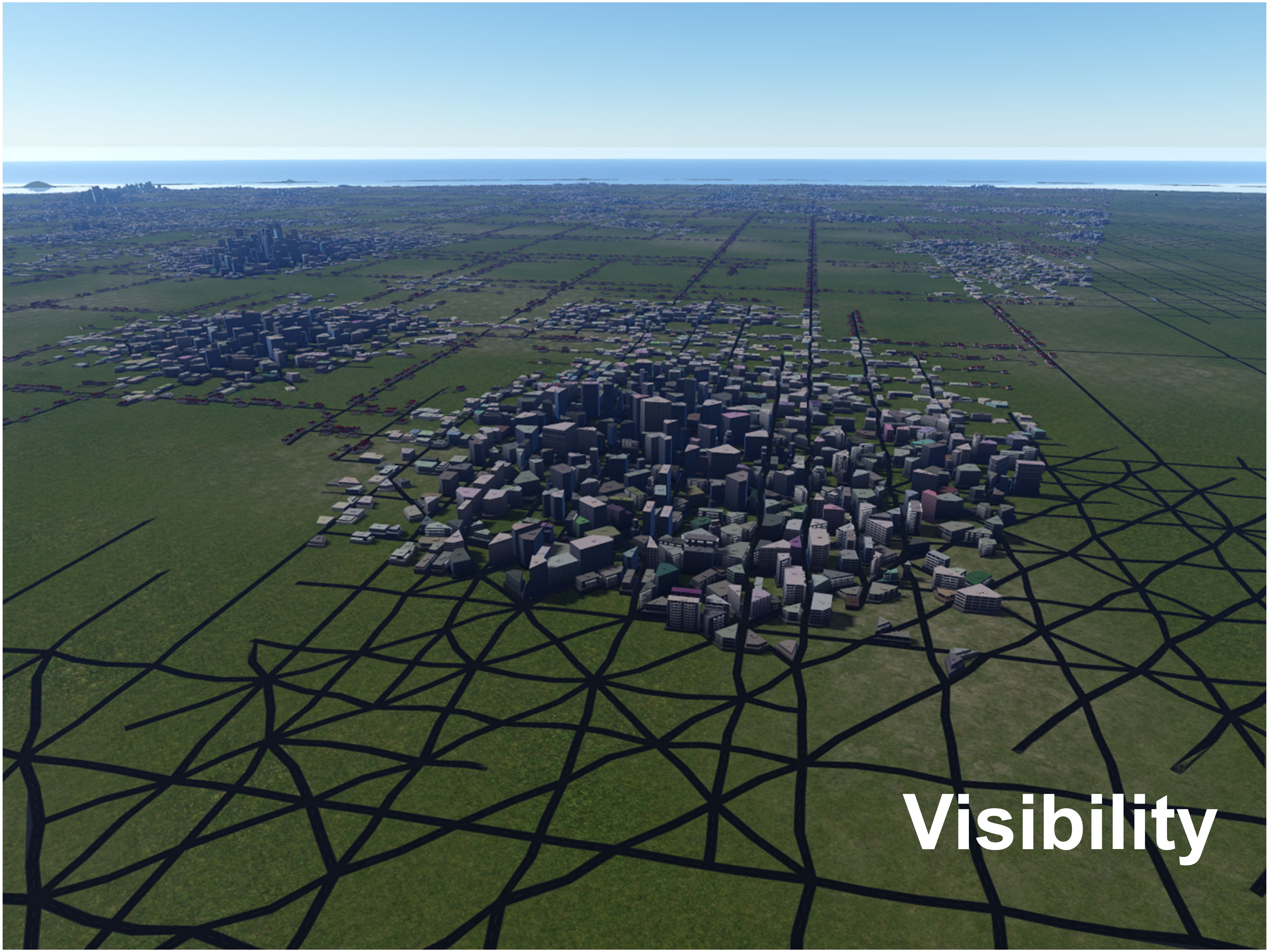

Visibility

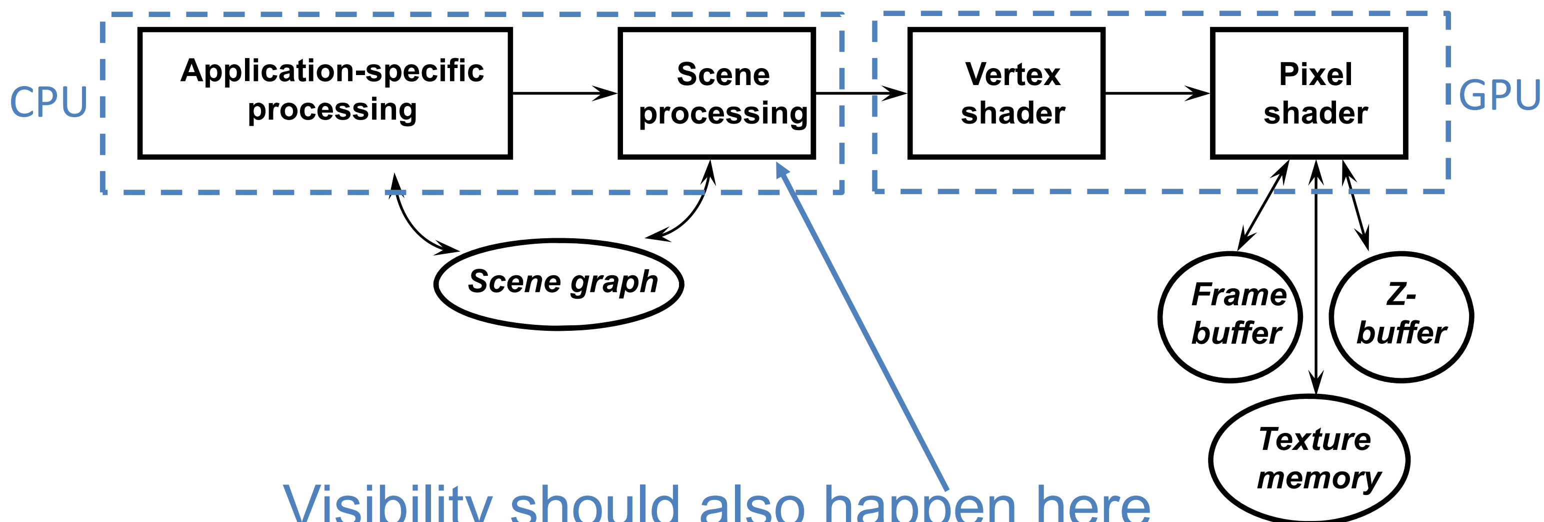
Recap: Depth Buffering

- Hardware to determine per-pixel visibility
  - 2D buffer for storing z-values per pixel
  - Pixel shader result only stored if z-value test passed
  - Allows drawing of unsorted geometry
- Sorting still greatly improves performance
  - *Early-z testing*: run depth test *before* pixel shader
  - Drawing objects close to camera early increases chances that a later pixel shader is not called

Why is the Z-Buffer Not Enough?

(Even with depth sorting:) Z-buffer…

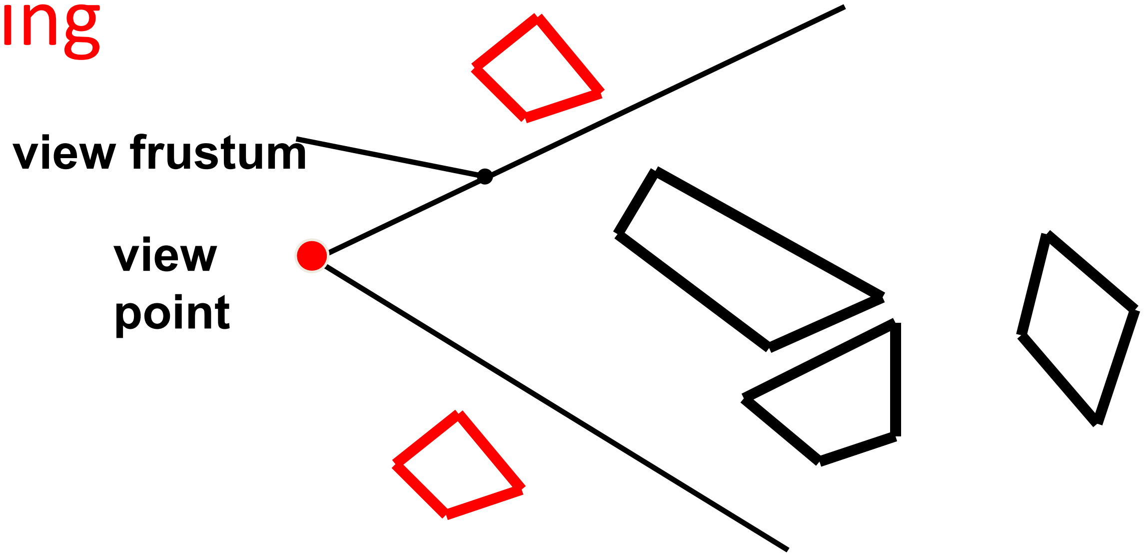
- …Does not eliminate depth-complexity (overdraw)
- …Does not eliminate vertex processing of occluded polygons

