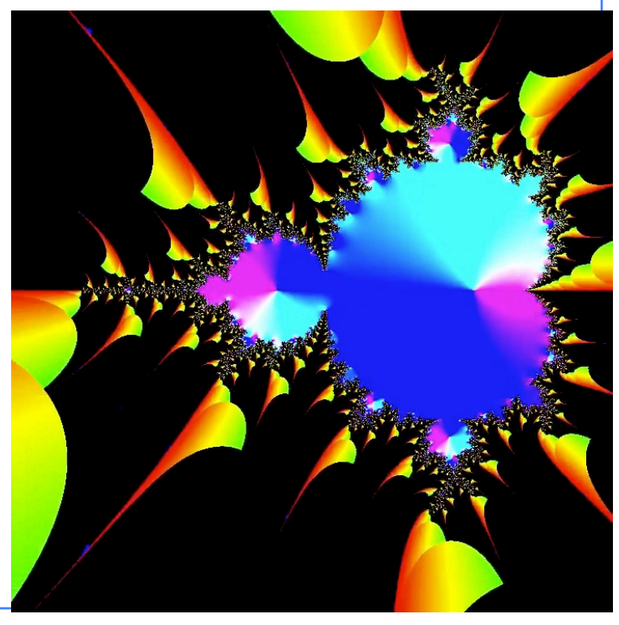
TSBK07/TSBK11

Computer Graphics

Ingemar Ragnemalm, ISY



Lecture 8

Picking

Rotation around arbitrary axis

Trackball controls

Large worlds, frustum culling

Occlusion culling

Lecture questions

1. How can you perform picking (detect a click in a specific object)?

2. How can flat shading be used for picking?

3. How do you rotate around any axis in 3D?

4. Where in the transformation chain do you insert the trackball rotation?

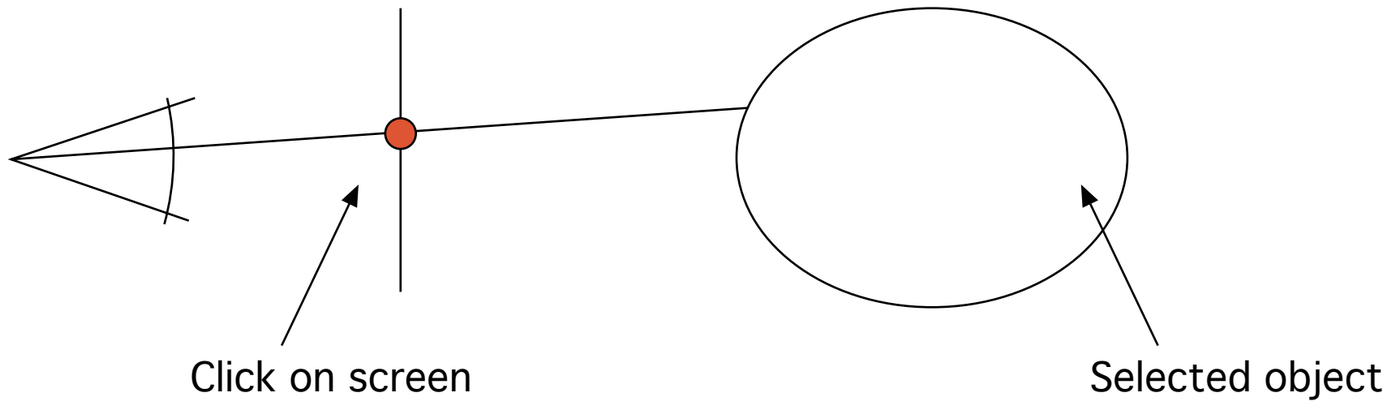
5. Do you perform frustum culling in view or world space?

6. How can you optimise a very large scene even more?

Picking

Interactively selecting objects with a mouse

Can be solved with raycasting!



WAAH!

PLEASE STOP PICKING ON ME!

But in our case... picking = selecting.

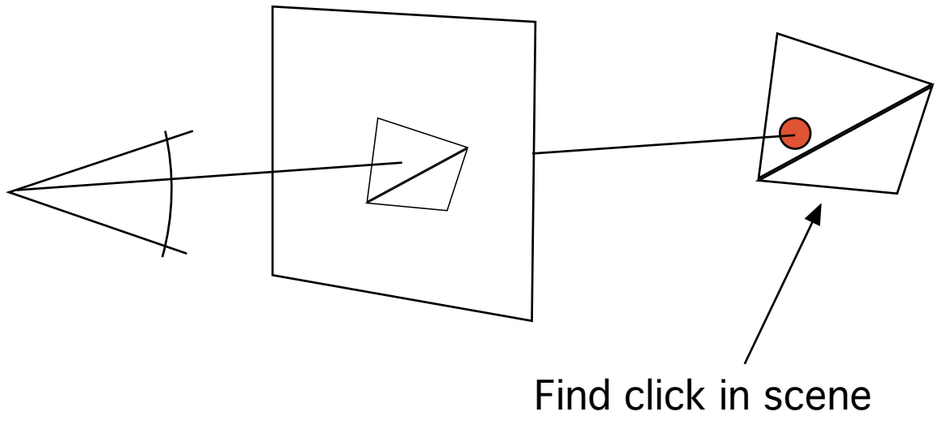
Picking in model space

Point in viewing plane

Create line through origin and point

Transform by inverse model-to-view

Find intersections with models



Picking in view space

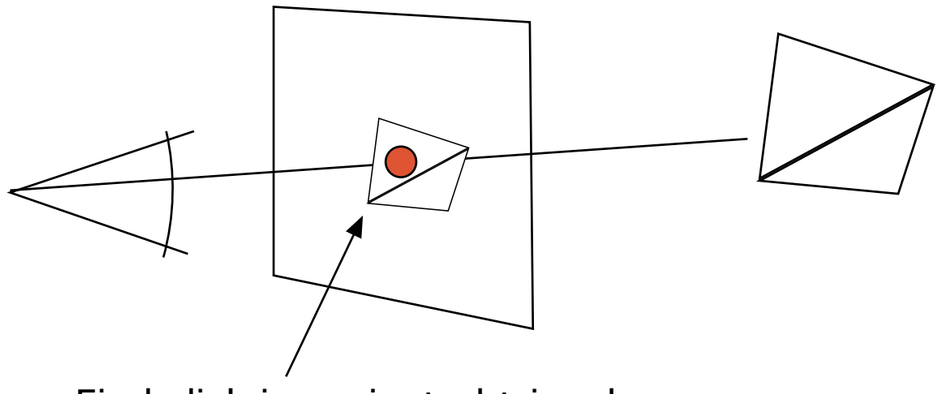
Point in viewing plane

Transform models by model-to-view

Find intersections with models

Cumbersome if all models need to be transformed

Cheap if done as part of drawing



Find click in projected triangles

Picking in image space

Draw all models with individual indices (colors)

Get resulting color at click from frame buffer

Requires drawing entire scene an extra time - but you can restrict drawing to a very small area

Colors for indexing

The standard setting allows 256 shades per channel, red, green and blue

Assign each model a unique index, split to R, G, B, 8 bits each, divide each by 255 to get color values.

Draw. Get pixel at mouse position with the not very common call "glReadPixels".

Method in old OpenGL, GL_SELECTION

Requires you to enter a special mode, selection mode, and assign indices to each model.

Similar in usage and result to the color indexing method. Obsolete! Use any of the ones above instead.

Best picking method?

Picking in model space: Typically done on CPU (somewhat slow but capable).

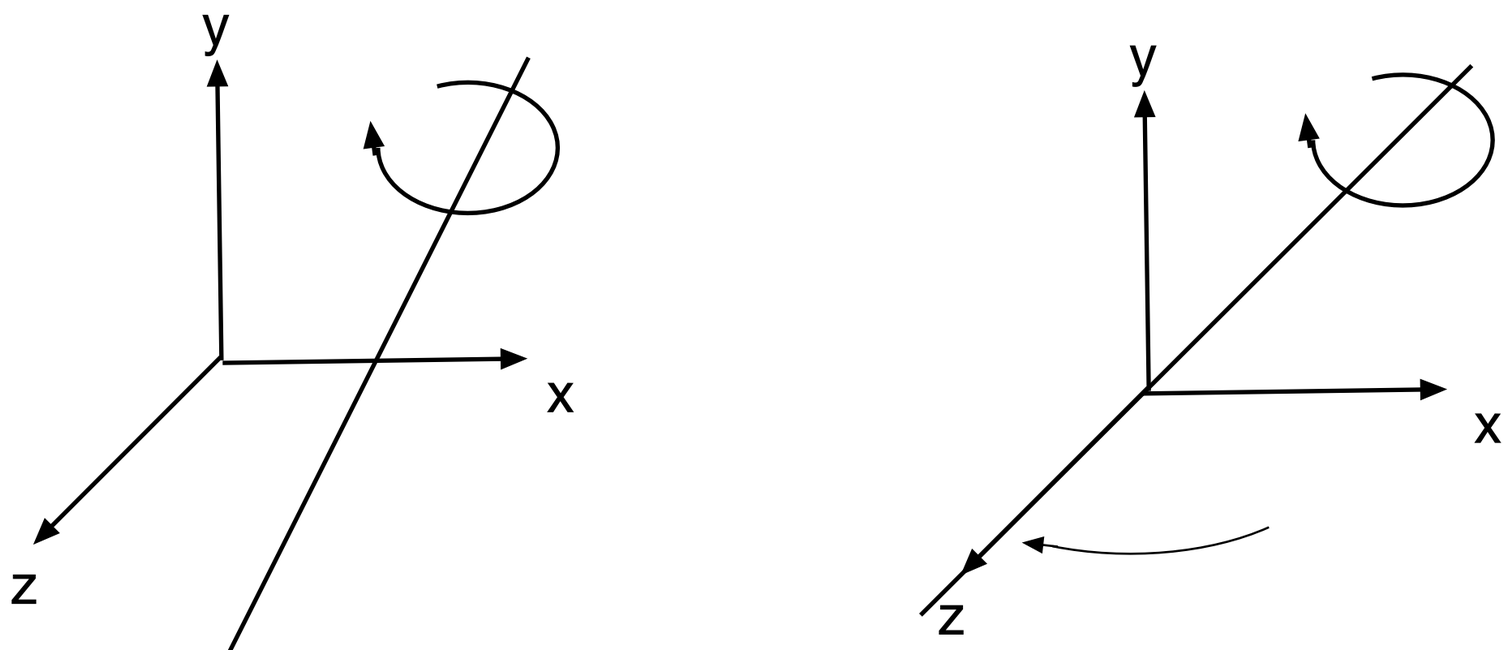
Picking in view space: Efficient if done in the geometry stage (needs some tricks to output result to host)

Picking by index colors: Easy but can not identify local parts of a model.

