



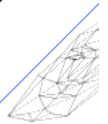
# Lecture 10

Managing large worlds:

High-level Visible surface detection

Level of detail

Hierarcical modelling



*etection of visible surfaces  
(revisited)*

*Det*

*A-buffer  
Scan-line method  
BSP trees  
Area subdivision  
trees*

*Backface culling  
Painter's algorithm  
Z-buffer  
Ray-casting  
Portals*

*Ó*



## More than just getting the right pixel in the right place:

Maximize performance by

- discarding polygons that will not be visible due to:
  - 1) Clipping to viewing frustum
  - 2) Back-face culling
  - 3) Other possibilities?
- drawing pixels only once (or as few times as possible)
- don't make many-to-many checks between polygons!



## BSP trees, old method:

Efficient sorting of static scenes for Painter's Algorithm, BUT

- Draws pixels multiple times
- Draws polygons that will be totally overdrawn
- Generating the tree is slow
- Generating an optimal tree is SLOWER!

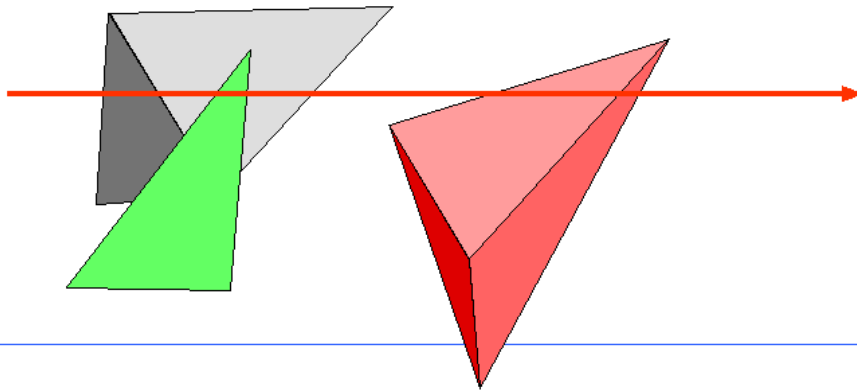


# Scan-line method

Drawing pixels only once

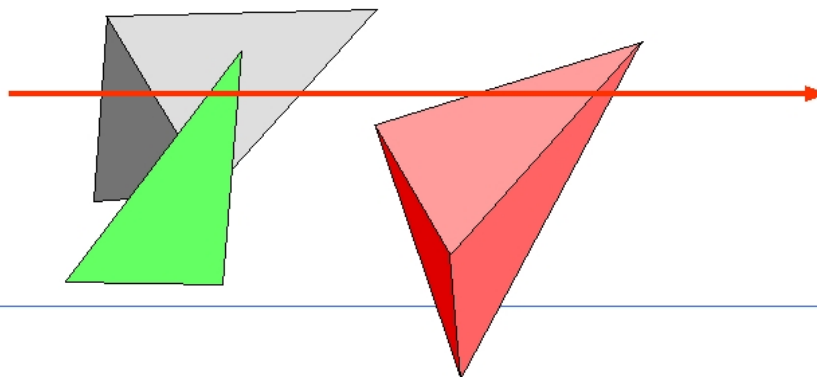
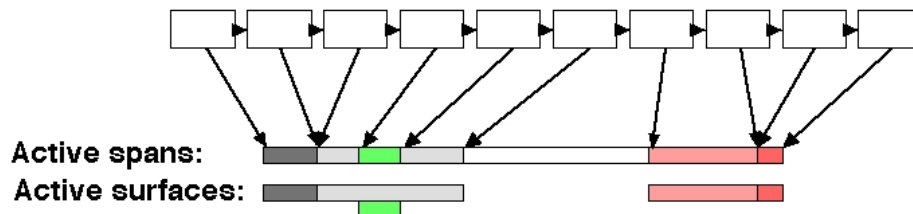
Like the polygon filling method:

- Work on one scan-line at a time
- keep a list of active edges/spans



# Scan-line method

Active edge list





## **Scan-line method**

**Highly efficient for 3D graphics with no GPU. Not of much interest for modern GPU-equipped systems.**

**...but stay tuned...**



**covered so far:**  
**Low-level VSD**  
(visible surface detection)

**Backface culling**  
**Painter's algorithm**  
**Z-buffer**  
**Scan-line method**  
**BSP trees (as presented before)**

