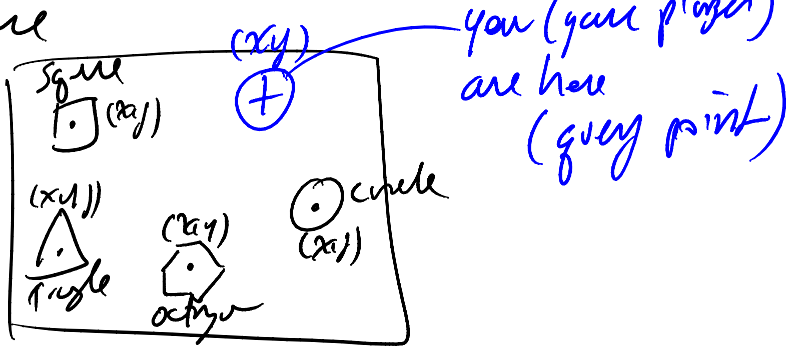
Tuesday, October 27, 2009
3:32 PM

Kd-trees (§12.6 in text)

- these are data structures for organizing spatial data
  e.g. k-dimensional data such as points in 2-space or 3-space
- in computer graphics they are a spatial subdivision method (ADT)

e.g., a 2D game



- each object has a center
- all objects are static (they don't move)
- the kd-tree is primarily used to answer the query: which is the closest object to ⊕

↪ nearest_neighbor(query-point)

k_nearest_neighbors(k, query-point)

↓

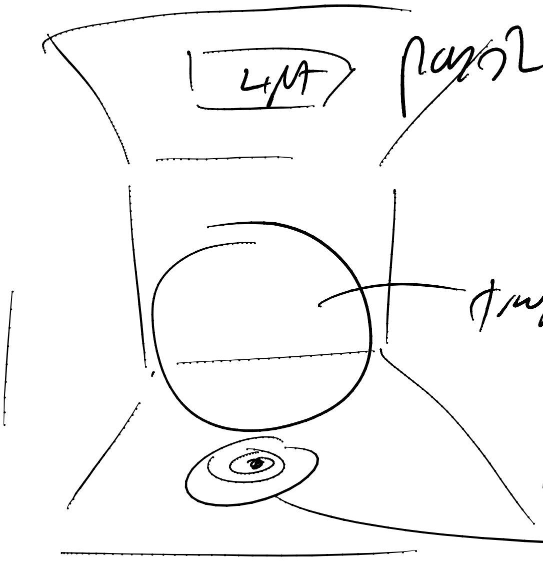
particularly useful in photon mapping

like a 2-pass ray tracer

1: direct photons from lights

pass 1: shoot photons from lights
(they "stick" to surfaces)

pass 2: at each ray / surface intersection, find k closest photons



- other spatial subdivision alg: BSP tree (binary space partition)

- main utility: nn queries.
- naive approach:

```
my location is $(x, y)$
$r_{min} = \infty$
for each object in scene {
    // compute distance
    $r = \sqrt{(x - x_i)^2 + (y - y_i)^2}$
    if ($r < r_{min}$) {
        $r_{min} = r$
```

```
            rmin = r
            closest_object = i
      }
}
```

- what's running time? $O(n)$ for n objects

- we want $O(\lg n)$, e.g. if $n = 1,000,000$

  $\log_{10} n = 6$

  ↓

  suggests a tree, but how to pick a key for each of (x, y) "pairs"?

- ans: pick x at first level, then alternate

Tuesday, October 27, 2009
3:53 PM

- so at $\boxed{\text{level 1}}$, use x as the comparison key to determine whether we search (traverse) down the
  left $(x < x_i)$ or
  right $(x \geq x_i)$

- at $\boxed{\text{level 2}}$ use y

  if $(y < y_i)$ go left
  else go right

- at level 3, flip back to x
- for 3 dimensions, alternate / cycle with x, y, z, x, y, z, x ..
- for 4 " $\underbrace{x, y, z, w, x, y, z, w, x \ldots}_{\text{depth \% dim}}$

