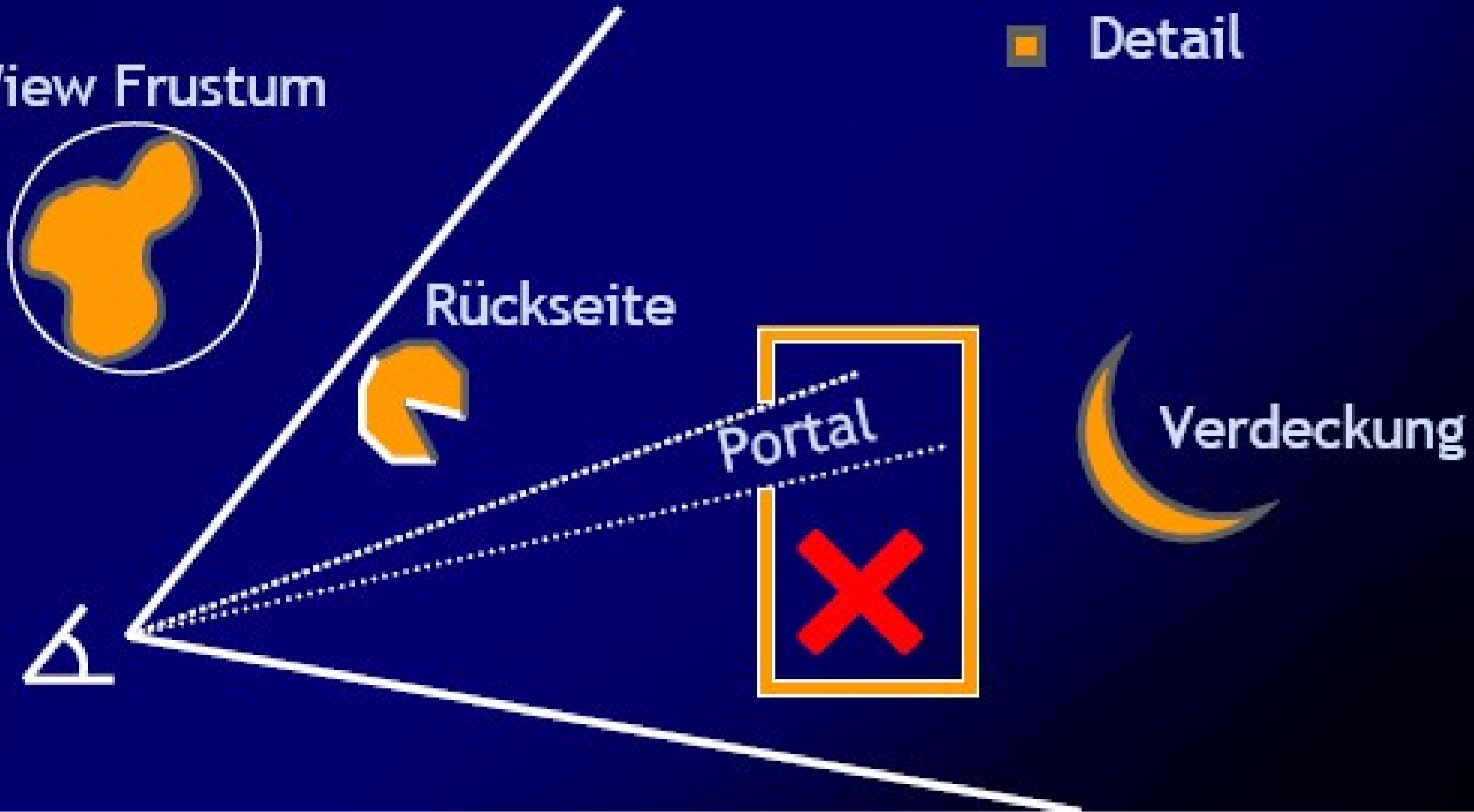
Visibility Culling

- Back face culling
- View-frustrum culling
- Detail culling
- Occlusion culling

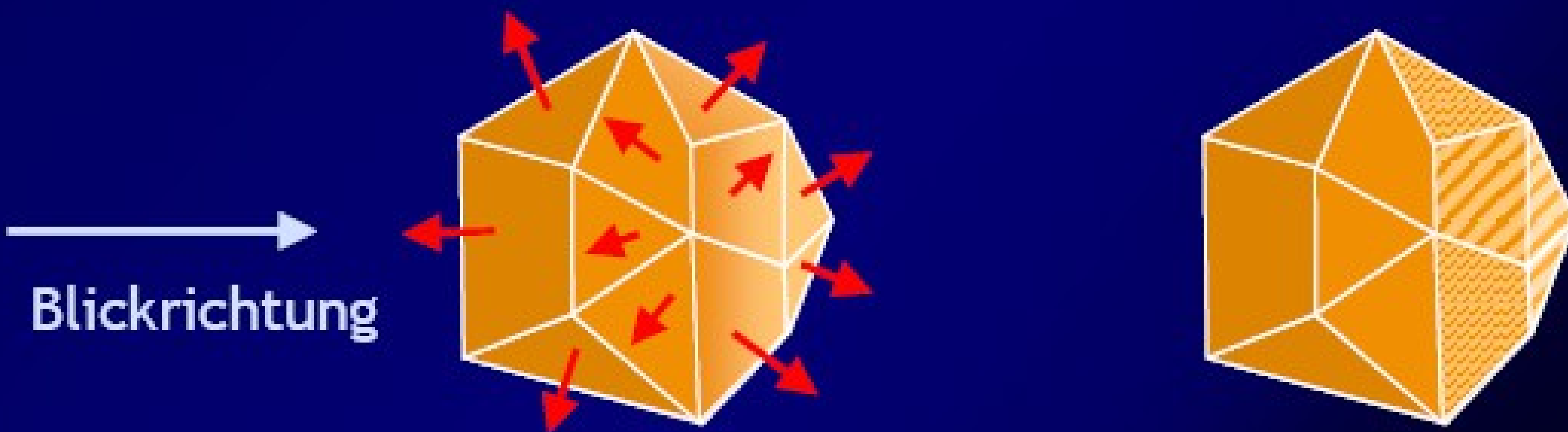
Culling Techniken

- Ziel: Reduktion der Flächen, die von einem Renderer (z.B. Z-Buffering) verarbeitet werden müssen



Backface Culling

- Oberflächen, dessen Normalen vom Augpunkt wegschauen, können nicht sichtbar sein



Schraffierte Flächen: entfernbar

⇒ vor dem Sichtbarkeitsverfahren entfernen

Backface Culling

- Flächen mit vom Betrachterstandpunkt wegweisenden Normalvektoren werden nicht dargestellt.
- Sei v der Blickvektor und n die Oberflächennormale einer Seite eines konvexen Polyeders, dann ist die Seite sichtbar, falls gilt:

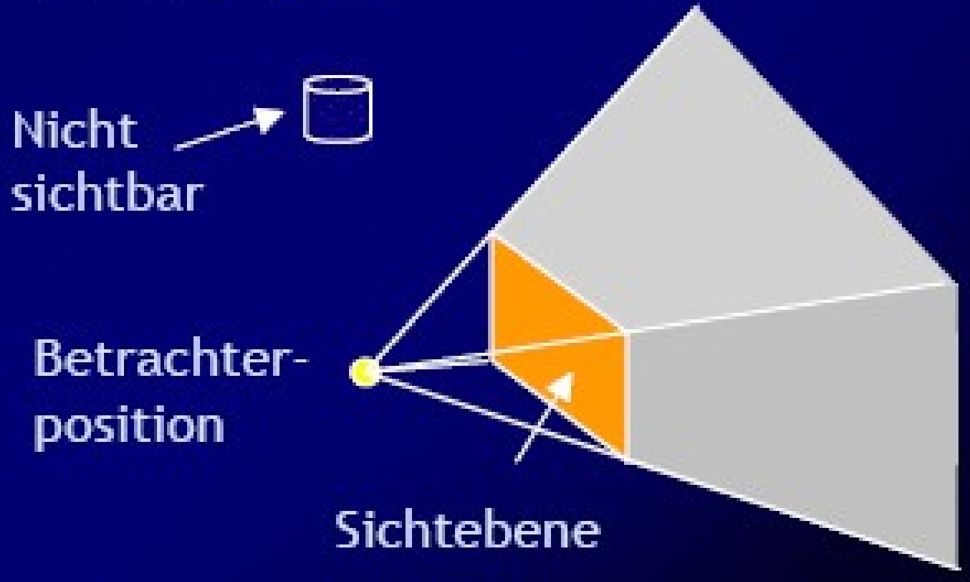
$$-1 \leq \frac{v \bullet n}{|v| \cdot |n|} \leq 0$$

  und sonst verdeckt

- Alle abgewandten Seiten erfüllen die Bedingung nicht $\Rightarrow$ 50% der Flächen fallen weg!

