Spatial Data Structures

References

Notes based on
Tomas Moller and Eric Haines, Real-Time Rendering, A K Peters, 2nd edition, 2002. Chapter 9.
Franco Preparata and Michael Shamos, Computational Geometry: An Introduction, Springer Verlag, 1991. Chapter 2.
Alan Watt and Mark Watt, Advanced Animation and Rendering Techniques, Addison WWesley, 1992. Chapter 9.

The Collision Detection Problem

The collision detection problem is to find collisions amongst a set of objects where one or more is moving.
One approach to the collision detection problem is to test for intersection detection amongst the objects — using intersection tests of the type considered previously — at discrete points in time at which the objects are regarded as fixed.
The interval of time between intersection tests must be small enough so that collisions between fast moving objects are not missed.

Brute Force Collision Detection: Pairwise Testing

The brute-force collision detection approach is to perform pairwise intersection tests of all object pairs at each

```
1:  for i = 1 to n − 1
2:    for j = i to n
3:      intersect(object(i), object(j))
```

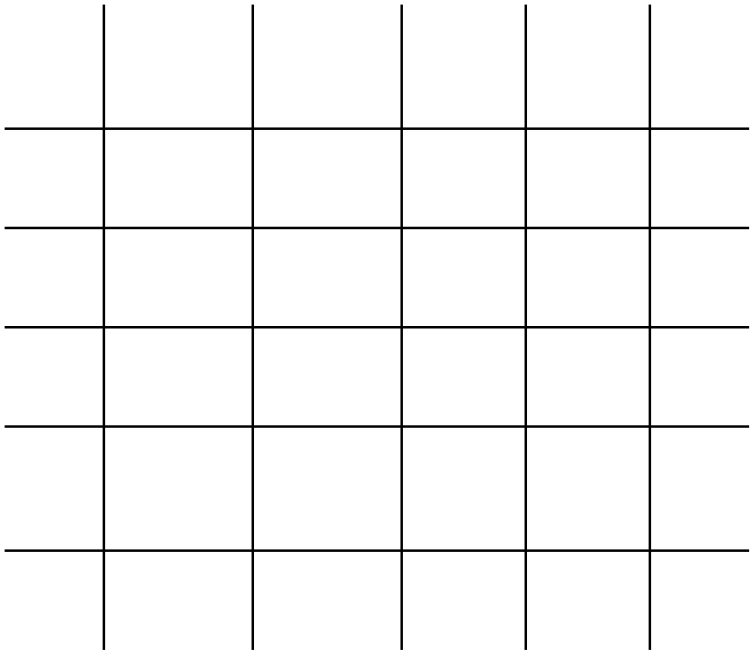
Figure: Brute-force collision detection

There are $\frac{n(n-1)}{2}$ pairs amonst n objects, and this is the number of object-object intersection tests the brute-force algorithm performs — regardless of the number of intersections.
Using the big-oh notation, the brute-force algorithm is $O(n^2)$ time complexity.

Intersections Amongst a Set of Line Segments

One approach to improve intersection detection performance is to use better algorithms.
Consider the problem of finding intersections amongst line-segments in 1D. In 1D there can be between 0 and $O(n)$ intersections amongst n line segments (intervals). The problem finding these intersections can be solved in $O(nlgn)$ time.
In 2D however, there can be between 0 and $O(n^2)$ intersections amongst n line segments. Consider the worst case below:

Intersections Amongst a Set of Line Segments (cont.)



Any algorithm which solves the problem *must* take $O(n^2)$ time in the worst case.

Intersections Amongst a Set of Line Segments (cont.)

Problems in which the size of the answer has a large range — between 0 and $O(n^2)$ for the line intersection problem in 2D — the computational complexity is often expressed as a sum of two parts:

$$O(f(n) + g(n))$$

where $f(n)$ is the answer-independent amount of work which must be done and $g(n)$ is the answer-dependent amount of work.
For the line intersection problem above a trivial lower bound is $O(n + k)$ where n is the number of line segments and k is the number of intersections (answers) and is $O(n^2)$.
A *plane sweep* based approach to the line intersection problem yields an algorithm with complexity

$$O(nlgn + k)$$

Intersections Amongst a Set of Line Segments (cont.)

