Lecture 10

Large worlds 2:

More on frustum culling

Occlusion culling

Level of detail

Billboards

High-level VSD

Large scenes, large or very large polygon count.

Only a small part of the scene is visible at a given time!

Process polygons in groups, with some kind of spatial information! Remove many polygons with each decision.

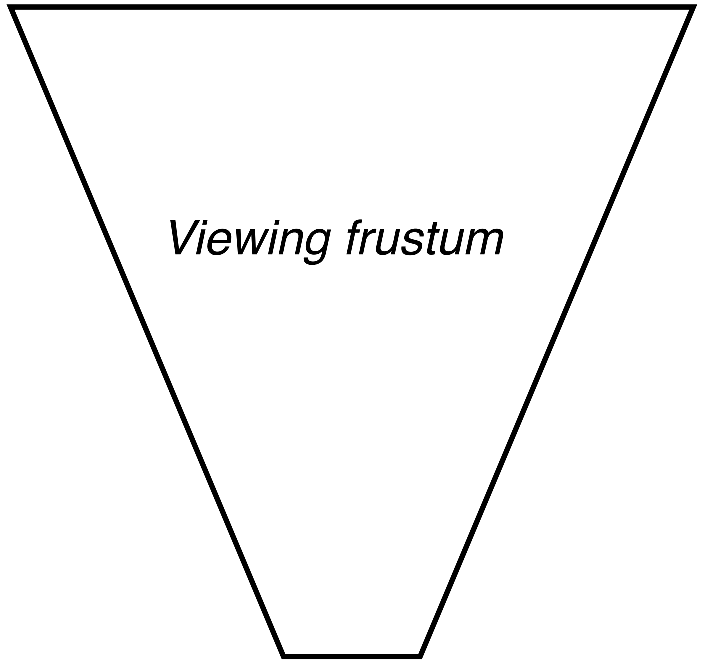
BSP trees (revisited)
Octrees
Domain-specific culling
Portals
PVS

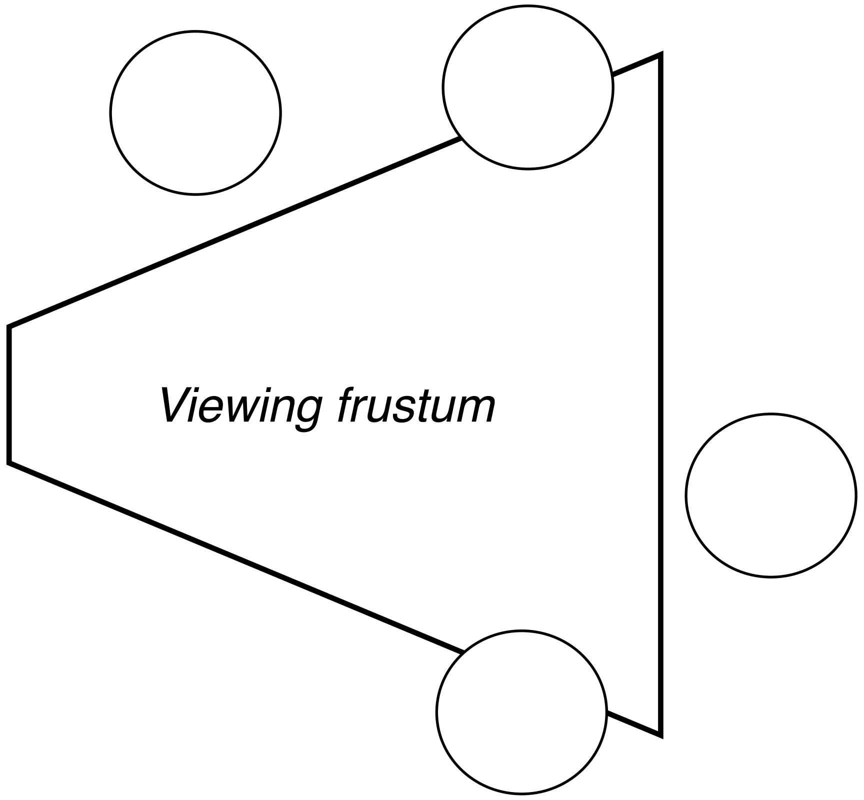
Frustum culling

Create plane equations for each frustum side

Transform to world coordinates

Test against bounding spheres of objects





BSP trees for high-level VSD

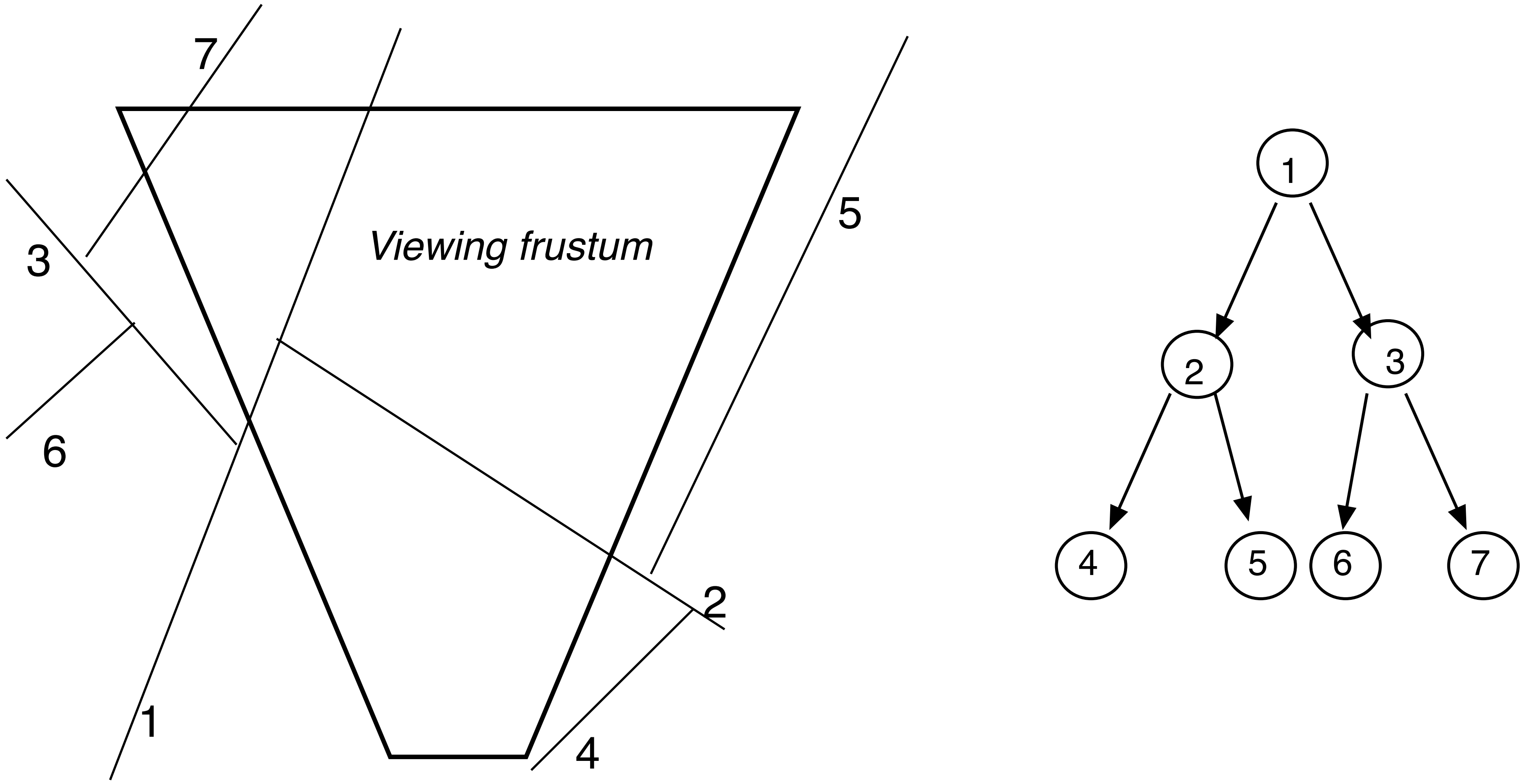
BSP trees simpify frustum culling!

