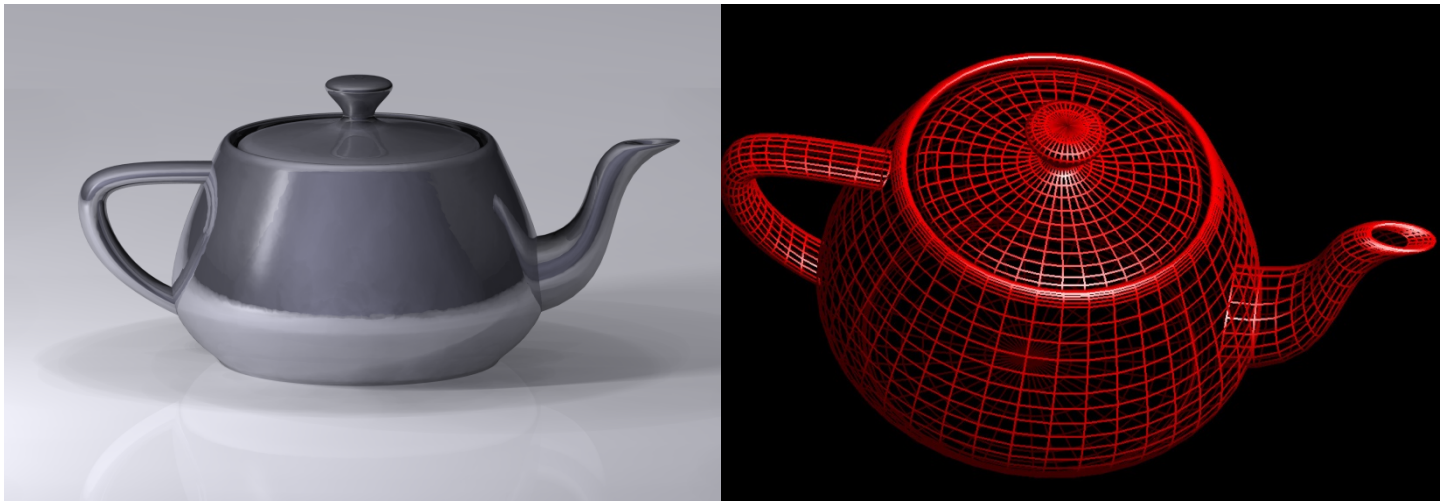


Foundations of Computer Graphics

Saurabh Ray

Today's topics:

Recap of recursive ray tracing. (Sections 12.1 - 12.3 of the textbook.)

Implementation. *See code on Github.*

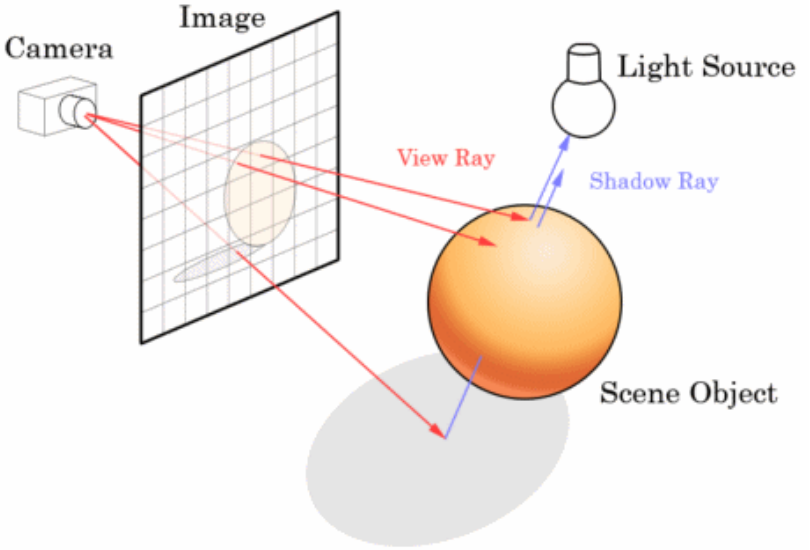
Instancing.

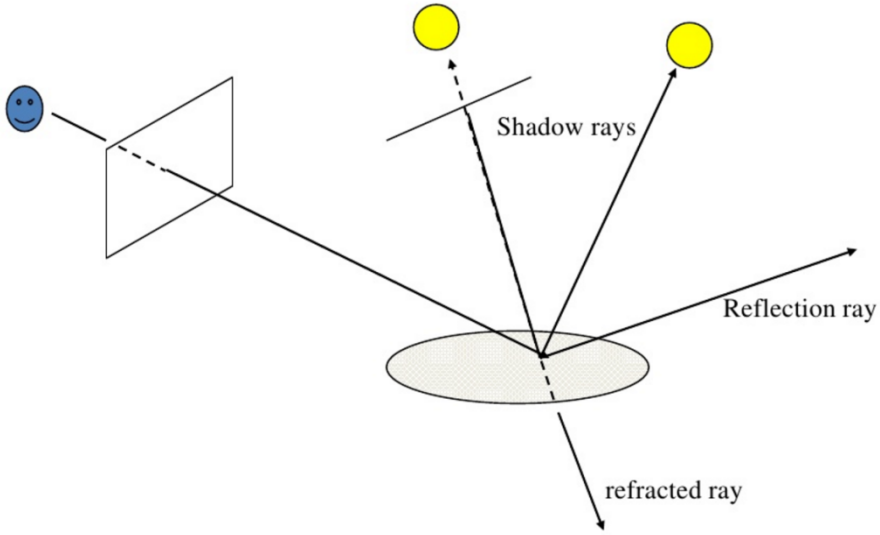
Constructive Solid Geometry.
(Section 9.10.1 of the textbook.)

