Spatial Data Structures and Speed-Up Techniques

Ulf Assarsson

Department of Computer Science and Engineering

Chalmers University of Technology

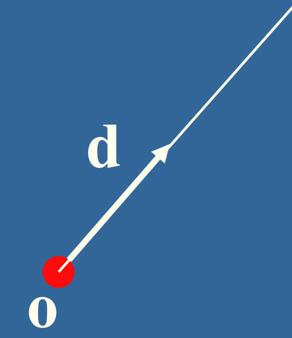
Have you done your "homework" ;-) ?

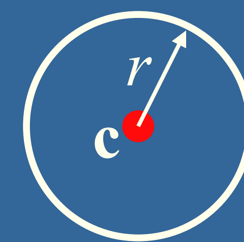
Exercises

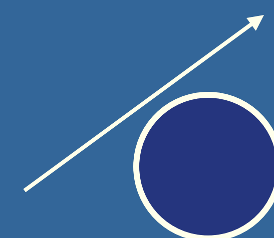
- Create a function (by writing code on paper) that tests for intersection between:
  - two spheres
  - a ray and a sphere
  - view frustum and a sphere
  - Ray and triangle (e.g. use formulas from last lecture)
- Make sure you understand matrices:
  - Give a scaling matrix, translation matrix, rotation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

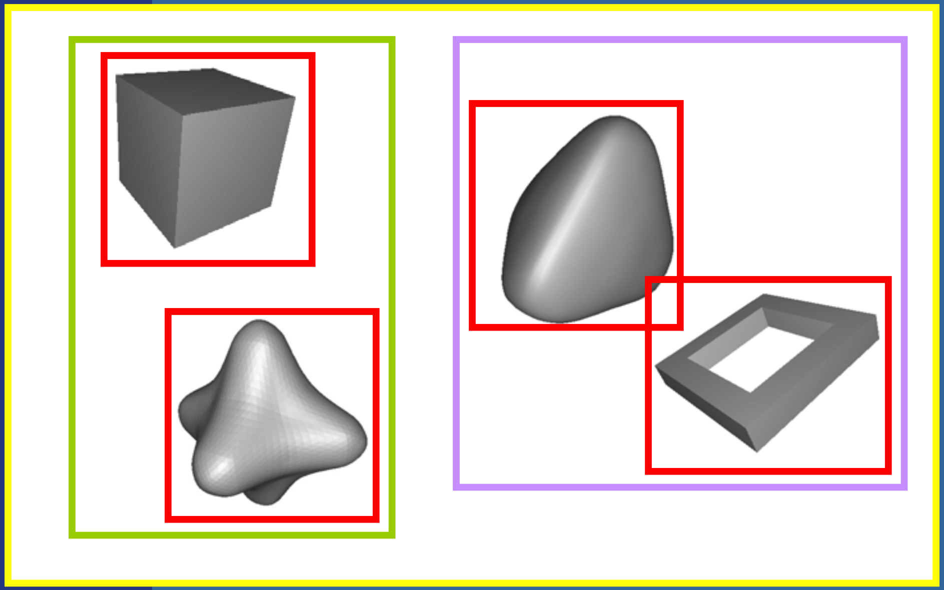
Spatial data structures

- 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 & Movie production tools use them extensively

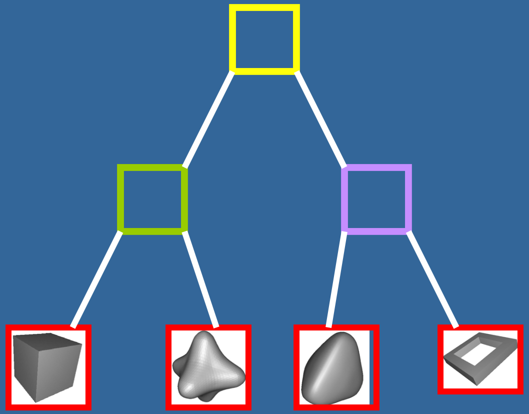
Bounding-Volume Hierarchy
– BOTTOM-UP construction:

- Organizes geometry in some hierarchy

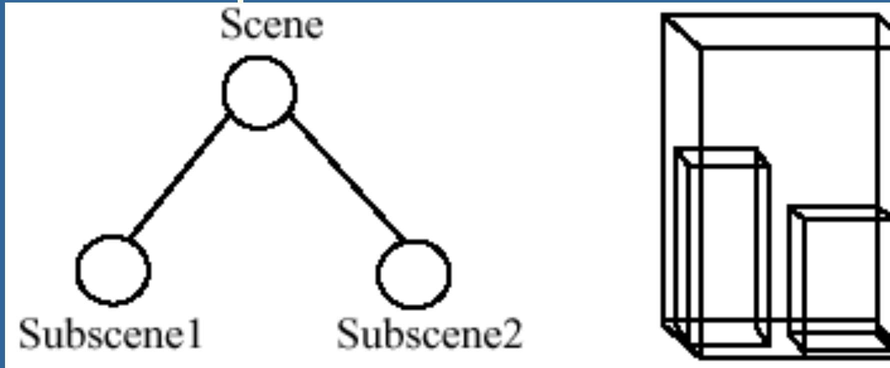
In 2D space



Data structure

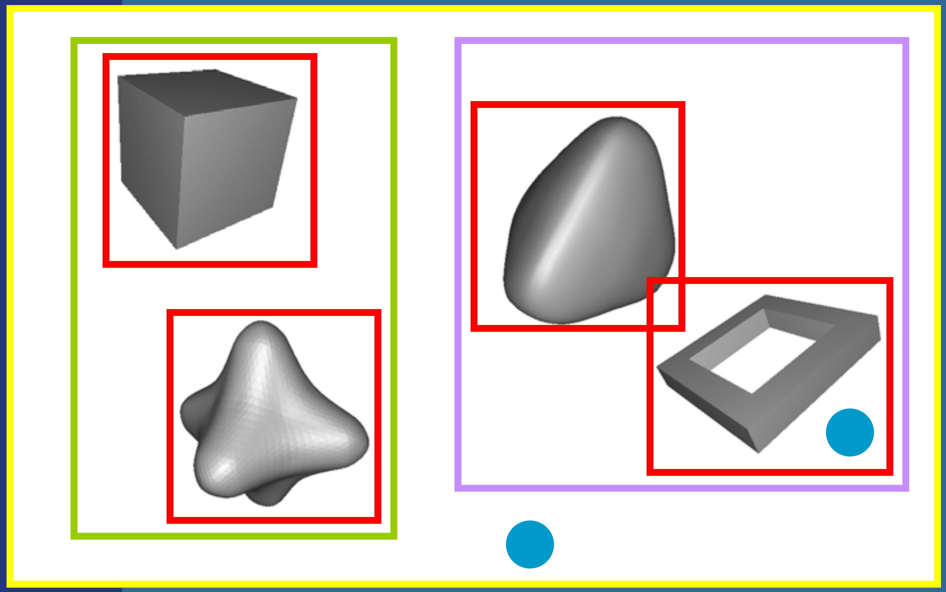


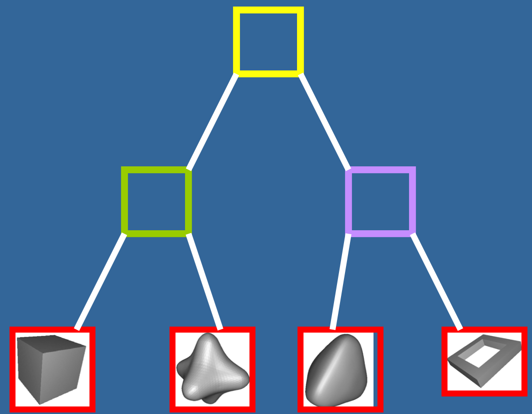
In 3D space:



What's the point with hierarchies? An example

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



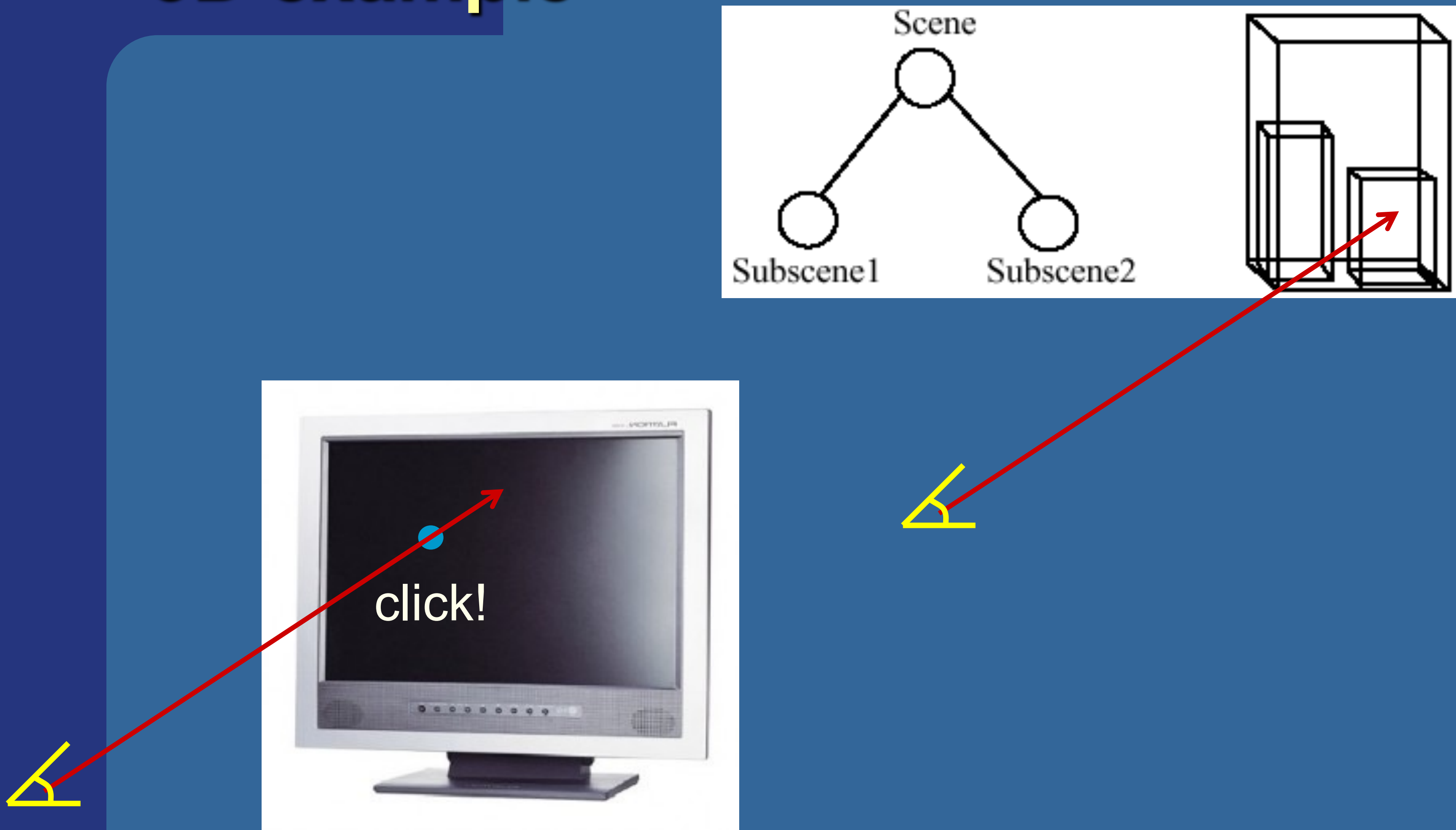


click!