More transformations

Rotation around arbitrary axis

Rotation around arbitrary axis



Transform to align axis with the Z axis, rotate, and transform back.

Rotation around arbitrary axis, using change of basis:

$v = p_2 - p_1$

$u_z = u = v / |v| = (u_x, u_y, u_z)$ Normalized!

$u_y = u \times (u_x,0,0) / | u \times (u_x,0,0) |$

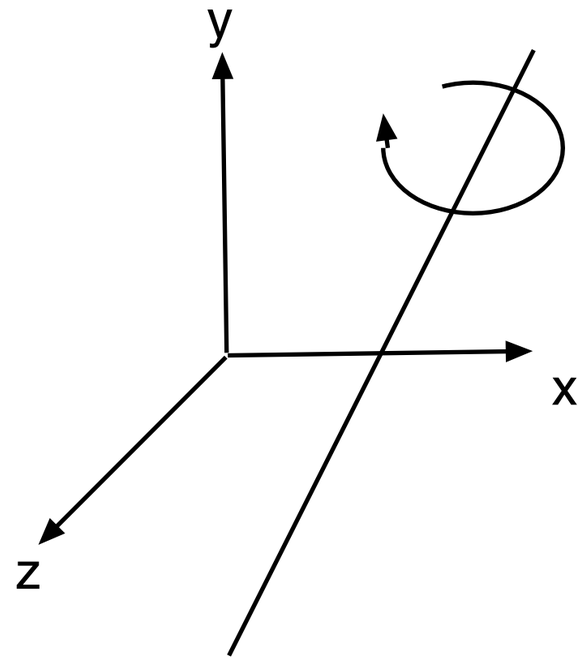
$u_x = u_y \times u_z$

$$R = \begin{bmatrix} u_{x1} & u_{x2} & u_{x3} & 0 \\ u_{y1} & u_{y2} & u_{y3} & 0 \\ u_{z1} & u_{z2} & u_{z3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Total transformation:

$$R(\theta) = T(p_1) * R^T * R_z(\theta) * R * T(-p_1)$$

Rotation around arbitrary axis in OpenGL



Create matrices, multiply on CPU, upload to uniform matrices.

Geometrical method

Alternative solution in the book.

Based on projections of vectors and smart building of rotation matrices.

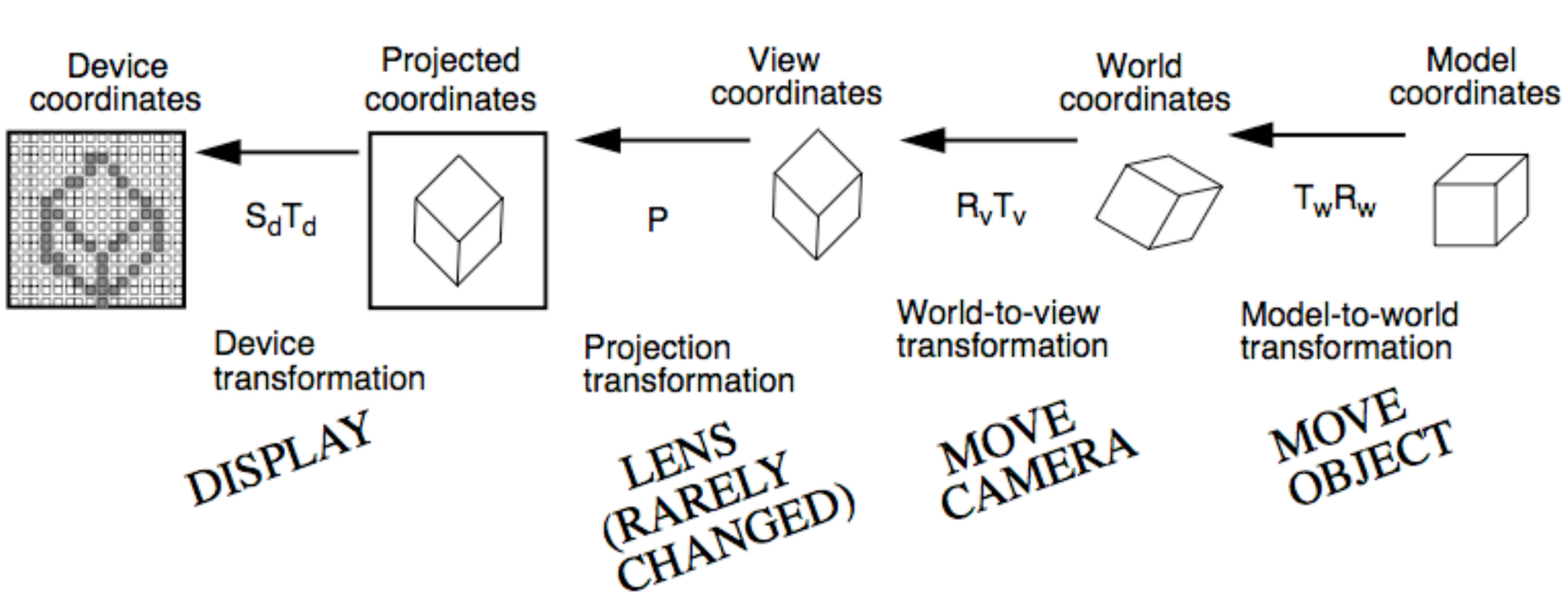
Interesting but not part of the course. No questions on it (6.3.1).

Application of rotation around arbitrary axis: Trackball control

Create rotation from user's mouse input

Picking for object selection

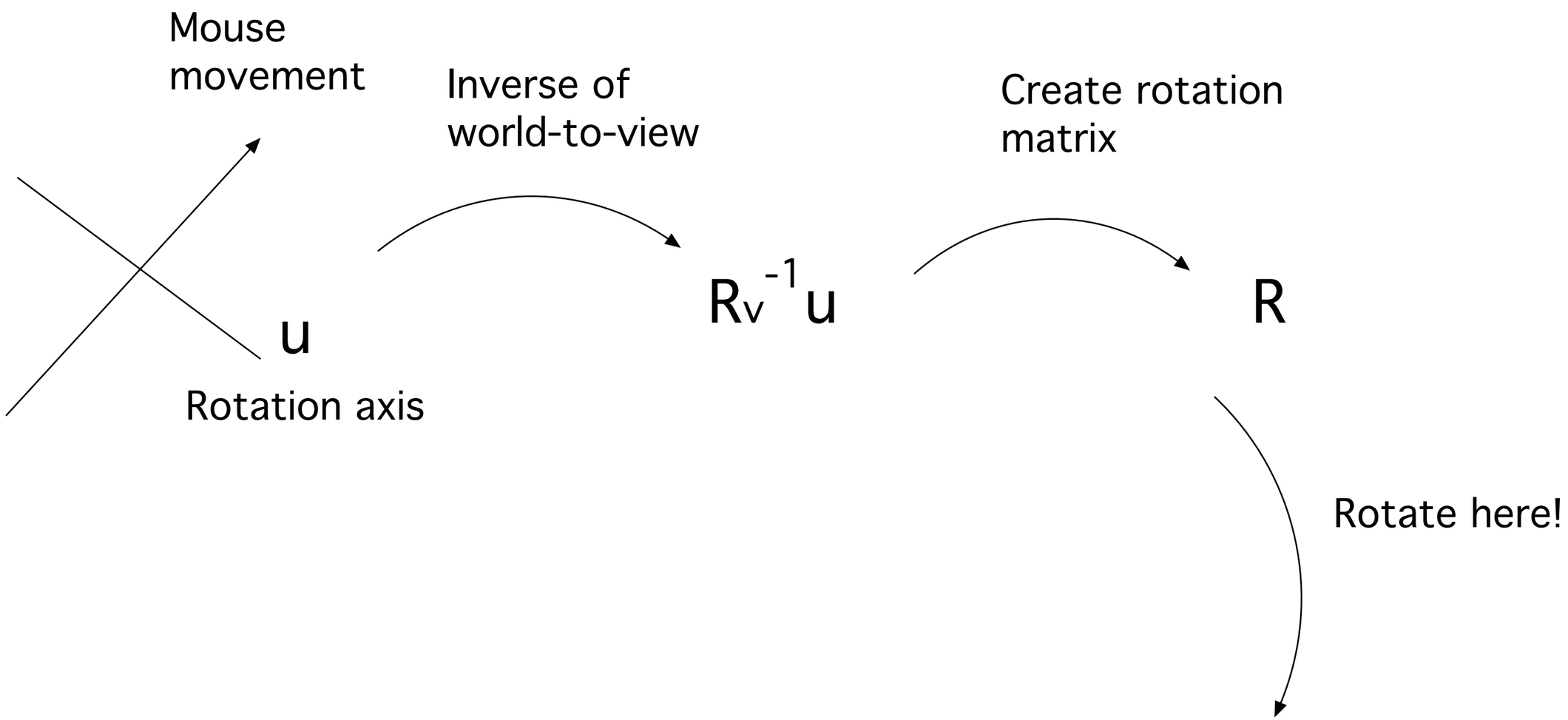
Trackball: What coordinate system?



