

# Tehnike pospeševanja upodabljanja v realnem času




# Motivation

- Graphics hardware 2x faster in 6 months!
- Wait... then it will be fast enough!
- NOT!
- We will never be satisfied
  - Screen resolution: 2000x1000
  - Realism: global illumination
  - Geometrical complexity: no upper limit!

Kako v realnem času upodobiti tako kompleksno pokrajino?



# Big Models: Plant Ecosystem Simulation

A detailed 3D simulation of a lush green meadow. The foreground is filled with various types of grasses, including tall blades and clumps of shorter grass. Scattered throughout the meadow are numerous yellow wildflowers and several white dandelion seed heads. In the middle ground, there are several large, leafy green plants. The background features a line of trees with dense green foliage under a bright blue sky with soft, white clouds. The overall scene is vibrant and detailed, showcasing the complexity of the plant ecosystem simulation.

16.7 million polygons (sort of)

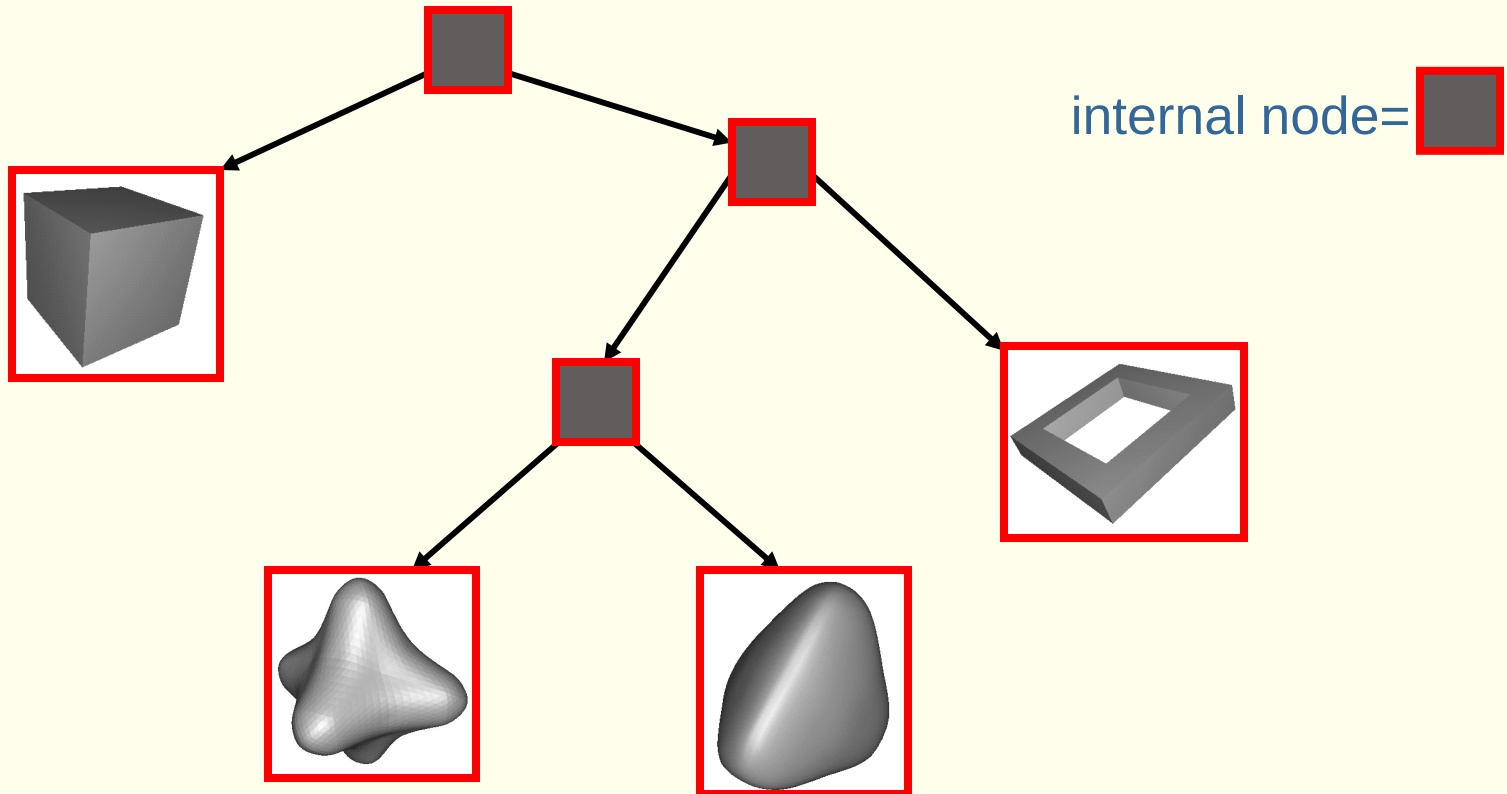
Deussen et al: *Realistic Modeling of Plant Ecosystems*

# Overview

- Graf scene
- Metode izločanja (culling techniques)
- Nivoji podrobnosti (LODs)
- Plakati (billboards)
- Kombiniranje metod
- Odkrivanje trkov

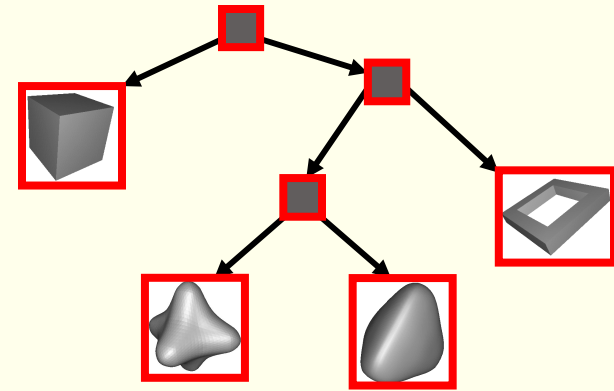
# Graf scene

- DAG – directed acyclic graph
  - Simply an  $n$ -ary tree without loops
- Uses: collision detection and faster rendering



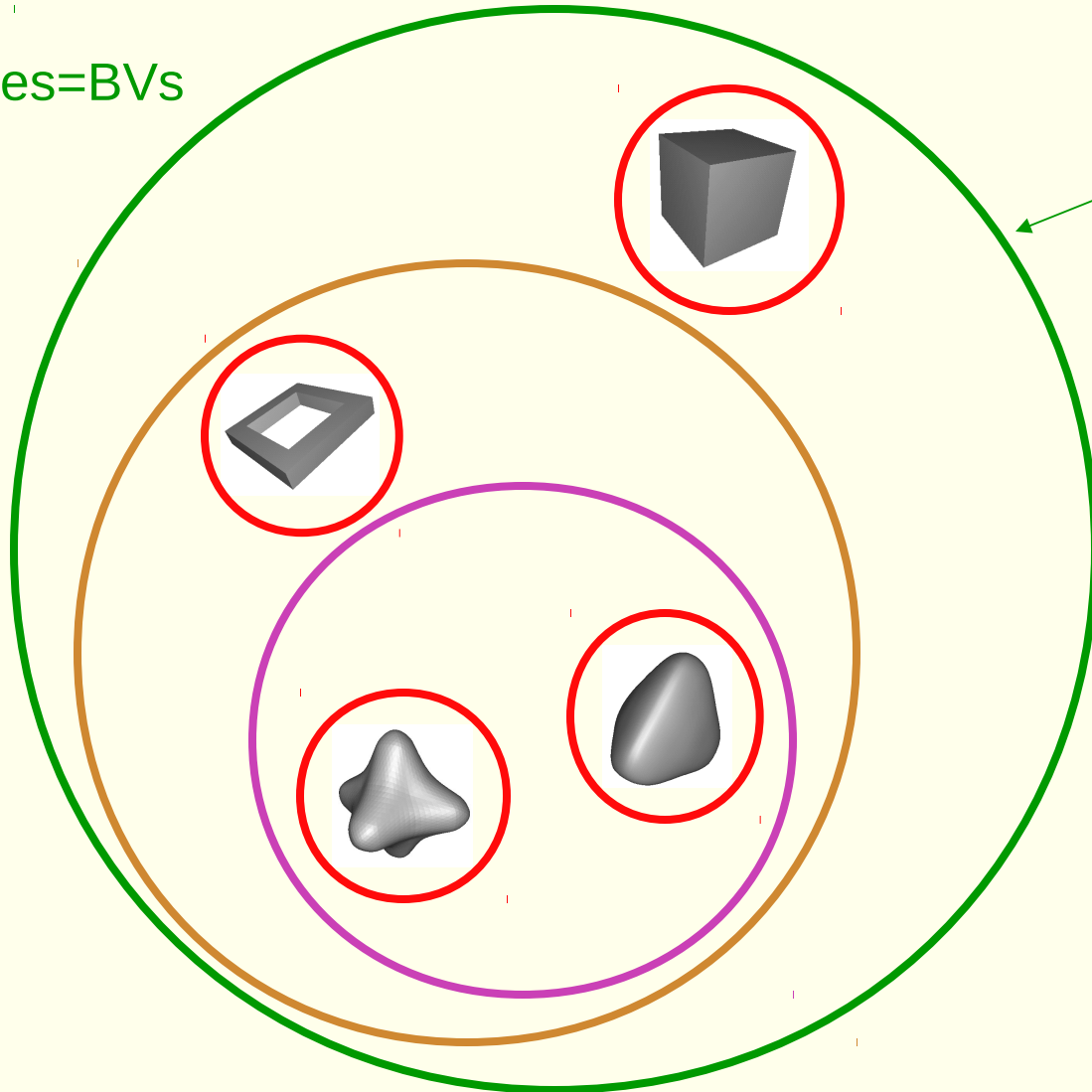
# Vseбина grafa scene

- Leaves contains geometry
- Each node holds a
  - Bounding Volume (BV)
  - pointers to children
  - possibly a transform
- Examples of BVs: spheres, boxes
- The BV in a node contains all geometry in its subtree

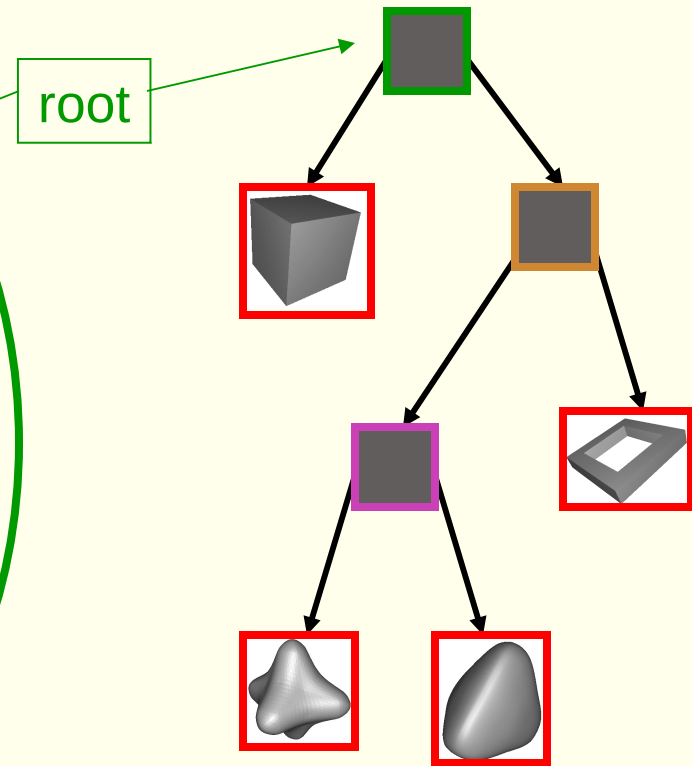


# Primer grafa scene

circles=BVs



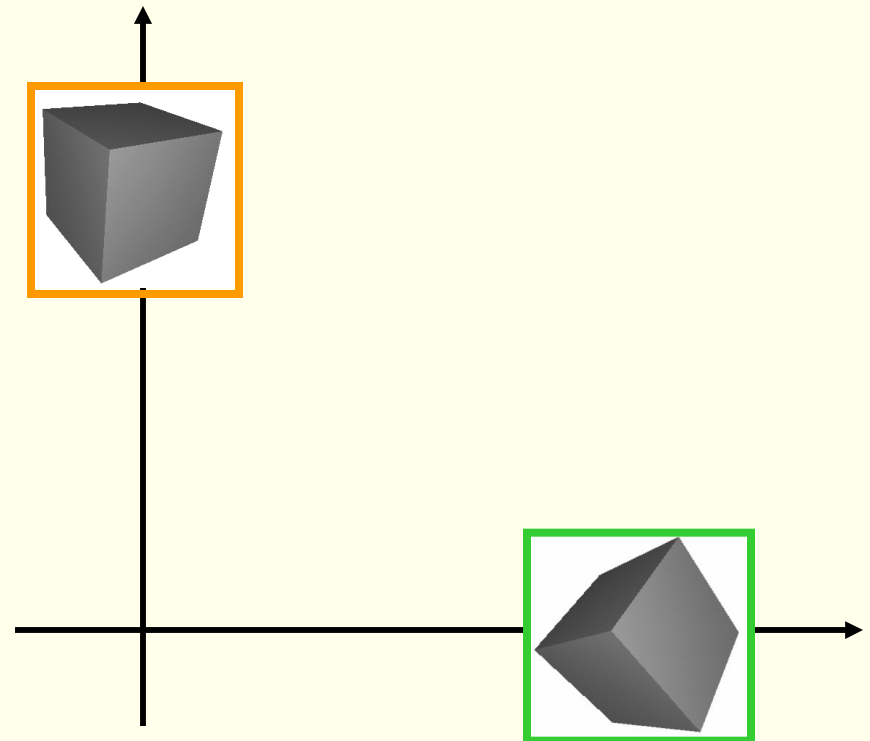
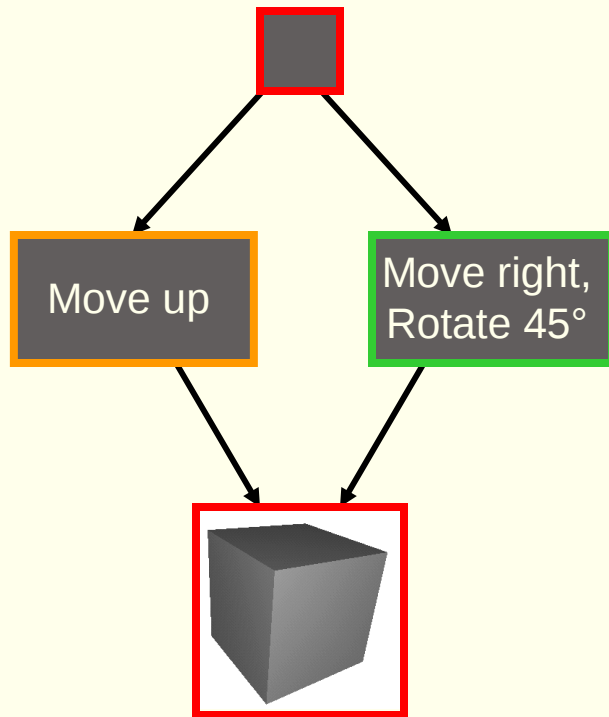
scene graph





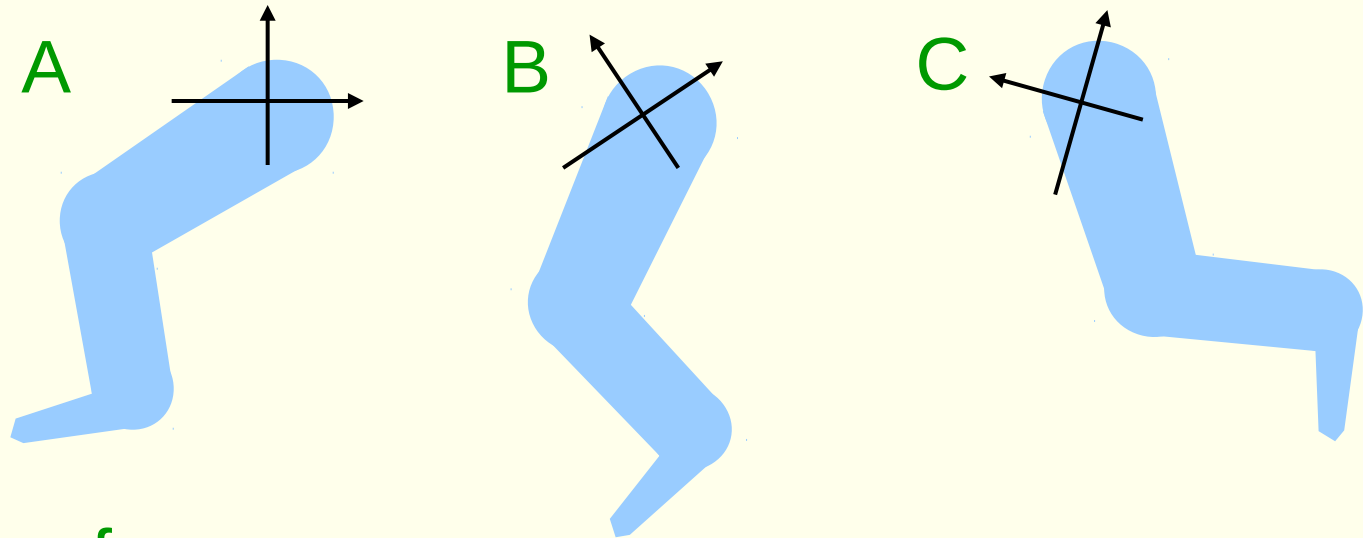
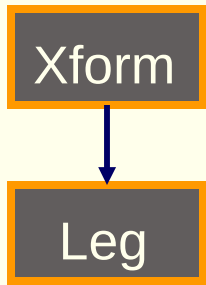
# Transformi v grafu scene