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)

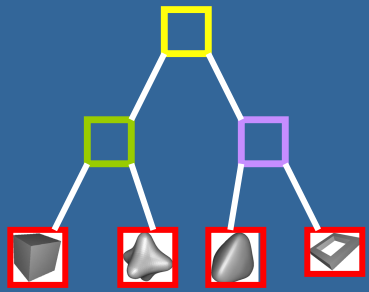
3D example



Bounding Volume Hierarchy (BVH)

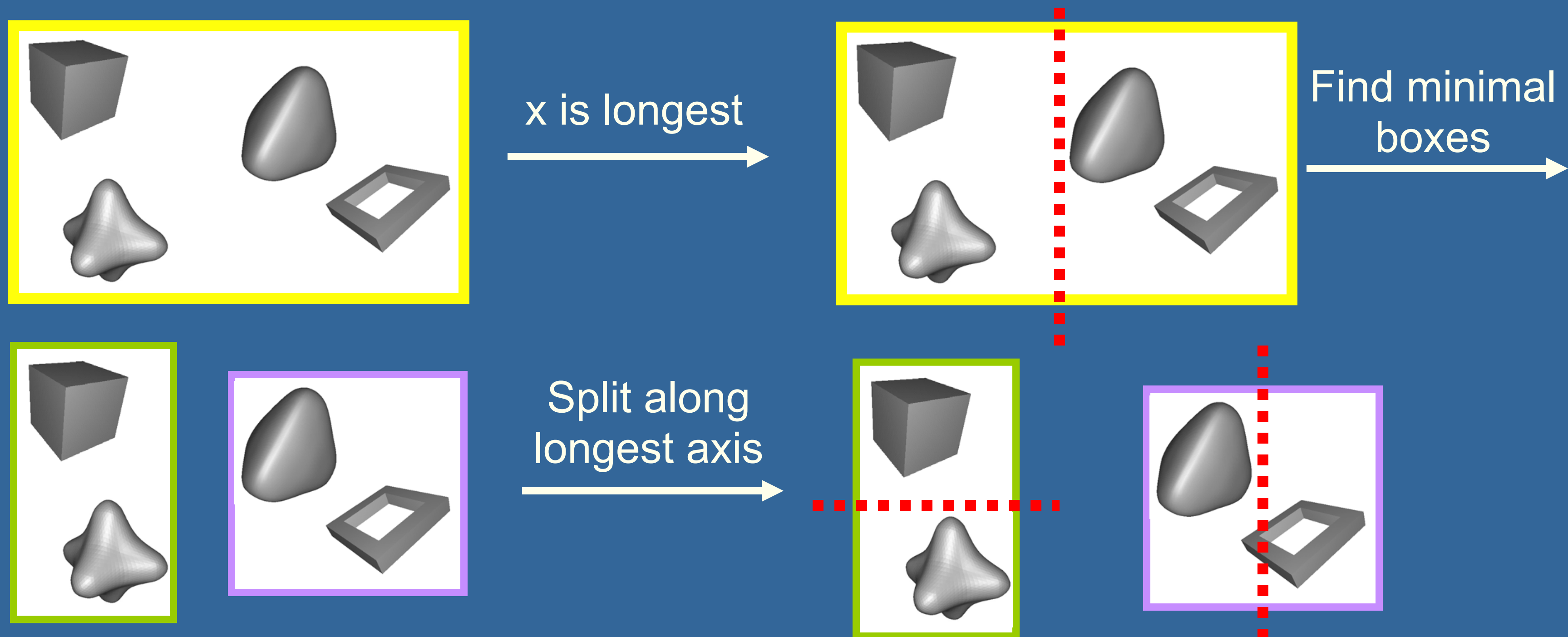
- Most common bounding volumes (BVs):
  - Axis-Aligned Bounding Boxes (AABB)
  - But can also use spheres and Oriented Bounding Boxes (OBBs)
  - AABB hierarchies are used by the NVIDIA RTX chip

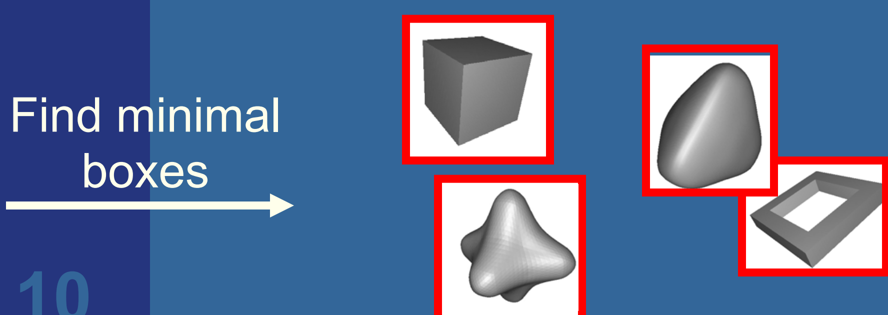
- The BV does not contibute to the rendered image -- rather, encloses an object
- The data structure is a **tree**
  - Leaves hold geometry
  - Internal nodes hold BVs that enclose all geometry in its subtree



Bounding-Volume Hierarchy
– TOP-DOWN construction:

- Find minimal box, then split along longest axis





Called TOP-DOWN method

Works similarly for other BVs

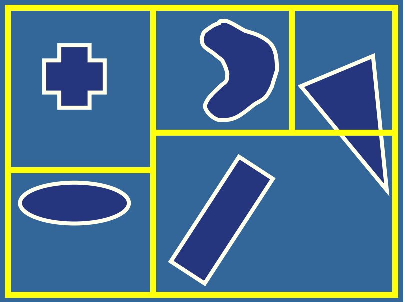
Example

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

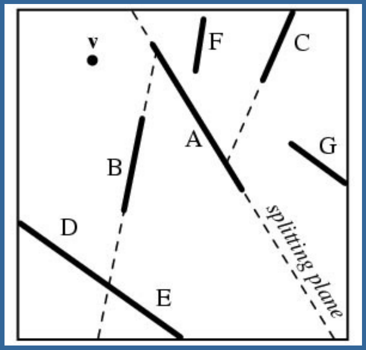
Kd-tree = Axis Aligned BSP tree

Binary Space Partitioning (BSP) Trees

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



Axis-aligned

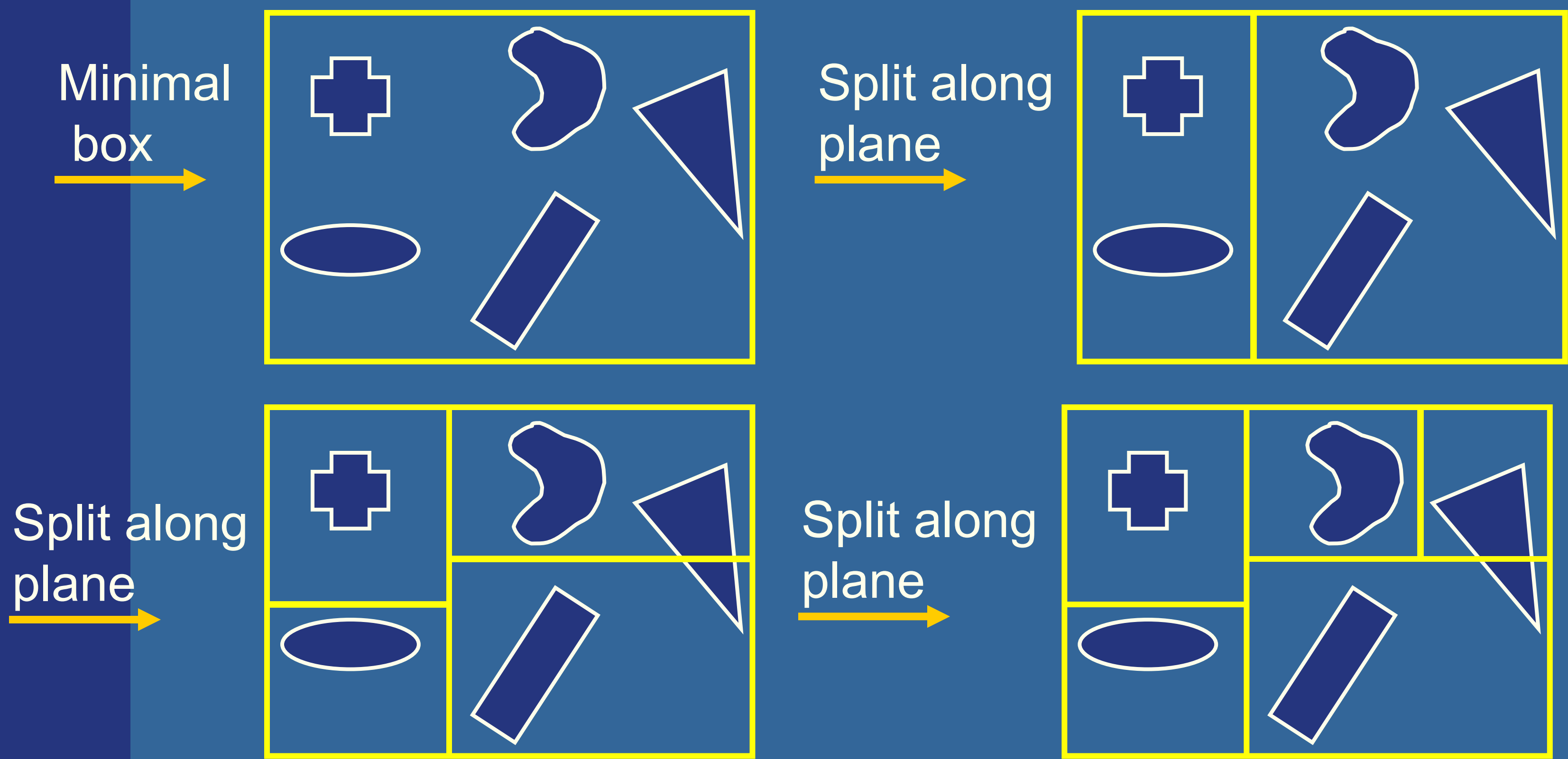


Polygon-aligned

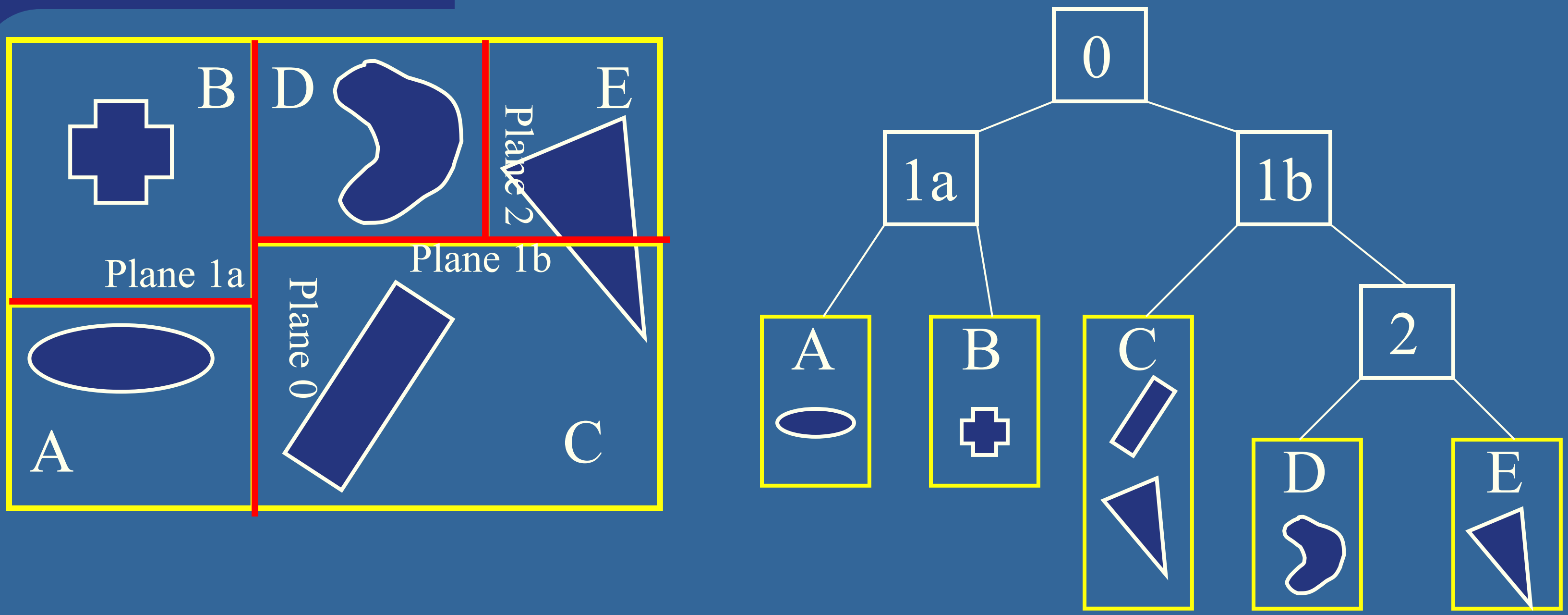
Axis-Aligned BSP tree
– TOP-DOWN construction

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

- Axis-aligned => Can only choose a splitting plane along x,y, or z



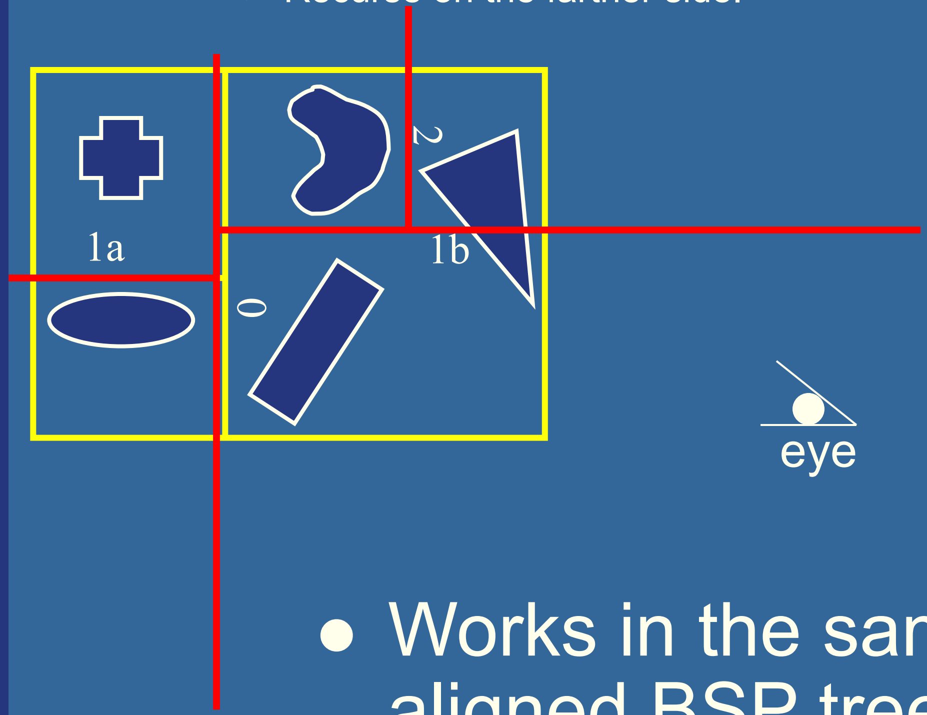
Axis-Aligned BSP tree
– tree structure

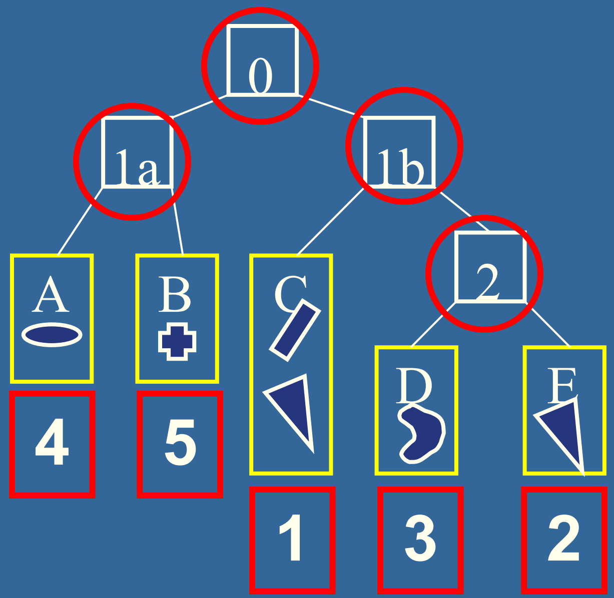


- Each internal node holds a divider plane
- Leaves hold geometry
- Differences compared to BVH
  - BSP tree encloses entire space and provides sorting
  - The BV hierarchy can have spatially overlapping nodes(no sort)
  - BVHs can use any desirable type of BV