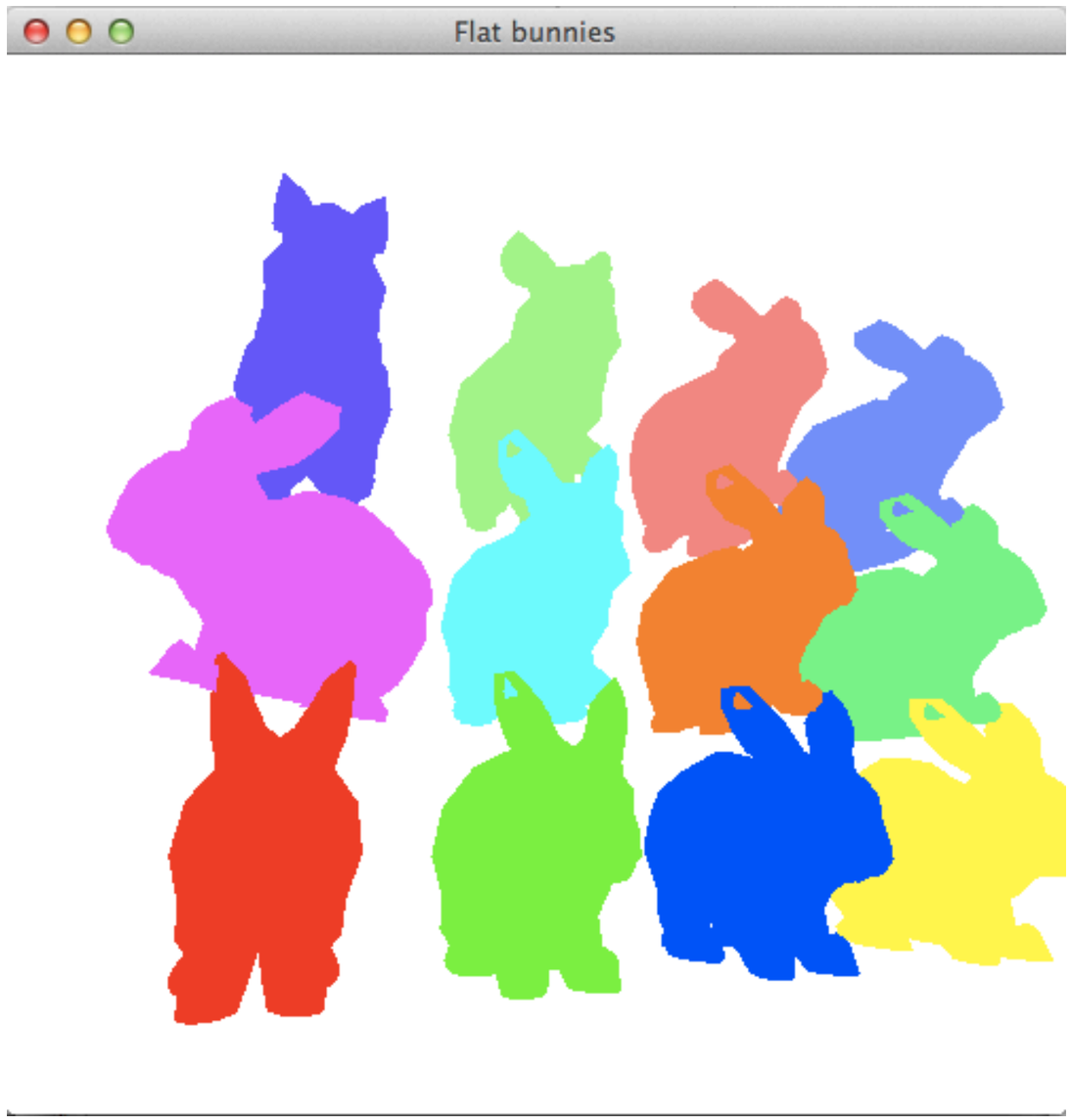
Input on screen -> view coordinates

Rotation of model -> rotate near model

Solution: Transform to model coordinates, rotations only (avoid rotation of translation)


Mouse
movement
Inverse of
world-to-view
Create rotation
matrix
u
$R_v^{-1}u$
R
Rotation axis
Rotate here!
Rv
Tv
Tw
Rw

Pretty easy, just rotation around arbitrary axis and knowing the transformation sequence!



Large worlds 1

Visible surface detection (VSD) (revisited)

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

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

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 process data that will not be used!
• don’t make many-to-many checks between polygons!

Low-level VSD

Works on pixel level or polygon level.

Always process all polygons in the scene.

Can never be sufficient for very large scenes!

Covered so far: Low-level VSD

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

Other methods:
Scan-line method
BSP trees

All polygons are treated individually

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

Clipping

Part of the OpenGL pipeline

Hardware supported

Clips away any part of a polygon that is outside the viewing frustum

One side at a time

Still individual 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.

Frustum culling
BSP trees (high level)
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:

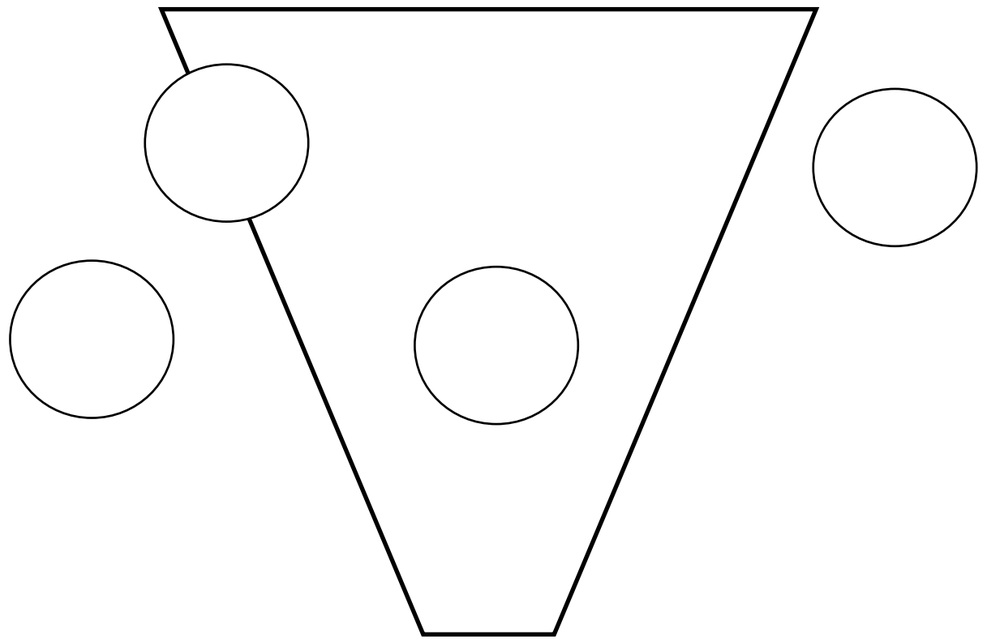
Finds most visible polygons, excludes most invisible ones
Fast. Some artifacts.

Conservative:

Finds all visible polygons but includes some invisible ones.
No artifacts. Potentially lower performance than “approximative”

Step 1. Frustum culling (View volume culling)

What polygons are inside the frustum?



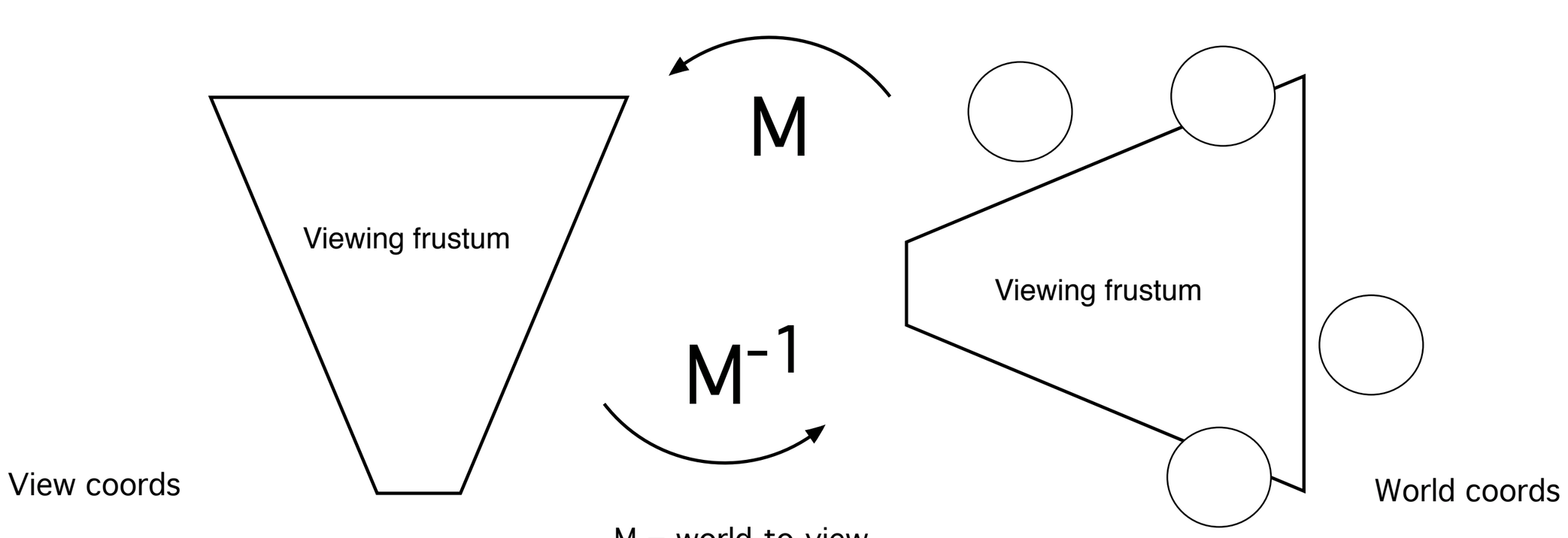
Principle: Make a subdivision of the scene, so tests can be done on groups, e.g. separate objects or limited parts of the scene.