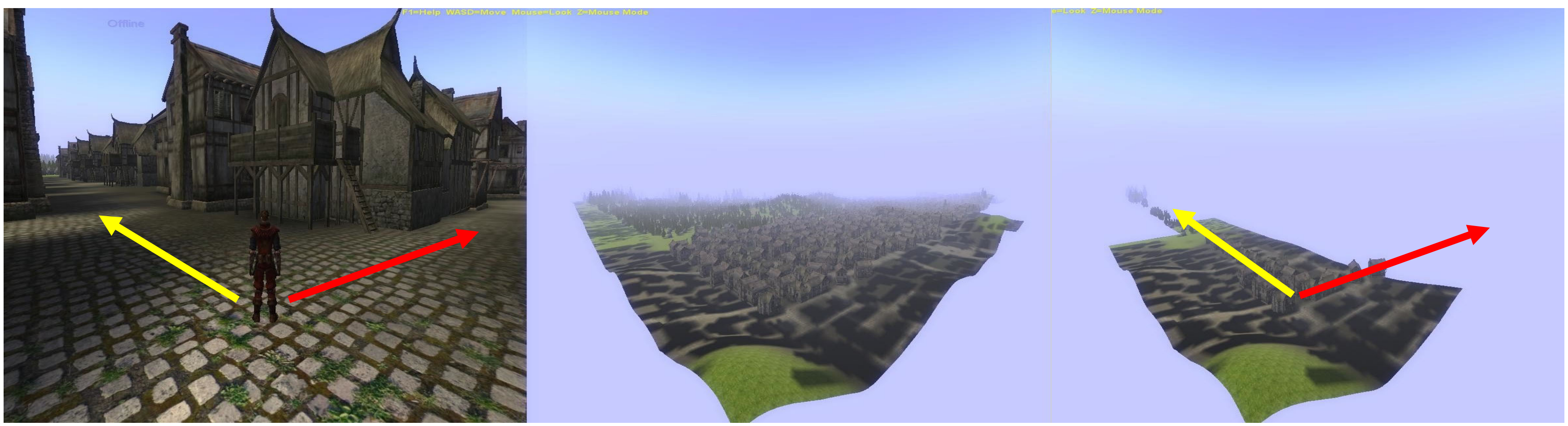


Visibility should also happen here
→ we need occlusion culling before vertex shader

View Frustum Culling

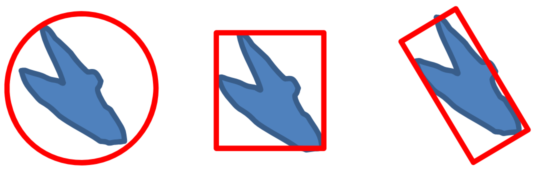
- View-frustum culling
- Occlusion culling
- Backface culling



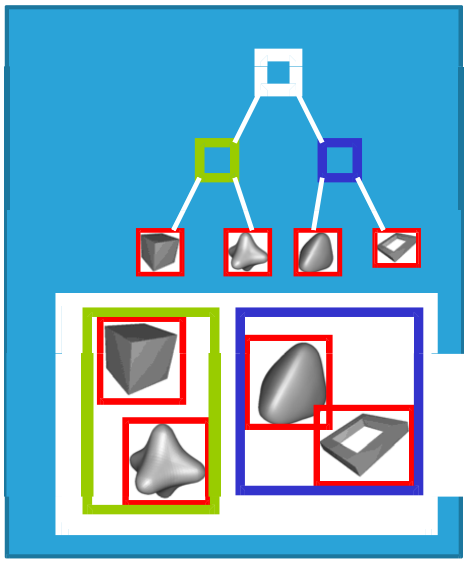


Bounding Volume Hierarchy

- Most important spatial data structure
- Common bounding volume types
  - Spheres
  - Axis-aligned bounding box (AABB)
  - Oriented bounding box (OBB)



- Encloses the bounded object
- Hierarchical data structure
  - Tree (binary or *n*-ary)
  - Leaves: store objects
  - Interior nodes: stores bounding volume enclosing all contained bounding volumes

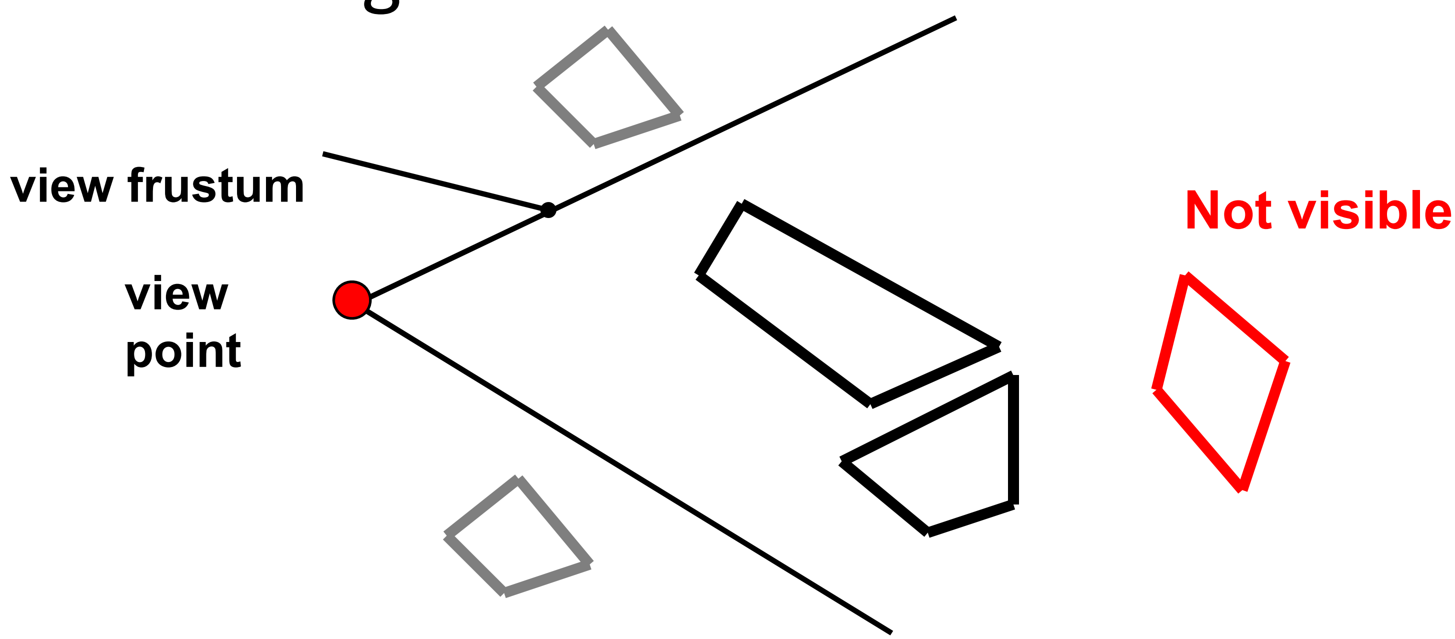


Frustum Culling with BVH

```
Function Cull(node)
{
   if not (intersect(node, frustum) = EMPTY)
      if node = LEAF then draw(node)
      else for all children C of node
         Cull(C)
}
```