- Put a transform in each internal node
- Gives instancing, and hierarchical animation

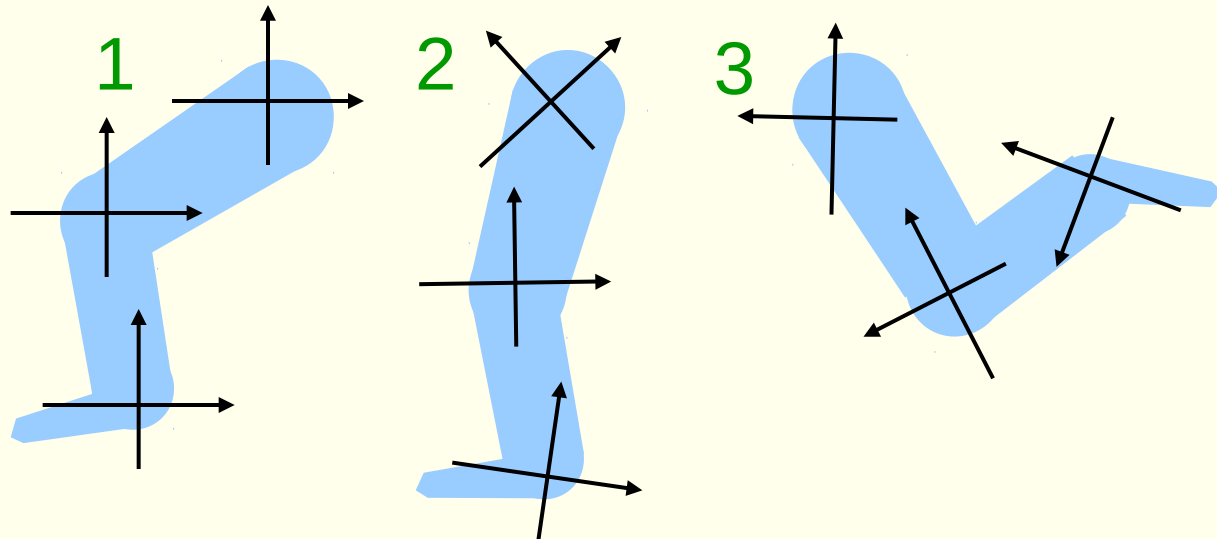
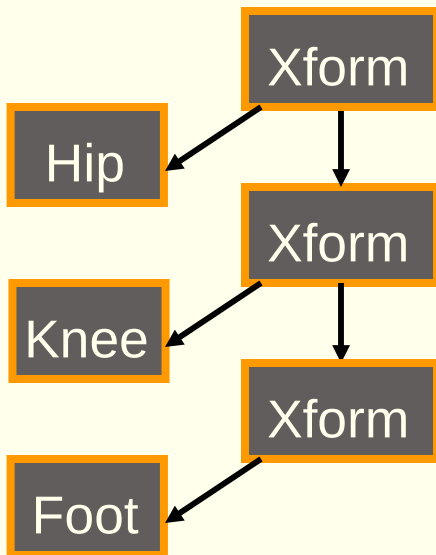


# Primer: noga tekača

No hierarchy:  
one transform



Hierarchy: 3 transforms



# Tehnike izločanja

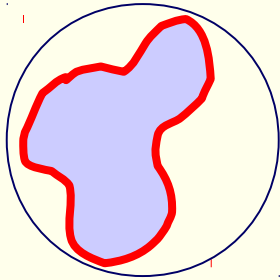
- “To cull” means “to select from group”
- In graphics context: do not process data that will not contribute to the final image
- The “group” is the entire scene, and the selection is a subset of the scene that we do not consider to contribute

# Izločanje: pregled

- Backface culling
- Hierarchical view-frustum culling
- Portal culling
- Detail culling
- Occlusion culling

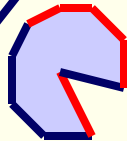
# Primeri izločanja

view frustum

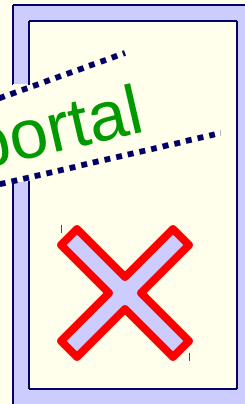


□ detail

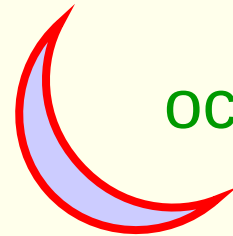
backface



portal



occlusion

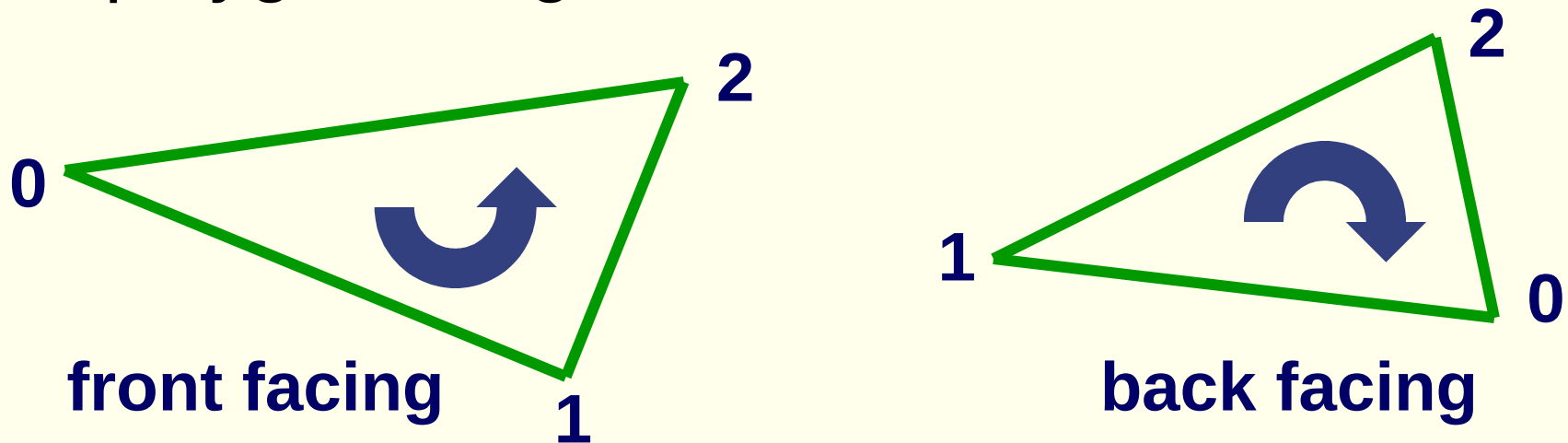


# Izločanje zadnjih strani

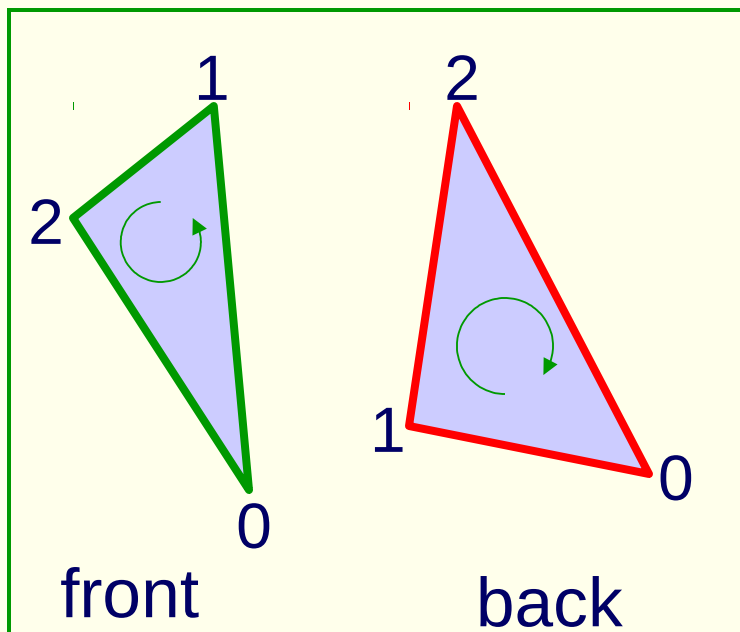
- Simple technique to discard polygons that faces away from the viewer
- Can be used for:
  - closed surface (example: sphere)
  - or whenever we know that the backfaces never should be seen (example: walls in a room)
- Two methods (screen space, eye space)
- Which stages benefits? Rasterizer, but also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

# Izločanje zadnjih strani

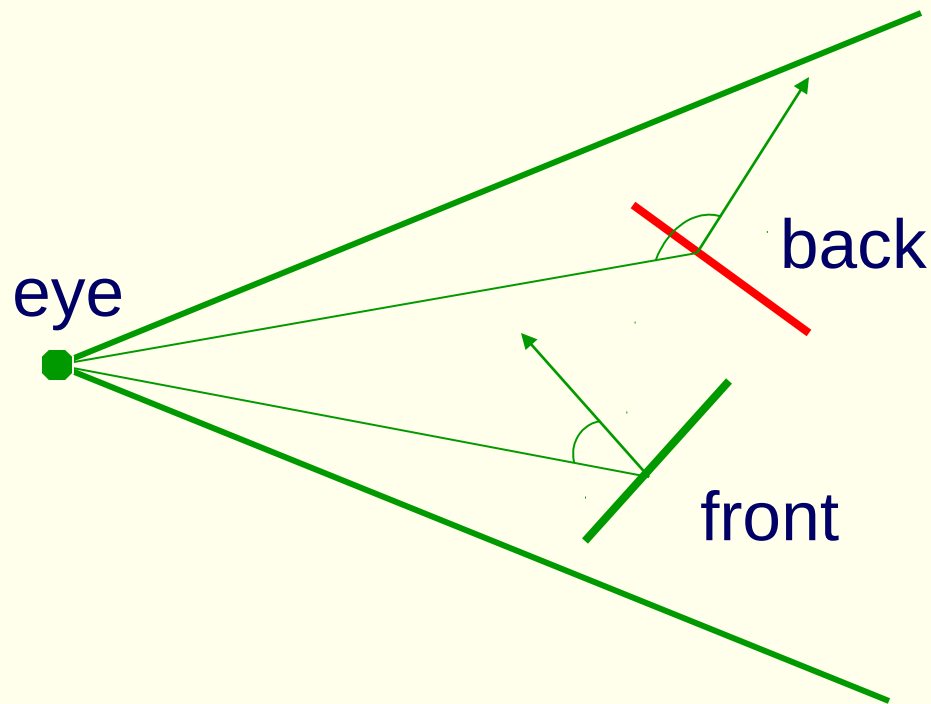
- Often implemented for you in the API
- OpenGL: **`glCullFace(GL_BACK);`**
- How to determine what faces away?
- First, must have consistently oriented polygons, e.g., counterclockwise



# Kako izločamo zadnje strani



screen space



eye space



# Izločanje zakritih ploskev

- How do you know which objects are visible to camera before you so they are not transformed or rendered?
- Commonly used techniques
  - Back-face removal
  - Bounding spheres test

# Izločanje zadnjih strani - 1

- Step 1:
  - Remove all polygons outside of viewing frustum
- Step 2:
  - If the dot product of the view vector and the surface normal is  $> 0$ , it is facing the viewer.
  - Remove all polygons that are facing away from the viewer.

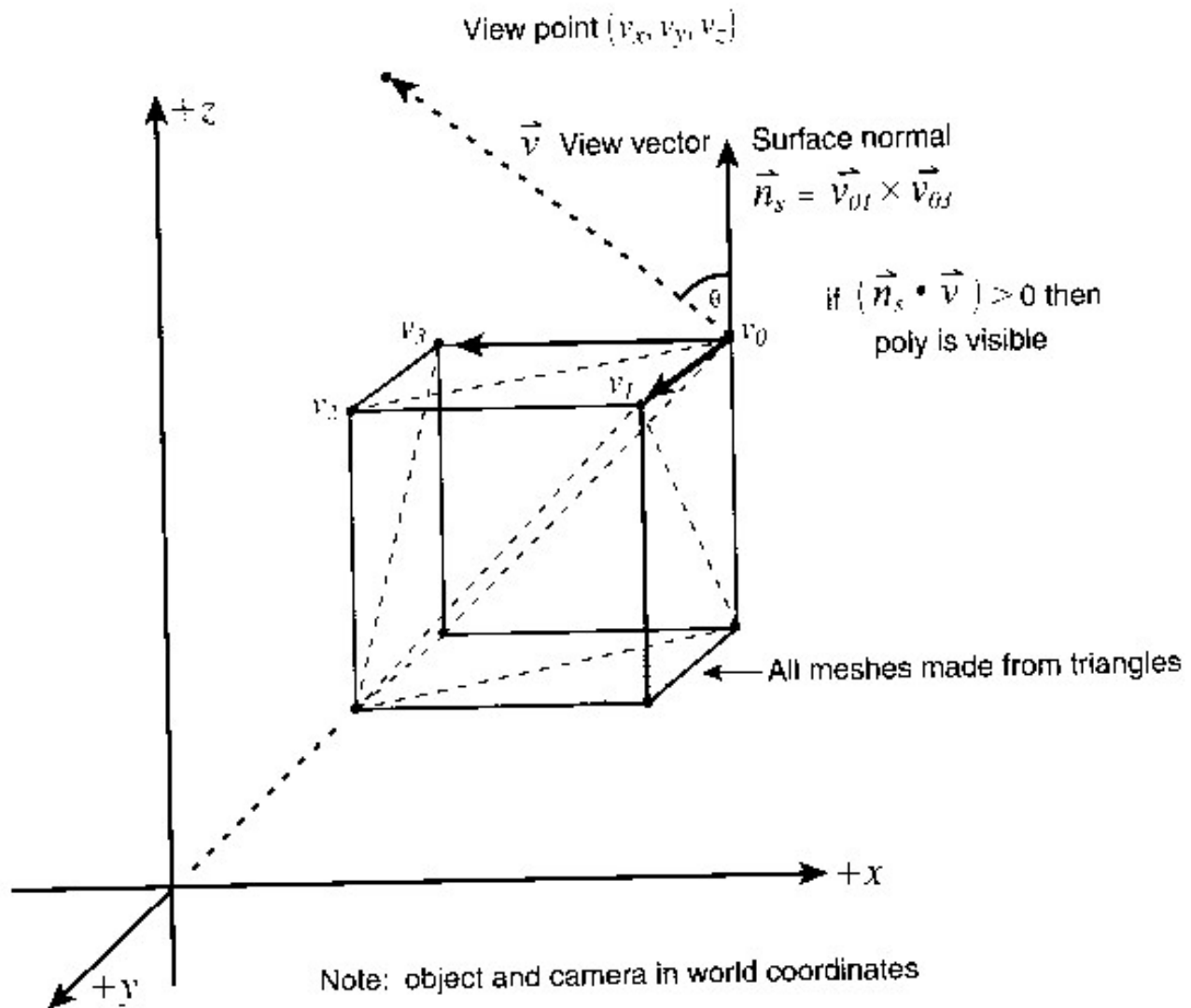
# Izločanje zadnjih strani - 2

## • Step 3:

- Draw the visible faces in an order so the object looks right.