Any node in a BSP tree is a convex volume!

Whenever a volume falls outside the clipping frustum, ALL polygons below that node are removed!

BSP = Binary Space Partitioning

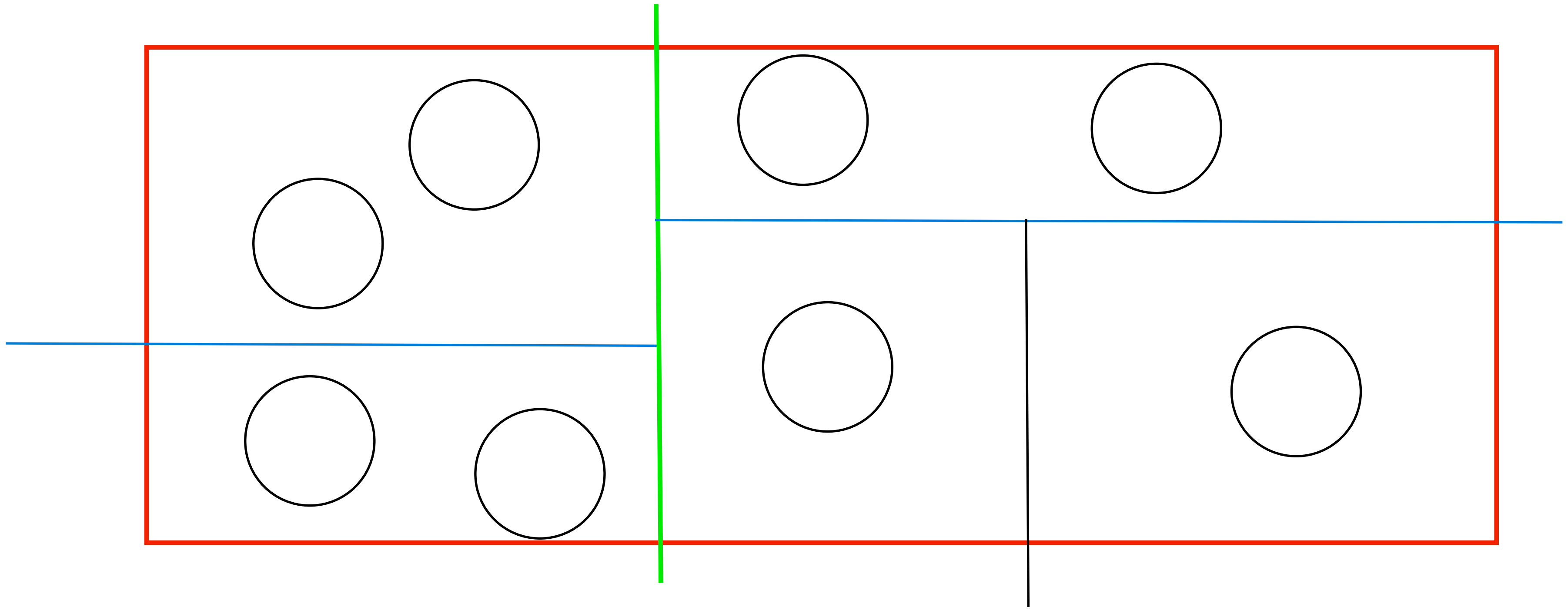
Frustum culling using a BSP tree



Frustum culling using a BSP tree:

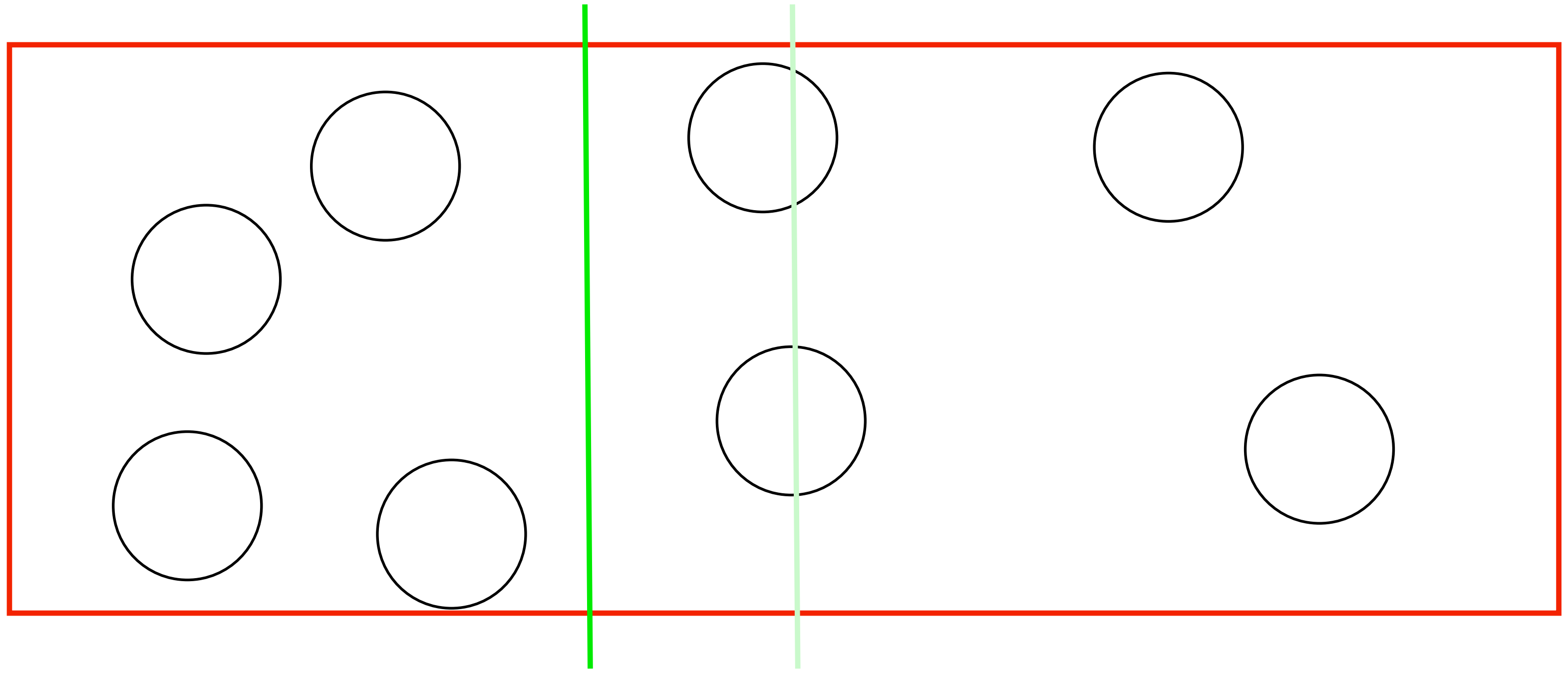
Usually axis aligned - ”kd-tree”

Very simple tests



Building a kd-tree

Split at median: Half of the geometry in each side of the splitting plane. Balanced kd-tree.

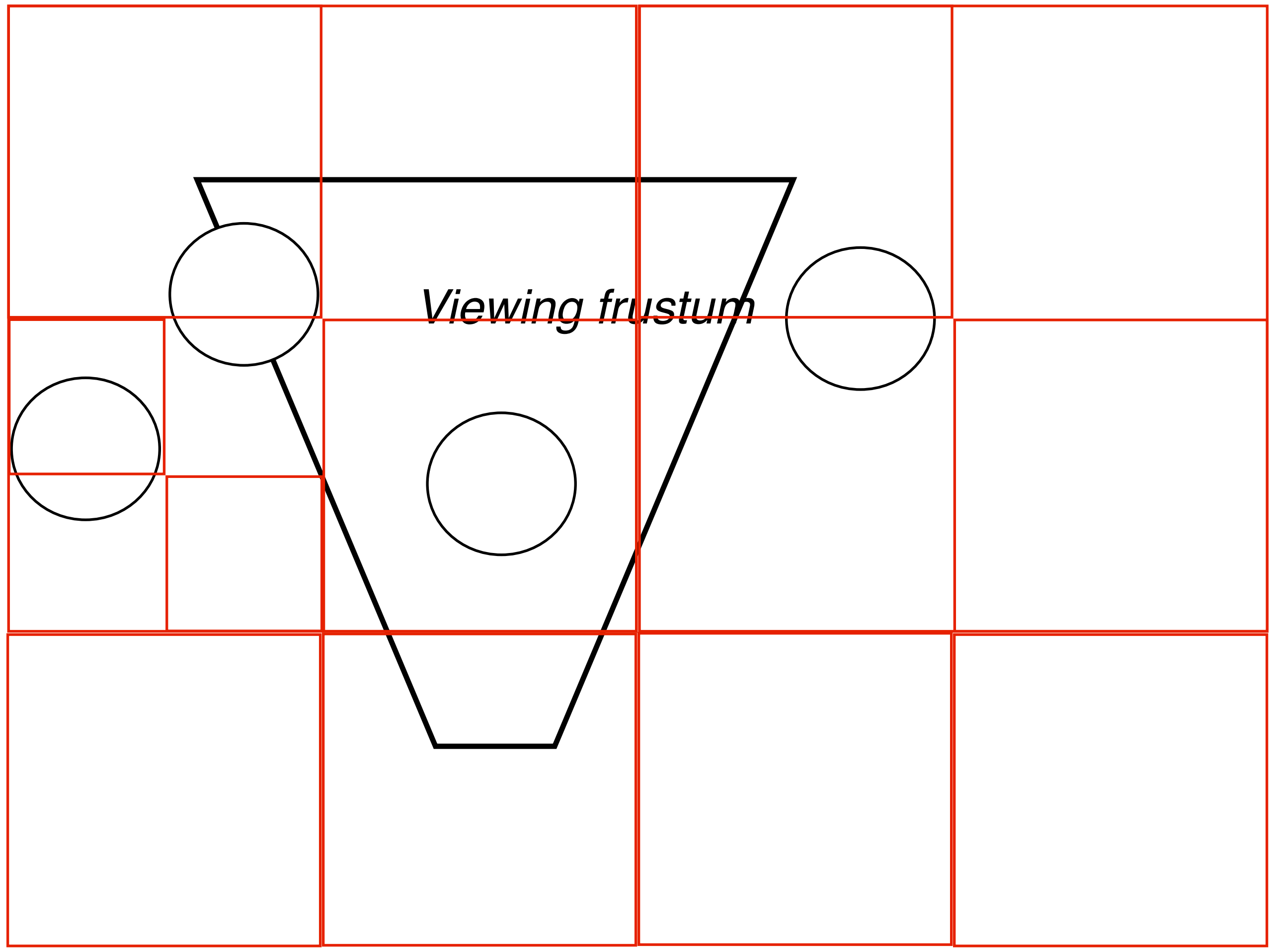

Middle - don’t split here

Median - split here

Octrees:

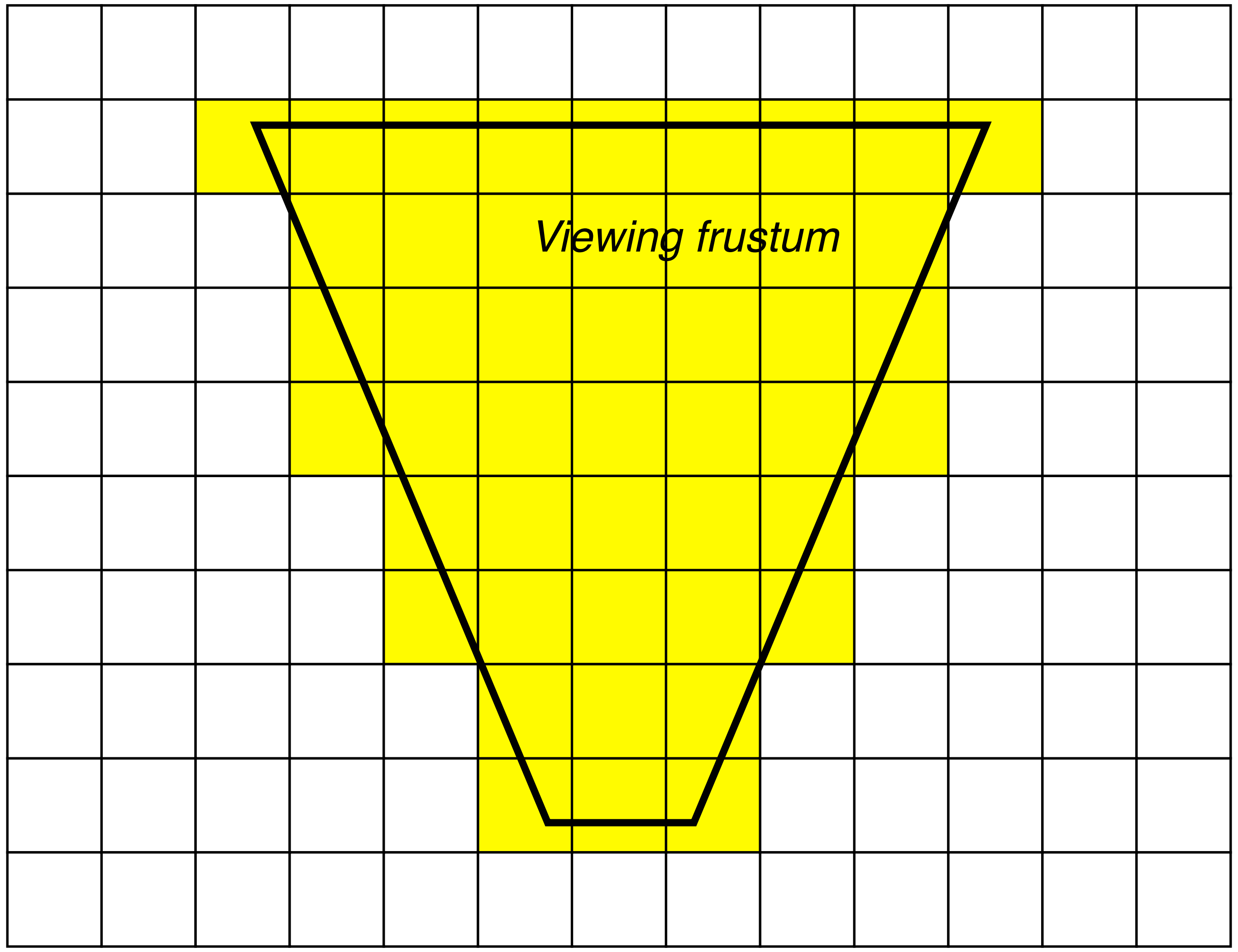
Non-uniform hierarcical space subdivision

Split cells in 8 until sufficient simplicity is achieved.



