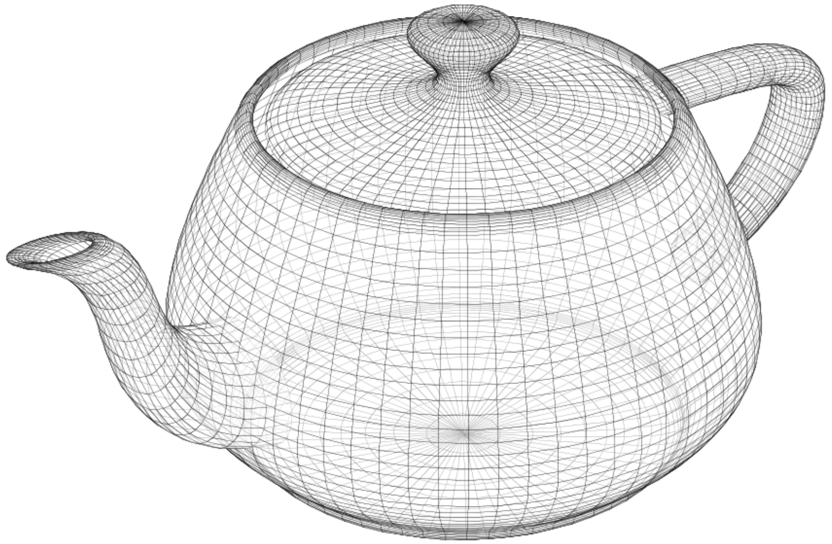


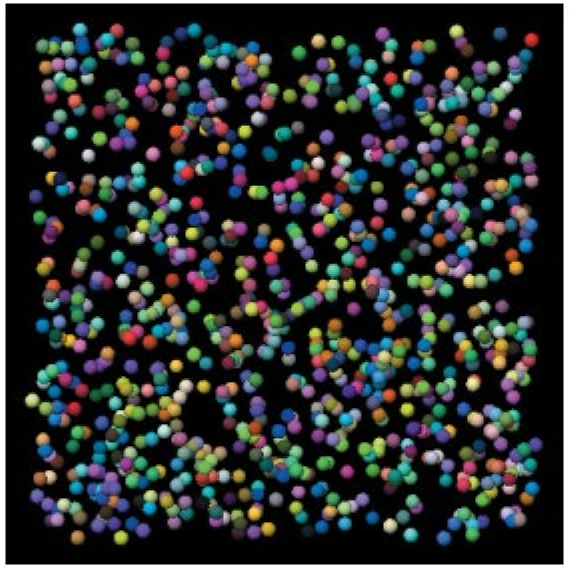
Geometric Data Structures

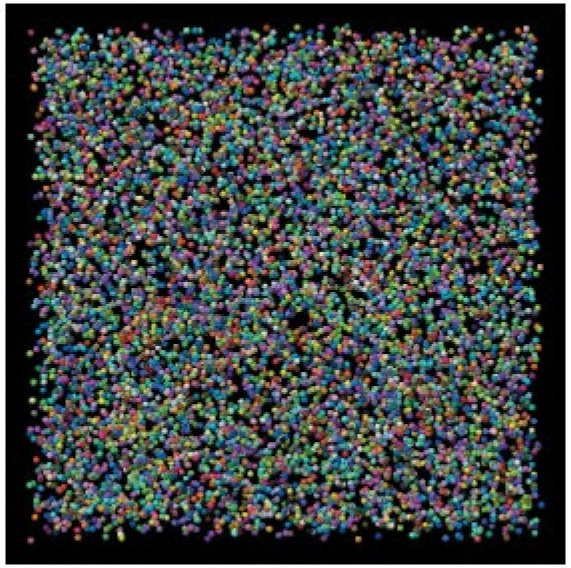
Regular Grids

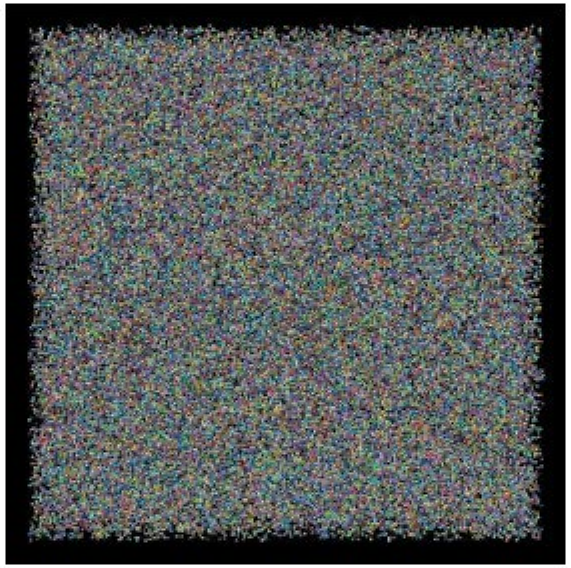
Production Computer Graphics

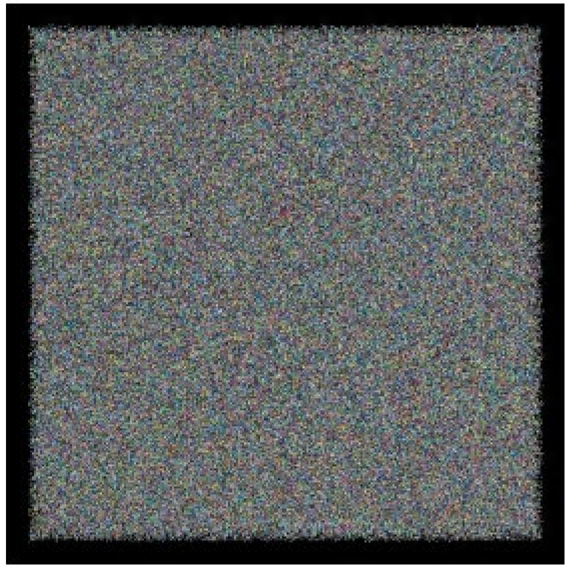
Eric Shaffer

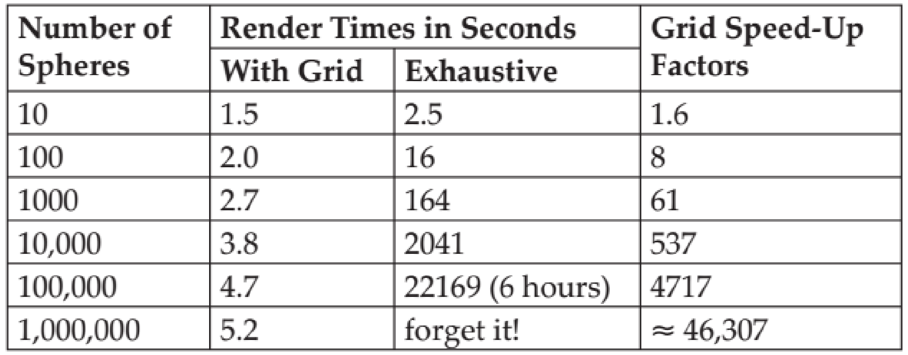
Ray Tracing is Intense


1000 Spheres


10,000


100,000


1,000,000



Number of Spheres	Render Times in Seconds		Grid Speed-Up Factors
	With Grid	Exhaustive	
10	1.5	2.5	1.6
100	2.0	16	8
1000	2.7	164	61
10,000	3.8	2041	537
100,000	4.7	22169 (6 hours)	4717
1,000,000	5.2	forget it!	≈ 46,307