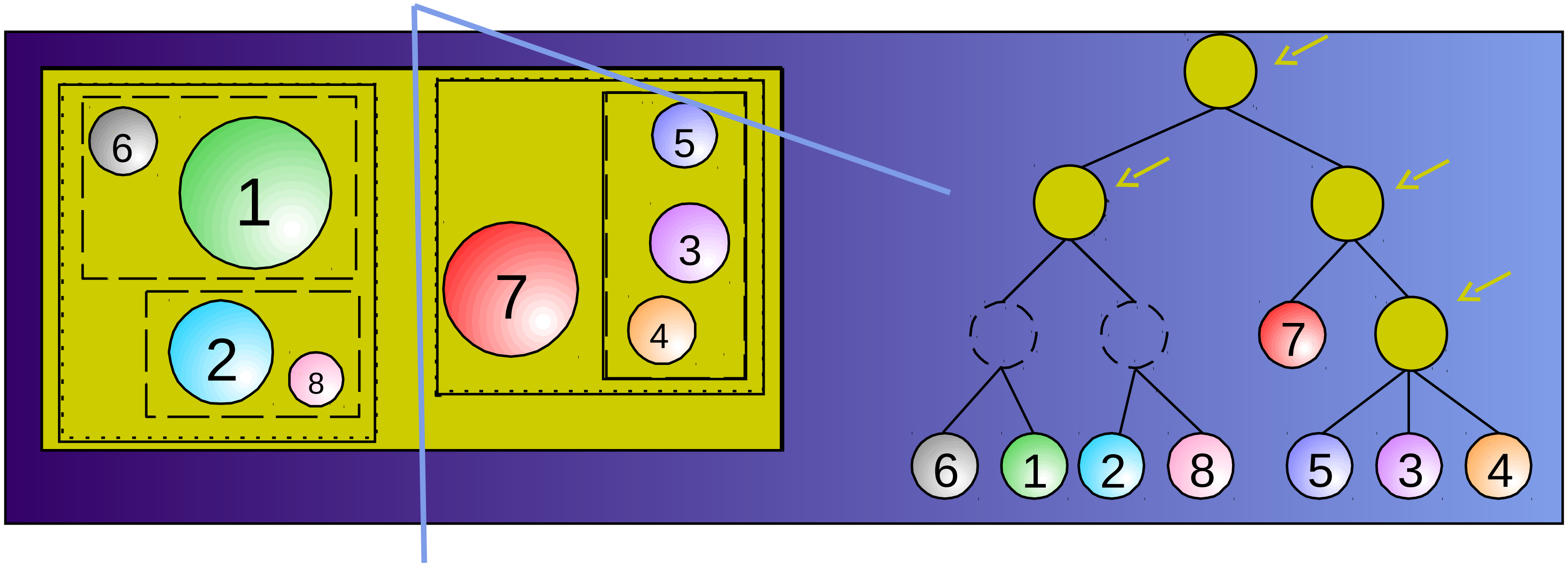
View Frustum Culling

- Durch Position und Blickrichtung des Betrachters wird ein fünfseitiger Halbraum, das *View Frustum*, definiert.
- Oft wird das View Frustum auch als sechseitiges Volumen definiert, das noch eine zusätzliche weite Clipping Ebene enthält.



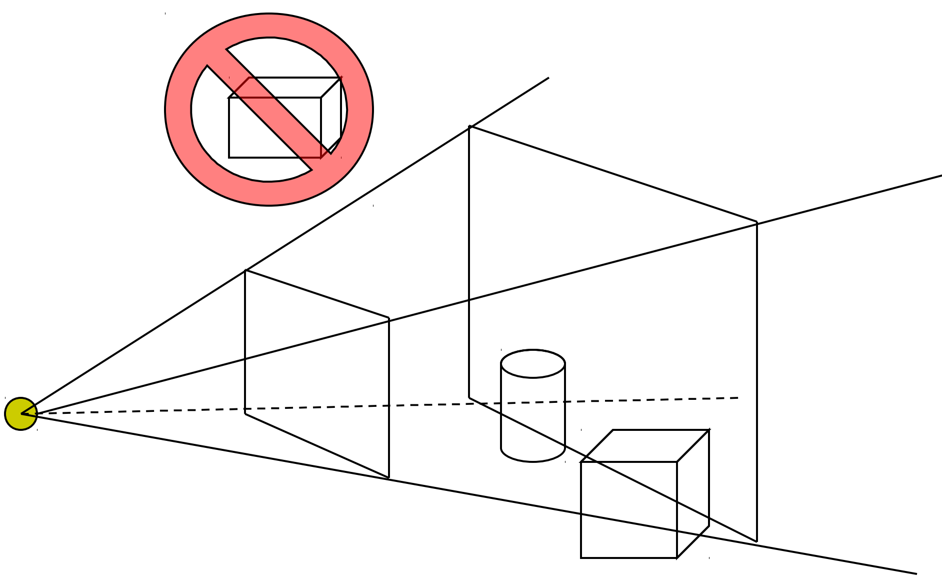
View-Frustum-Culling mit Objekthierarchien

- Beim Traversieren einer Objekt-Hierarchie wird in jedem inneren Szenenknoten ein Sichtbarkeitstest unter Verwendung von Hüllkörpern durchgeführt.



View-Frustum Culling

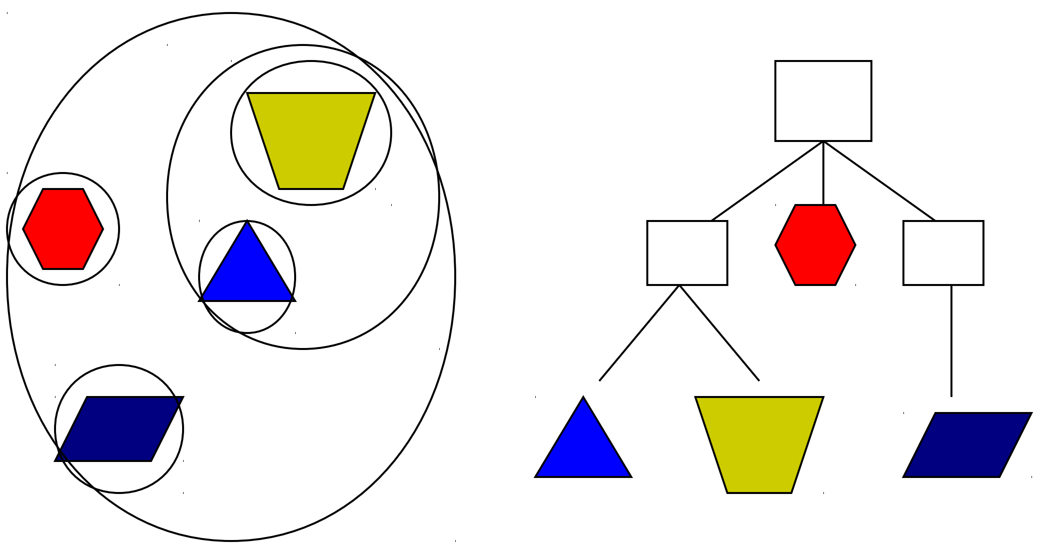
- Done in the application stage
- Remove objects that are outside the viewing frustum
- Can use BVH, BSP, Octrees



1. Create hierarchy
2. BV/V-F intersection tests

View-Frustum Culling

- Often done hierarchically to save time

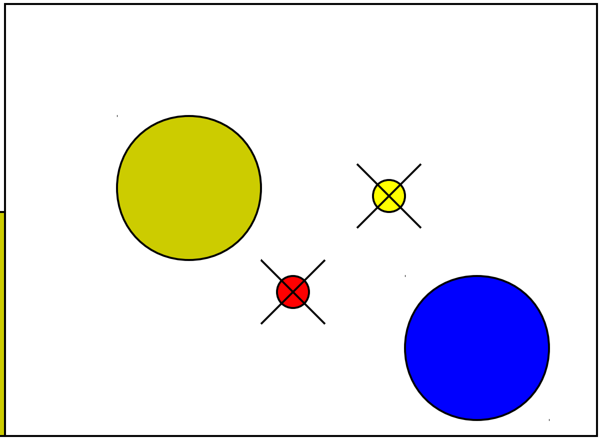


In-order, top-down traversal and test

Detail Culling

- A technique that sacrifices quality for speed
- Base on the size of projected BV – if it is too small, discard it.
- Also often done hierarchically

Always helps to create a hierarchical structure, or scene graph.



