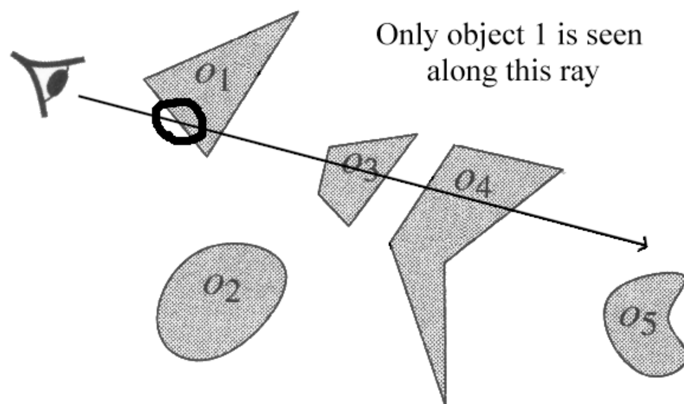
Drawing the Visible Objects

We want to generate the image that the eye would see, given the objects in our space

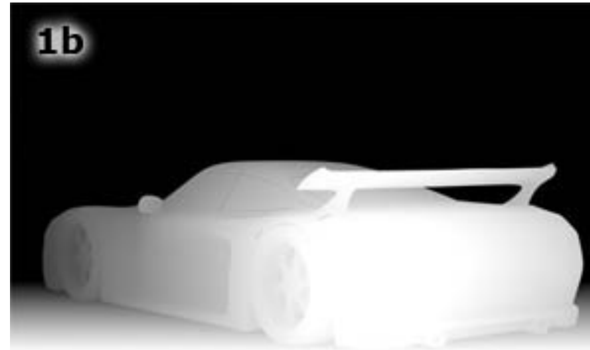
How do we draw the correct object at each pixel, given that some objects may obscure others in the scene?



Hidden surface removal

A Simple Solution:

- Keep a buffer that holds the z-depth of the pixel currently at each point on screen
- Draw each polygon: for each pixel, test its depth versus current screen depth to decide if we draw it or not



Z-buffer

Drawbacks to Z-buffering

This used to be a **very expensive** solution!

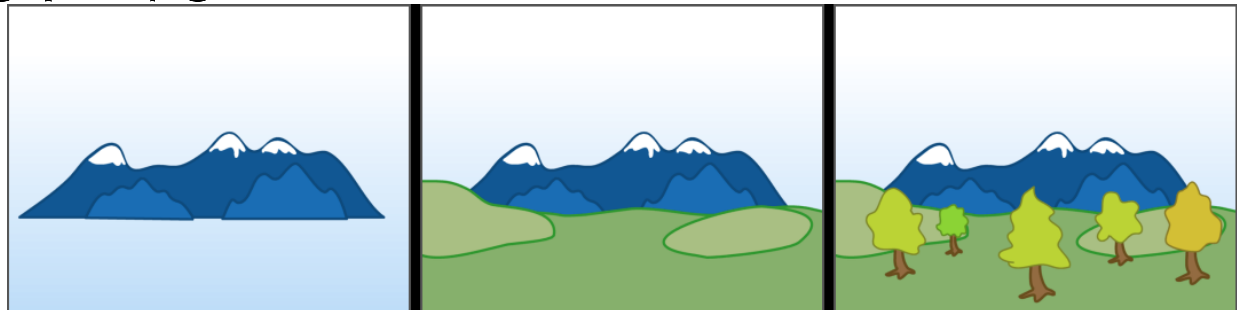
- Requires memory for the z-buffer
  - extra hardware cost was prohibitive

- Requires extra z-test for every pixel

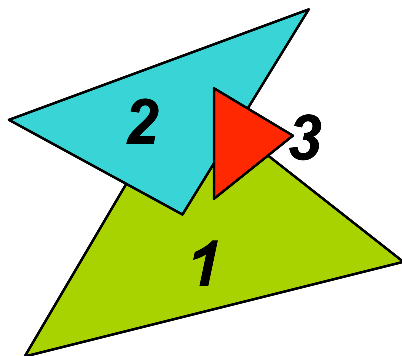
So, a software solution was developed …

The Painter's Algorithm

Avoid extra z-test & space costs by scan converting polygons in back-to-front order



