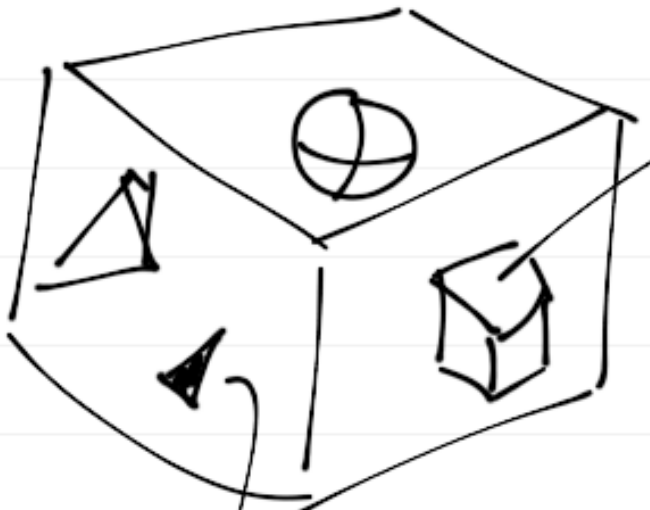
kd-tree : §12.6 in text

- data structure for organizing spatial data
  (e.g. our photons, which are 3D points in space)
- k: the dimensionality of the data
  (k = 2 or 3 for example)
- in graphics, kd-tree is a spatial subdivision data struct.
  (others include binary space partition or BSP)

eg. in a game:



objects in 3D world each with (x, y, z) ctr.

game player (in 3D world)

query:

which object is closest to me? (nearest neighbor query)

— NOTE: ALL OBJECTS STATIC

(if objects move, need to rebuild kd-tree)

- nearest neighbor naive approach:

my location (x, y) (in 2D)

$r_{min} = \infty$

for each obj in scene {

// compute distance

$$r = \sqrt{(x - x_i)^2 + (y - y_i)^2}$$

if $(r < r_{min})$ {

$r_{min} = r$

closest obj = i

}

}

- running time: $O(n)$ for n objects
- given $n = 25{,}000$
  want $O(\lg n)$
  (if $n = 1{,}000{,}000$, $\log_{10} n = 6$)
- we want a tree
- how to pick a key?
  for each of (x, y) "pairs" (of photons)