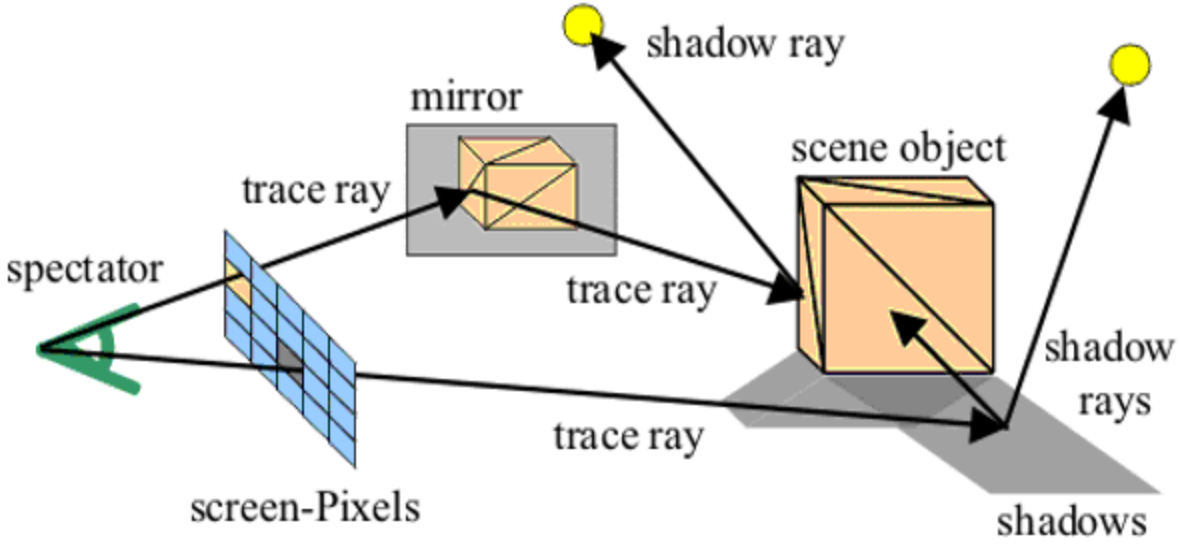
} *Allow us to design complex models with which we can intersect a ray efficiently.*

Spatial data structures. *Data structures for computing the first intersection with a large number of objects efficiently.*

Recursive Ray Tracing







Ray Tracing Implementation

Scene Components: Objects, Materials, Lights, Camera

We also need a few other details:

- size of the image we are producing

- trace depth : how many times the rays bounce before we stop

```
var nx = 512;              // width of image in pixels
var ny = 512;              // height of image in pixels
var trace_depth = 2;       // number of bounces of a ray

var Objects = [
    { type: "sphere", c: vec3(-0.6,0.6,1.5), r: 0.2, material_id: 0},
    { type: "sphere", c: vec3(0,0,0), r: 1, material_id: 1},
    { type: "sphere", c: vec3(-1,-0.3,1.5), r:0.3, material_id: 2},
    { type: "sphere", c: vec3(0.5,0,1.5), r:0.1, material_id: 3}
];
```

Ray Tracing Implementation

```
var Lights = [
    { position: vec3(0,10,0), color: vec3(1,1,1), intensity: 0.6},
    { position: vec3(0,0,10), color: vec3(1,1,1), intensity: 0.5}
];

var ambient_light = vec3(0.1,0.1,0.1);
var background_color = vec3(0,0,0);

var Materials = [
    {color: vec3(1,0,1), reflectivity: 0.2, shininess: 100},
    {color: vec3(0.8,0.8,0.8), reflectivity: 0.8, shininess: 2400},
    {color: vec3(0.1,0.7,0.2), reflectivity: 0.1, shininess: 150},
    {color: vec3(0.6,0.1,0.1), reflectivity: 0.1, shininess: 50}
];



var Camera = {
    location: vec3 (0,0,3.6),
    up: vec3(0,1, 0),
    lookat: vec3(0,0,0),
    fov: 60,         // field of view : in degrees
};
```