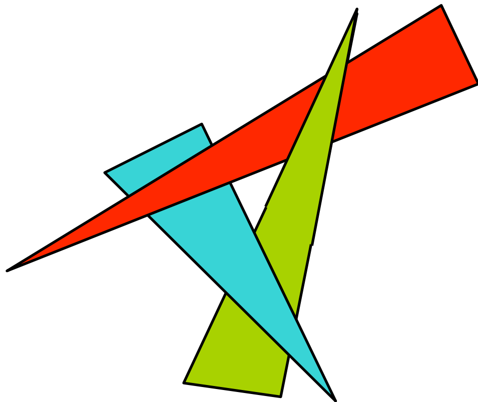
From wikipedia



Is there always a correct back-to-front order?

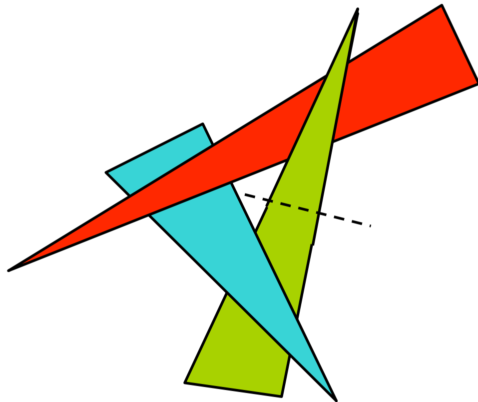
How Do We Deal With Cycles?

In 3 dimensions, polygons can overlap, creating cycles in which no depth ordering would draw correctly



How do we deal with these cases?

How Do We Deal With Cycles?



We can break one polygon into two and order them separately

- Which polygon?
- Where?

View dependent

BSP Trees

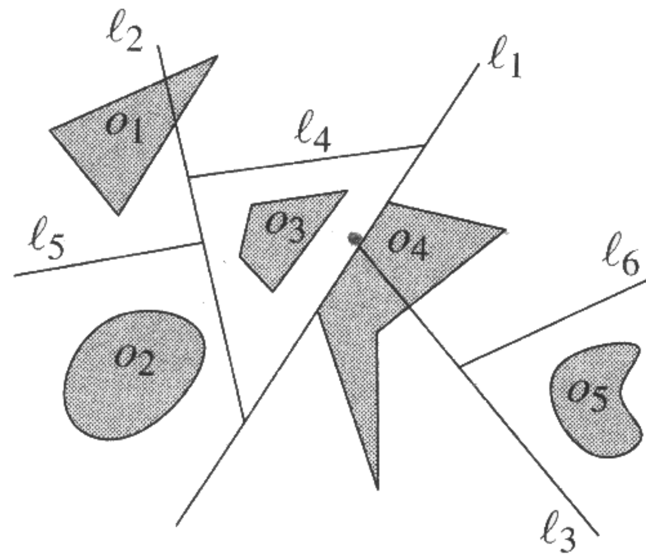
Having a pre-built BSP tree will allow us to get a correct depth order of polygons in our scene for any point in space.

We will build a data structure based on the polygons in our scene, that can be queried with any point input to return an ordering of those polygons.