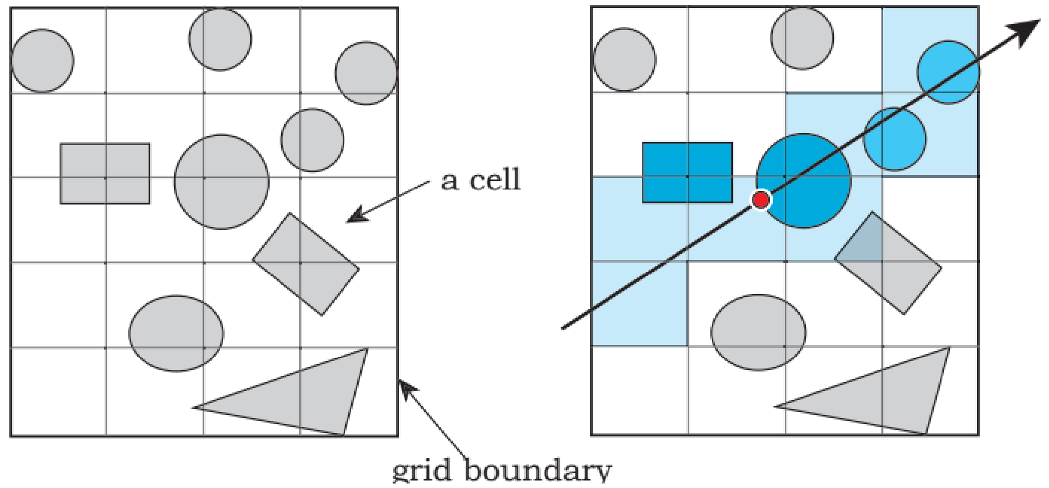
From *Ray Tracing from the Ground Up* by Kevin Suffern (2007)

- 400x400 image using 1 ray per pixel, no shadows
- 450 MHz Mac G3(!?) with 512 MB RAM

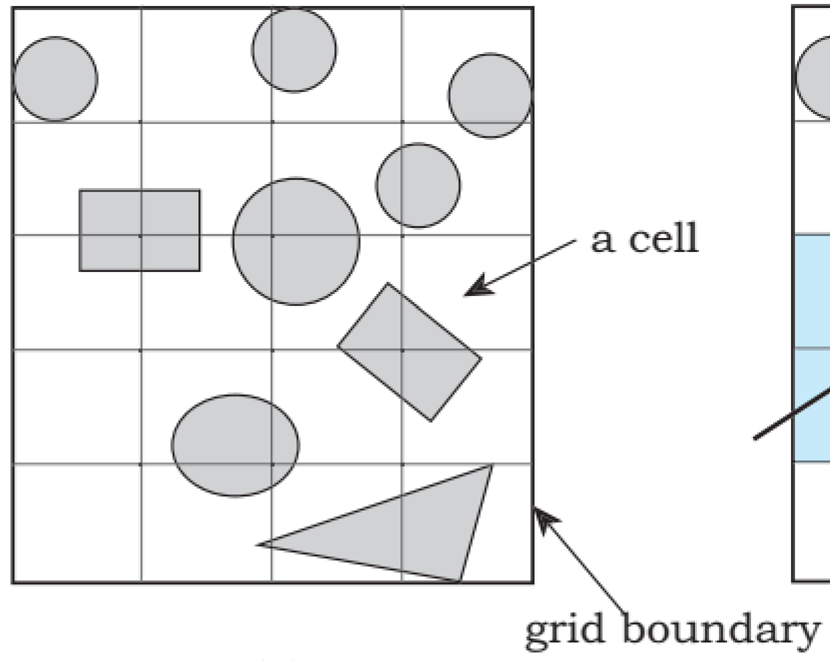
Ray Tracing is Intense

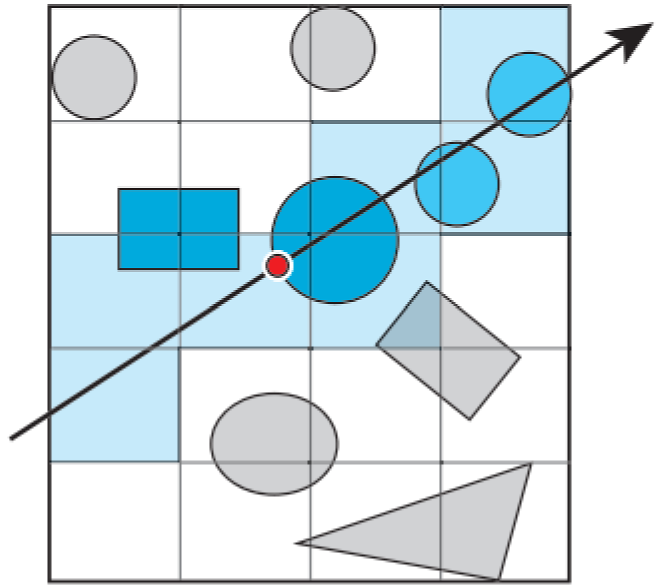
Testing all rays against all objects is…computationally expensive

Should be possible exclude objects far away from a ray from being tested

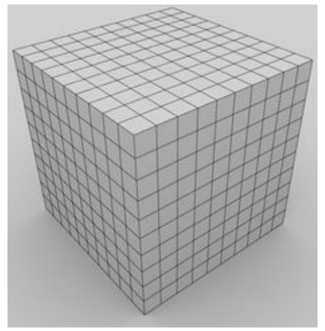


Objectives





The diagrams here are in 2D but the grid generalizes to 3D which is where we will use it.



