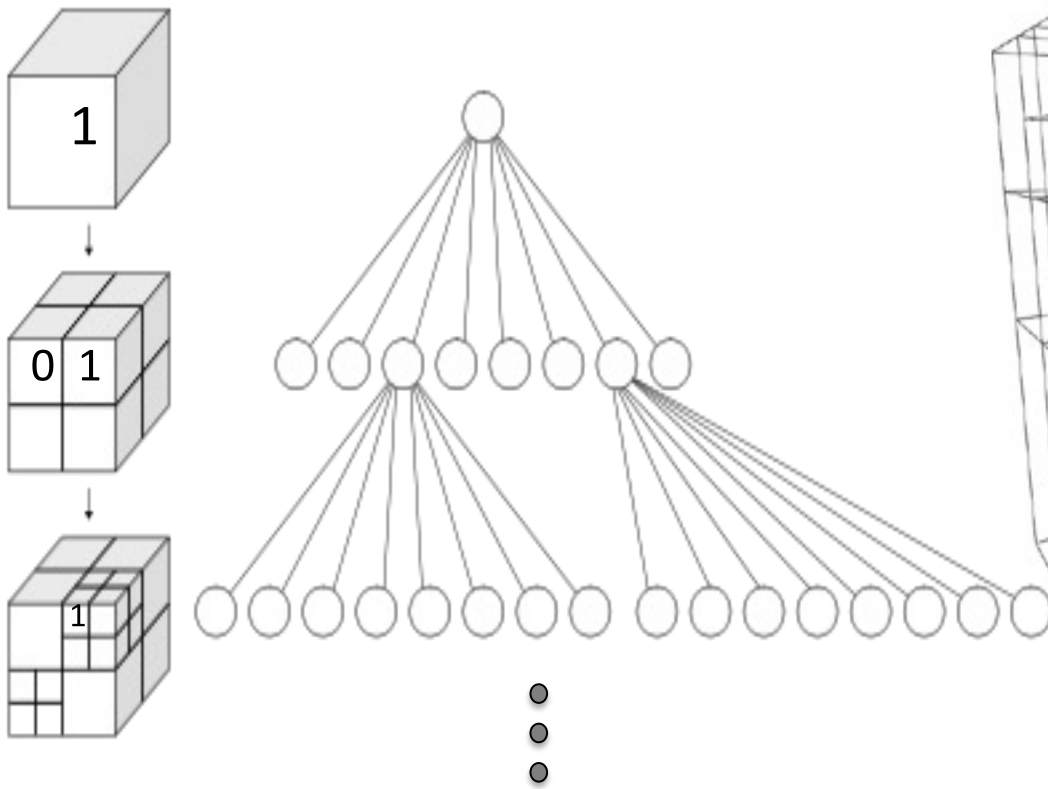
- Voxel Grids – 3D array of 0:s and 1:s

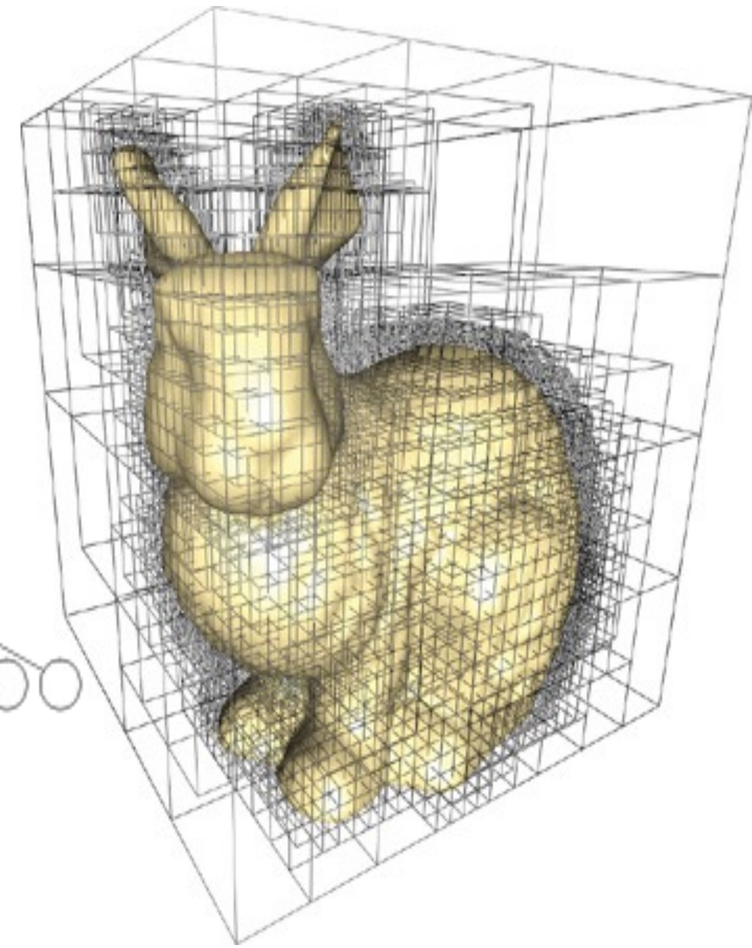




Sparse Voxel Octree

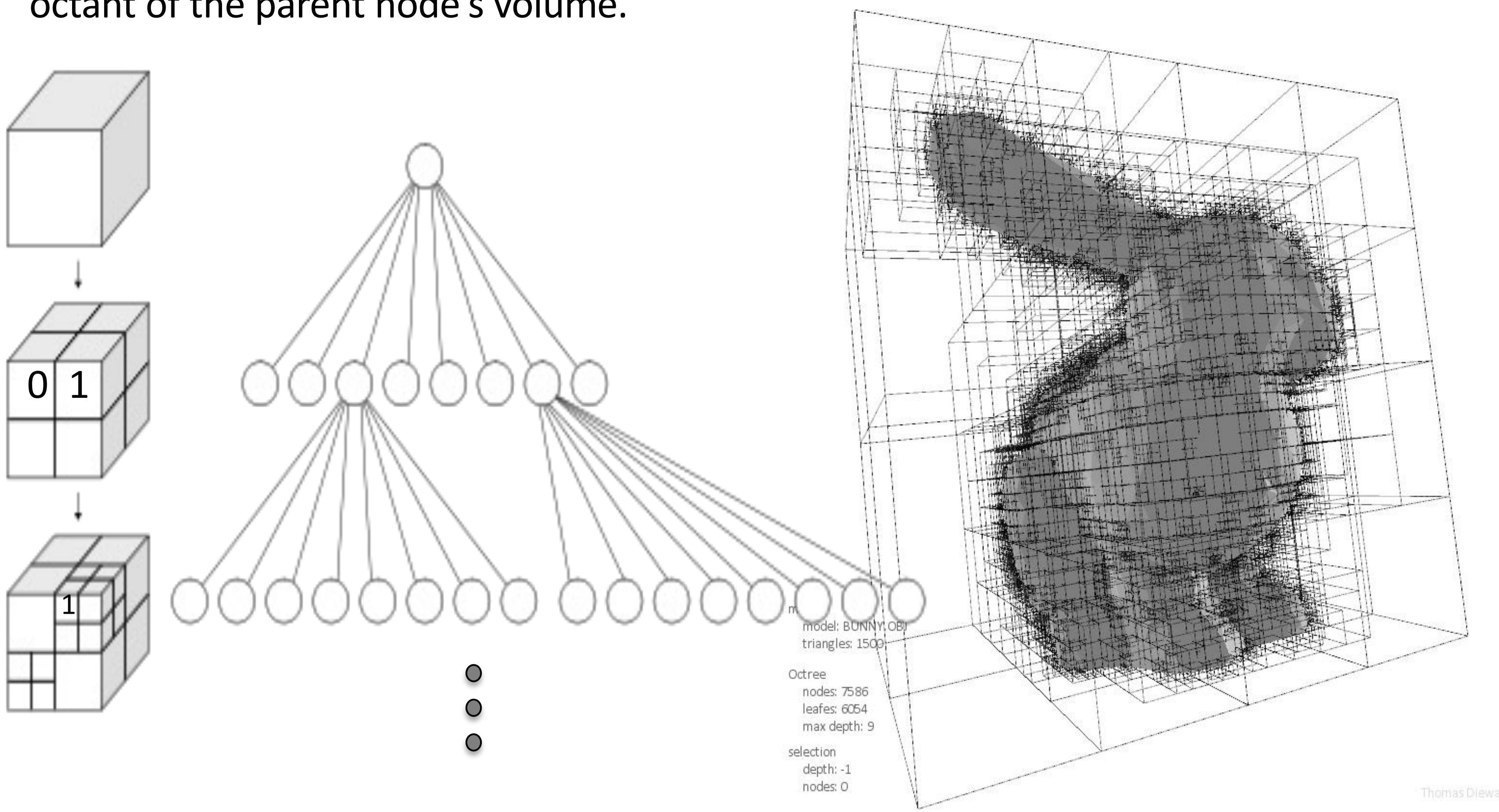
Each node has eight children, representing an octant of the parent node's volume.