  ANS: alternate

- at each level,

  level 0 use x
  level 1 use y
  level 2 use x
  level 3 use y
  ⋮

photon.dim() { return 3 }

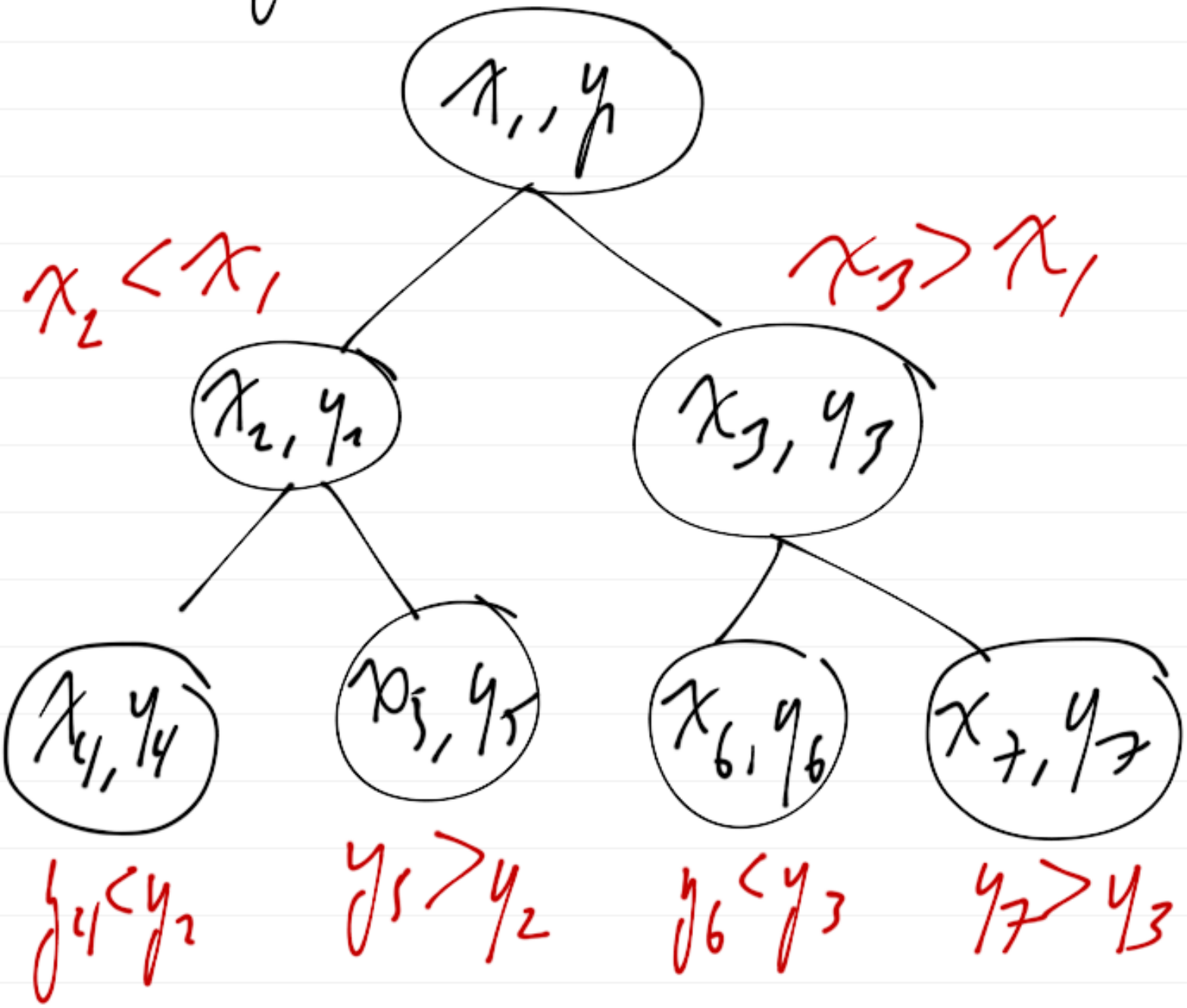
- for 3D data cycle thru x, y, z, x, y, z, ...
- at each tree level, use int axis;