Frustum culling

Create plane equations for each frustum side

Transform from view to world coordinates

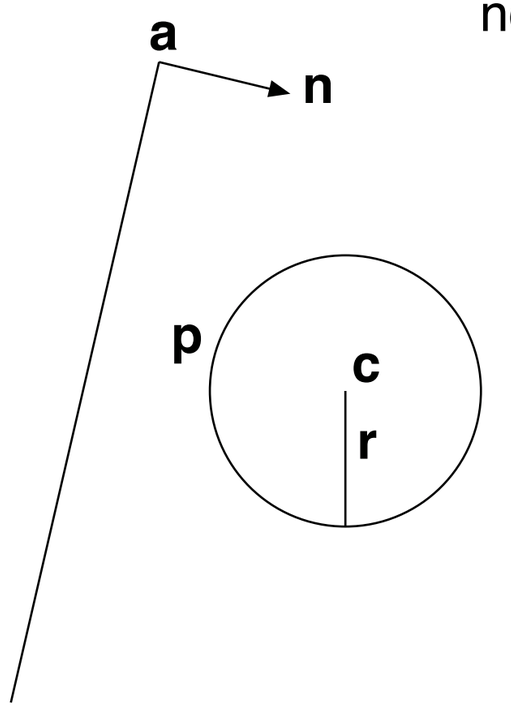
Test against e.g. bounding spheres of objects



The frustum culling test against a sphere

You need

center of sphere **c**
radius of sphere r
normalized normal of plane **n**
A point in the plane **a**



Point p nearest plane = **c** - **n**·r

Project **p** on n by **p**•**n**

Project **a** on **n** by **a**•**n**

Check sign of **a**•**n** - **p**•**n**

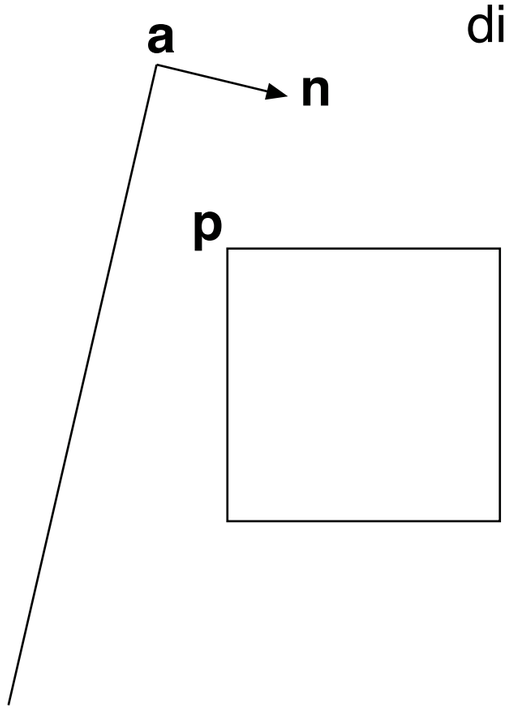
The frustum culling test against an AABB

Relevant for testing square patches!

AABB = Axis Aligned Bounding Box

You need

dimensions of cube, top, left, bottom, right, near, far
normalized normal of plane **n**
A point in the plane **a**



Testing all corners works, but:

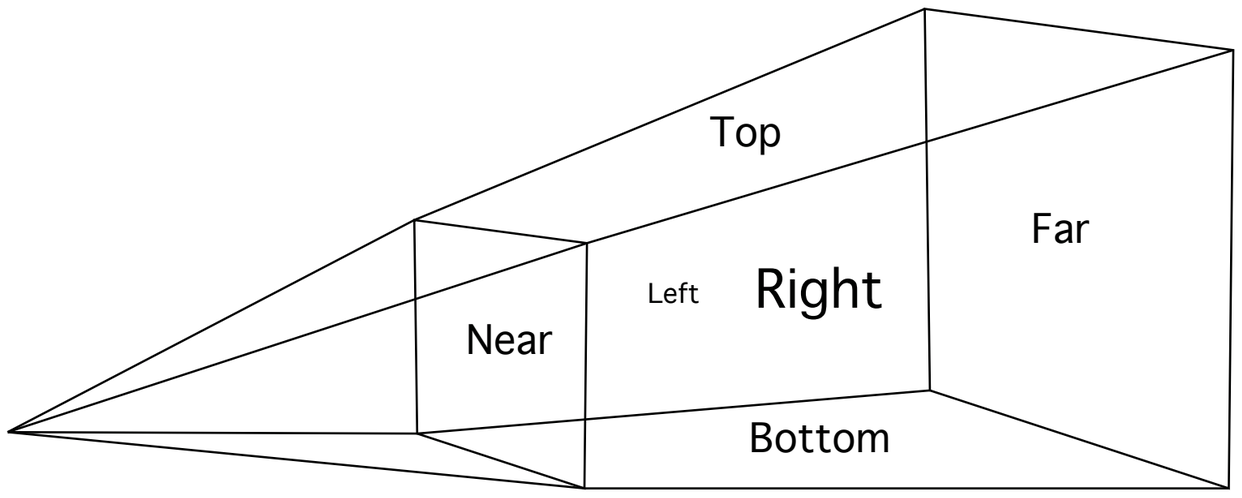
Find relevant corner (farthest along -n) from signs of the normal!

Test that point with dot product like before:
Check sign of **a•n - p•n**

What planes to test?

Which ones do you think?

All of them?

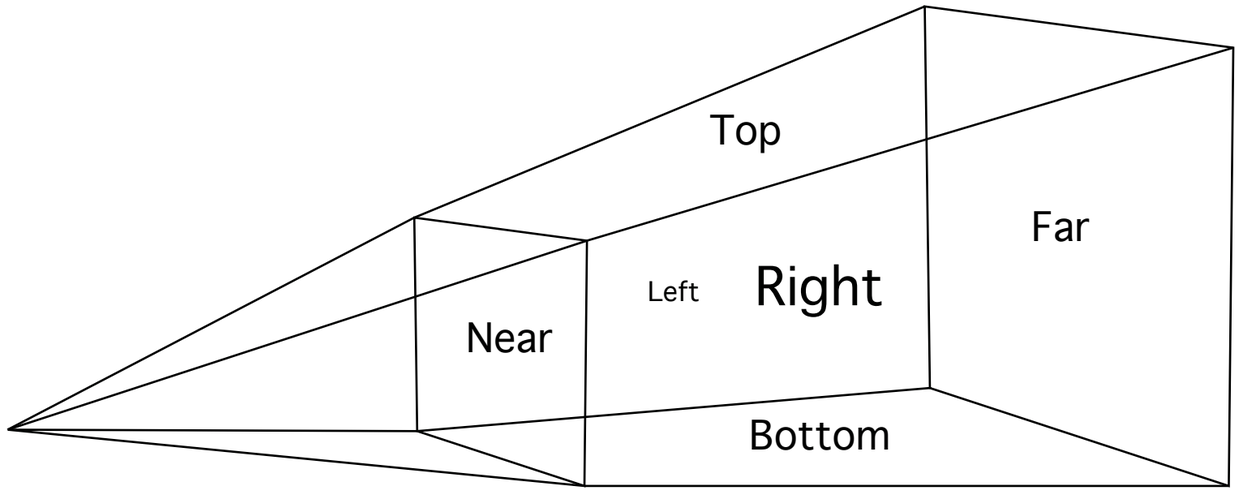


What planes to test?

You never need to test the near plane! (Why?)

You usually need right, left and far.

Top and bottom are often unnecessary - especially when discussing terrains!