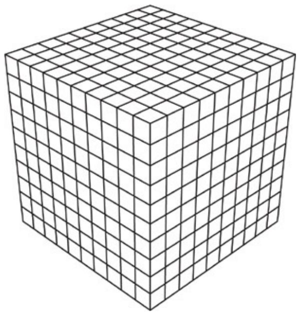


Sparse Voxel Octree

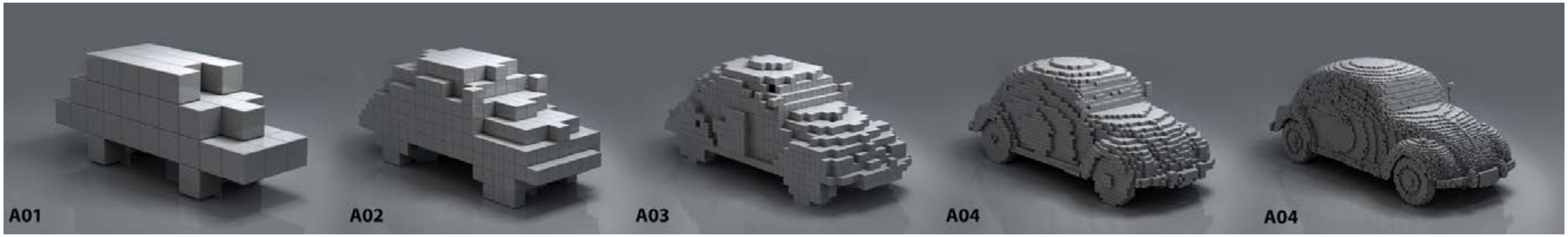
Each node has eight children, representing an octant of the parent node's volume.



Sparse Voxel DAGs

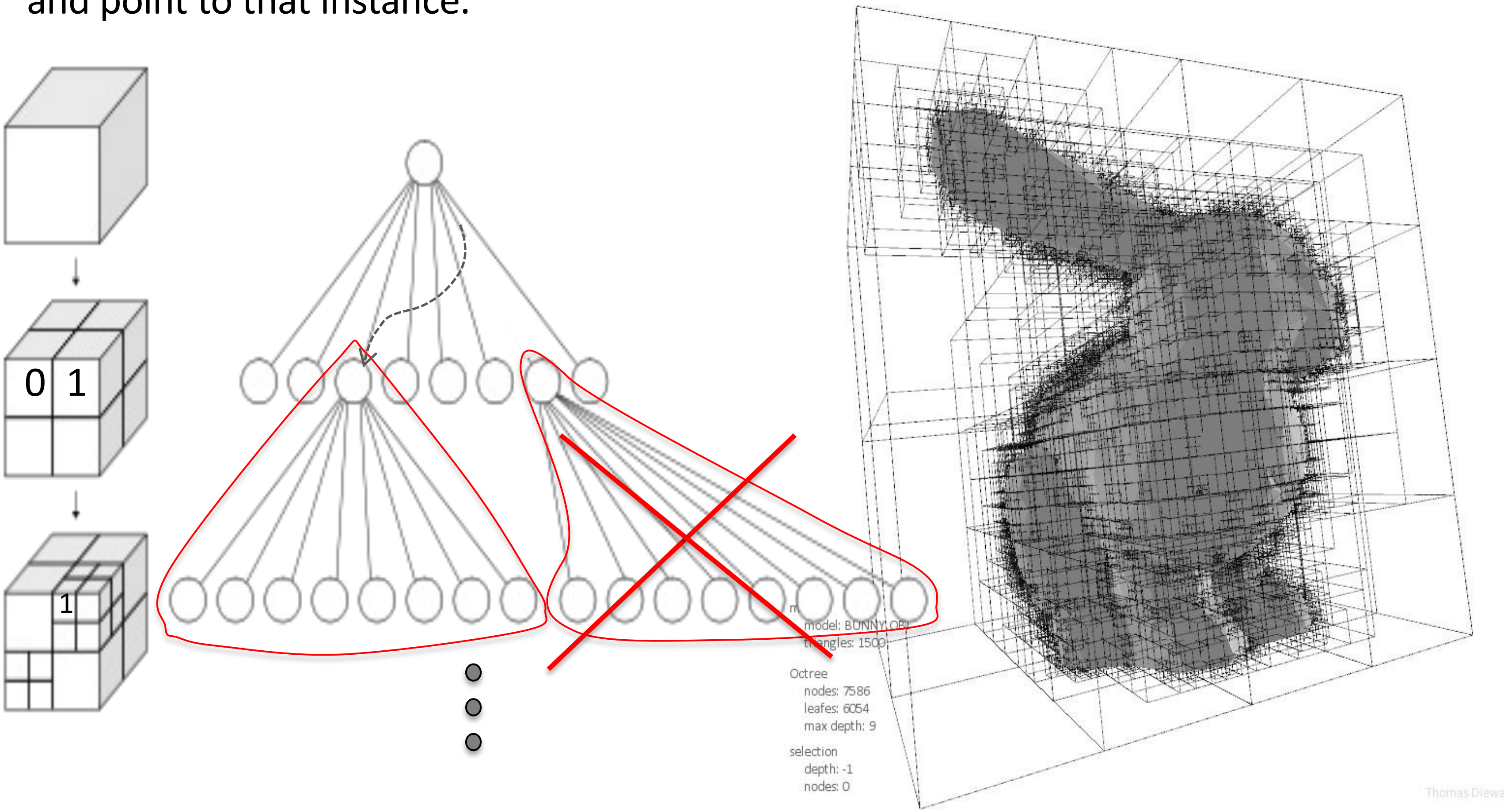


- Voxel = 1 bit.
- SVDags can currently handle scene of res = 128.000^3
  - Naively with bit grid: 262 TB
  - SVDAGs => < 1GB



Sparse Voxel DAGs

For identical subgraphs, only store one instance, and point to that instance.