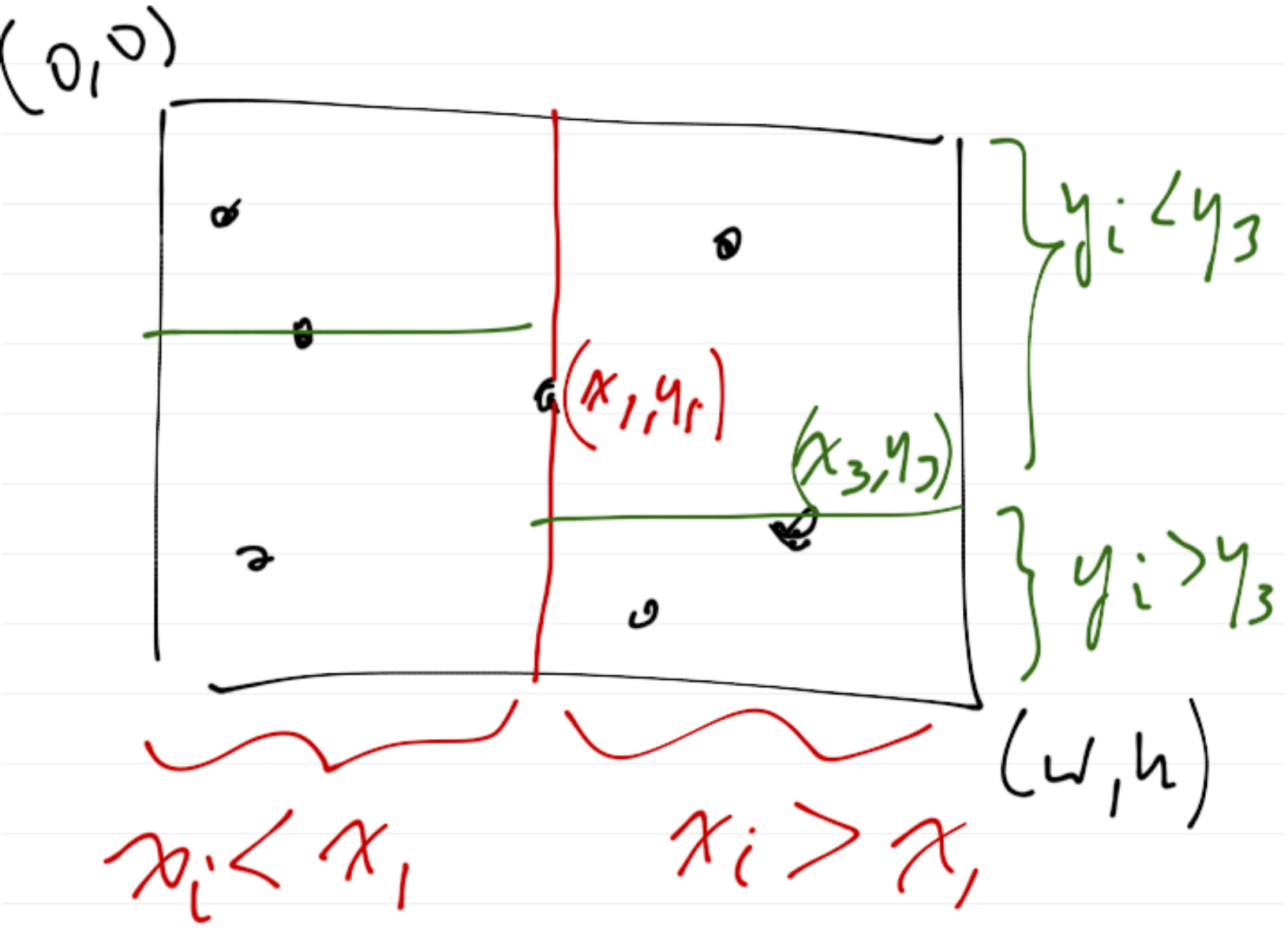
  mod

  axis = (axis + 1) % dim

returned by data stored on tree

- think of tree like this:




(0,0)
(x_1, y_1)
(x_3, y_3)
$y_i < y_3$
$y_i > y_3$
(w,h)
$x_i < x_1$
$x_i > x_1$

```
template <typename T, typename P,
          typename C>
class kdTree_t {
  private:
    struct kdNode_t
    {
      P   data;
      T   min, max;
      kdNode_t *left, *right;
      int  axis;
```

