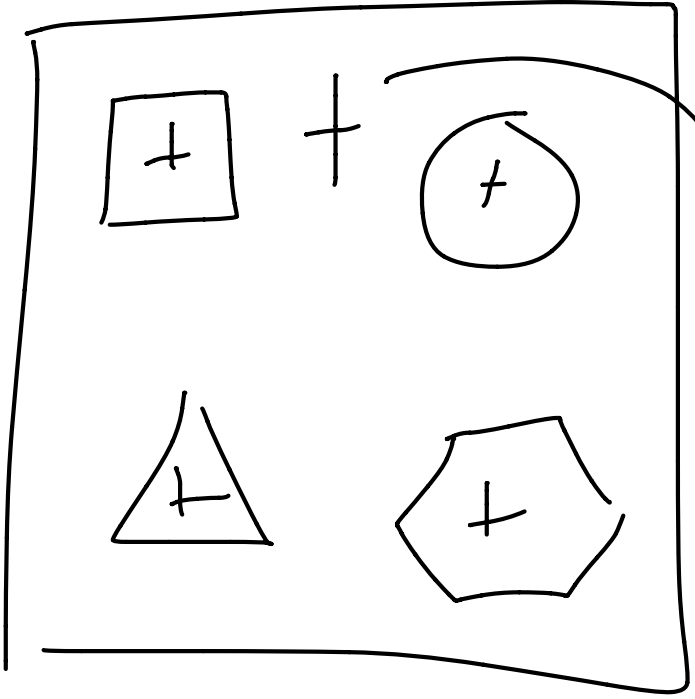
kd-tree: §12.6 in text

- data structure for organizing spatial data

  k-dim data

  (k = 2, 3 for us)

- in graphics, spatial subdivision data structure



your (e.g. game player)

- each obj; has (x,y) center

ALL OBJECTS STATIC,

kd-tree intended for

query: which obj. is

closest to me?

- naive approach:

my location (x, y)

$r_{min} = \infty$

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
- we want $O(\lg n)$
  e.g. 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, e.g.,

level 0, use x

level 1, use y

level 2, use z

level 3, use y

- for 3D, cycle thru

x, y, z, x, y, z, ...