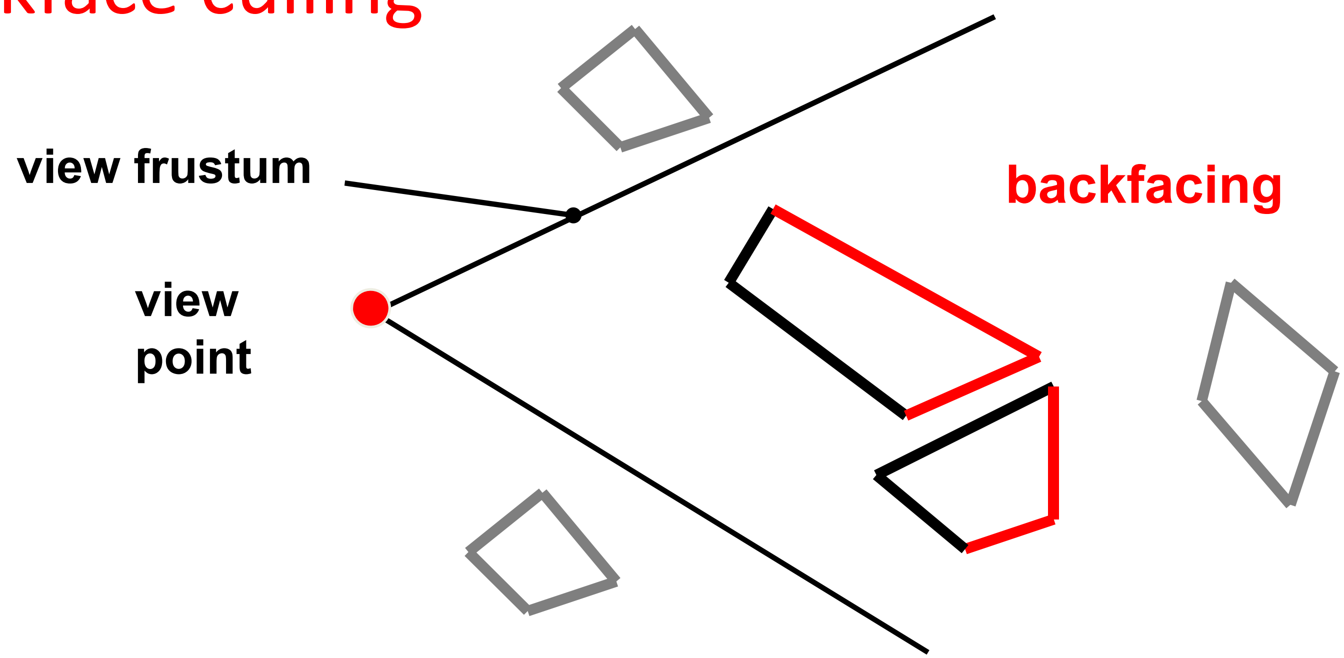
Occlusion Culling

- View-frustum culling
- Occlusion culling
- Backface culling



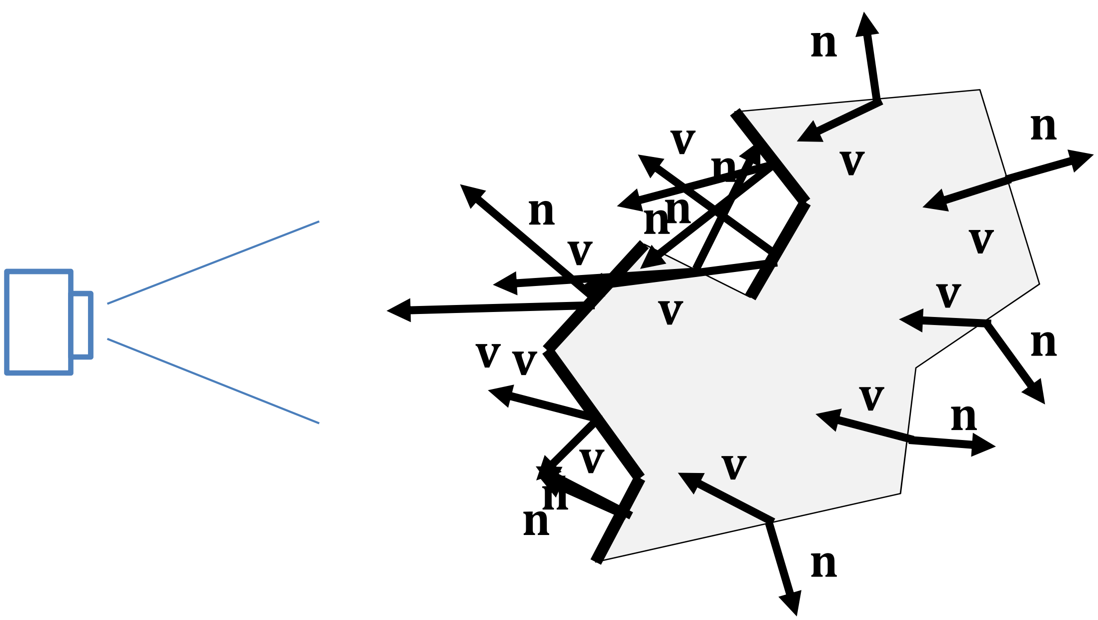
Backface Culling

- View-frustum culling
- Occlusion culling
- Backface culling



Backface Culling

- If an object is watertight, we cannot see the interior
- We only must draw those primitives facing the camera
- Can save up to 50% of primitives on average
- Simple to implement using dot product of normal $\mathbf{n}$ + view vector $\mathbf{v}$

