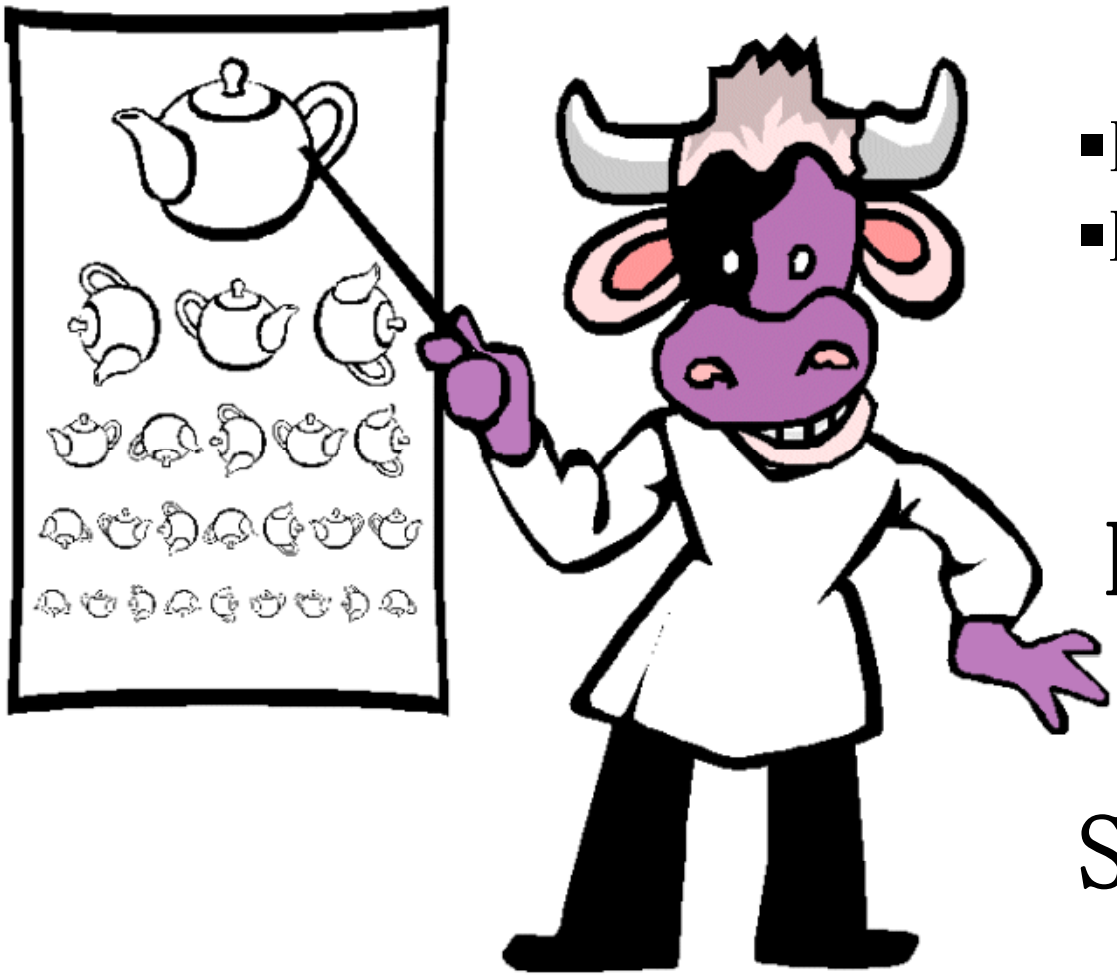
Visibility



- Portal-portal
- BSP Trees

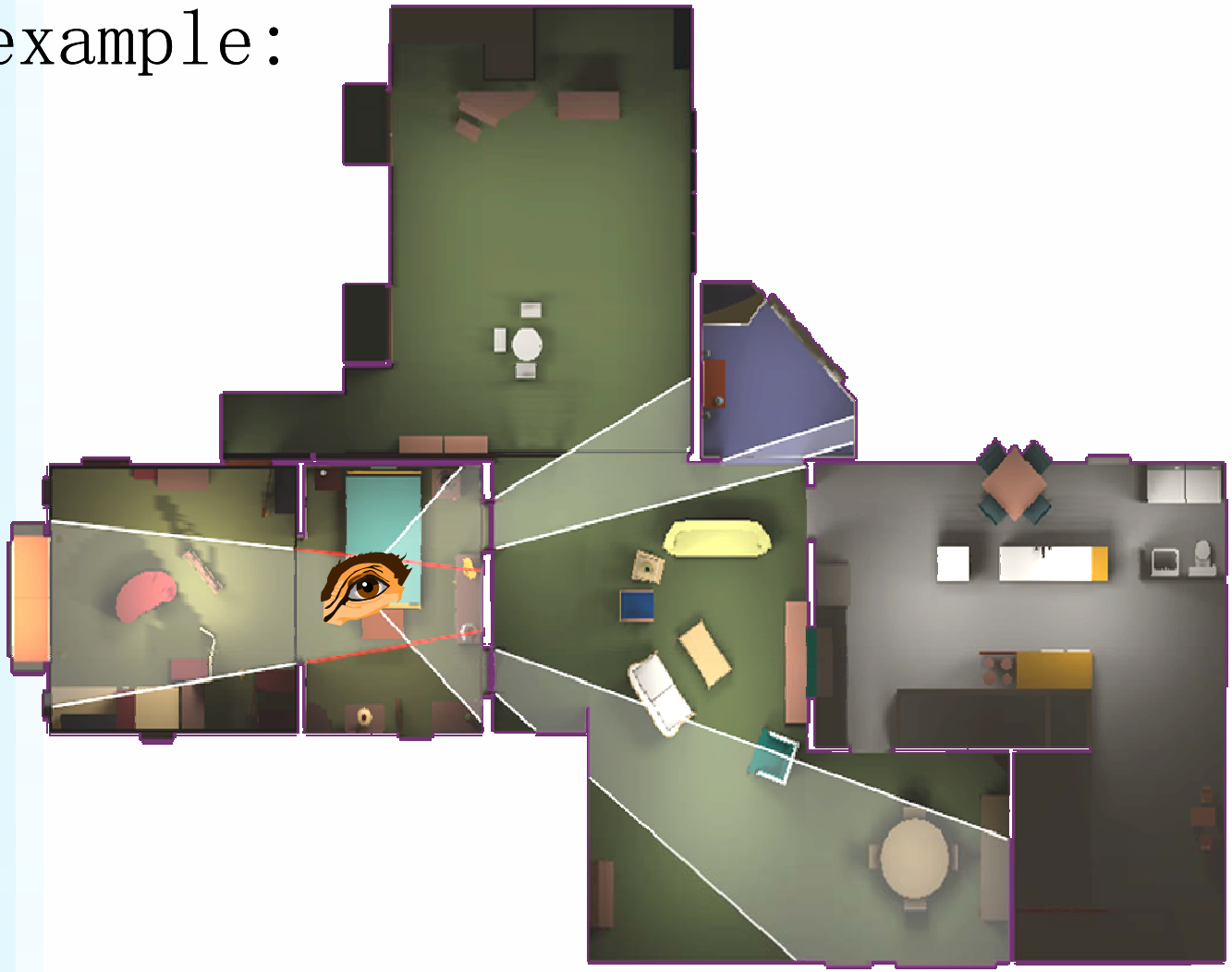
Lecture 10

CISC 829

Spring 2008

Cells & Portals

- An example:



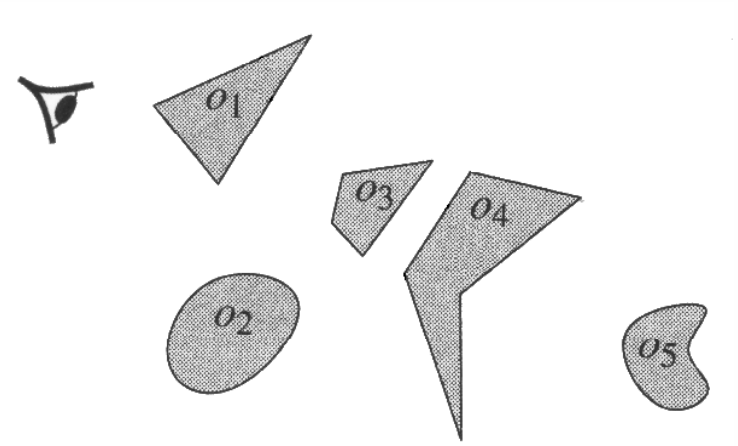
Cells & Portals

- Idea:
  - Cells form the basic unit of PVS
  - Create an *adjacency graph* of cells
  - Starting with cell containing eyepoint, traverse graph, rendering visible cells
  - A cell is only visible if it can be seen through a sequence of portals
    - So cell visibility reduces to testing portal sequences for a *line of sight…*

Motivation for BSP Trees: The Visibility Problem

We have a set of objects (either 2d or 3d) in space.

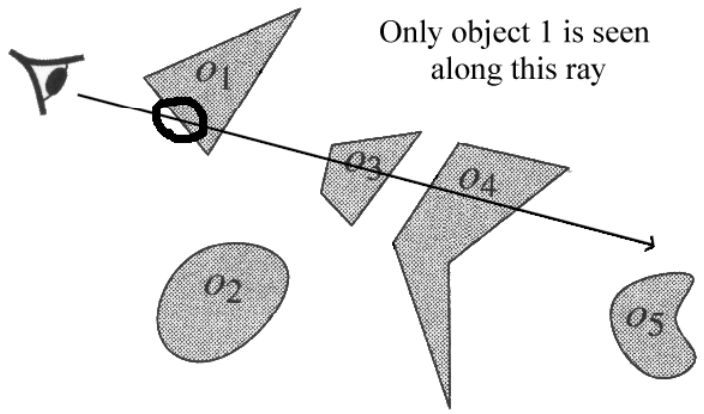
We have an “eye” at some point in this space, looking at the objects from a particular direction.



Drawing the Visible Objects

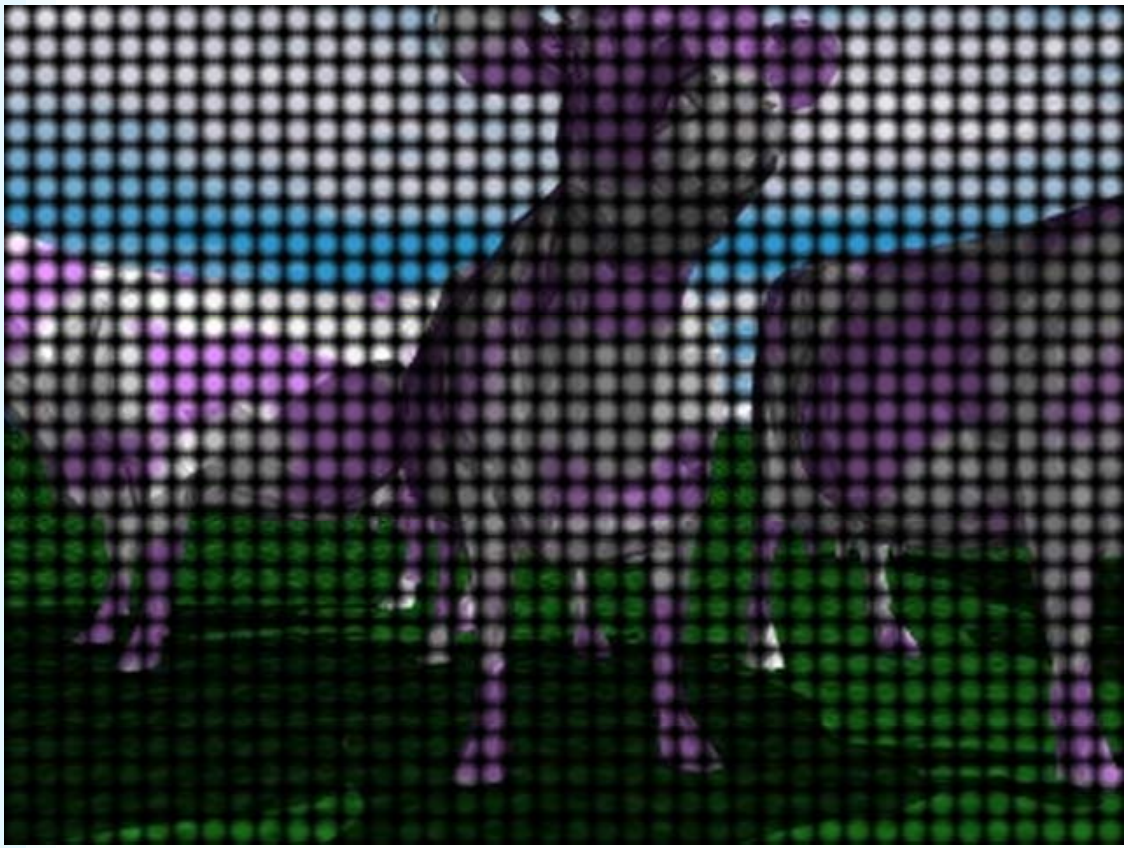
We want to generate the image that the eye would see, given the objects in our space.