Axis-aligned BSP tree
– Rough sorting front-to-back w.r.t camera

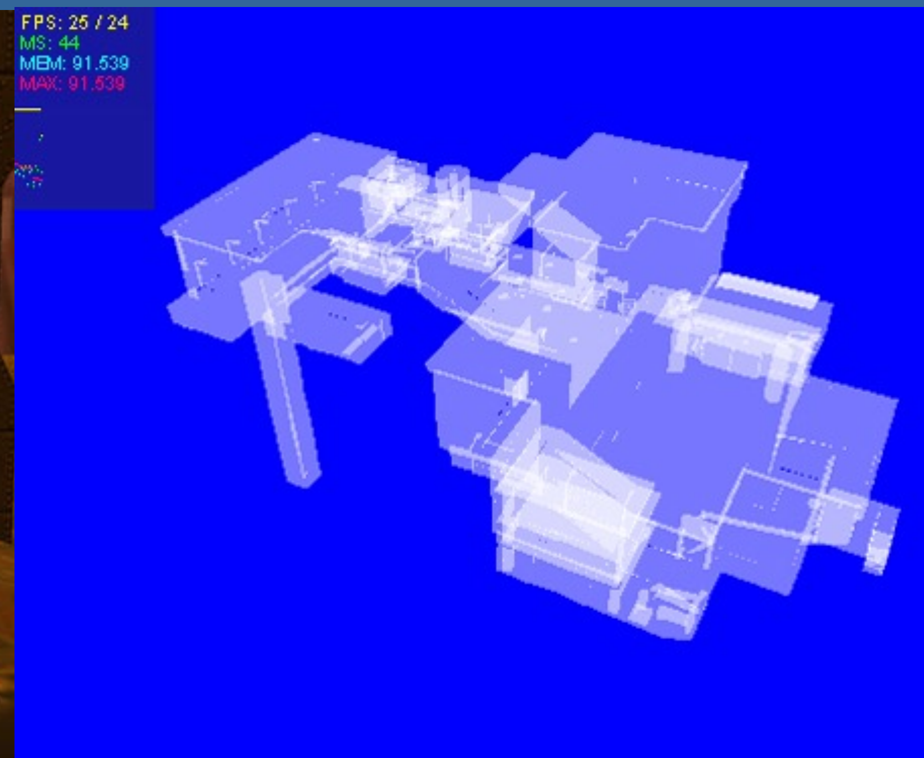
- Test the planes, recursively from root, against the point of view. For each traversed node (for front-to-back rendering):
  - If node is leaf, draw the node's geometry
  - else
    - Recurse on the "hither" side with respect to the eye (to sort front to back)
    - Recurse on the farther side.





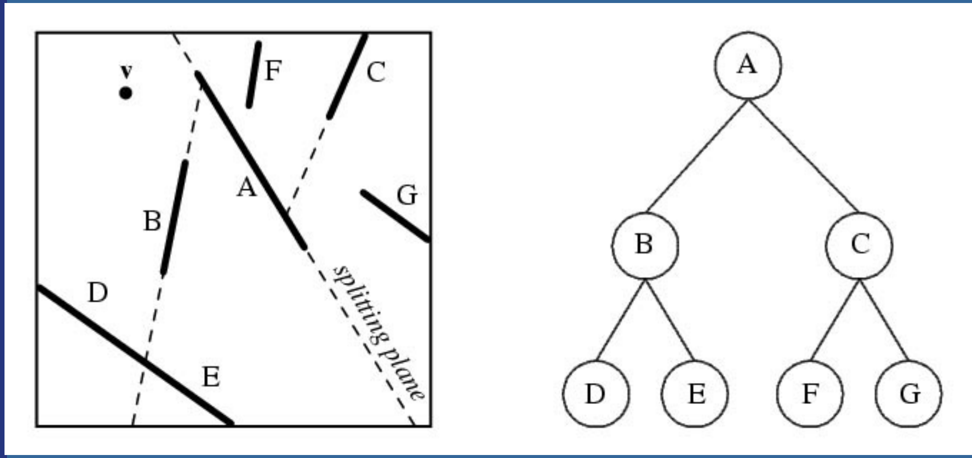
- Works in the same way for polygon-aligned BSP trees --- but that gives exact sorting

Polygon Aligned BSP tree – Quake 2



Polygon-aligned BSP tree

- Allows exact sorting from camera
  - Since planes clip intersecting triangles
- Very similar to axis-aligned BSP tree
  - But the triangle planes are used as the splitting planes.



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
```

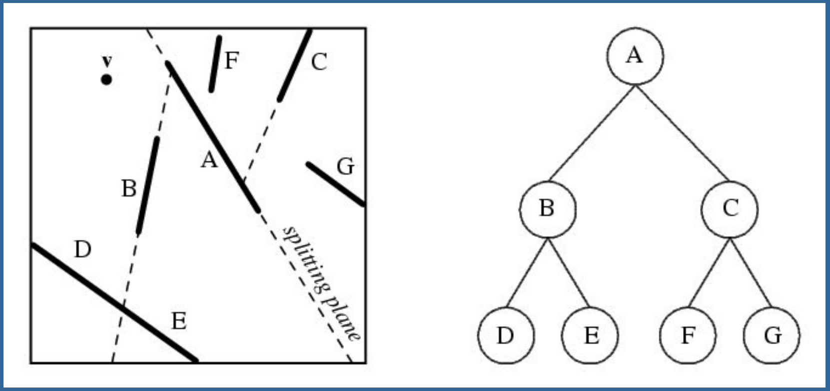
Polygon-aligned BSP tree