Be able to use geometric data structures to accelerate ray-tracing

- Know how to compute/intersect a bounding box
- Be able to implement a regular grid to accelerate ray tracing

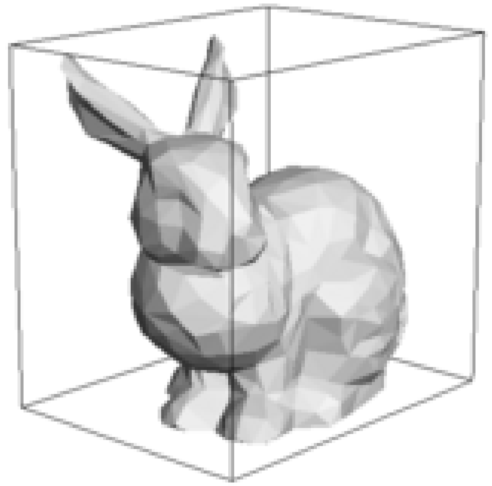
Bounding Boxes

Complex geometric models require expensive tests for ray intersections

Boxes are fast to test for ray intersection

Can implement a “quick rejection” test on the bounding box of a complex object

- Usually, lots of rays will miss an object



Axis-Aligned Bounding Box (AABB)

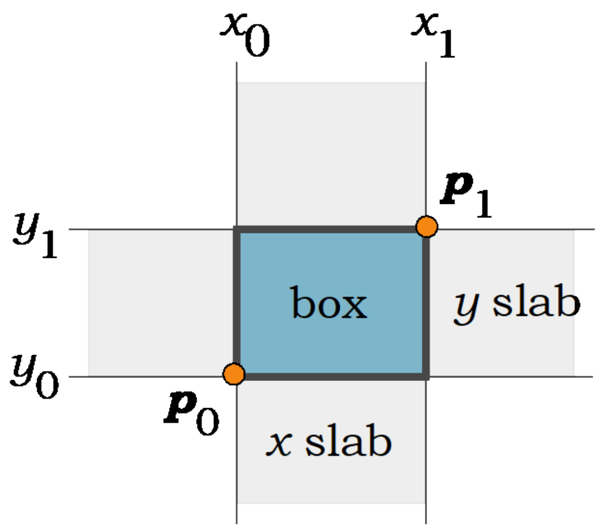
Box is defined by

- min point $p_0=(x_0,y_0,z_0)$
- max point $p_1=(x_1,y_1,z_1)$

Box is $[x_0, x_1] \times [y_0, y_1] \times [z_0, z_1]$

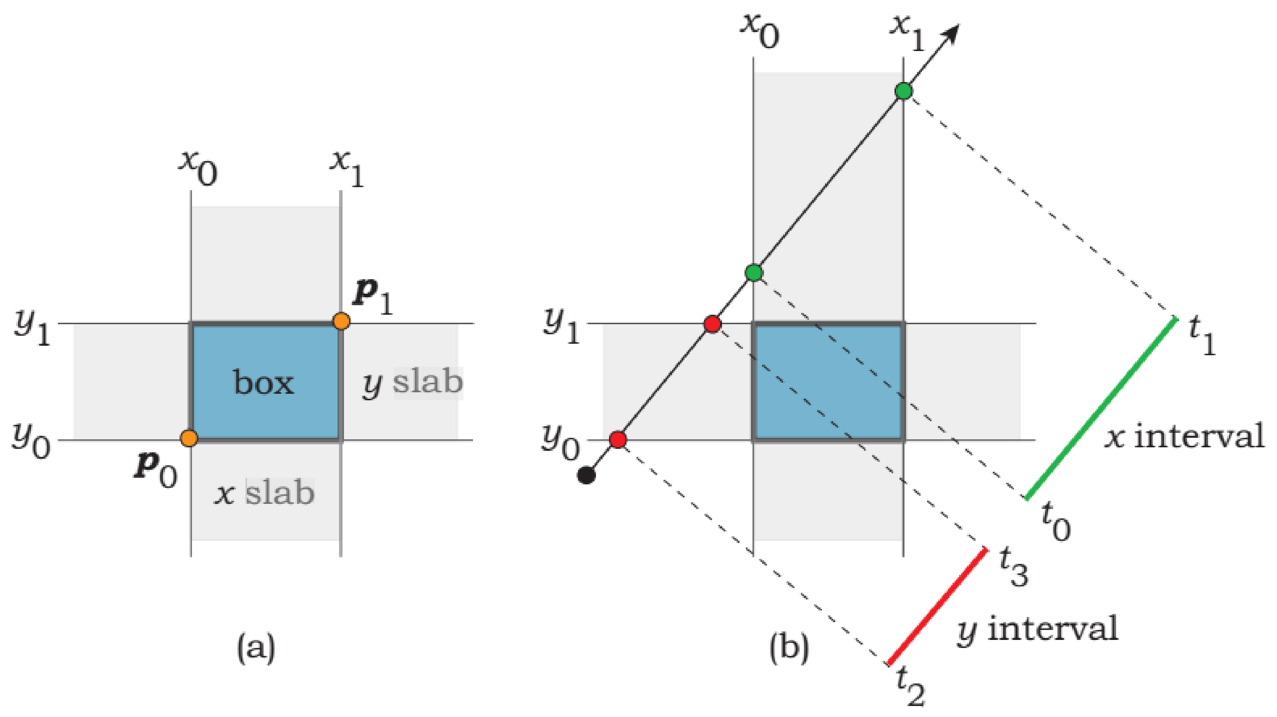
How can we efficiently compute the box?

- Imagine you are given a triangle mesh…
- What is the bounding box for all those triangles?