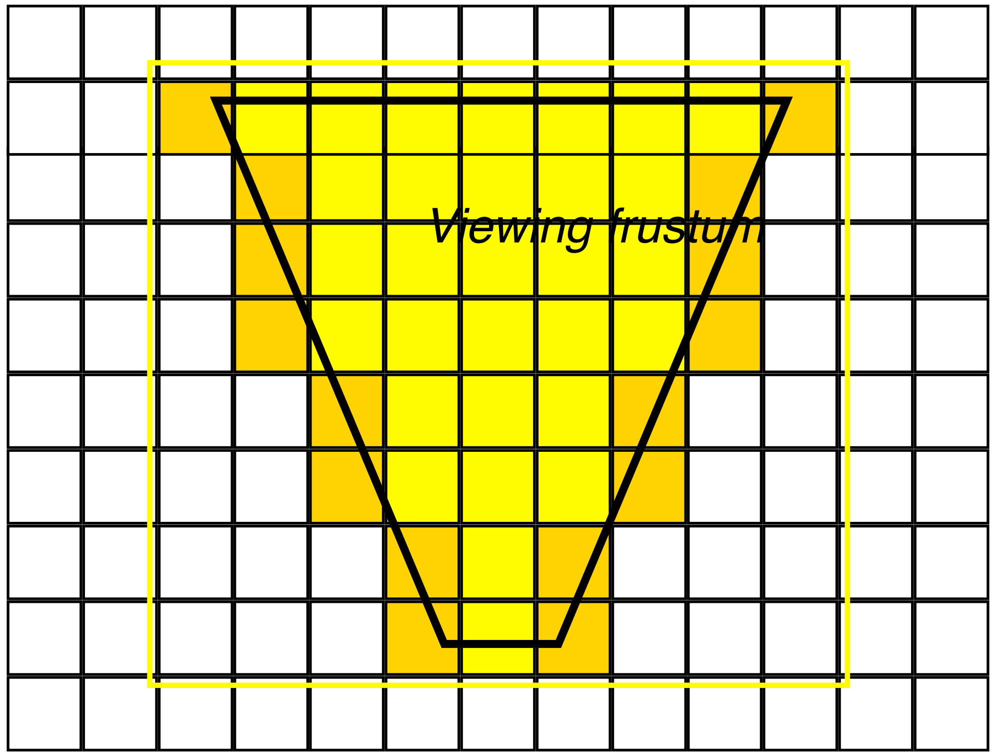
Uniform space subdivision

Simple common case: Terrain defined by a regular grid



Map the frustum edges to the grid coordinates

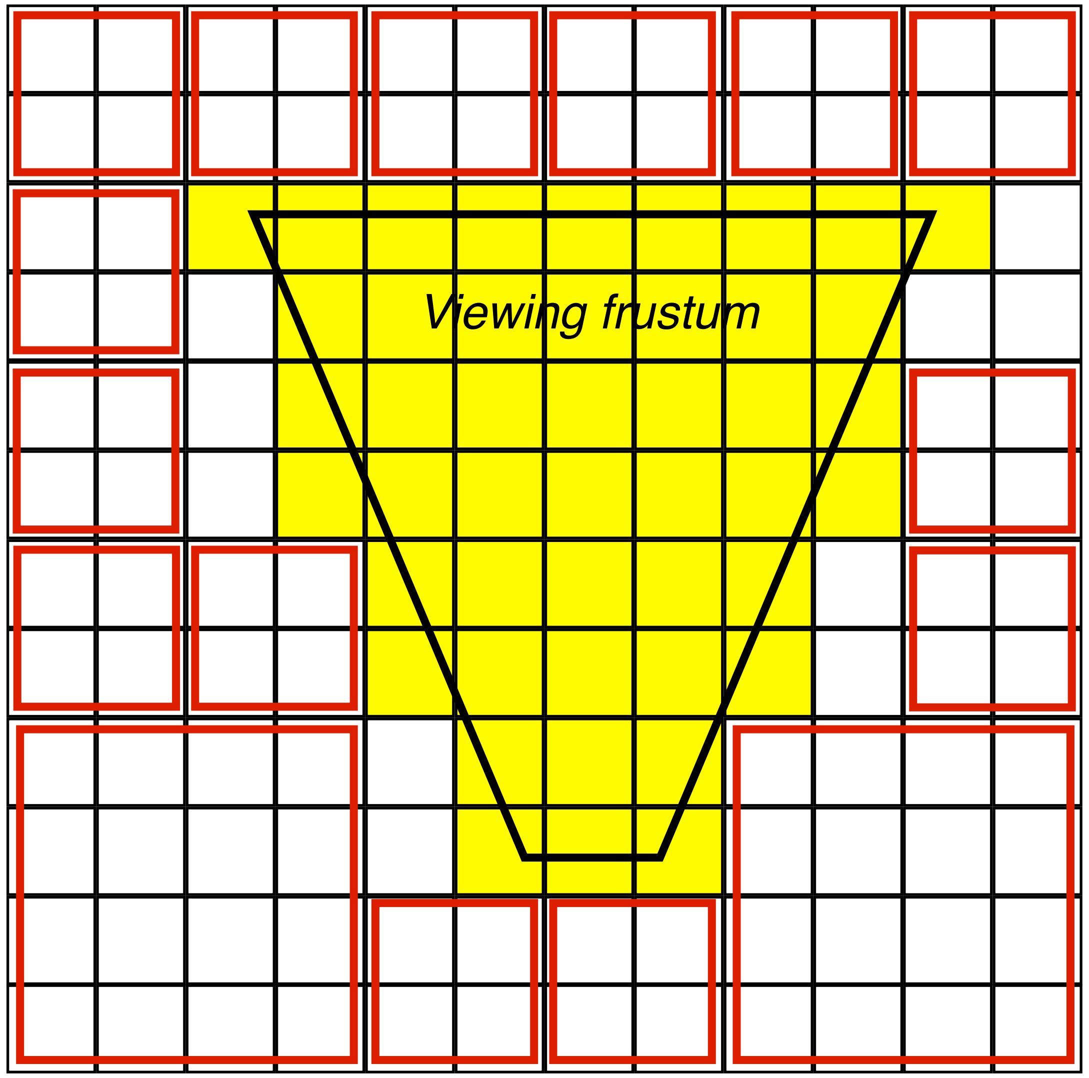
Draw all polygons between edges



Cheap quick hack version:

Find the bounding box of the frustum. Gives a simple 2D rectangle with grid spaces to draw. Up to 50% unnecessary polygons.

Grid, alternative approach: quadtree



Real-world example: Bugdom series

Fairly sparse environment, frustum culling is sufficient.

Step 2: Occlusion culling

Even though we can remove all polygons outside the vieweing frustum, polygons within often occlude each other.

How do you know what polygons in the viewing frustum are hidden?

- **Portals**
- **Potentially Visible Set**

Cells and portals method

(often referred to only as "portals")

Suitable for buildings, with many enclosures.

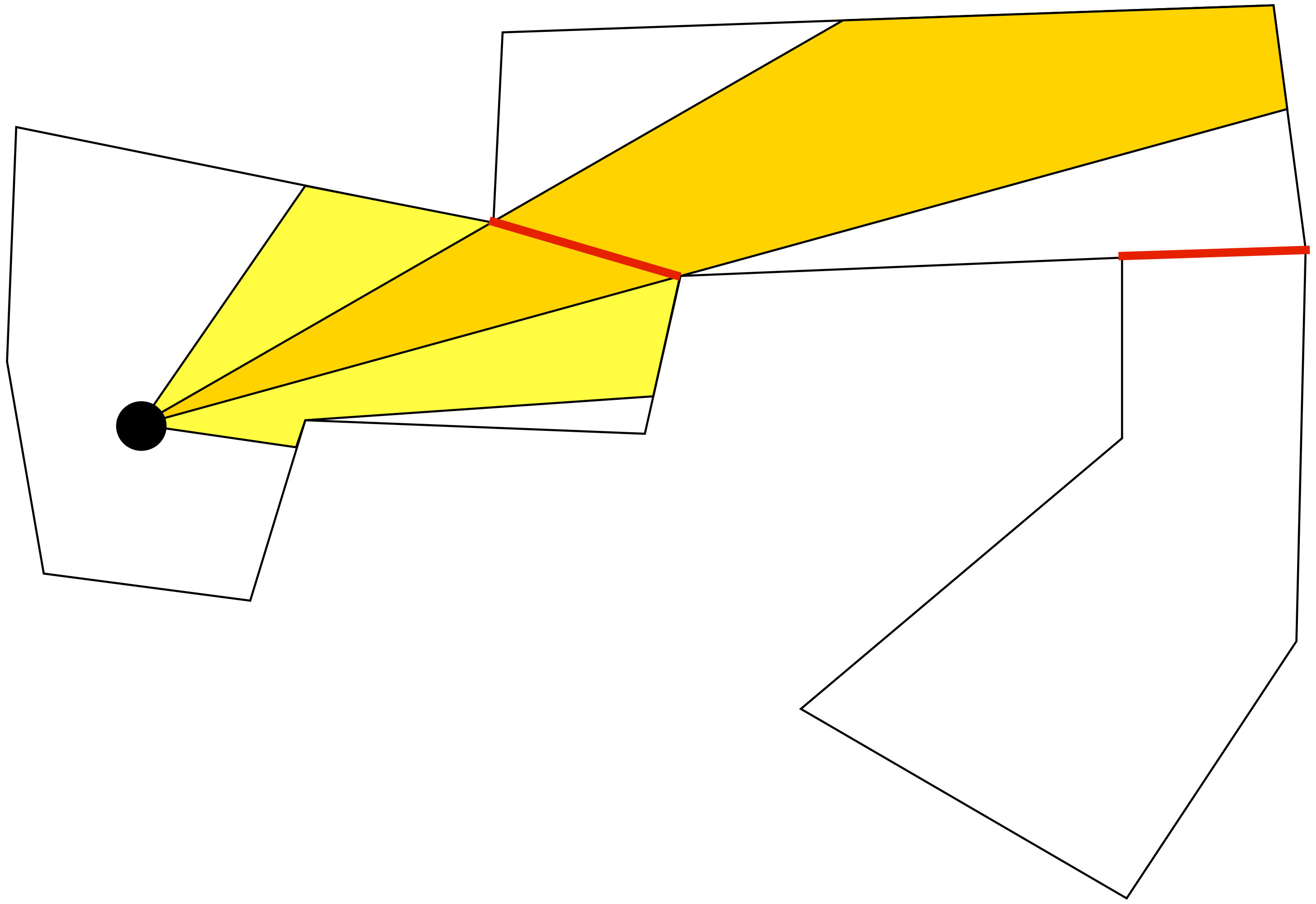
Split the world into smaller parts, create connections as "portals" between them. (Dark Forces, Tomb Raider)