Ray Tracing Implementation

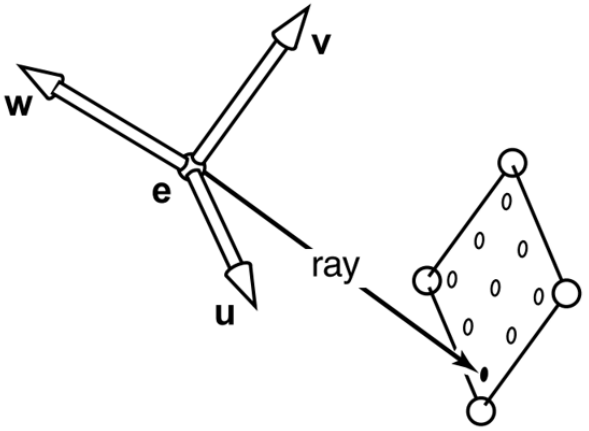
Here is the function that creates the image:

```
window.onload = function init() {
            var i,j;
            var color;
            var canvas = document.getElementById("gl-canvas");
            var ctx = canvas.getContext("2d");

            var imgData = ctx.createImageData(nx, ny);
            var data = imgData.data;

            for(i=0; i<nx; ++i){
                for(j=0; j<ny; ++j){
                    index = 4*(j*nx + i);
                    color = trace(pixelRay(i,j), trace_depth);
                    data[index] = Math.floor(color[0]*255);
                    data[index+1] = Math.floor(color[1]*255);
                    data[index+2] = Math.floor(color[2]*255);
                    data[index+3] = 255;    //opaque
                }
            }

            ctx.putImageData(imgData, 0,0);
}
```

returns ray from camera through the $(i,j)^{th}$ *pixel*

Ray Tracing Implementation

Figuring out the rays:



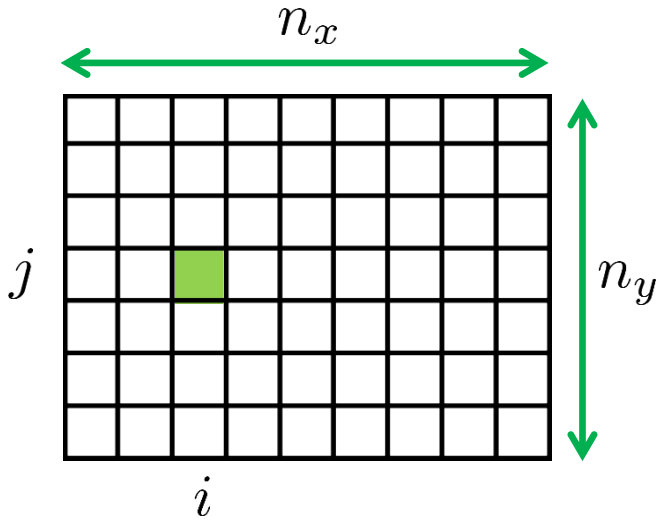
1. Figure out the camera coordinate frame basis vectors $\vec{u}, \vec{v}$ and $\vec{w}$.

2. Aspect Ratio $= nx/ny$ (width / height).

3. Assuming that the distance to the projection plane is 1, figure out the *left (l)*, *right (r)*, *top (t)* and *bottom (b)* on the projection plane.

4. Figure out coordinate of the center C^{ij} of pixel (i, j) in the camera frame.

$$C^{ij} = (l + (i + 0.5)(r - l)/n_x, \\ b + (j + 0.5)(t - b)/n_y, \ -1)$$



5. Ray R_{ij} in world coordinate frame:

   origin: eye position

   direction: $C_x^{ij}\vec{u} + C_y^{ij}\vec{v} + C_z^{ij}\vec{w}$

Ray Tracing Implementation

```
function pixelRay(i,j){
    // return the ray from camera through (i,j)^th pixel
    // i,j increase to right and down as in a picture

    j = ny - j; // invert j

    var n = 1;                  // distance to near plane i.e., projection plane
    var aspect = ny/nx;         // aspect = height/width
    var fov_radians = Camera.fov*Math.PI/180;
    var t = n*Math.tan(fov_radians/2);
    var r = t*aspect;
    var b = -t, l = -r;

    // compute camera basis
    var w = normalize(subtract(Camera.location, Camera.lookat));
    var u = normalize(cross(Camera.up,w));
    var v = cross(w,u);

    var ucomp = l + (r-l)*(i+0.5)/nx;
    var vcomp = b + (t-b)*(j+0.5)/ny;
    var wcomp = -n;

    var raydir = scale(ucomp, u);
    raydir = add(raydir, scale(vcomp, v));
    raydir = add(raydir, scale(wcomp, w));;

    return { e: Camera.location, d: normalize(raydir)};
}
```

Ray Tracing Implementation

Our ray tracer currently only supports spheres.

We need a function to find the intersection of a sphere and a ray.

```
function hit(ray){
    var i, x;
    var obj = null, t = Infinity, loc;
    for(i = 0; i< Objects.length; ++i){
        x = intersect(ray, Objects[i]);
        if(x>=0 && x < t){
            obj = Objects[i];
            t = x;
        }
    }
    if(t < Infinity){
        loc = add(ray.e, scale(t, ray.d));
    }

    return {obj: obj, point: loc, t:t};
}
```

For lighting, we also need to find the normal at a given point on a sphere.

```
function computeNormal(obj, point){
    if(obj.type === "sphere"){
        return normalize(subtract(point, obj.c));
    }
}
```

Ray Tracing Implementation

We need a function to report the first object hit by a ray.

```
function hit(ray){
    var i, x;
    var obj = null, t = Infinity, loc;
    for(i = 0; i< Objects.length; ++i){
        x = intersect(ray, Objects[i]);
        if(x>=0 && x < t){
            obj = Objects[i];
            t = x;
        }
    }
    if(t < Infinity){
        loc = add(ray.e, scale(t, ray.d));
    }

    return {obj: obj, point: loc, t:t};
}
```

We intersect with all objects to find out.