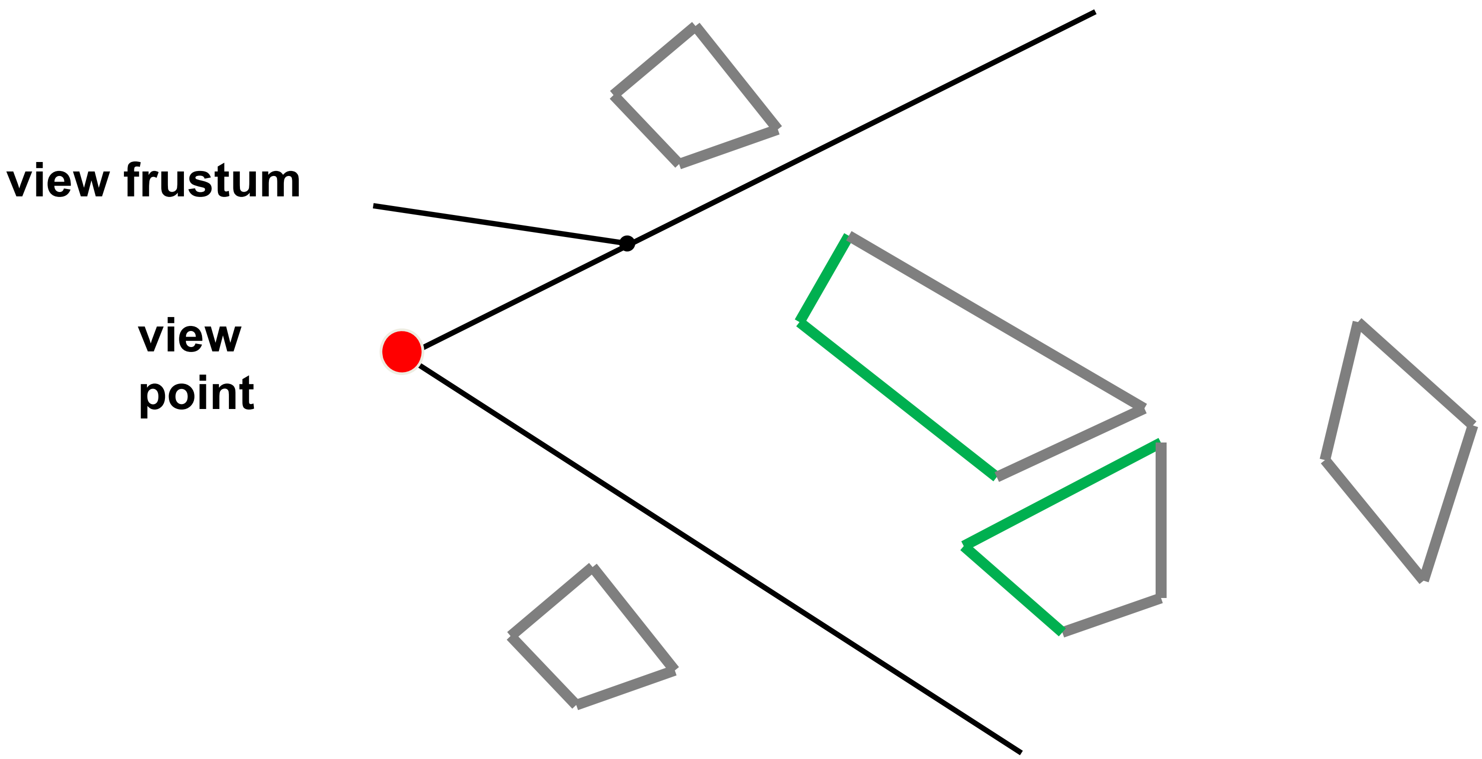


$\mathbf{n} \cdot \mathbf{v} > \mathbf{0}$
→ must draw

$\mathbf{n} \cdot \mathbf{v} < \mathbf{0}$
→ not visible

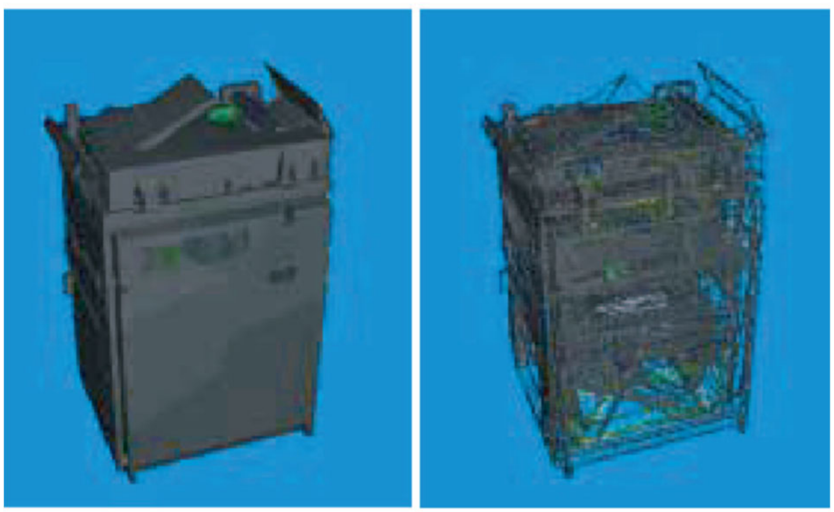
Visibility Culling Result

Result of frustum + occlusion + backface culling



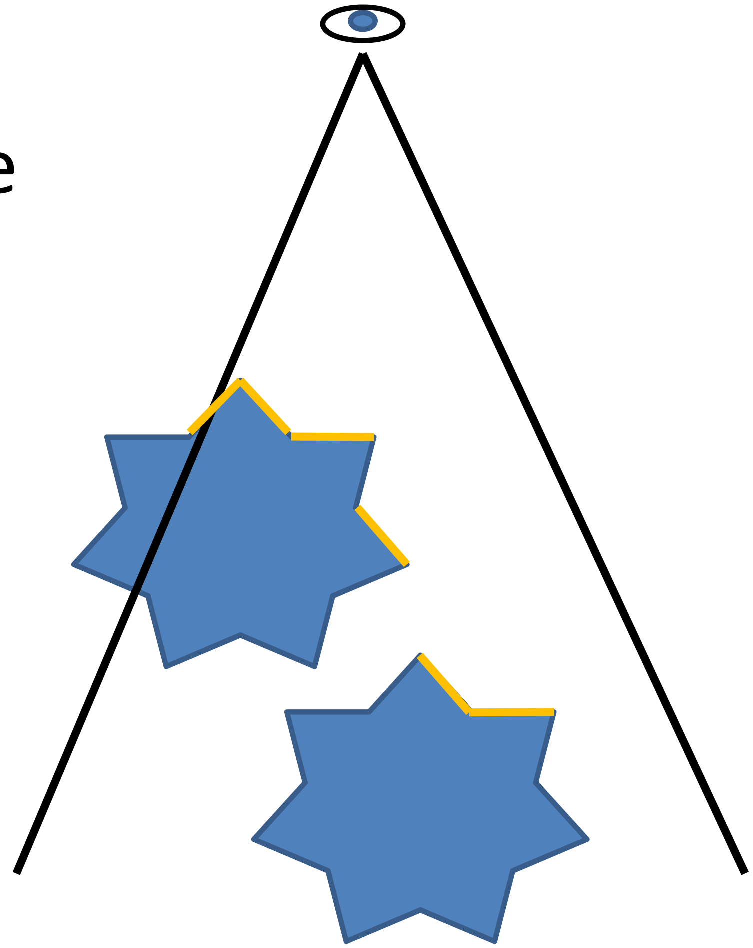
Exactly Visible Set

- All primitives that are visible
- No more, no less



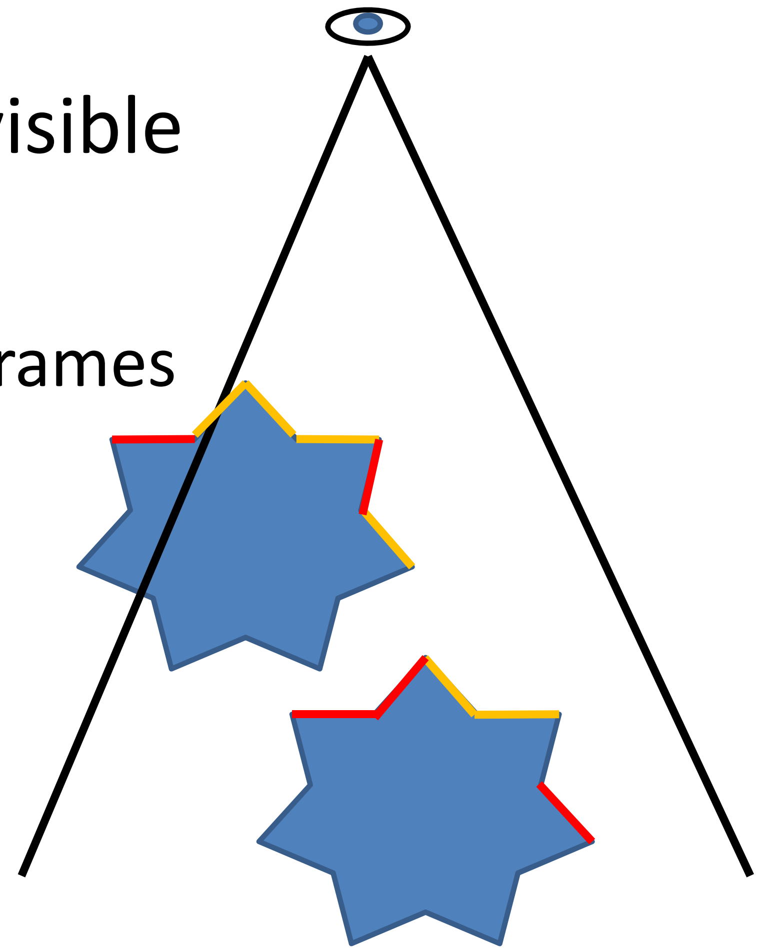
(a)

(b)



Potentially Visible Set

- PVS = all primitives that are visible
- And a bit more
  - E.g., visible within the next n frames
  - Exact hidden surface removal done later by z-buffer
- How much more?
- What do we want to do with the PVS?



PVS Classification

- The exact visible set (EVS) is unknown: O(N^9)…
- Potentially visible set (PVS) =
  set of objects that *could* be visible
- PVS can be
  - Aggressive, PVS $\subseteq$ EVS
  - Conservative, PVS $\supseteq$ EVS (preferred)
  - Approximate, PVS ~ EVS
- PVS can be precomputed
  - But we need discretize viewpoints into view *regions*

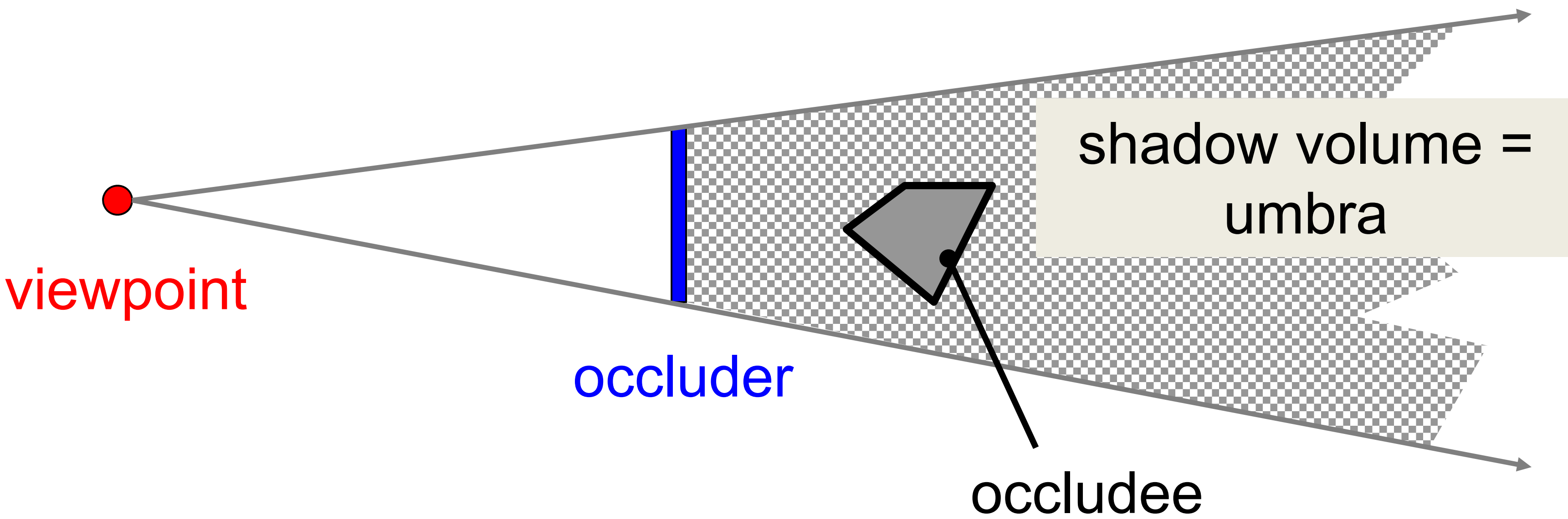
Naïve Occlusion Culling

- Select some promising occluders
  - Large objects
  - Close to camera → large in screen space
- Test all other objects against occluders
  - Remove the objects that are occluded
  - Maybe speed up by testing bbox instead of object

- Why does it not work well for real-life scenes?