The Big Picture

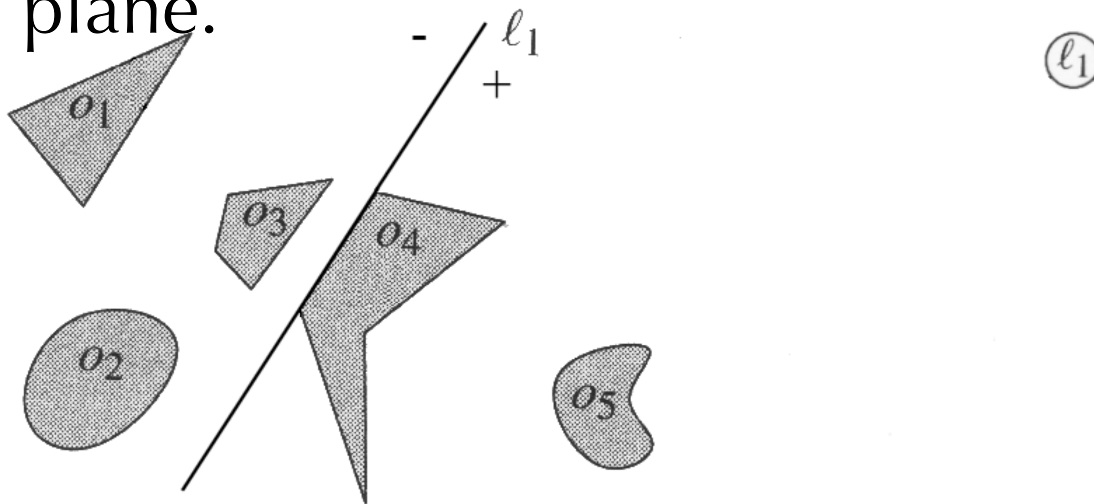
Assume that no objects in our space overlap

Use planes to recursively split our object space, keeping a tree structure of these recursive splits.



Choose a Splitting Line

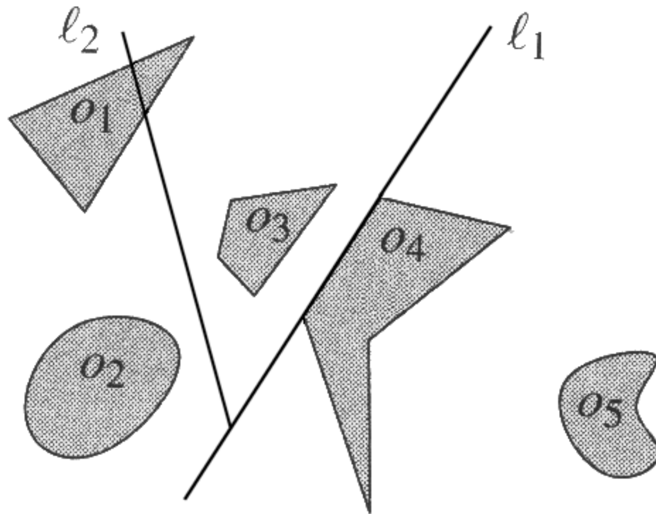
Choose a splitting plane, dividing our objects into three sets – those on each side of the plane, and those fully contained on the plane.

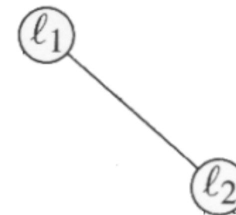


Choose More Splitting Lines

What do we do when an object (like object 1) is divided by a splitting plane?

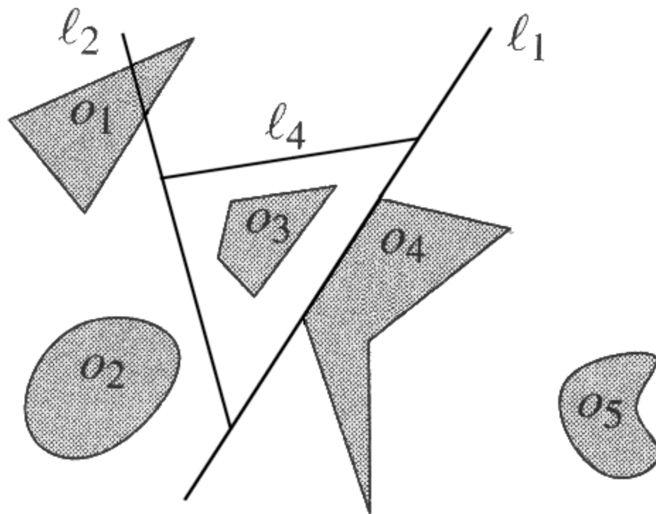
It is divided into two objects, one on each side of the plane.