**All polygons are treated individually**

**Good for small scenes**  
**(small total number of polygons).**



## 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



## Types of “visibility processing” algorithms:

- Exact algorithms
- Approximate algorithms
- Conservative algorithms



### **Exact:**

**Finds all visible or partially visible polygons**

**Drawback: Usually too computationally expensive**

### **Approximative:**

**Finds most visible polygons, excludes most invisible ones**

**Fast. Some artifacts.**

### **Conservative:**

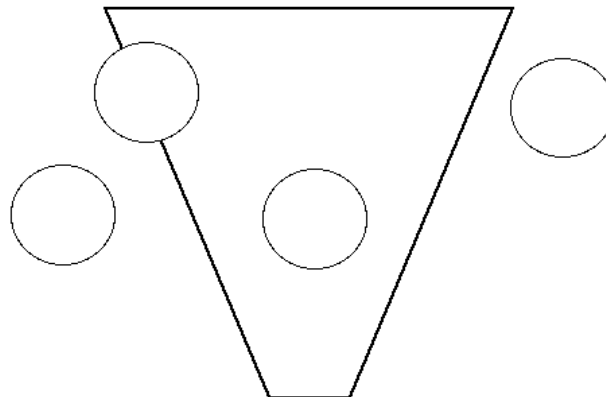
**Finds all visible polygons but includes some invisible ones.**

**No artifacts. Potentially lower performance than “approximative”**



### **Step 1. View volume culling (Frustum culling)**

**What polygons are inside the view volume?**



**Principle: Make a subdivision of the scene, so tests can be done on groups, e.g. separate objects or limited parts of the scene.**

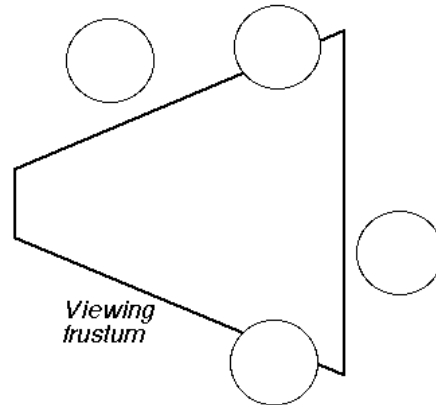
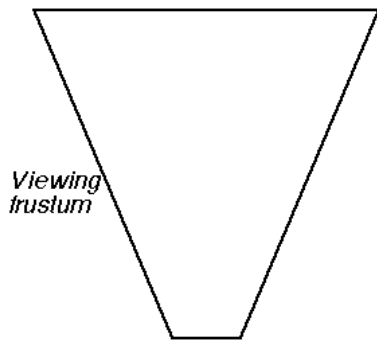


## View volume culling

Create plane equations for each frustum side

Transform to world coordinates

Test against bounding spheres of objects



## Smarter use of BSP trees

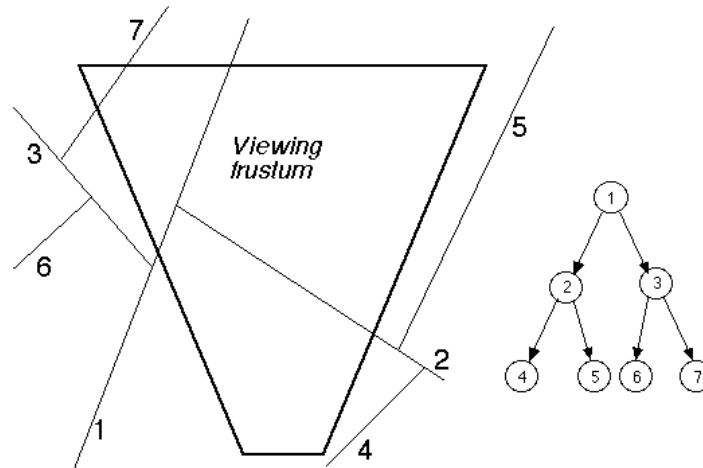
**BSP trees simplify frustum culling!**

**Any node in a BSP tree is a convex volume!**

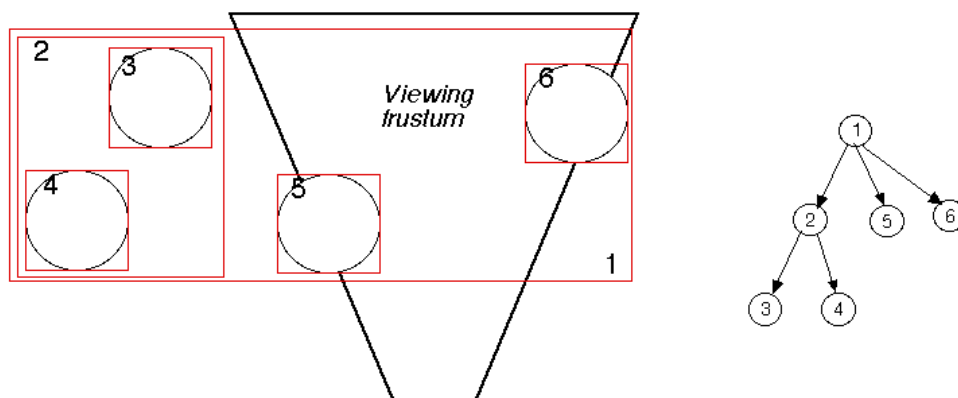
**Whenever a volume falls outside the clipping frustum, ALL polygons below that node are removed!**