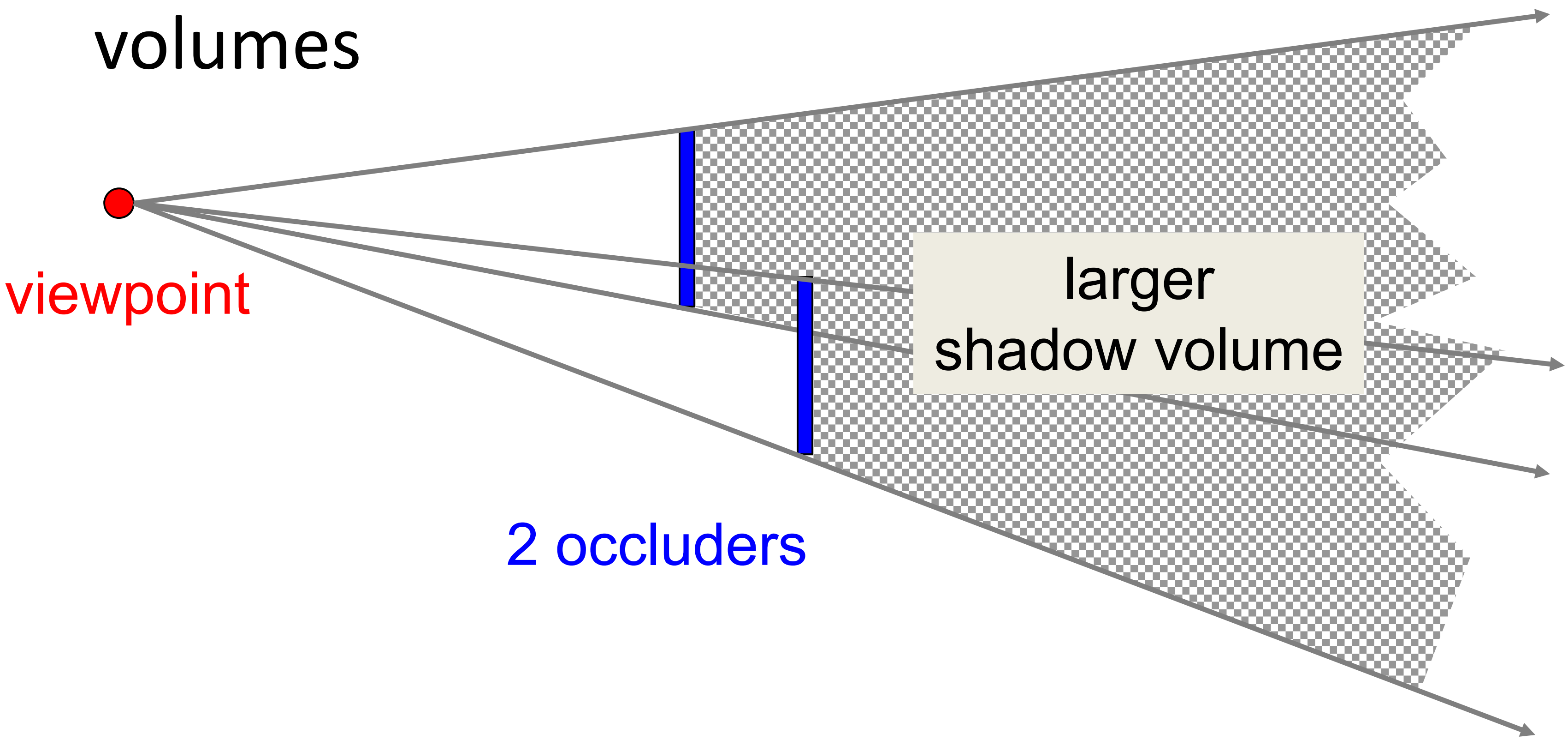
Visibility from a Point

- Terms: occluder, occludee, shadow volume, umbra



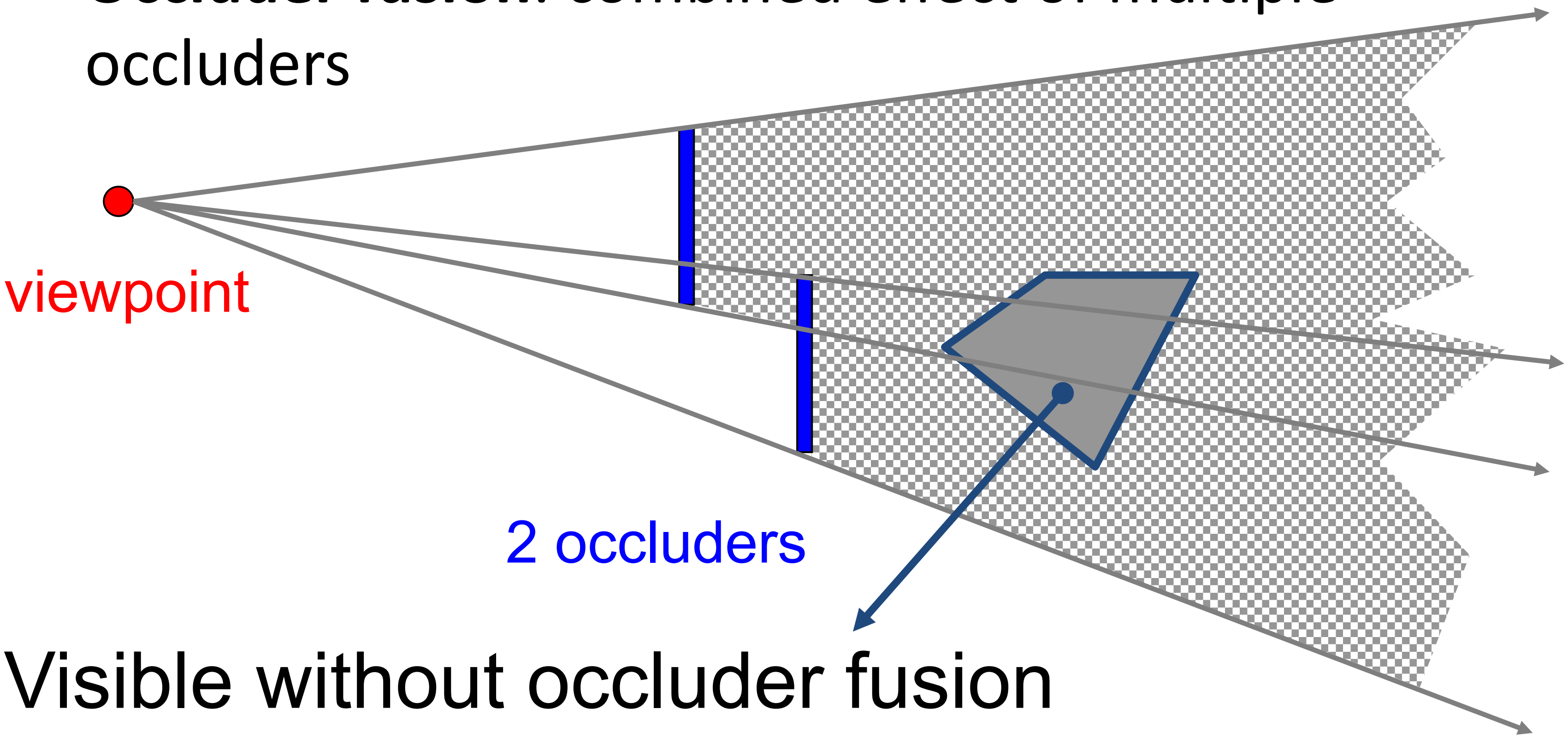
Visibility from a Point

- Complete shadow volume for occluders occ_1, …, occ_n = **union** of all individual shadow volumes



Occluder Fusion

- **Occluder fusion**: combined effect of multiple occluders



Visible without occluder fusion
Invisible with occluder fusion

Occlusion Culling in Practice

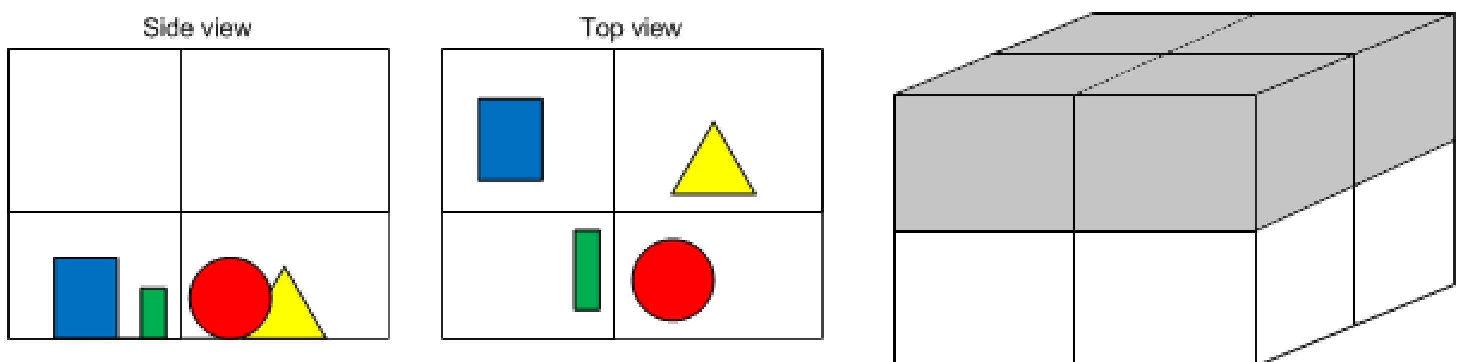
- Use two two spatial data structures
  1. **Scene data structure** (SDS)
     - Stores the objects in the scene
  2. **Shadow volume data structure** (SVDS)
     - Generated by occluders (selected from scene), or
     - Generated by virtual occluders (synthesized)
- Cull SDS using SVDS

Simple Algorithm for Point Visibility

- Shadow volume data structure (*SVDS*) = empty
- For each occluder occ_i
  - Calculate shadow volume SV_i
  - Add SV_i to *SVDS*
- For every object o_j
  - Test o_j against the *SVDS*
  - Cull o_j if occluded

Spatial Data Structures

- For quickly finding/culling large portions of scene with a single test
- All kinds of (hierarchical) data structures used
- E.g., bounding boxes, bounding volume hierarchy, grids, quadtree, octree, k-d tree, BSP tree

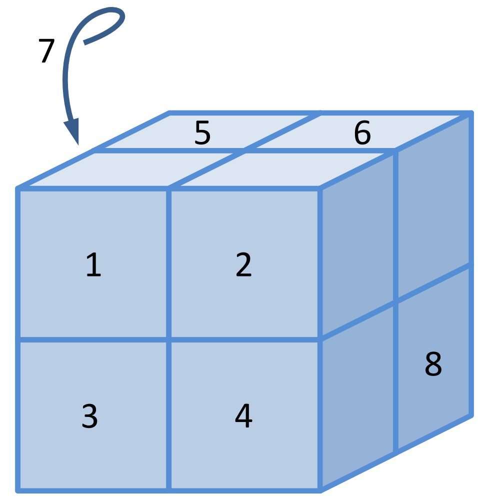


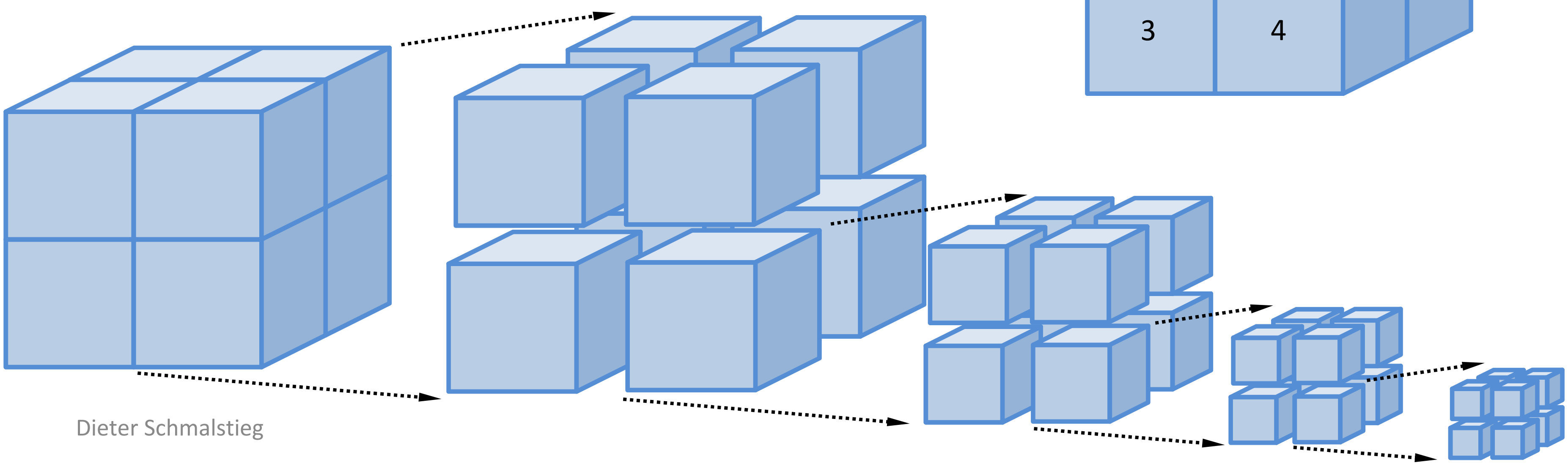
Simple 3D grid

Example: Octree

- 3D equivalent of quadtree
- Hierarchical subdivision of a cube into 8 octants

Region in 3D space





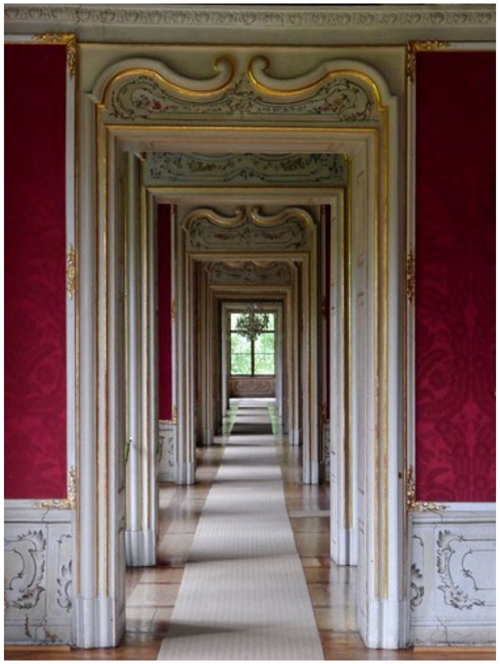
Simple Algorithm, Revisited

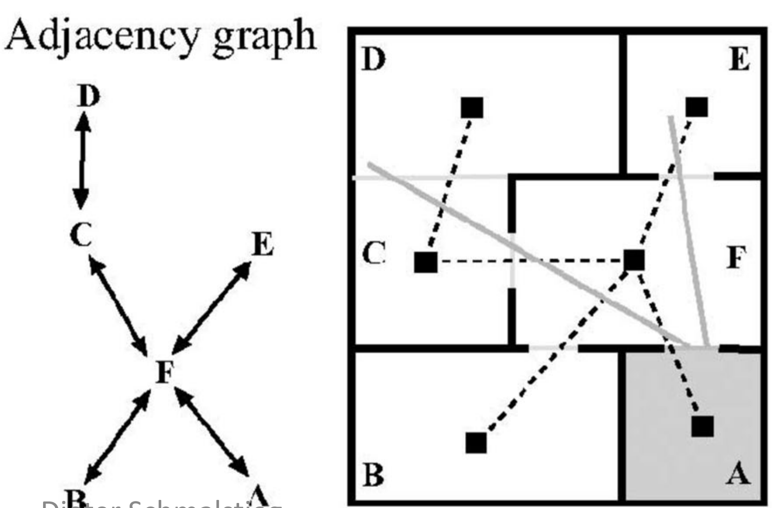
- Shadow volume data structure (*SVDS*) = empty
- For each occluder occ_i
  - Calculate shadow volume SV_i
  - Add SV_i to *SVDS*
- For every object o_j
  - Test o_j against the *SVDS*
  - Cull o_j if occluded