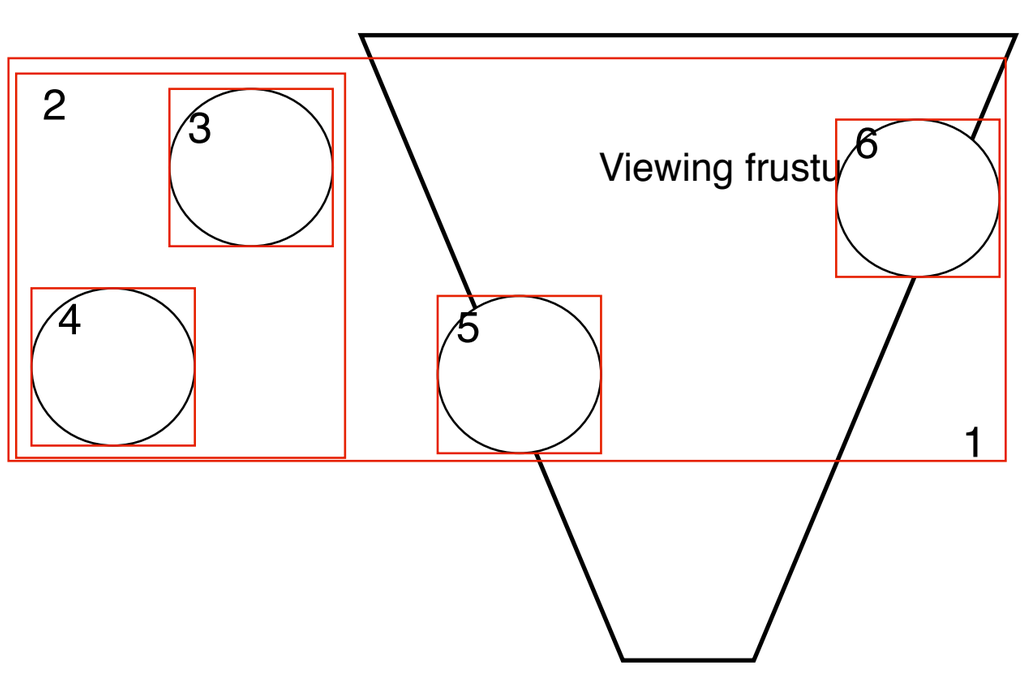
Going from single objects to groups of objects

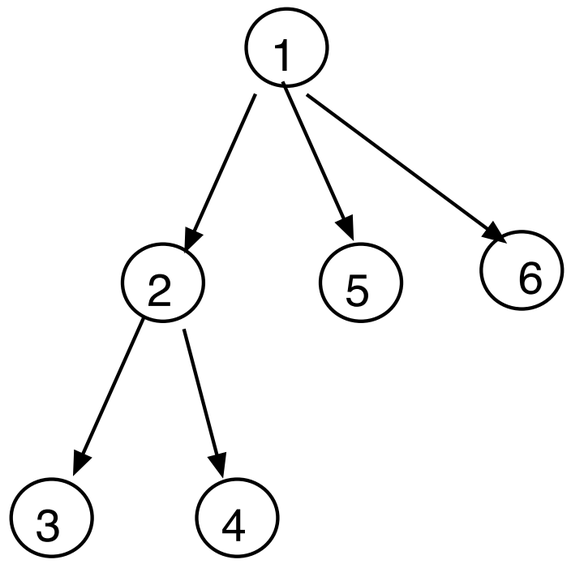
The methods above will handle a single object at a time - a great improvement!

But how about groups? Structures for only finding objects nearby?

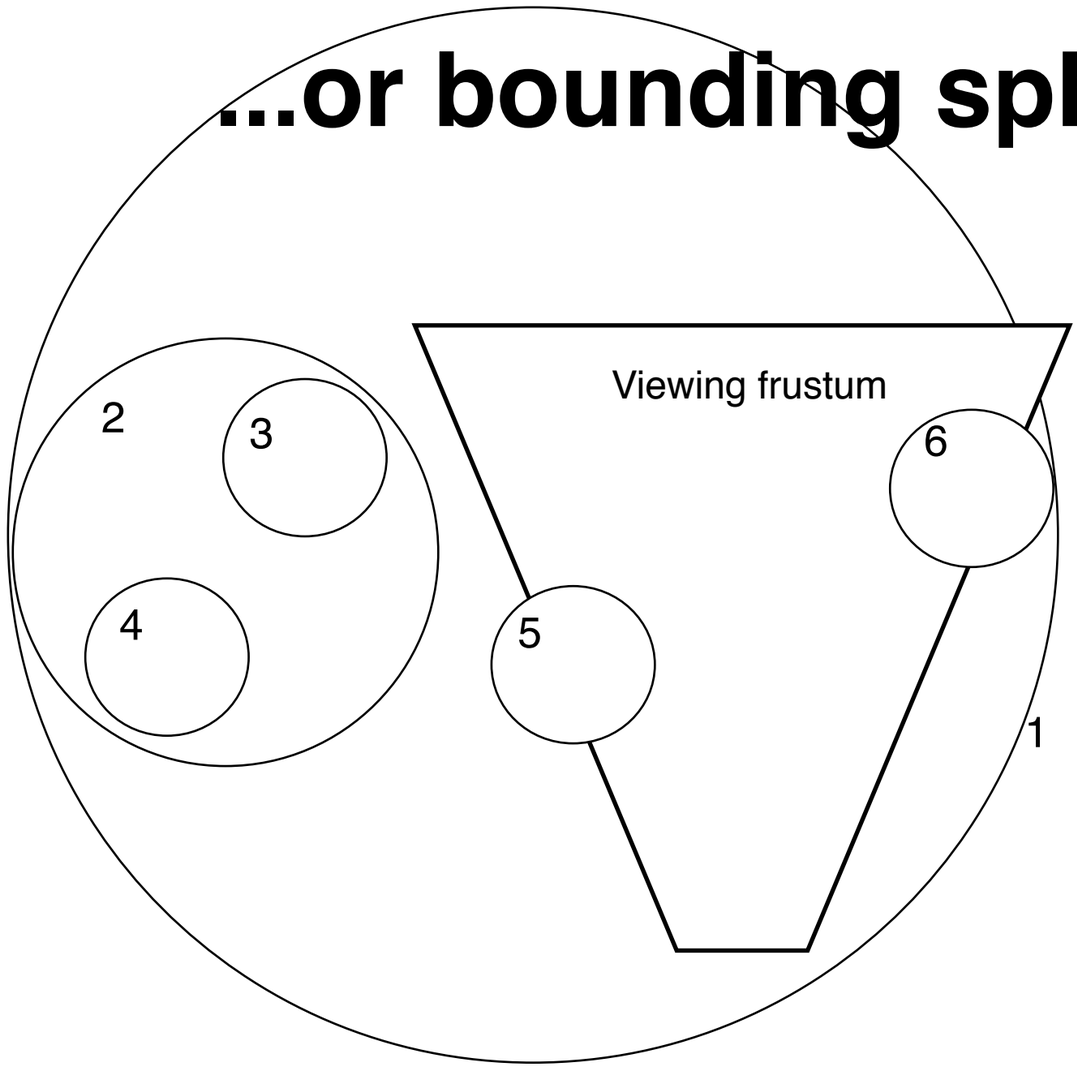
- Groups of objects
- Scene subdivision

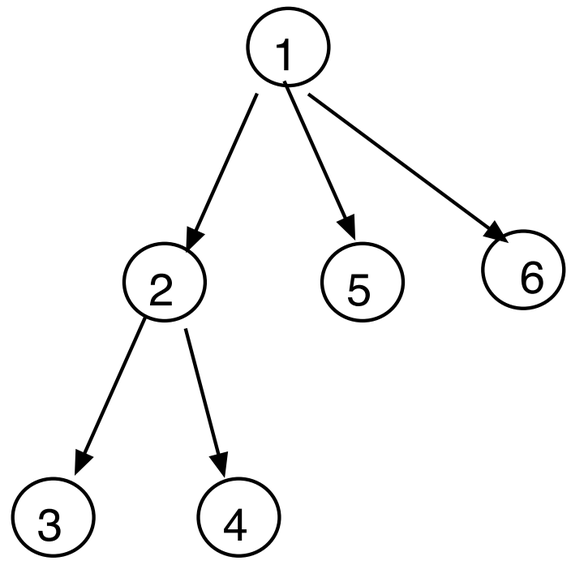
Frustum culling using a group hierarchy with bounding boxes





...or bounding spheres





BSP trees for high-level VSD

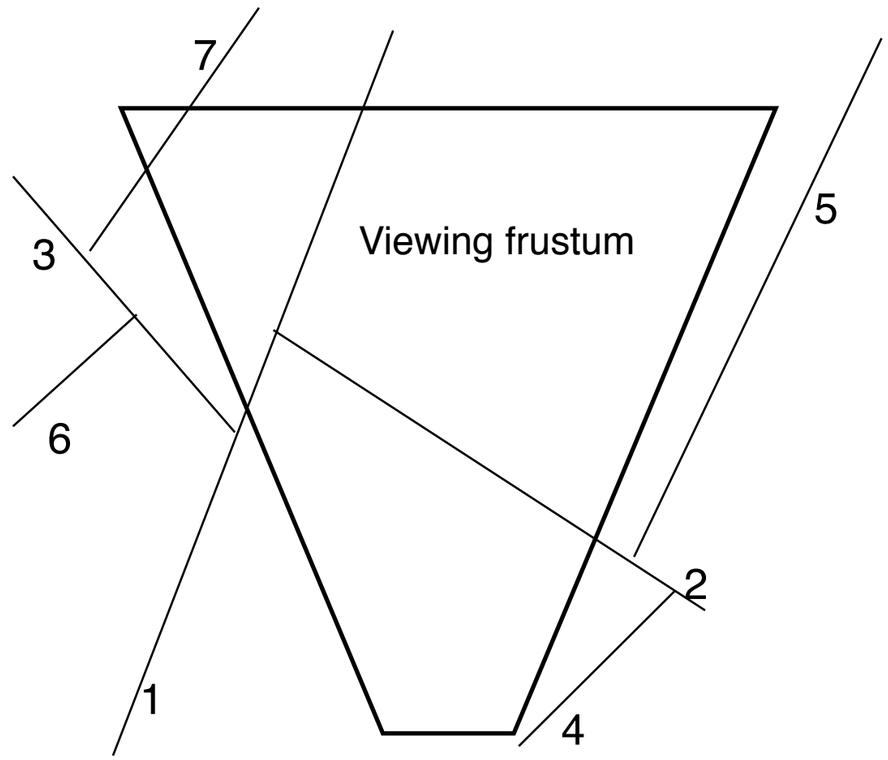
BSP trees simpify frustum culling!