Ray Tracing Implementation

We need a function that does Phong Lighting at a point, given the light direction, view direction, the normal vector and material properties.

```
function shade(position, normal, viewDir, material){
    var i,l, shadowRay;
    var diffuseIntensity, specularIntensity, toLight, lightDir;
    var halfvector;
    var shininess = material.shininess;
    var color = material.color;

    var final_color = vec3(0,0,0);
    var diffuse = ambient_light;
    var specular = vec3(0,0,0);

    for(i = 0; i< Lights.length; ++i){
        l = Lights[i];
        toLight = subtract(l.position, position);
        lightDir = normalize(toLight);

        shadowRay = {e:position, d: lightDir};
        if(hit(shadowRay).t > length(toLight)){
            diffuseIntensity = l.intensity * Math.max(dot(lightDir, normal),0);
            diffuse = add(diffuse, scale(diffuseIntensity, l.color));
            halfvector = normalize(add(viewDir, lightDir));
            specularIntensity = Math.pow(Math.max(dot(halfvector, normal), 0), shininess);
            specular = add(specular, scale(specularIntensity, l.color));
        }
    }

    final_color = add(final_color, mult(diffuse, color));
    final_color = add(final_color, specular);

    return final_color;
}
```

Ray Tracing Implementation

Here is the recursive function that traces a ray:

```
function trace(ray, depth){

    var color = background_color;
    var h = hit(ray);

    if(h.obj!==null){
        var normal = computeNormal(h.obj, h.point);
        var material = Materials[h.obj.material_id];
        var viewDir = scale(-1, ray.d);
        color = shade(h.point, normal, viewDir, material);

        if(depth === 0) return color;

        var reflected_dir = add(ray.d, scale(-2*dot(ray.d, normal),normal));
        var reflected_ray = {e:h.point, d:reflected_dir};
        var reflected_col = trace( reflected_ray, depth-1);
        color = add(color, scale(material.reflectivity, reflected_col));
    }

    return color;
}
```

Instancing

Object —*Transformation M*→ Transformed object

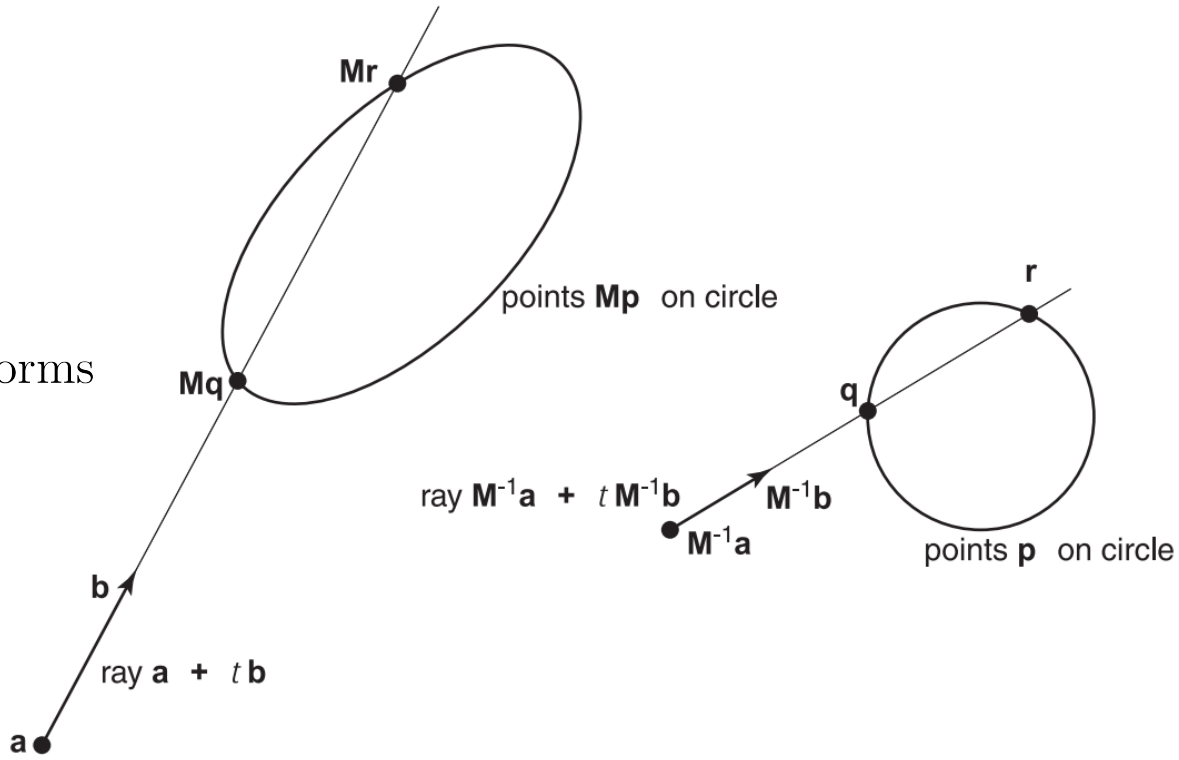
We need to intersect the transformed object with a ray r.