Each portal is a branch in an adjacency graph

Intuitive and (fairly) simple, but inefficient for outdoor scenes.

Portals

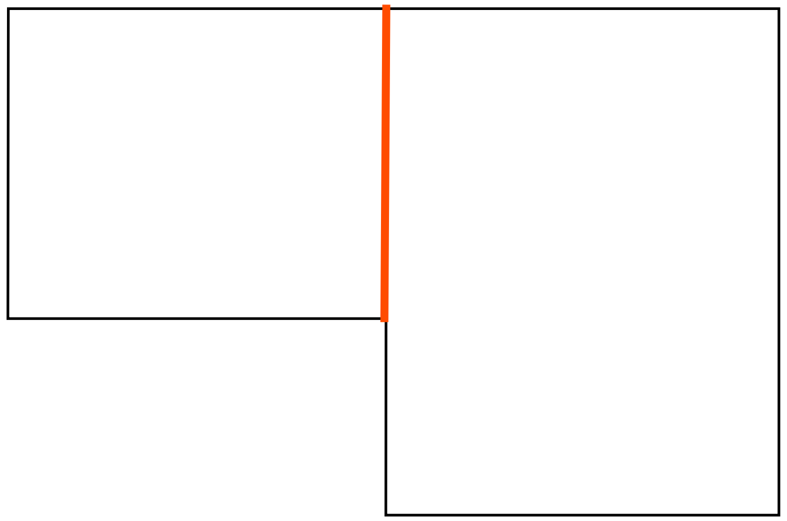
Polygons are grouped into cells, “rooms”