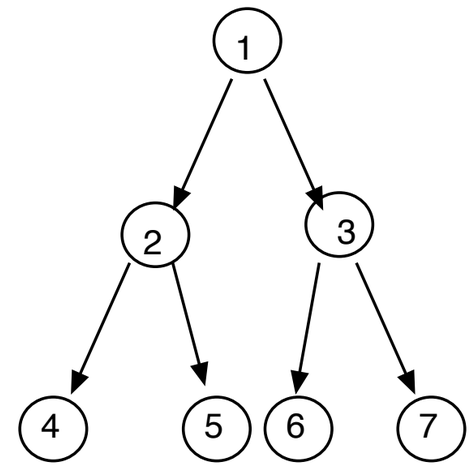
Any node in a BSP tree is a convex volume!

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

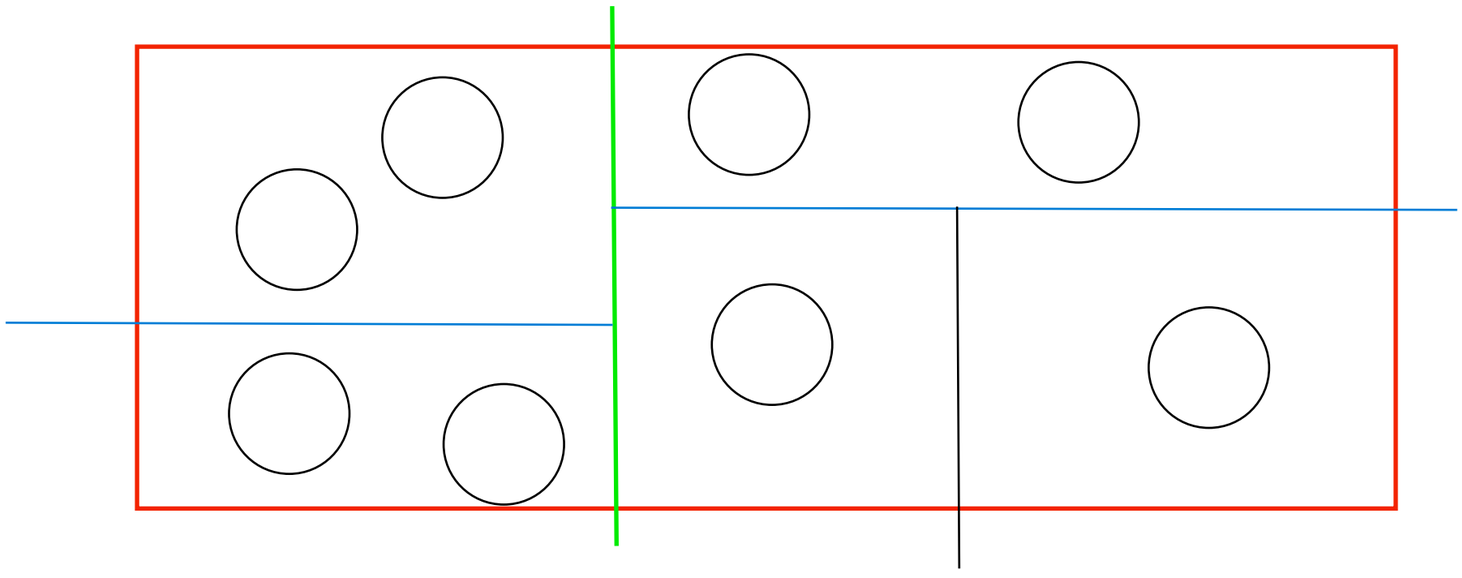




Frustum culling using a BSP tree:

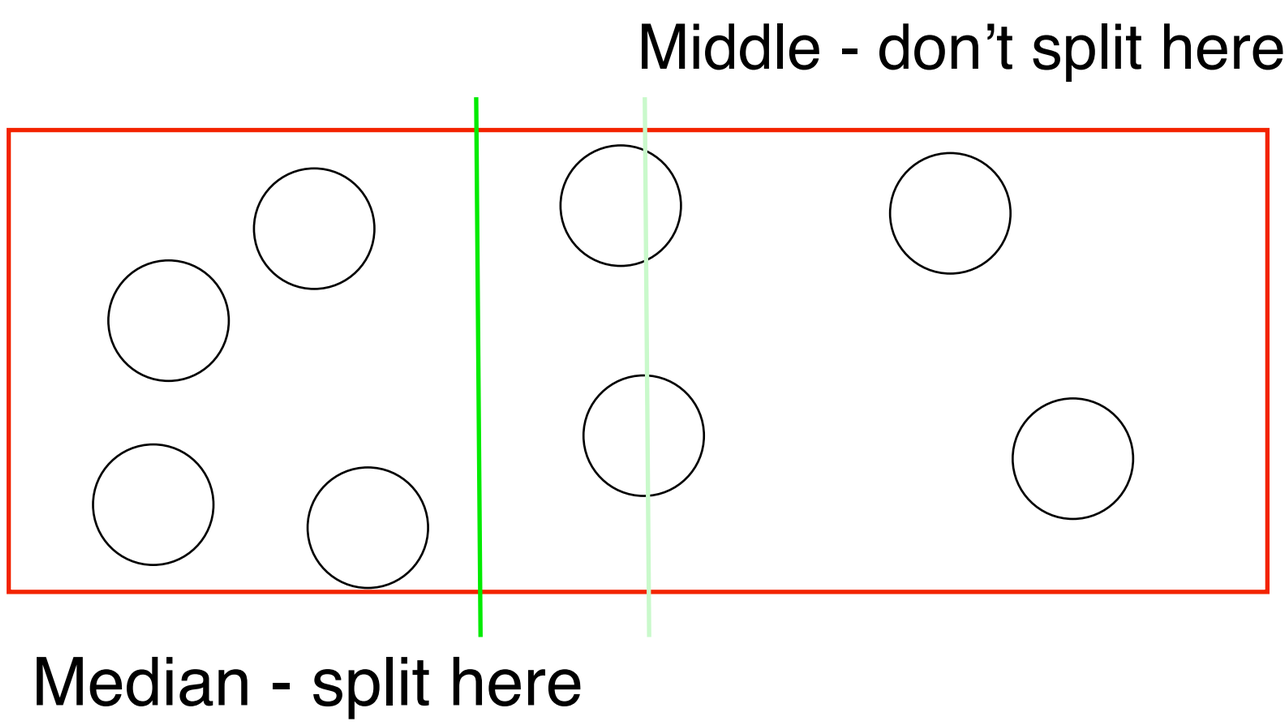
Usually axis aligned - ”kd-tree”

Very simple tests



Building a kd-tree

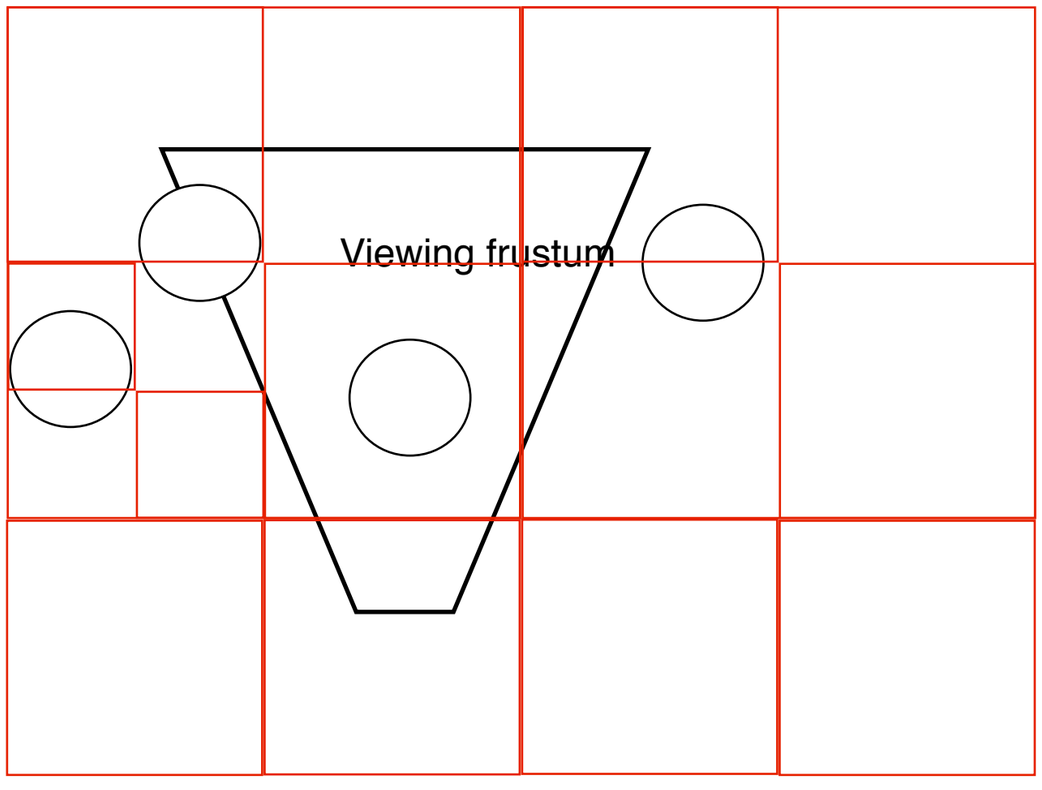
Split at median: Half of the geometry in each side of the splitting plane. Balanced kd-tree.