Many intersection problems are much harder in higher dimensions. Some scale exponentially with dimension, although in computer graphics we tend to stop at 3D.

Spatial Data Structures

Another general approach to improve intersection detection, and thereby collision detection, performance is to use *spatial data structures*.

In spatial or geometric data structures objects are organised based on space or geometry. In intersection testing they guide the application of intersection tests to pairs of objects which are "close" — spatial divide-and-conquer.

Spatial data structures have many more uses than intersection detection and colllision detection. They allow problems (queries) about spatial relationships of objects to be solved.

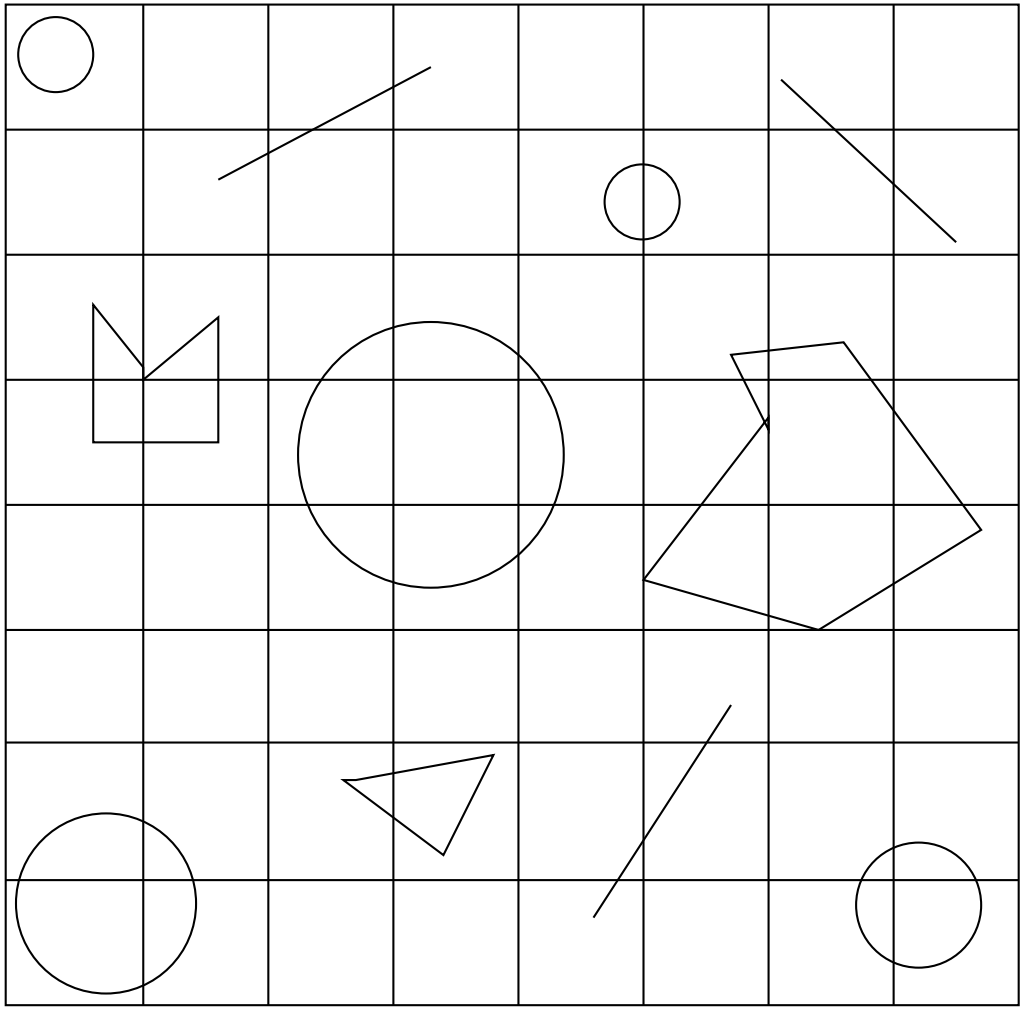
Spatial data structures may or may not be *space filling*. If they are not then they contain *voids*. A space-filling spatial data structure can be regarded as a result of *spatial subdivision*.