Occlusion Culling

- Discard objects that are occluded
- Z-buffer is not the smartest algorithm in the world (particularly for high depth-complexity scenes)
- We want to avoid processing invisible objects

Occlusion Culling (2)

```
OcclusionCulling (G)
Or = empty
For each object g in G
    if (isOccluded(g, Or))
            skip g
    else
            render (g)
        update (Or)
    end
End
```

G: input graphics data

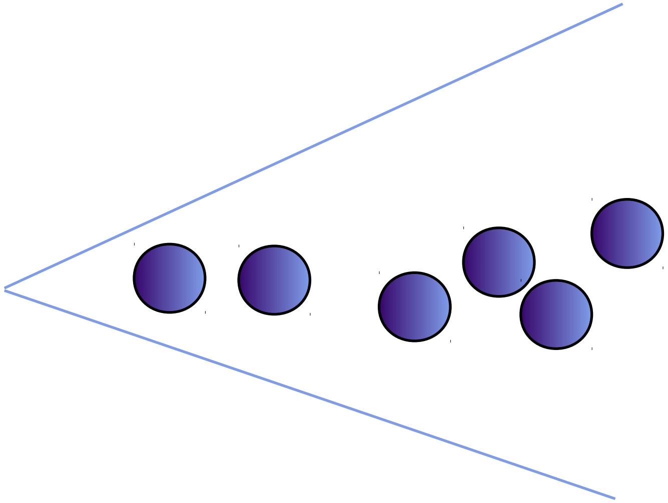
Or: occlusion hint

Things needed:

1. Algorithms for isOccluded()
2. What is Or?
3. Fast update Or

Hierarchisches Occlusion-Culling 2

HOcclusionCulling (N)
if not (isOccluded(N_{BV}, O_R))
    for each primitive $p \in N$
        Render(p)
        Update(O_R, p)
    end
    for each child node $C \in N$ in front-to-back order
        HOcclusionCulling (C)
    end
end



O_R = empty
HOcclusionCulling(root)

Hierarchisches Occlusion-Culling 3

HOcclusionCulling (*N*)

if not (isOccluded(N_{BV}, O_R))

for each primitive $p \in N$

Render(*p*)

Update(O_R, *p*)

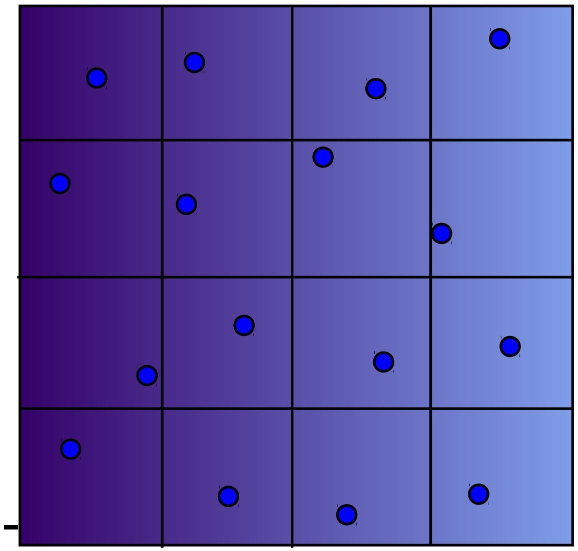
end

for each child node $C \in N$ in front-to-

HOcclusionCulling (C)

end

end



O_R = sampled

HOcclusionCulling(root)

Hierarchical Visibility

- One example of occlusion culling techniques
- Object-space octree
  - Primitives in an octree node are hidden if the octree node (cube) is hidden
  - A octree cube is hidden if its 6 faces are hidden polygons
  - Hierarchical visibility test:

Hierarchical Visibility

From the root of octree:

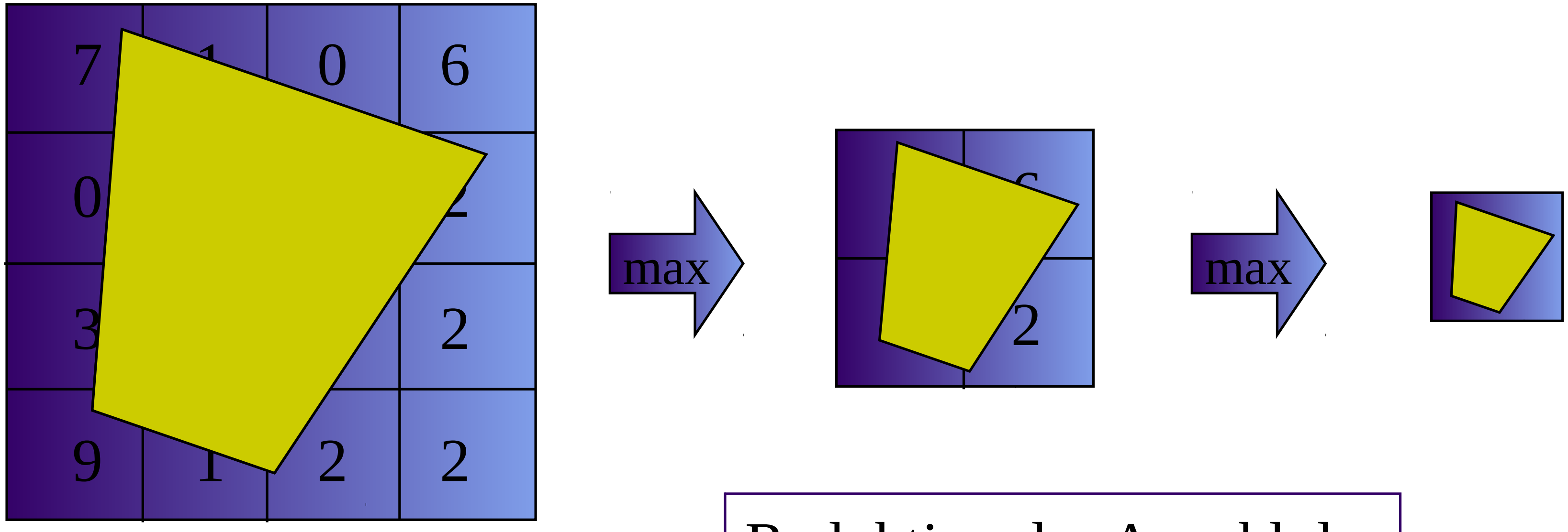
- View-frustum culling
- Scan conversion each of the 6 faces and perform z-buffering
- If all 6 faces are hidden, discard the entire node and sub-branches
- Otherwise, render the primitives inside and traverse the front-to-back children recursively

A conservative algorithm – why?

Hierarchical Visibility

- Scan conversion the octree faces can be expensive – cover a large number of pixels (overhead)
- How can we reduce the overhead?
- Goal: quickly conclude that a large polygon is hidden
- Method: use hierarchical z-buffer !