## Frustum culling using a BSP tree



## View volume culling using a group hierarchy with bounding boxes

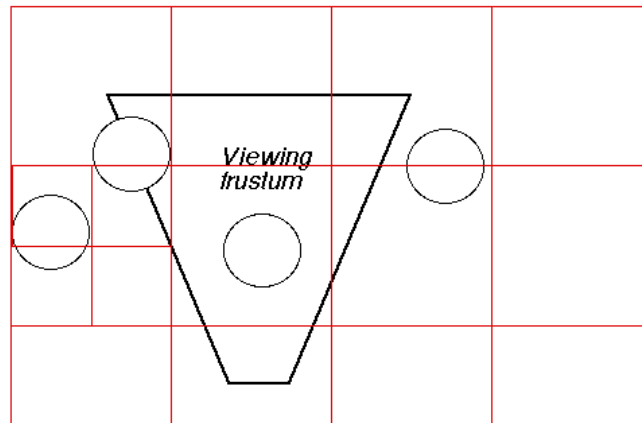




## Octrees:

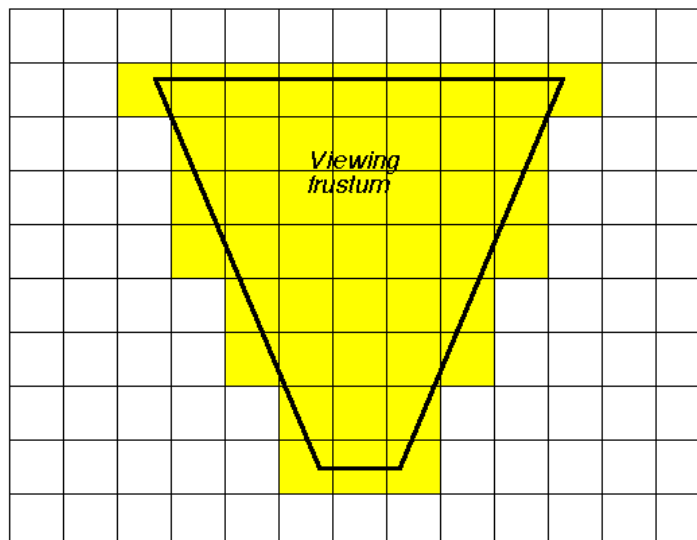
Non-uniform hierarcical space subdivision

Split cells in 8 until sufficient simplicity is achieved.



Uniform space subdivision

Simple common case: Terrain defined by a regular grid

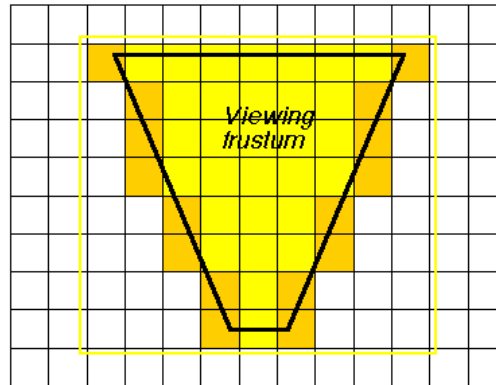




## Information Coding / Computer Graphics, ISY, LiTH

Map the frustum edges to the grid coordinates

Draw all polygons between edges



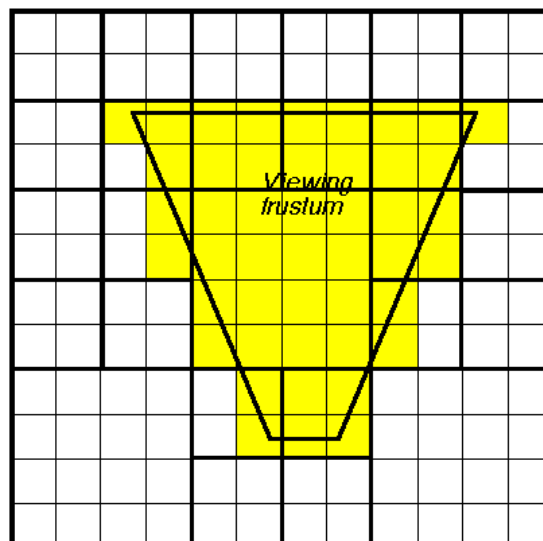
Cheap quick hack version:

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



## Information Coding / Computer Graphics, ISY, LiTH

Terrain generation, 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 viewing 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, check for the “portals” between them. (Dark Forces, Tomb Raider)**