Spatial Data Structures (cont.)

Unlike 1D, in 2D and 3D worst-case optimal data structures and algorithms are often not available, and average case performance — often measured using benchmark test suites — are used.
A combination of spatial data-structures and approaches may be used to solve spatial problems.
For example, a real-time rendering situation typicaly has a mixture of moving and stationary (static) objects. A data structure for the static objects can be built once in a preprocessing phase.

Uniform Grid

The simplest spatial data structure is *the uniform grid*.



Uniform Grid (cont.)

Grids are space-filling.
Each cell — or *voxel (volume pixel)* — has a list of objects which intersects it.
The uniform grid is used to determine which objects are near to an object by examining object-lists of the cells the object overlaps.
Intersections for a given object are found by going through the object lists for all voxels containing the object, performing intersection tests against objects on those lists.

A grid based collision detection algorithm then works as follows.

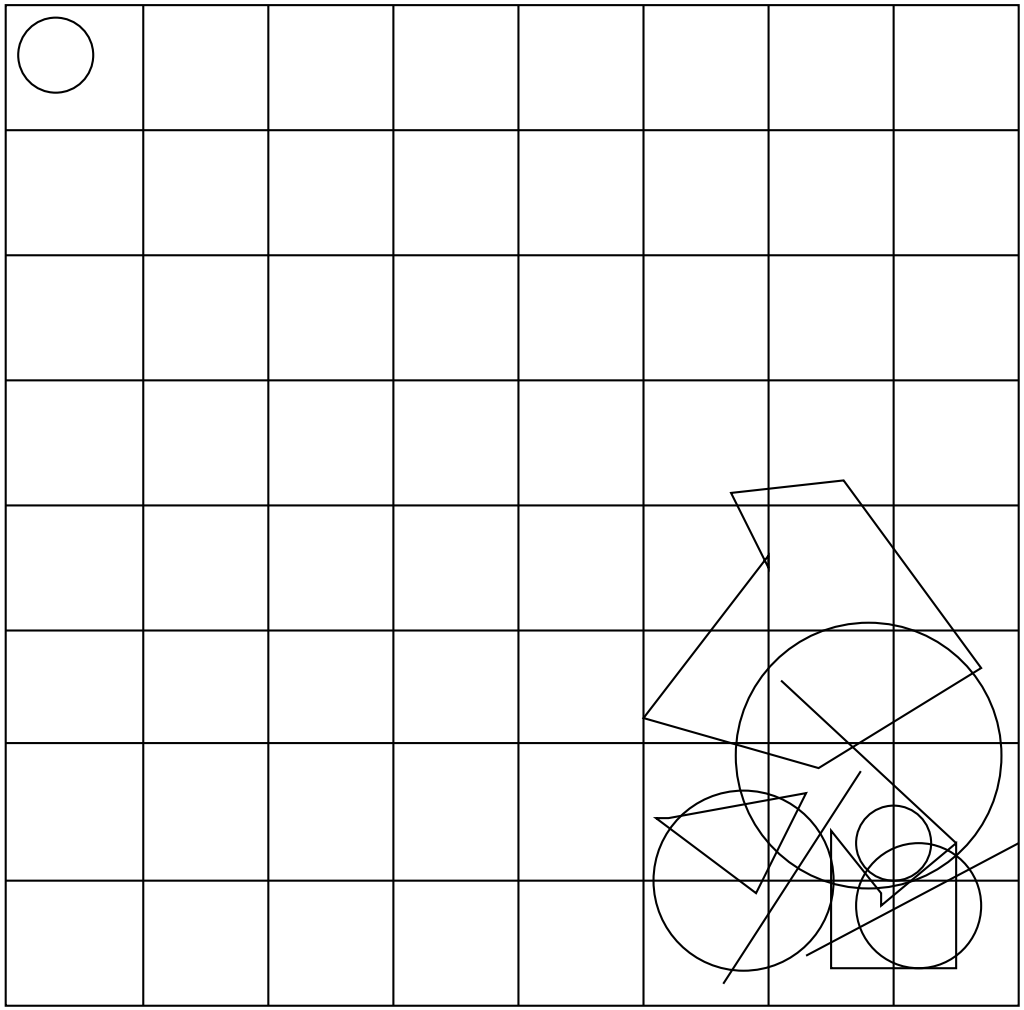
```
1:  for i = 1 to n
2:    v_min = voxel(min(bbox(object(i))))
3:    v_max = voxel(max(bbox(object(i))))
4:    for x = v_min_x to x = v_max_x
5:      for y = v_min_y to x = v_max_y
6:        for z = v_min_z to x = v_max_z
7:          for j = 1 to n_objects(voxel(x,y,z))
8:            if (not tested(object(i), object(j)))
9:              intersect(object(i), object(j))
```