Hierarchischer Z-Buffer

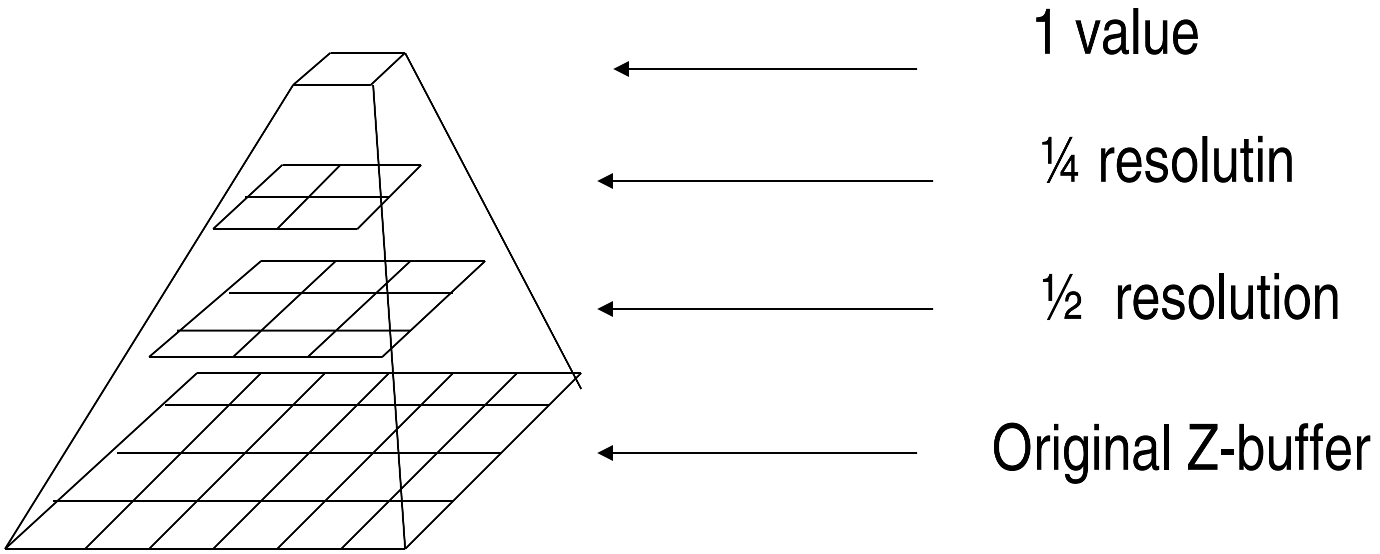


Reduktion der Anzahl der Pro-Pixel-Operationen

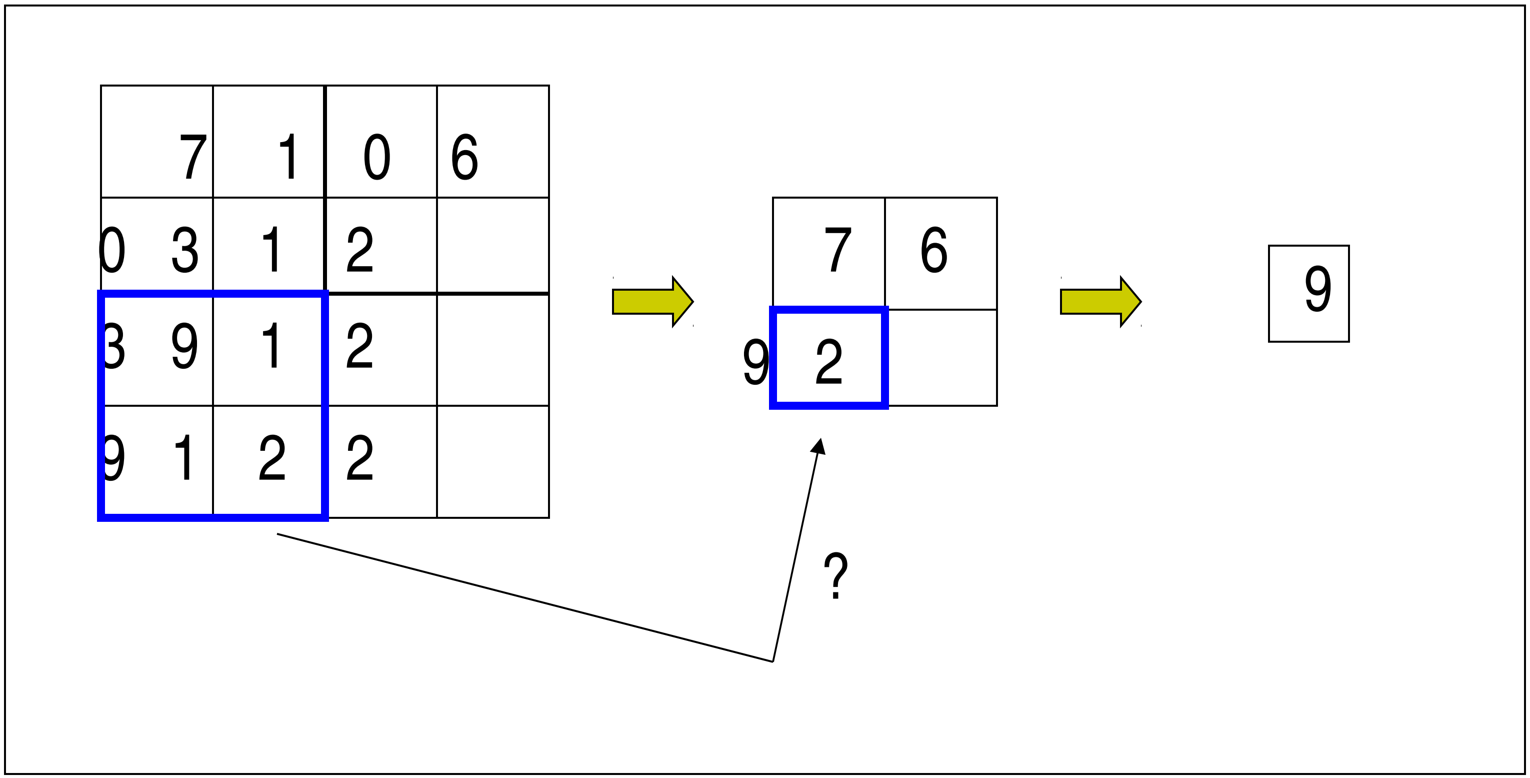
Hierarchical Z-buffer

An image-space approach

- Create a Z-pyramid

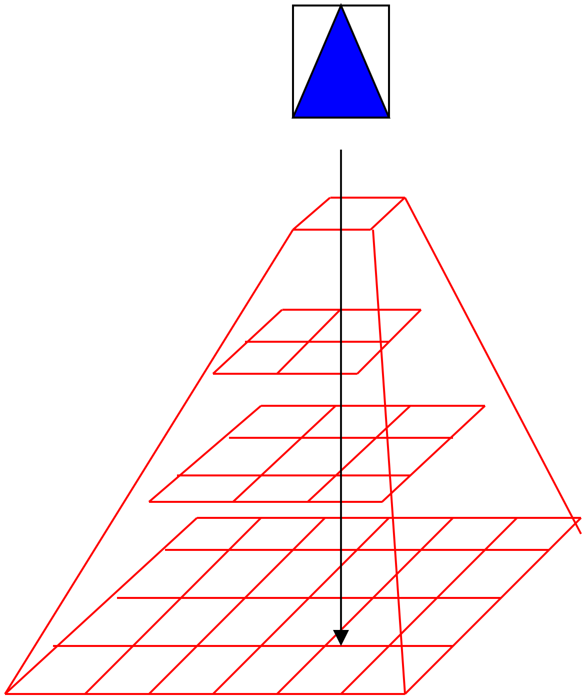


Hierarchical Z-buffer (2)



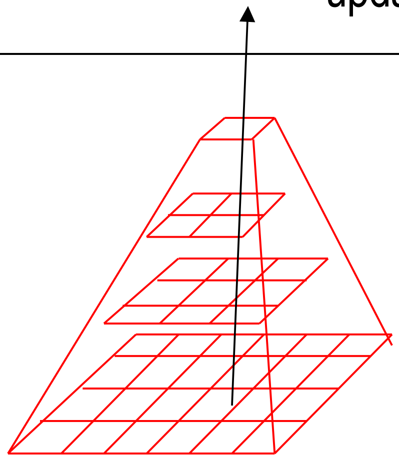
Hierarchical Z-buffer (3)

```
Isoccluded(g, Zp)
  near z = nearZ(BV(g))
  if (near Z behind Zp_root.z)
     return true
  else
    return ( Isoccluded(g,Zp.c[0]) &&
             Isoccluded(g,Zp.c[1]) &&
             Isoccluded(g,Zp.c[2]) &&
             Isoccluded(g,Zp.c[3])
           )
  end
```



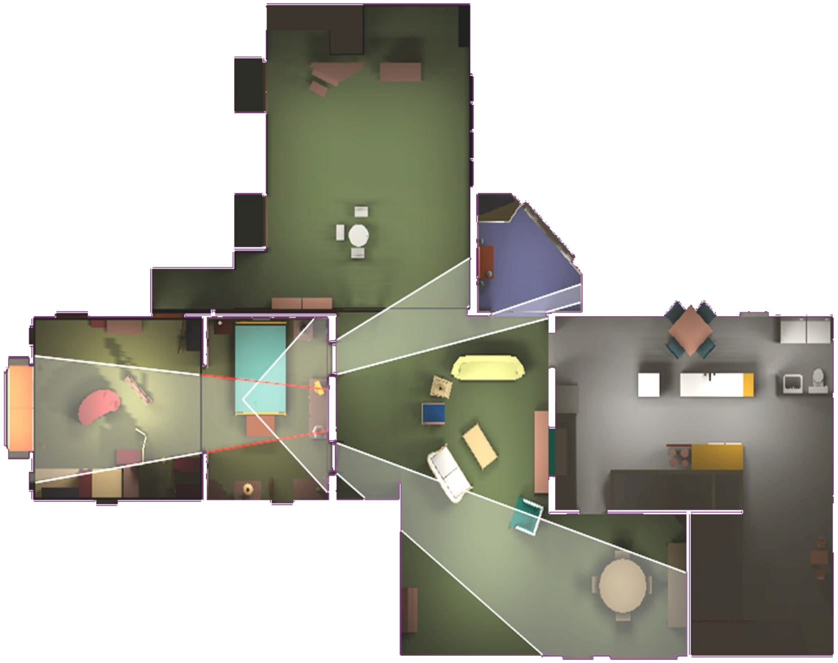
Hierarchical Z-buffer (4)

```
Cull_or_Render (OctreeNode N)
    if (isOccluded (N, Zp) then return;
    for each primitive p in N
        render and update Zp
    end
    for each child node C of N in front-to-back order
        Cull_or_Render( C )
    end
```