How do we draw the correct object at each pixel, given that some objects may obscure c



Pixel-Level Visibility

Thus far, we've considered visibility at the level of primitives. Now we will turn our attention to a class of algorithms that consider visibility at each pixel.



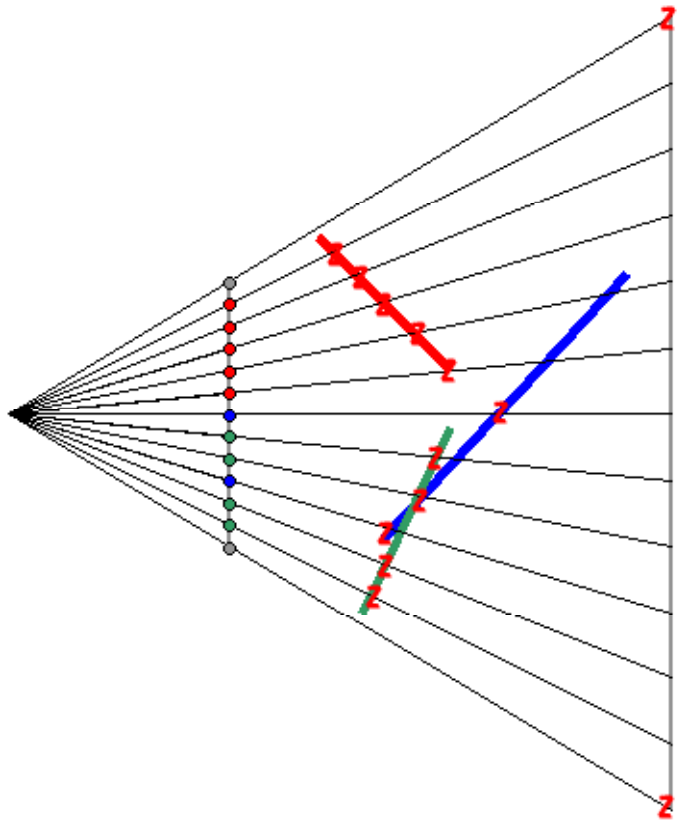
Depth Buffering

Algorithm:

Cast a ray from the viewpoint through each pixel to find the closest surface

Rendering Loop:

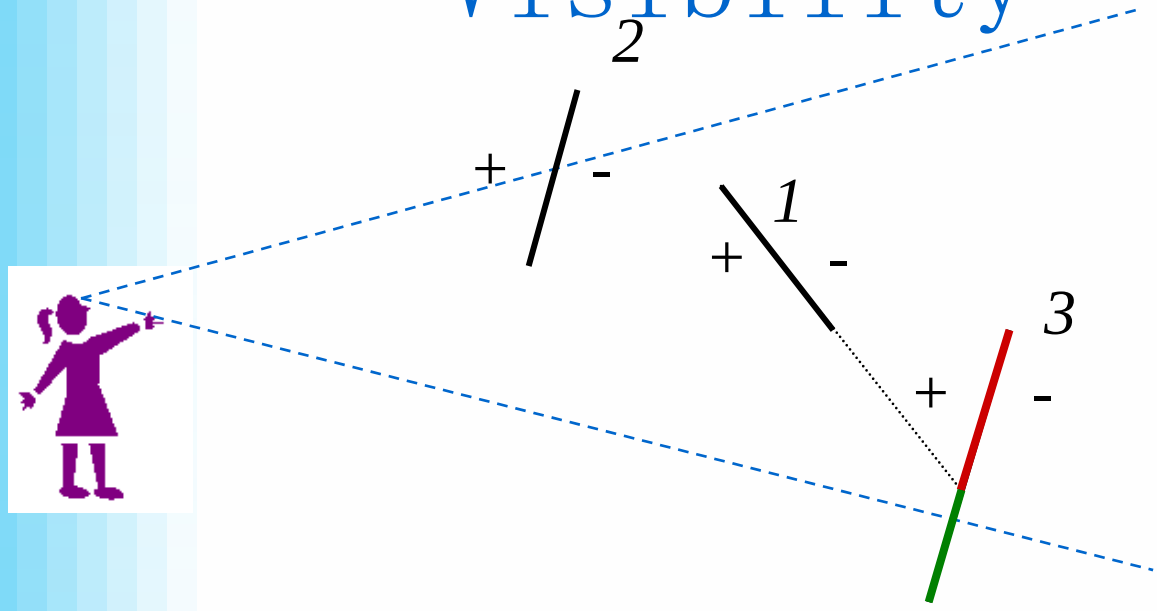
```
set depth of all pixels to Z_MAX
foreach primitive in scene
    determine pixels touched
    foreach pixel in primitive
        compute z at pixel
        if (z < depth) then
            pixel = object color
            depth = z
        endif
    endfor
endfor
```



Depth Buffering

- Advantages
  - Primitives can be processed immediately (Hence: Immediate mode graphics API's)
  - Primitives can be processed in any order (Exception: primitives at same depth)
  - Well suited to H/W implementation simple control of low-level (per pixel) operations
  - Spatial coherence
    - Incremental evaluation of loops
    - Good memory access pattern
- Disadvantages
  - Visibility determination is coupled to sampling (Subject to aliasing)
  - Requires a Raster-sized arrray to store depth
  - Read-Modify-Write (Hard to make fast)
  - Excessive over-drawing

Backface Culling for General Visibility



Well almost... it would work if there were no overlapping faces. However, notice how the overlapping facets partition each other. Suppose we build a tree of these partitions.

A Painter's Algorithm

•The painter's algorithm, sometimes called depth-sorting, gets its name from the process which an artist renders a scene using oil paints. First, the artist will paint the background colors of the sky and ground. Next, the most distant objects are painted, then the nearer objects, and so forth. Note that oil paints are basically opaque, thus each sequential layer completely obscures the layer that its covers.

A very similar technique can be used for rendering objects in a three-dimensional scene. First, the list of surfaces are sorted according to their distance from the viewpoint. The objects are then painted from back-to-front.

While this algorithm seems simple there are many subtleties. The first issue is which depth-value do you sort by? In general a primitive is not entirely at a single dep[illegible]e, we must choose some point on the primitive to sor[illegible] by.