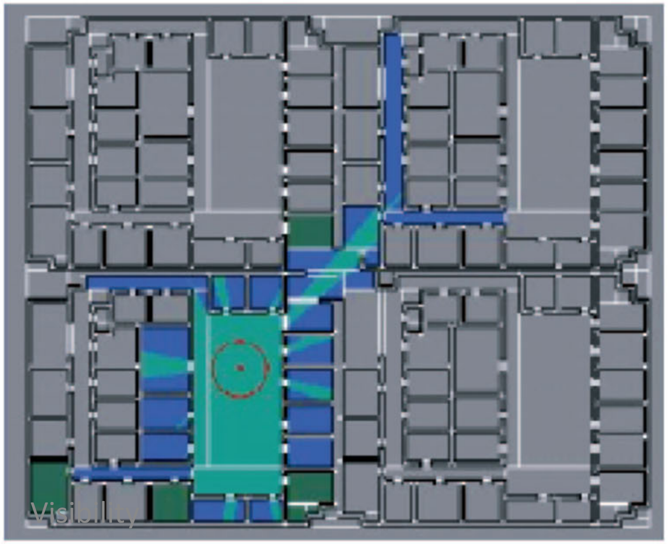
What shall we use for SVDS?

Cells and Portals

- Indoors
  - Most rooms (*cells*) occluded by walls
  - Store *portals* (windows, doors) instead of occluders (walls)
- Cells and portals form nodes and edges of a *portal graph*

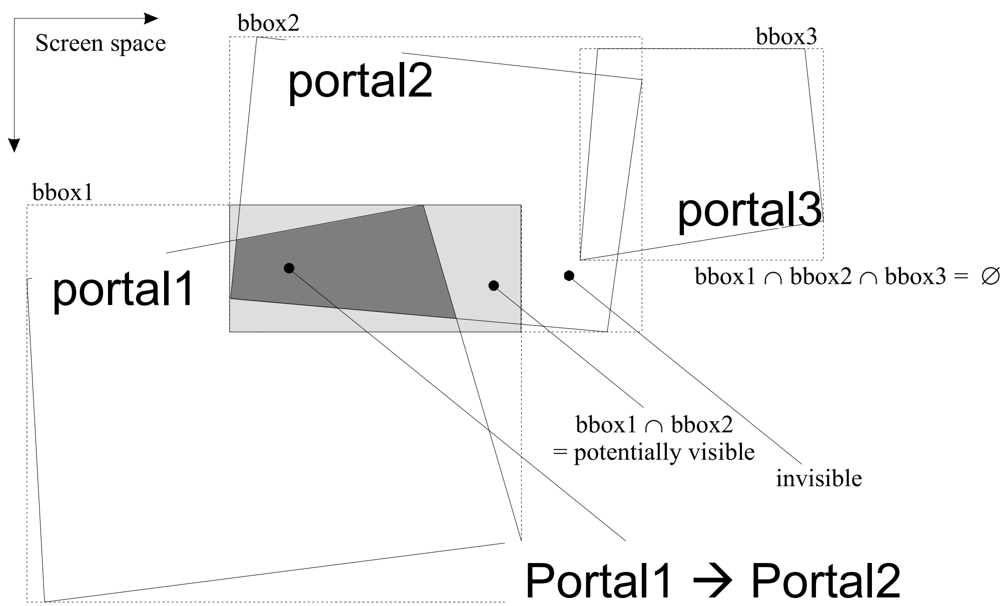






Screen-Space Portals

- In runtime, find the portals of the current cell that are in the frustum
- Traverse through all found portals to the adjacent cells and find all portals that are visible to the camera through the original portal



Regular Depth Buffer

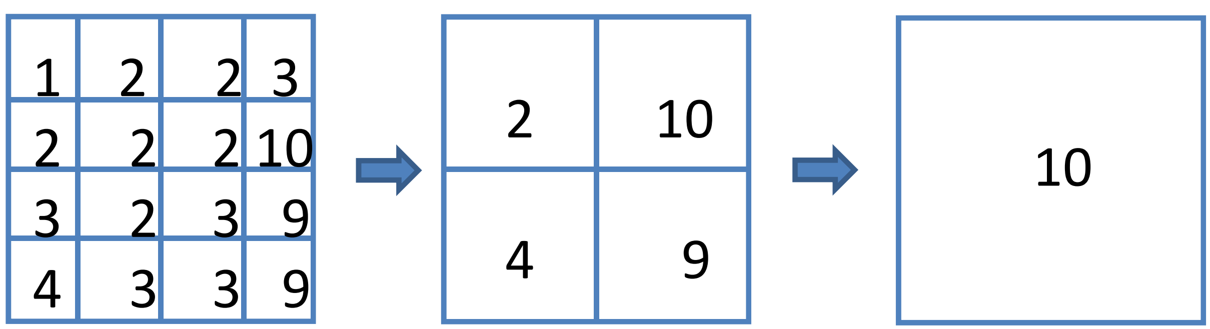
- *Depth Prepass* in deferred rendering
  - Pass 1: Rasterize the geometry,
    only write depth + object id to G-buffer
  - Pass 2: Read + collect all visible ids from G-buffer
- Can be expensive
  - For large scenes with many primitives
  - For high framebuffer resolutions

Virtual Occluders

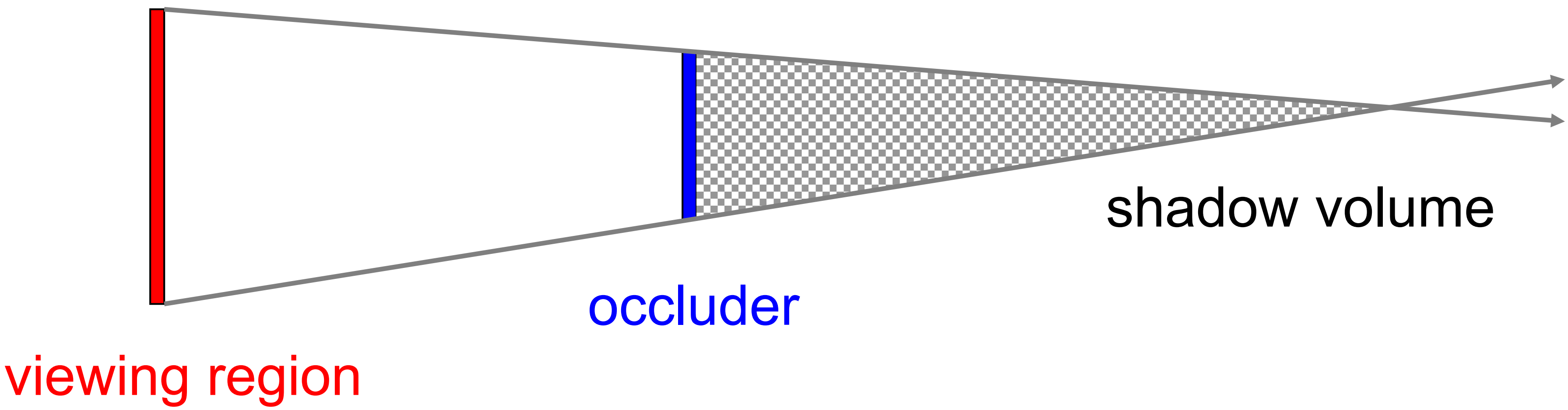
- Expensive to use objects with many primitives as occluders
- Cannot use *bounding* volumes of objects, since this is not a conservative test
- But we can use *bounded* volumes completely contained inside objects as *virtual occluders*
- Virtual occluders can be simple boxes

Hierarchical Depth Buffer

- Regular depth buffer can be expensive for high resolution
- Replace depth buffer with a depth *pyramid*
  - Bottom of the pyramid: full-resolution depth buffer
  - Higher levels: smaller resolution depth buffers, where a pixel represents max. z-value of group of pixels on lower level
- Hierarchically rasterize polygon, starting from highest level
  - If polygon is further than recorded pixel, early exit
  - If polygon is closer, hierarchically test lower levels
  - If bottom of pyramid is reached and polygon still closer, propagate the value up the pyramid