Figure: Grid-based collision detection

Uniform Grid Performance

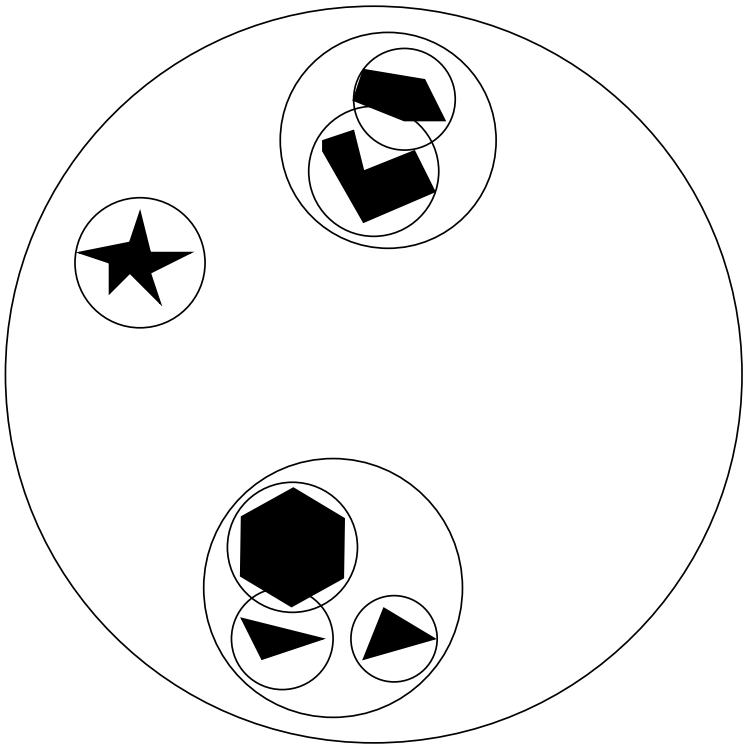
The uniform grid has excellent (in fact, optimal) average case performance if objects are reasonably uniformly distributed.
As soon as the objects become (highly) non-uniformly distributed, or *clustered*, the uniform grid approach degenerates into pairwise testing, with $O(n^2)$ performance.

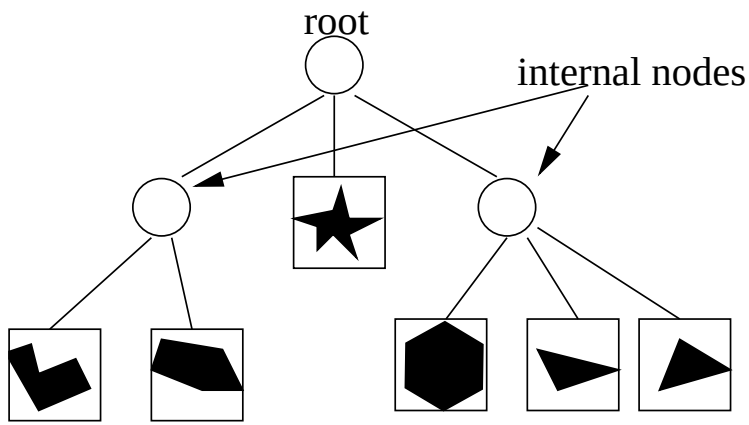
Uniform Grid Performance (cont.)



Bounding Volume Hierarchies

In a bounding volume hierarchy (BVH), bounding volumes (totally) contain other bounding volumes and these are organised into a tree.