## • Note:

- Surface normal = cross product of two co-planar edges.
- View vector from normal point to viewpoint



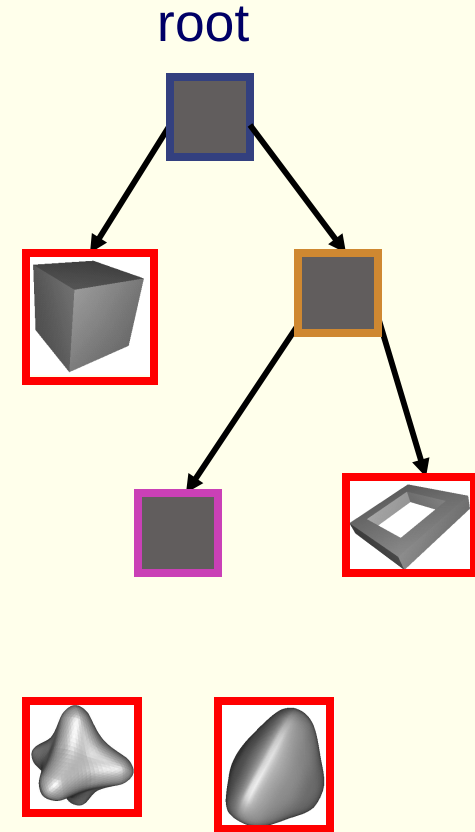
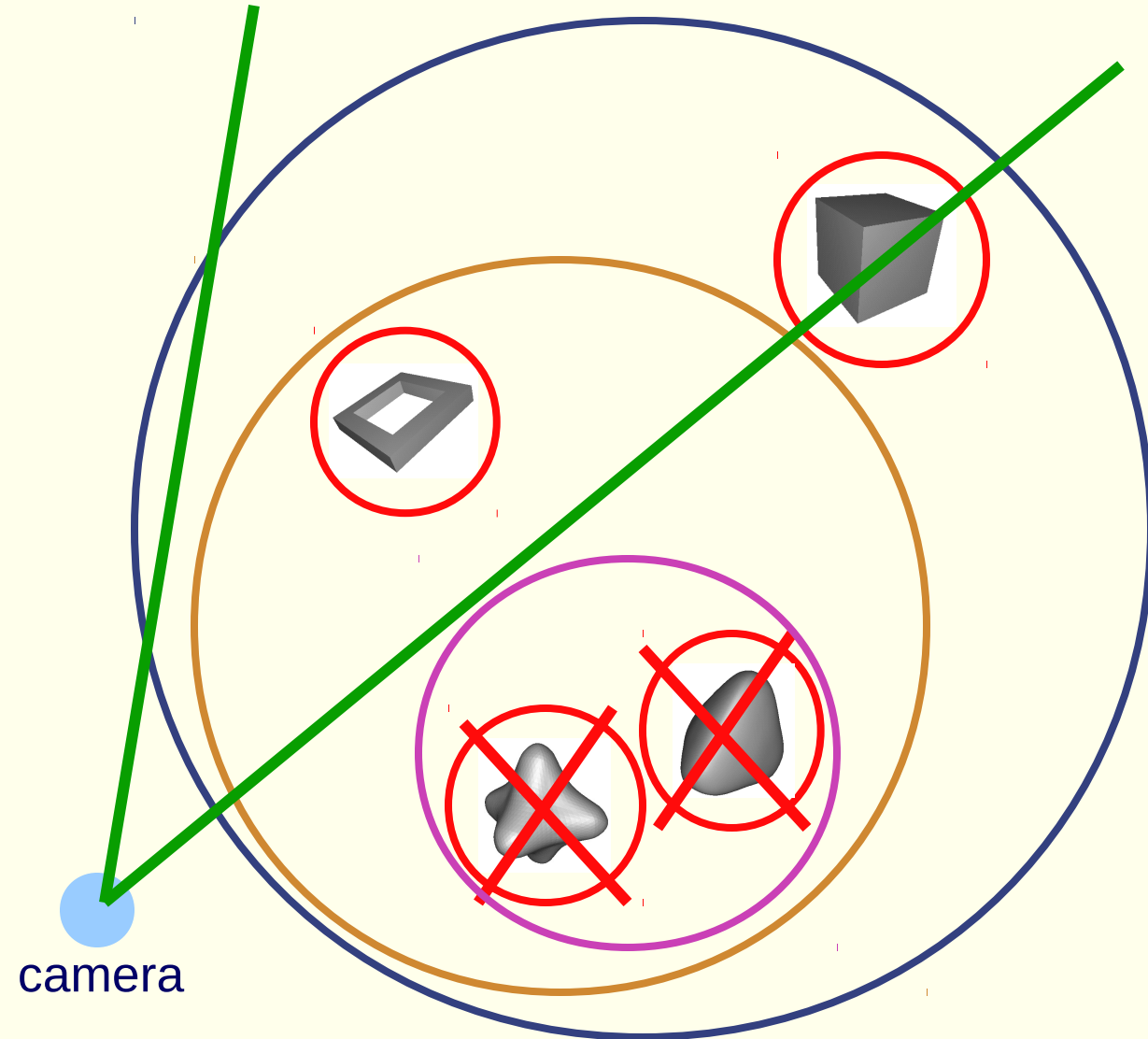
# View-Frustum Culling

- 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)
- Avoid further processing of such BV’s and their containing geometry

# Can we accelerate VF culling further?

- Do what we always do in graphics...
- Use a hierarchical approach, e.g, the scene graph
- Which stages benefits?
  - Geometry and Rasterizer
  - Bus between CPU and Geometry

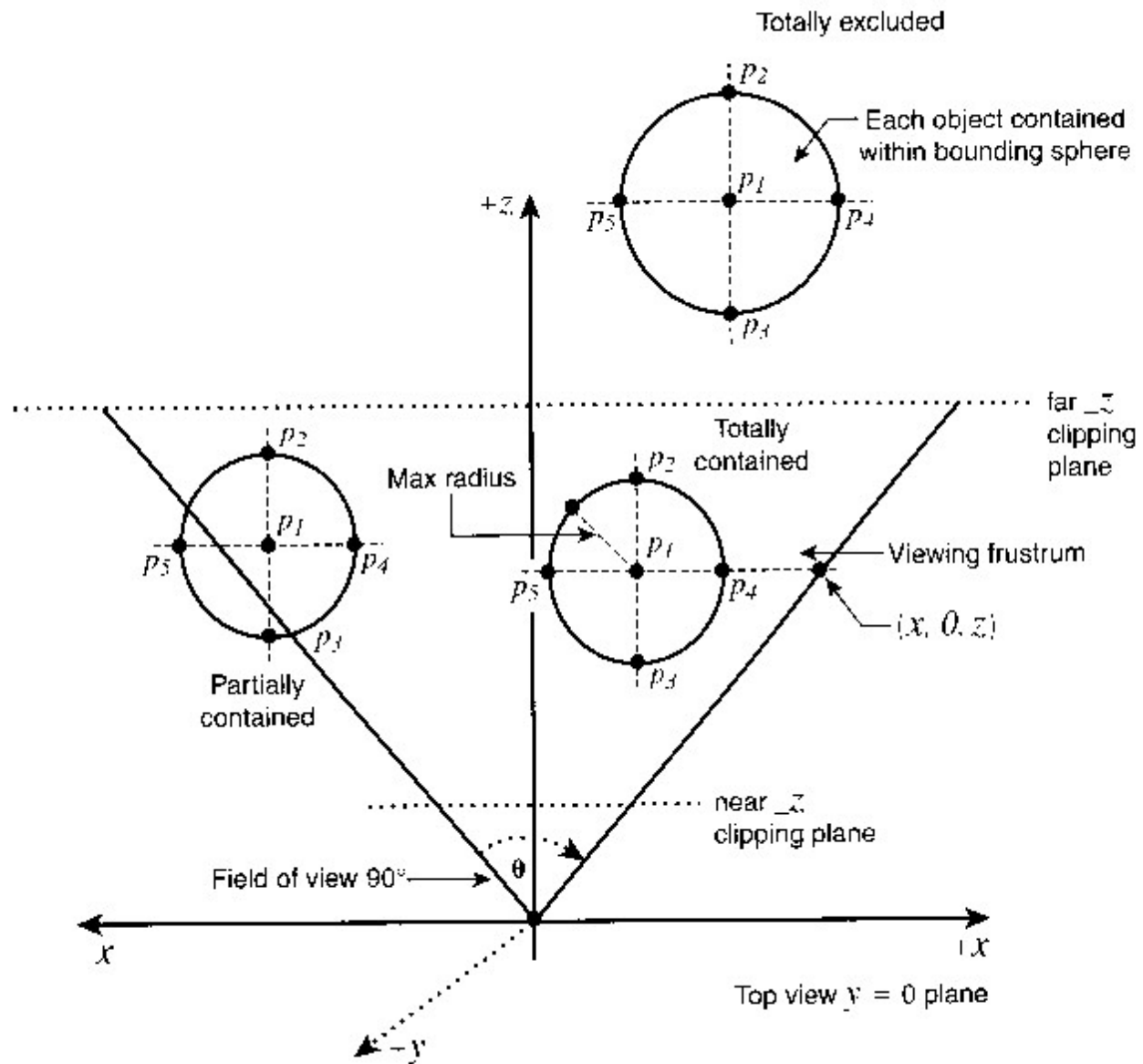
# Example of Hierarchical View Frustum Culling



# Test obsegajoče krogle –1

- Can be used **before** back-face removal to discard entire objects all at once
- This test only works if you have object partitioned in convex areas, won't work for a game world composed of a single mesh
- The idea is to create a bounding sphere around each object and then transform  $T_{wc}$  the center of each sphere and see if the entire sphere is outside the viewing frustum





# Test obsegajoče krogle – 2

- If none of the points p1-p5 is in the viewing frustum the object can not be visible, the general test for point p(x,y,z) is outside

```
if ((z > far_z) || (z < near_z) || // z-axis
    (fabs(x) < z) || //x-z plane
    (fabs(y) < z)) //y-z plane
{
    // point not in viewing frustum
}
```

# Test obsegajoče krogle – 3

- It is important to note that just because a portion of the bounding sphere is inside the viewing frustum there is no guarantee that any portion of the object is visible (unless the object is a sphere that fill the entire bounding sphere exactly)
- Might be better to use a bounding cube or parallelepiped depending on the shape of the object

# Cells & Portals

- Goal: walk through architectural models (buildings, cities, catacombs)
- These divide naturally into *cells*
  - Rooms, alcoves, corridors...
- Transparent *portals* connect cells
  - Doorways, entrances, windows...
- Notice: cells only see other cells through portals

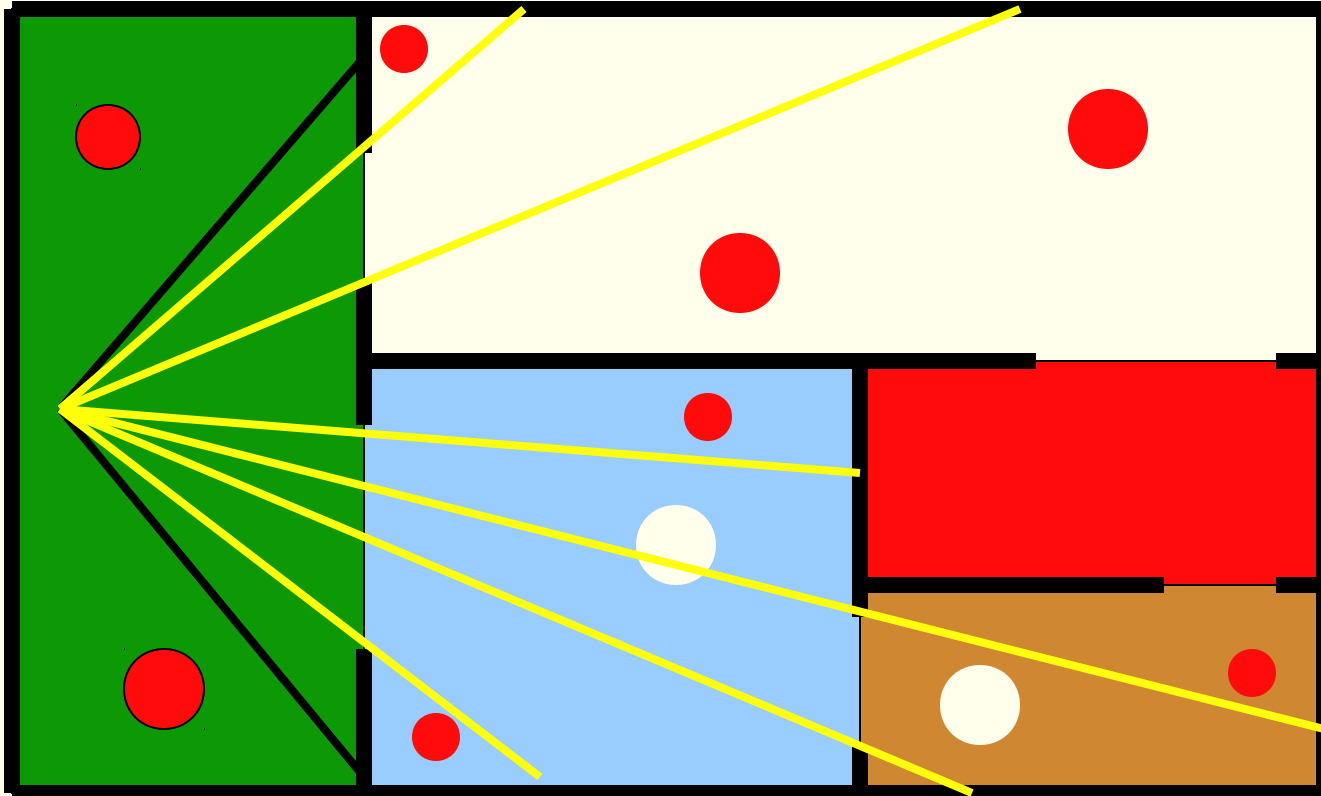
# Portal Culling



- 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

- 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 VF cull w.r.t. AABB
  - Traverse to closest cells (through portals)
  - Intersection of AABB & AABB of traversed portal
  - Goto \*

# Tipi geometrije

- Points
- Lines, Rays and Line Segments
- Spheres, Cylinders and Cones
- Cubes, rectilinear boxes - Axis aligned or arbitrarily aligned
  - **AABB: Axis aligned bounding box**
  - **OBB: Oriented bounding box**
- **k-dops – shapes bounded by planes at fixed orientations**
- Convex, manifold meshes – any mesh can be triangulated
  - Concave meshes can be broken into convex chunks, by hand
- Triangle soup
- More general curved surfaces, but not used (often) in games



AABB



OBB

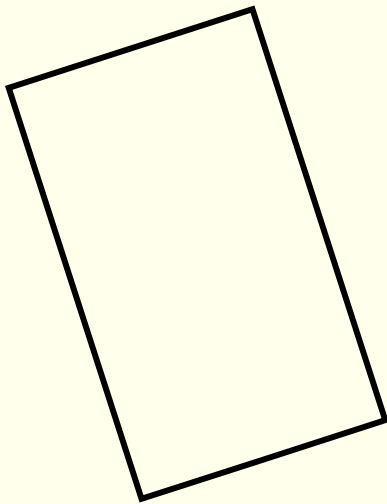


8-dop

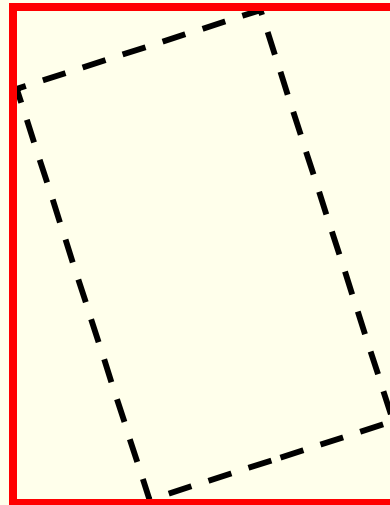


# Portal overestimation

- To simplify:



actual portal



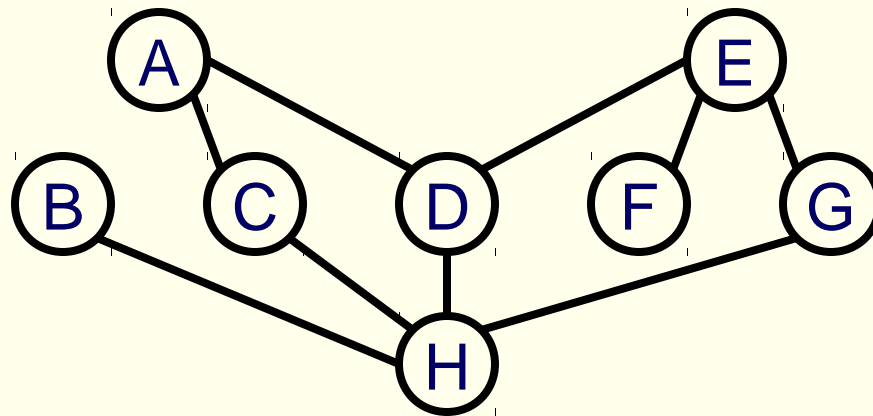
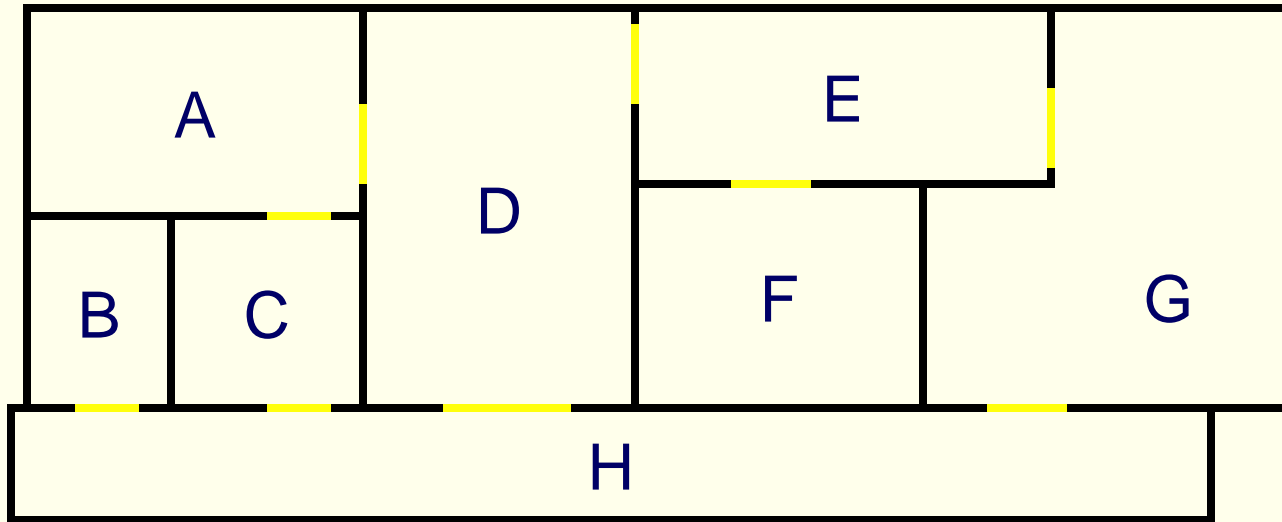
overestimated portal