0 1 2 3 4 5

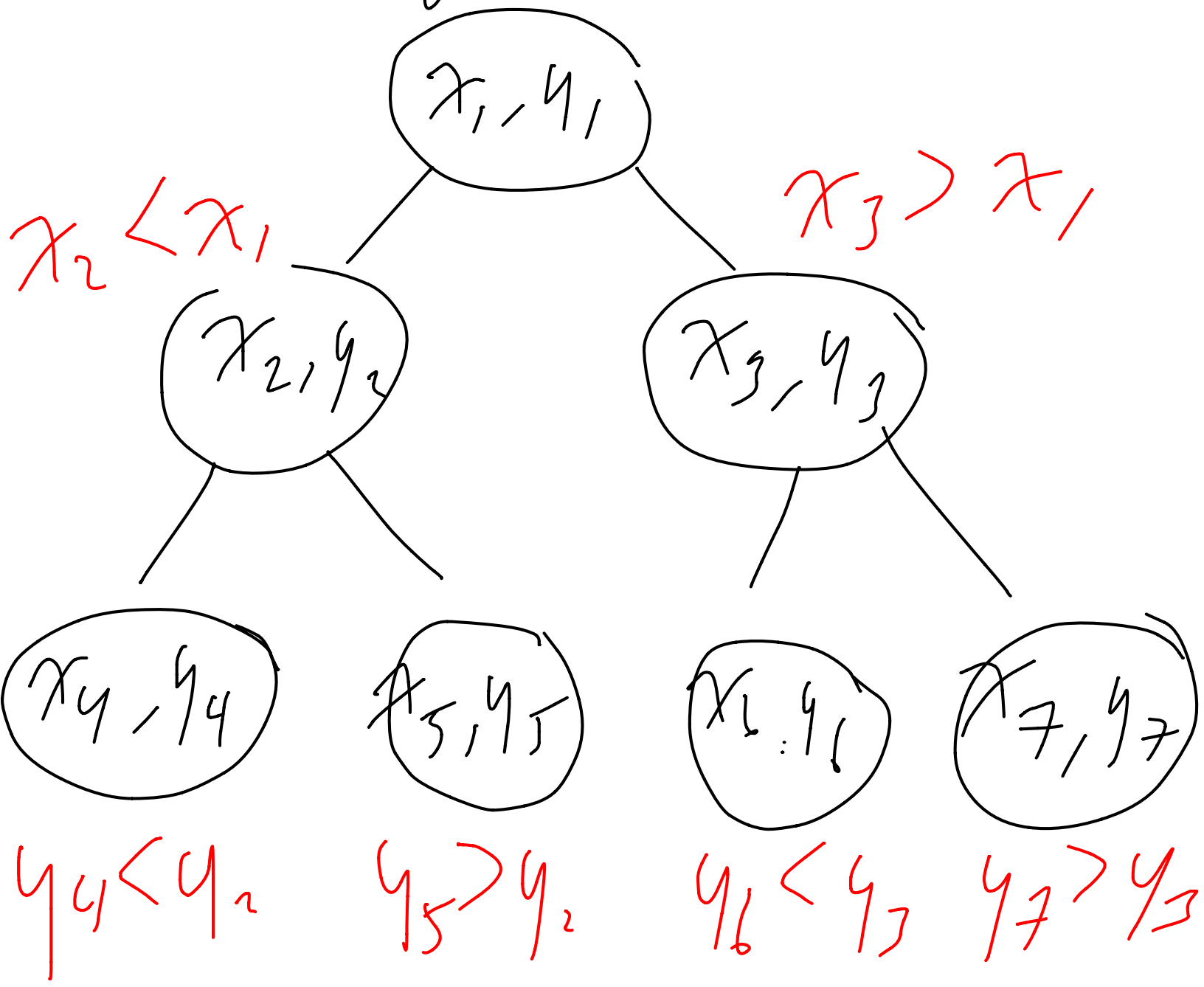
- at each level, use 'axis' index:

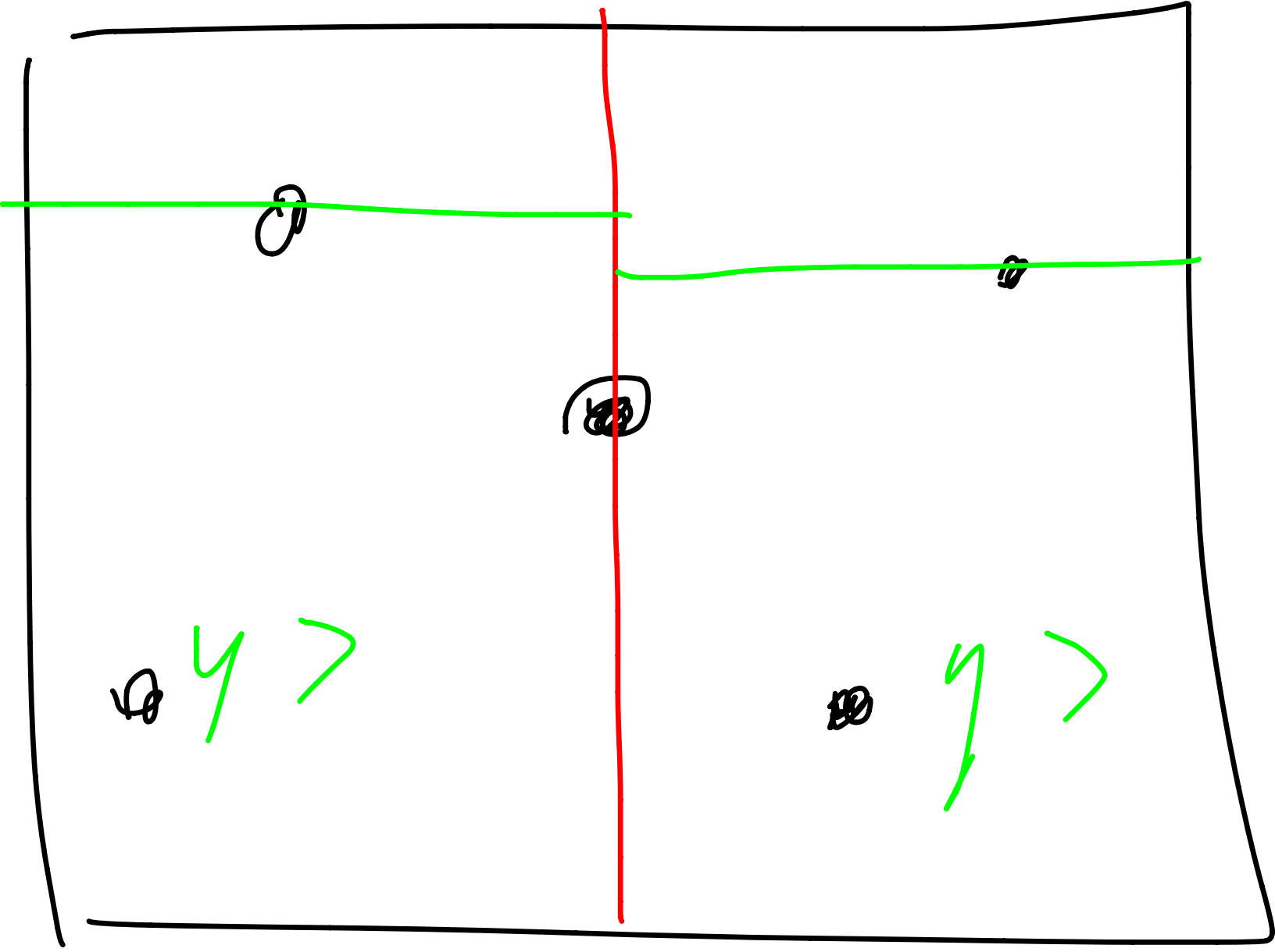
$$axis = (axis + 1) \% dim$$

mod

returned by obj stored in the photon.dim() == 3)

think often like this:





all $x <$ all $x >$

template <typename T, typename P,
typename C>

class KdTree_t {

private:

struct KdNode_t

{

photon_t * → P data;

→ T min, max;