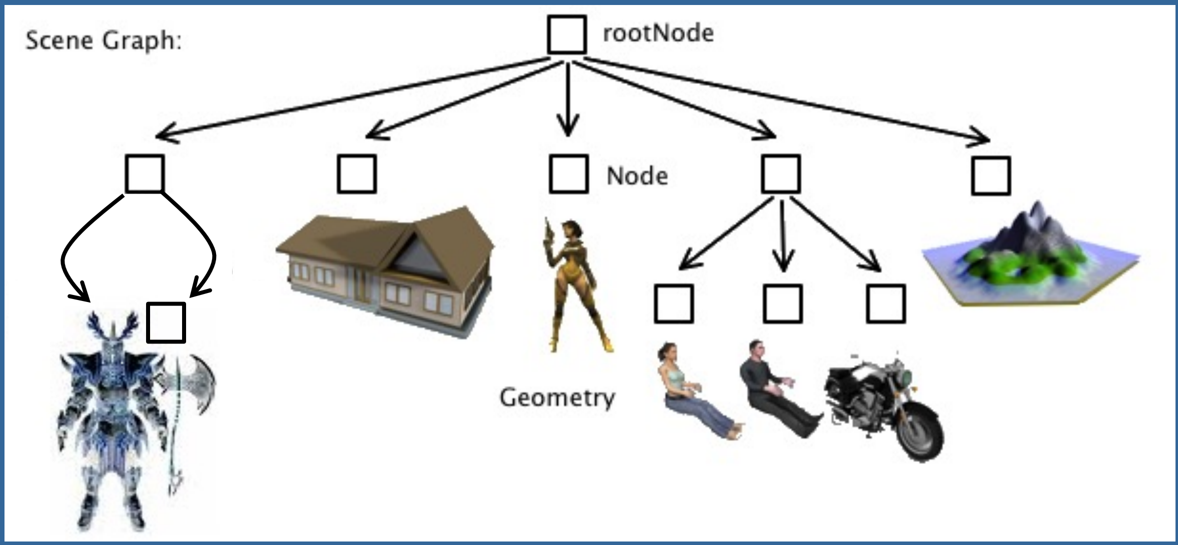


Sparse Voxel DAGs

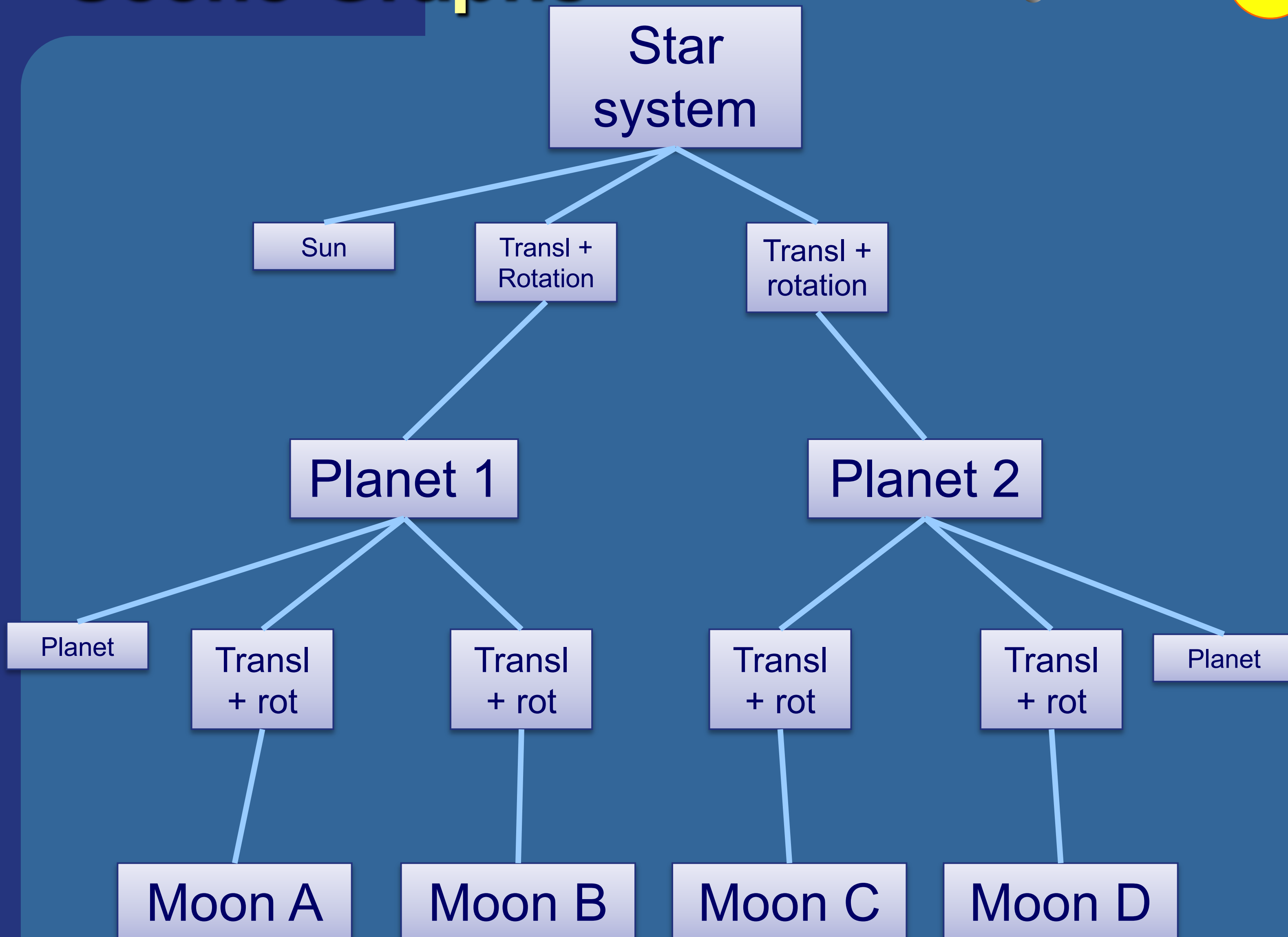
https://youtu.be/6zpbV6hZPWU

Scene graphs
– or node hierarchies

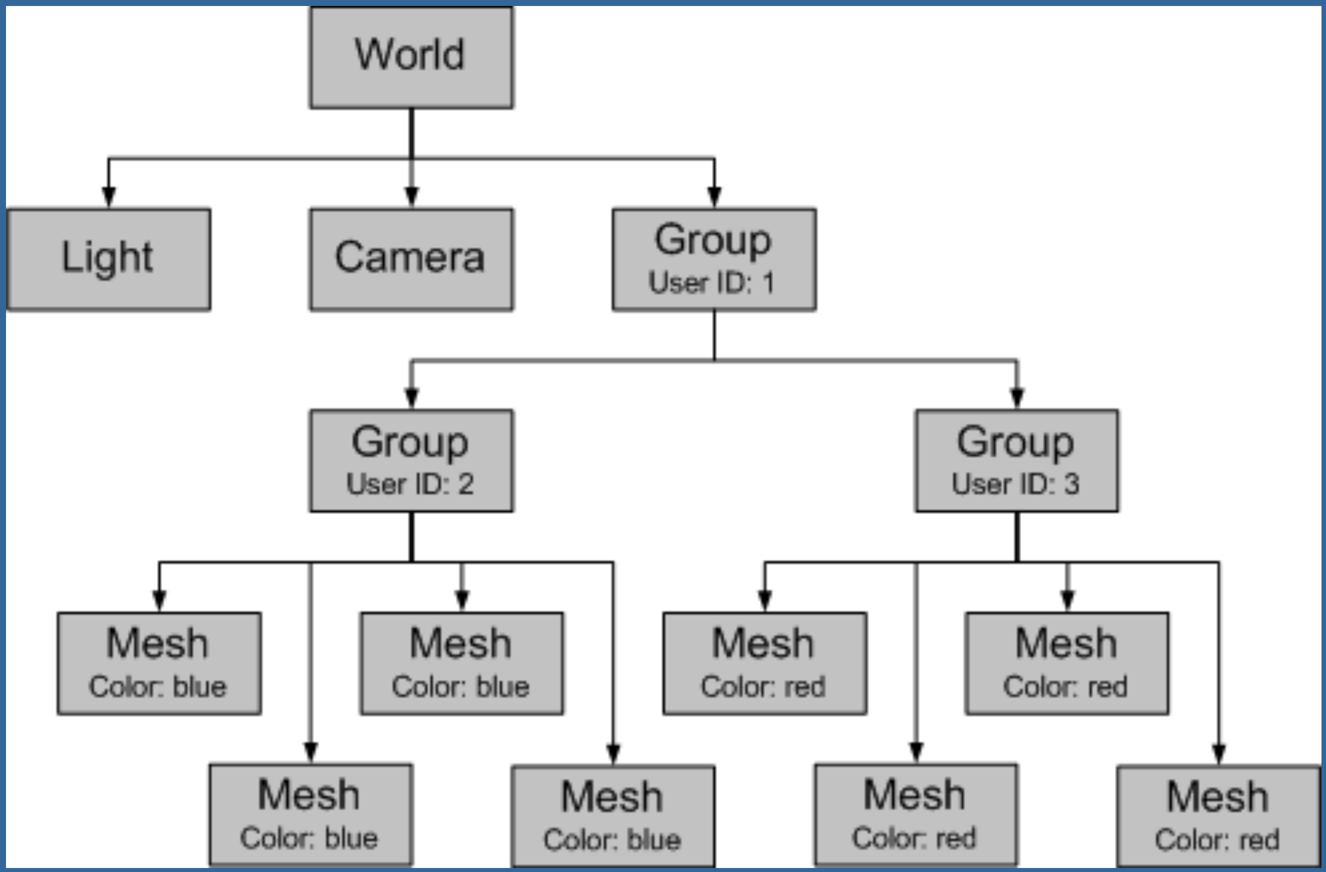
- A scene graph is a node hierarchy, which often reflects a logical hierarchical scene description
  - often in combination with a BVH such that each node has a BV.
- Common hierarchical features include:
  - Lights
  - Materials
  - Transforms
  - Transparency
  - Selection