```
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.
}
```

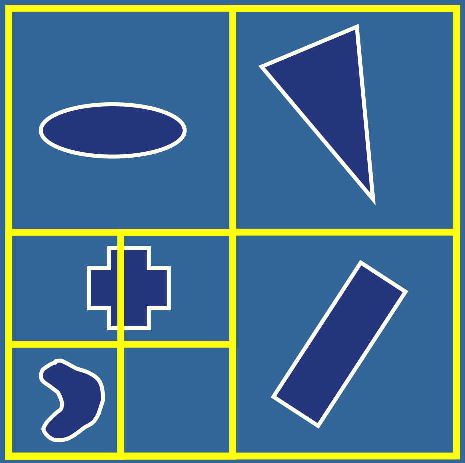
Drawing Back-to-Front {

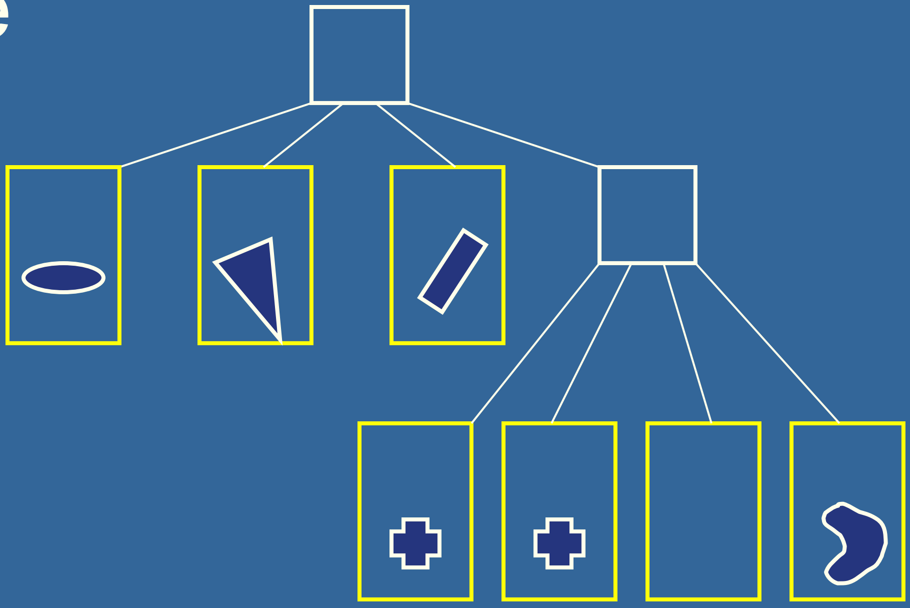
```
    recurse on farther* side of P;
    Draw P;
    Recurse on hither* side of P;
}
*With respect to viewpoint v
```



Octrees (1)

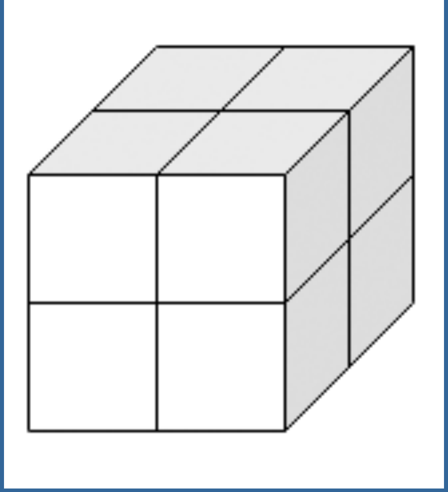
- A bit similar to axis-aligned BSP trees
- Will explain the quadtree, which is the 2D variant of an octree



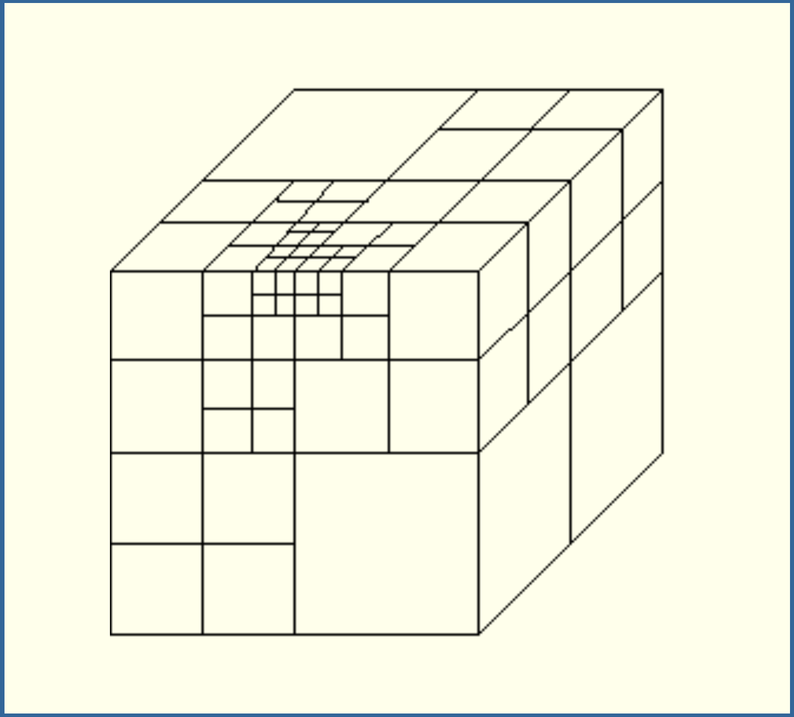


- In 3D, each square (or rectangle) becomes a box, and 8 children

Example of Octree



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



Example of octree

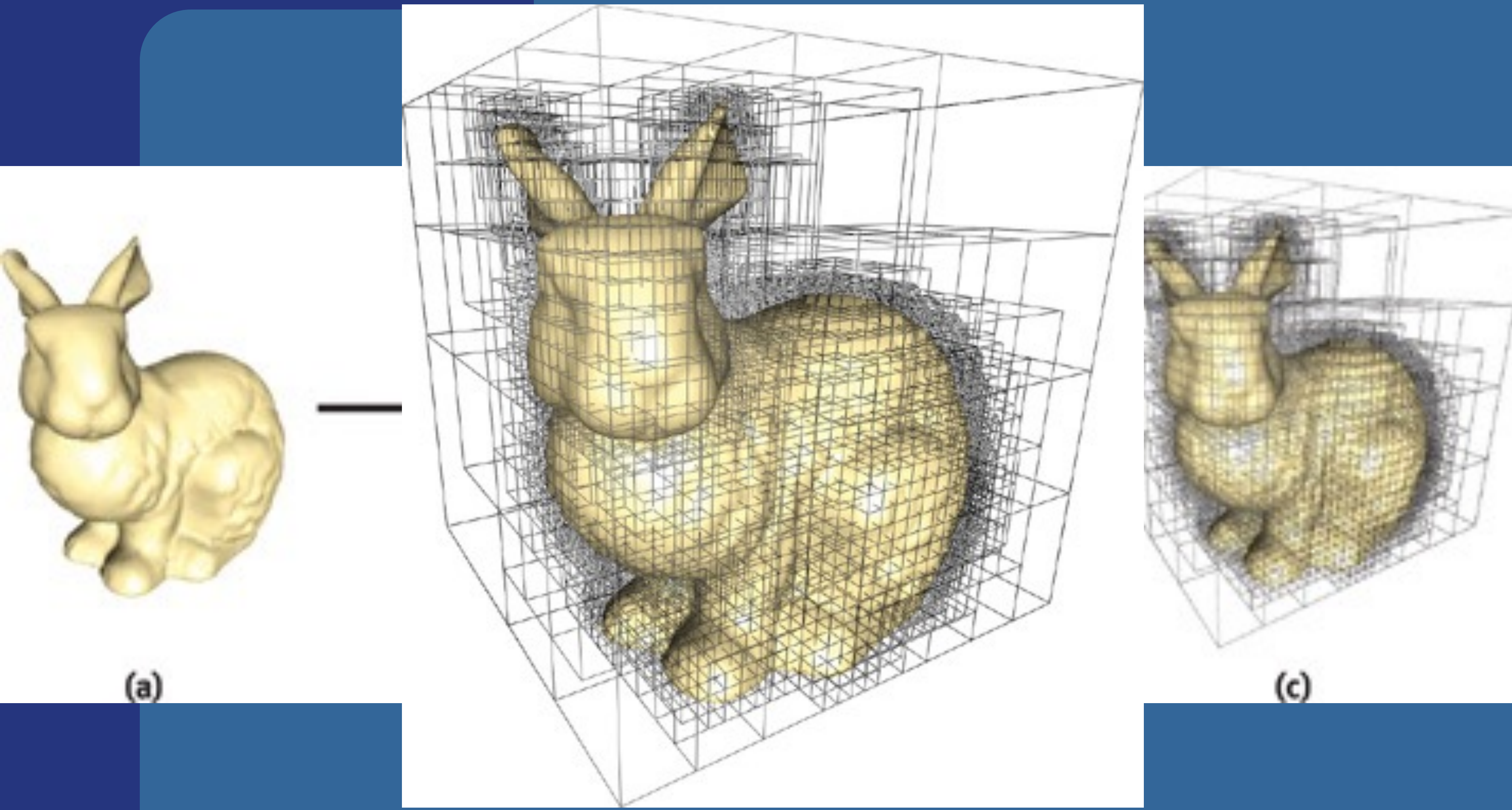


Image from Lefebvre et al.

Example of octree

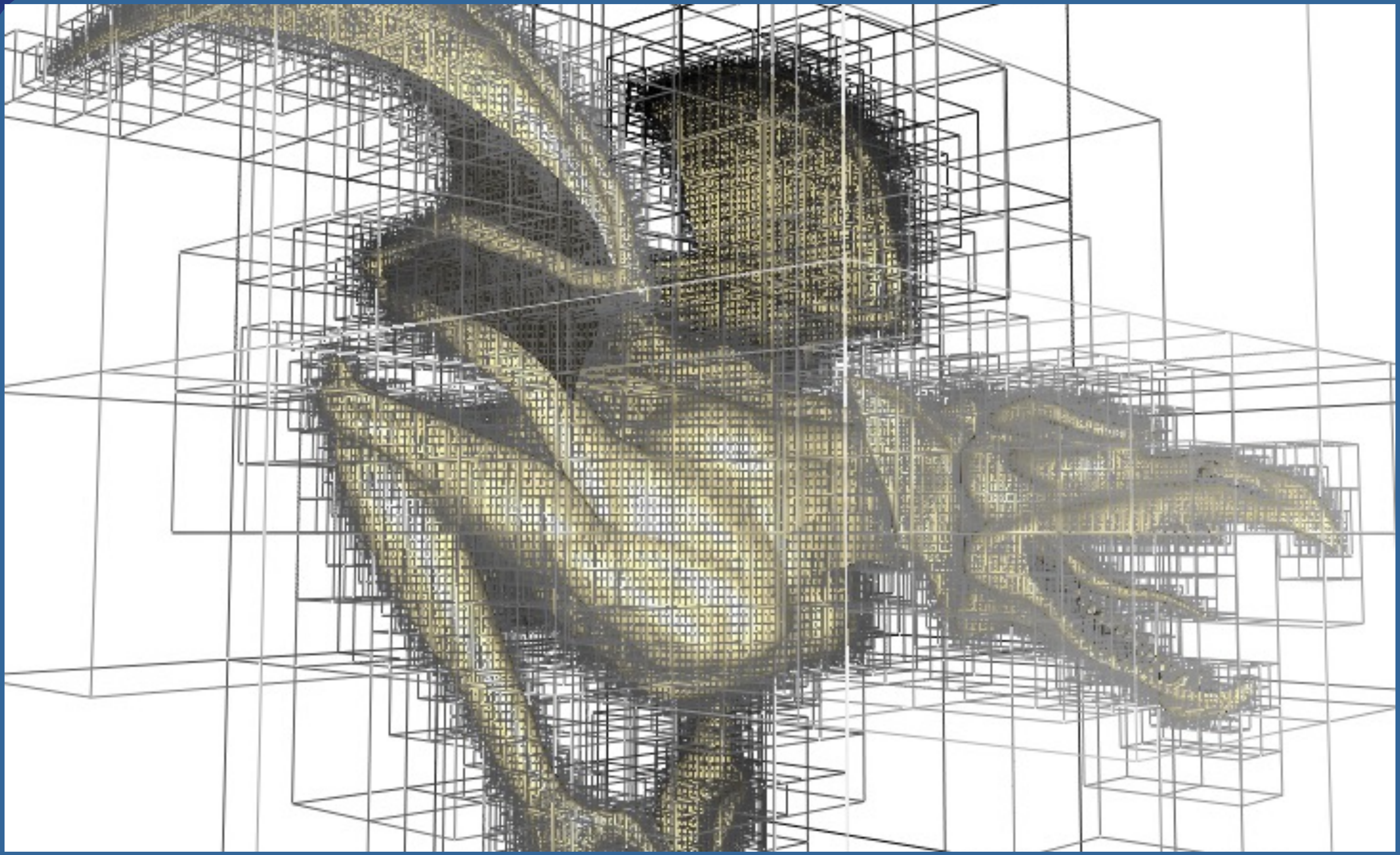


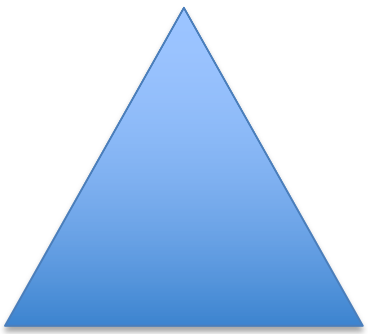
Image from Lefebvre et al.

Octrees (2)

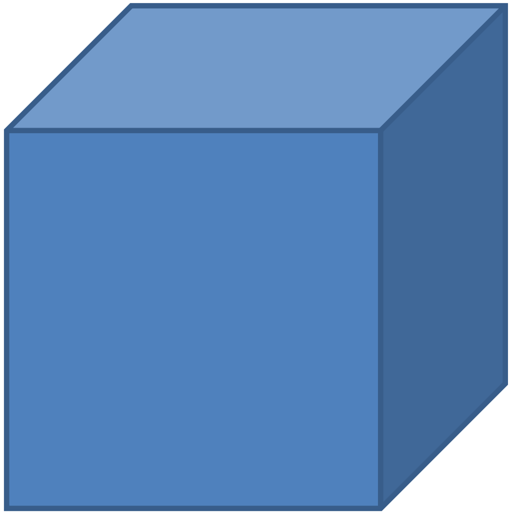
- Expensive to rebuild (BSPs are too)

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

Voxels



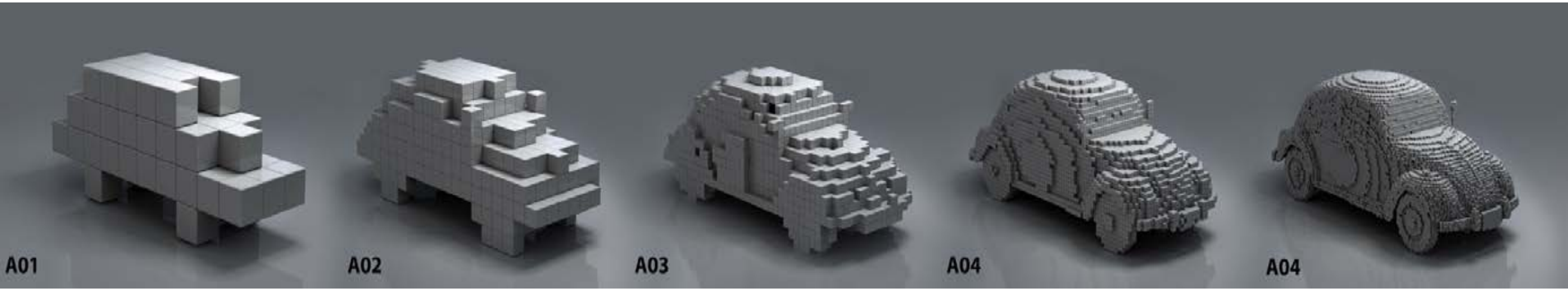
Triangle