When you find a “portal”, clip to it and render the next room

Real-world example: Dark Forces

Level editor reveals portal-based engine

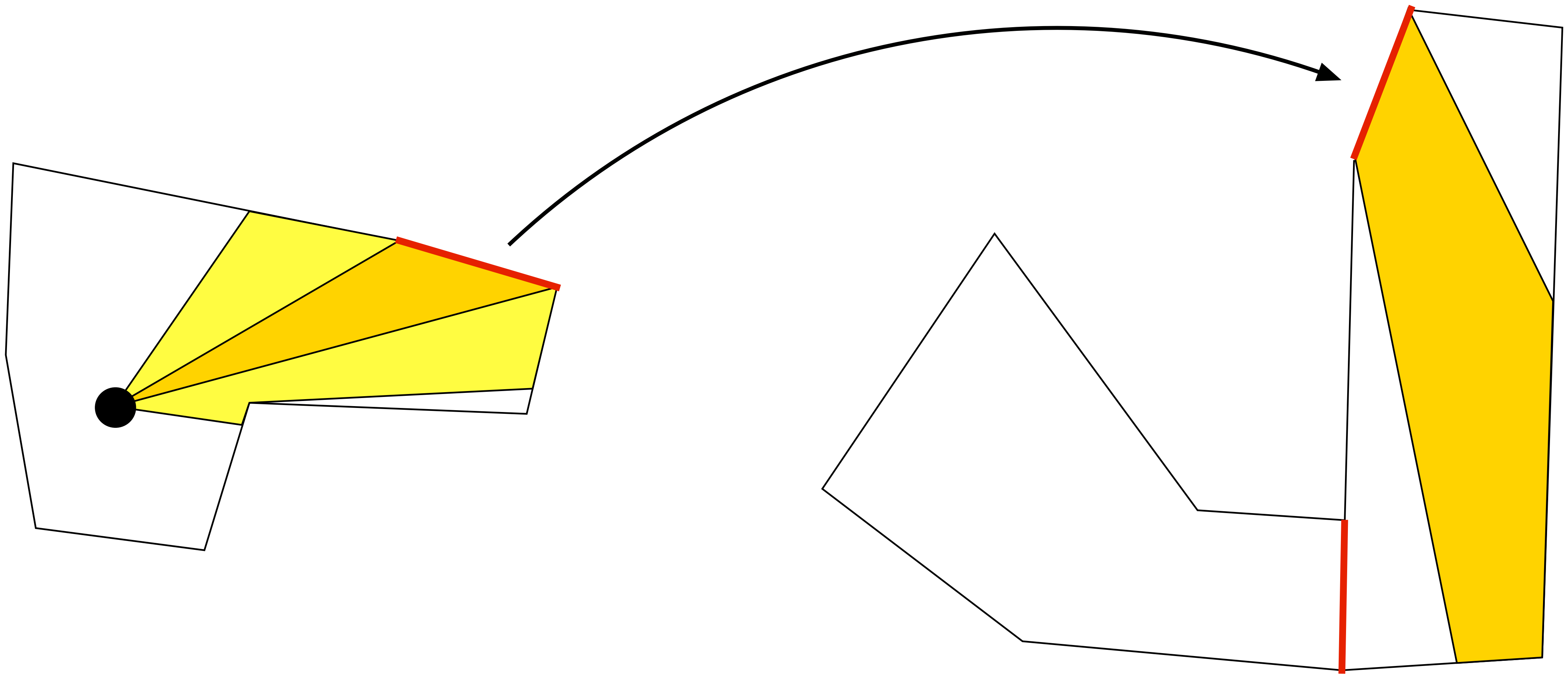


Edit levels by drawing 2D polygons, connect them as portals

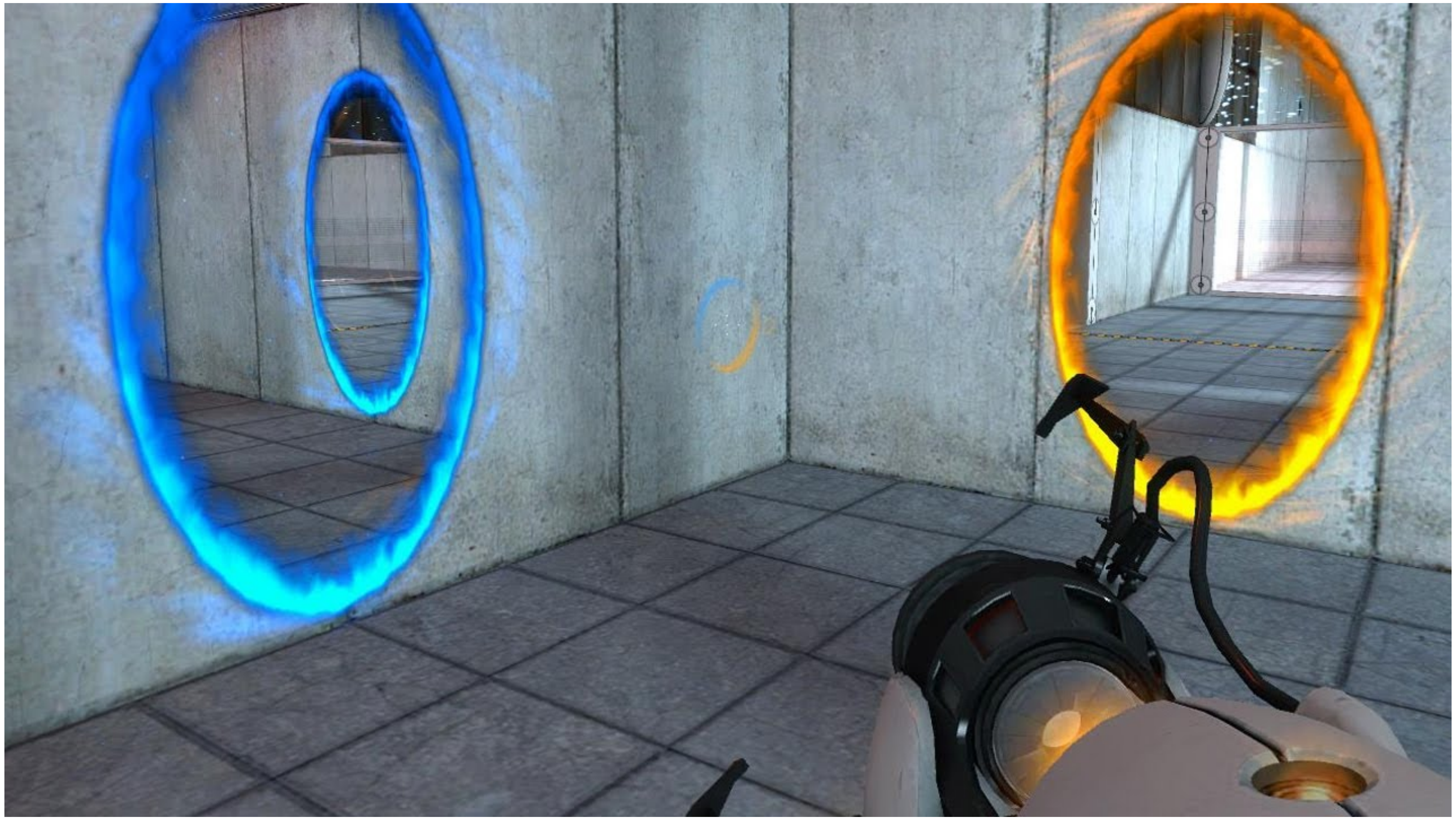
Notable limitation in game: Two windows/openings can never be on top of each other!

Portal transformations

Nothing stops you from putting a transformation in your portals!



...a trick used in a well-known game,
named after the technique!



Potentially visible set (PVS)

A bit list for some part of the world (cell),
specifying what polygons may be visible. (Quake)