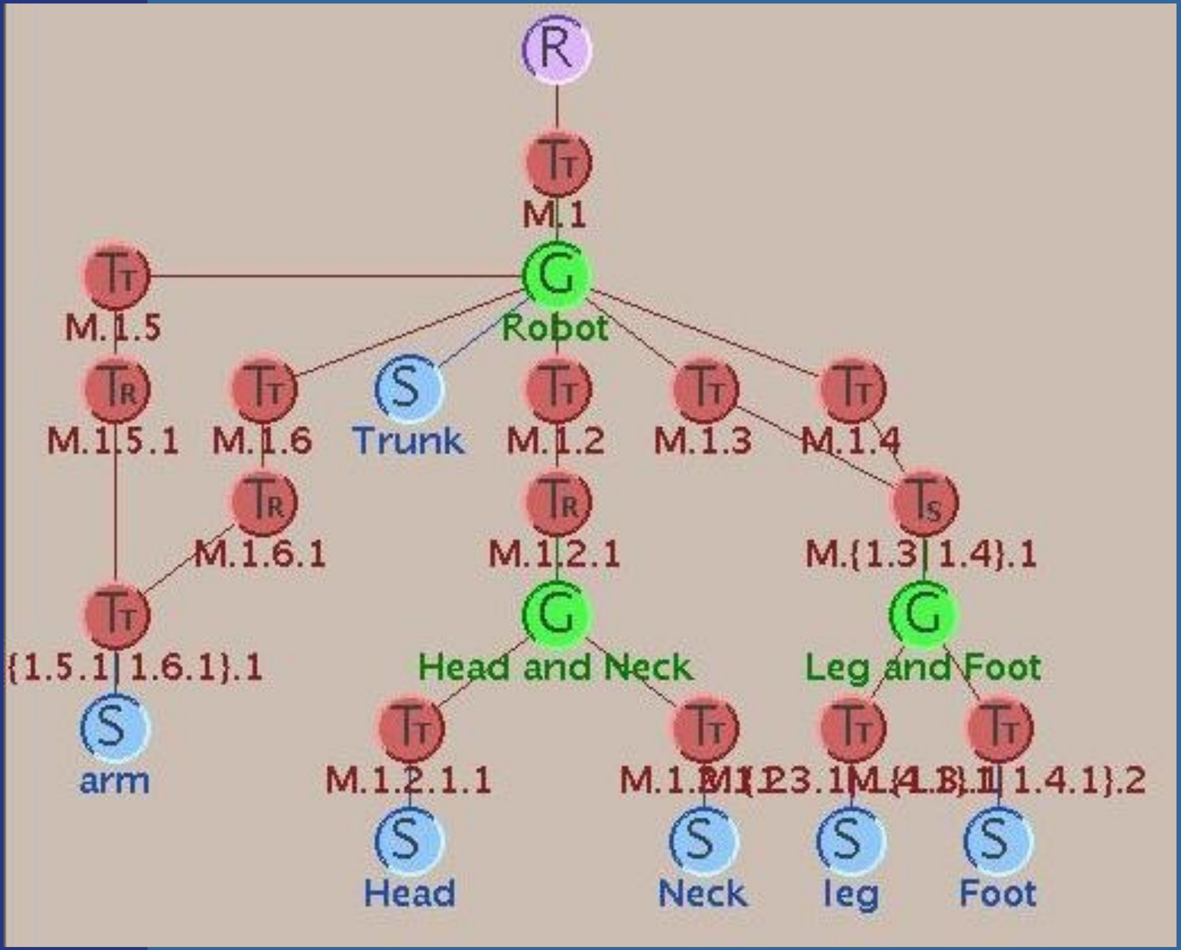
Scene Graphs

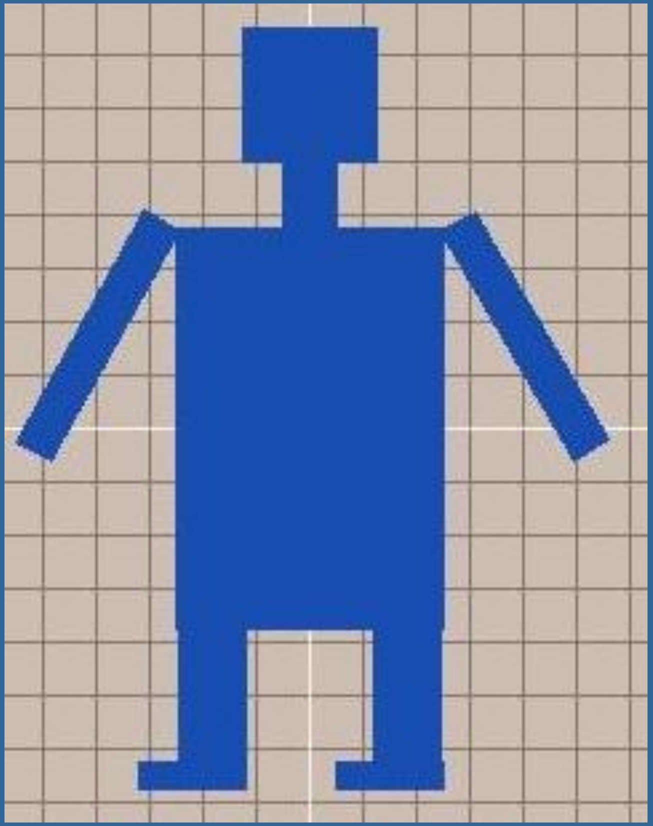


Scene Graphs



Scene Graphs





Speed-Up Techniques

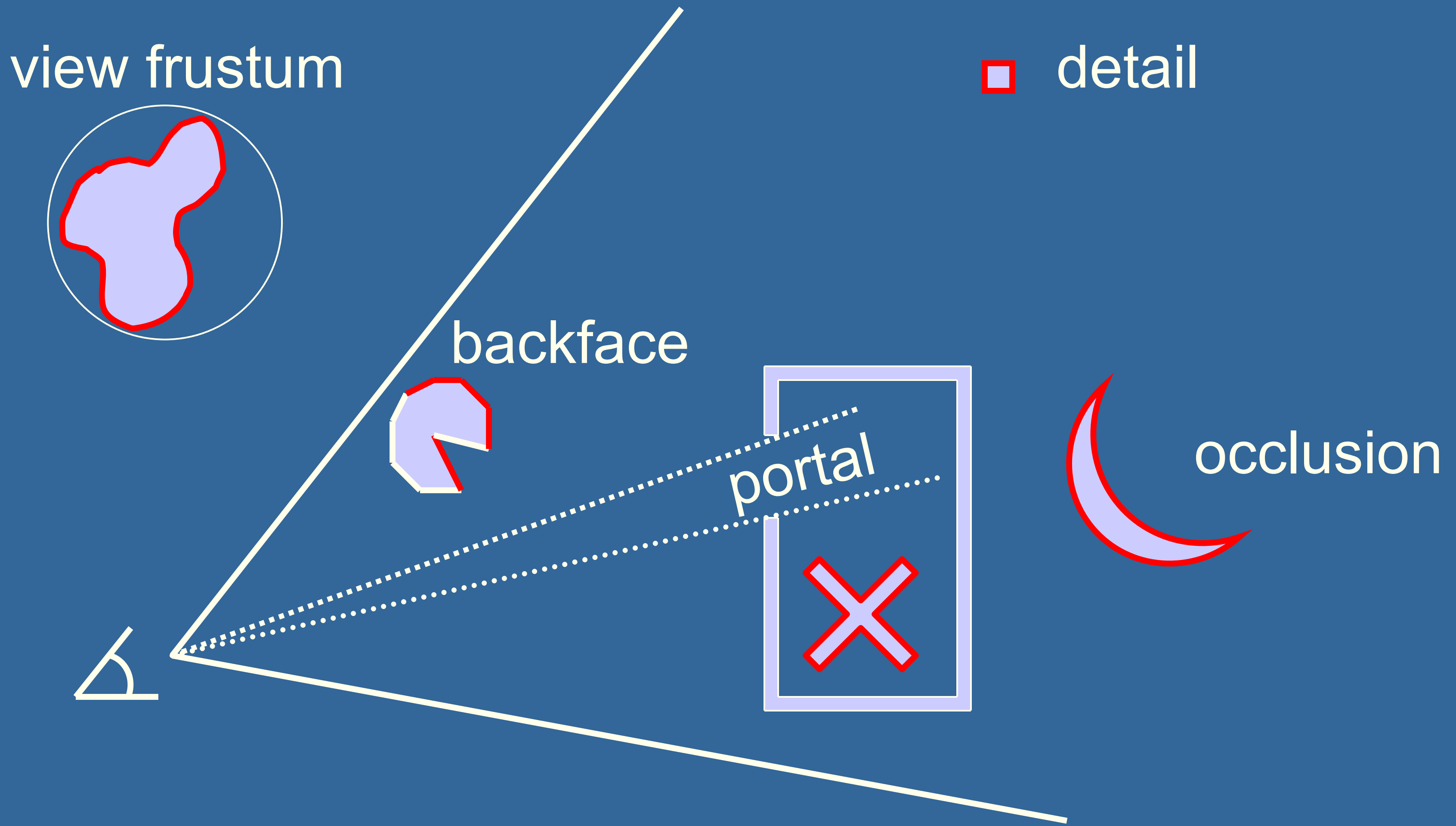
- Spatial data structures are used to speed up rendering and different queries
- Why more speed?
- Graphics hardware 2x faster in ~2 years!
- Wait… then it will be fast enough!
- NOT!
- We will not be satisfied for a long time
  - Screen resolution: angular resolution in eye's macula (gula fläcken) ~0.02 degrees
    - Apple's retina screen: e.g., 2880 x 1800.
  - Realism: global illumination, stereo, holographic displays
  - Geometrical complexity: virtually 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)