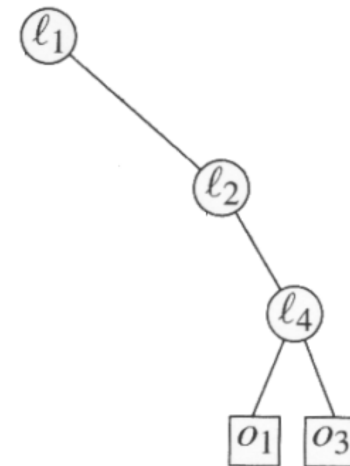




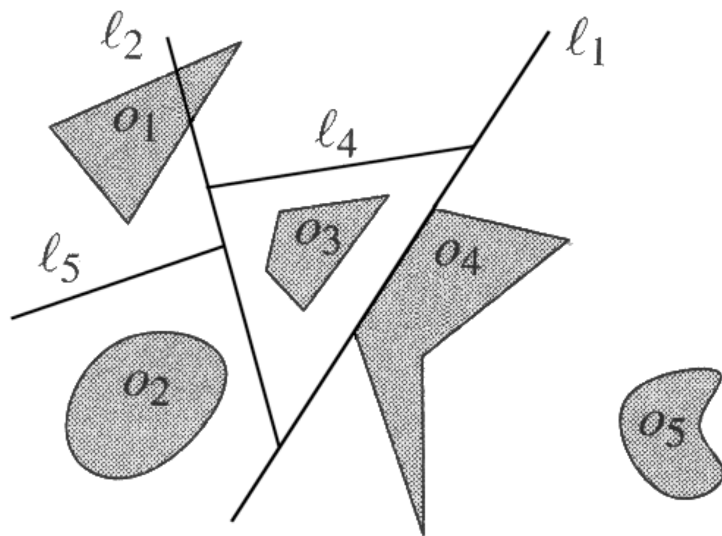
Split Recursively Until Done

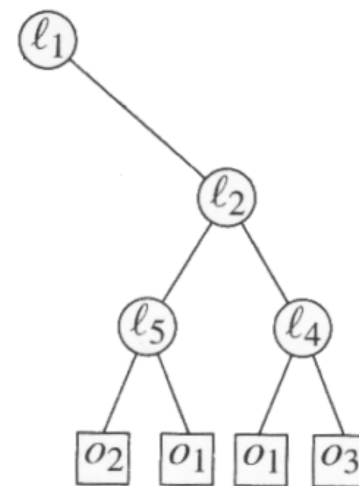
When we reach a convex space containing exactly zero or one objects, that is a leaf node.