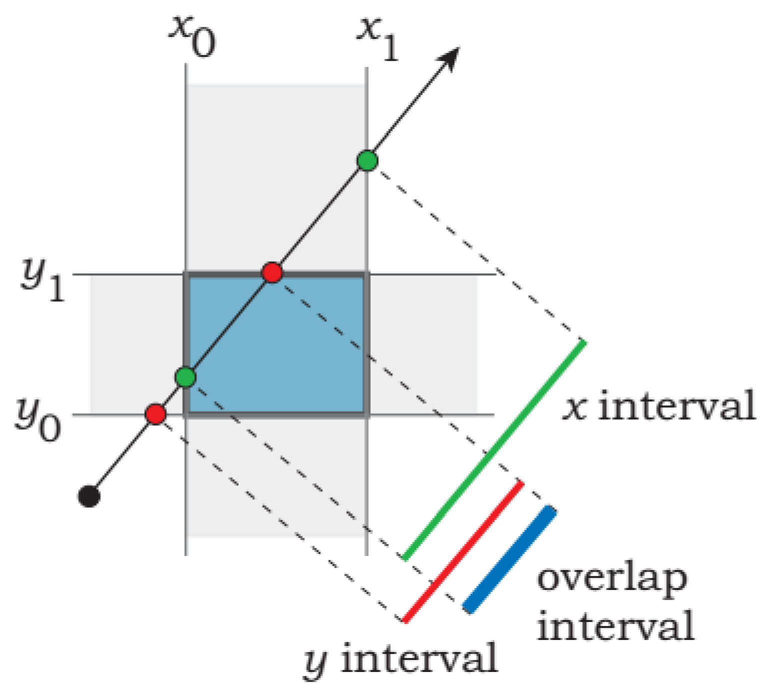
Box-Ray Intersection Test

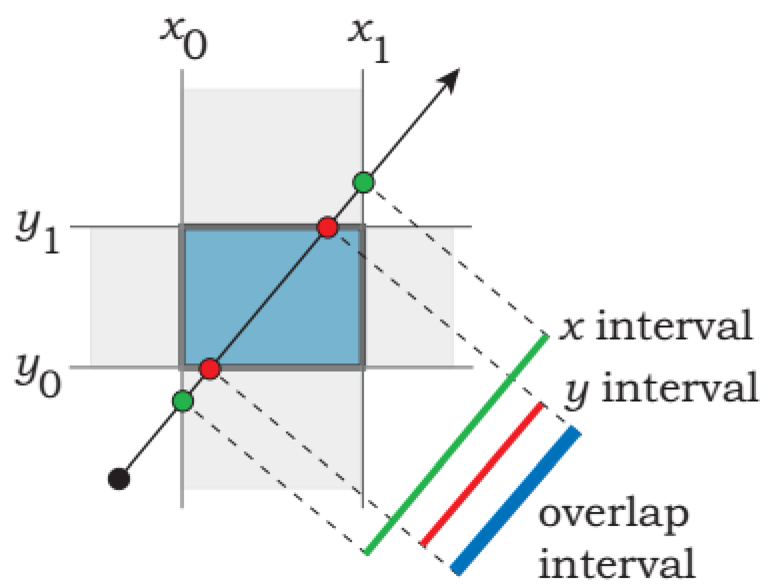
- Box is defined by slabs along each axis
- We will look at 2D case and generalize to 3D
- Ray misses the box when the slab intersection intervals do not overlap
  - How would you test this?



Box-Ray Intersection Test

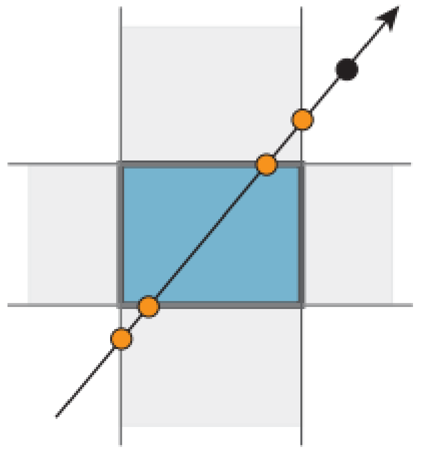
Check if largest entering t value is less than smallest exiting t value





Box-Ray Intersection Test

But ray can also miss if smallest exiting t value is negative



Computing Slab Intersections

```
bool
BBox::hit(const Ray& ray) const {

     double ox = ray.o.x; double oy = ray.o.y; double oz = ray.o.z;
     double dx = ray.d.x; double dy = ray.d.y; double dz = ray.d.z;

     double tx_min, ty_min, tz_min;
     double tx_max, ty_max, tz_max;

     double a = 1.0 / dx;
     if (a >= 0) {
          tx_min = (x0 - ox) * a;
          tx_max = (x1 - ox) * a;
     }
     else {
          tx_min = (x1 - ox) * a;
          tx_max = (x0 - ox) * a;
     }

     double b = 1.0 / dy;
     if (b >= 0) {
          ty_min = (y0 - oy) * b;
          ty_max = (y1 - oy) * b;
     }
     else {
          ty_min = (y1 - oy) * b;
           ty_max = (y0 - oy) * b;
     }

     double c = 1.0 / dz;
      if (c >= 0) {
          tz_min = (z0 - oz) * c;
          tz_max = (z1 - oz) * c;
     }
     else {
          tz_min = (z1 - oz) * c;
          tz_max = (z0 - oz) * c;
     }

     double t0, t1;

     // find largest entering t value

     if (tx_min > ty_min)
          t0 = tx_min;
     else
          t0 = ty_min;

     if (tz_min > t0)
          t0 = tz_min;

     // find smallest exiting t value

     if (tx_max < ty_max)
          t1 = tx_max;
     else
          t1 = ty_max;

     if (tz_max < t1)
          t1 = tz_max;

     return (t0 < t1 && t1 > kEpsilon);
}
```

Geometric Data Structures

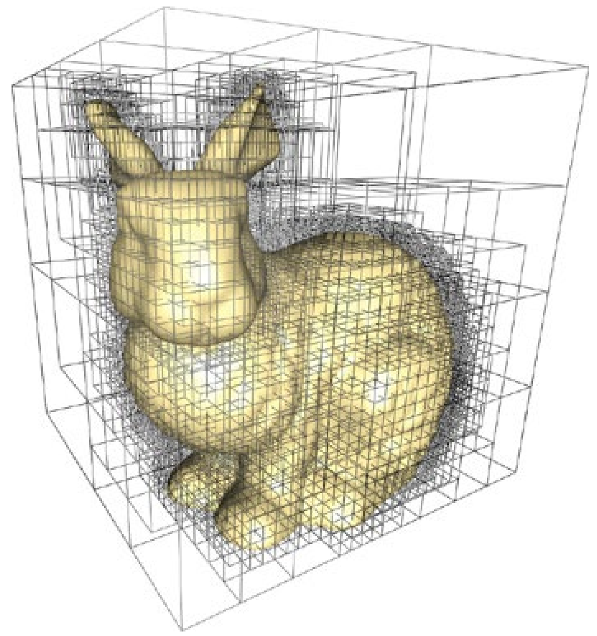
We can do more than just use bounding boxes

What about a data structure that partitions space?