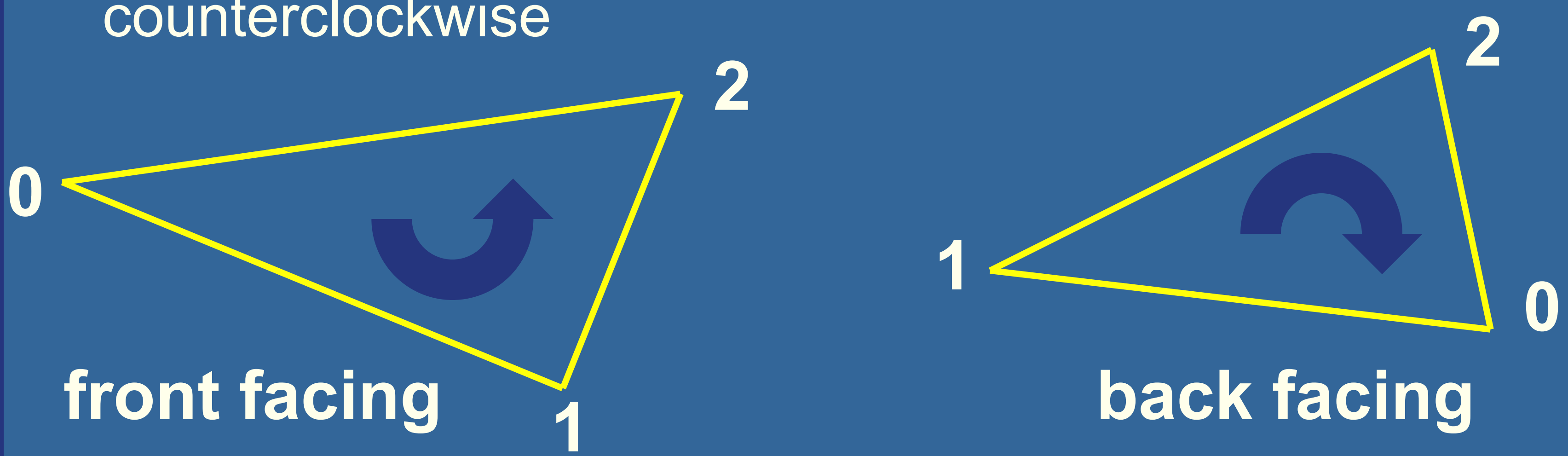


Backface Culling

- 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

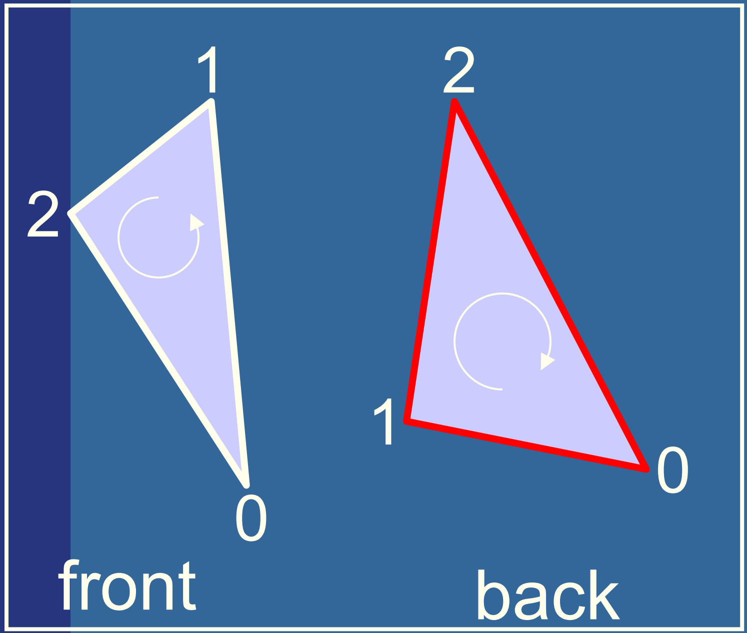
Backface culling (cont'd)

- 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

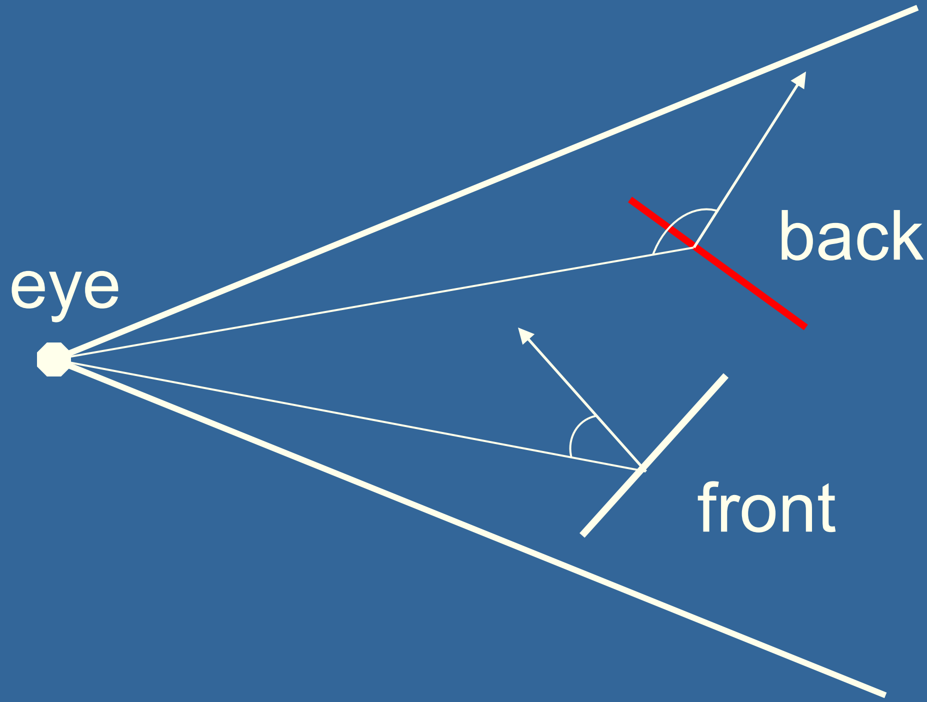


How to cull backfaces

- Two ways in different spaces:



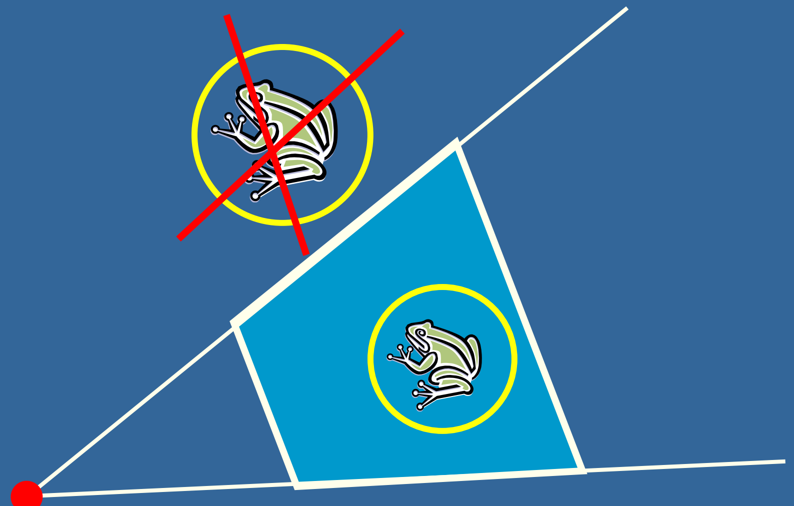
screen space



eye space

View-Frustum Culling

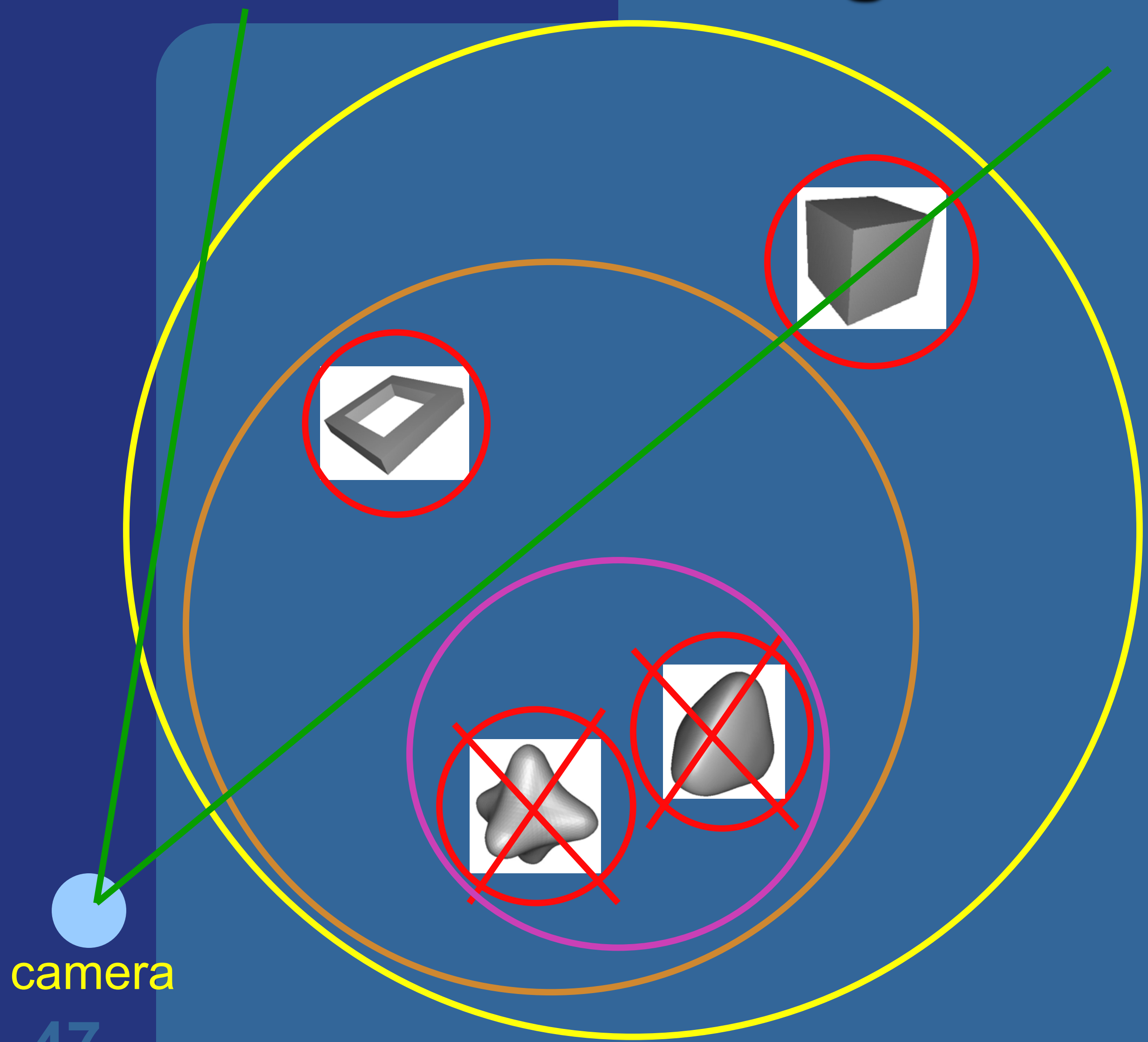
- Bound every “natural” group of primitives by a simple volume (e.g., sphere, box)
- If a bounding volume (BV) is outside the view frustum, then the entire contents of that BV is also outside (not visible)

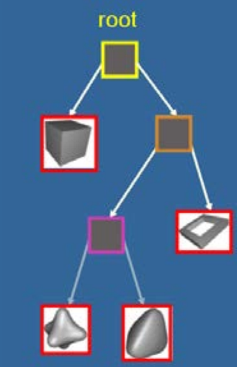


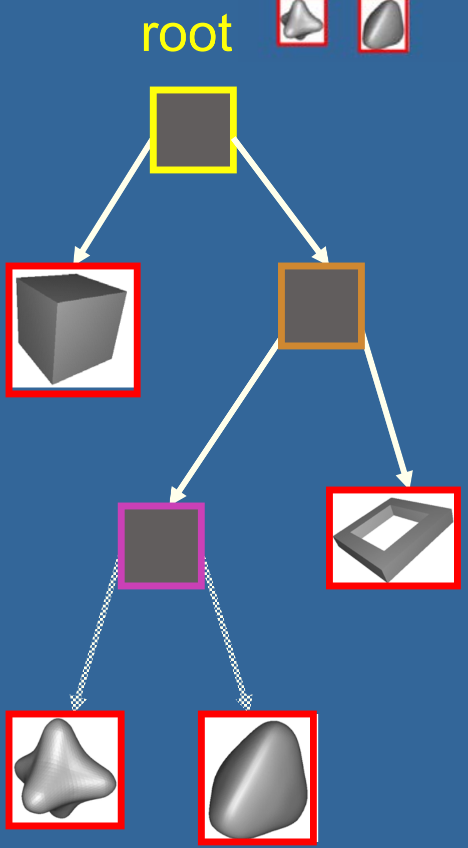
Can we accelerate view frustum culling further?

- 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

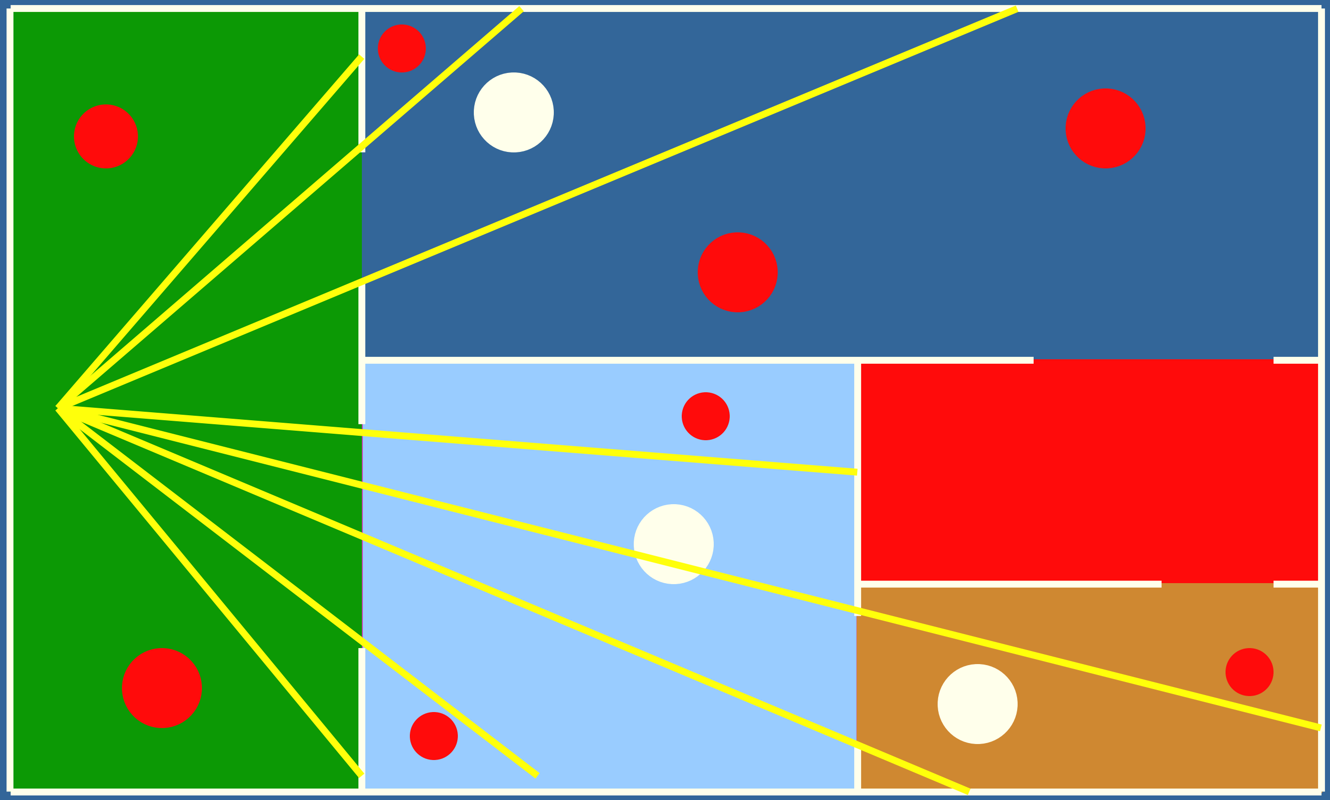
Portal Culling

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:

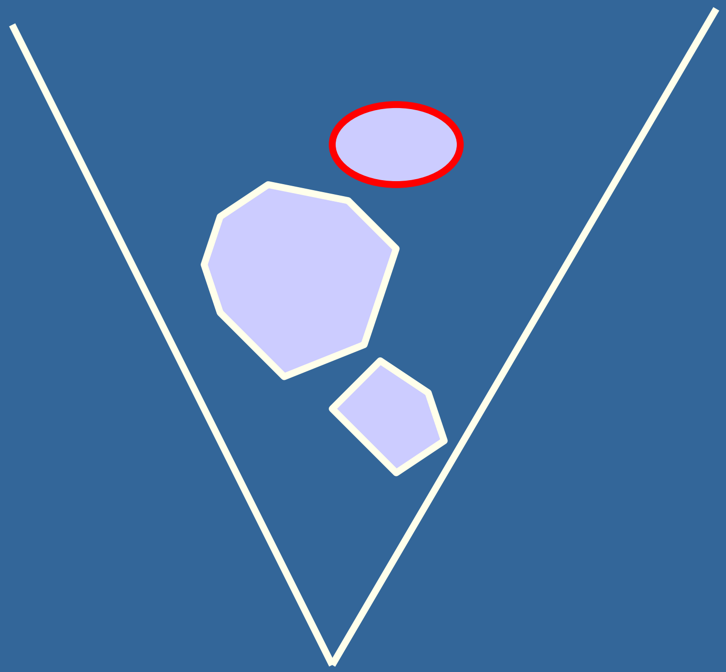
- Divide into cells with portals (build graph)
- For each frame:
  - Locate cell of viewer and init 2D AABB to whole screen
  - * Render current cell with View Frustum culling w.r.t. AABB
  - Traverse to closest cells (through portals)
  - Intersection of AABB & AABB of traversed portal
  - Goto *

Portal Culling Algorithm (2)

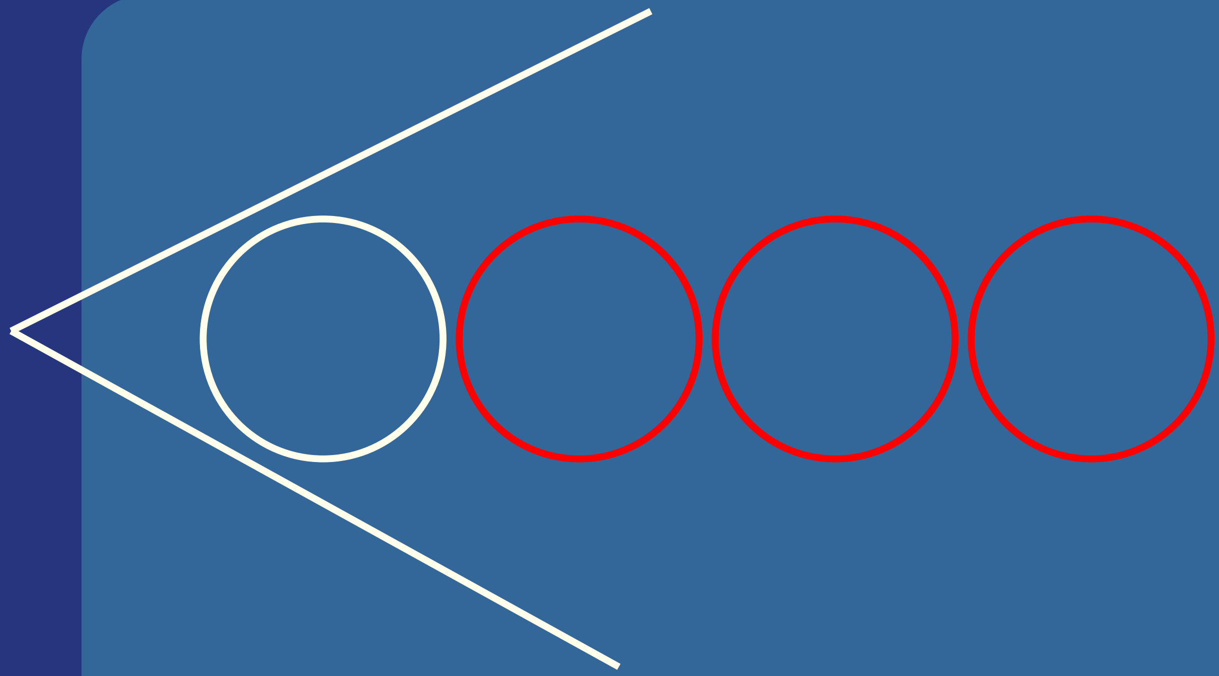
- When to exit:
  - When the current AABB is empty
  - When we do not have enough time to render a cell ("far away" from the viewer)
- Also: mark rendered objects

Occlusion Culling

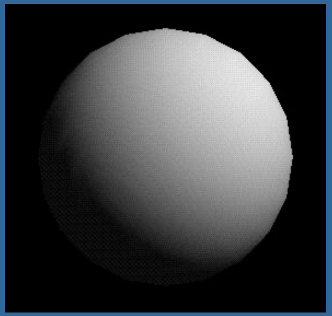
- Main idea: Objects that lies completely “behind” another set of objects can be culled
- Hard problem to solve efficiently
- Has been lots of research in this area
  - OpenGL: “Occlusion Queries”