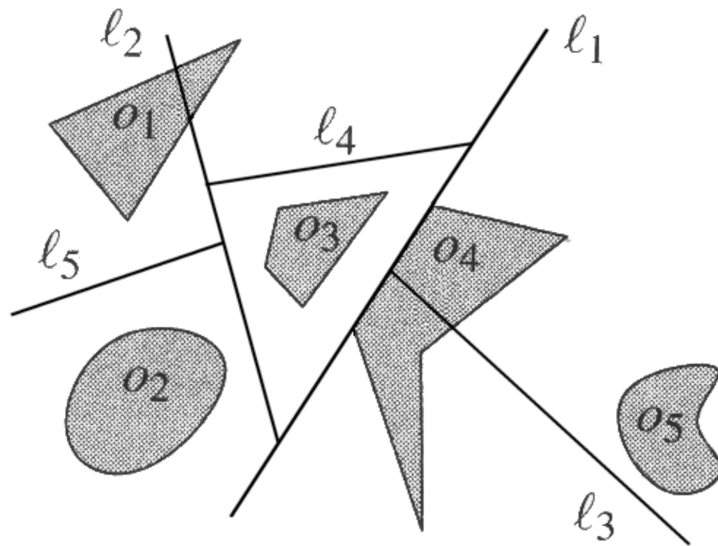


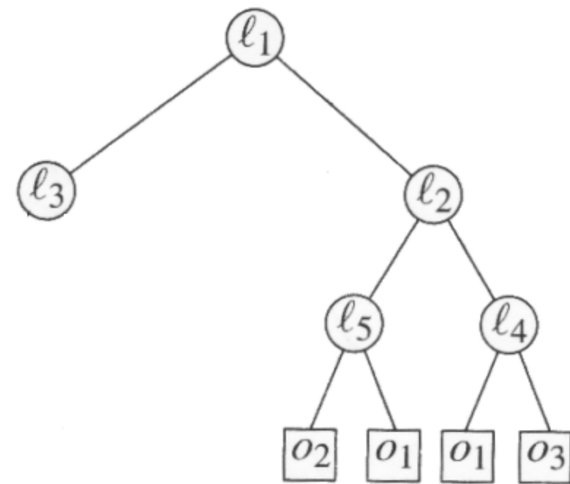
Continue





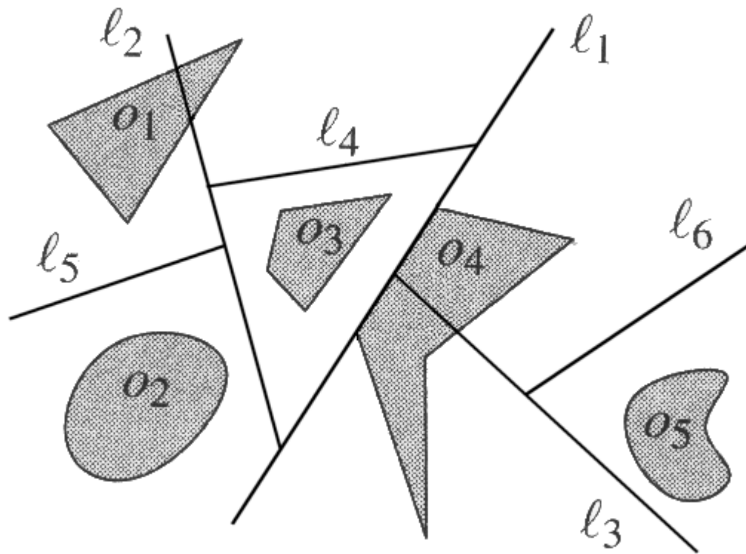
Continue

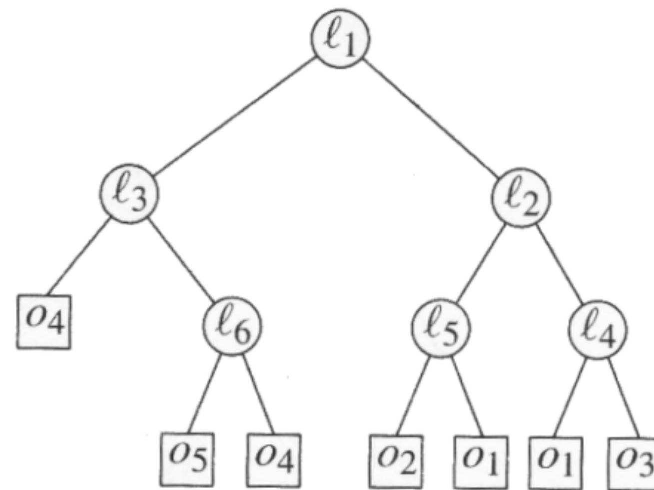




Finished

Once the tree is constructed, every root-to-leaf path describes a single convex subspace.