# Cells & Portals

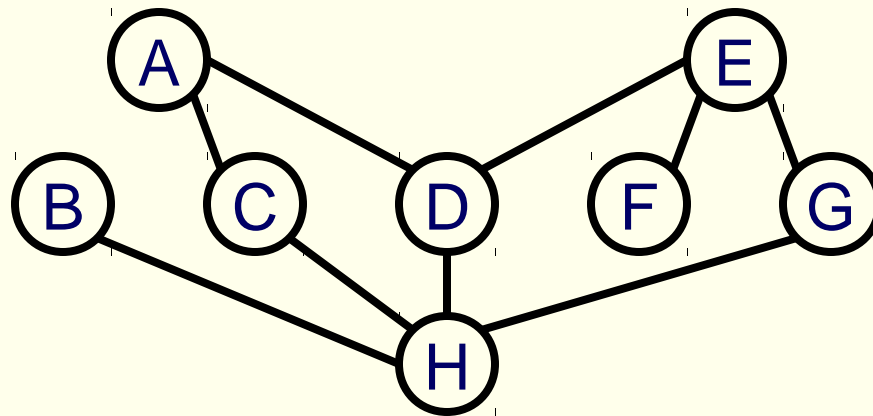
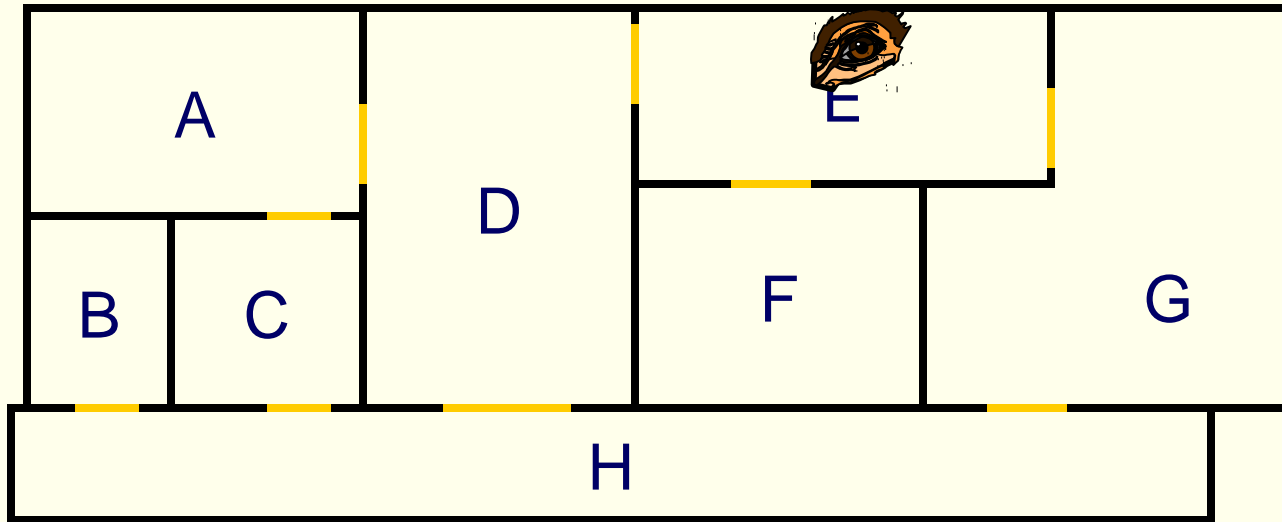
## • Idea:

- Cells form the basic unit of PVS
- Create an *adjacency graph* of cells
- Starting with cell containing eyepoint, traverse graph, rendering visible cells
- A cell is only visible if it can be seen through a sequence of portals
  - So cell visibility reduces to testing portal sequences for a *line of sight...*

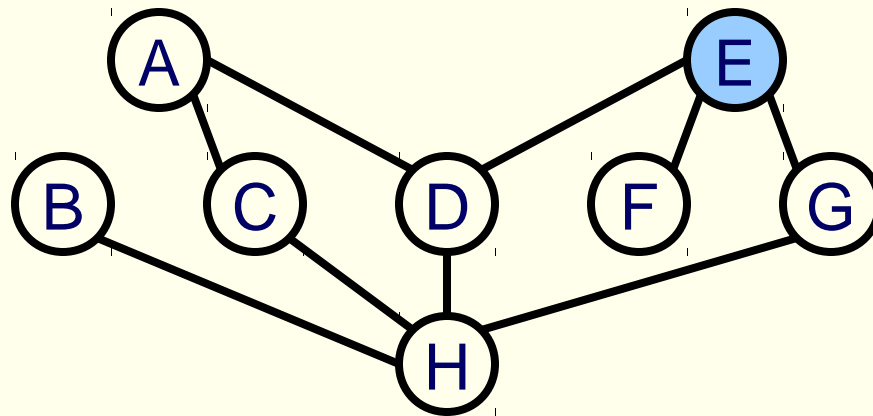
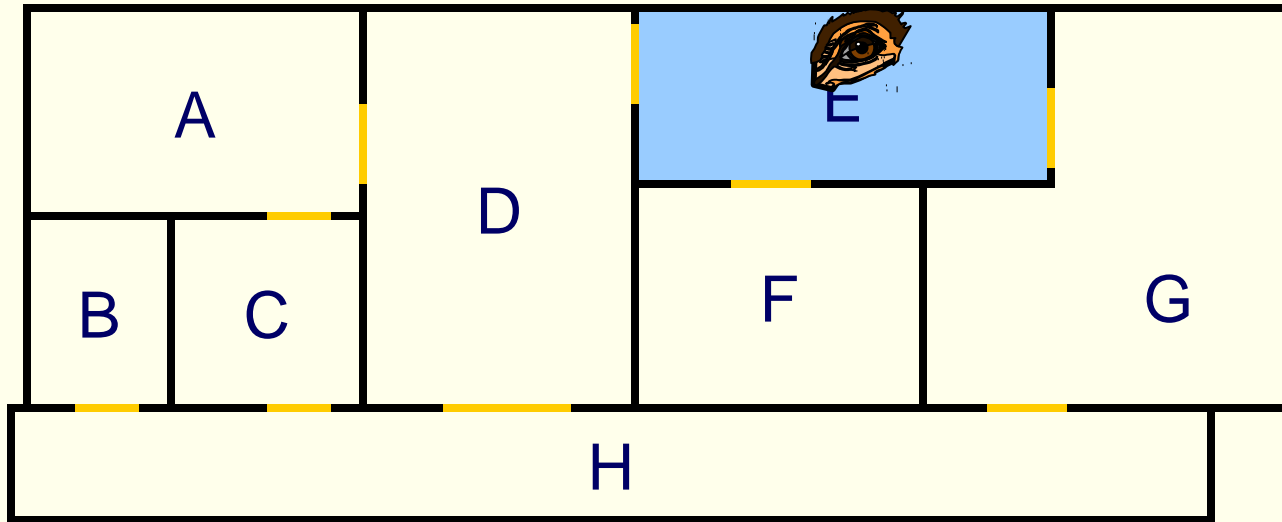
# Cells & Portals



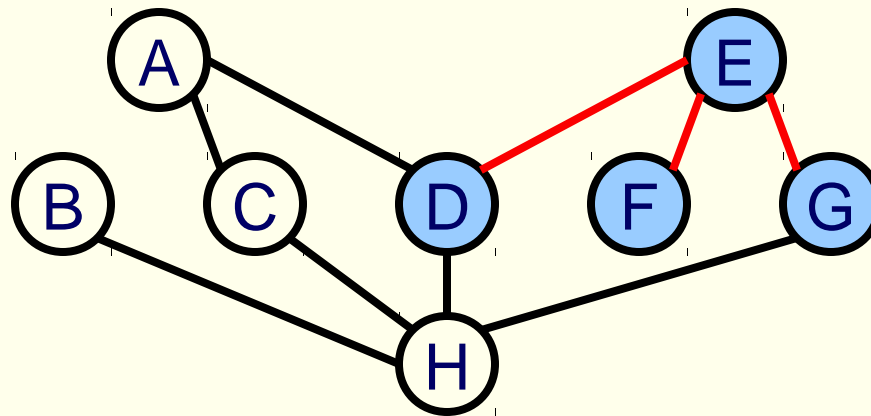
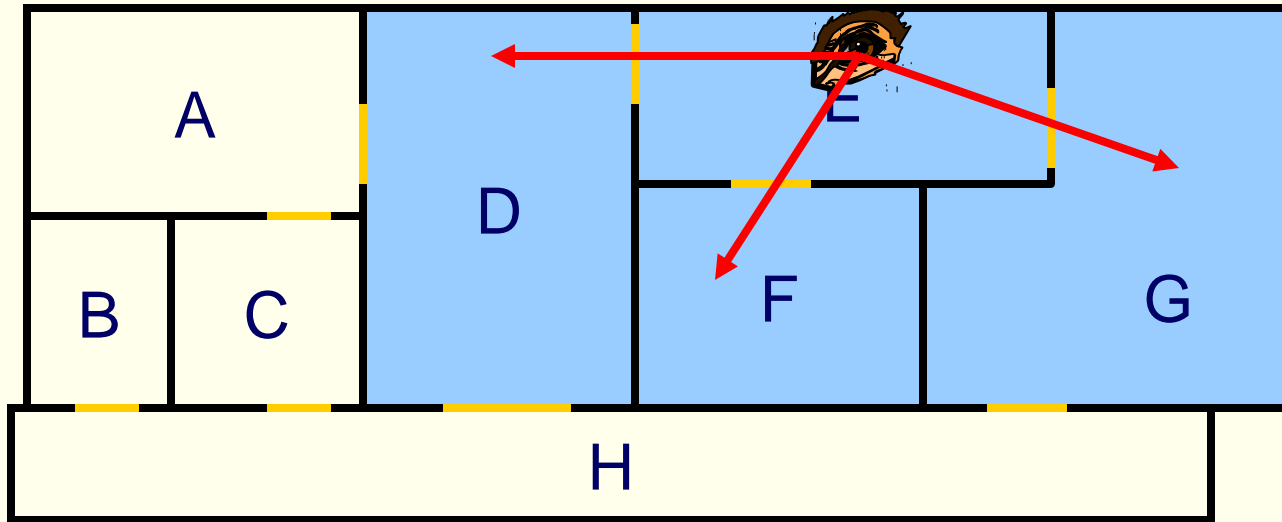
# Cells & Portals



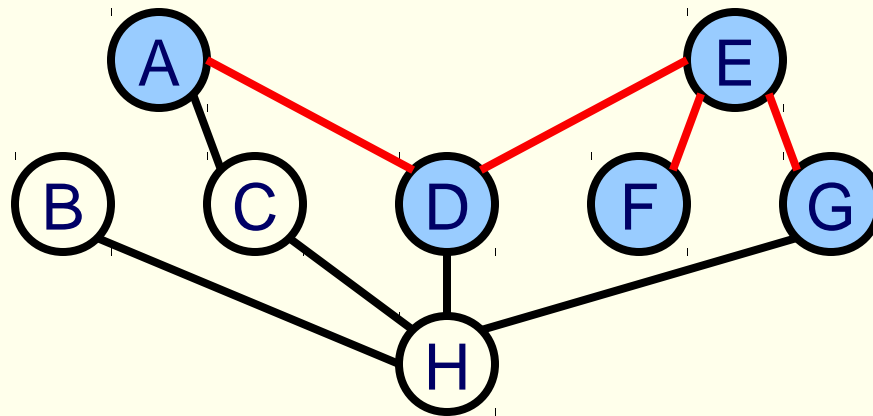
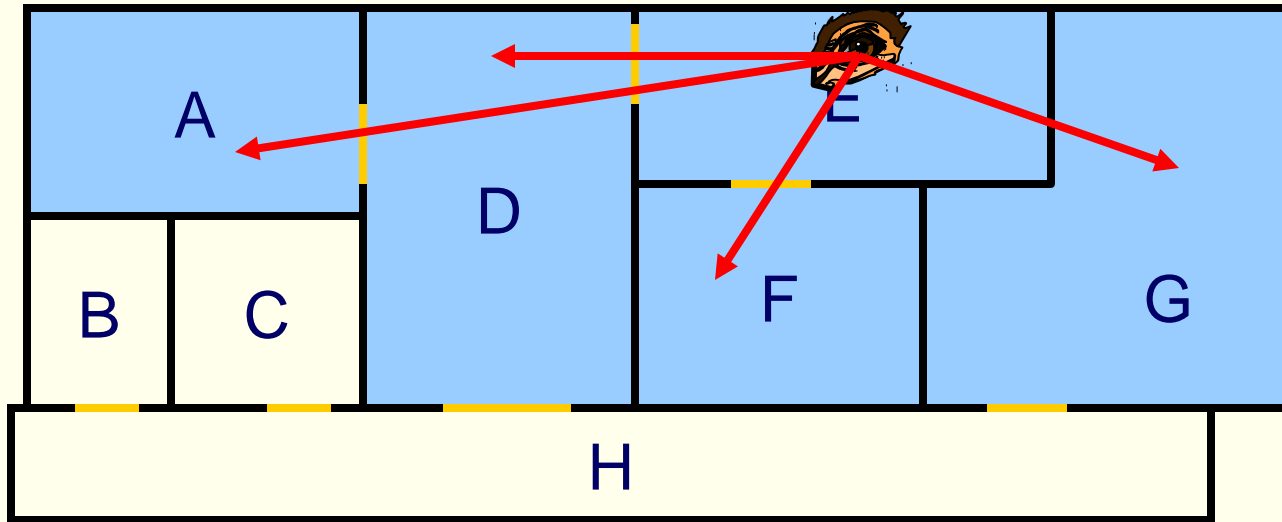
# Cells & Portals



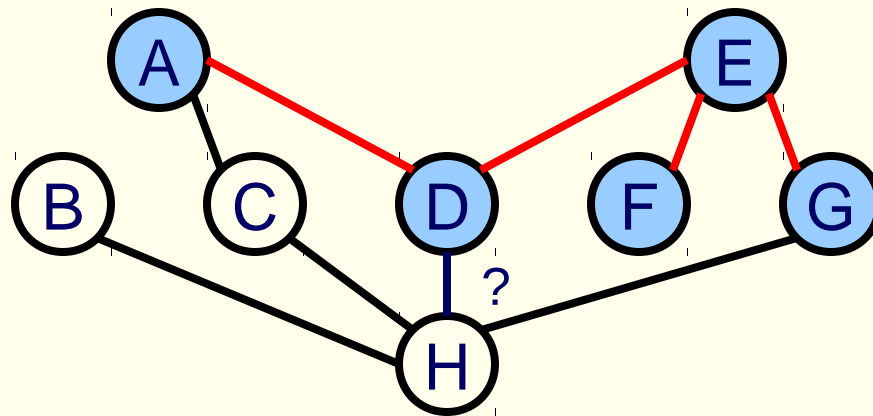
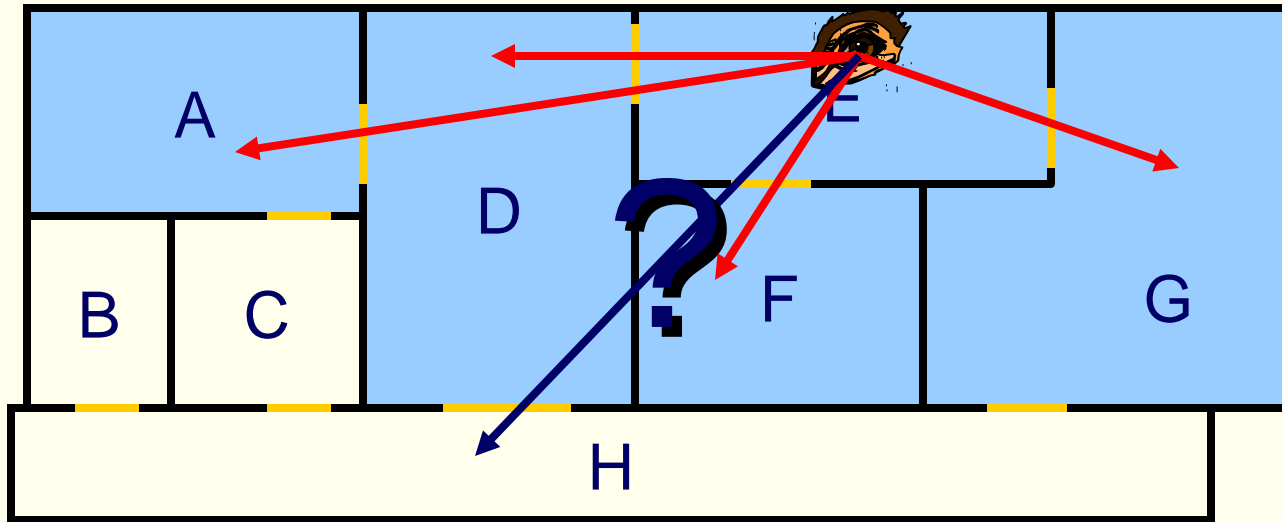
# Cells & Portals