Implementation

•The algorithm can be implemented very easily. First we extend the drawable interface so that any object that might be drawn is capable of supplying a z value for sorting.

```
import Raster;
public abstract interface Drawable {
   public abstract void Draw(Raster r);
   public abstract float zCentroid();
}
```

•Next, we add the required method to our triangle routine:

```
public final float zCentroid() {
   return (1f/3f) * (vlist[v[0]].z + vlist[v[1]].z + vlist[v[2]].z);
}
```

Rendering Code

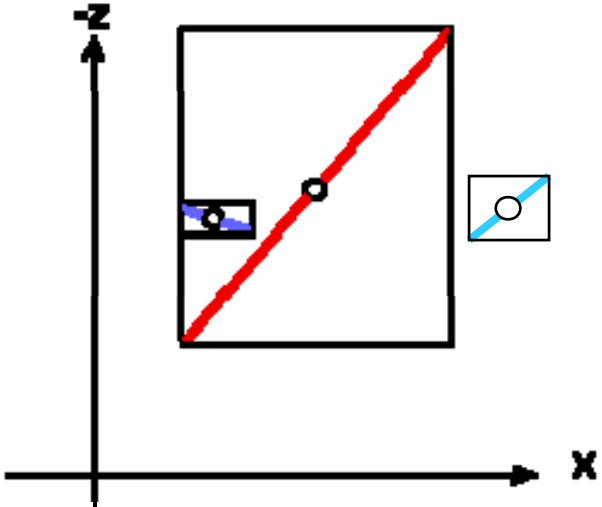
Here is a painter's method that we can add to any rendering applet:

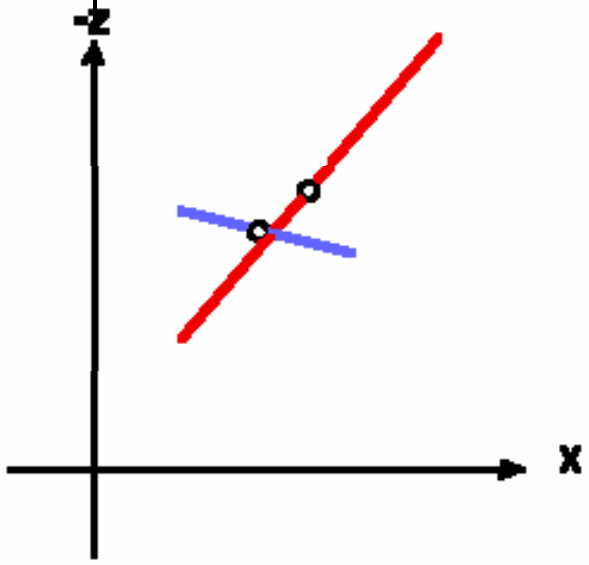
```
void DrawScene() {
   view.transform(vertList, tranList, vertices);
   ((FlatTri) riList[0]).setVertexList(tranList);
   raster.fill(getBackground());
   sort(0, triangles-1);
   for (int i = triangles-1; i >= 0; i--) {
      triList[i].Draw(raster);
   }
}
```

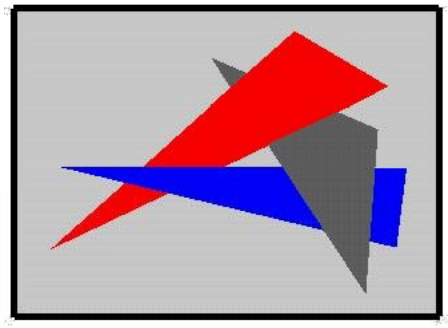
Problems with Painters

•The painter's algorithm works great... unless one of the following happens:

- Big triangles and little triangles. This problem can usually be resolved using further tests. Suggest some.
- Another problem occurs when the triangle from a model interpenetrate as shown below. This problem is a lot more difficult to handle. generally it requires that primitive be subdivided (which requires clipping).
- Cycles among primitives







BSP Trees

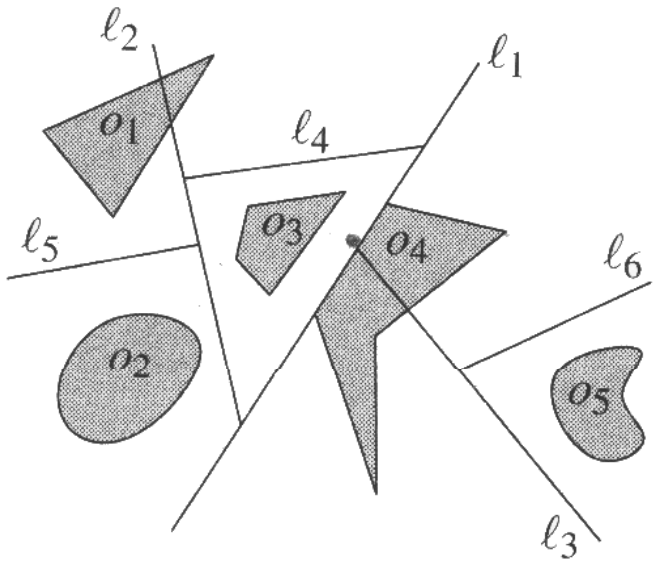
Having a pre-built BSP tree will allow us to get a correct depth order of polygons in our scene for any point in space.

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

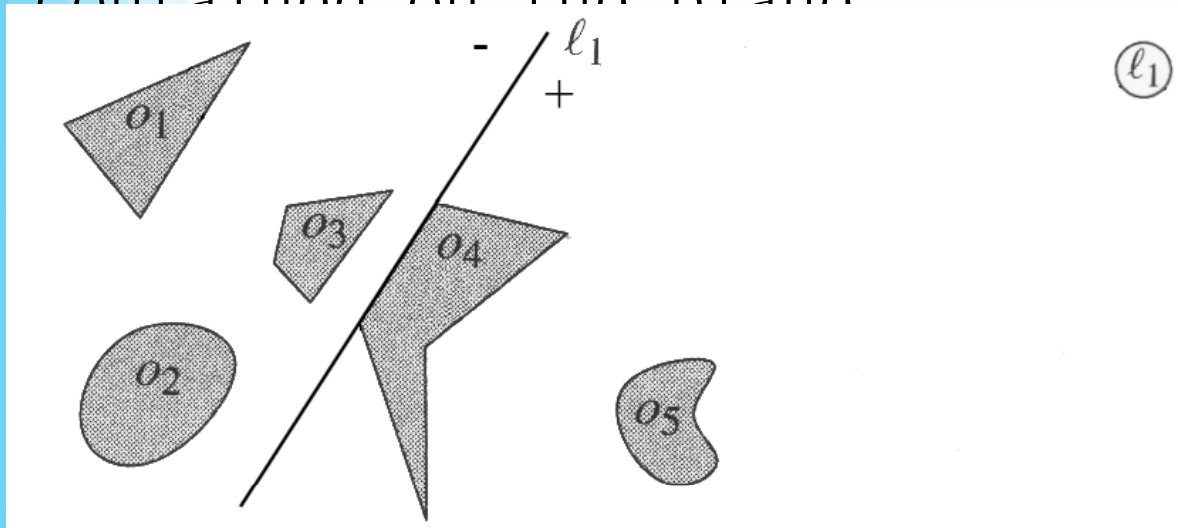
Assume that no objects in our space overlap.