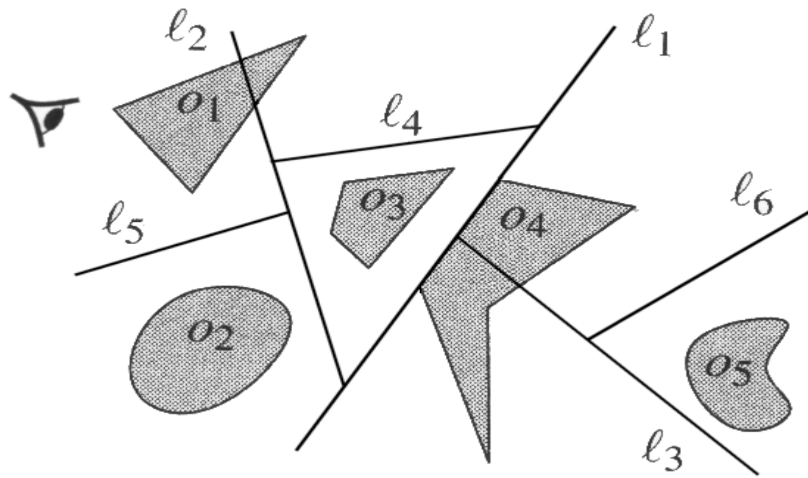


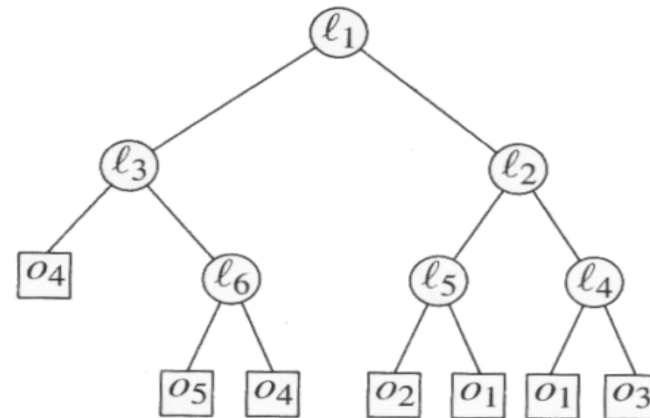
Querying the Tree

If a point is in the positive half-space of a plane,

- everything in the negative half-space is farther away -- so draw it first, using this algorithm recursively
- draw objects on the splitting plane
- draw objects into the positive half-space, recursively

What Order Is Generated From This Eye Point?





How much time does it take to query the BSP tree, asymptotically?

Structure of a BSP Tree

- Each internal node has a +half space, a -half space, and a list of objects contained entirely within that plane (if any exist).

- Each leaf has a list of zero or one objects inside it, and no subtrees

- The *size* of a BSP tree is the total number of objects stored in the leaves & nodes of the tree
  - This can be larger than the number of objects in our scene because of splitting

K-d tree and Octree are special cases of BSP

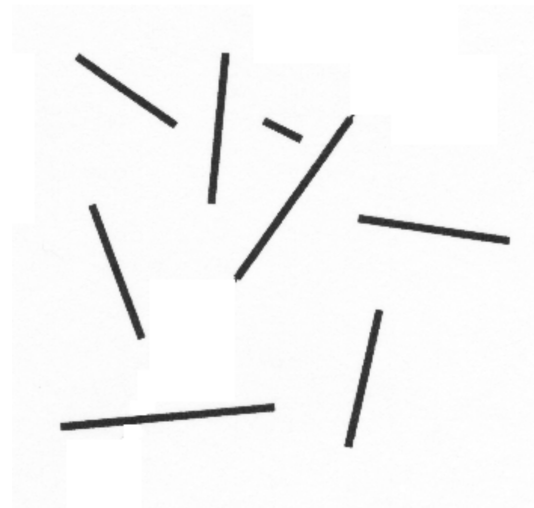
Building a BSP Tree

- How do we pick splitting lines/planes?

Building a BSP Tree From Line Segments in the Plane

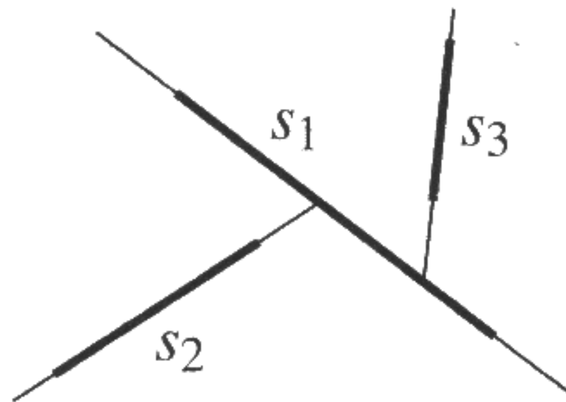
We'll now deal with a formal algorithm for building a BSP tree for line segments in the 2D plane.

This will generalize for building trees of D-1 dimensional objects within D-dimensional spaces.