- And we only test objects in the sections of space the ray traverses

Lots of choices

- Octree
- BSP-Tree
- KD-tree
- ***Regular grid***

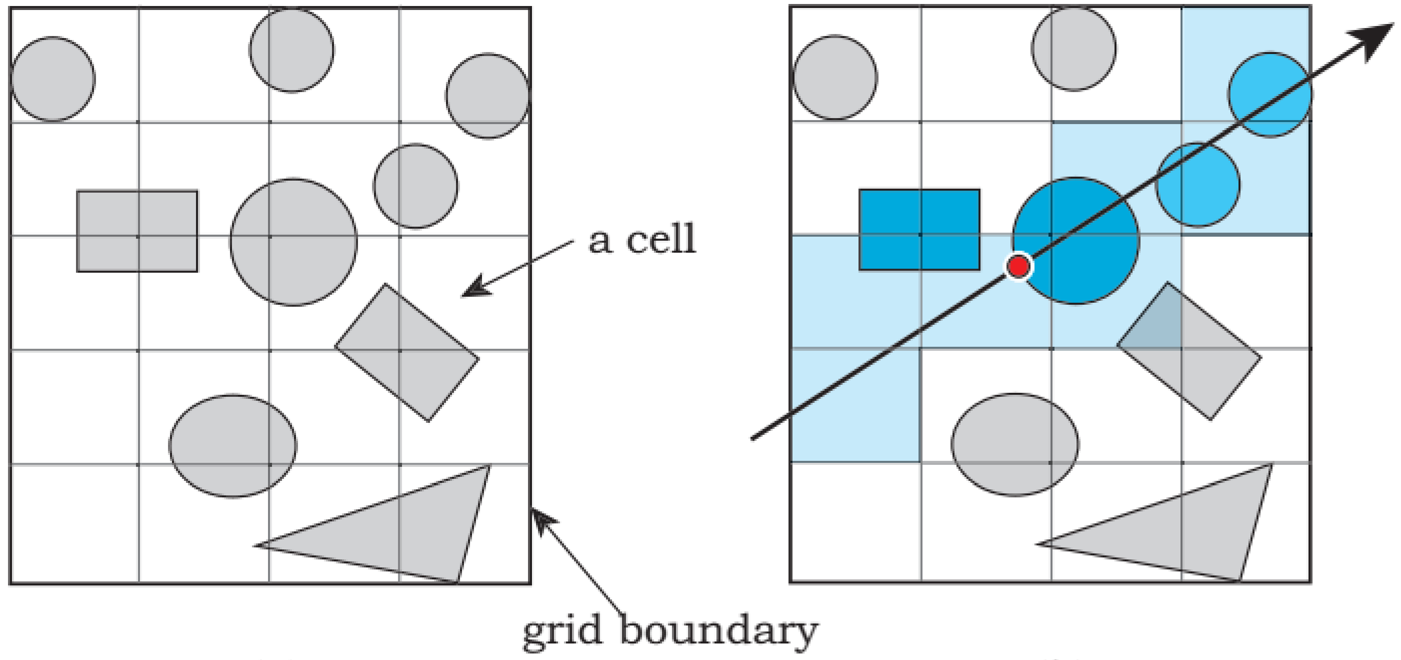


An advantage of grids is that they are fast to construct…in what rendering situation would this be important?

Regular Grids

- A regular grid is an axis-aligned box
- Subdivided into smaller axis-aligned boxes called ***cells***
- Each cell has the same shape and size

Might be more accurate to call this a ***uniform*** grid…but *Ray Tracing from the Ground Up* calls it ***regular***



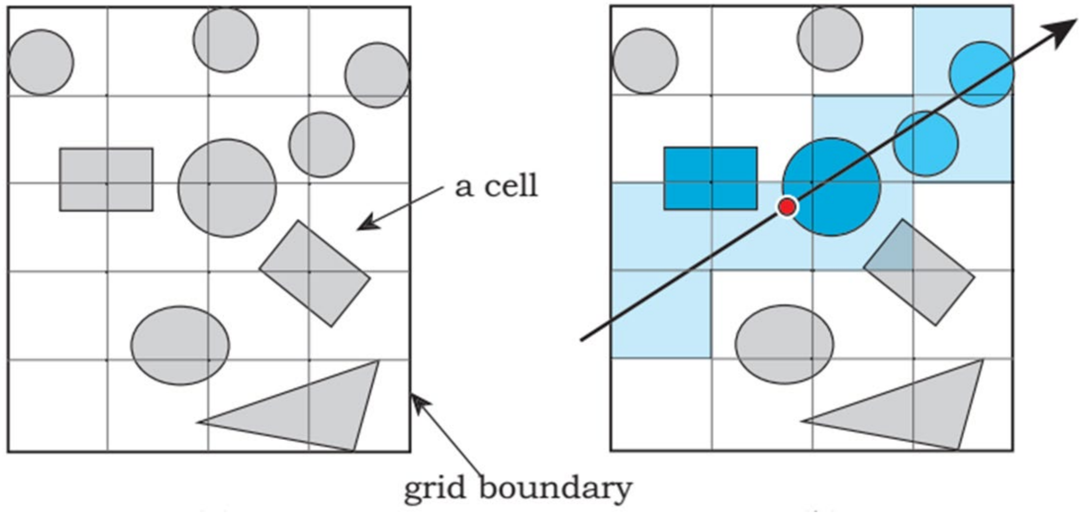
ILLINOIS

Regular Grids

Each cell stores a list of object that

- Are contained in the cell
- Are partially contained in the cell
- *Might* be in the cell….

Why might? What insertion procedure would result in that condition?



ILLINOIS

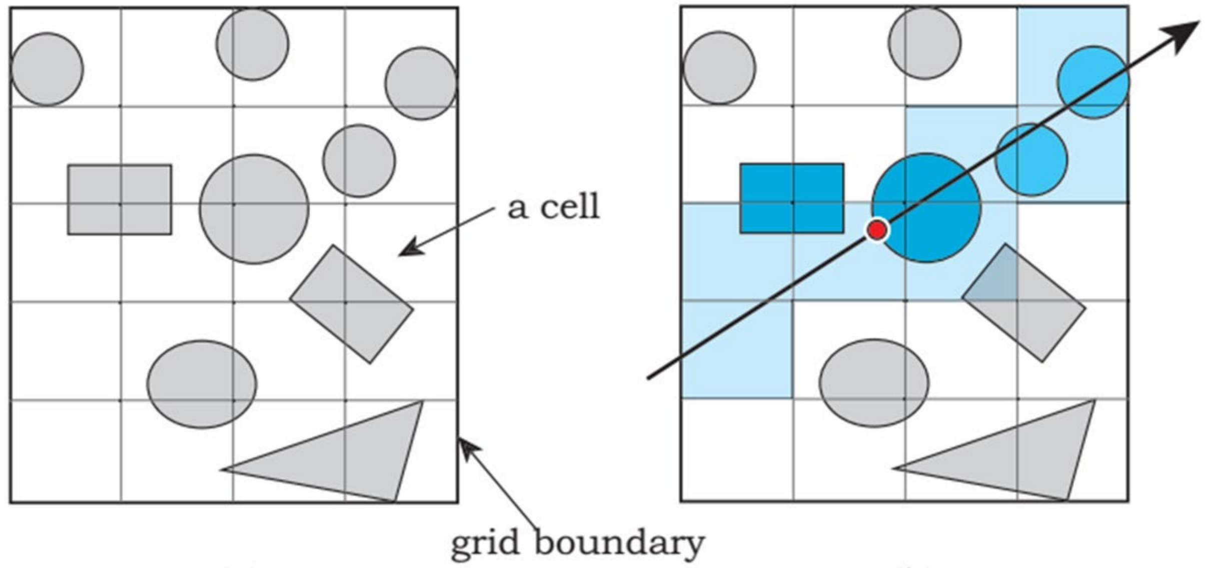
Tracing Rays through Regular Grids

You can march a ray through the grid

- Only test for intersections against objects in the cells you march through

You can also terminate the march on the closest hit

- Is closest = first?