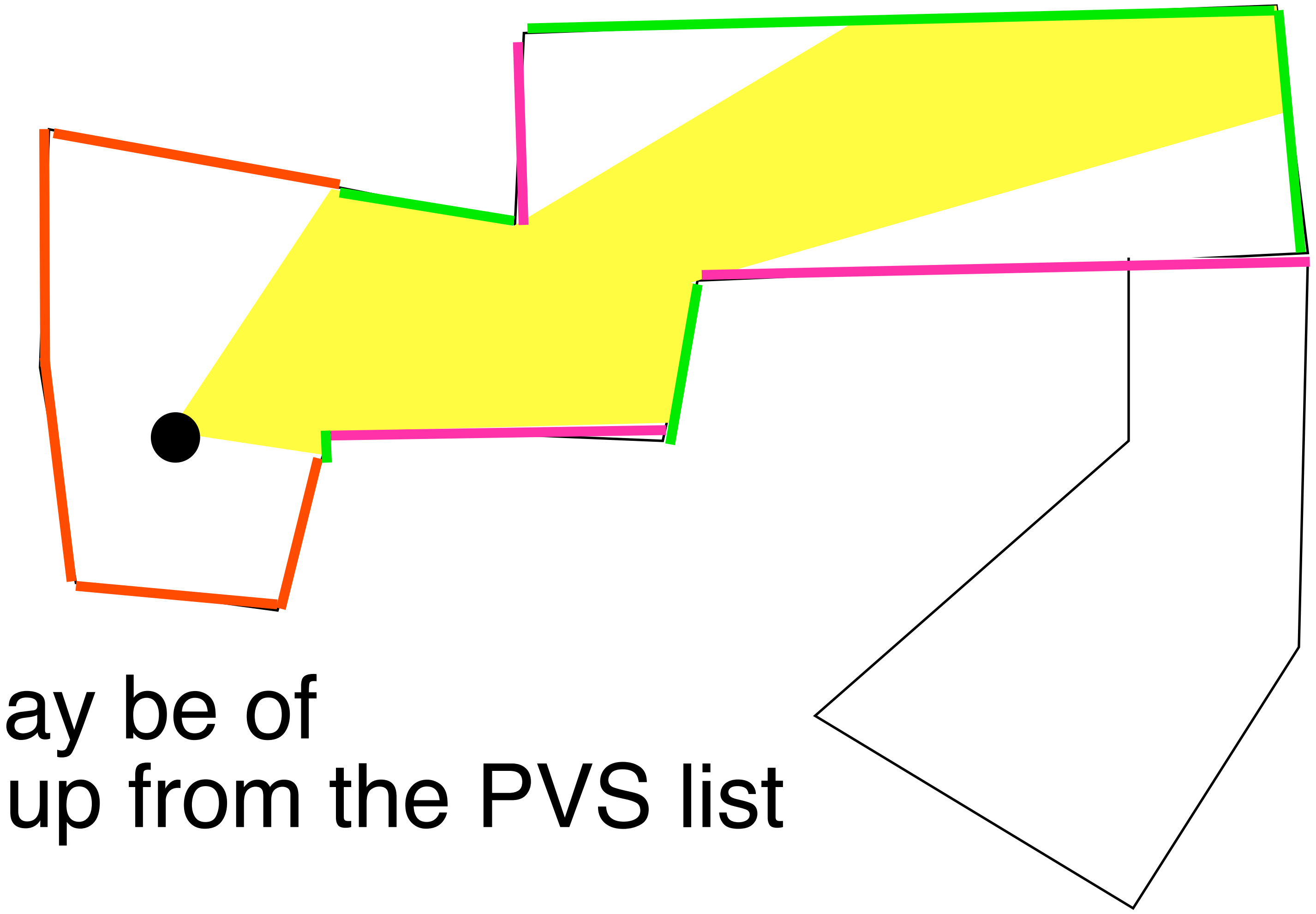
The list is huge, but can be compressed.

Pre-compute the list for a static scene.

Use BSP trees for creating cells automatically.

Potentially Visible Set

More general method, faster for very complex scenes.



All polygons that may be of
interest are looked up from the PVS list

Pre-generating the PVS

Done either for a point or for a cell

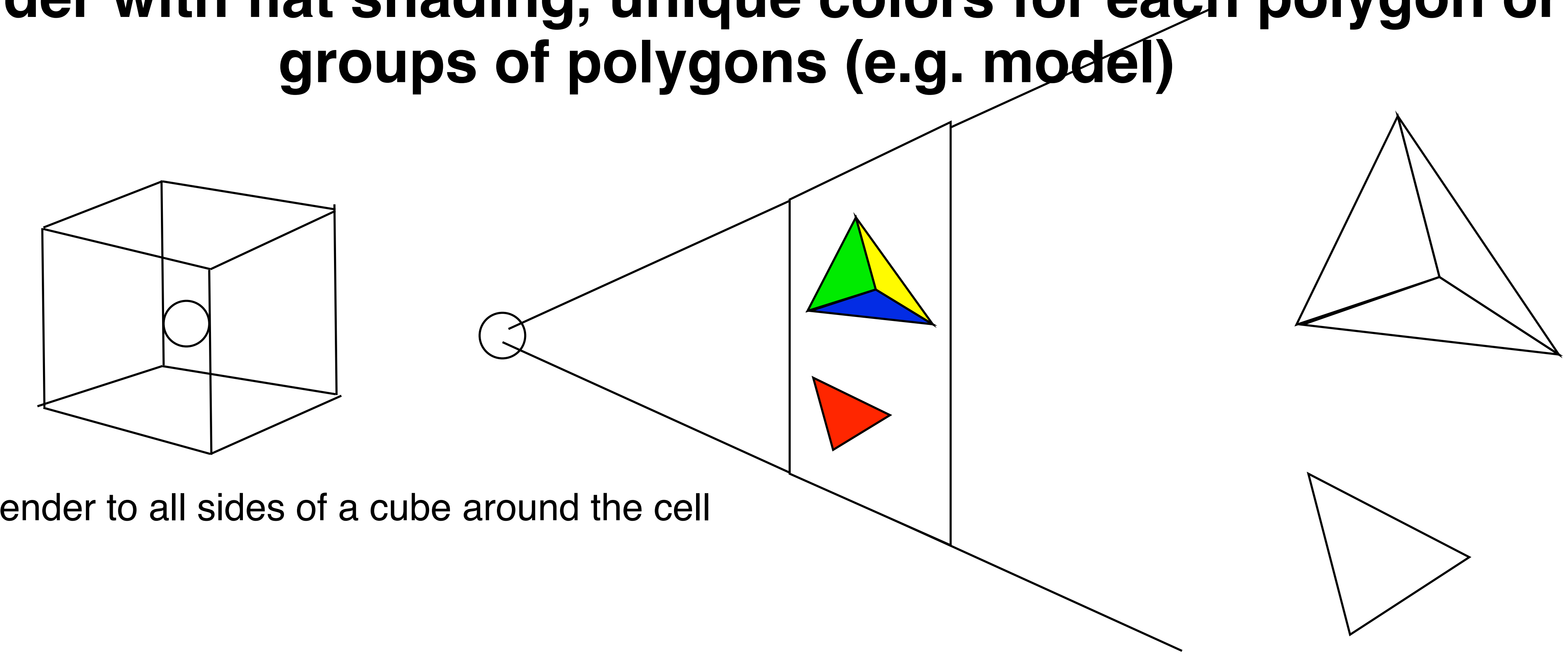
1) Image-space method

2) Object-space method

Image-space PVS generation

Render 6 images, all covering 1/6 of direction space

Render with flat shading, unique colors for each polygon or groups of polygons (e.g. model)



Render to all sides of a cube around the cell

Inspect the resulting images. For every color that appears, the corresponding polygon(s) is/are added to the PVS

Conclusions about Visibility processing/ High level VSD:

- Frustum culling easy!
- Doesn’t have to be perfect - some waste at edges are OK
- Complex scenes can need more