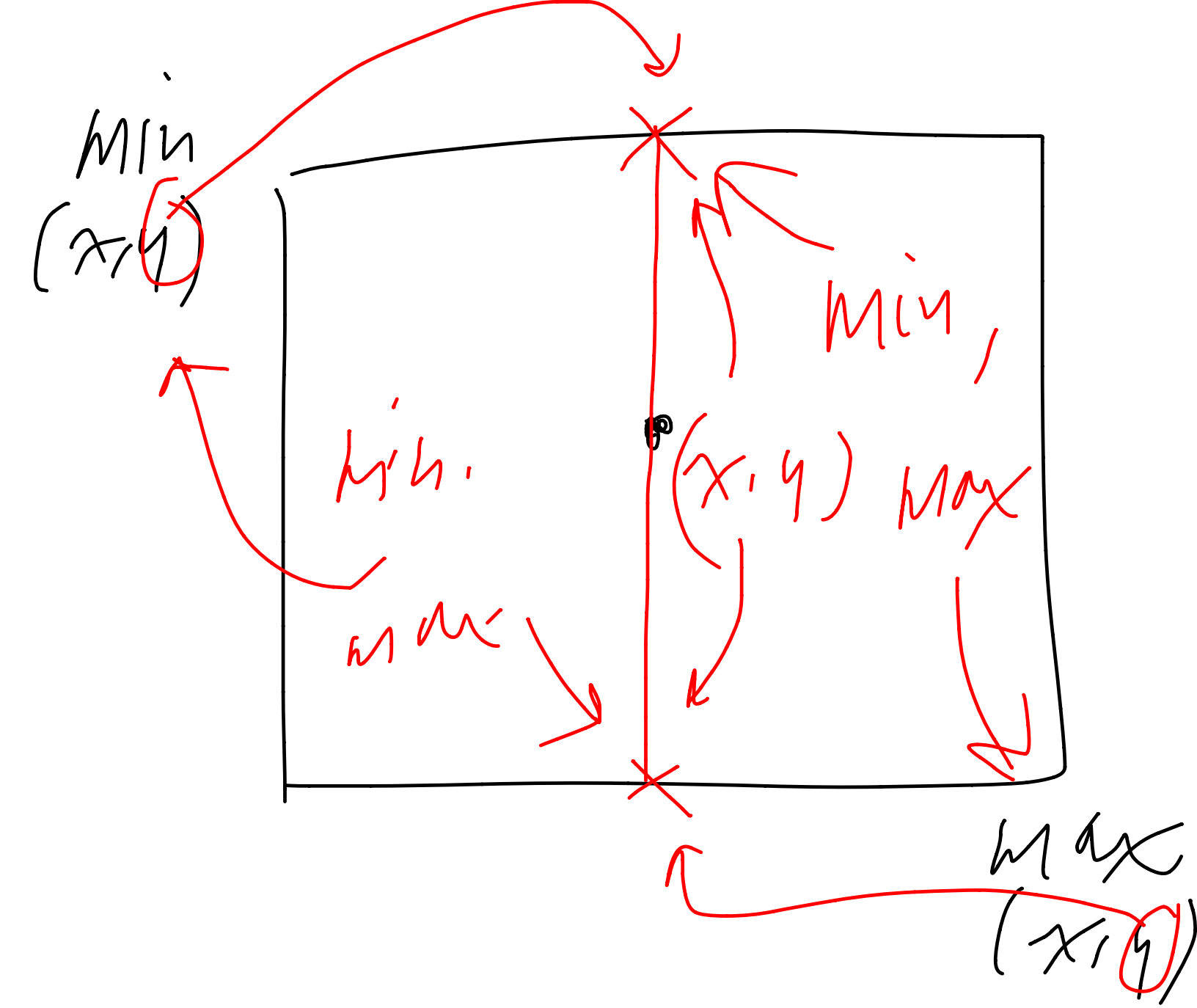
photon_t

KdNode_t *left;

KdNode_t *right;

int axis;

```
kdnode_t(const PA& d=P(),
    const TA& in=T(),
    const TA& 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,
Min.
(x,y)
Max
max
Max
(x,y)

- min, max $\underline{\underline{T}}$ values
(photon-t, of dim = 3)
set at insertion level
(each node has its
own min, max)

- insertion to tree is
done en masse since

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

photon_t min(
    Vec_t(HUGE,
          HUGE,
          HUGE));

photon_t max(
    Vec_t(-HUGE,
          -HUGE,
          -HUGE));
```

```
kdtree_t<photon_t,
         photon_t*,
         photon_c>
kdtree;

model.shoot(photons);
// find min, max
// photons (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);
}
```

operator<

operator>

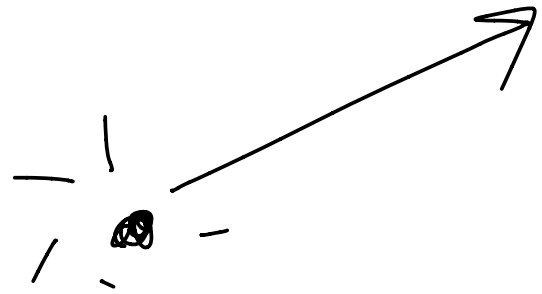
operator=

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

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

```
KdTree.insert(photons,
              min, max);
//ray trace
```

- photon data structure



(photon: has position,
has direction,
has power
(wattage,
like a light bulb)

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

class photon_t : public
ray_t
{

↑ we get
dis (a double),
pos (a vec_t),
dir (a vec_t)

pwr (a vec_t)

.
.
.

private:
vec_t pwr;

class photon_t : public ray_t {

public:

// Constructor (overloaded)

photon_t() : \

ray_t(), \

pwr(100.0, 100.0, 100.0) \

{ };

photon_t (const Vec_t &o):

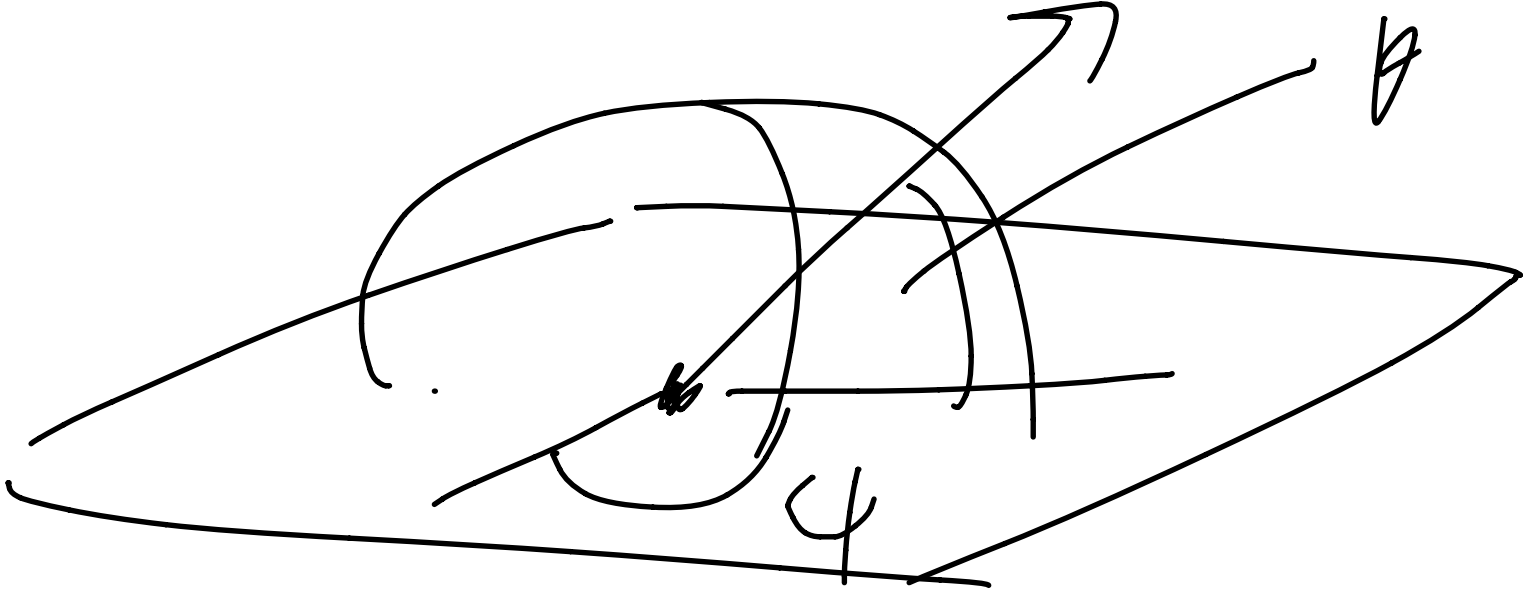
ray_t (0, gen_rand_hemi(),

0.0), \

pwr (100., 100., 100.) \ { };

ray pos

ray dir



θ : elevation $(0 : \pi)$ ~~2π~~

φ : azimuth $(0 : 2\pi)$

In photon_t class interface:

```
bool operator<(const
                photon_t &
                rhs)
{
  return true
  if all x,y,z of this <
    x,y,z of rhs
}

bool operator>(...)
```

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

```
class photon_t : ray_t {
    :
};

class photon_c
{ // function object
    (functor)
  public:
  photon_c(int inAxis=0):
      axis(inAxis) { };
bool operator()(const photon_t& p1,
  const photon_t& p2) const {
    private:          return(p1[axis]
int axis;                 < p2[axis]);}
};
```