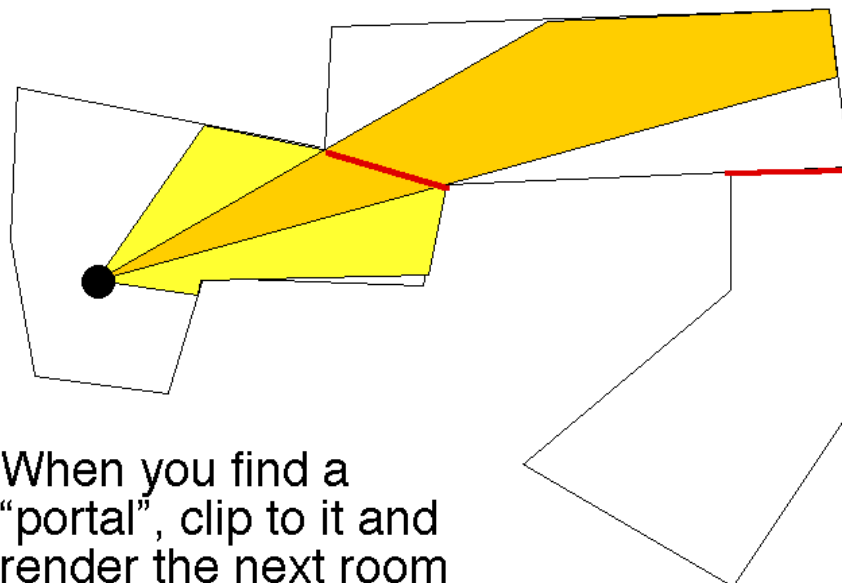
**Each portal is a branch in an adjacency graph**

**Straight-forward and simple, but inefficient for outdoor scenes.**



## Portals

Polygons are grouped into cells, “rooms”

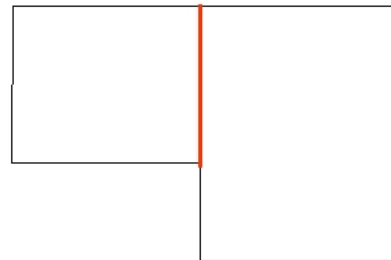


When you find a “portal”, clip to it and render the next room



## 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!



## 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.

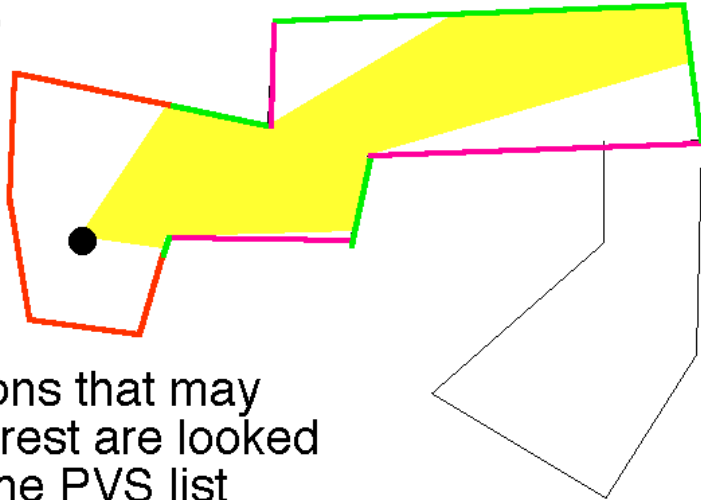
Pre-compute the list for a static scene.

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

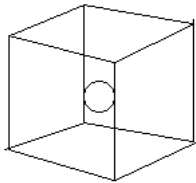
- 1) Image-space method
- 2) Object-space method



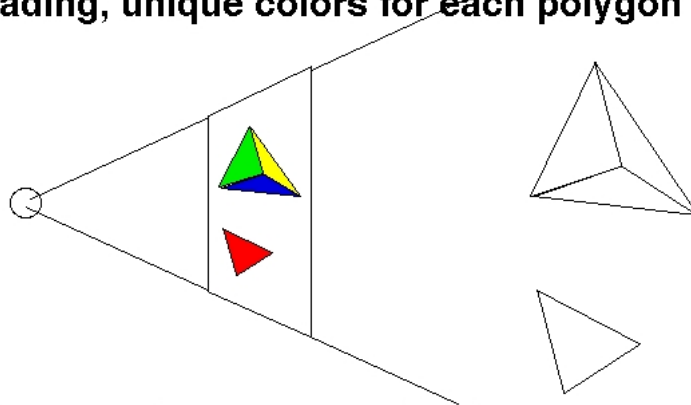
## Image-space PVS generation

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

Render with flat shading, unique colors for each polygon



Render to all sides of a cube around the cell



Inspect the resulting images. For every color that appears, the corresponding polygon is added to the PVS