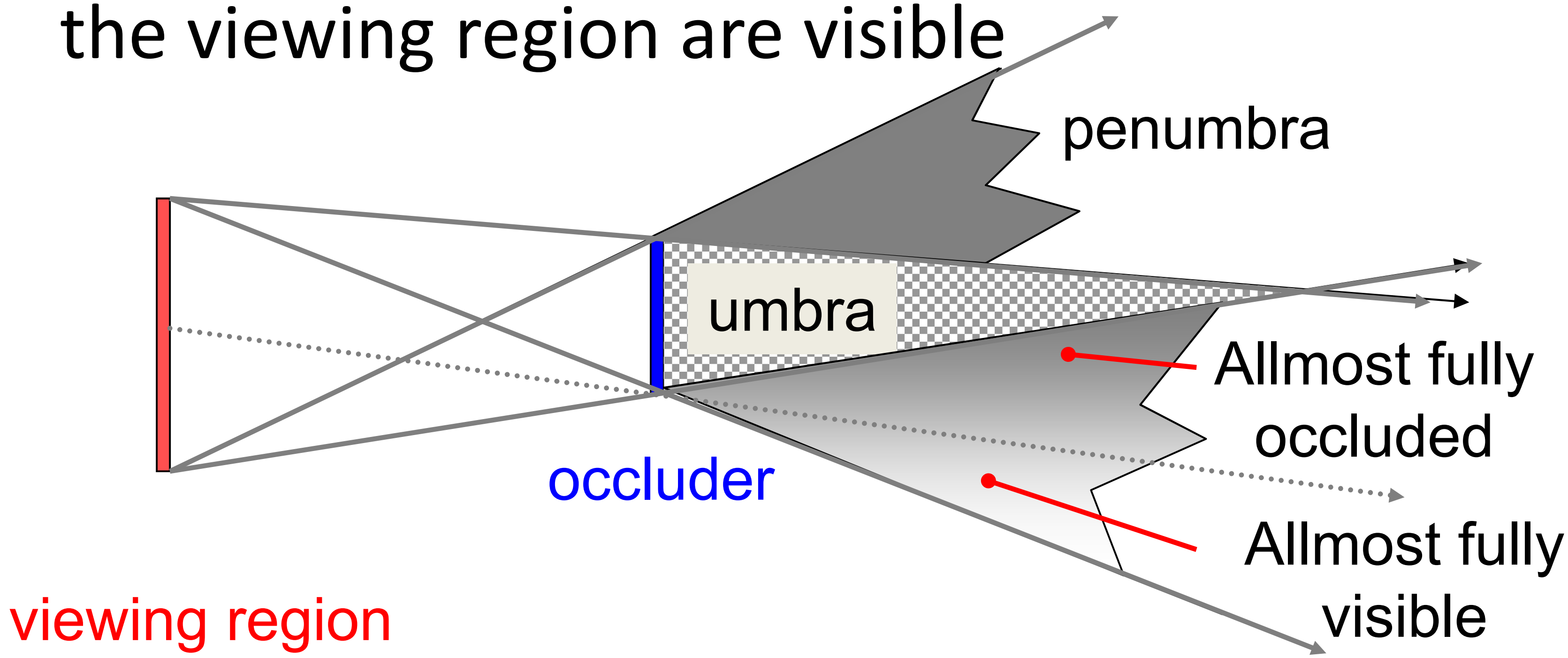
Visibility from a Region (Example in 2D)



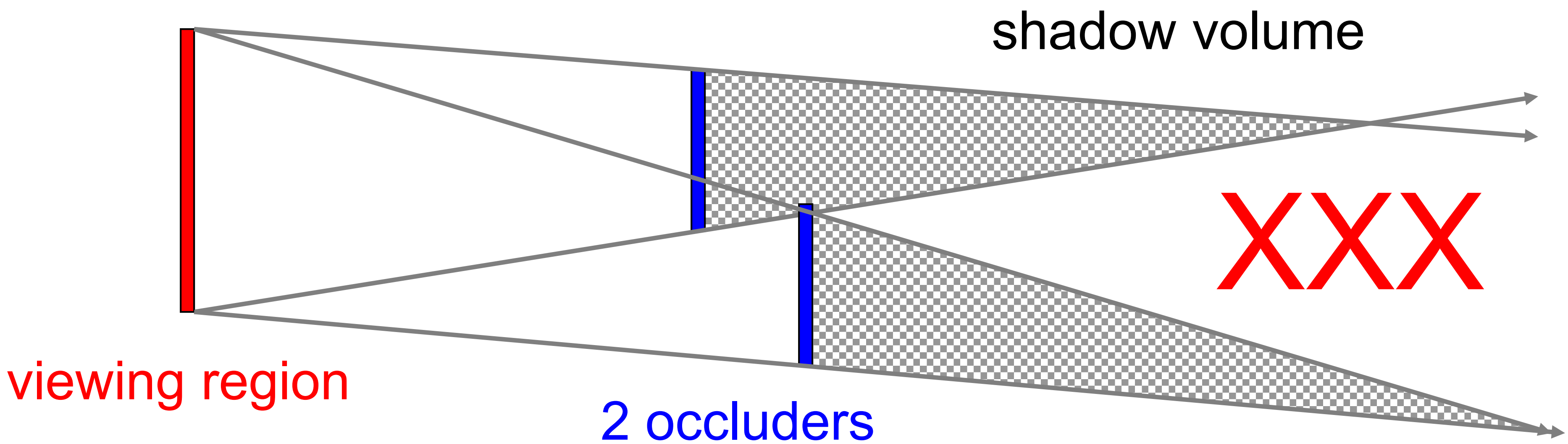
- If we want to precompute visibility, we must discretize viewpoints into view regions
- Only works for static scenes

Umbra and Penumbra

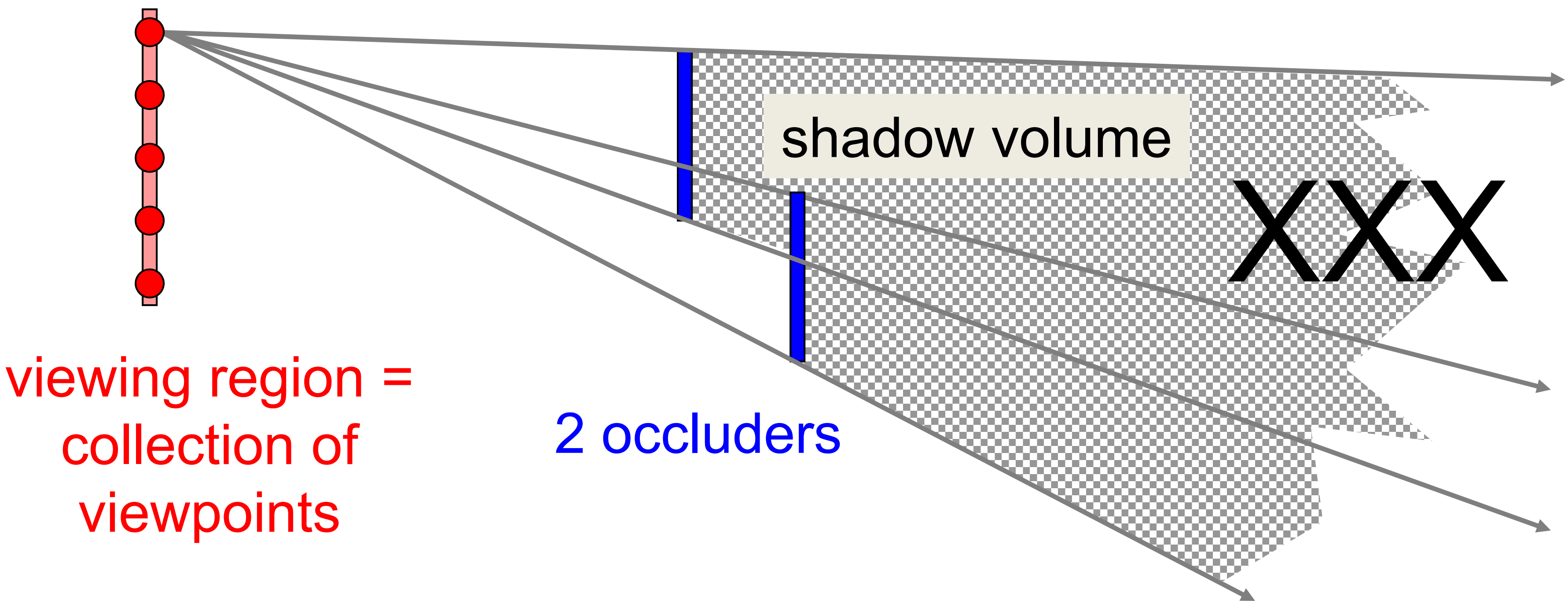
- **Umbra** = full shadow, penumbra = half-shadow
- **Umbra** is a simple in/out classification
- **Penumbra** additionally encodes which parts of the viewing region are visible