# Cells & Portals

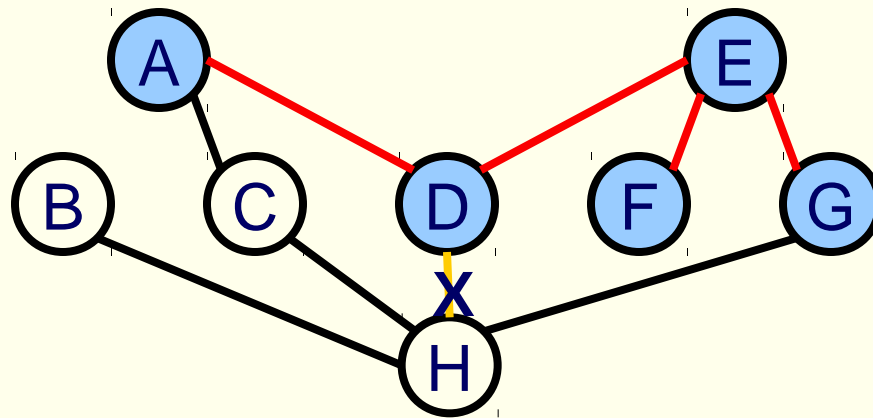
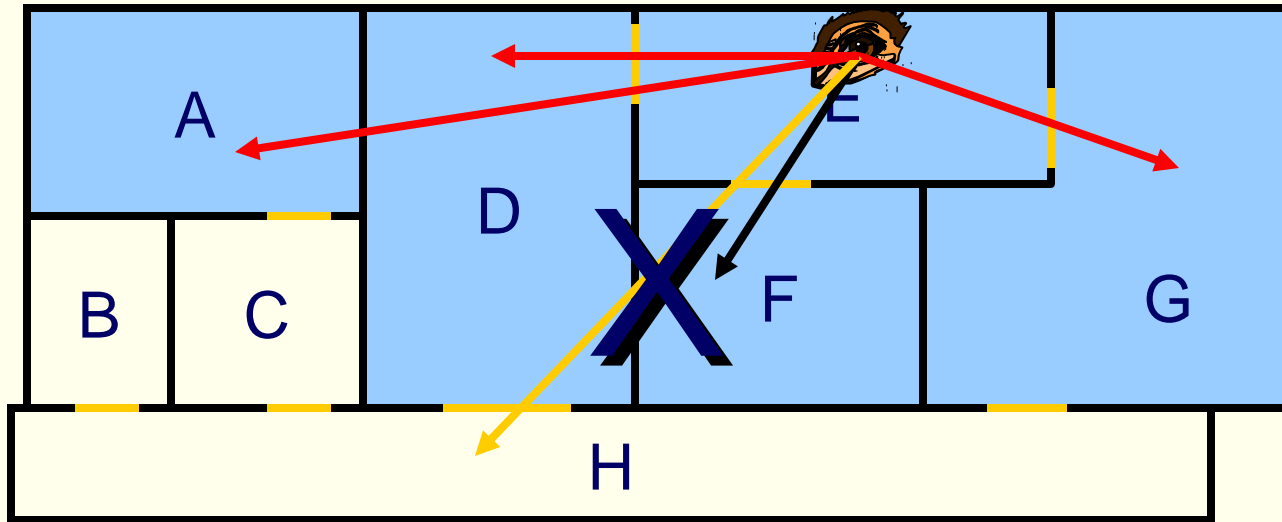


# Cells & Portals



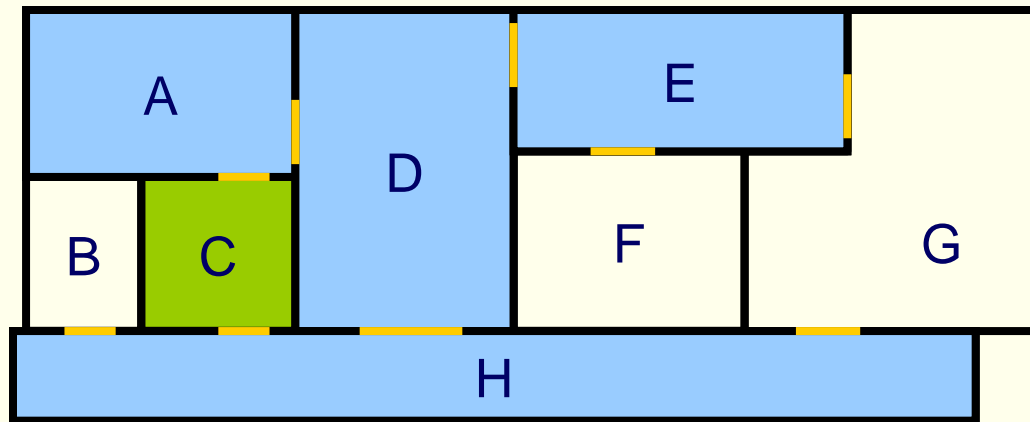


# Cells & Portals



# Cells & Portals

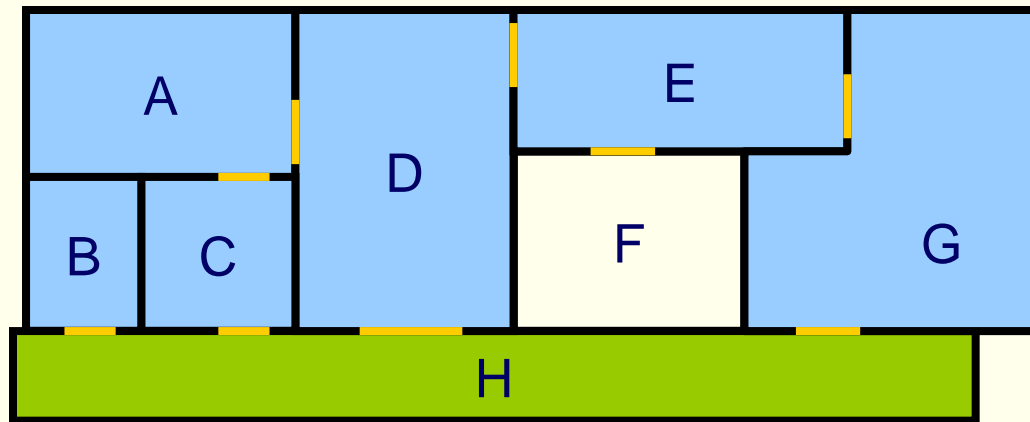
- *View-independent* solution: find all cells a particular cell could *possibly* see:



C can *only* see A, D, E, and H

# Cells & Portals

- *View-independent* solution: find all cells a particular cell could *possibly* see:



H will *never* see F

# Cells and Portals

- Questions:

- *How can we detect whether a given cell is visible from a given viewpoint?*
- *How can we detect view-independent visibility between cells?*

- The *key insight*:

- These problems reduce to eye-portal and portal-portal visibility

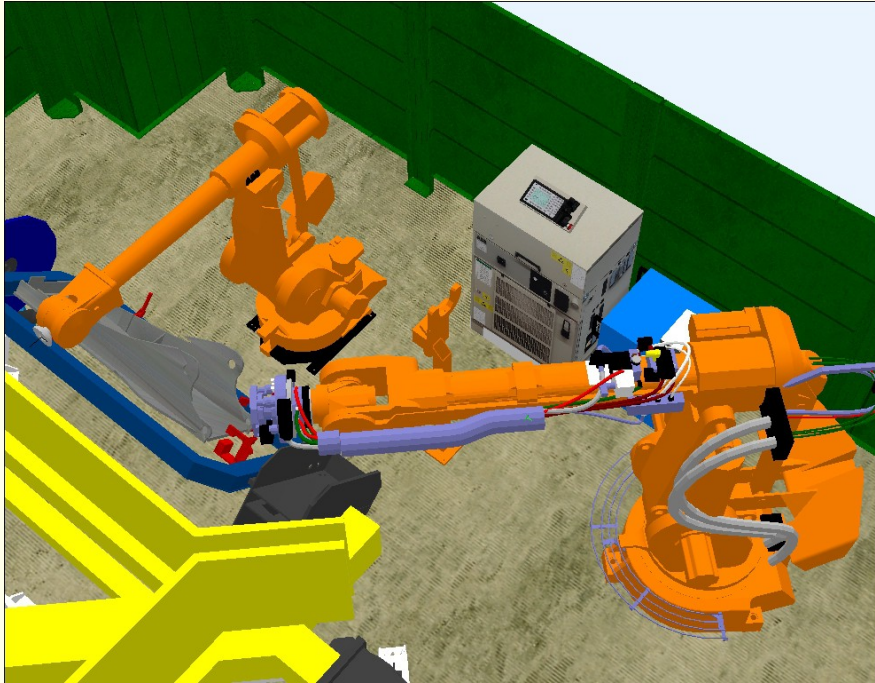
# Portal Culling Algorithm

- 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
- Which stages benefits?
  - Geometry, Rasterizer, and Bus
- Source (for Performer):  
<http://www.cs.virginia.edu/~luebke/>

# Izločanje podrobnosti

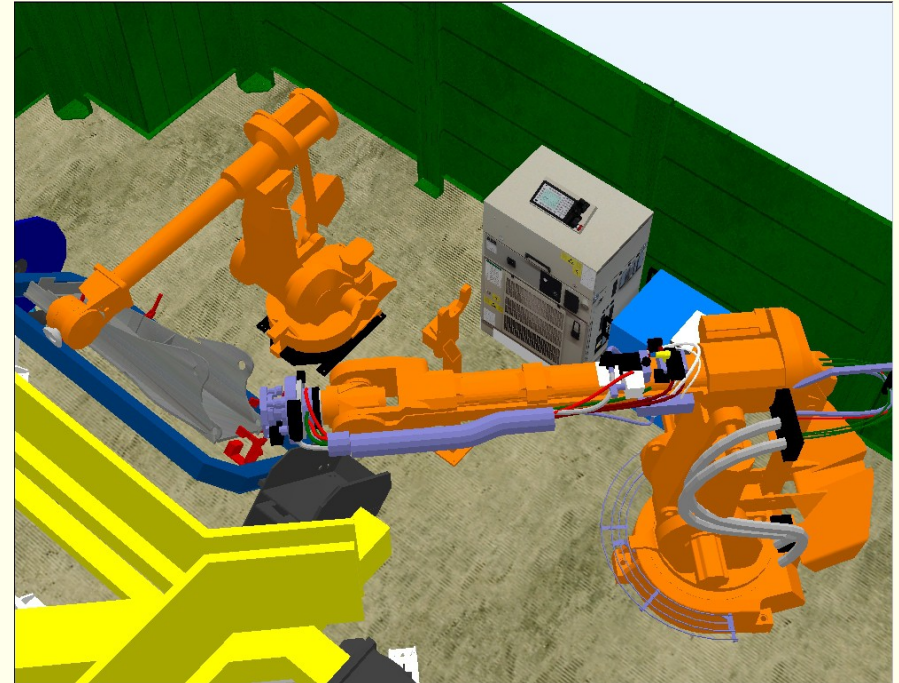
- Idea: objects whose projected BV occupy less than  $N$  pixels are culled
- This is an approximative algorithm as the things you cull away may actually contribute to the final image
- Advantage: trade-off quality/speed
- Which stages benefits?
  - Geometry, Rasterizer, and Bus

# Primer izločanja podrobnosti



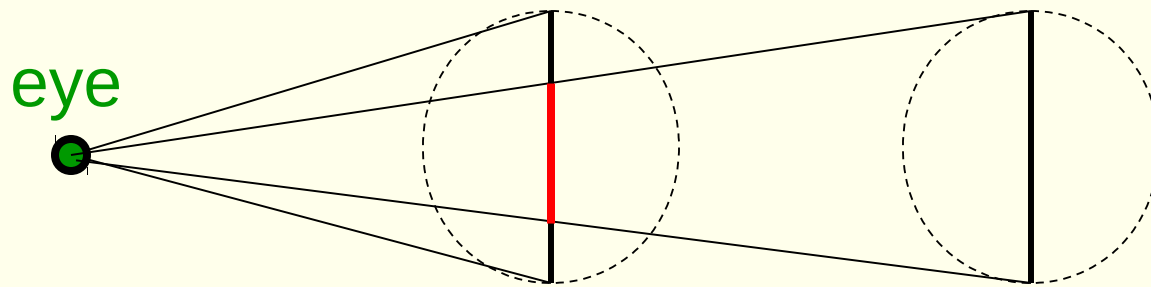
**detail culling OFF**

- Not much difference, but 80-400% faster
- Good when moving



**detail culling ON**

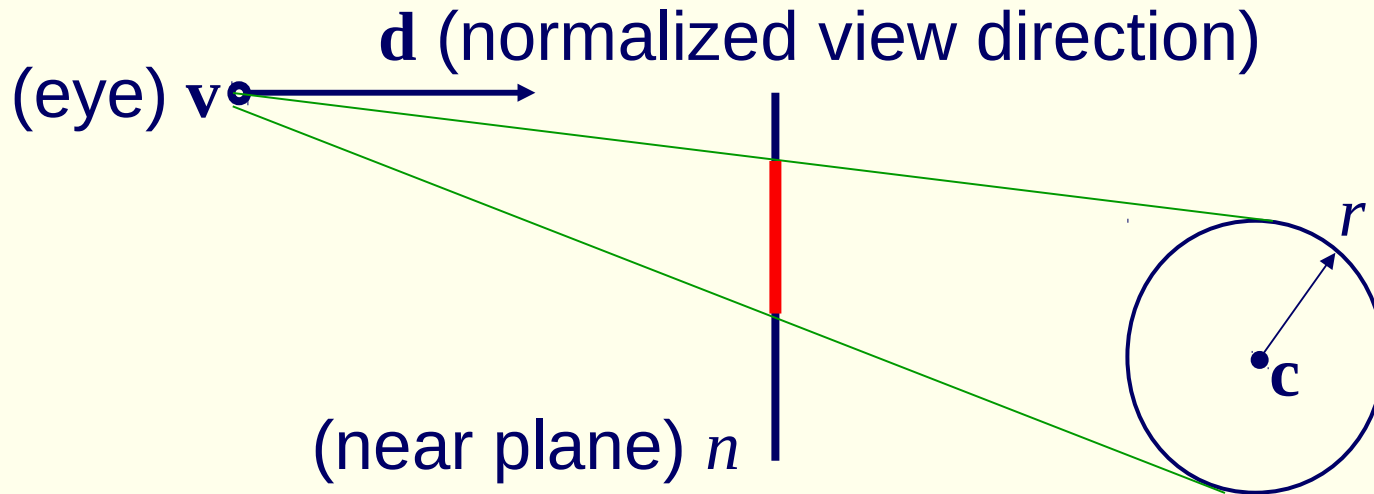
# Projekcija



- Projection gets halved when distance is doubled



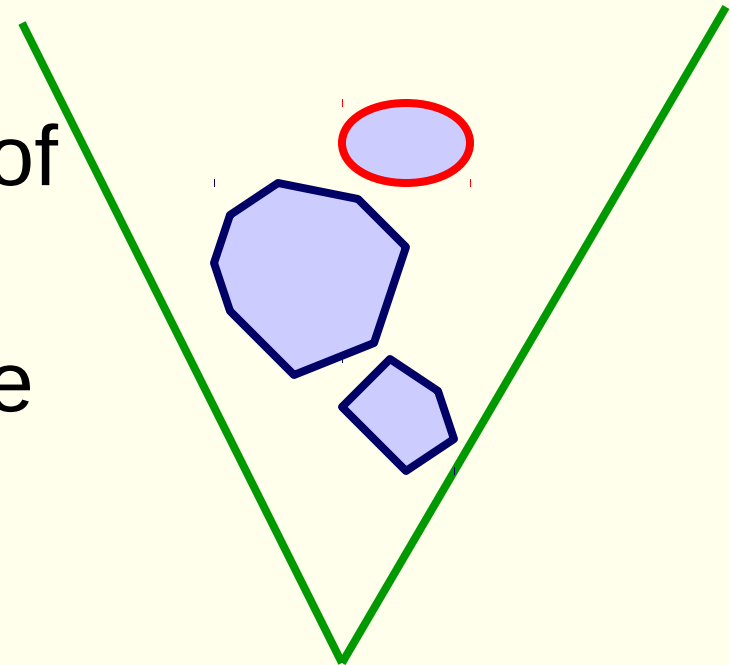
# Projekcija



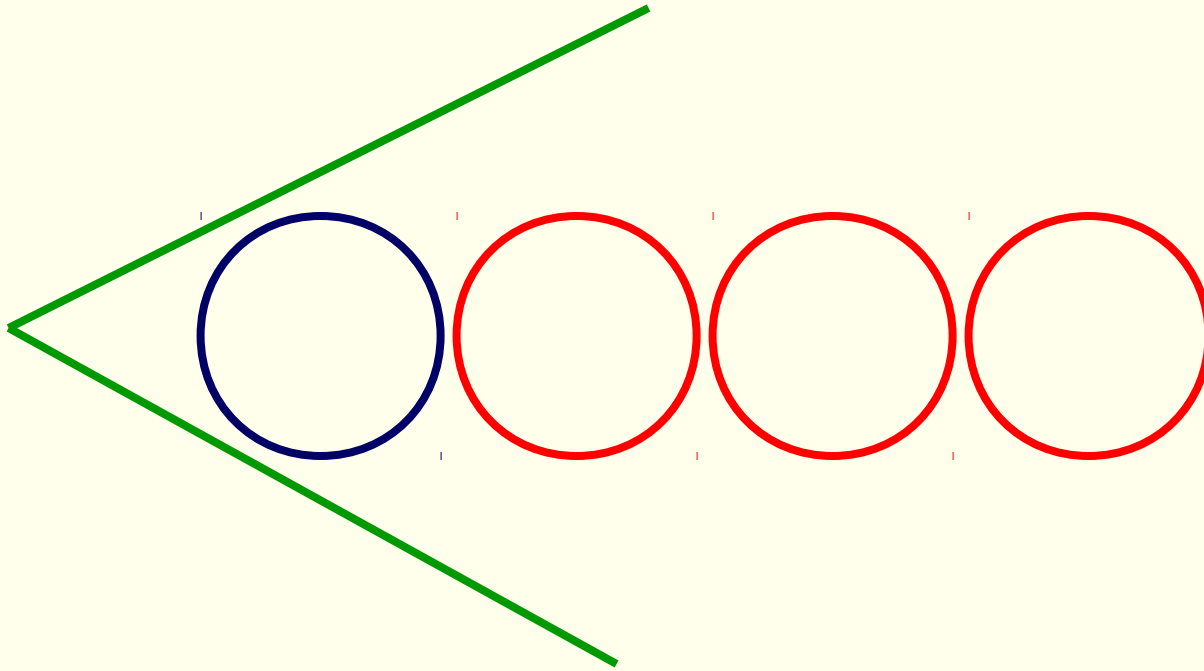
- $\text{dot}(d, (c-v))$  is distance along  $d$
- $p = nr / \text{dot}(d, (c-v))$  is estimation of projected radius
- $\pi p^2$  is the area