Bounding Volume Hierarchies (cont.)

Any of the bounding volumes previously discussed can be used to create a BVH.
A BVH is not space filling.
Bounding volume hierarchies may reduce collision detection from an $O(n^2)$ algorithm to $O(nlg(n))$ or even $O(n)$.

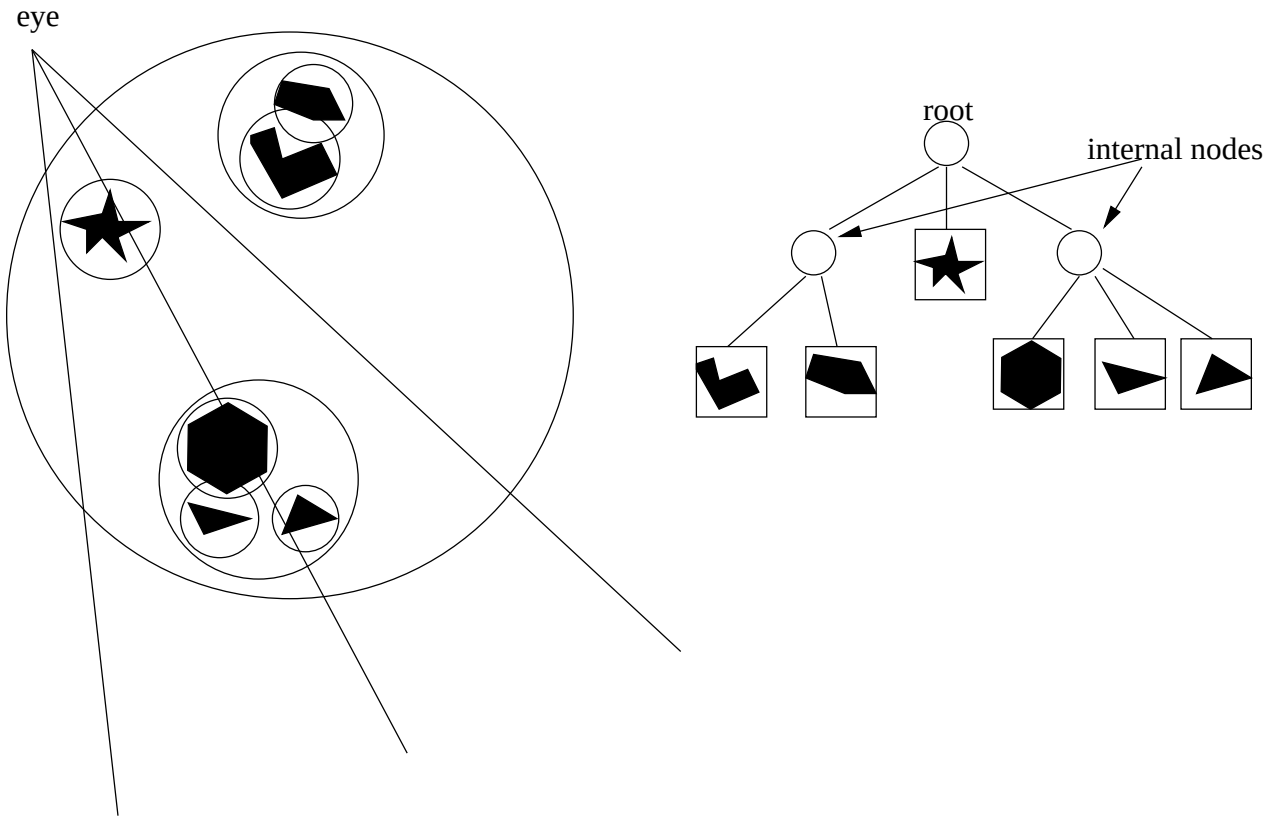
Collision detection between two polygonal objects each with their own bounding volume hierarchy is peformed recursively as follows:

```
     FindFirstHitCD(A, B)
     returns (TRUE,FALSE);
1:   if (not overlap(A_BV, B_BV) return FALSE;
2:   else if (isLeaf(A))
3:     if (isLeaf(B))
4:       for each triangle pair T_A ∈ A_c and T_B ∈ B_c
5:         if (overlap(T_A, T_B) return TRUE;
6:     else
7:       for each child C_B ∈ B_c
8:         FindFirstHitCD(A, C_B)
9:   else
10:    for each child C_A ∈ A_c
11:      FindFirstHitCD(C_A, B)
12: return FALSE;
```