36 bytes



Voxel
Volume – element
1 bit

Voxels

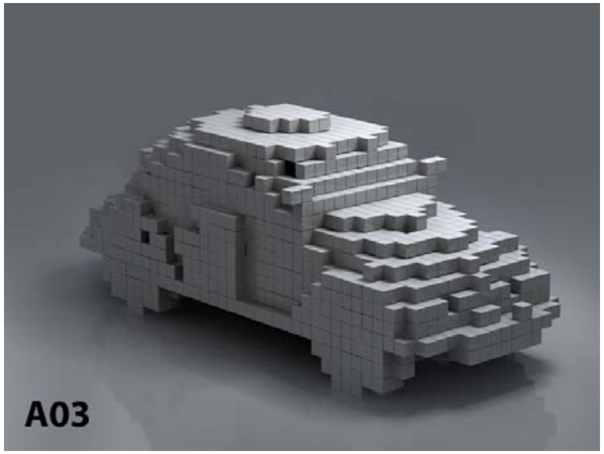
- Desirable to be able to use very high resolutions

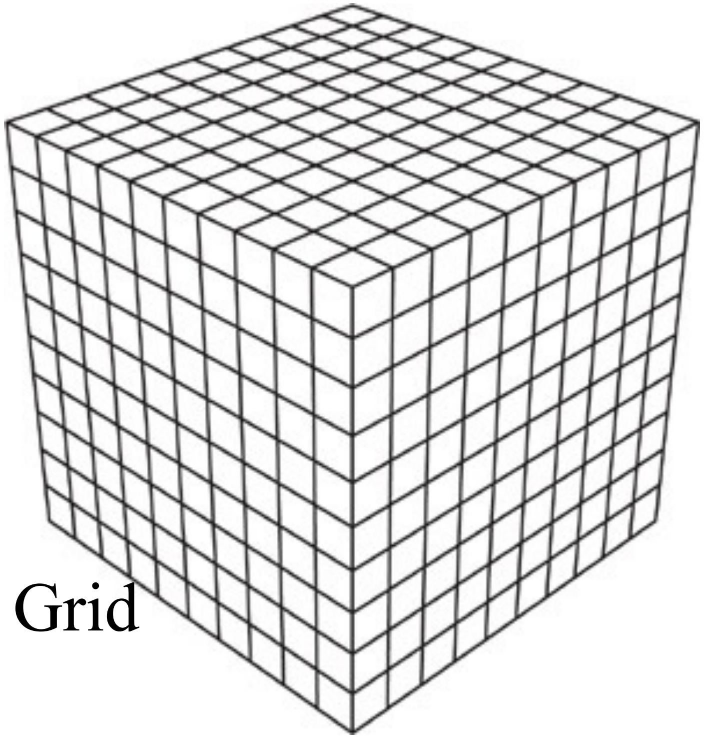


Voxels

One possible data structure:

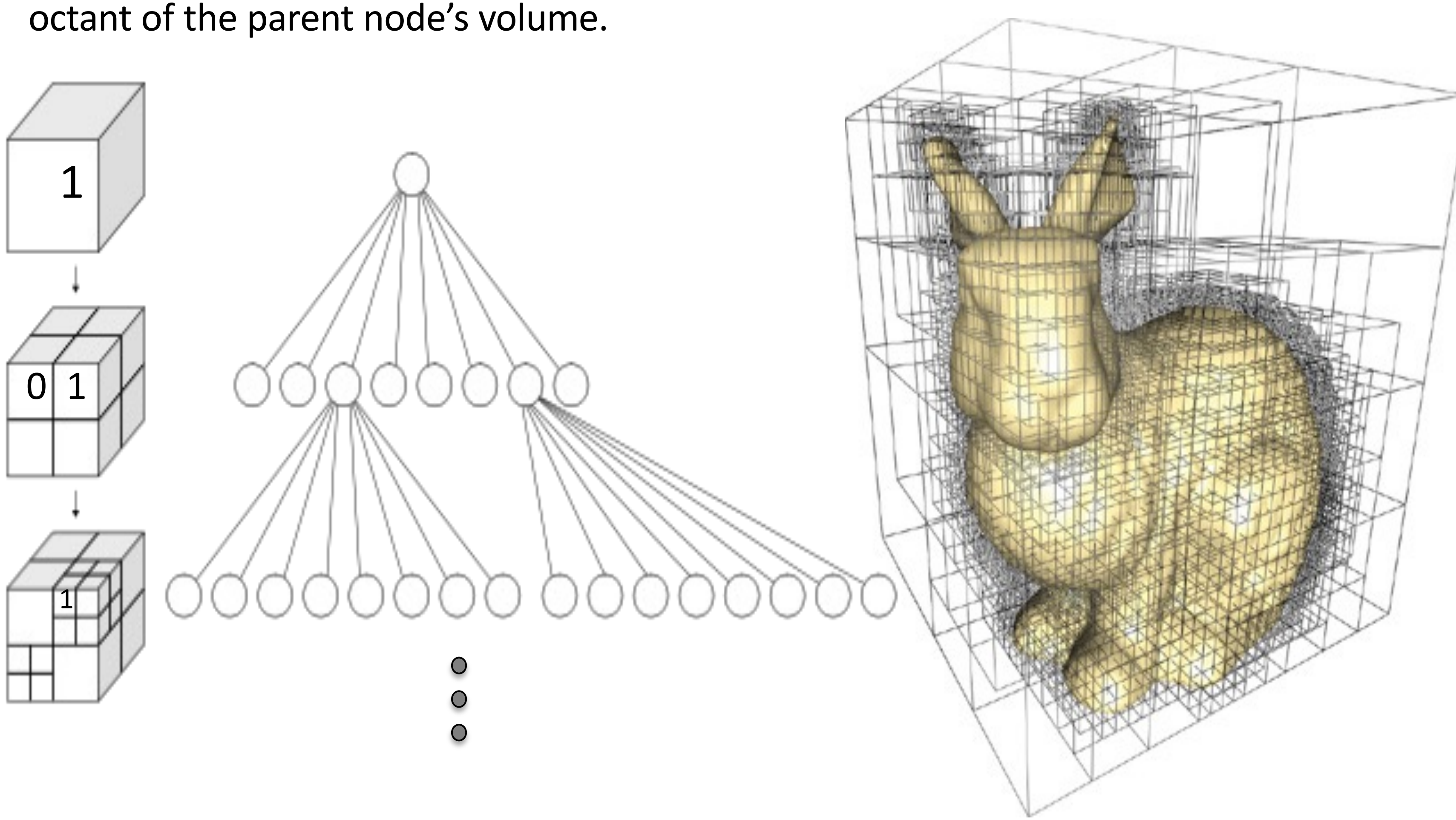
- Voxel Grids – 3D array of 0:s and 1:s





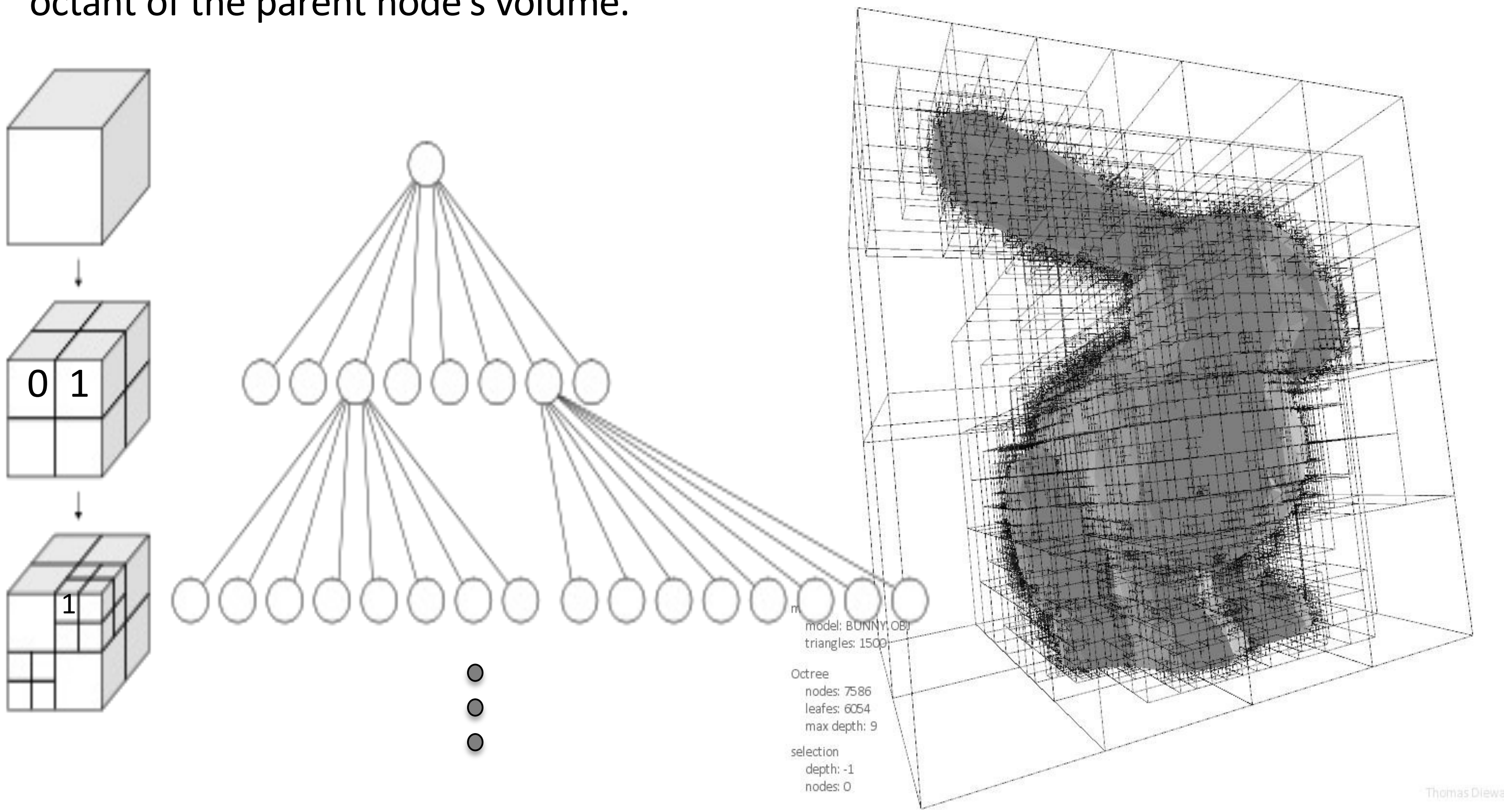
Sparse Voxel Octree

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



Sparse Voxel Octree

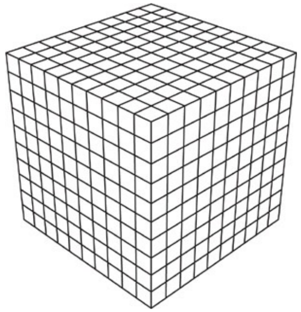
Each node has eight children, representing an octant of the parent node's volume.



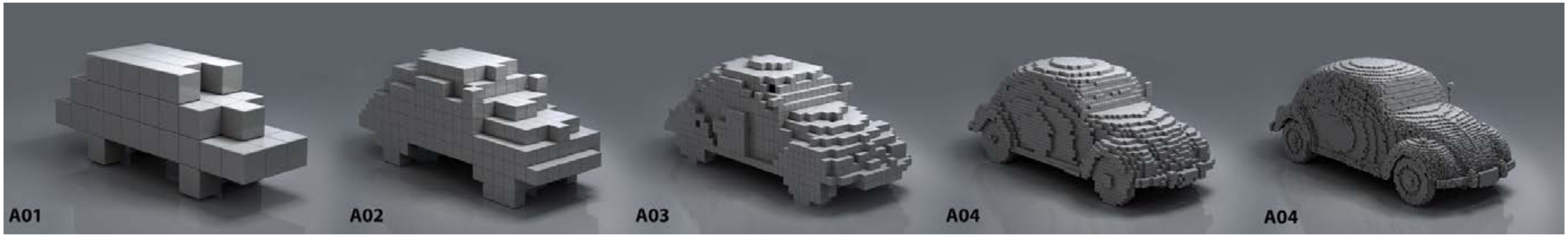
Sparse Voxel Octree

- SVO: Id Software, rage 6
- 1.15 bits/ non-empty voxel
- DAGs: e.g., down to 0.08 bit/non-empty voxel

Sparse Voxel DAGs

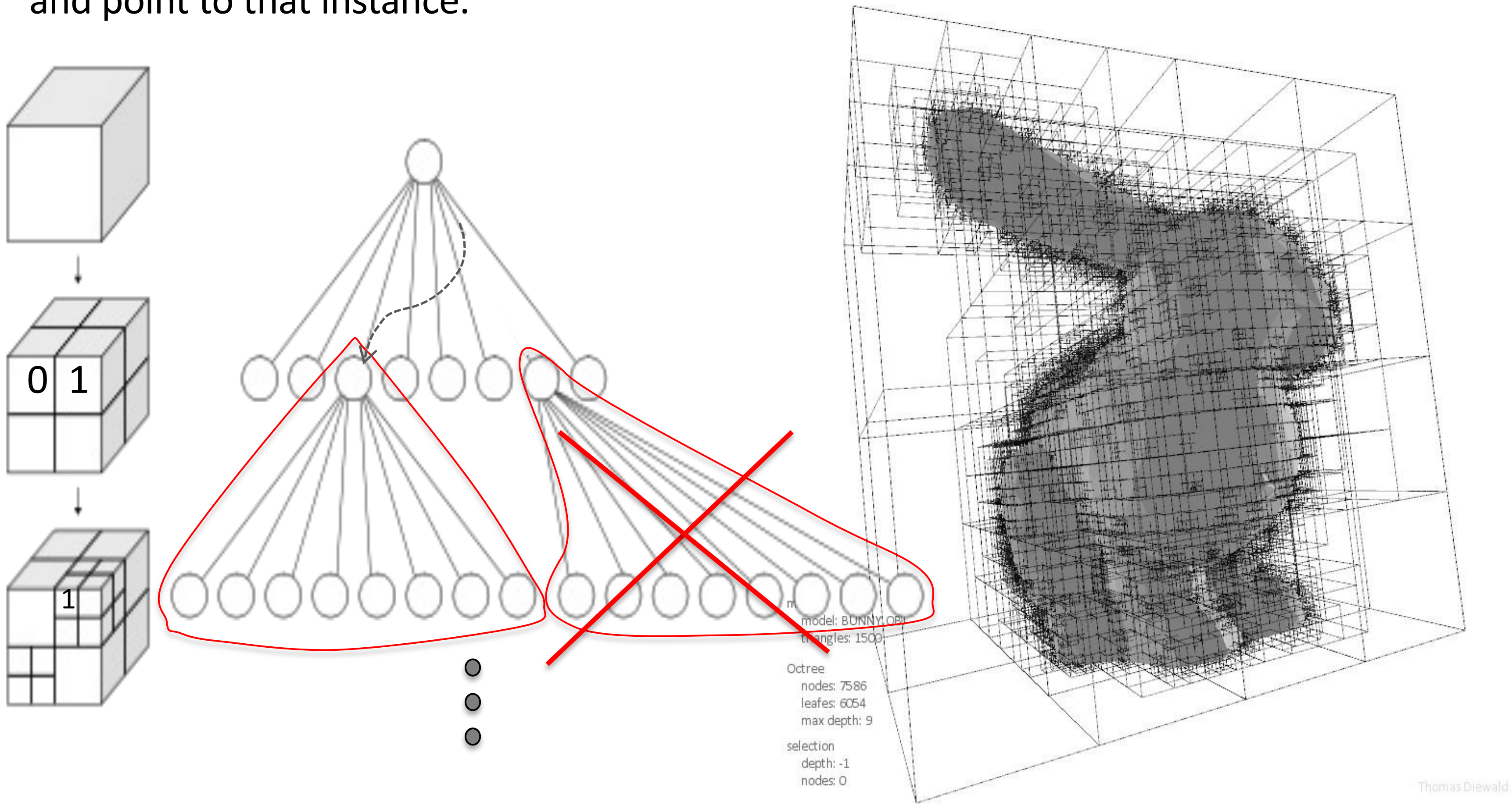


- Voxel = 1 bit.
- SVDags can currently handle scene of res = 128.000^3
  - Naively with bit grid: 262 TB
  - SVDAGs => < 1GB can be possible



Sparse Voxel DAGs

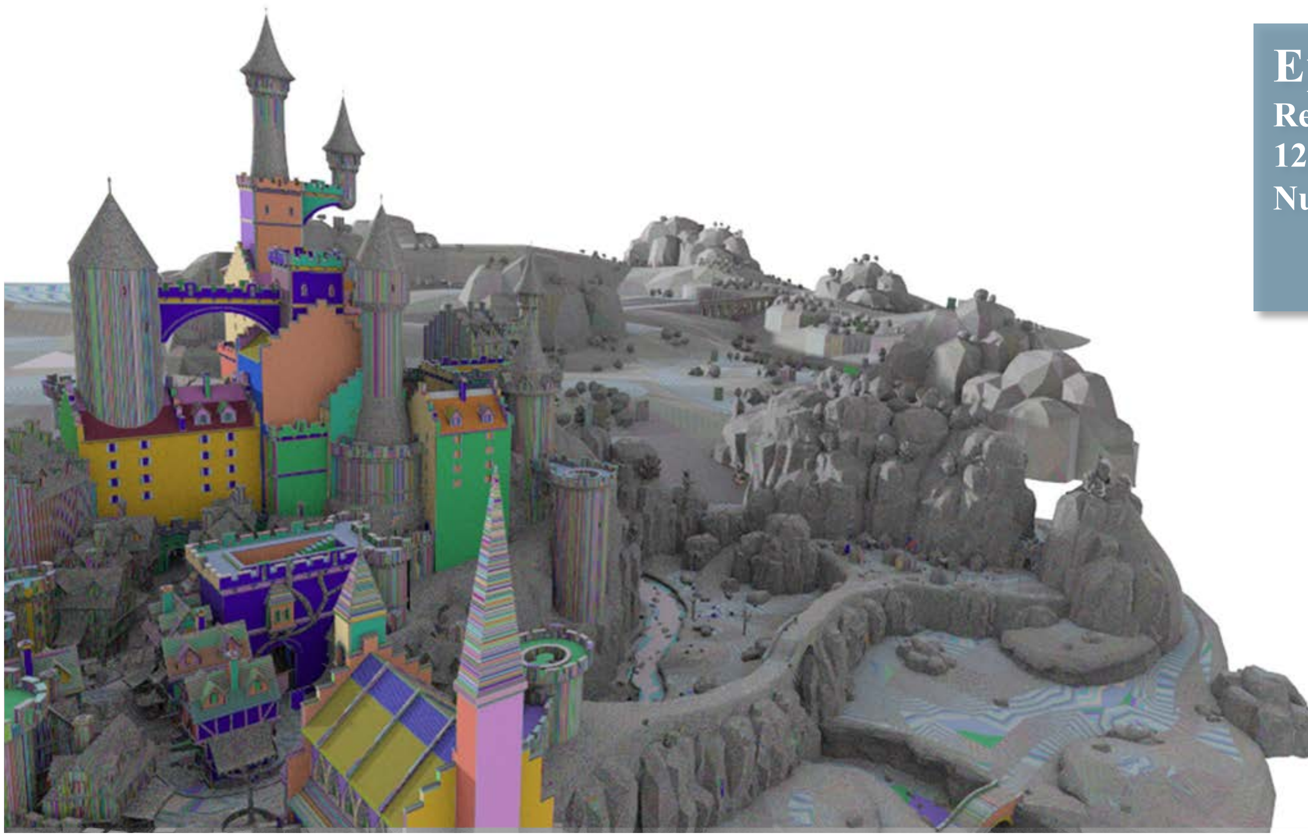
For identical subgraphs, only store one instance, and point to that instance.