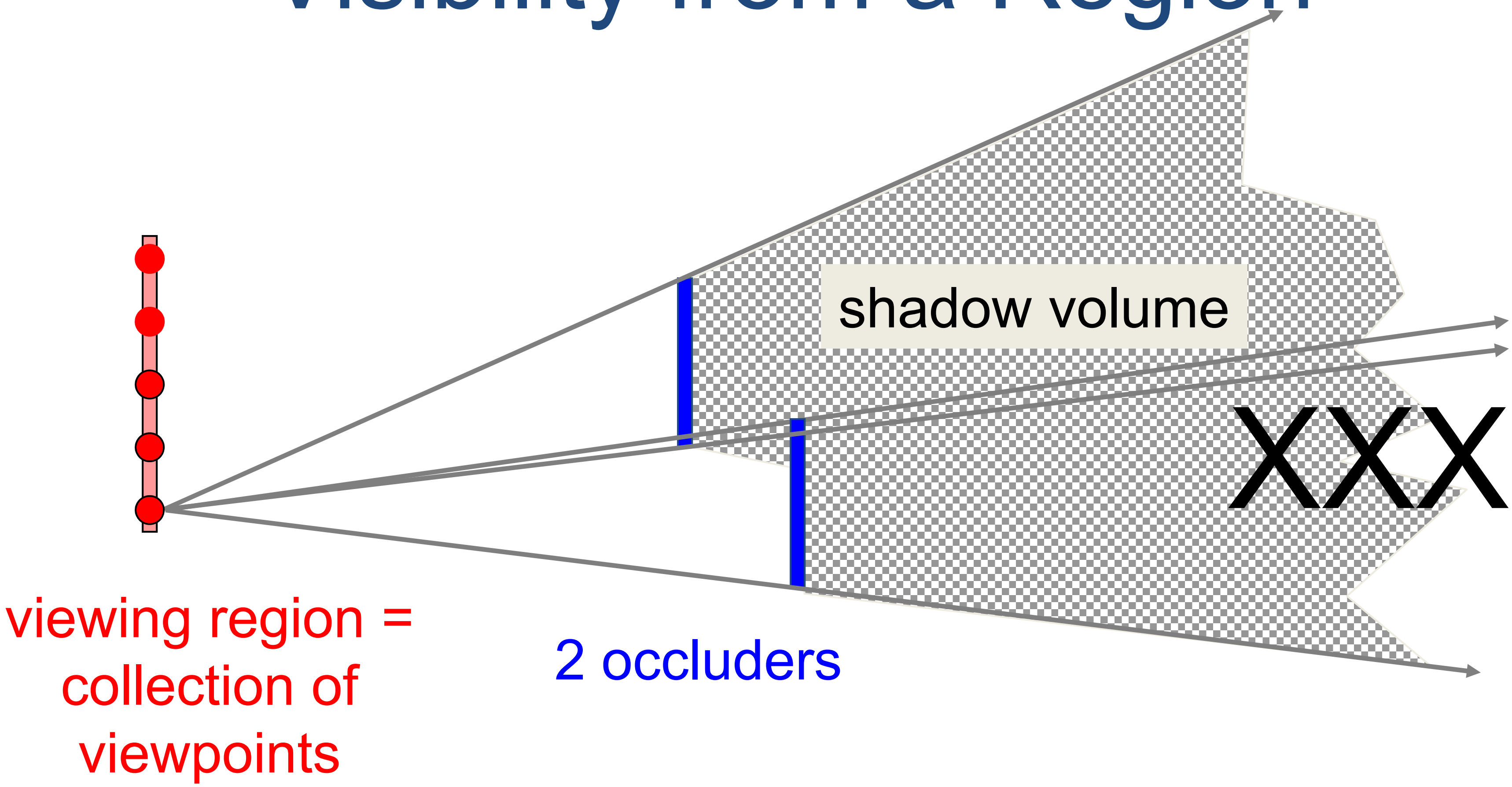
Visibility from a Region



Visibility from a Region

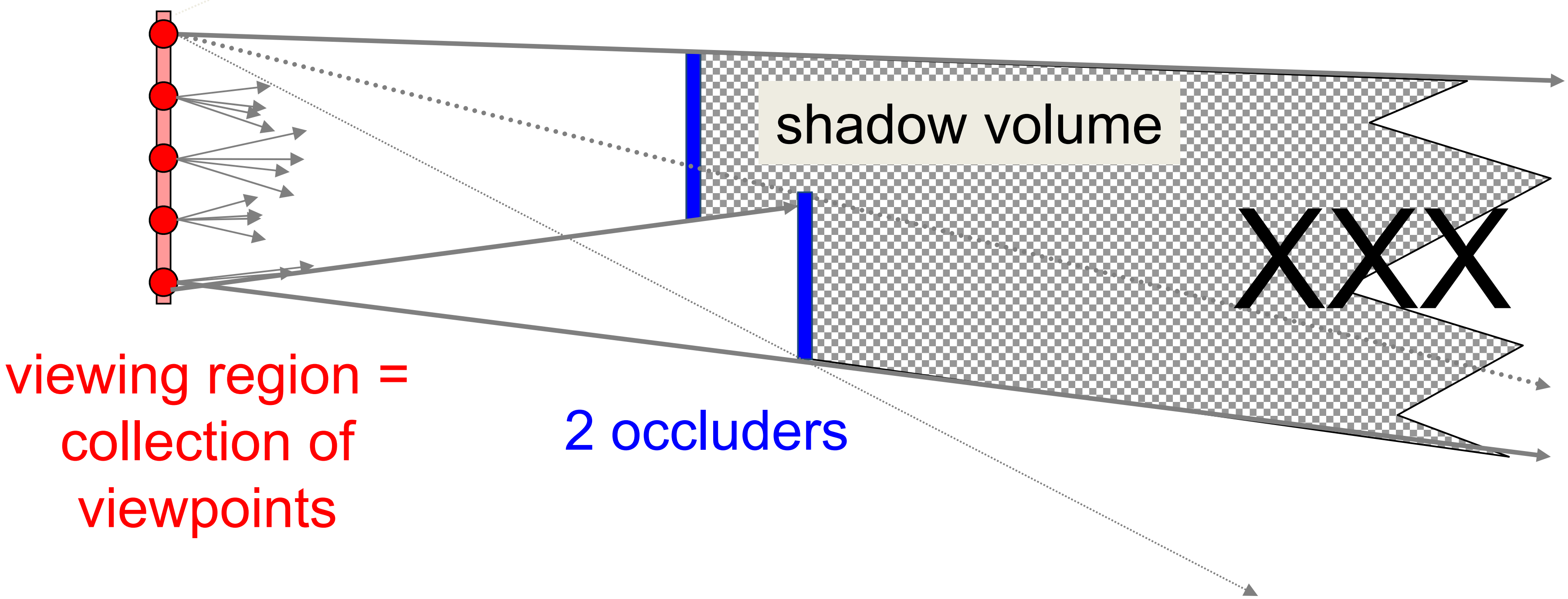


Visibility from a Region



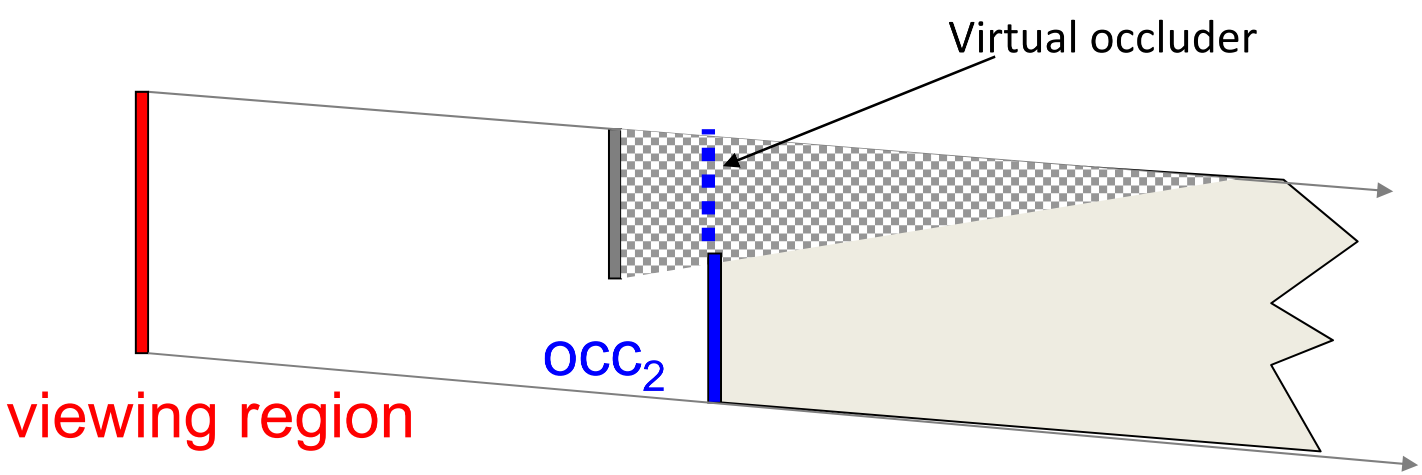
Visibility from a Region

- Area XXX is always occluded → complete shadow volume is **more** than the union of individual shadow volumes



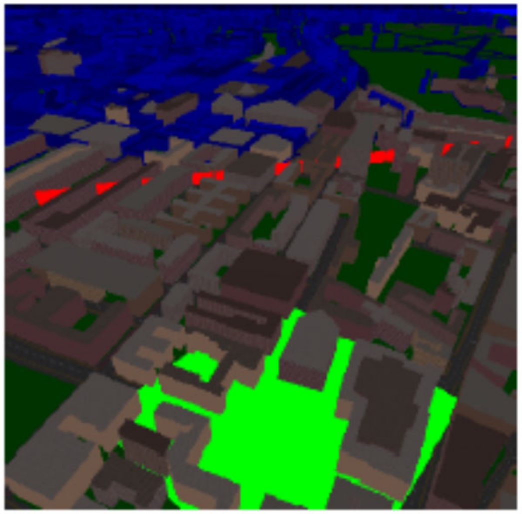
Virtual Occluders for Regions

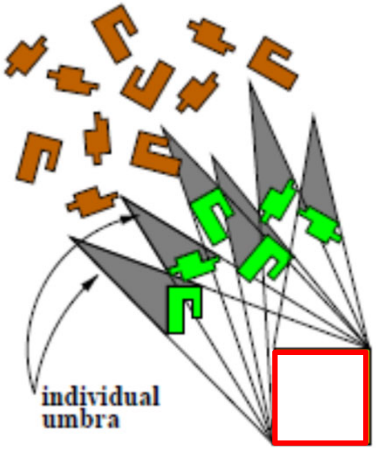
- Shadow volume data structure = empty
- For each occluder occ_i in front-to back order
  - Expand occluder inside existing shadow volume as far as possible
  - Calculate shadow volume SV_i
  - Add SV_i to shadow volume data structure
- Test the scene against the SVDS

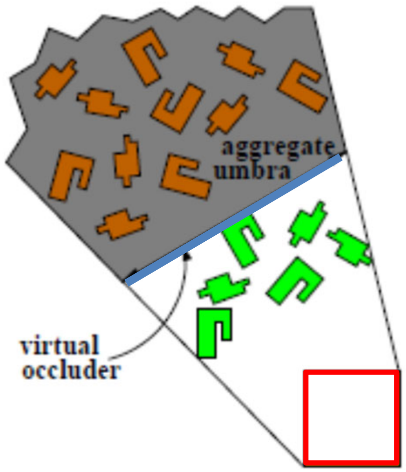


Virtual Occluders from Fusion

- Occluder fusion via intermediate data structure
- Search inside umbra for adjunct occluders and add it to umbra
- Virtual occluder is guaranteed to be occluded from cell
- Virtual occluder grows large







V. Koltun, Y. Chrysanthou, and D. Cohen-Or, "Virtual Occluders: An Efficient Intermediate PVS Representation," Rendering Techniques 2000: Proc. 11th Eurographics Workshop Rendering, pp. 59-70, June 2000.

Questions?