# Izločanje zakritih predmetov

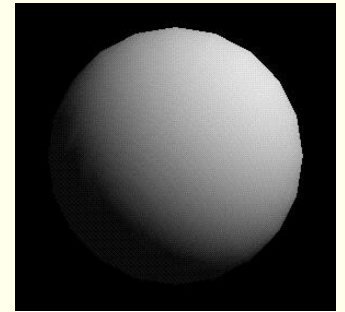
- Main idea: Objects that lies completely “behind” another set of objects can be culled
- Hard problem to solve efficiently



# Primer



final image



- Note that “Portal Culling” is an algorithm for occlusion culling

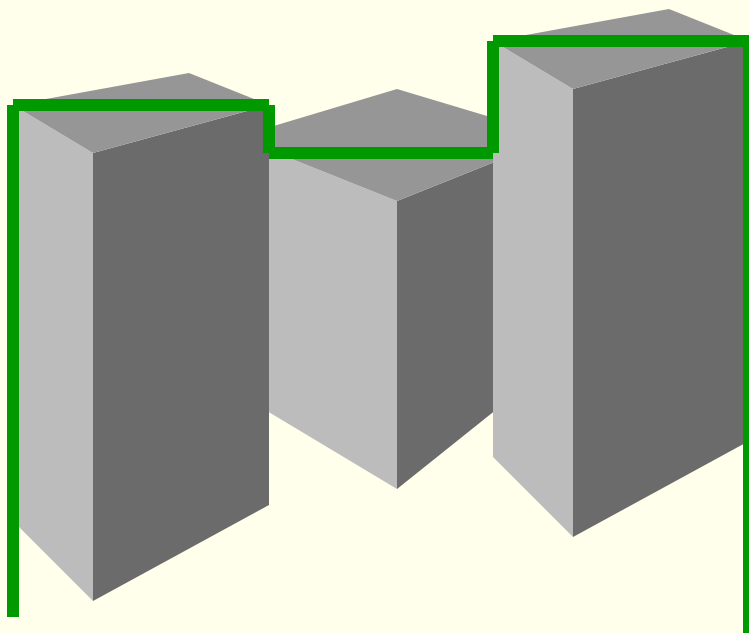
# Algoritem izločanja zakritih predmetov

Use some kind of occlusion representation  $\mathbf{O}_R$

```
for each object  $\mathbf{g}$  do:  
  if( not Occluded( $\mathbf{O}_R, \mathbf{g}$ ))  
    render( $\mathbf{g}$ );  
    update( $\mathbf{O}_R, \mathbf{g}$ );  
  end;  
end;
```

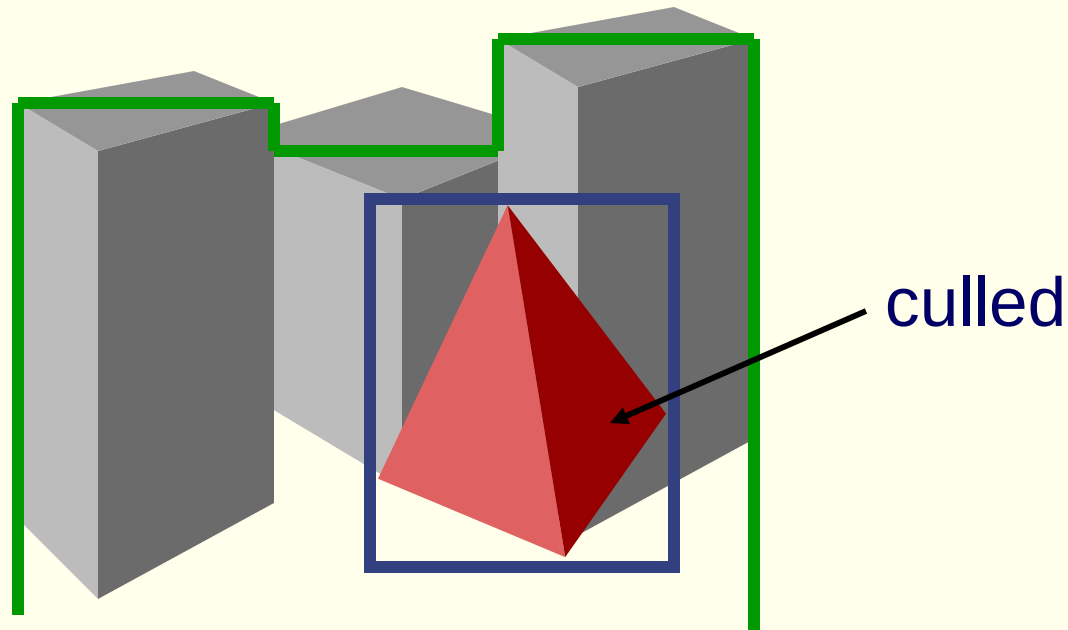
# Primer z algoritmom izločanja zakritih predmetov

- Process from front to back
- Maintain an occlusion horizon (green)



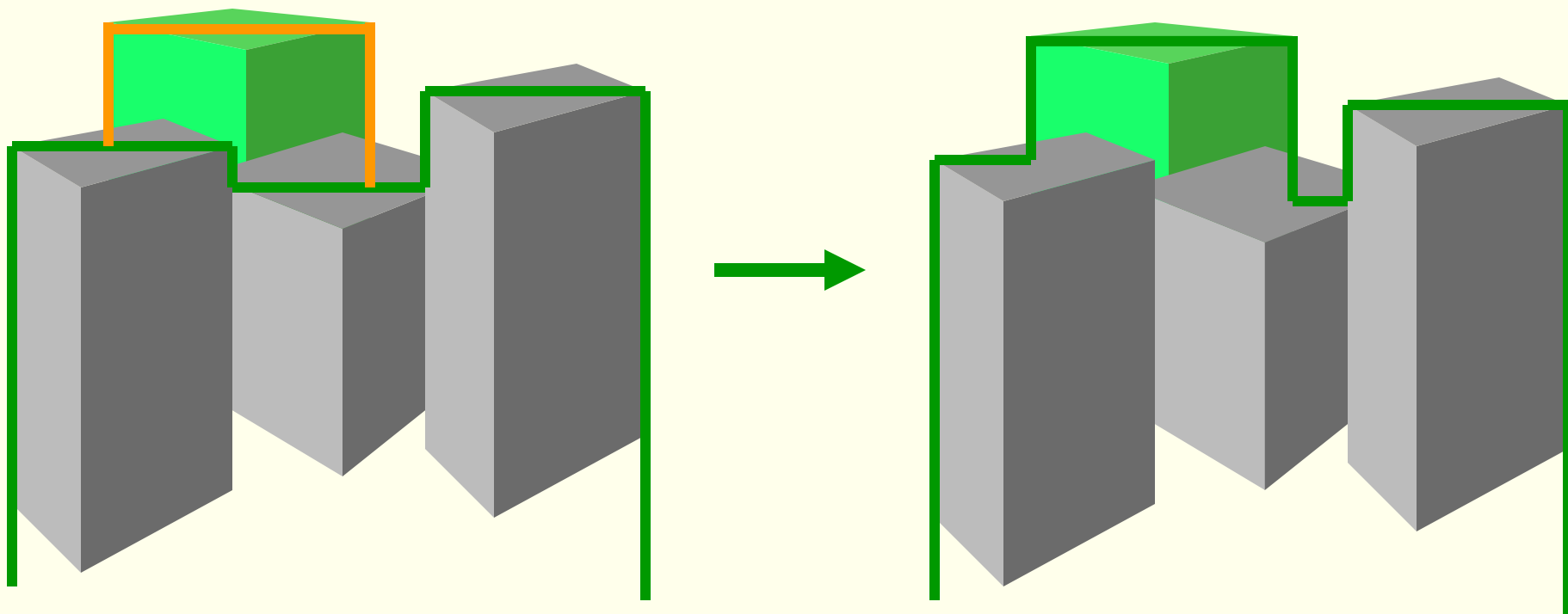
# Primer z algoritmom izločanja zakritih predmetov

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



# Primer z algoritmom izločanja zakritih predmetov

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



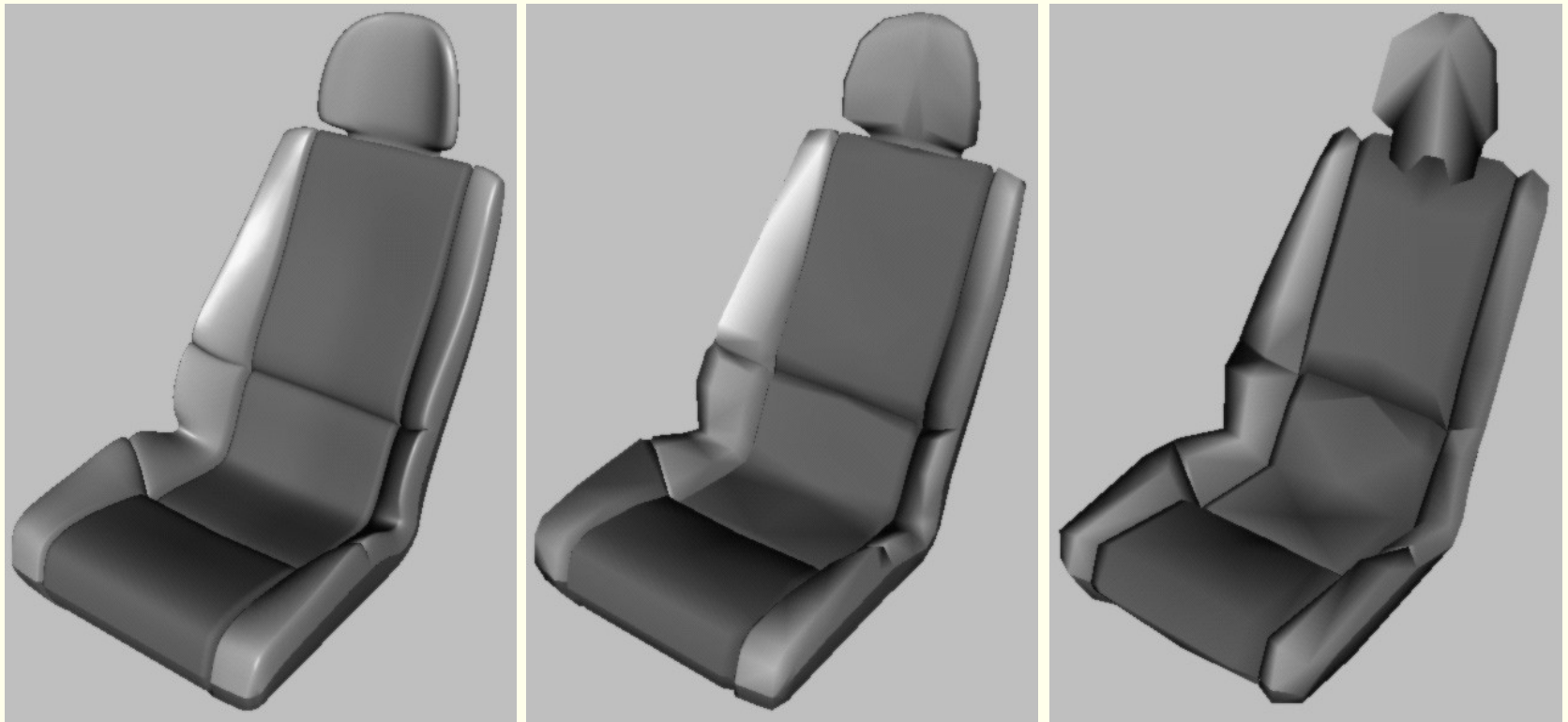
# Tvorba LOD

- **Geometrical simplification** which removes polygons from the original model. This may or may not preserve topology. For an other overview, you might want to take a look at:
- **Structural simplification** which uses a new, simpler representation of the object.
- **Scene simplification** which replaces areas of the scene with simpler representations.



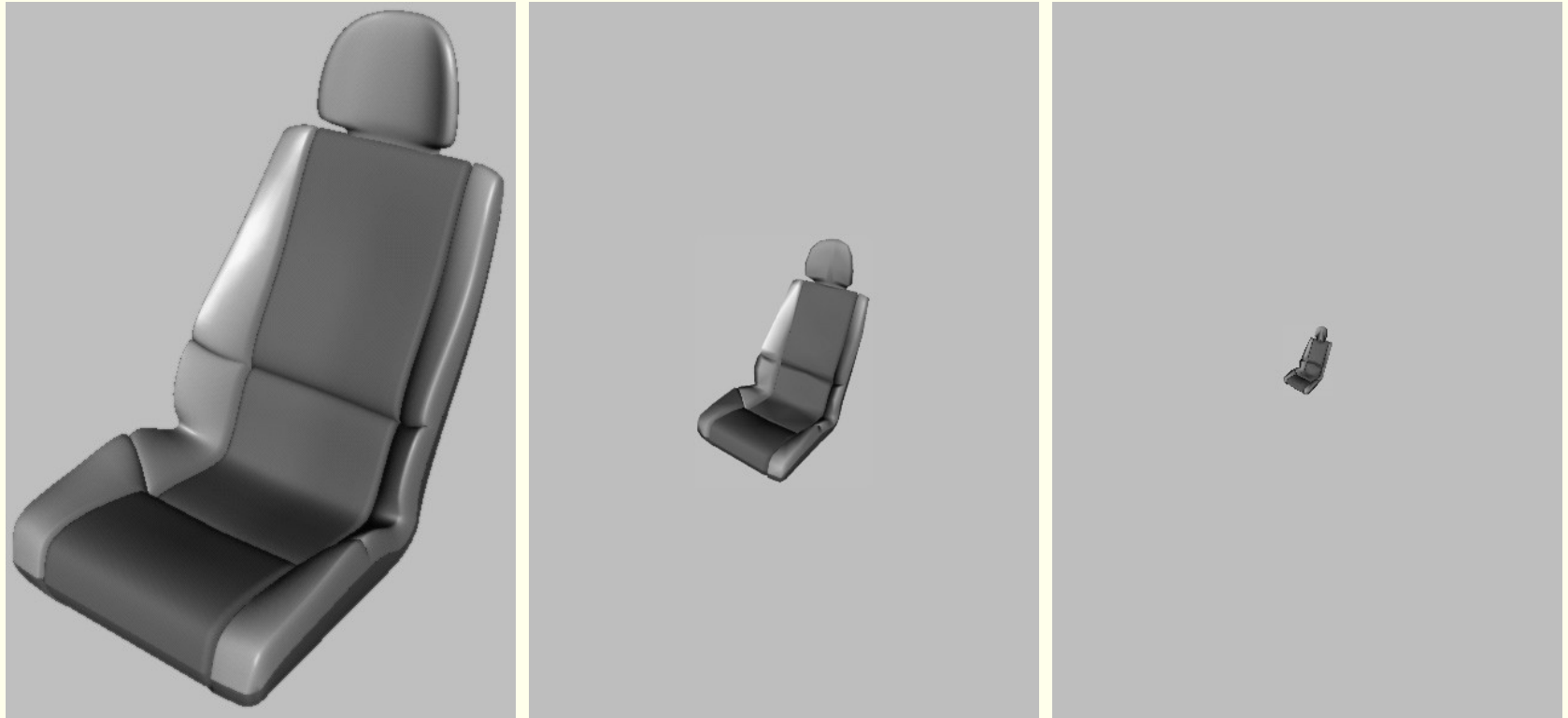
# Nivoji podrobnosti upodabljanja (LOD)