Instead we intersect the untransformed object with an *inverse transformed ray*.

Advantages:

- Simpler intersection routine

- Many objects may be transforms of the same simple object

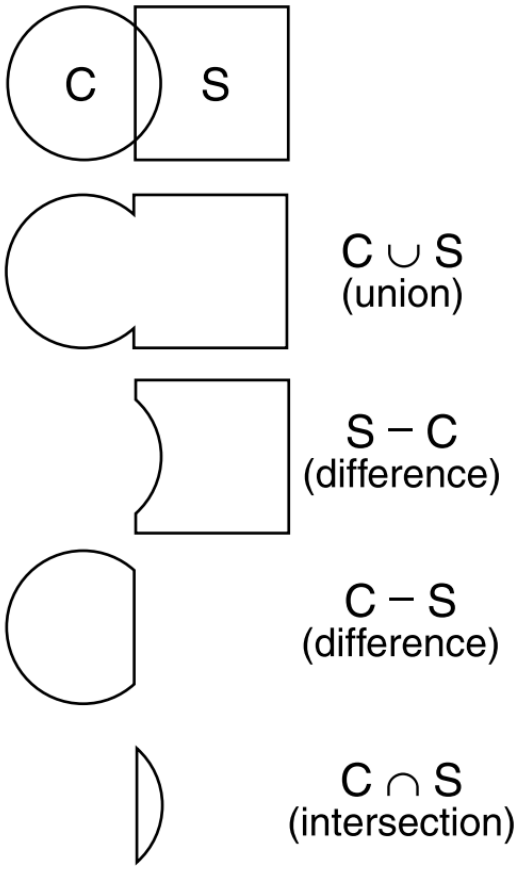


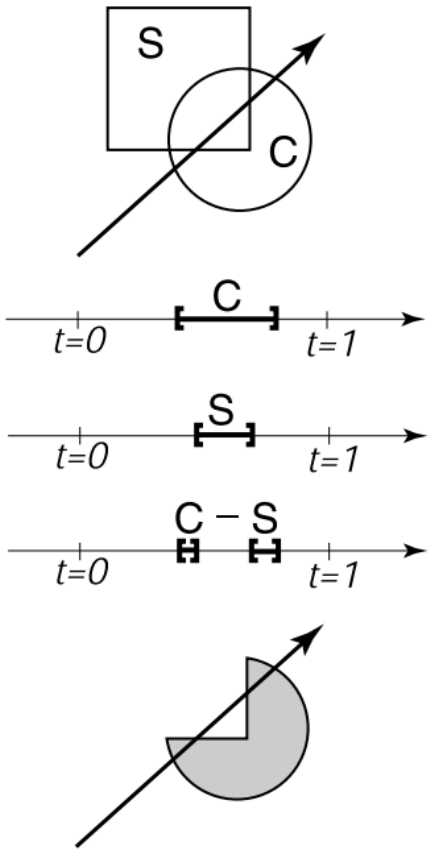
Constructive Solid Geometry

Use set operations to combine solid shapes.

We do not explicitly change the model.

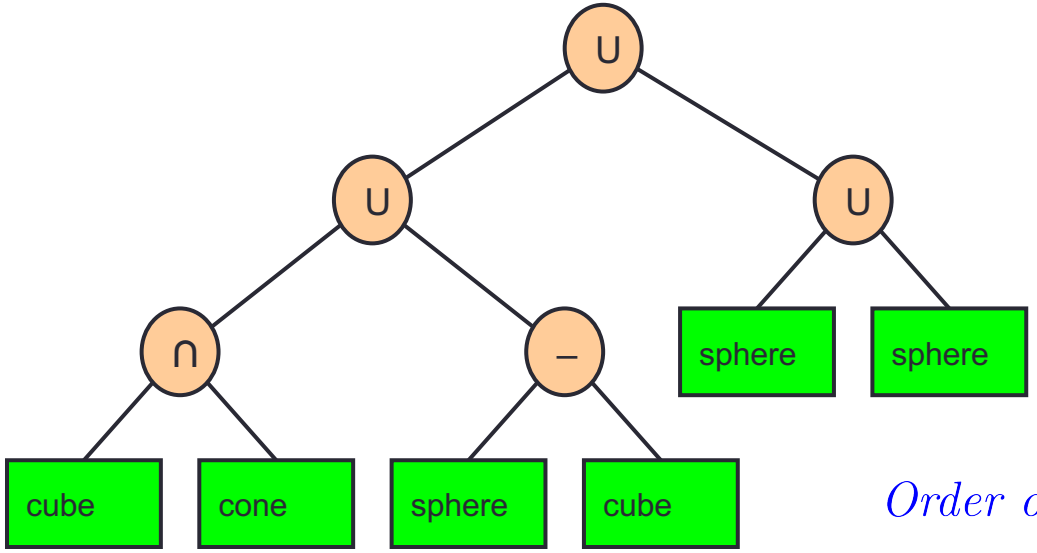
We do set operations on the intersections of a ray with the model.