Auto-Partitioning

Auto-partitioning: splitting only along planes coincident on objects in our space



Algorithm for the 2d Case

- If we only have one segment 's', return a leaf node containing s.
- Otherwise, choose a line segment 's' along which to split

- For all other line segments, one of four cases will be true:
    1) The segment is in the +half space of s
    2) The segment is in the -half space of s
    3) The segment crosses the line through s
    4) The segment is entirely contained within the line through s

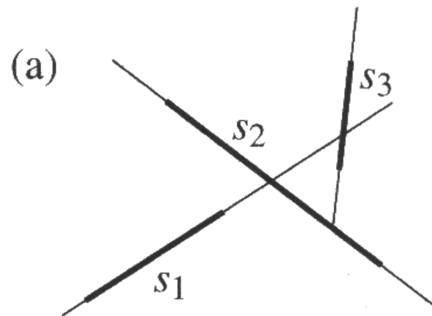
- Split all segments who cross 's' into two new segments -- one in the +half space, and one in the -half space

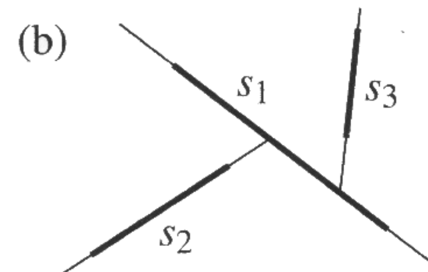
- Create a new BSP tree from the set of segments in the +half space of s, and another on the set of segments in the -half space

- Return a new node whose children are these +/- half space BSP's, and which contains the list of segments entirely contained along the line through s.

How Small Is the BSP

- Different orderings result in different trees





- Greedy approach?
  - pick the line with fewest intersections
  - doesn't always work -- sometimes it does very badly
  - it is costly to find

Random Approach Works Well

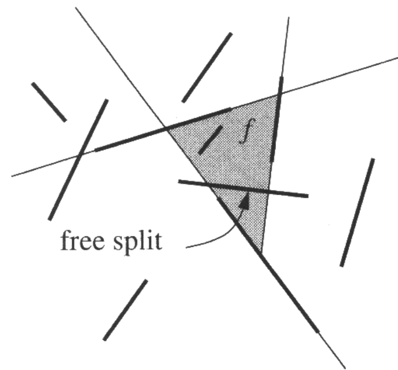
If we randomly order segments before building the tree, then we do well in the average case

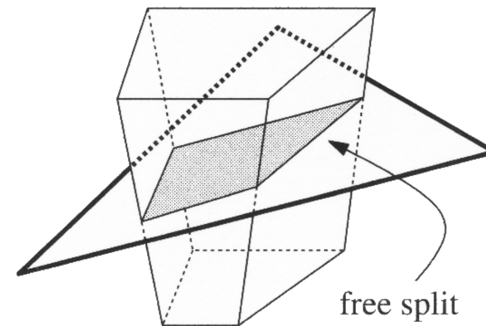
- Expected number of fragments, i.e., size of leaves $O(n \log n)$

- Expected running time: $O(n^2 \log n)$

Optimization: Free Splits

- Sometimes a segment will entirely cross a convex region described by some interior node -- if we choose that segment to split on next, we get a "free split" -- no splitting needs to occur on the other segments since it crosses the region



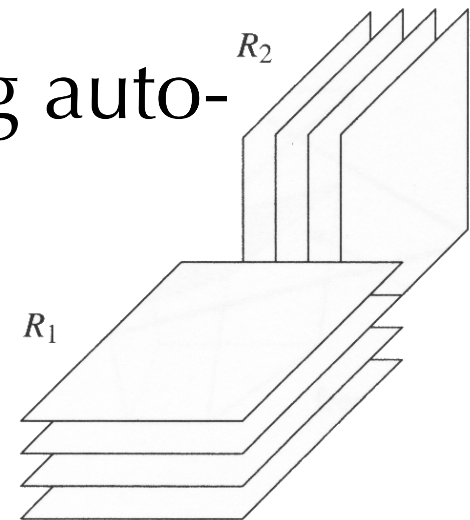


Our Building Algorithm Extends to 3D!