Octrees:

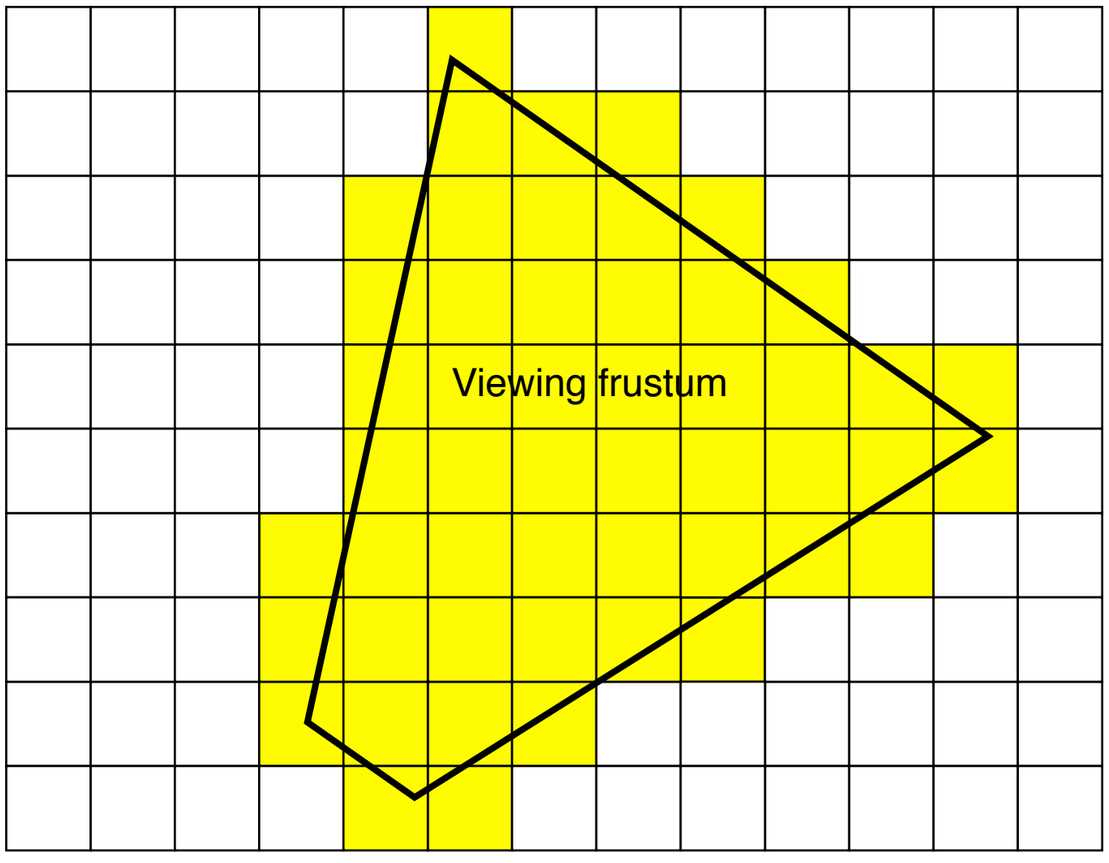
Non-uniform hierarcical space subdivision

Split cells in 8 until sufficient simplicity is achieved.



Uniform space subdivision

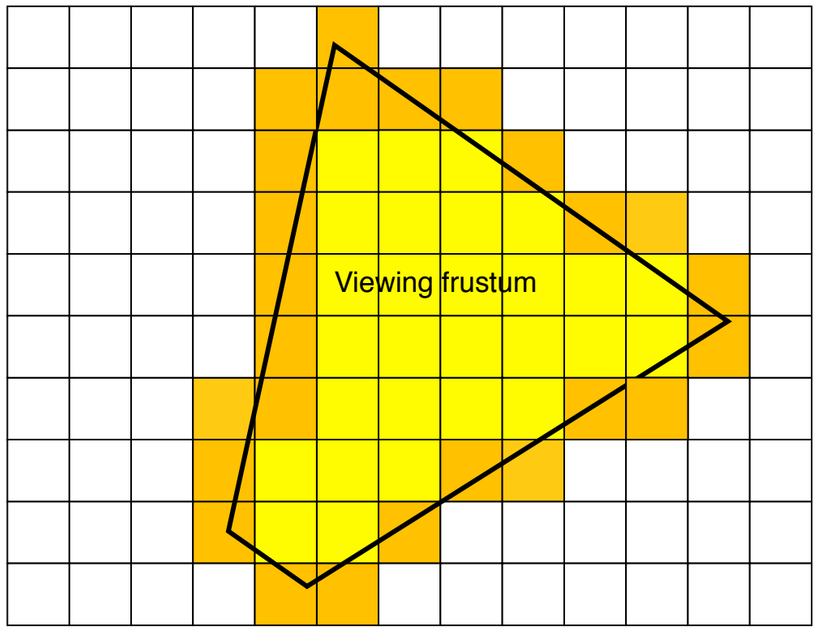
Subdivide by a regular grid. Every object is in a list for each grid cell. Use extra margin for the size of the objects!



Simple common case: Terrain defined as multiple patches of same size; forms a regular grid

Map the frustum edges to the grid coordinates

Draw all patches 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.

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.

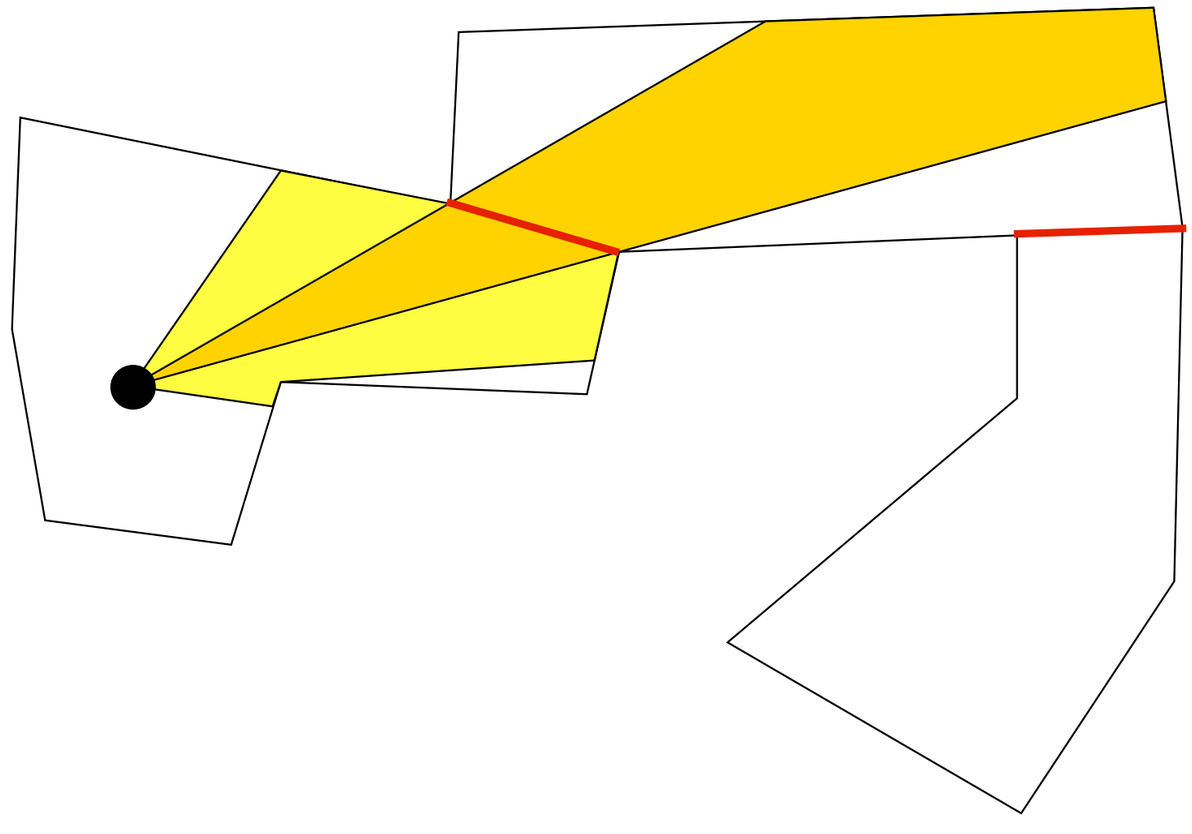
Split the world into smaller parts, create connections as “portals” between them. (Dark Forces, Tomb Raider)

Each portal is a branch in an adjacency graph

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

Portals

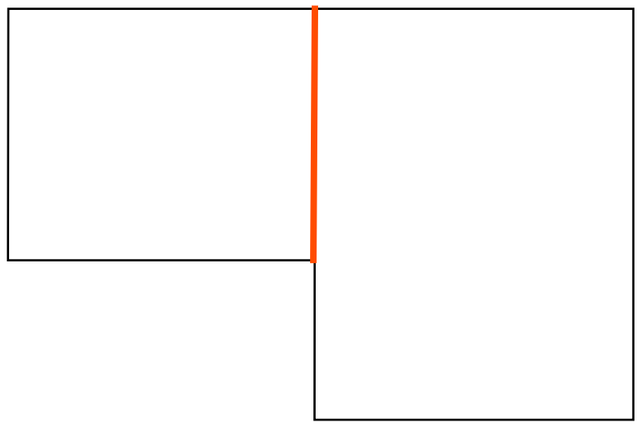
Polygons are grouped into cells, “rooms”