Figure: BVH-BVH collision detection

BVH Based View Volume Culling

BVHs are often used for view-volume culling:



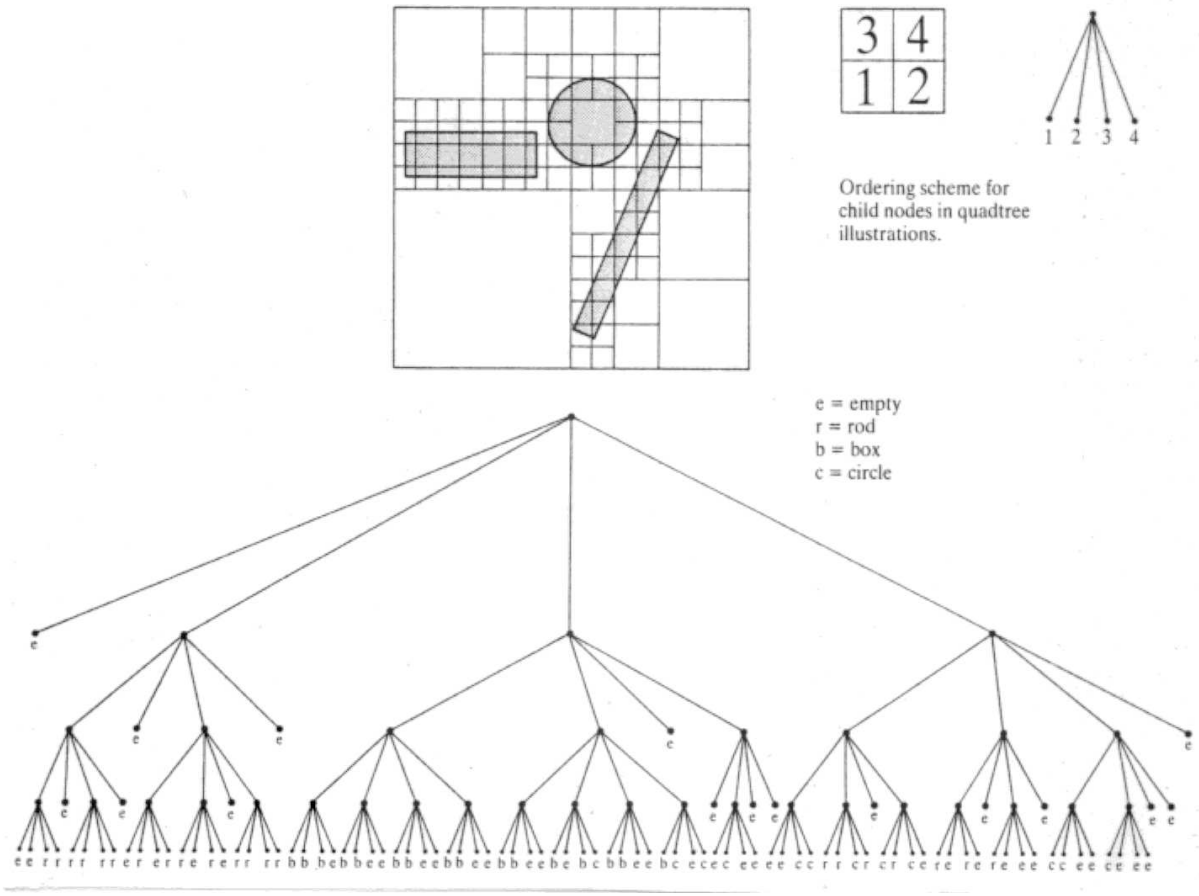
BVH Based View Volume Culling (cont.)

View-volume culling is an important real-time rendering acceleration or speed-up technique. *Scene-graph* graphics libraries provide view-volume culling automatically.