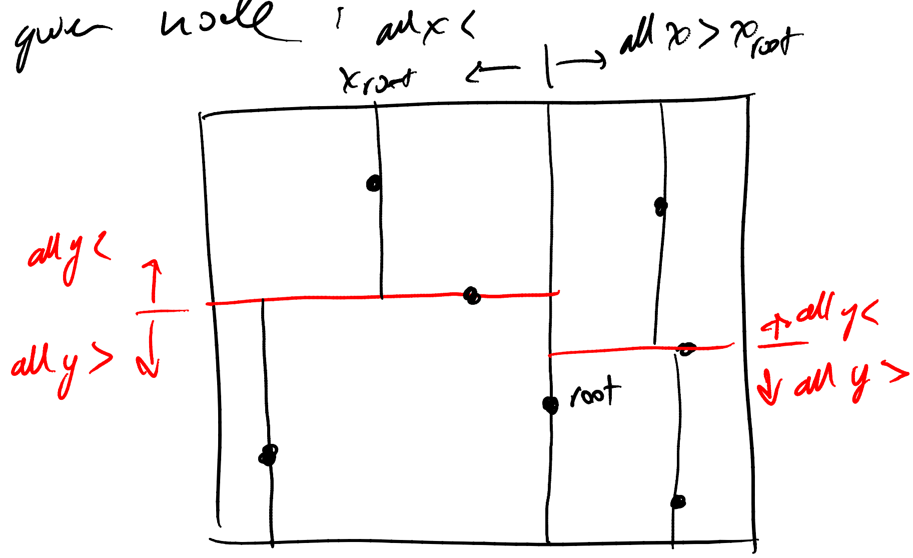
  ↑
  mod.

- each time (in the recursion) increment the key by 1 and mod by dim, call key <u>axis</u>

  $\text{axis} = (\text{axis} + 1) \,\%\, \text{dim}$

- why call key axis? Because at each level of the tree, space is split along the axis

U of given node :



- level 0 : use x as axis (key),
  find median of x-coords
  (7/2 = 3 [int div], so find 3rd largest point in x-sorted sequence)

- level 1 : use y as axis (key),
  find median of y-coords
  left : 3/2 = 1
  right : 3/2 = 1

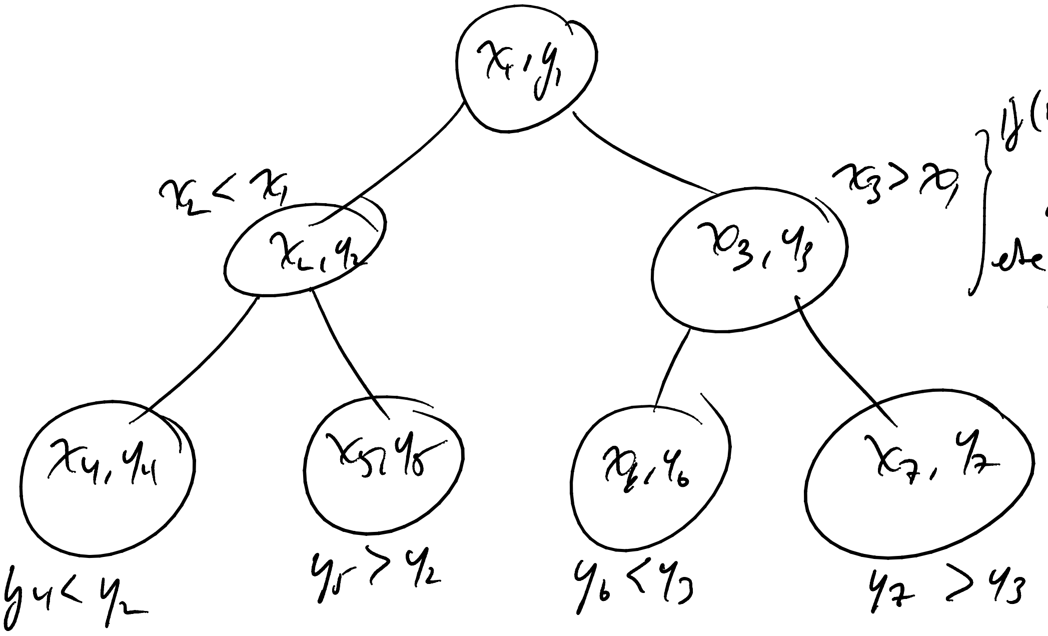
- level 2 : use x ...

- keeping one of the coords constant at each level, splits the plane (or 2D) by a line, an axis

---

Tuesday, October 27, 2009
4:09 PM

- you can think of the tree this way:



```
if (node.x < root.x)
    add(left)
else
    add(right)
```

```
if (node.y < root.y)
    add(left)
else
    add(right)
```

- key comparison changes at each level
- otherwise, basically same as binary search tree
- in some sense, very easy because:
  - no balancing
  - because meant for static scenes, & tree needs to be rebuilt entirely if scene changes, no need for deletion