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

Real-world example: Dark Forces

Level editor reveals portal-based engine

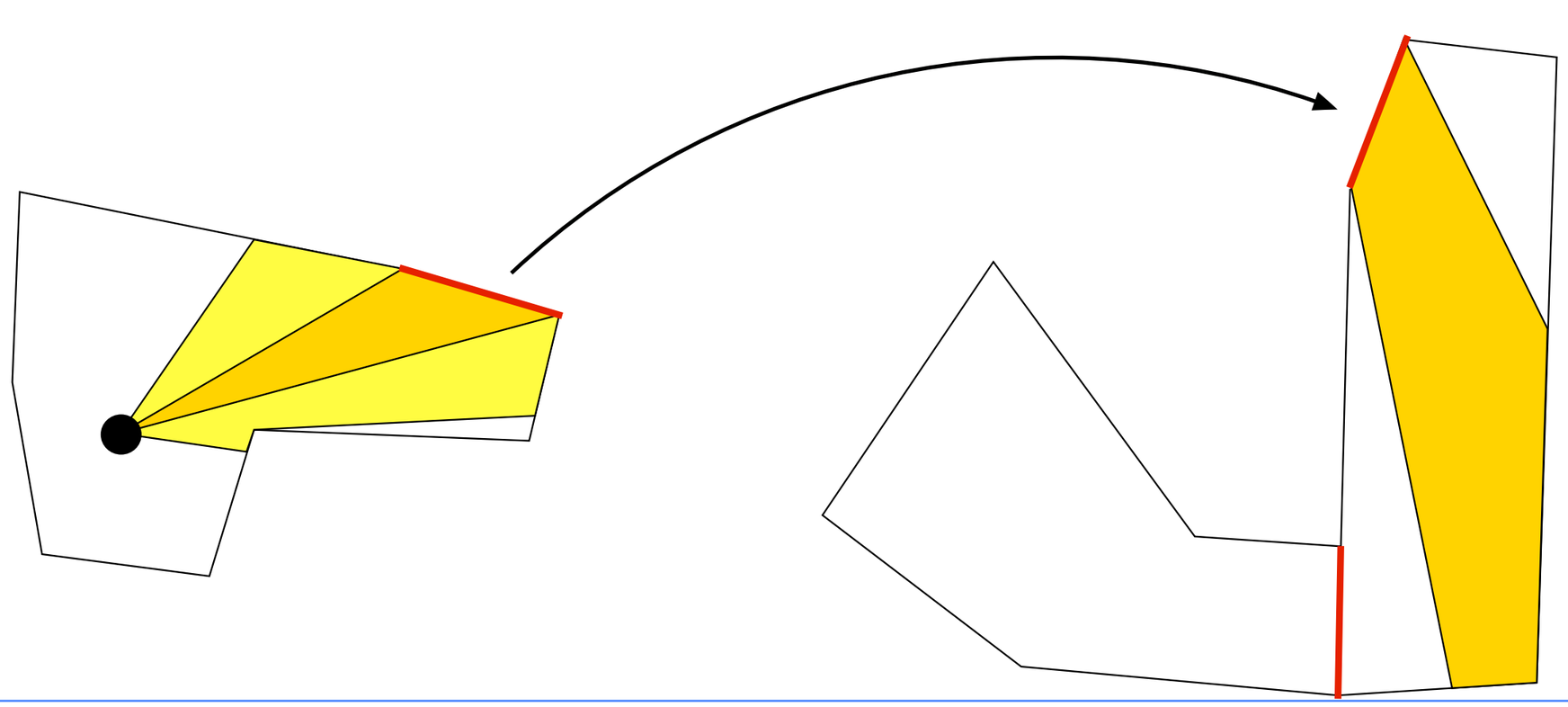


Edit levels by drawing 2D polygons, connect them as portals

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!

Portals are

- An optimization method
- A special effect method

Both are based on the same principles.

Types of portals for effects:

- Conflict free environments. Works with course material.
- Ray-tracing portals, also feasible

These need , advanced course material:

- Stencil portals
- Texture portals
- FBO portals

Potentially visible set (PVS)

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

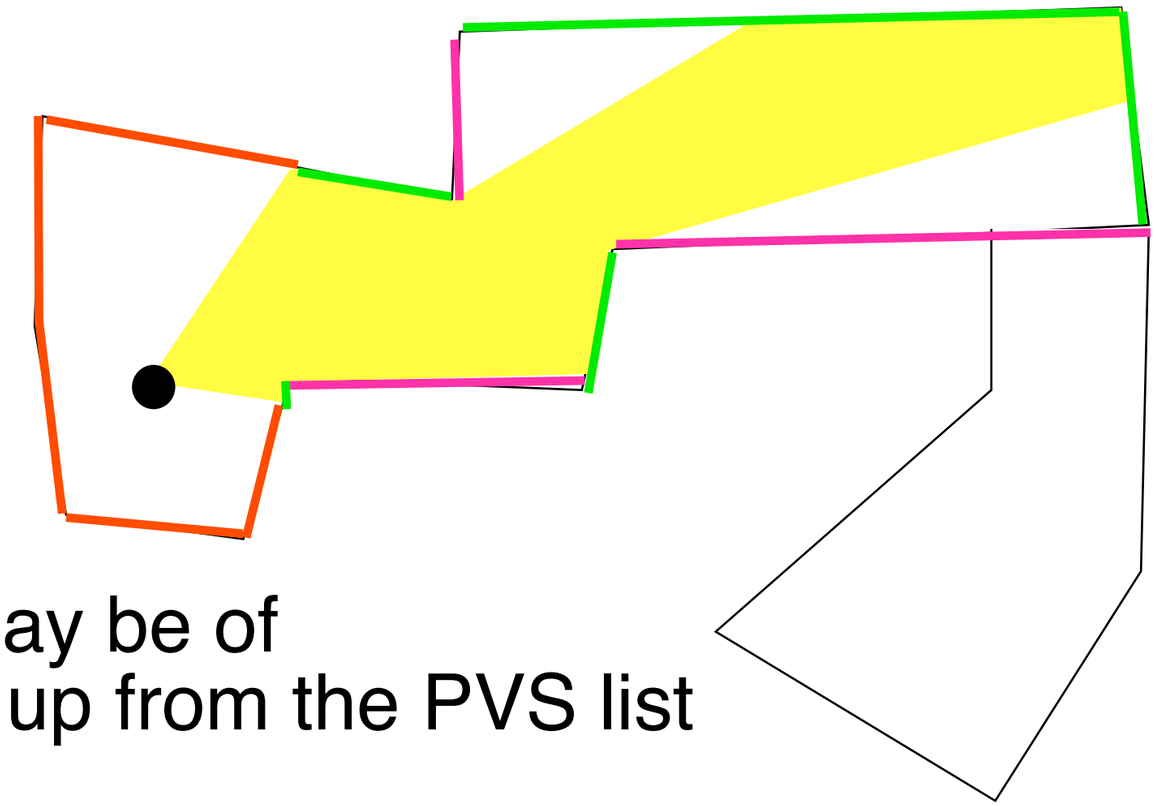
The list is huge, but can be compressed.

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

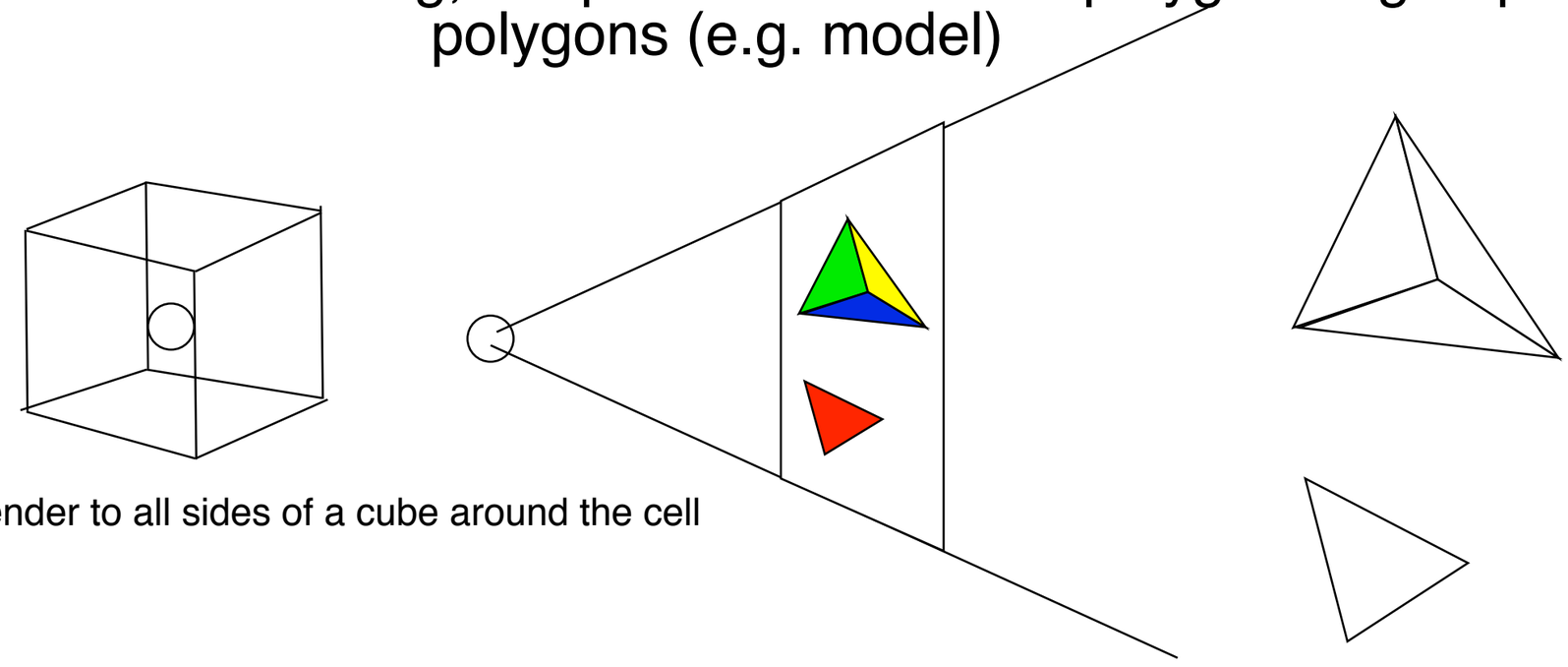
1) Image-space method

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

Next lecture

Some fun with portals

Level of detail

Billboards

Shape representation, continued