The same procedure we used to build a tree from two-dimensional objects can be used in the 3D case – we merely use polygonal faces as splitting planes, rather than line segments.

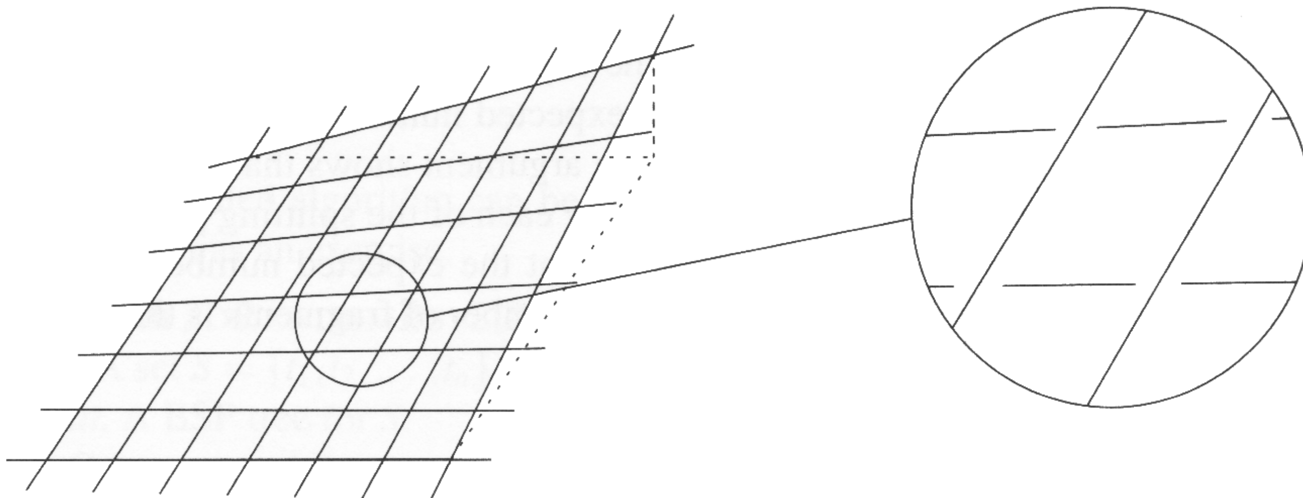
More analysis: How good are auto-partitions?

- There are sets of triangles in 3-space for which auto-partitions guarantee $\Omega(n^2)$ fragments
- Can we do better by not using auto-partitions?



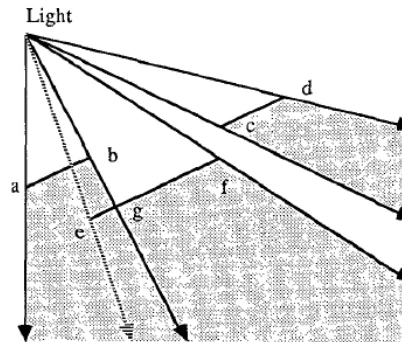
Sometimes, No

There are actually sets of triangles for which *any* BSP will have $\Omega(n^2)$

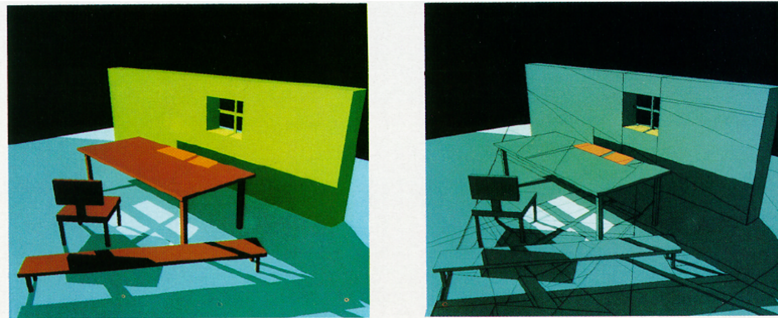


Fortunately, these cases are artificial and rarely seen…
BSP usually performs well in practice

More applications



- Querying a point to find out what convex space it exists within
- Shadows



- Visibility
- Blending
  - Opengl stuff