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



Choose a Splitting Line

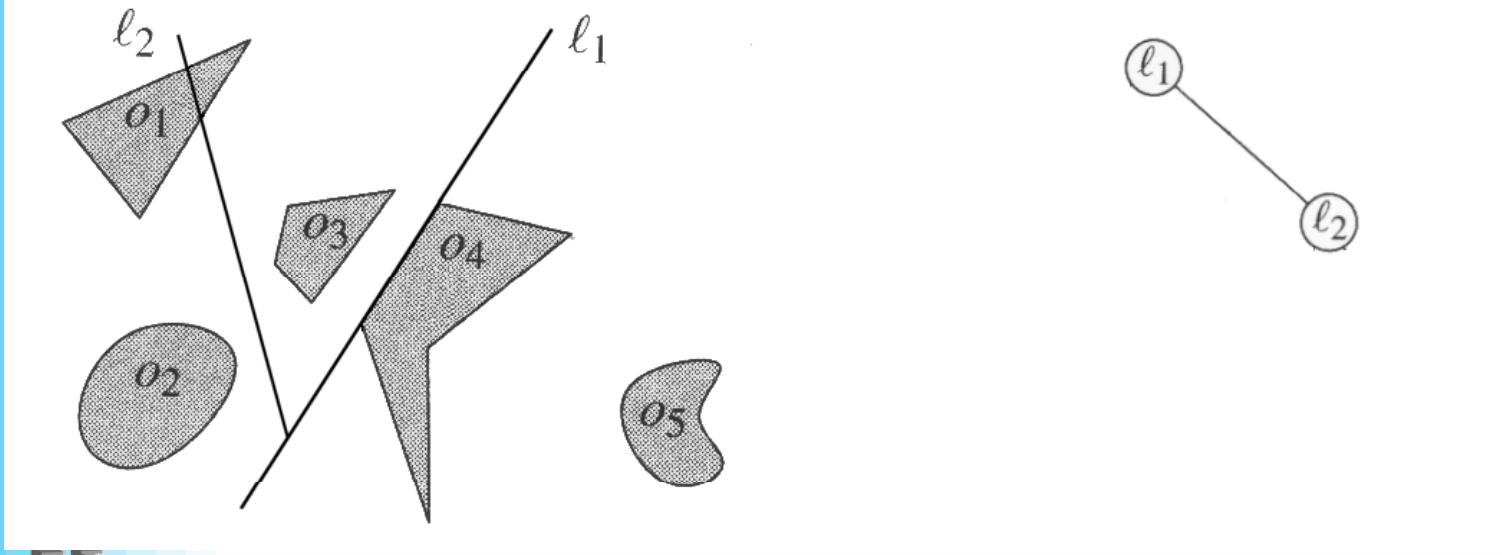
Choose a splitting plane, dividing our objects into three sets - those on each side of the plane, and those fully contained on the plane



Choose More Splitting Lines

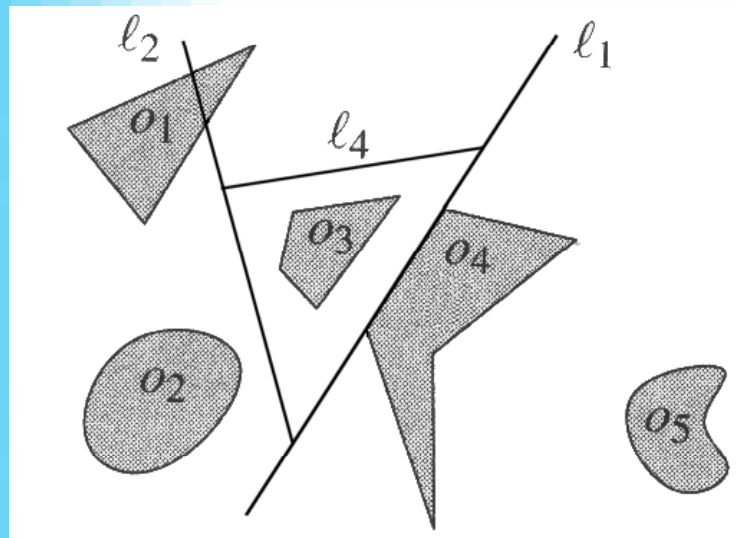
What do we do when an object (like object 1) is divided by a splitting plane?

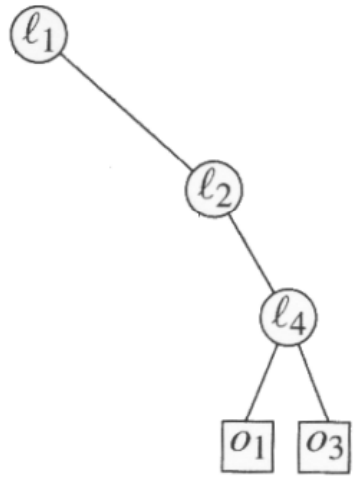
It is divided into two objects, one on each side of the plane.



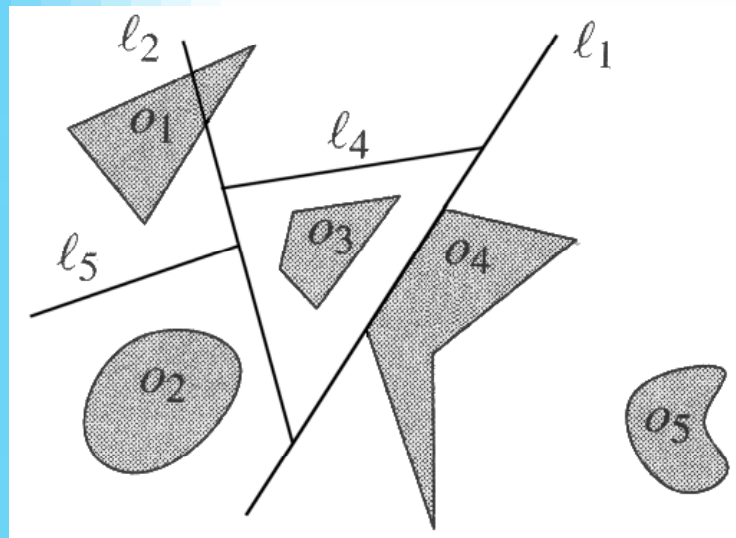
Split Recursively Until Done

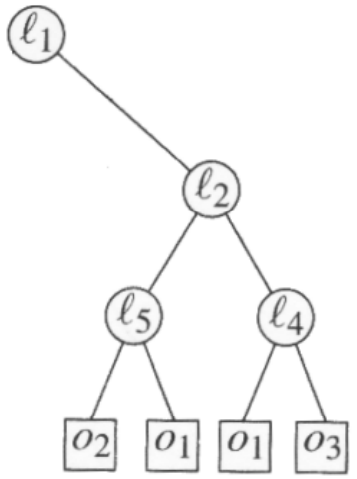
When we reach a convex space containing exactly zero or one objects, that is a leaf node.