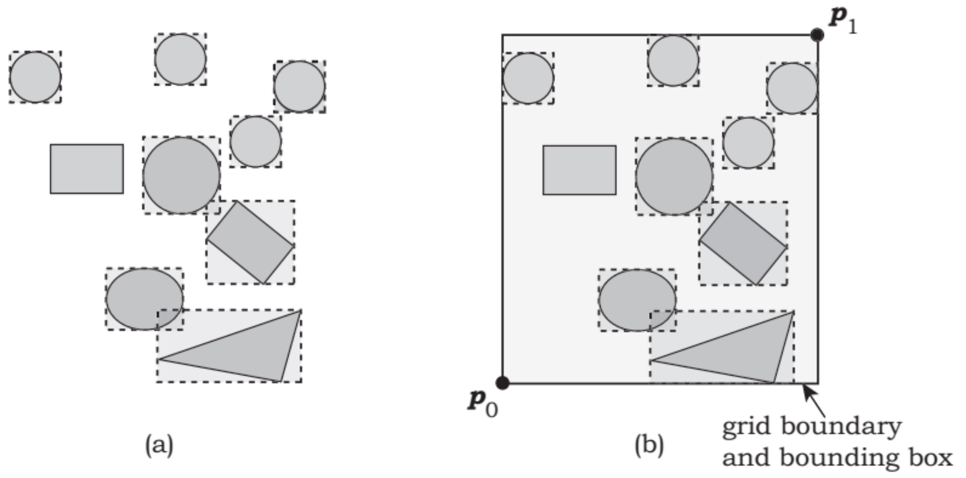
Constructing a Regular Grid

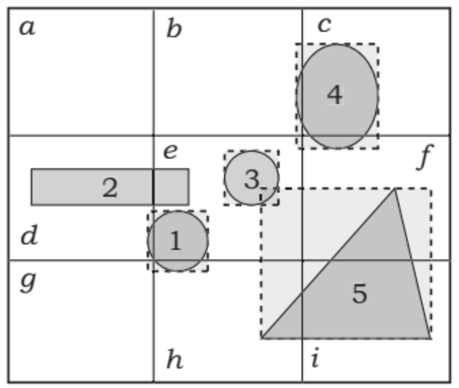
Procedure is simple:

- Compute bounding box of all objects
- Divide up the bounding box into cells
- Insert the objects into the grid

Implementation detail:

- Underlying data structure should be a hash table
- Hash functions should map cell (i,j,k) to array index
- Only allocate space for a cell if it is occupied





A Hash Function for 3D Points

1. Convert a point p=(x,y,z) into (i,j,k) indices
   - In other words, map p to grid cell address
2. Covert (i,j,k) to binary
3. Interleave the bits of the 3 binary numbers

$$\ldots z_1 y_1 x_1 z_0 y_0 x_0$$

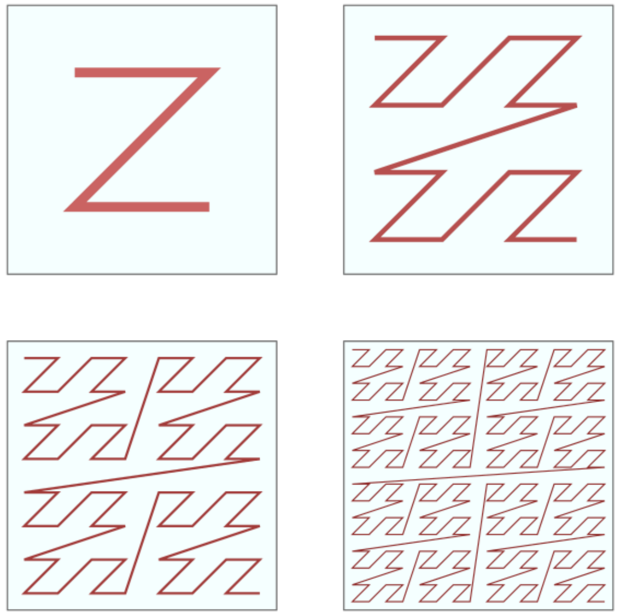
This generates a ***Morton Code***

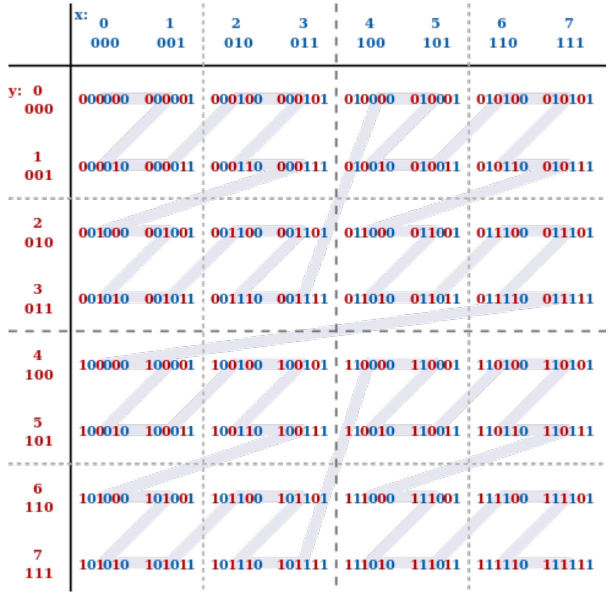
- Also known as a Z-order curve

Example: (5,2,1)

- (0101,0010,0001)
- 000001010101 = index 85 in a linear array

Where does the Z-order curve place neighbor grid cells in the array? Why is this important?





	x: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
y: 0 000	000000	000001	000100	000101	010000	010001	010100	010101
1 001	000010	000011	000110	000111	010010	010011	010110	010111
2 010	001000	001001	001100	001101	011000	011001	011100	011101
3 011	001010	001011	001110	001111	011010	011011	011110	011111
4 100	100000	100001	100100	100101	110000	110001	110100	110101
5 101	100010	100011	100110	100111	110010	110011	110110	110111
6 110	101000	101001	101100	101101	111000	111001	111100	111101
7 111	101010	101011	101110	101111	111010	111011	111110	111111

How Many Cells?

Want to avoid having large number of objects in a single cell

Isotropic cells are probably best for uniformly distributed objects

- We'll try to make cells as cube-like as we can

- n_x, n_y, n_z are the number of cells along each axis
- w_x, w_y, w_z is the length of the grid along each axis
- Let n be the number of objects we have to store
- m allows you to pick how many cells per object
    - m=2 yields 8ish cells per object, good number
    - why use +1?

$$s = \sqrt[3]{\frac{w_x w_y w_z}{n}}$$

$$n_x = \left\lfloor \frac{m w_x}{s} \right\rfloor + 1$$

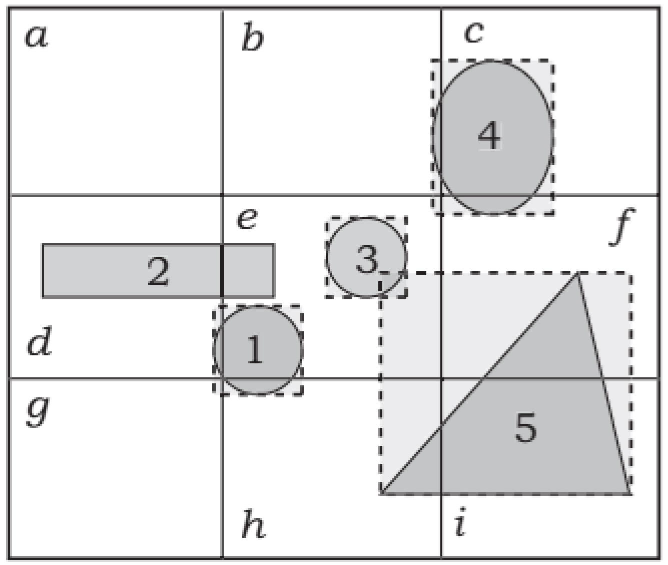
$$n_y = \left\lfloor \frac{m w_y}{s} \right\rfloor + 1$$