Portal Culling

- The following slides are taken from Prof. David Luebke's class web site at U. of Virginia



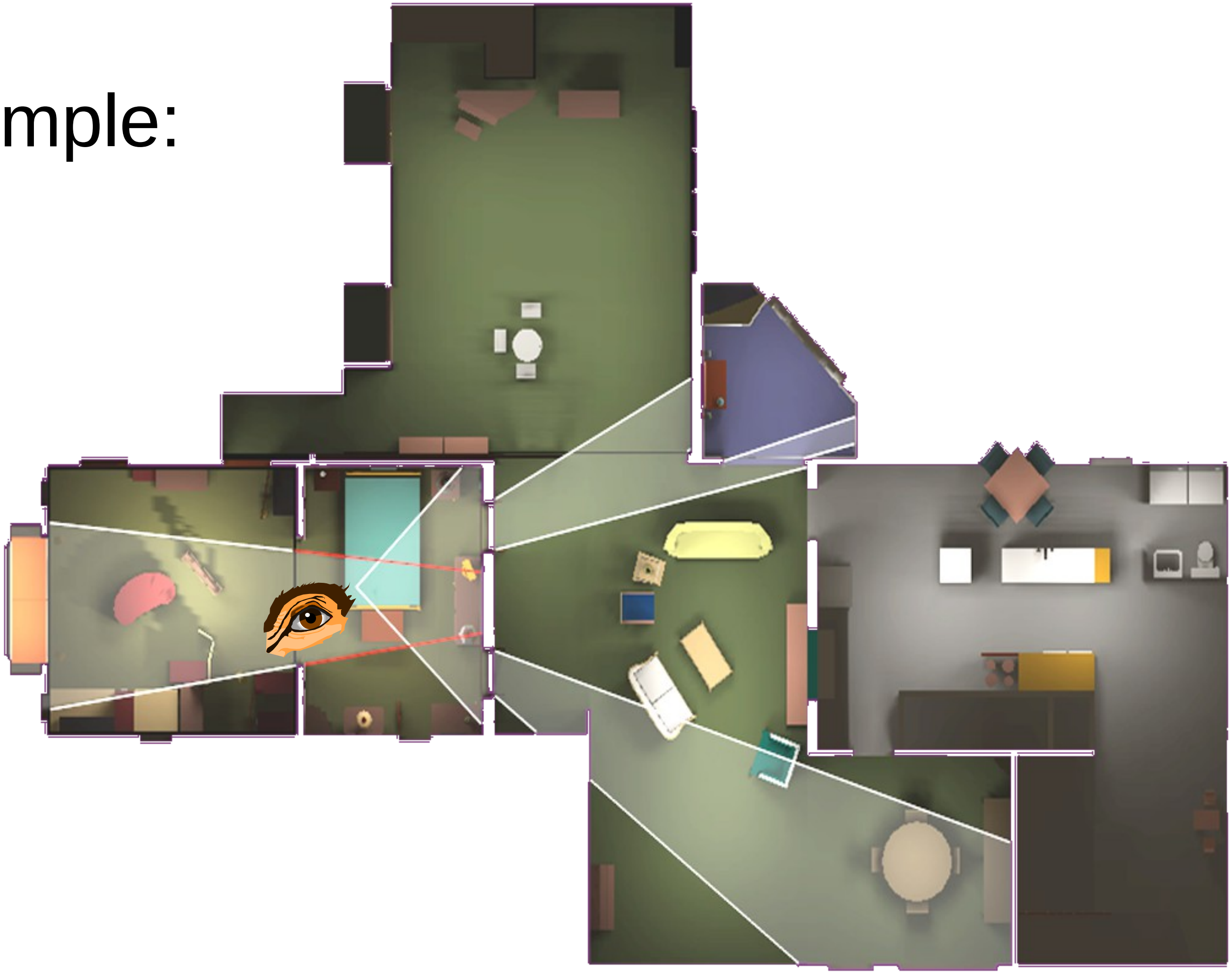
Portal Culling

- 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

Cells & Portals

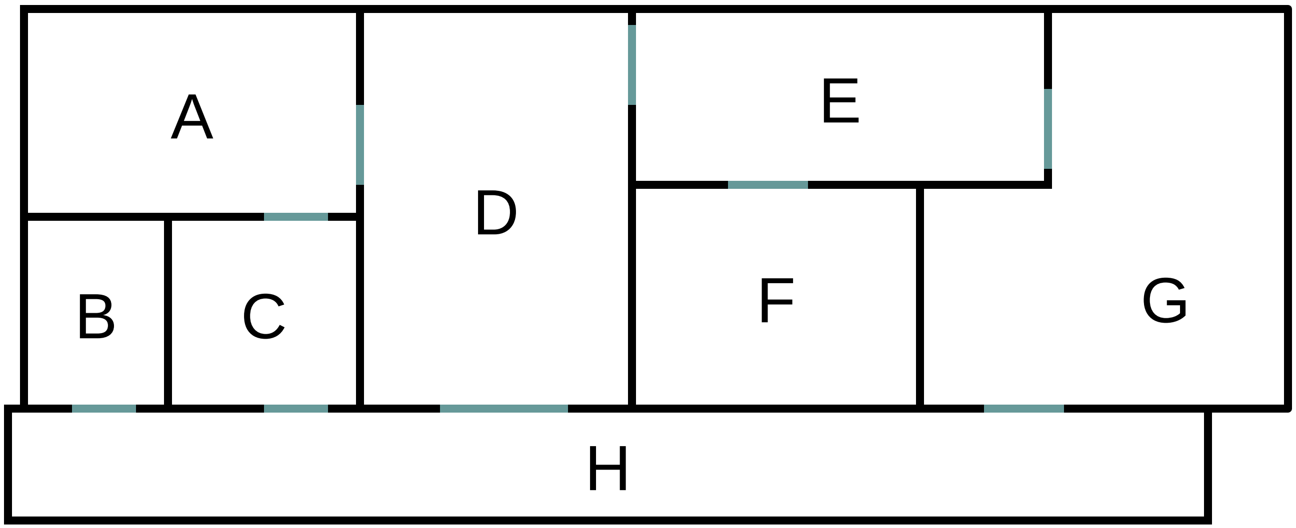
- An example:

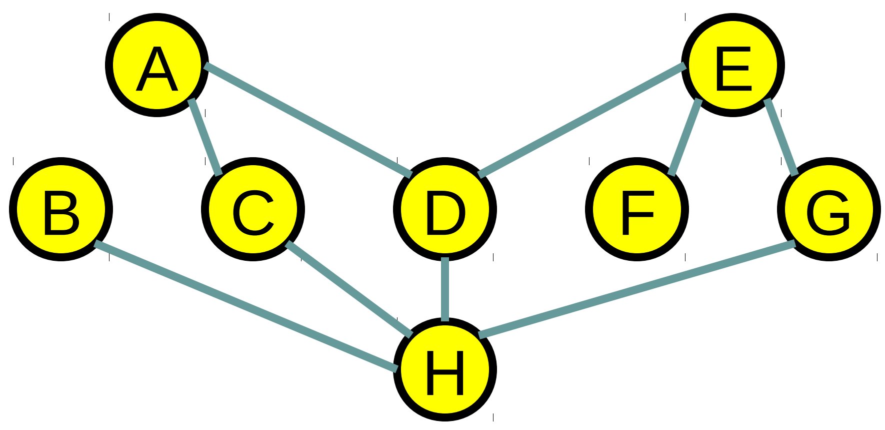


Cells & Portals

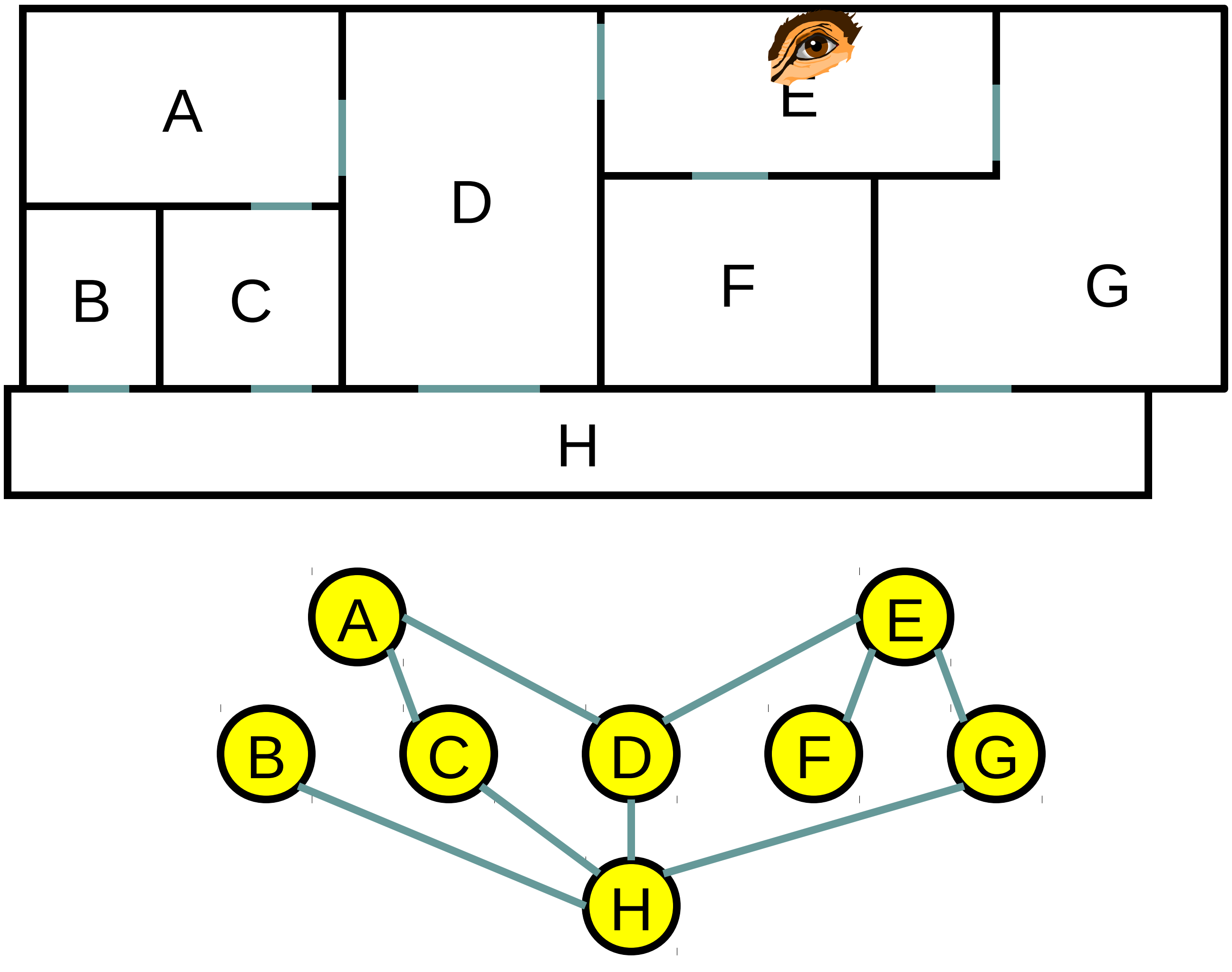
- Idea:
  - Create an *adjacency graph* of cells
  - Starting with cell containing eyepoint, traverse graph, rendering visible cells
  - A cell is only visible if it can be seen through a sequence of portals
    - So cell visibility reduces to testing portal sequences for a *line of sight…*