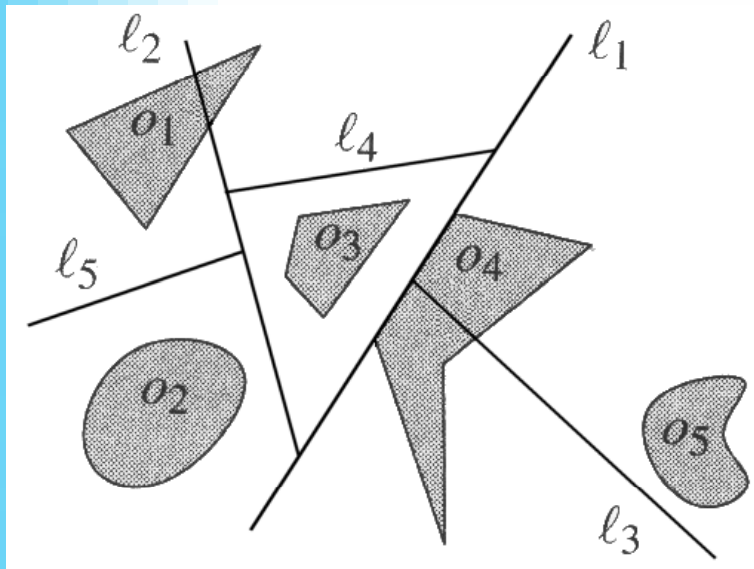


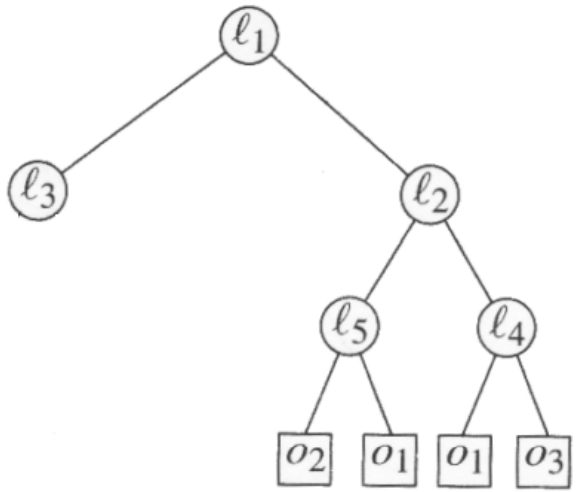
Continue


ℓ_2
ℓ_1
o_1
ℓ_4
o_3
o_4
ℓ_5
o_2
o_5


ℓ_1
ℓ_2
ℓ_5
ℓ_4
o_2
o_1
o_1
o_3

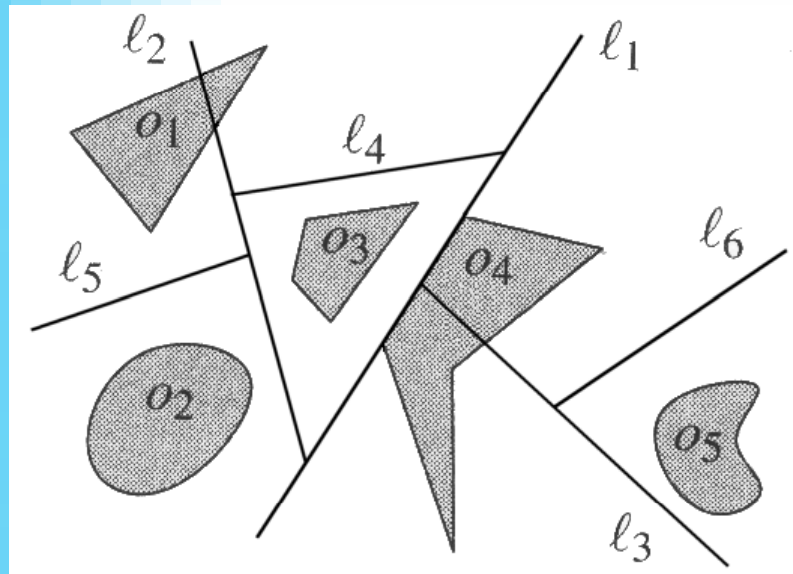
Continue

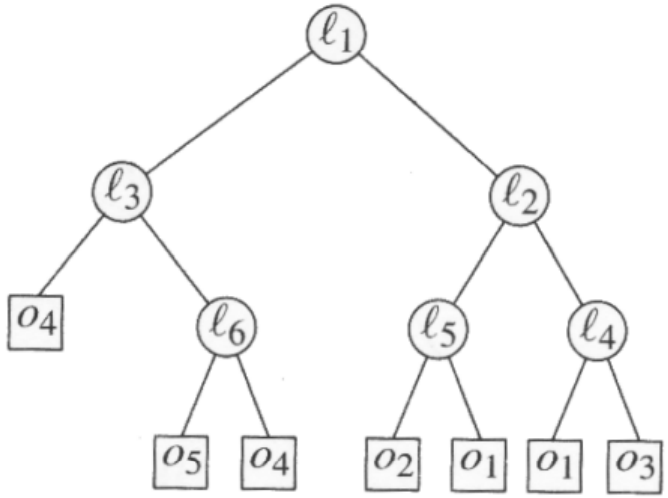

ℓ_2
ℓ_1
o_1
ℓ_4
o_3
o_4
ℓ_5
o_2
o_5
ℓ_3


ℓ_1
ℓ_3
ℓ_2
ℓ_5
ℓ_4
o_2
o_1
o_1
o_3

Finished

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



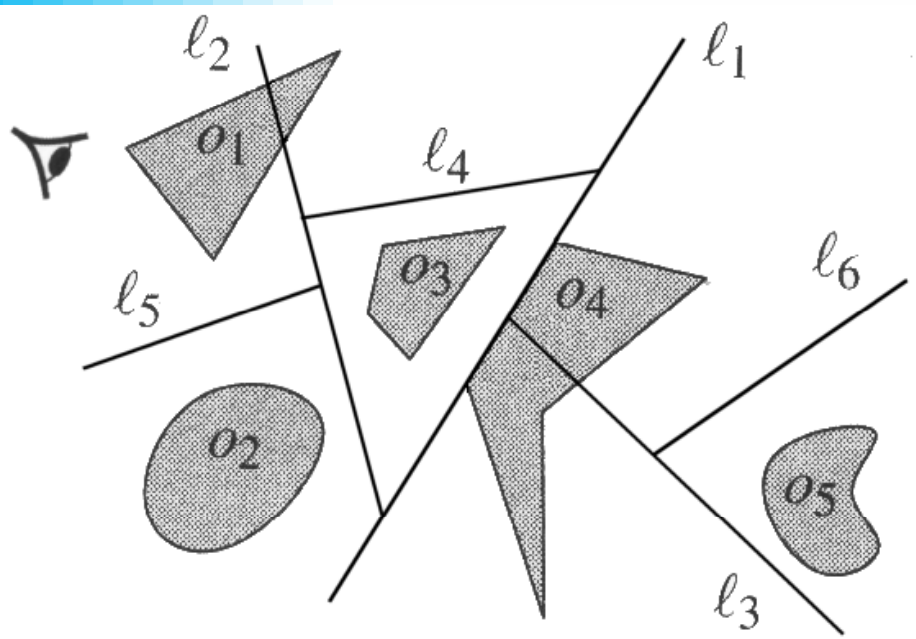


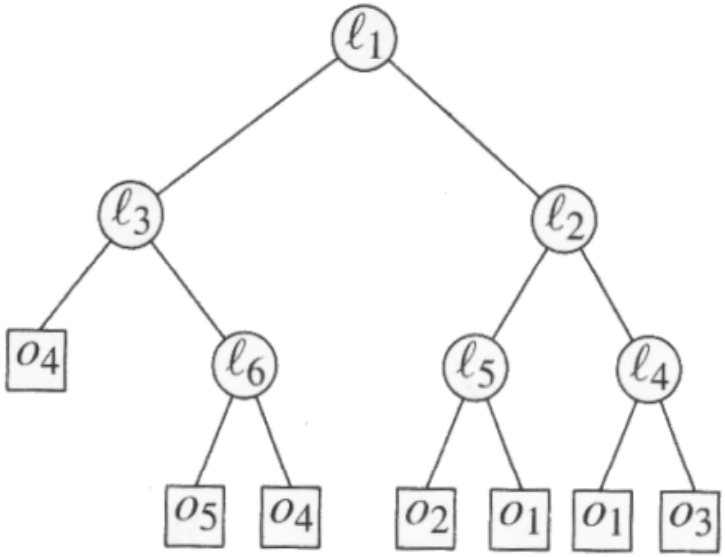
Querying the Tree

If a point is in the positive half-space of a plane, then everything in the negative half-space is farther away -- so draw it first, using this algorithm recursively.

Then draw objects on the splitting plane, and recurse into the positive half-space.

What Order Is Generated From This Eye Point?