Constructive Solid Geometry

An object can be represented using a tree in which the interior nodes denote set operations:

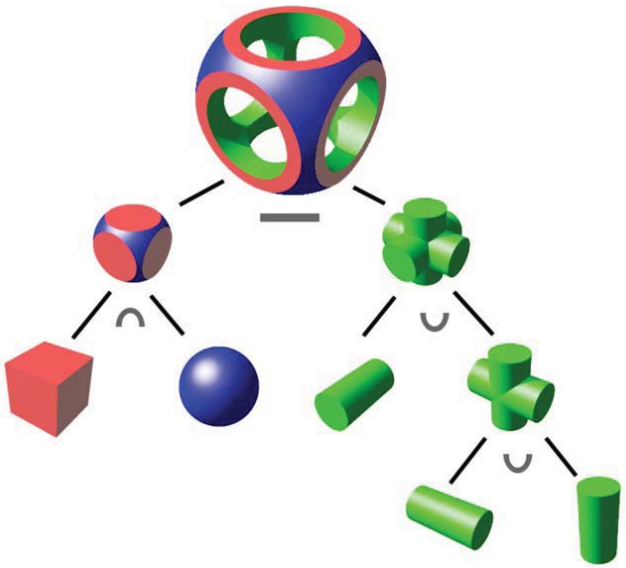


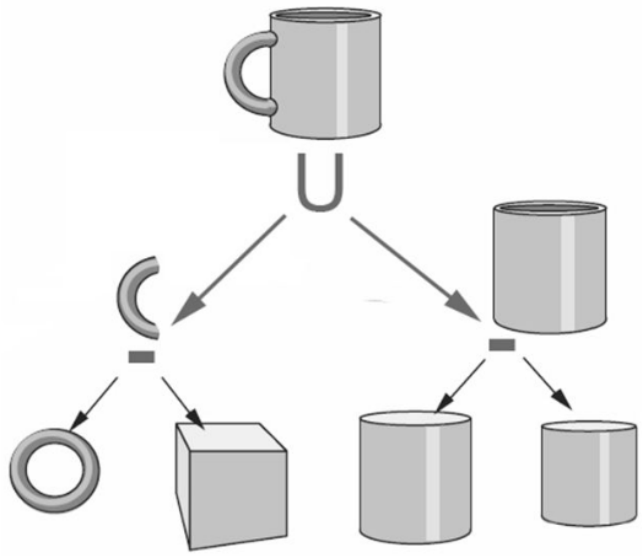
Order of children matters.

We compute the intervals for the leaves and do the set operations given by the tree on these intervals.

How do we compute the normal vector at a point on the object ?

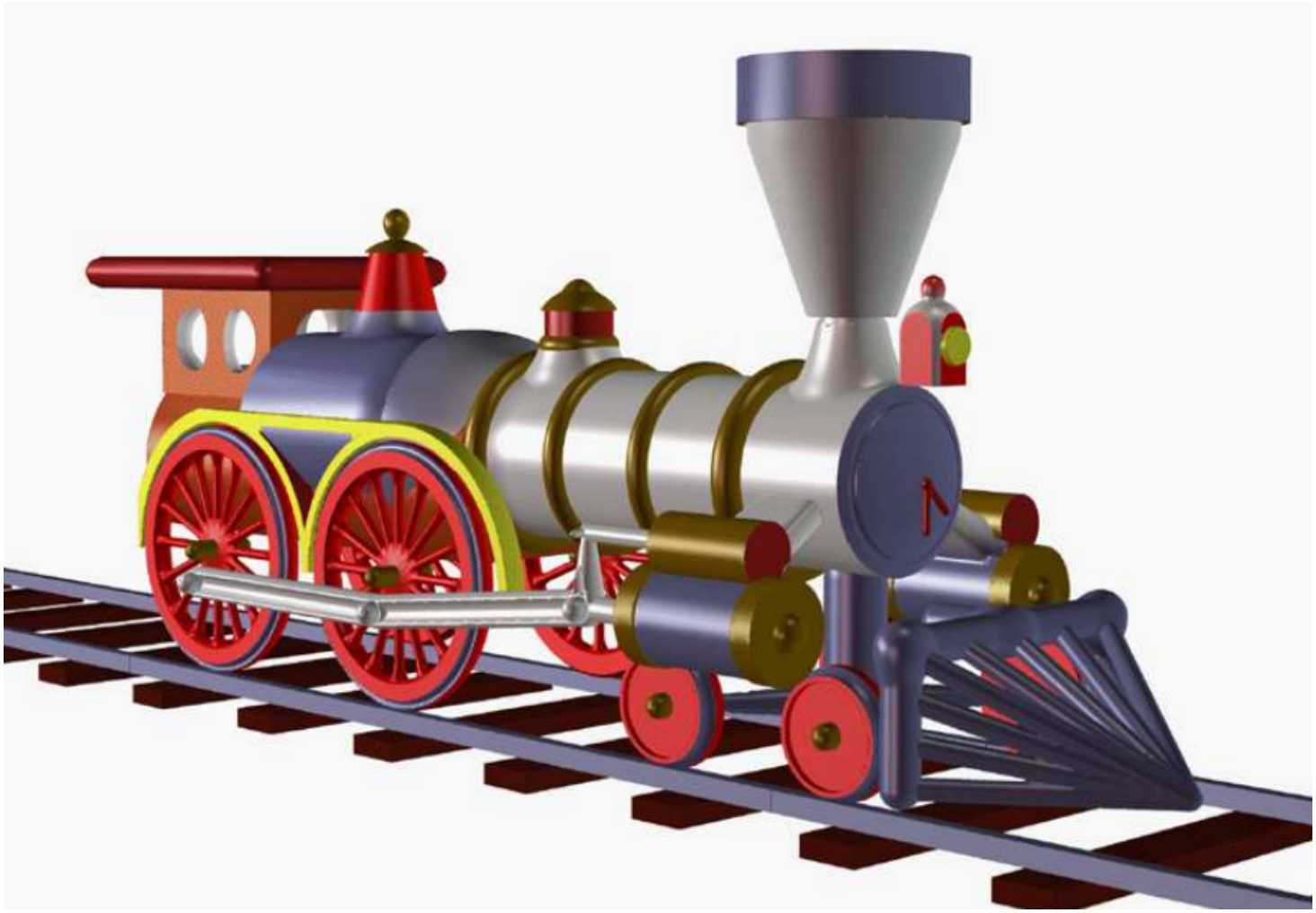
Examples of CSG Trees:





Constructive Solid Geometry

Example CSG Model:



Spatial Data Structures

Goal: to quickly figure out which objects a ray intersects.

Checking all objects for intersection is wasteful.

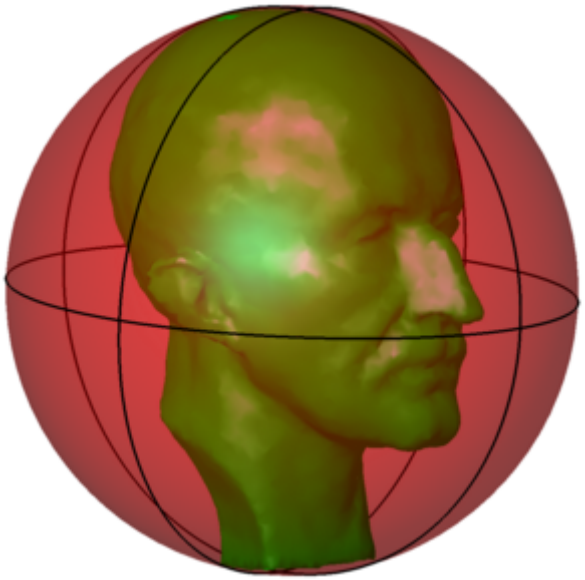
Object Partitioning Schemes

- *divide objects into disjoint groups*
- *these groups may overlap in space*

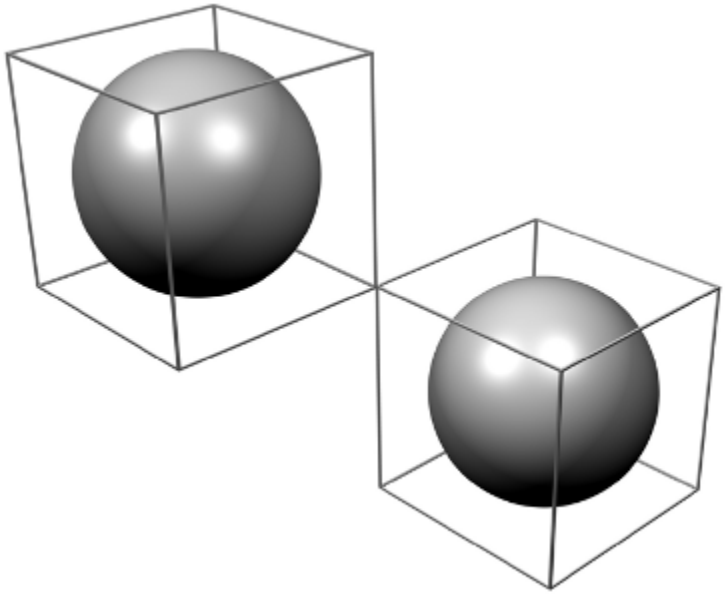
Space Partitioning Schemes

- *space is partitioned*
- *an object may intersect multiple regions*

Spatial Data Structures



Bounding Sphere



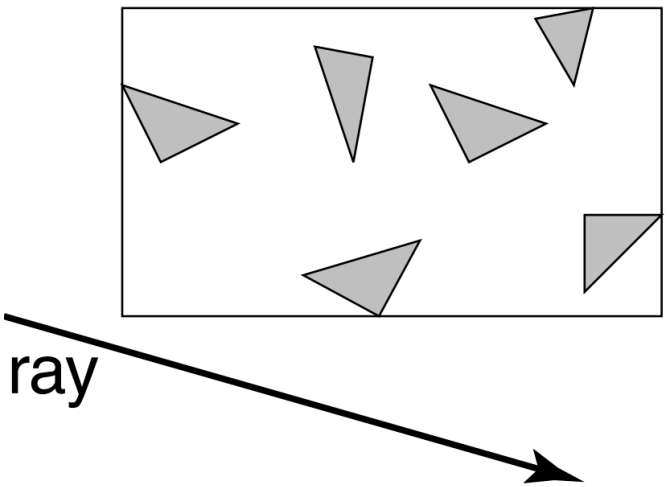
Bounding Boxes

Idea: Before checking intersection with the object, check intersection with the bounding box.