Quad-Trees and Oct-Trees

An oct-tree is a non-uniform subdivision of space where a axis-aligned box region is split into eight octants by three axis-aligned dividing planes.
A quad-tree is a non-uniform subdivision of area where a axis-aligned box region is split into four quadrants by two axis-aligned dividing lines.
In an oct-tree each node has eight children. In a quad-tree each node has four children.

The following diagram shows a 2D quad-tree.



The advantages of an oct-tree over a uniform grid are:

1. it handles clustering of objects reasonably
2. space requirements may be reduced

However some disadvantages are:

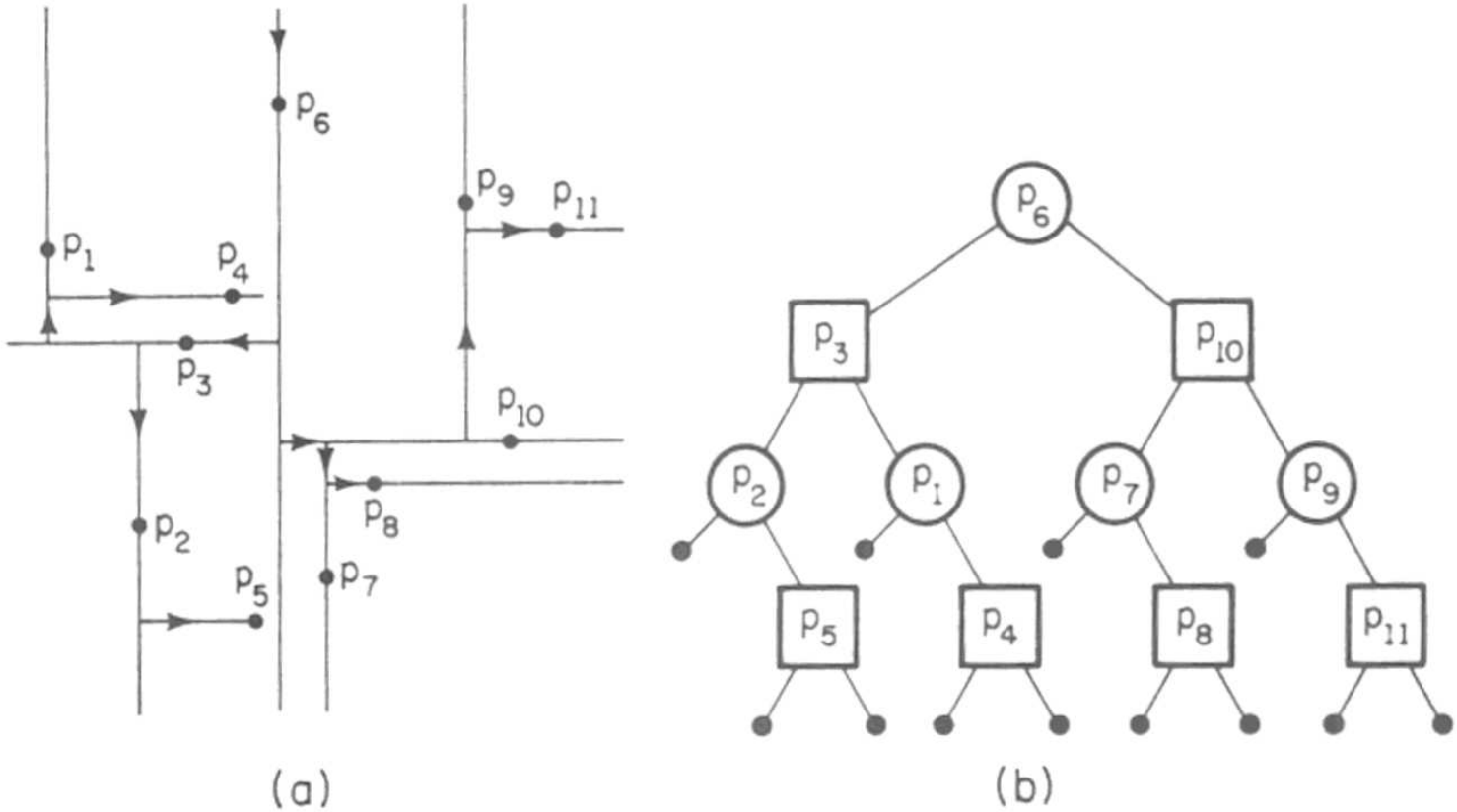
1. more expensive traversal costs
2. large volumes of space may be occupied by only small objects