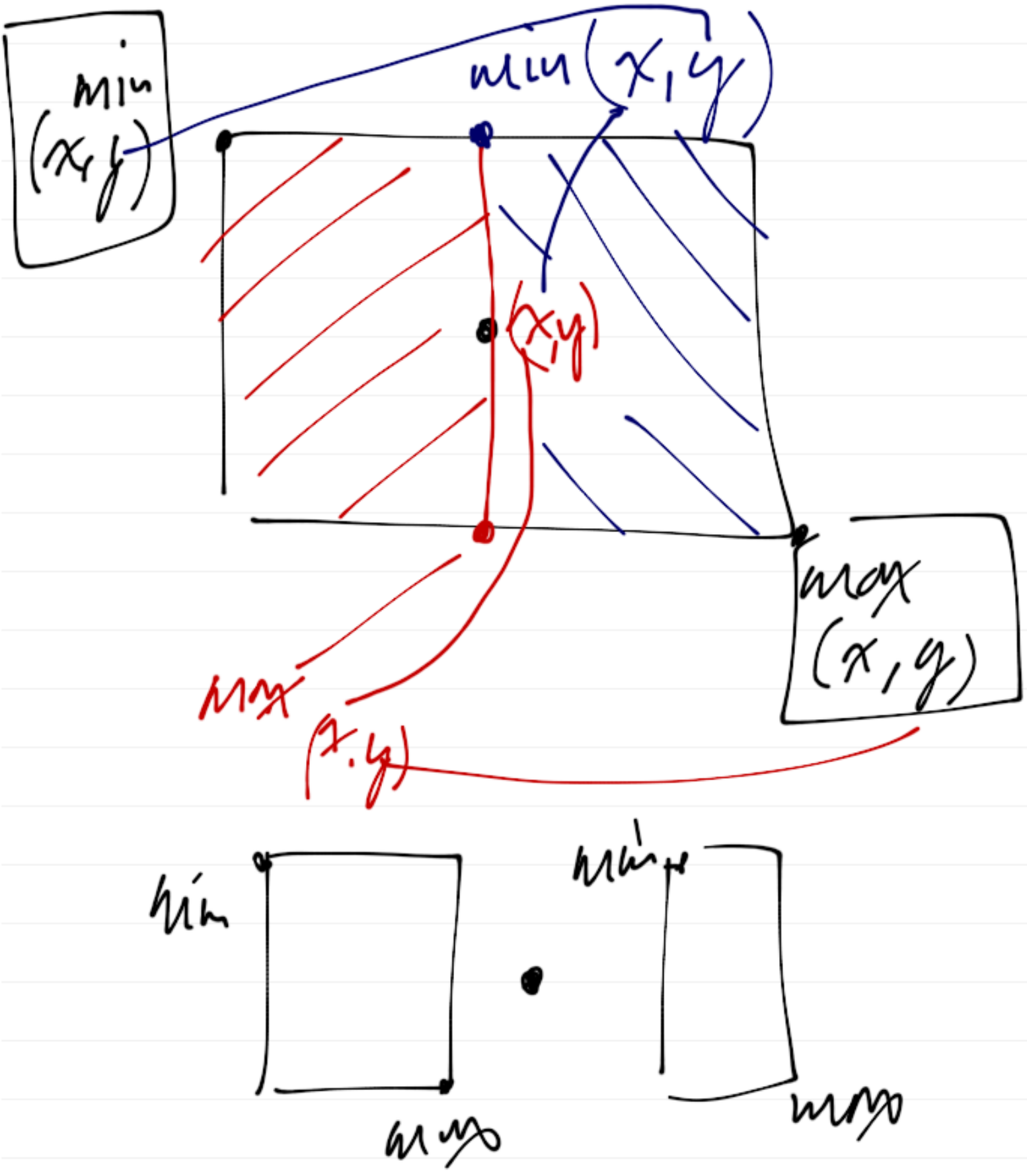
functor

photon_t * → P data;

photon_t → T min, max;

↓ cont'd

```
kdnode_t (const P& d=P(),
    const T& in=T(),
    const T& ix=T(),
    kdnode_t *l=NULL,
    kdnode_t *r=NULL,
    int a=0) : \
    data(d), min(in), max(ix),
    left(l), right(r), axis(a)
    {} };
}; // struct kdnode_t
```


min (x, y)
min (x, y)
(x, y)
max
(x, y)
max
(x, y)

- min, max are T typename values
  (photon_t type of dim = 3)
  or
  (point_t type of dim = 2)

  set at insertion (each node has its own min, max)