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

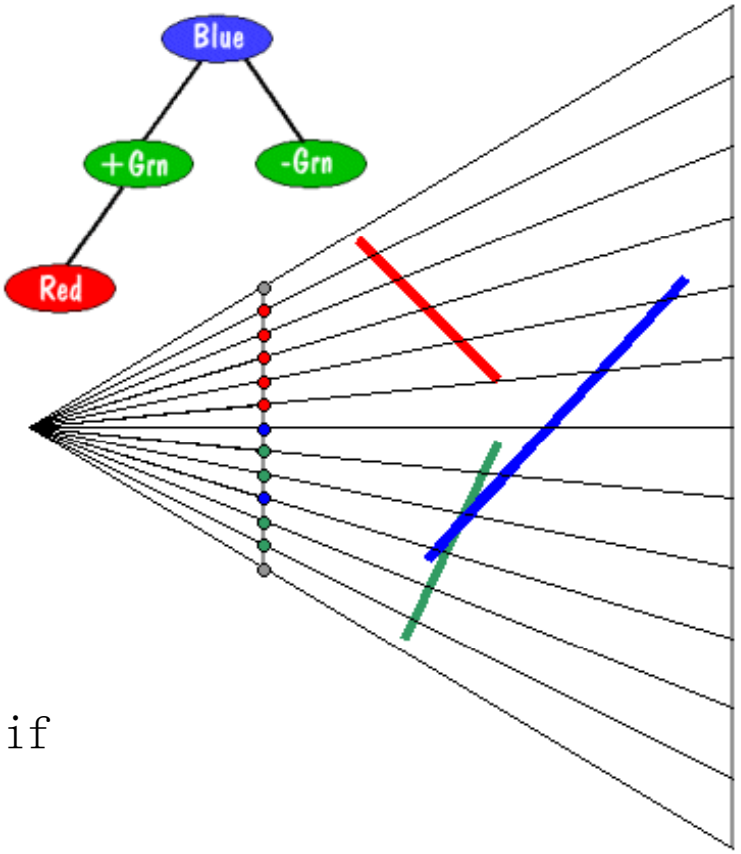
Structure of a BSP Tree

- 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.

Recap for line segments

A Binary Space Partition (BSP) tree is a simple spatial data structure:

1. Select a partitioning plane/face.
2. Partition the remaining planes/faces according to the side of the partitioning plane that they fall on (+ or -).
3. Repeat with each of the two new sets.

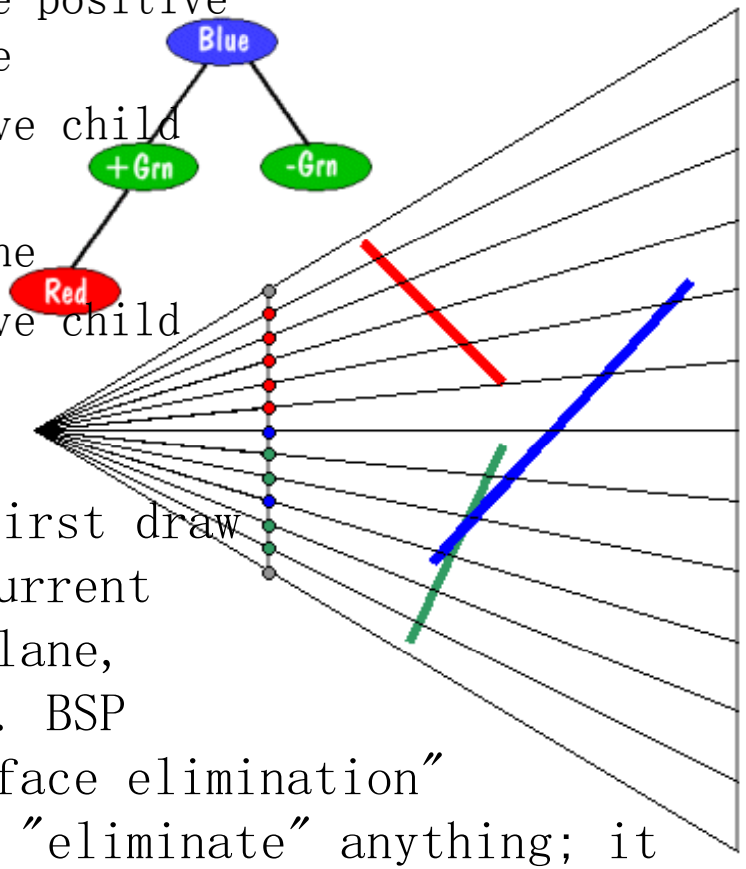


Partitioning facets:
Partitioning requires testing all facets in the active set to find if they 1) lie entirely on the positive side of the partition plane, 2) lie entirely on the negative side, or 3) if they cross it. In the 3rd case of a crossing face we clip the offending face into two halves (using our plane-at-a-time clipping algorithm).