Sparse Voxel DAGs

https://youtu.be/6zpbV6hZPWU

Visualizing Identical Subtrees

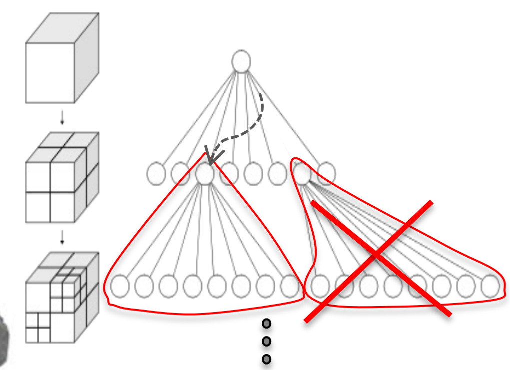


Epic Citadel

Resolution: 128K × 128K × 128K

Number of nodes

- **SVO: 5.5 billion**
- **DAG: 45 million *(0.8%)***



Visualizing Identical Subtrees

Hairball

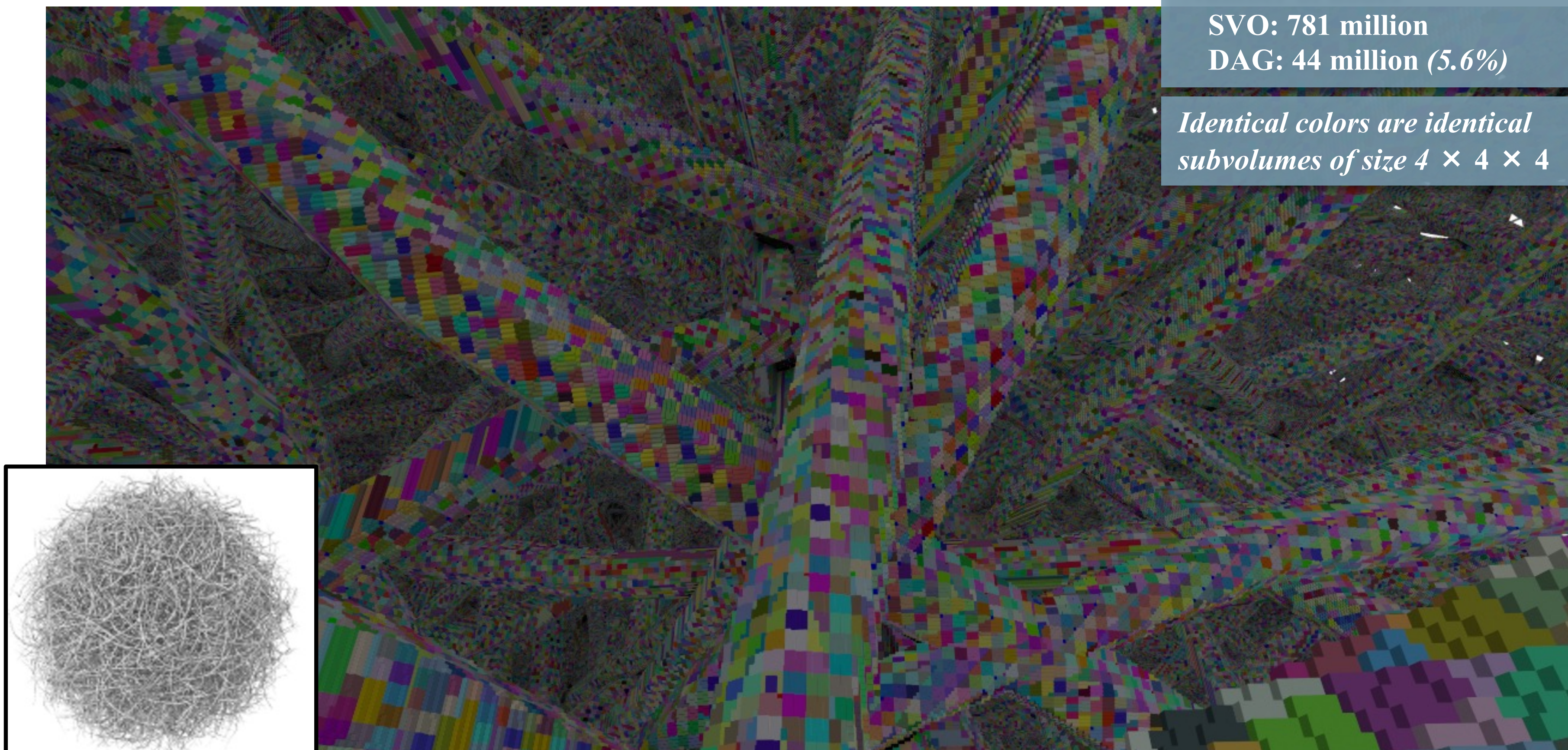
Resolution: 8K × 8K × 8K

Number of nodes

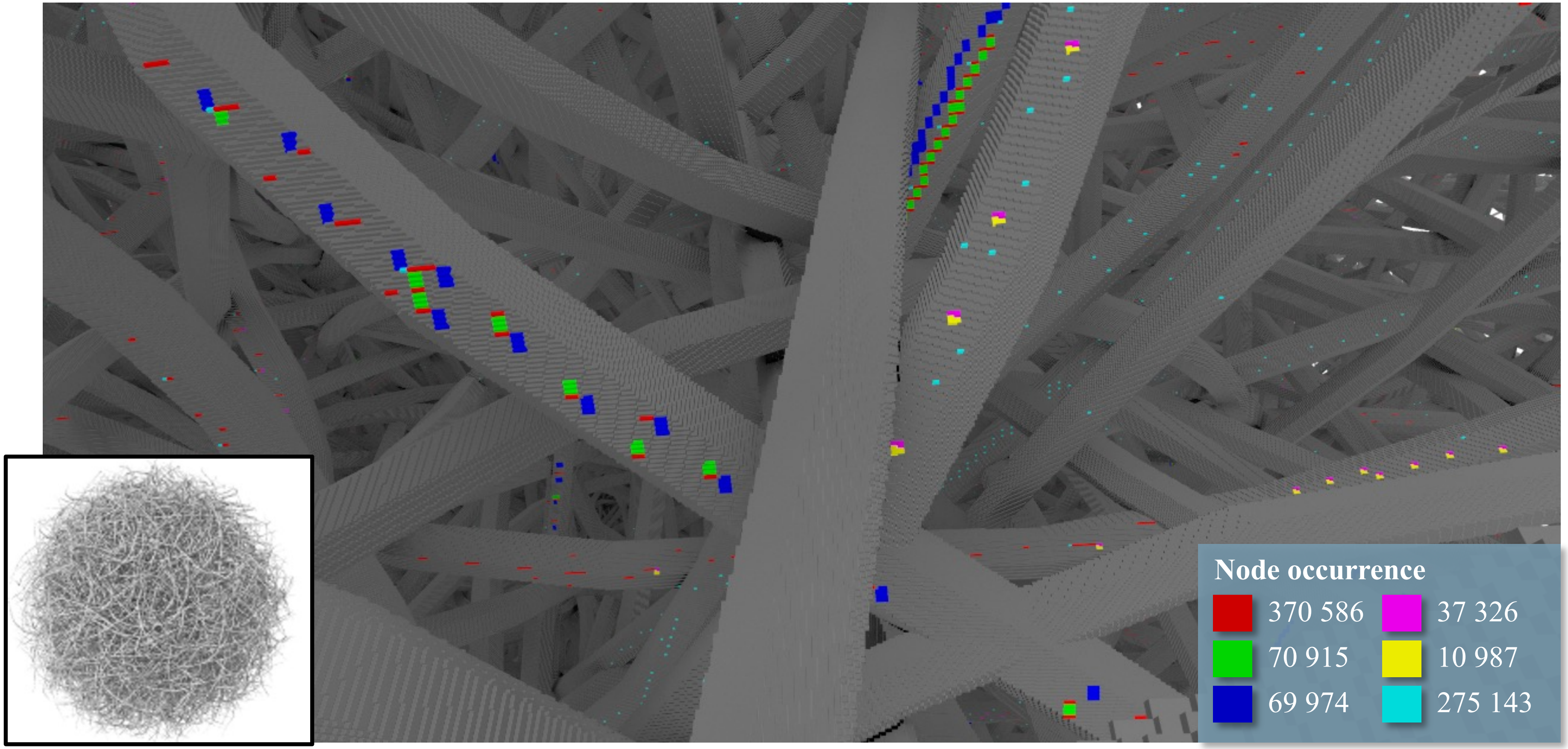
SVO: 781 million

DAG: 44 million *(5.6%)*

Identical colors are identical subvolumes of size 4 × 4 × 4



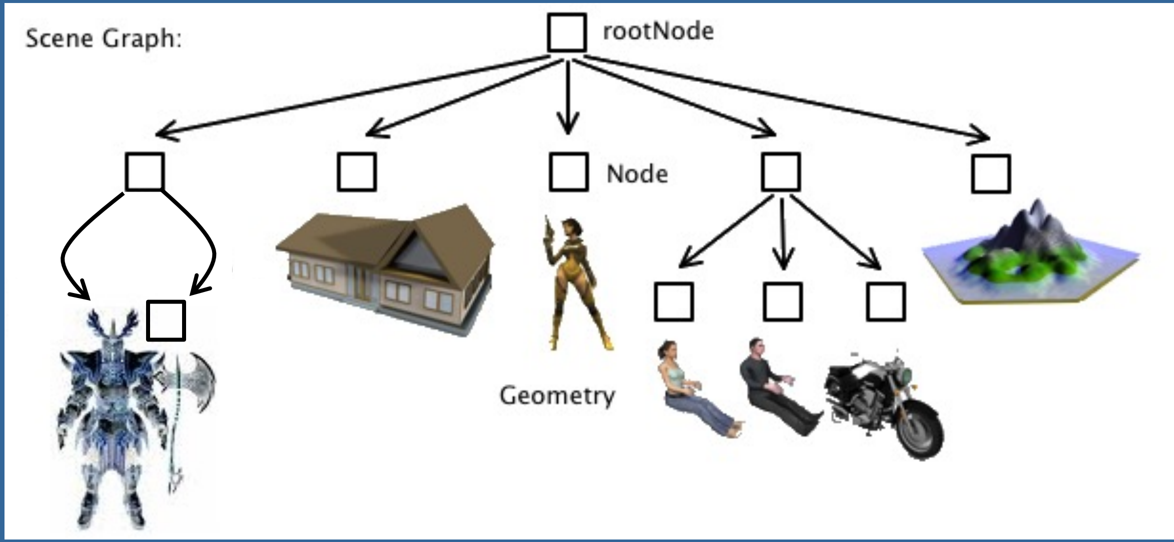
Visualizing Identical Subtrees



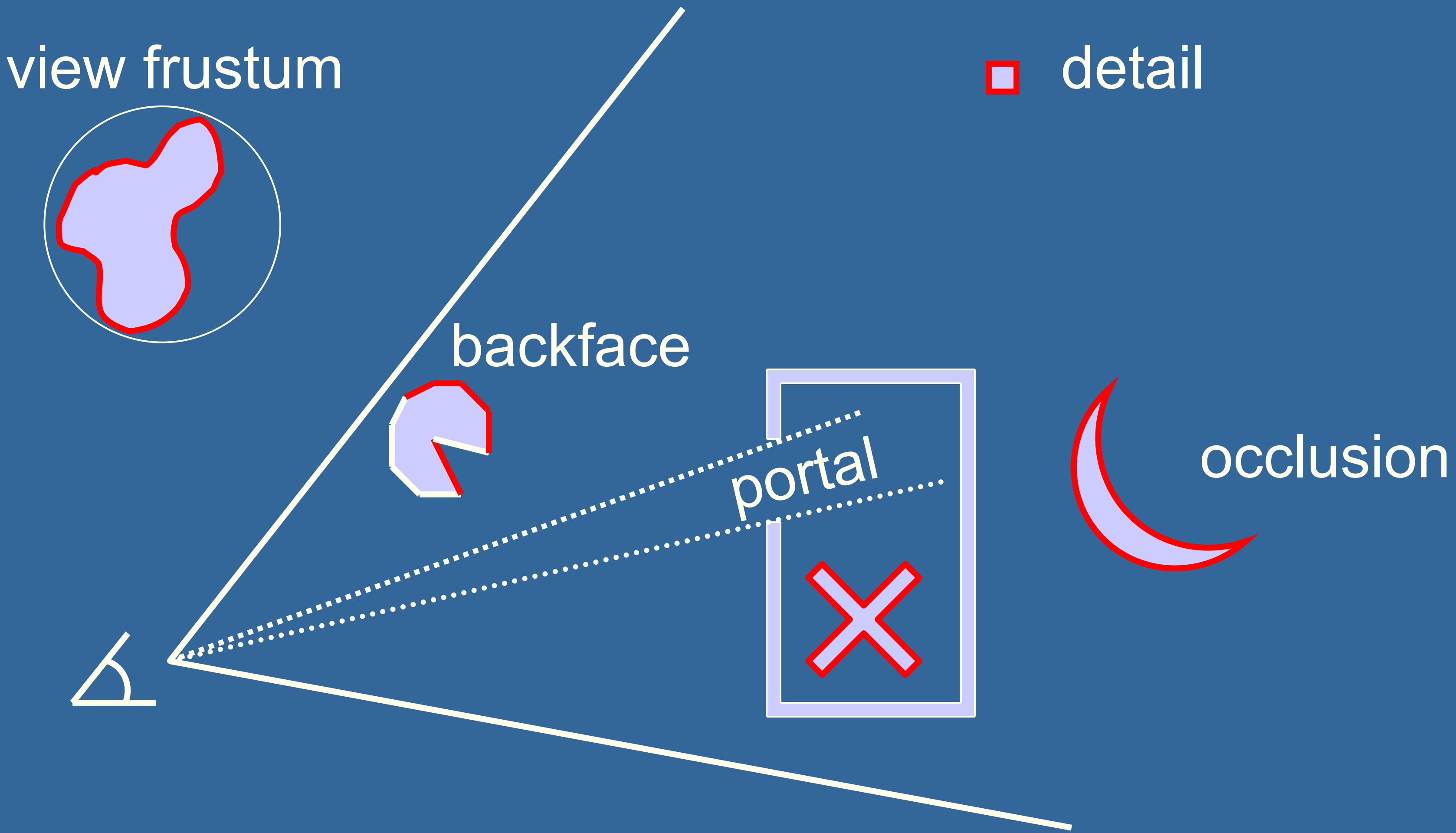
A Scene Graph is a hierarchical scene description – more typically a **logical** hierarchy (than e.g. **spatial)**

Scene graphs
– a node hierarchy

- 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

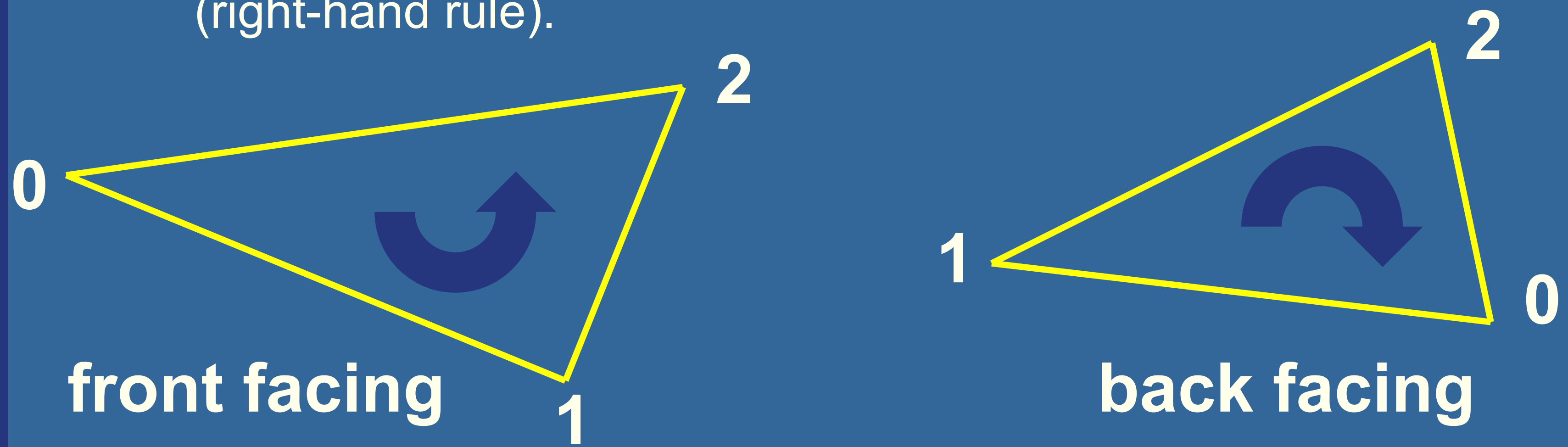


Different culling techniques
(red objects are skipped)



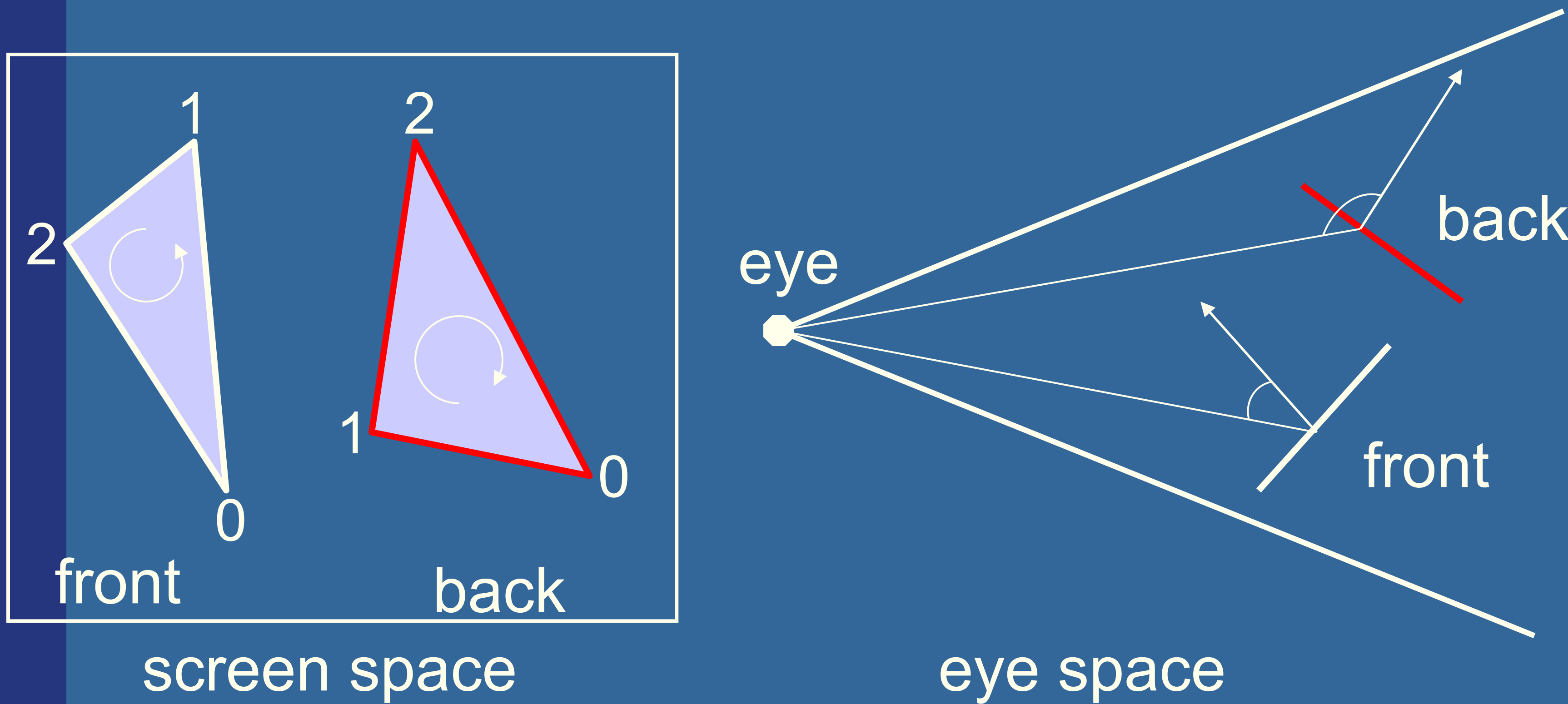
Backface culling

- Can be used when back-faces are never seen (closed objects)
- OpenGL:
  - `glCullFace(GL_BACK);`
  - `glEnable(GL_CULL_FACE);`
- First, define front/back-faces
  - Let counterclockwise vertex-winding order define front face (right-hand rule).



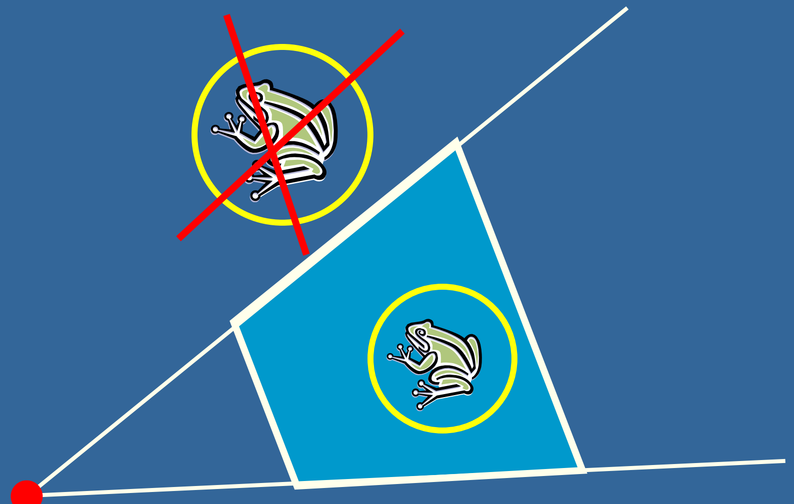
How to cull backfaces

- Two ways in different spaces:

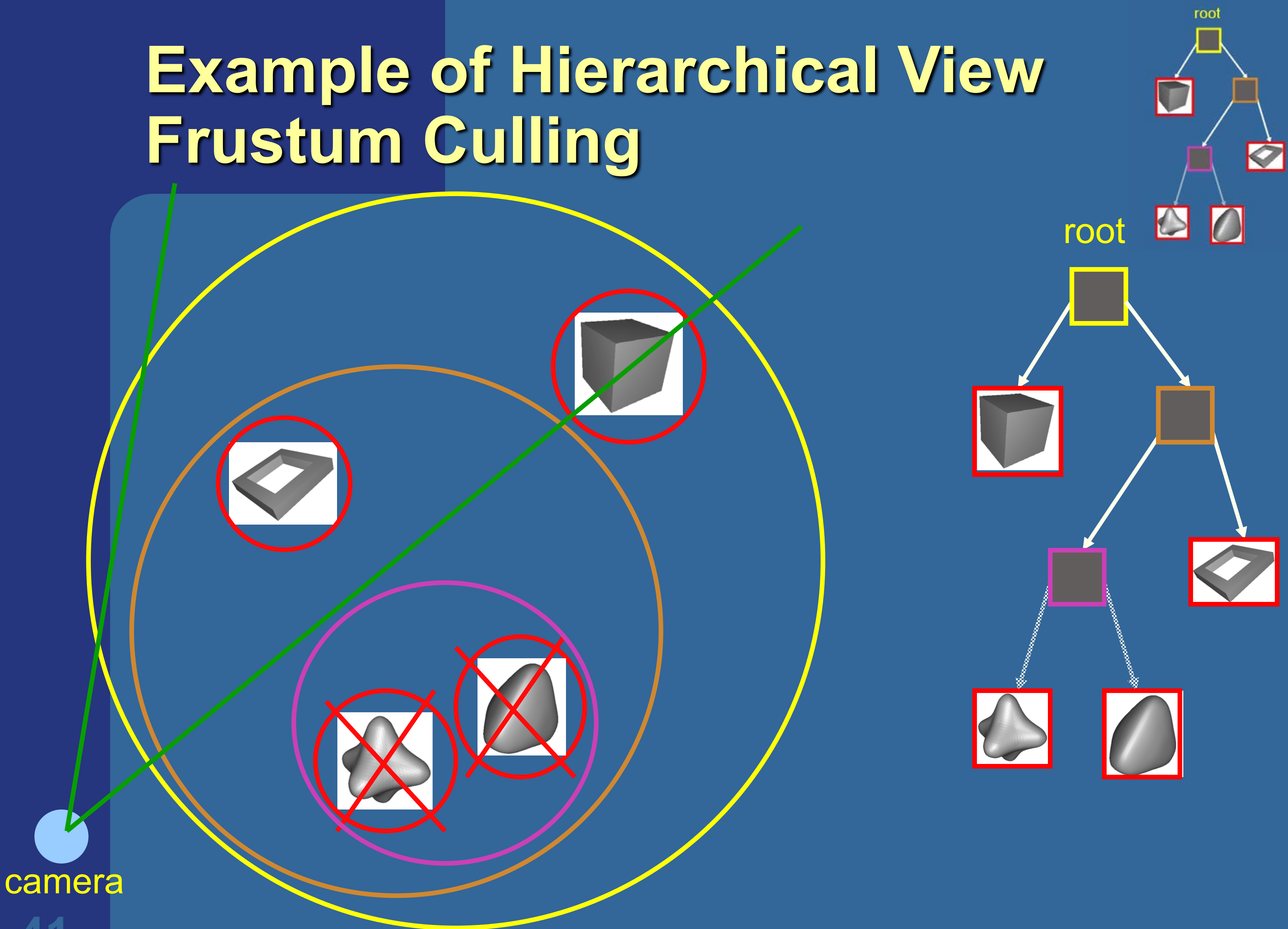


View-Frustum Culling

- 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)



Example of Hierarchical View Frustum Culling



Portal Culling

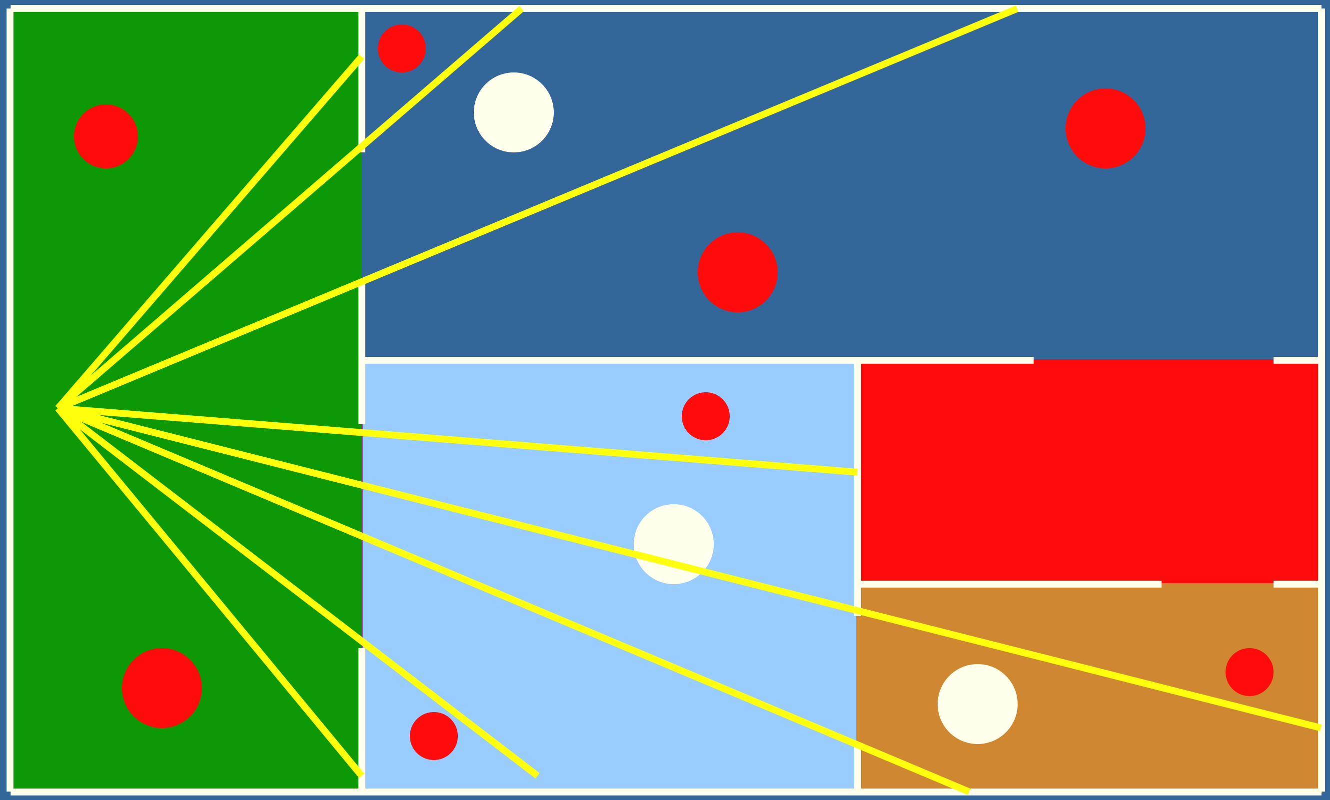
Refined view frustum culling: frustum gets smaller for each door

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)

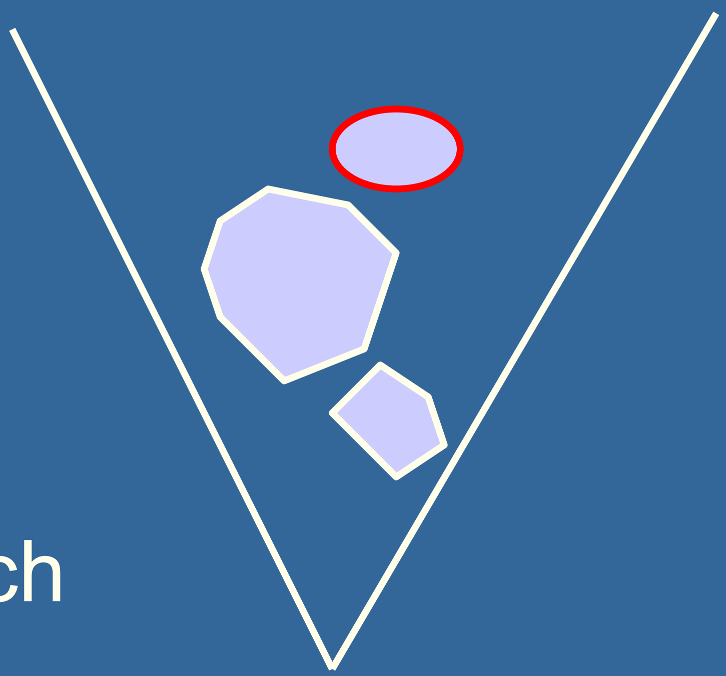
- **”Recursively do VFC through visible portals (i.e., doors & mirrors)”**

Algorithm:

- Build a graph of the scene with cells (rooms) and portals (doors/mirrors)
- 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 *

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”



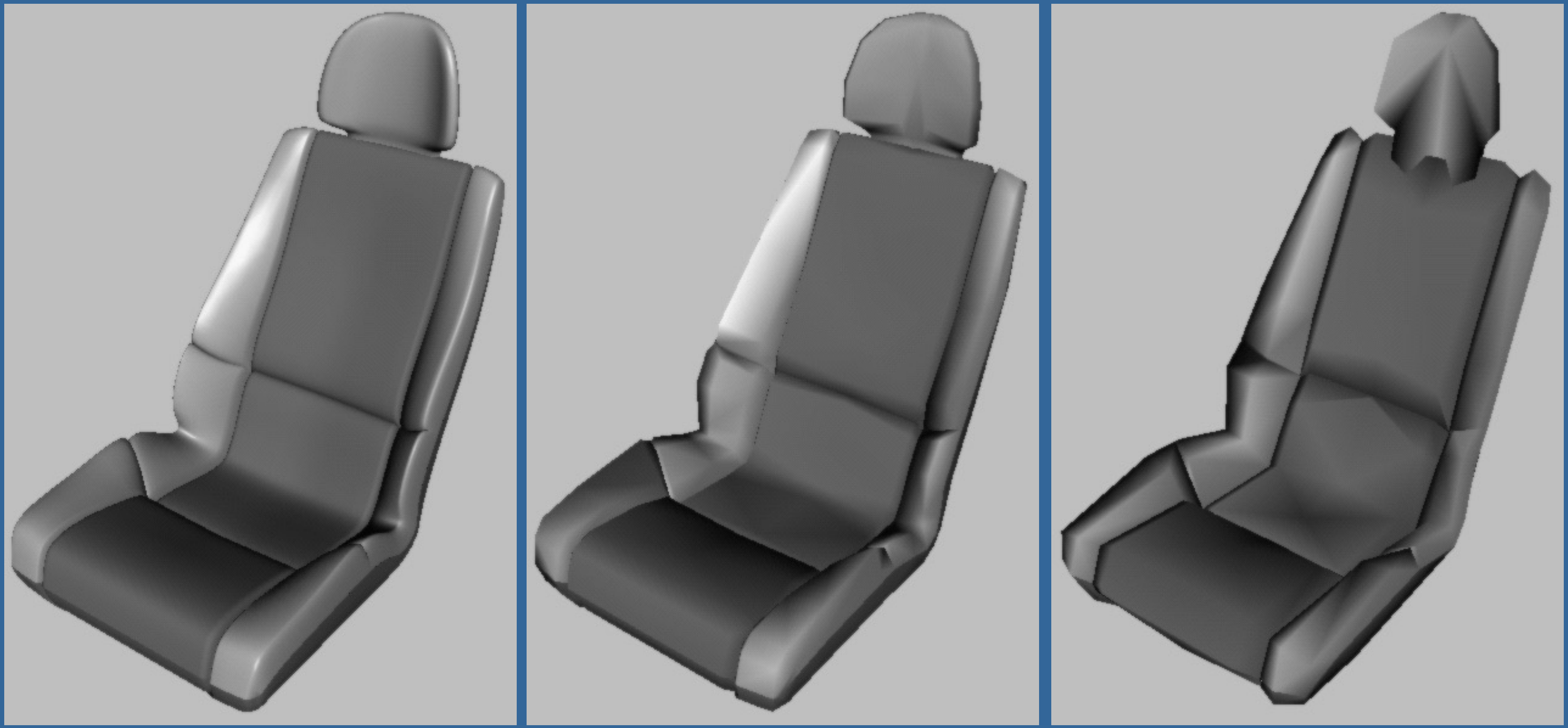
Occlusion culling algorithm

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;
```