- insertion into tree is done en masse once

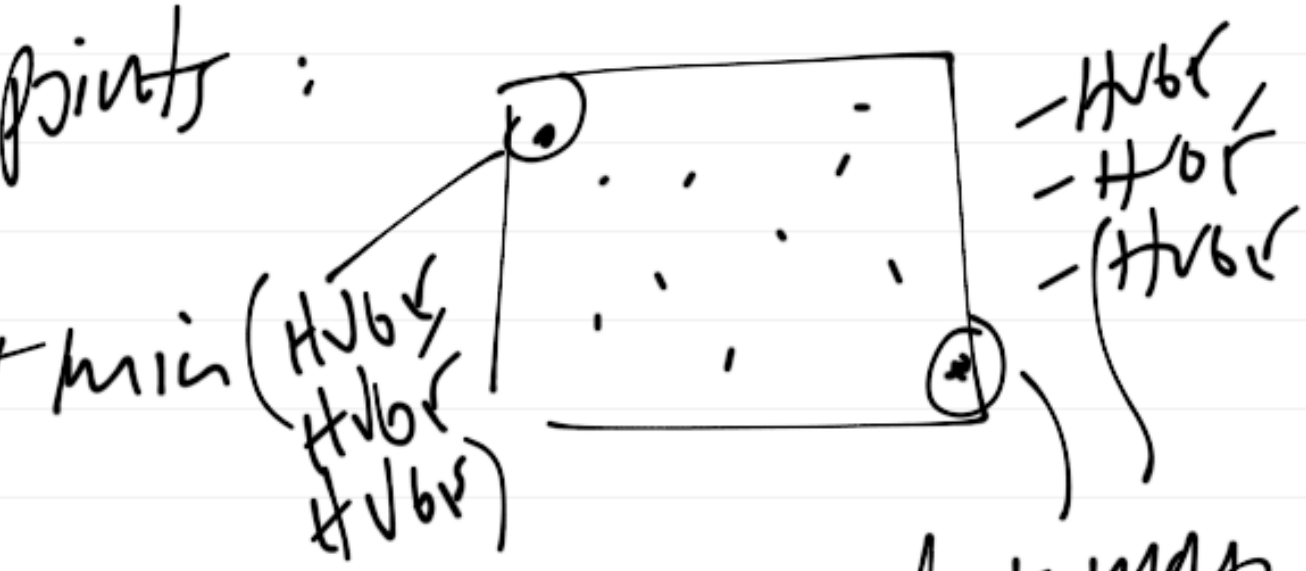
```
std::vector<photon_t*>
    photons;

photon_t min(vec_t(HUGE,
                   HUGE,
                   HUGE));
photon_t max(vec_t(-HUGE,
                   -HUGE,
                   -HUGE));
```

from #include <cmath>

setting position of min & max photons

#ifndef INFINITY
#define INFINITY HUGE
#endif

points :



point_t min(HVbr, HVbr, HVbr)

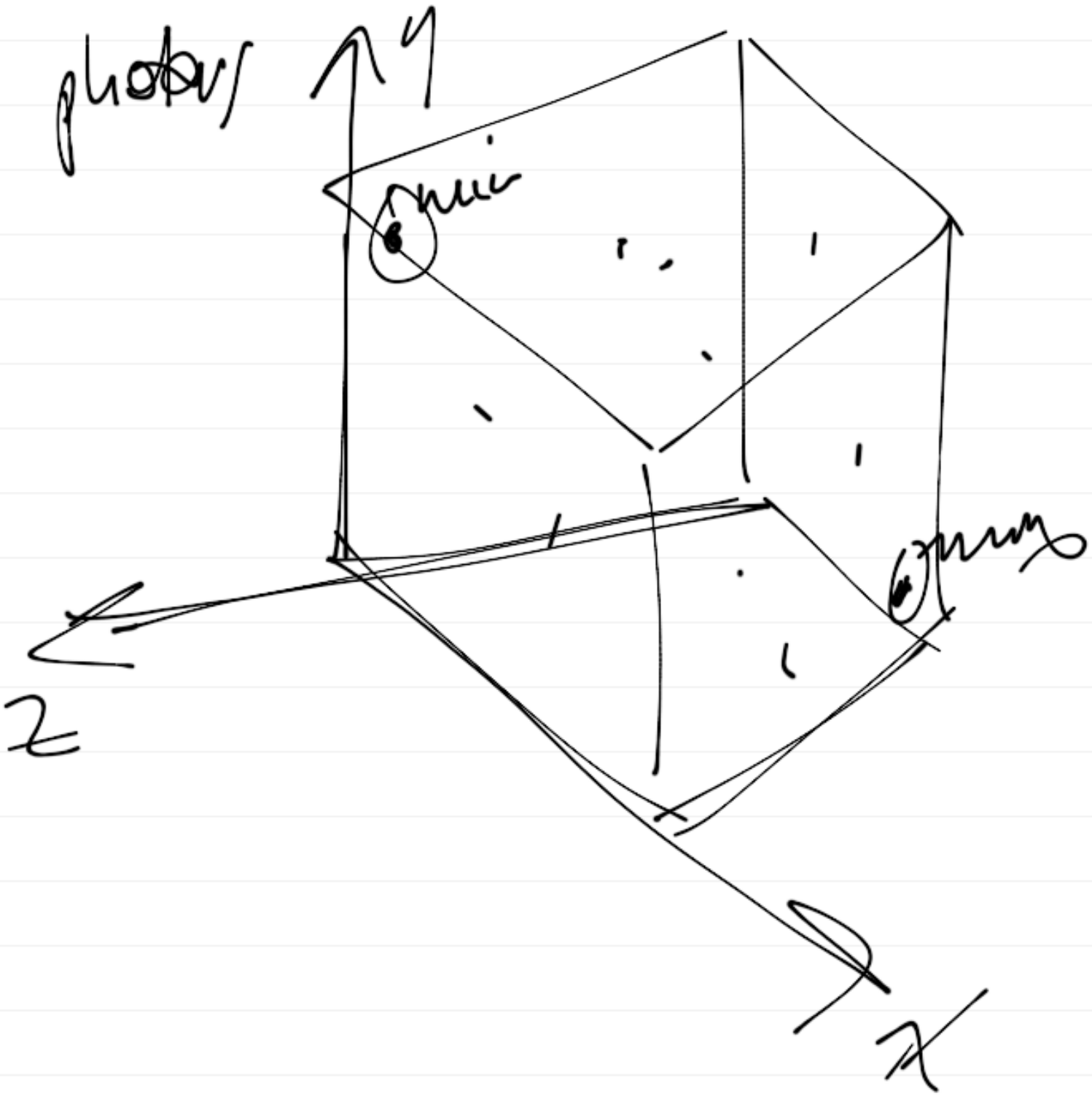
point_t max(HVbr, HVbr, HVbr)

```
for(i=0; i < points.size(); i++)
    if (points[i] < min)
        min = points[i];
    if (points[i] > max)
        max = points[i];
}
```