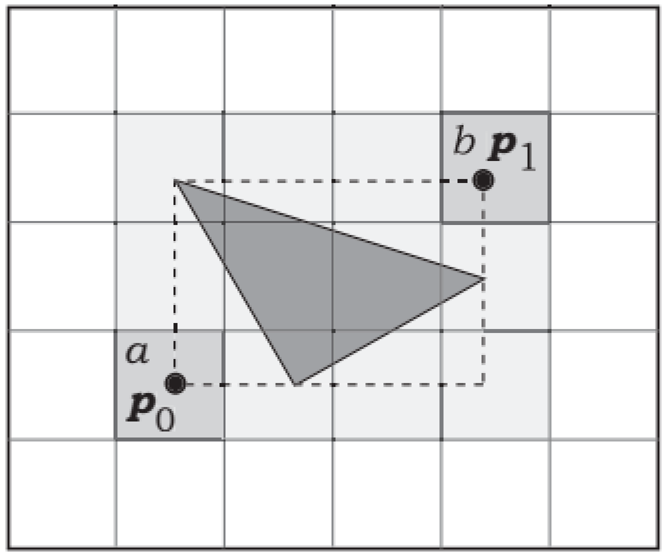
$$n_z = \left\lfloor \frac{m w_z}{s} \right\rfloor + 1$$

ILLINOIS

Inserting Objects into the Grid

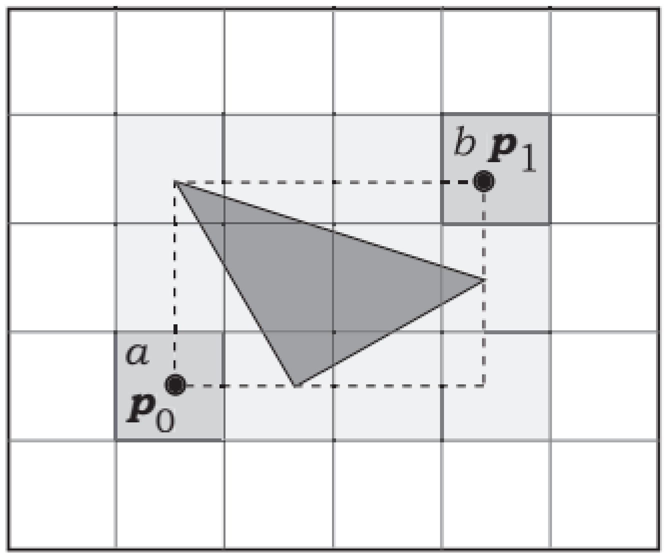
1. Compute the bounding box for the object
2. Intersect the box with the grid
3. Insert object into all cells overlapped by box





ILLINOIS

Inserting Objects into the Grid



Compute cell indices

$$C(p_0) = (i_o, j_0)$$
$$C(p_1) = (i_1, j_1)$$

Iterate…

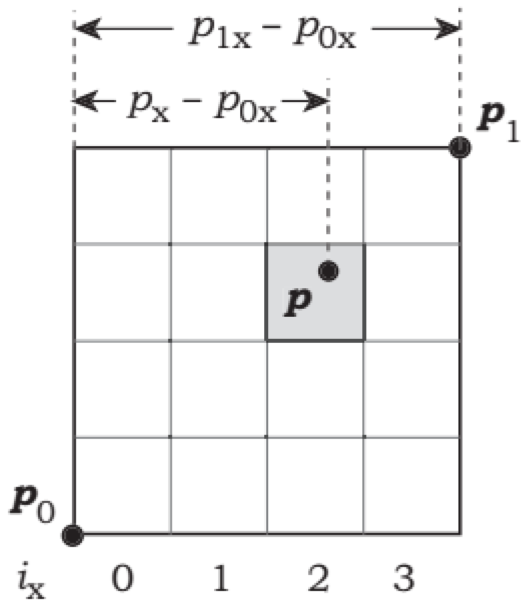
```
for (i=i0; i++;i<=i1)
  for (j=j0; j++;j<=j1)
    //insert object into cell (i,j)
```

Computing a Cell Index for a Point

Computing x-axis index for a point p

$$f(p_x) = {(p_x - p_{0x})}/{(p_{1x} - p_{0x})}$$

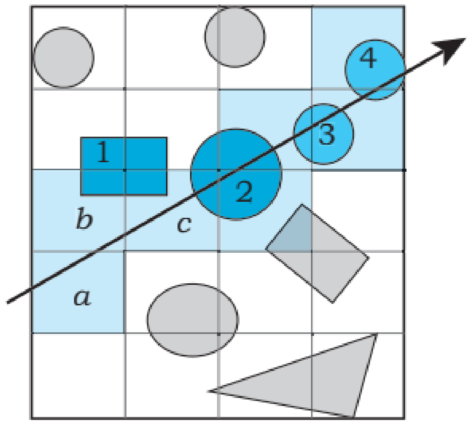
$$i_x = \lfloor n_x f(p_x) \rfloor$$

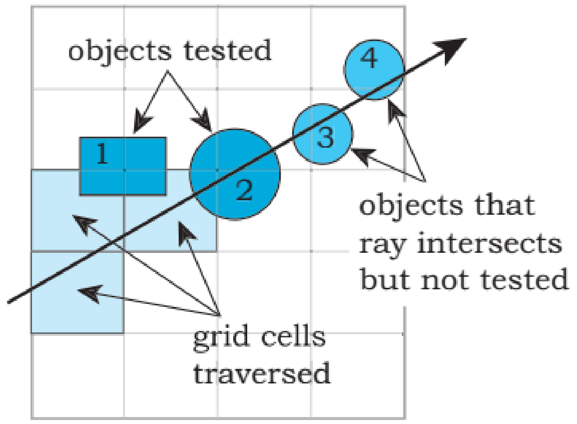


Note that indices must be clamped in range [0,n_x-1]