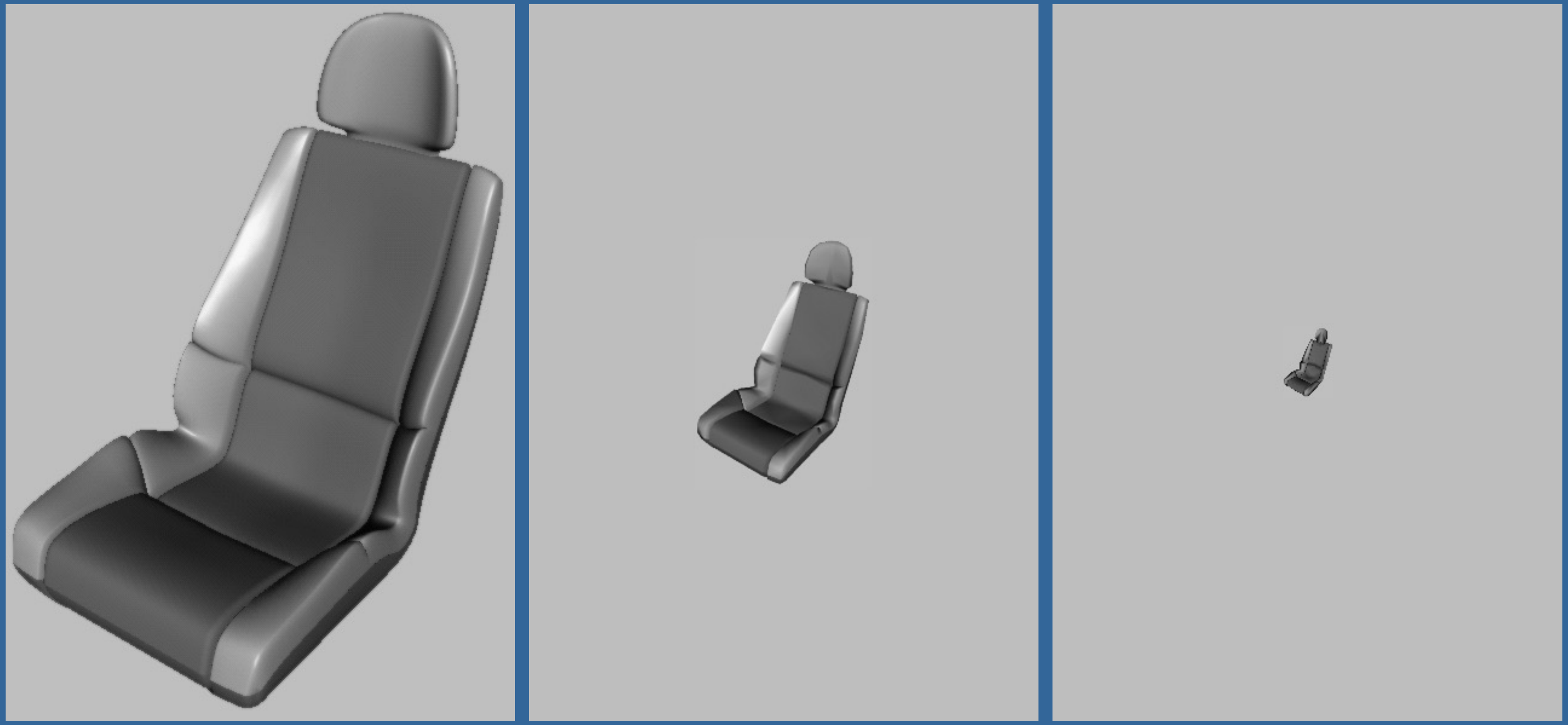
Level-of-Detail Rendering

- 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

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(BVHnode* node)

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