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



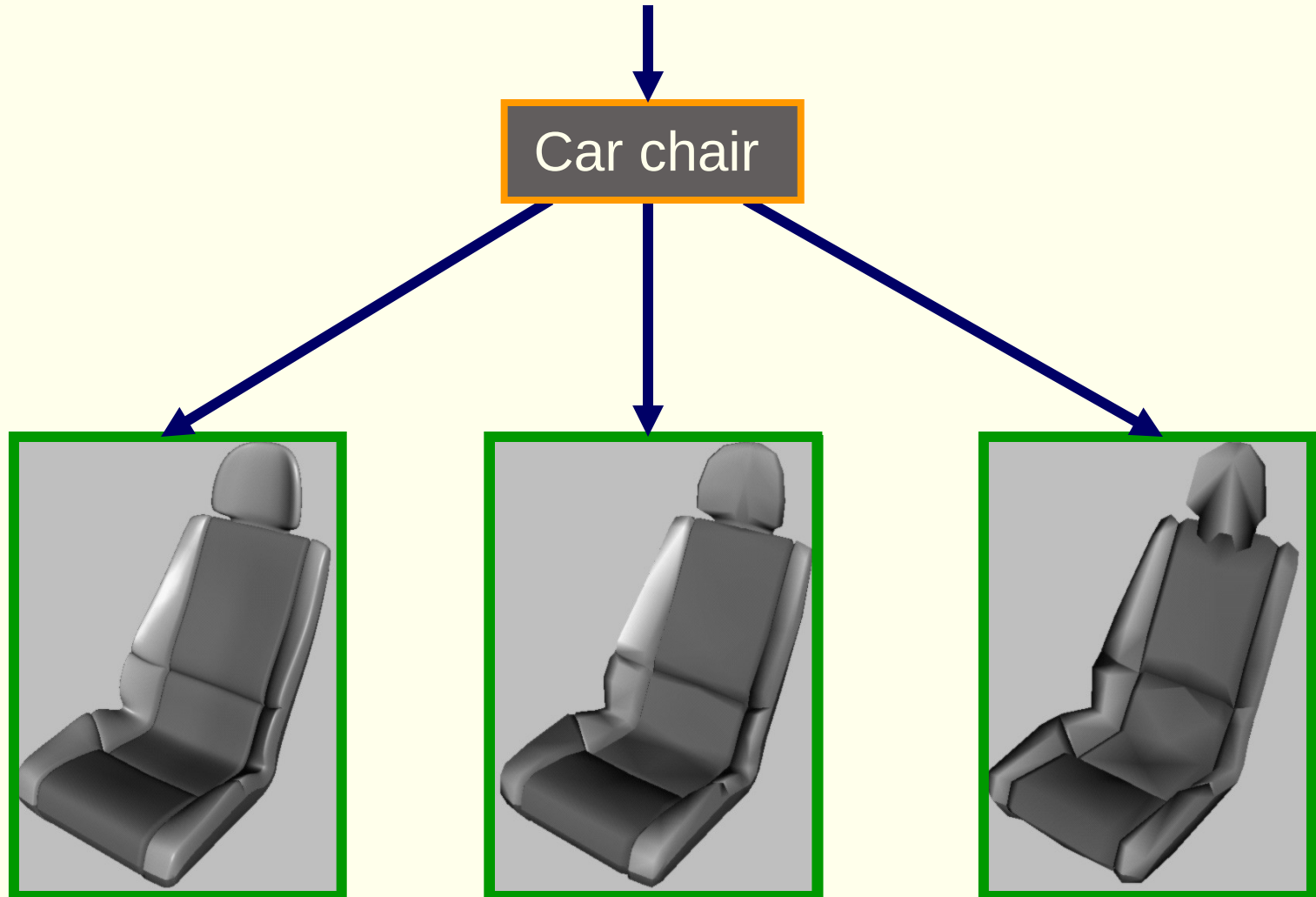
# Nivoji podrobnosti upodabljanja (LOD)

- Not much visual difference, but a lot faster



Use area of projection of BV to select appropriate LOD

# Graf scene z LOD

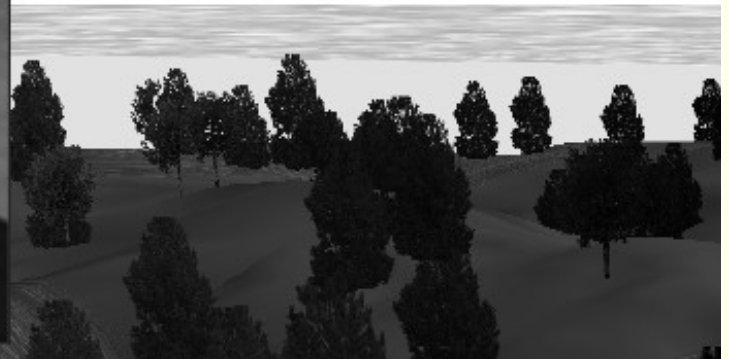
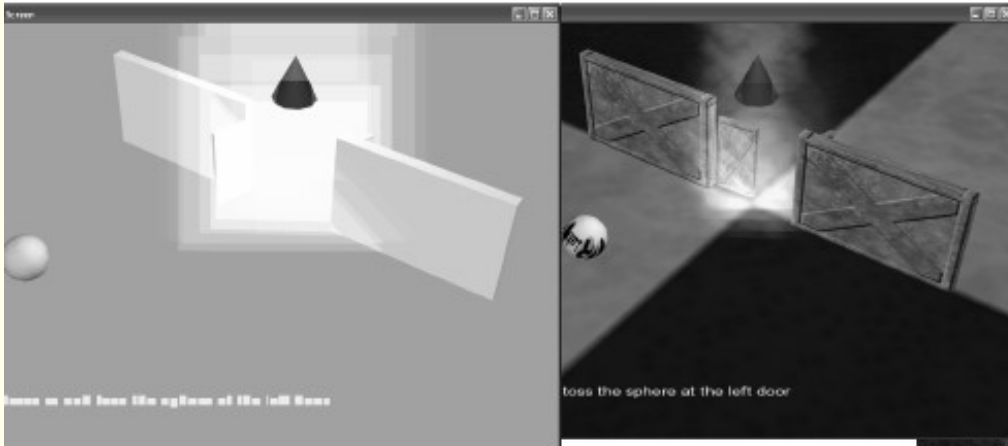
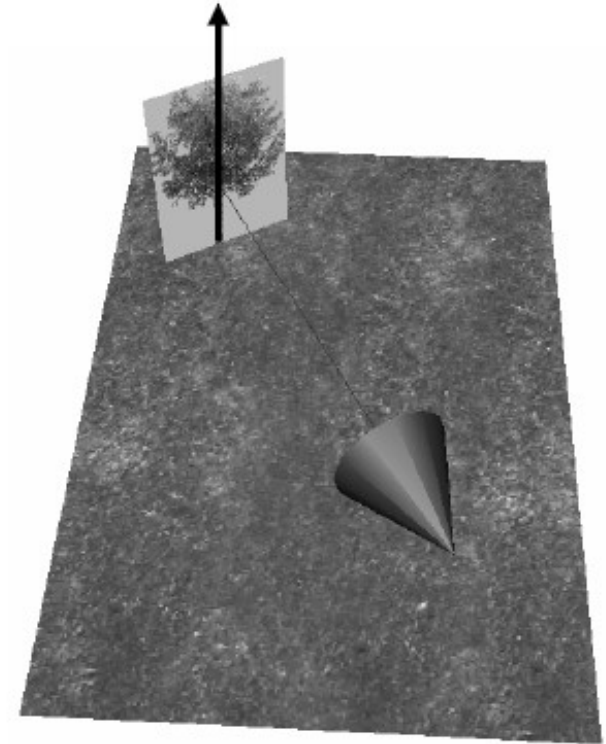


# Plakati (billboards)

## Demo

- Orienting a polygon relative to the viewer
- Different alternatives
  - Always facing the viewer
    - Smoke, fire, fog, explosions, clouds
    - Align the surface normal to the view vector
  - Constraining one or more axis
    - A tree.

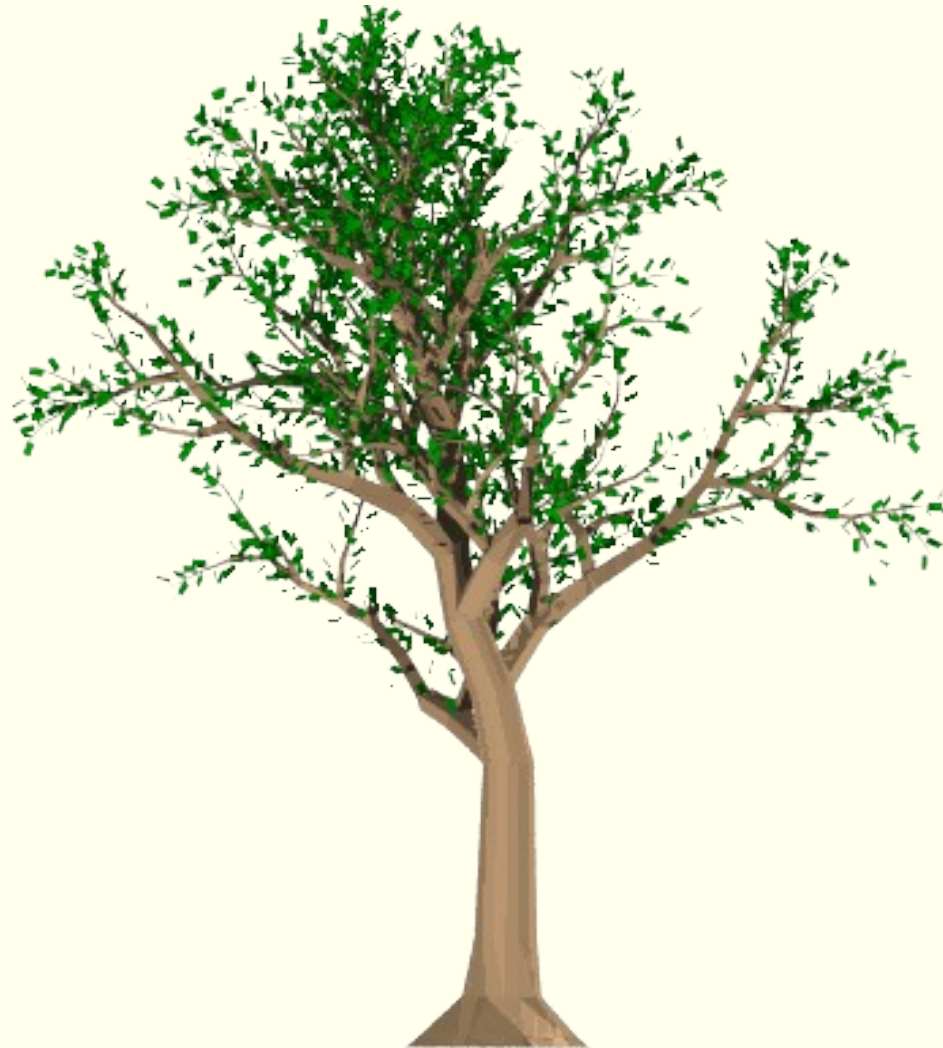
Constrained to the Z-axis



# Namesto modela plakat

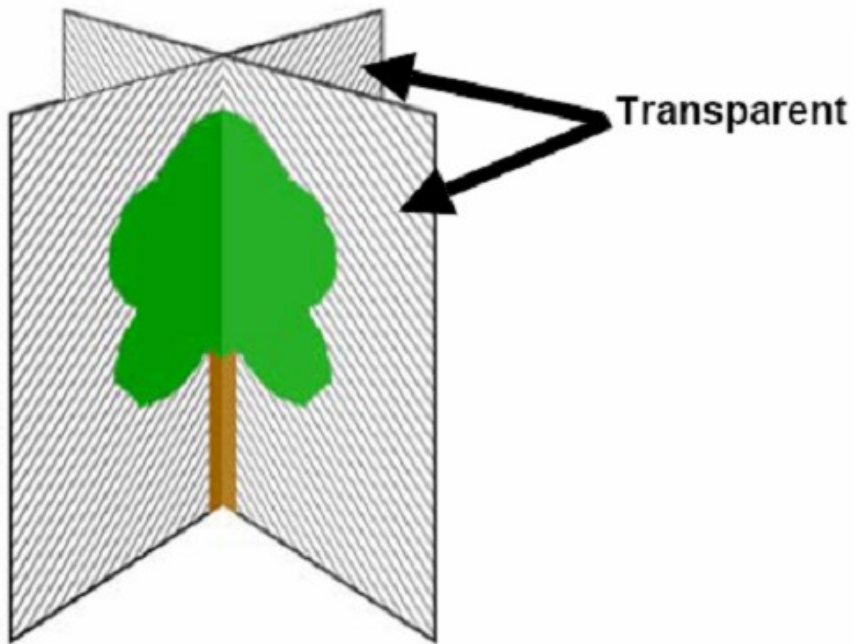
Tehnika plakata

Namesto kompleksnih modelov le k nam obrnjena tekstura



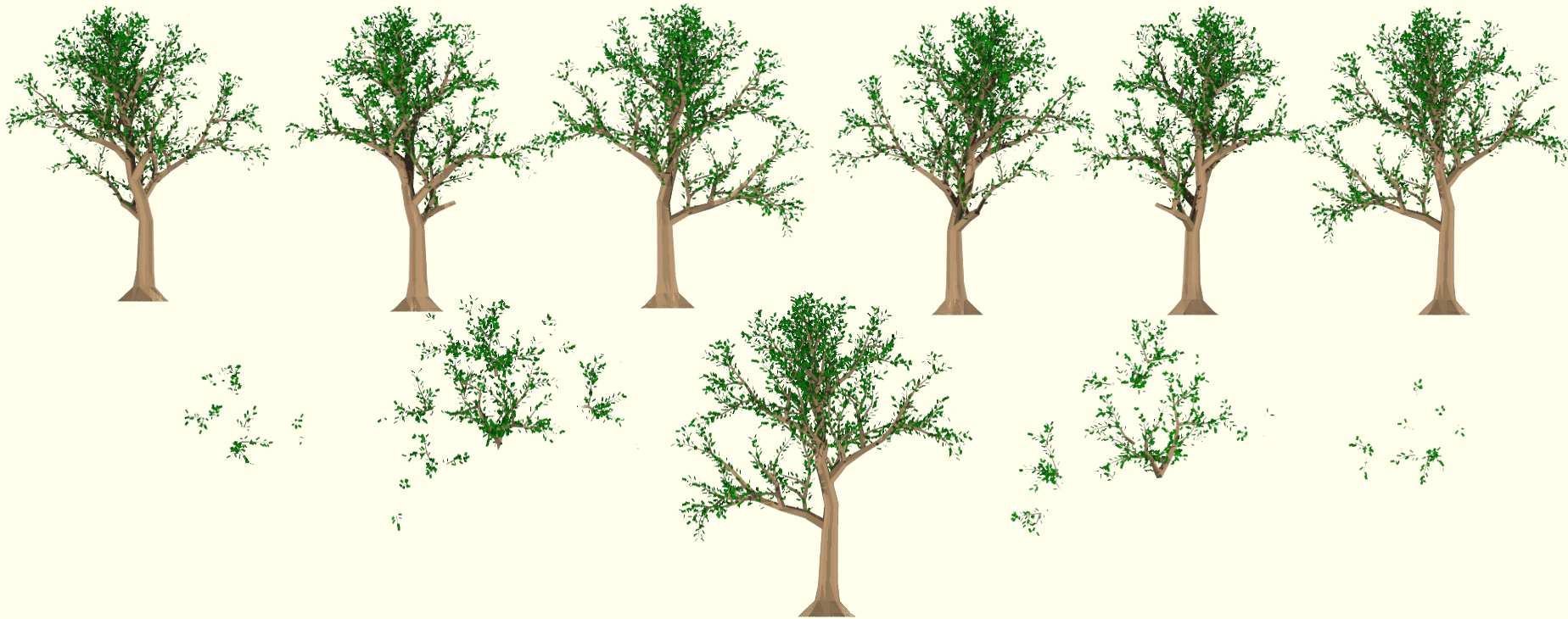
Free movement around  
aslicing-based tree and the  
original mesh tree.