Cells & Portals

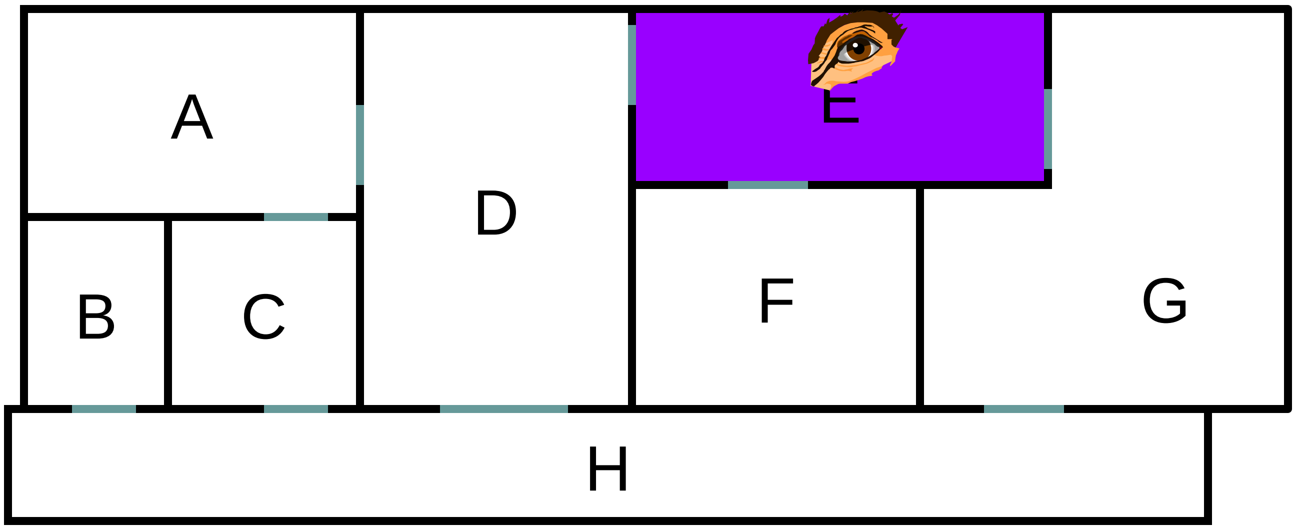


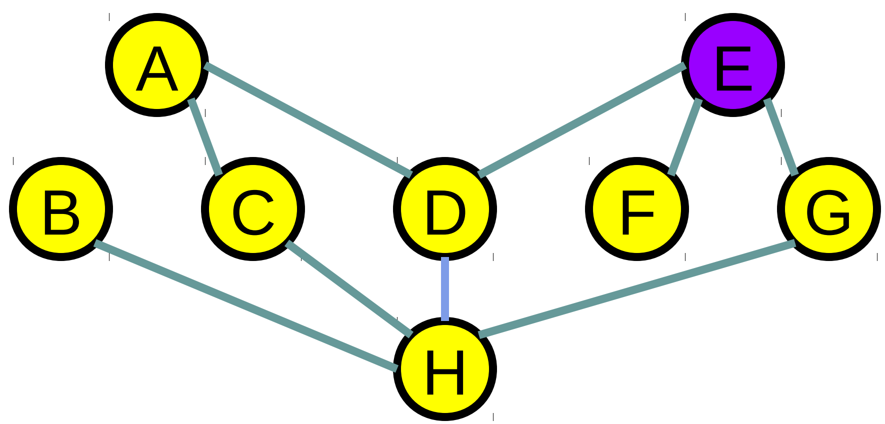


Cells & Portals

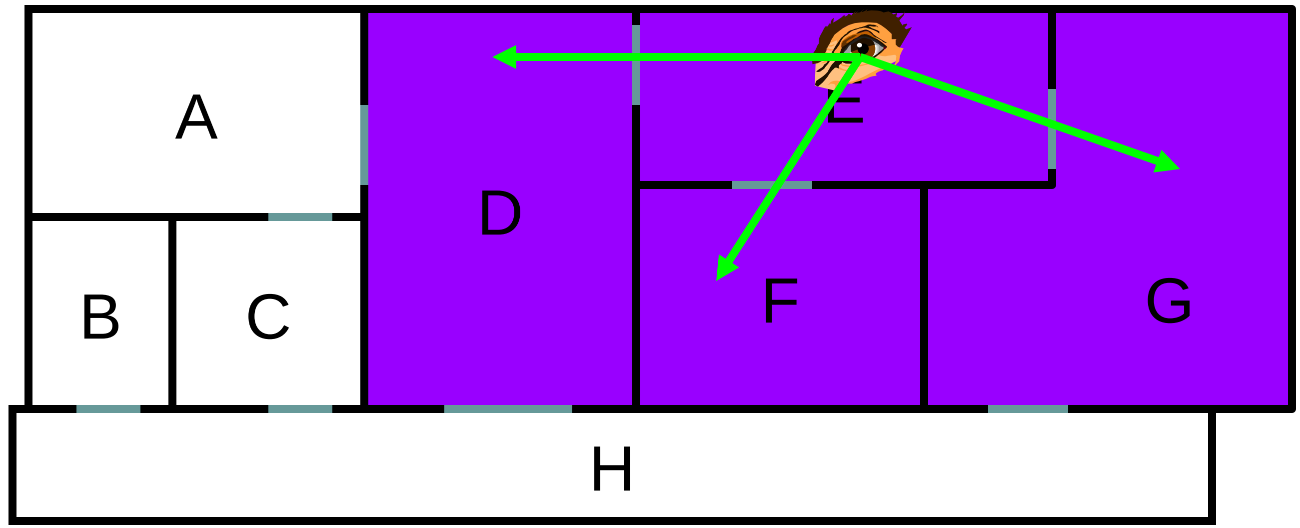


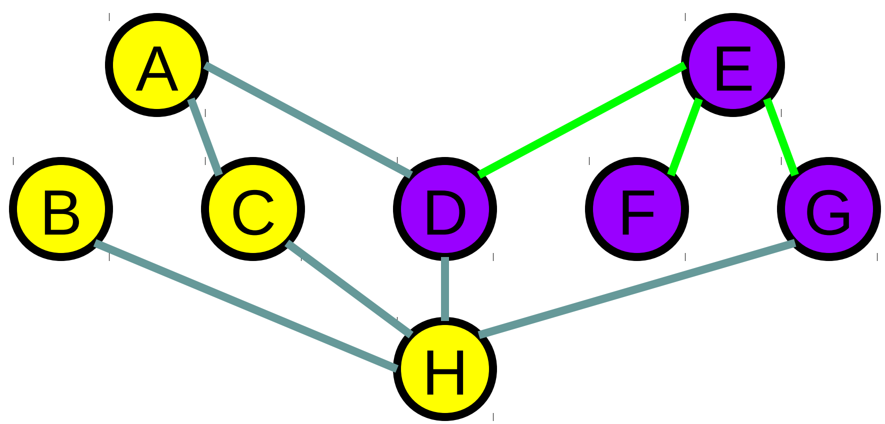
Cells & Portals



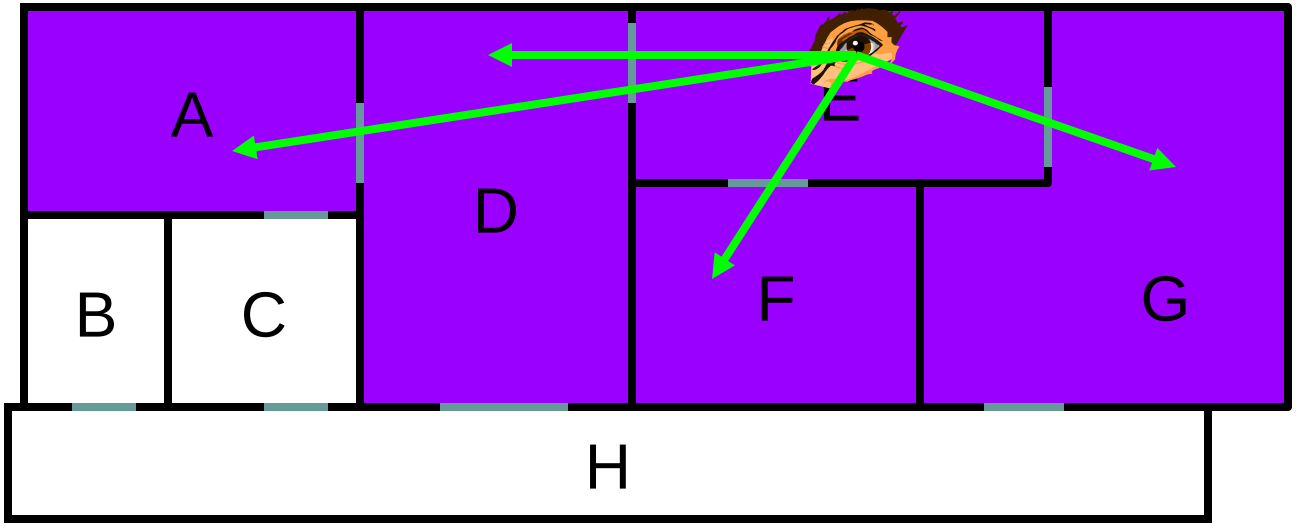


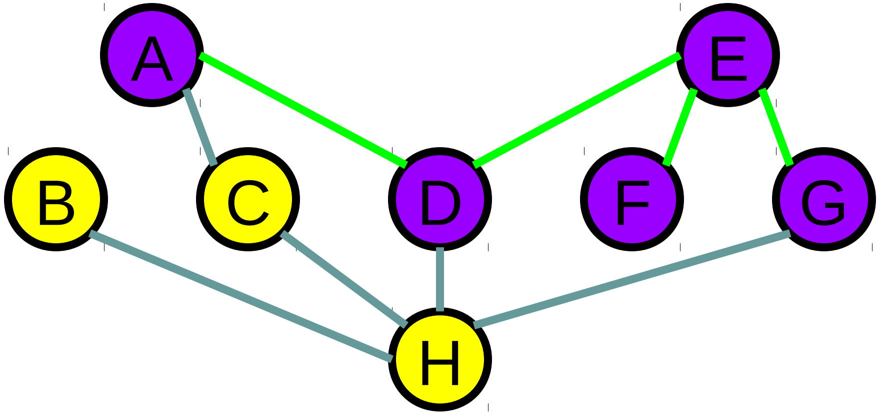
Cells & Portals



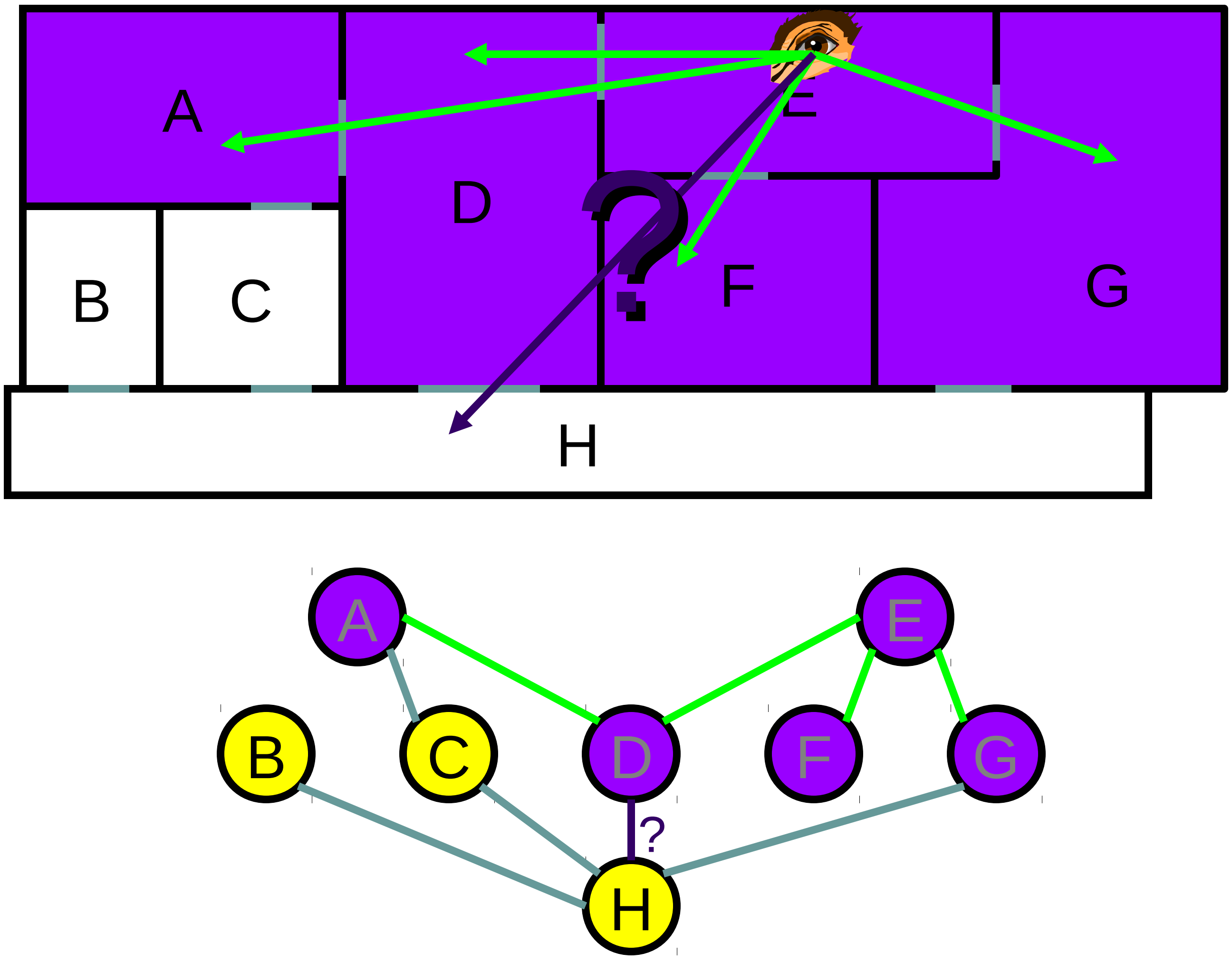


Cells & Portals

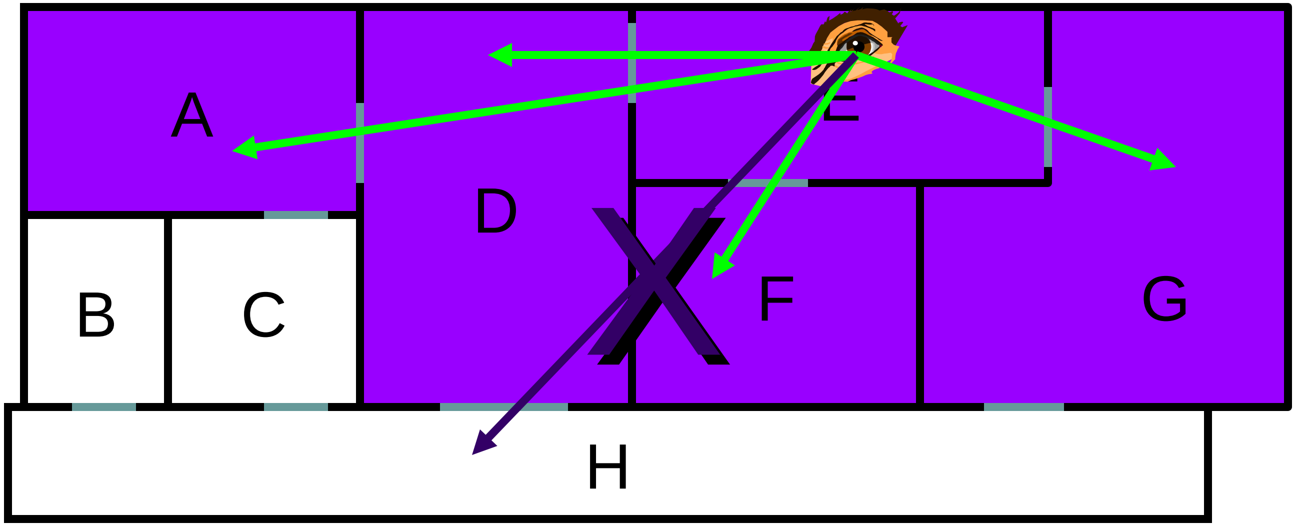


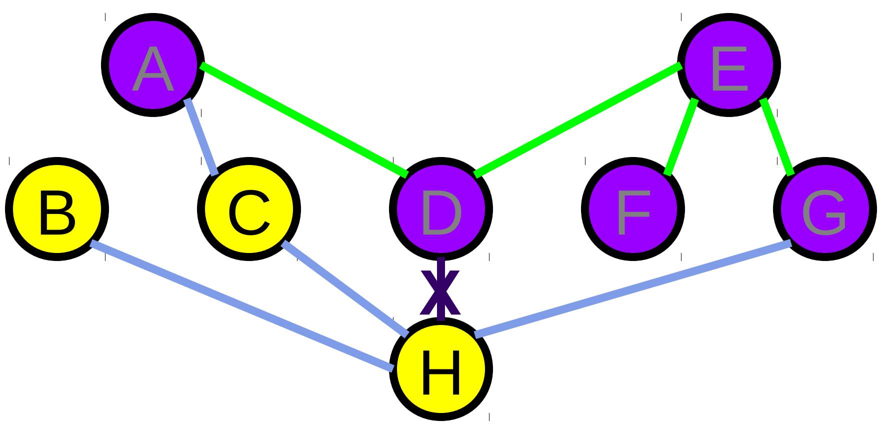


Cells & Portals



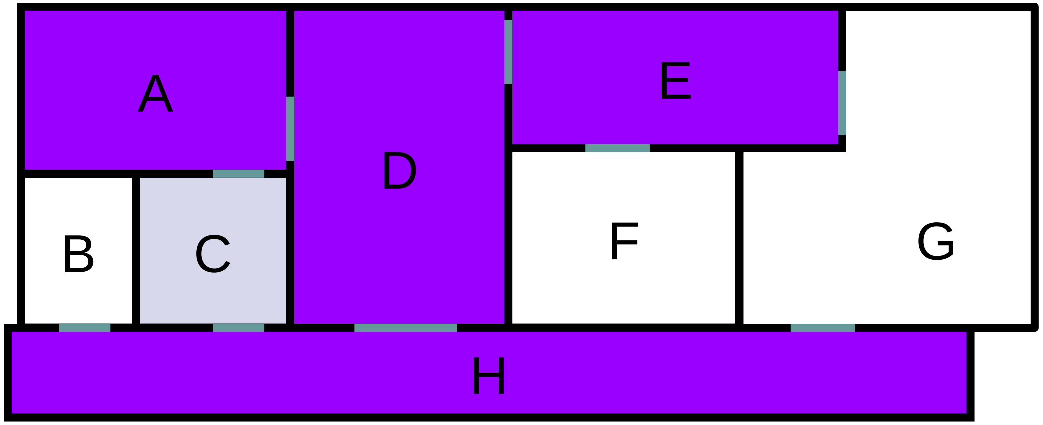
Cells & Portals





Cells & Portals

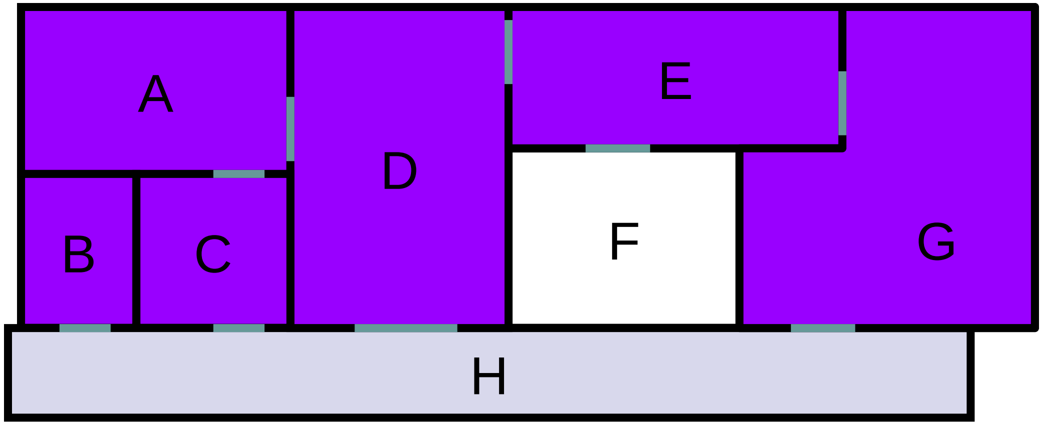
- *View-independent* solution: find all cells a particular cell could *possibly* see:



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

- Question: *How can we detect the cells that are visible from a given viewpoint?*
- Idea (textbook pp 366):
  - Set the view box (*P*) as the entire screen
  - Compare the portal (*B*) to the neighbor cell (*C*) against the current view box *P*
    - If *B* outside *P* – the neighbor cell *C* cannot be seen
    - Otherwise – the neighbor cell *C* is visible
      - New view box P = intersection of *P* and the portal *B*
      - For each neighbor of C, depth first traverse the adjacency graph of C and recurse

Example

- Text pp 367