Example



final image



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

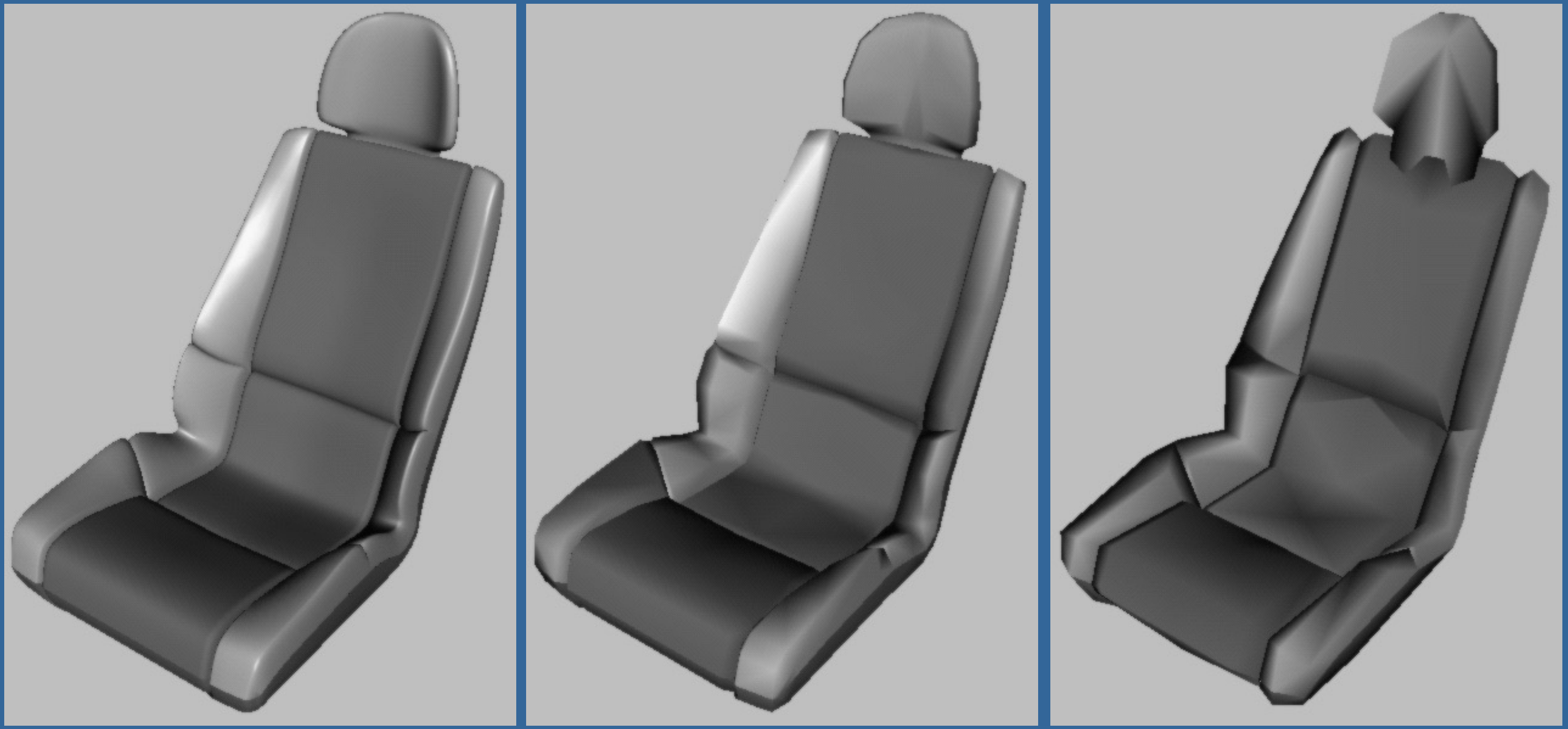
Occlusion culling algorithm

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;
```

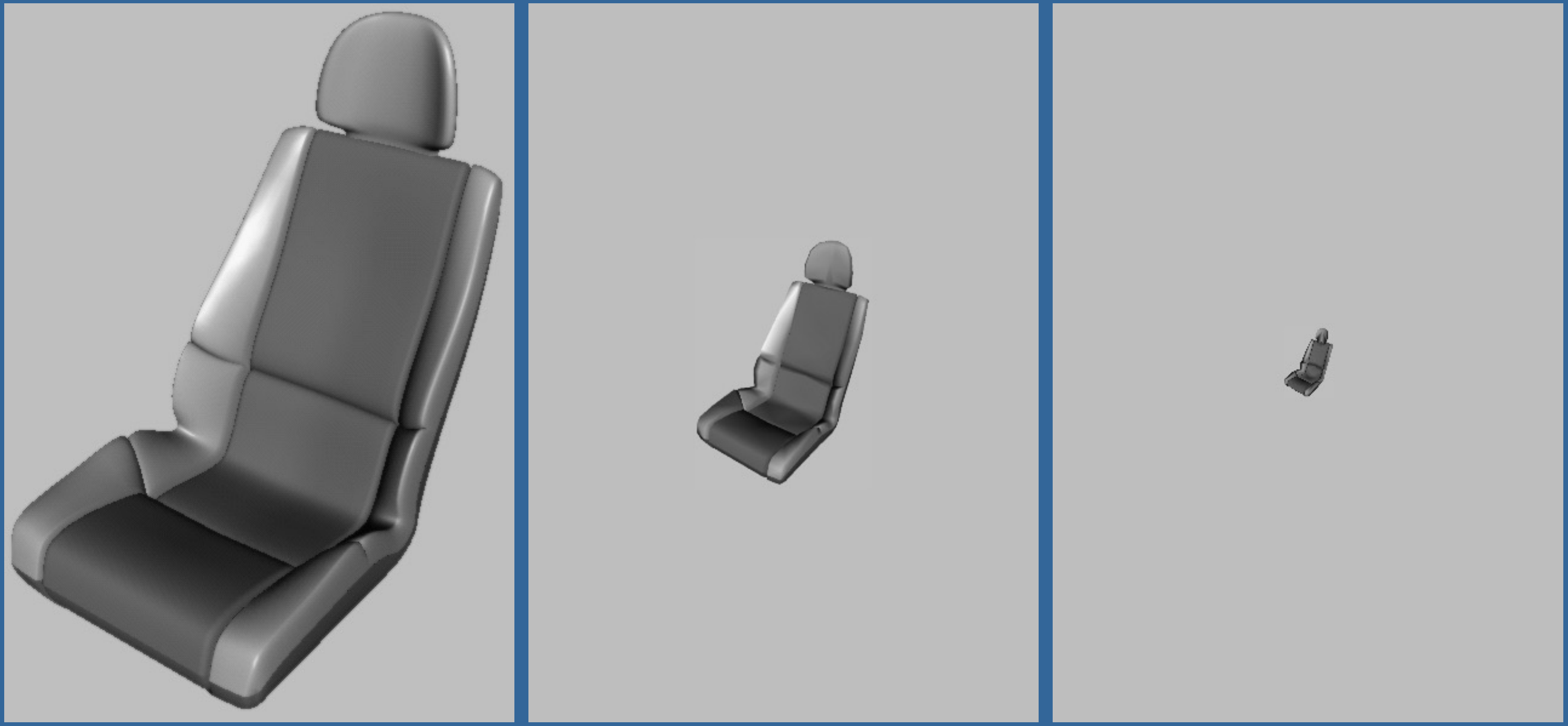
Level-of-Detail Rendering

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



LOD rendering

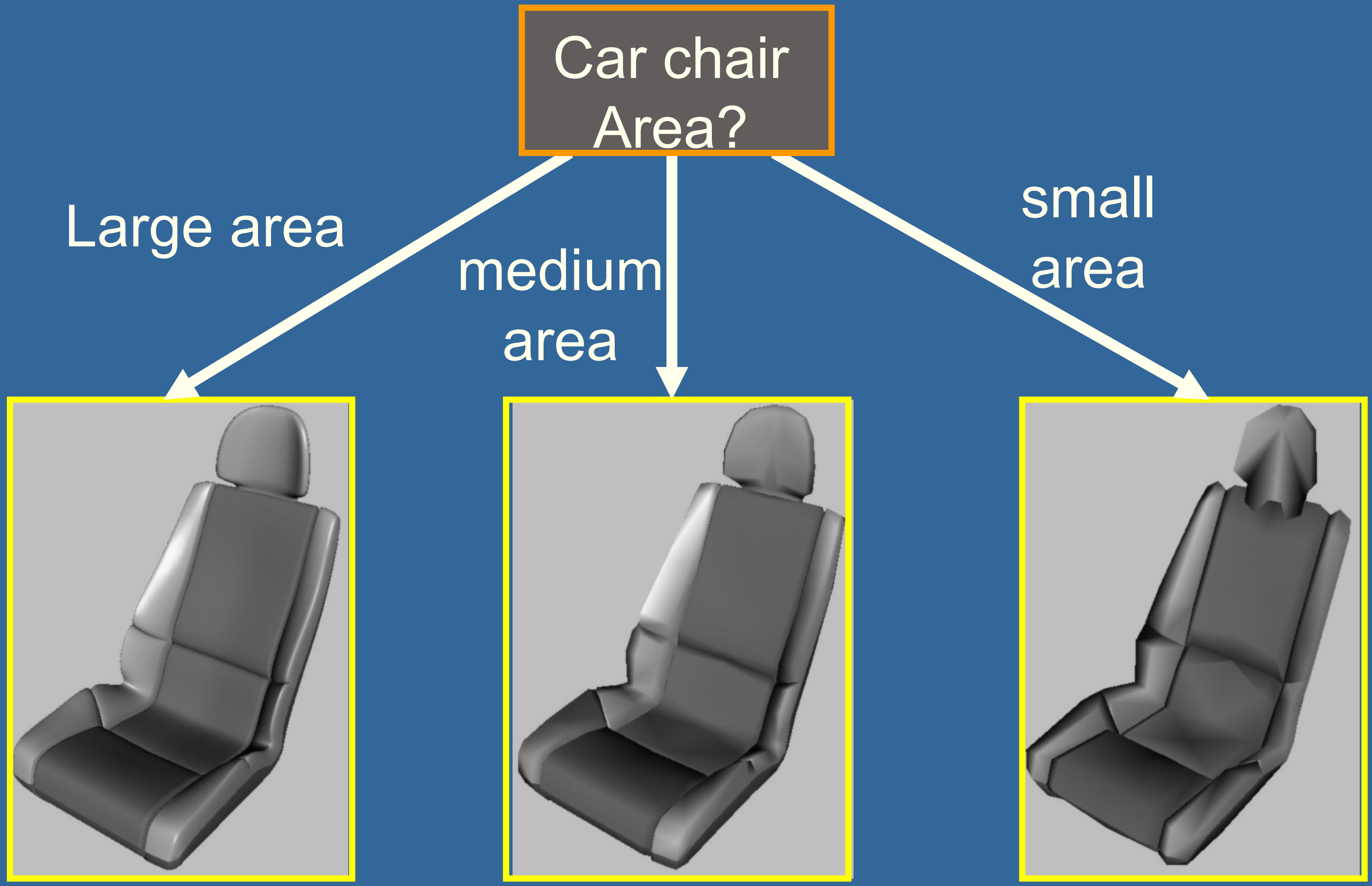
- Not much visual difference, but a lot faster



- Use area of projection of BV to select appropriate LOD

LODs

Let the render system select most suitabe model based on projected size on the screen.



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

Exercise

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

What you need to know

- Describe how use BVHs.
- Top-down construction of BVH, AABSP-tree,
- Construction + sorting with AABSP and Polygon-Aligned BSP
- Octree/quadtree
- Culling – VFC, Portal, Detail, Backface, Occlusion
  - Backface culling – screenspace is robust, eyespace non-robust.
- What is LODs

THE END