Grid Traversal Along a Ray

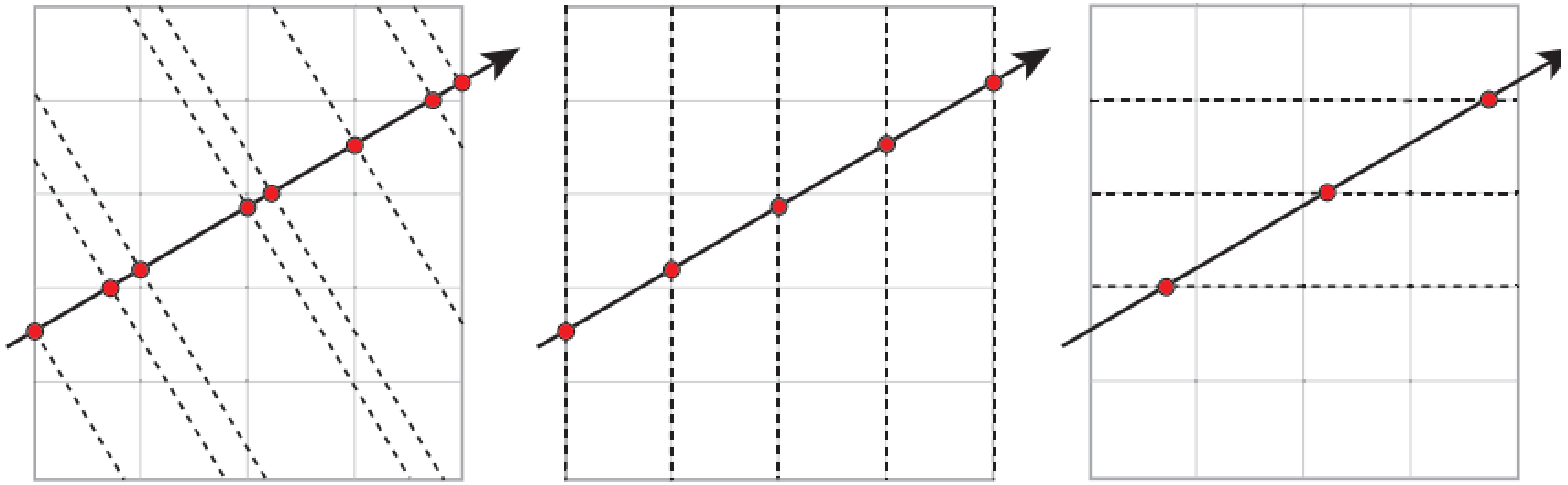
```
if the ray misses the grid's bounding box
     return false

if the ray starts inside the grid
     find the cell that contains the ray origin
else
     find the cell where the ray hits the grid from the
        outside

traverse the grid
```



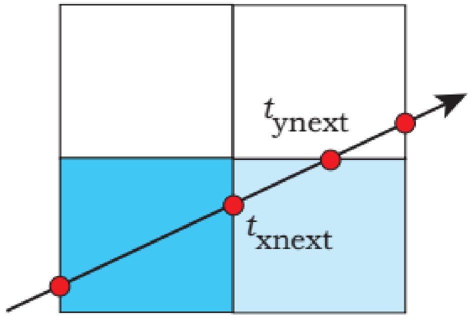


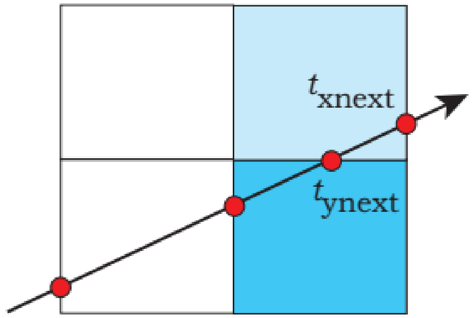
Traversal: Marching Along the Ray

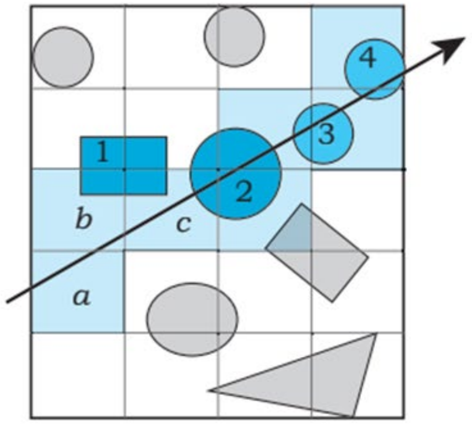
- Intersection of rays with cell faces can be unequally spaced
- They are equally spaced in the x, y, and z directions
- We can compute parametric increments across the cell
  - Figure out, based on smallest t, which face the ray next hits

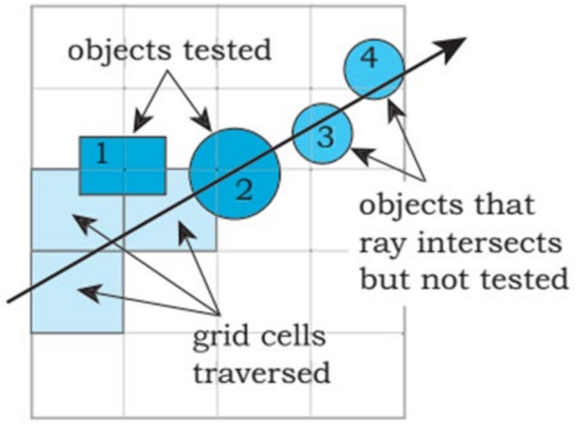


Traversal: Marching Along the Ray









```
float dtx = (tx_max - tx_min) / nx;
tx_next = tx_min + (ix + 1) * dtx;
```

```
float dty = (ty_max - ty_min) / ny;
ty_next = ty_min + (iy + 1) * dty;
```

Theoretical Performance

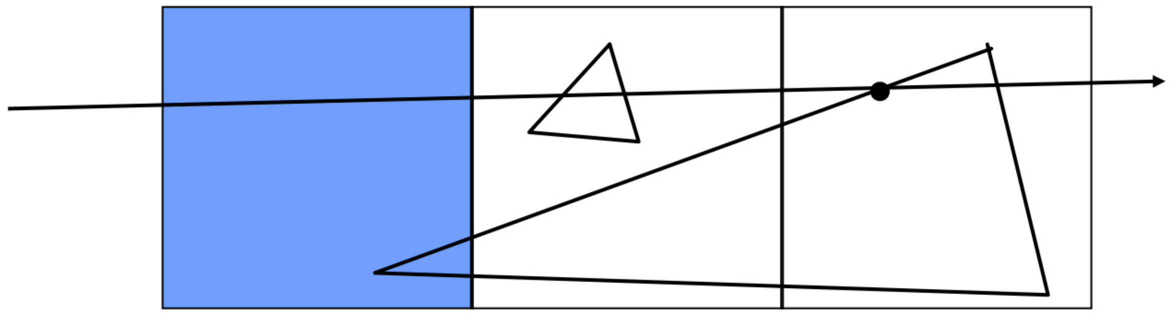
Grid will have $O(\sqrt[3]{n})$ cells in each cardinal direction

In theory rays tested on $O(\sqrt[3]{n})$ objects

In practice $O(\log n)$ tests per ray for a dense scene

ILLINOIS

Mailboxing



A hit on an object may not be in the cell

Need to check the the intersection point falls in cell

- If not, keep marching

Will we end up testing same object multiple times?

- Use mailboxing in which
  - Each ray gets an integer id
  - Store most recent checked ray id and test result with object…like caching

Uniform Grids and Scene Complexity

Work best for uniformly distributed objects

- Objects all have similar complexity

Poor performance for non-uniform complexity in scene

- "Teapot in a stadium" problem – Eric Haines
- Can nest grids or use a tree-based structure