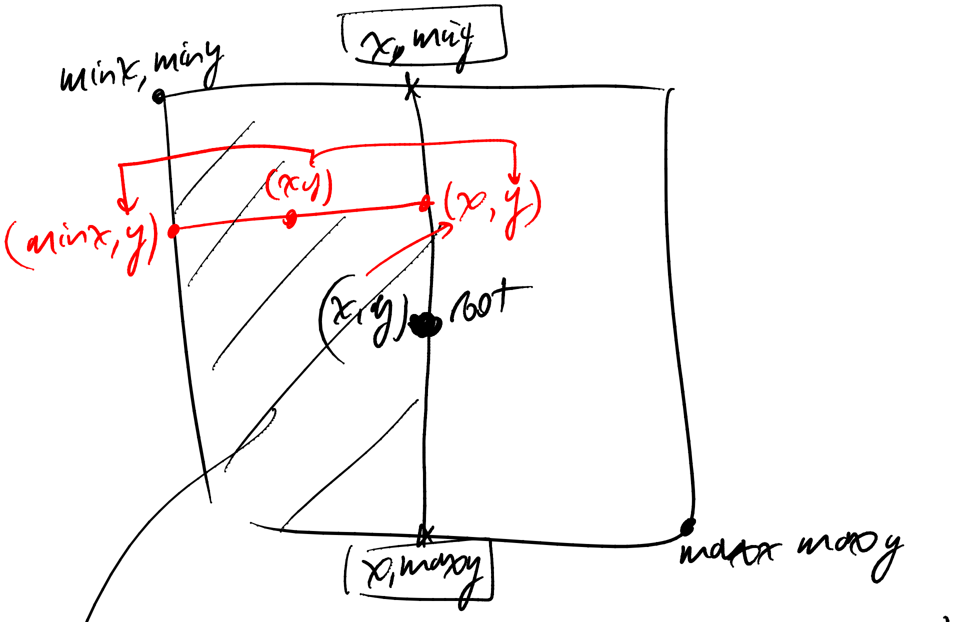
Tuesday, October 27, 2009
4:13 PM

- basic info needed at nodes:

```
struct KdNode {
    int axis // which axis this node split on
    T data, min, max;
    KdNode *left
    KdNode *right
}
```

for range queries

(can also add KdNode *parent)



left subtree's bounding box: (minx, miny), (x, maxy)

- Note this is different from what's in lab —
lab's kd-tree stores data only in leaves,
here, we store data at tree node

Tuesday, October 27, 2009
4:23 PM

- because the kd-tree is meant for static "scenes", what we'll do is read a bunch of points from file & send the whole list to the tree so it can construct itself.

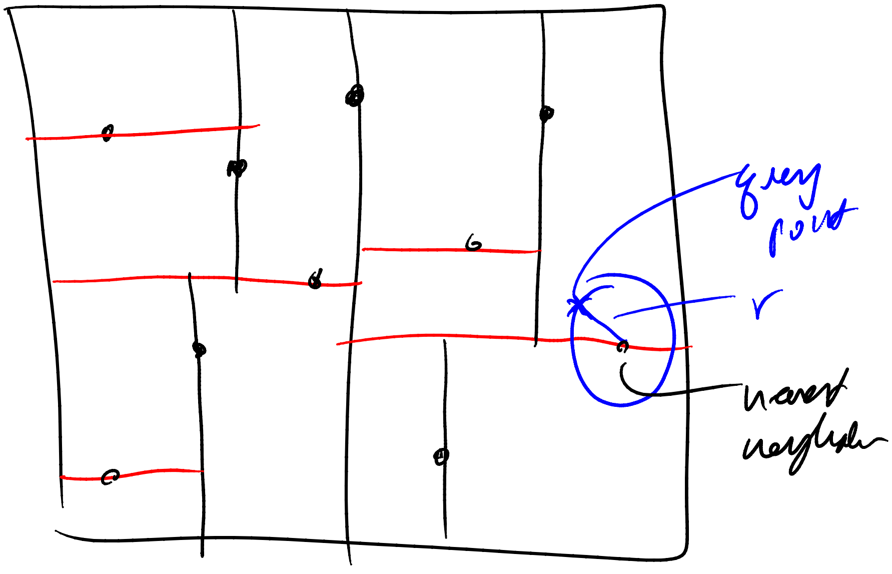
- diff from insert() we've done before where we added element by element

- e.g.;

```
Point min(-640, -480), max(640, 480)
Point *ptp;
std::vector<Point> pts;
KdTree<Point>     kdtree;

while(std::cin >> (ptp= new Point(0.0,0.0)))
      pts.push_back(*ptp);
kdtree.insert(pts, min, max)
```

- you build the kd-tree, then execute queries;



query point
r
nearest neighbor