operator=

operator>

operator<


plotar
y
z
x

- for both points & planar

bool operator< (const
point_t& rhs)
return true if all of
x, y of this < x, y of
rhs
}

bool operator< (const
point_t& rhs)
return true if all of
x, y, z of this < x, y, z of rhs

- std::vector<photon_t*> photons

returned by

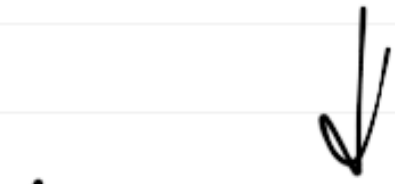
model_t::shoot(photons)

passed by ref.

- in main.cpp:

kdtree_t<photon_t, photon_t*, photon_c> kdtree;

↓



model.shoot(photons);

// find min, max
photons in photons vector
// (these define bounding
volume)

```
std::vector<photon_t*>::
    iterator pitr;

for(pitr=photons.begin();
    pitr<photons.end();
    pitr++) {
  if(**pitr < min)
    min=(**pitr);
  if(**pitr > max)
    max=(**pitr);
}
```

```
std::cerr << "printing
        photons";
std::ofstream ofs;
ofs.open("photons.pts");
for(pitr = ...; pitr++)
    ofs << *pitr << " "
        << (*pitr)->getPwr()
        << std::endl;
ofs.close();
std::cerr << "done." << std::endl;
```

- for photon mapper,
  next, for each photon,
  scale its power
  by $\frac{1}{\text{number of photons}}$

kdtree.insert(photons,
  min,
  max)

// ray trace

- photon data structure:

pwr (power, wattage) like light bulb

- photons either bounce (reflect), transmit, or stick

class photon_t : public
ray_t

{

:

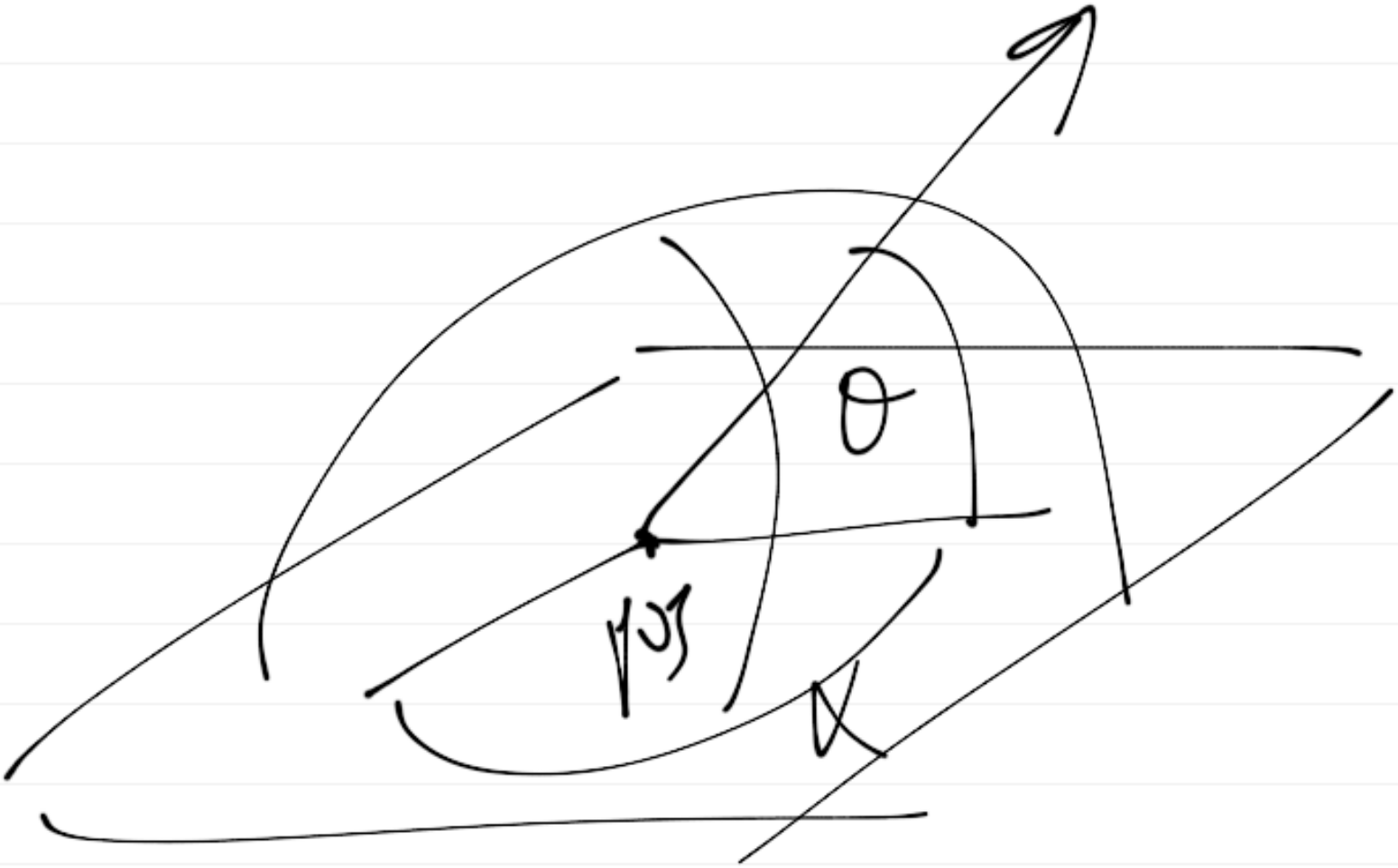
:

private:

Vec_t pwr;

we get from ray_t
dis (a double)
pos (Vec_t)
dir (Vec_t)

```
class photon_t : public ray_t {
  public:
    // constructor
    photon_t() : \
        ray_t(), \
        pw(10.0, 10.0, 10.0)
    { }

    photon_t(const vec_t &o)
      : ray_t(o, grand_hemi(),
              0.0),
        pw(10.0, 10.0, 10.0) { }
```



θ: elevation

α: azimuth

```
double photon_t::distance(vec_t& rhs)
{
    vec_t diff = pos - rhs;
    return (sqrt(diff.dot(diff)))
}
int dim() const { return 3; }
const double& operator[](int k)
    return pos[k]
```