A trade-off inherent in oct-trees is limiting tree depth, thereby increasing leaf node voxel size, but increasing the the number of objects per voxel.

k-d Trees

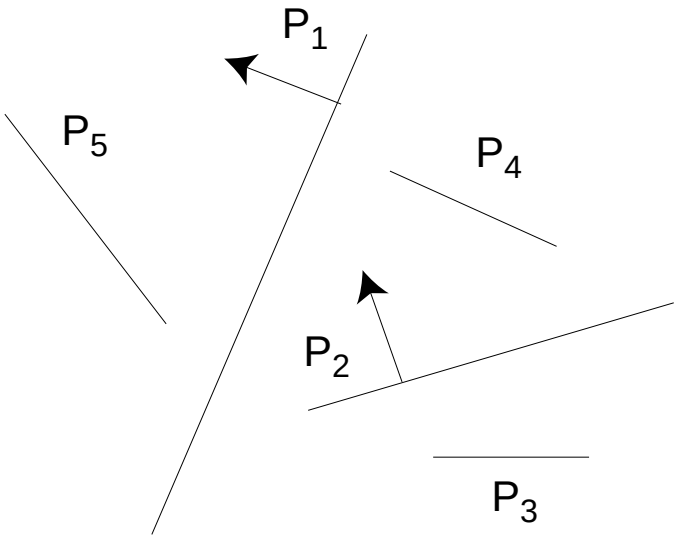
The k-d tree is a spatial subdivision approach relying on *binary* subdivision. D-dimensional *hyperspace* is cyclically sub-divided along each of the d dimensions.
Thus each node has *two* children — regardless of the dimensionality of the data (c.f. *four* for a quad-tree and *eight* for an oct-tree).
In a $2d$ tree the plane is first divided into two at some value of x.
The next subdivision in each of the children is at some value of y.
And then back to x, and so on.

k-d Trees (cont.)

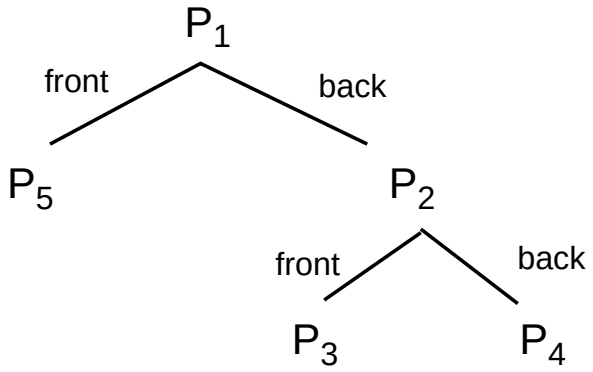


Binary Space Partition (BSP) Trees

BSP trees are similar to k-d trees. Space is subdivided by splitting it in two at each level of the tree.



Sample set of polygons



BSP tree

Binary Space Partition (BSP) Trees (cont.)

There are two forms of BSP tree — distinguished by whether the splitting object is axis-aligned or object-aligned. Axis aligned BSP trees are in fact (basically) the same as k-d trees.

Binary Space Partition (BSP) Trees (cont.)

To build a BSP tree a spliting plane is chosen and then used to divide all objects into two lists, one in front of the plane and the other list behind the plane. The same algorithm is then recursively applied on each sub list until each list only contains a single polygon. If any polygon (object) crosses the splitting plane it is cut into into two each part added to the appropriate sublists.
BSP trees, like all trees, work best when balanced. However, balancing BSP trees, and quad-trees, oct-trees and $k-$d trees, tends to be difficult as rotations are difficult.
One method suitable for (largely) static data is to choose a limited number (say 6) of test splitting planes, compare them based on the number of polygons the place each side of the split, and use the best. Thus the balancing algorithm is *top-down* rather than *bottom-up* — and no rotations take place.

Creating a BSP Tree

The following diagrams illustrate the process of creating a BSP tree.