# Namesto modela plakat



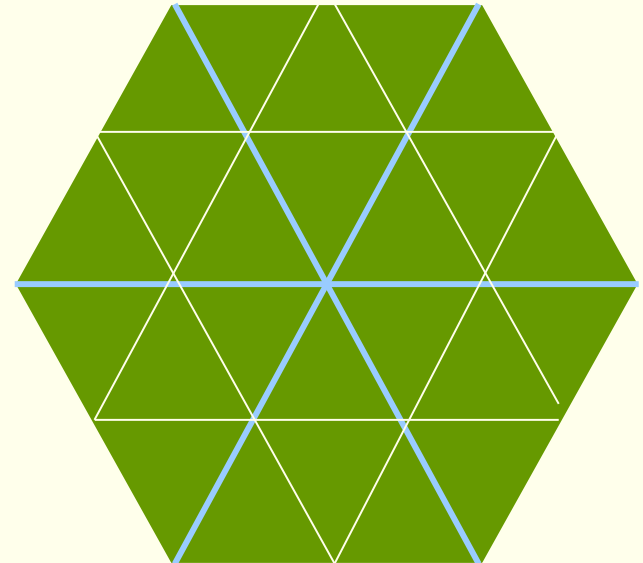
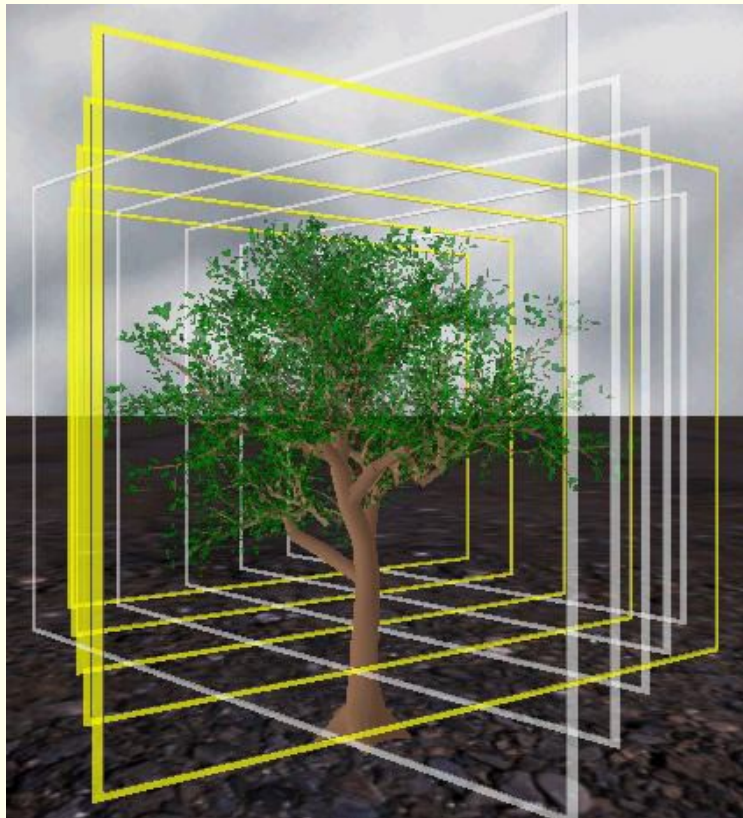
# Rešitev

- Drevo posnamemo iz več smeri.
  - 6 pogledov, ortogonalna projekcija



# Implementacija

- Postavimo vse poglede v prostor.

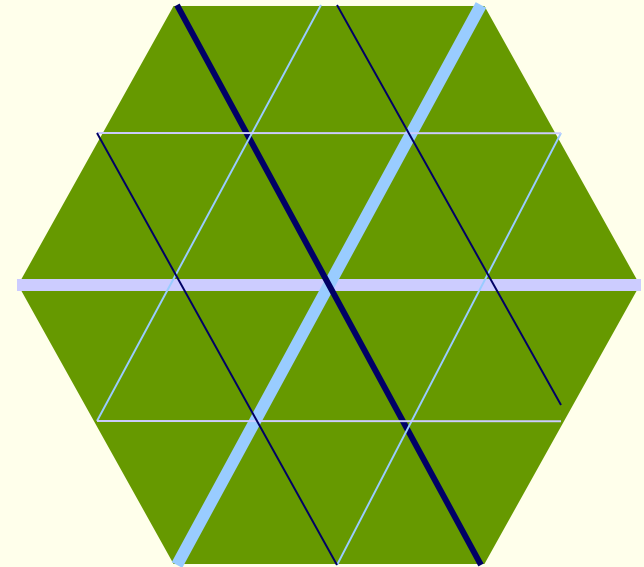




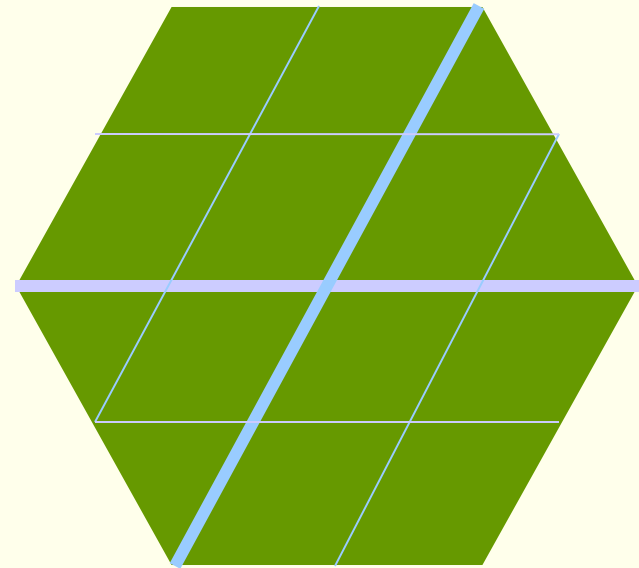
# Podrobnosti implementacije

- Naložiti moramo teksture, jim spremeniti velikost (povprečenje z upoštevanjem kanala alfa), da ne porabimo preveč pomnilnika.
- Pazimo na `atan2()`, inicializiramo 3D kartico.

- Najdemo najbližja dva pogleda.
- Projeciramo poligone v koordinatni sistem zaslona.
- Določimo prosojnost posameznega pogleda.
  - Funkcija za blending je  $(1-\alpha) \cdot \text{ozadje} + \alpha \cdot \text{poligon}$
  - Prosojnost bližnjega pogleda je med 100% in 50%, oddaljenega med 50% in 0%. Prehajanje mora biti mehko.
- Rišemo od zadaj naprej, najprej bolj oddaljen pogled, potem pa še bližnji.
  - Kompliciranje z z-bufferji ali rezanjem oddaljenega pogleda ob bližnji ne naredijo velike razlike.

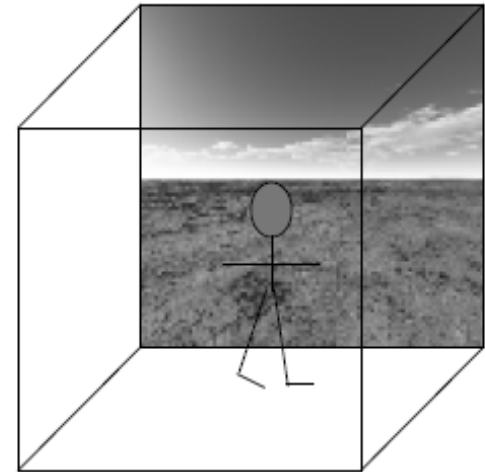


- Rišemo od zadaj naprej, najprej bolj oddaljen pogled, potem pa še bližnji.
  - Kompliciranje z z-bufferji ali rezanjem oddaljenega pogleda ob bližnji ne naredijo velike razlike.



# Skybox

- A surrounding environment, always present. Relative to the camera.
- Simulates the sky, far away mountains, etc...
- Always drawn "behind" all other objects
- To render it:
  1. Clear the depth and color buffers
  2. Apply camera transformation
  3. Disable the depth buffer test and writes
  4. Draw the skybox
  5. Enable depth buffer test and writes
  6. Draw the rest of the scene



# Skybox



# Kombinacija tehnik

- Not trivial to combine!
- VF Cull + LOD == minimal if serious about performance
- Indoors: LOD + portal culling
- Outdoors (cities, suburbia): VF Cull + LOD + occlusion culling

# Real-Time Rendering?

- In computer graphics, “real-time” is used in a soft way: say  $>30$  fps for most frames
- In other contexts, it’s a tougher requirement: the framerate must never be  $<30$  fps, i.e., constant framerate
- What can we do?
  - Reactive LOD algorithm
  - Reactive detail culling
  - Reactive visual quality

# Odkrivanje trkov

- To get interesting and more realistic interaction between geometric objects in an application
- Cannot test each triangle against each other:  $O(n*m)$
- Alas, need smarter approaches
- Use BVs because these are cheap to test
- Use a scene graph (hierarchy)
  - For efficiency use only one triangle per leaf

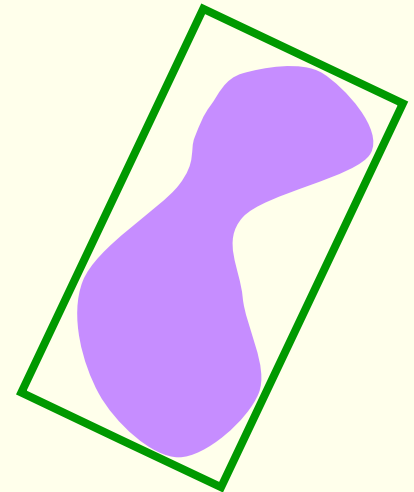
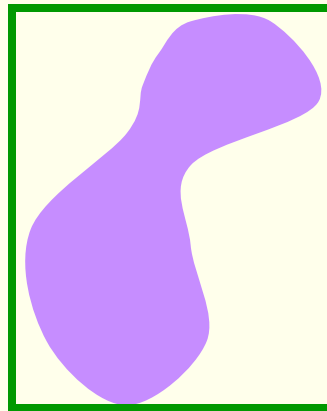
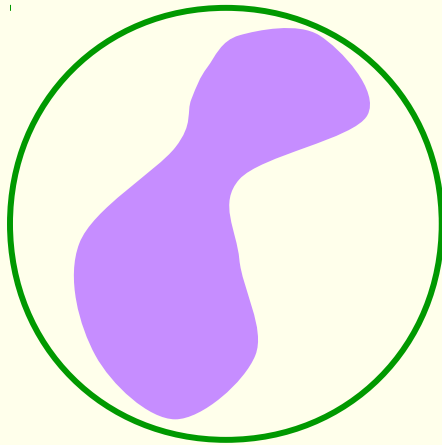


# Kolizija 2 grafov scene?

- Three cases:
- Reach two internal nodes' BVs
  - If not overlap, then exit
  - If overlap, then descend into the children of the internal node with largest volume
- Reach an internal node and a triangle
  - Descend into the internal node
- If you reach two triangles
  - Test for overlap
- Start recursion with the two roots of the scene graphs

# Bounding Volumes in Collision Detection

- Wanted: tight fitting BV, and fast to test for overlap
- Common choices are
  - Spheres
  - Axis aligned box
  - Oriented box

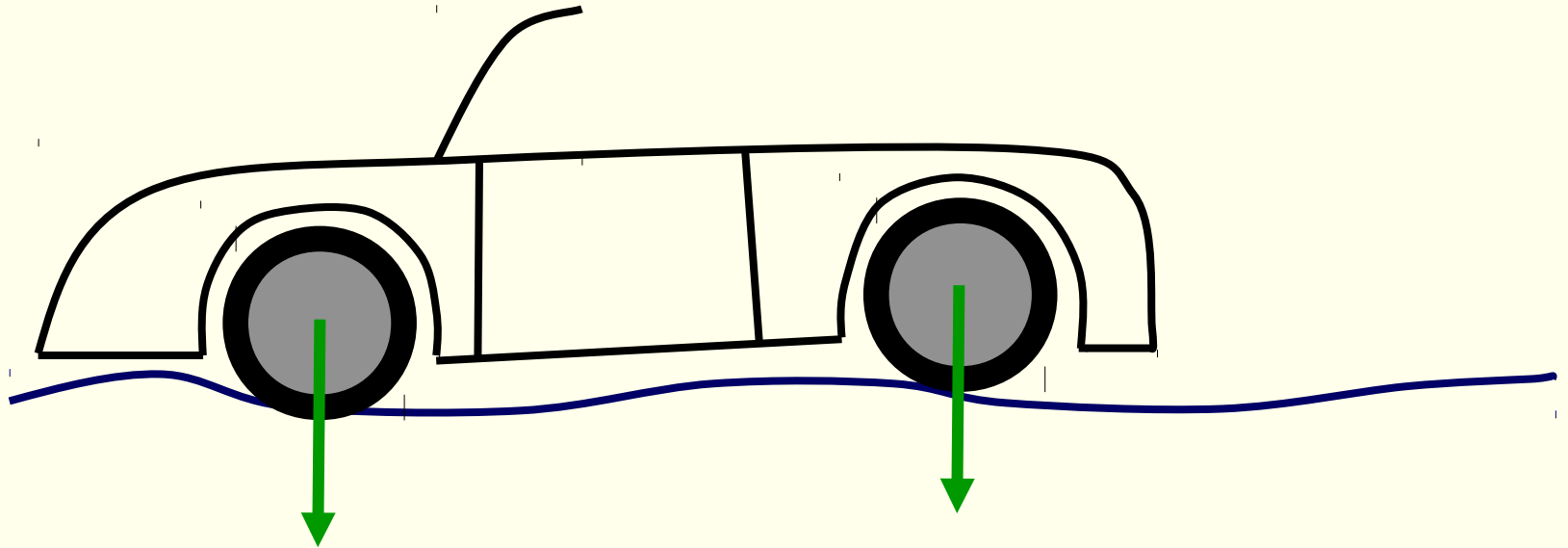


# Triangle-triangle overlap test

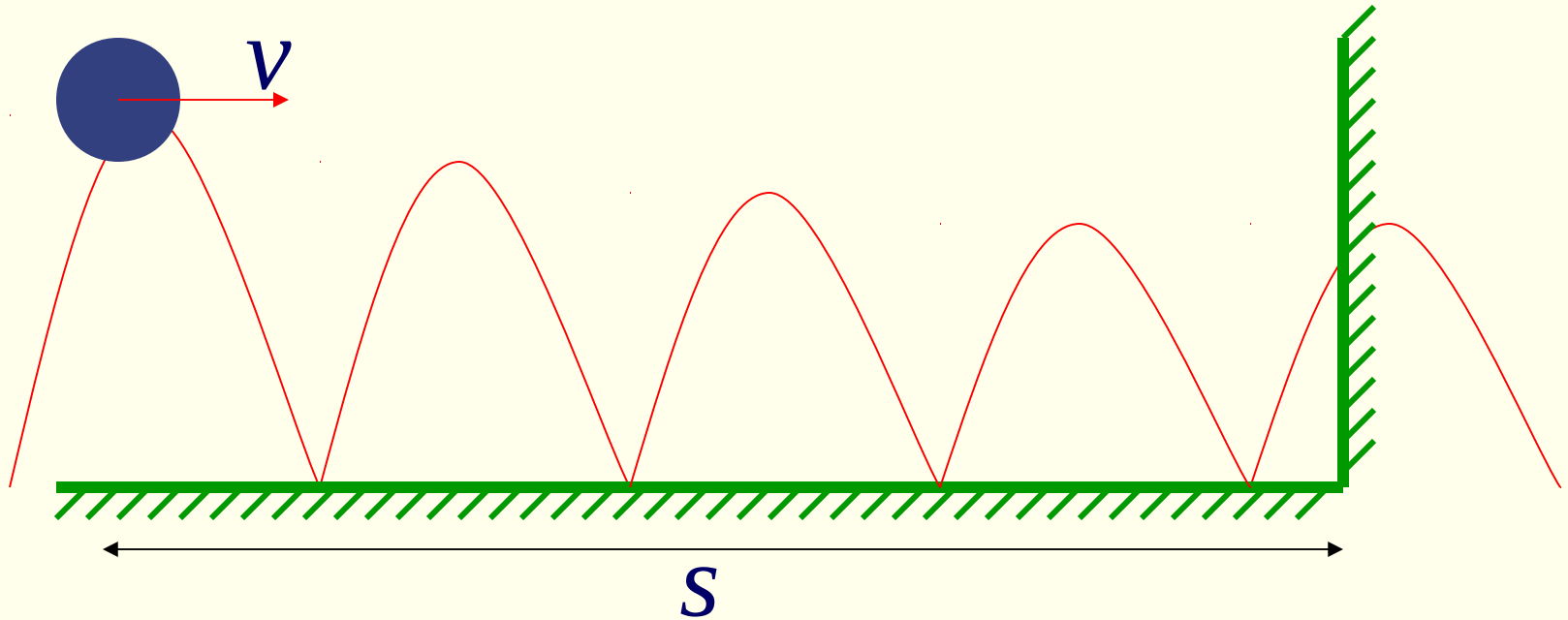
- Compute the line of intersection between the two planes
- Compute the intersection interval between this line and triangles
  - gives two intervals
- If intervals overlap then triangles overlap!

# Simpler collision detection

- Only shoot rays to find collisions, i.e., approximate an object with a set of rays
- Cheaper, but less accurate
- Can use the scene graph again or a BSP tree



# Can you compute the time of a collision?



- Move ball, test for hit, move ball, test for hit... can get “quantum effects”!
- In some cases it’s possible to find closed-form expression:  $t = s/v$