Computing Visibility with BSP trees

Starting at the root of the tree.

1. Classify viewpoint as being in the positive or negative halfspace of our plane
2. Call this routine with the negative child (if it exists)
3. Draw the current partitioning plane
4. Call this routine with the positive child (if it exists)



Intuitively, at each partition, we first draw the stuff further away than the current plane, then we draw the current plane, and then we draw the closer stuff. BSP traversal is called a "hidden surface elimination" algorithm, but it doesn't really "eliminate" anything; it simply orders the drawing of primitive in a back-to-front order.

BSP Tree Example

Computing visibility or depth-sortir
with BSP trees is both simple and fa

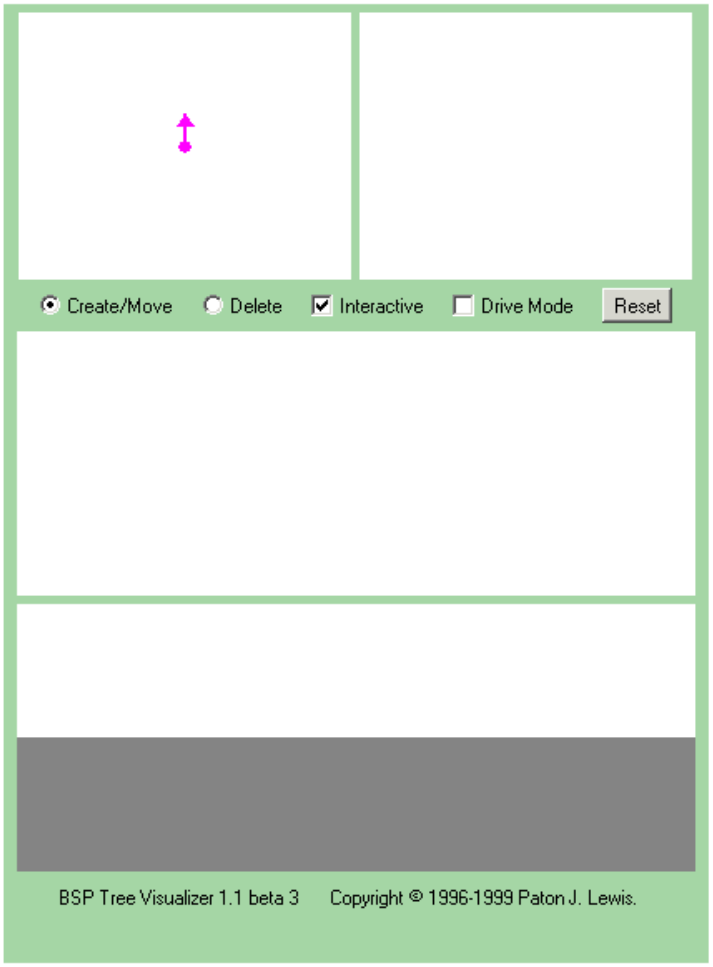
It resolves visibility at the primit

Visibility computation is independer
screen size

Requires considerable preprocessing
the scene primitives

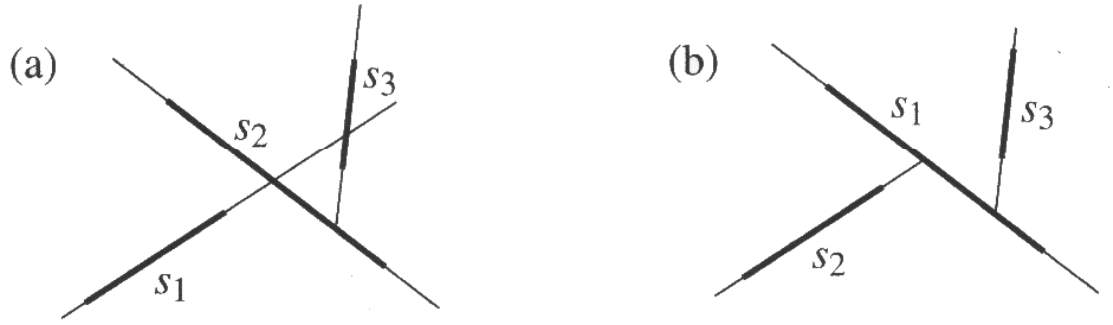
Primitives must be easy to subdivide
along planes

Supports Constructive Solid Geometry
(CSG) modeling



How Small Is the BSP From This Algorithm?

- Different orderings result in different trees



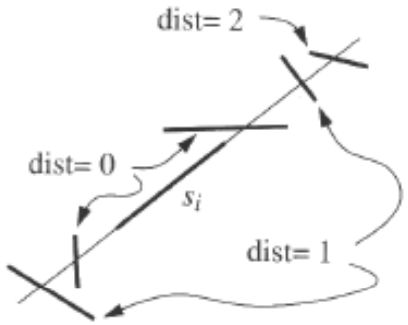
- Greedy approach doesn't always work -- sometimes it does very badly, and it is costly to find

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: O(n Log n)

Let $\text{Dist}_{si}(sj)$ = The # of segments intersecting line(si) between si and sj, if line(si) intersects sj, or +infinity otherwise



$$\Pr[\ell(s_i) \text{ cuts } s_j] \leqslant \frac{1}{\text{dist}_{s_i}(s_j)+2}$$

$$\begin{aligned} \text{E[number of cuts generated by } s_i] &\leqslant \sum_{j \neq i} \frac{1}{\text{dist}_{s_i}(s_j)+2} \\ &\leqslant 2\sum_{k=0}^{n-2} \frac{1}{k+2} \\ &\leqslant 2\ln n. \end{aligned}$$

Thus, the expected number of fragments = O(n + n log n).

Expected Running Time: $O(n^2 \log n)$

- The time taken at any particular node is linear in the number of fragments in its input
- Each call to the algorithm thus takes $O(n)$ time, and the number of recursive calls is bounded by $O(n \log n)$
- The total expected running time is then $O(n^2 \log n)$