Other kinds of bounding volumes like ellipsoid or polytopes can also be used.

We generally try to find a close fit while keeping the bounding volume simple.

Bounding Boxes



First check whether the ray intersects the bounding box.

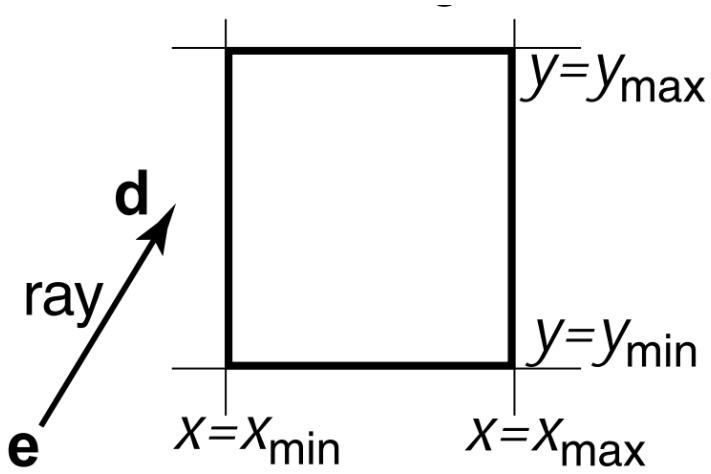
If it does, check for intersection with objects inside.

There could be false positives.

2D Situtation:

We only need to figure out whether the ray intersects the box.

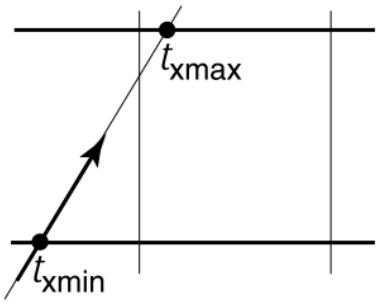
We don't need to know where it intersects the box.

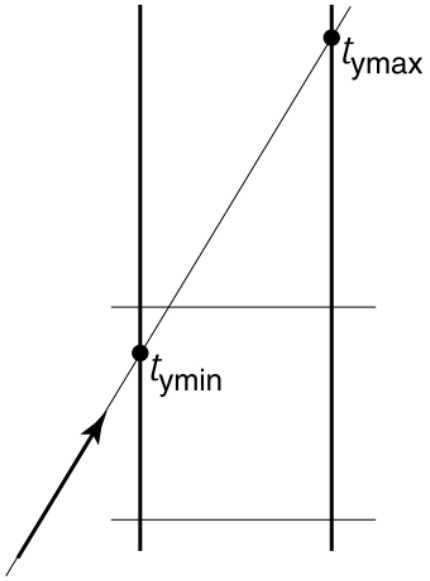


Ray: $\mathbf{e} + t\mathbf{d}$

Let $\mathbf{e} = (x_e, y_e, z_e)$ and $\mathbf{d} = (x_d, y_d, z_d)$.

$$t_{\text{xmin}} = \frac{x_{\text{min}} - x_e}{x_d}$$





similar computations for

t_{xmax}, t_{ymin}, and t_{ymax}

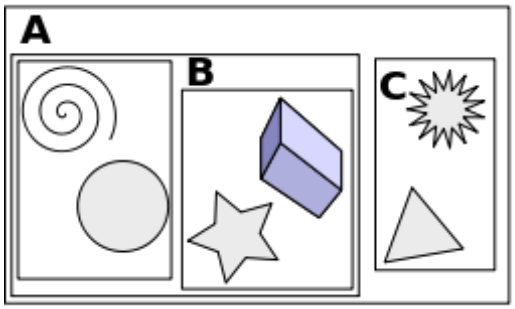
$t \in [\, t_{\text{xmin}}, t_{\text{xmax}} \,]$

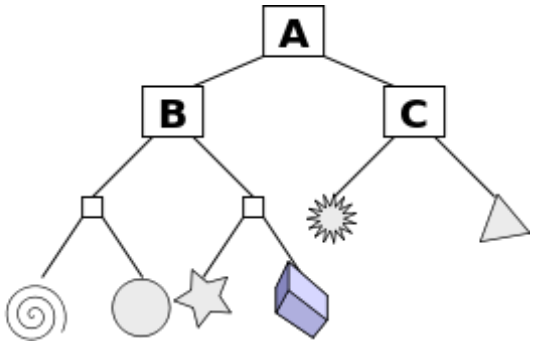
$t \in [\, t_{\text{ymin}}, t_{\text{ymax}} \,]$

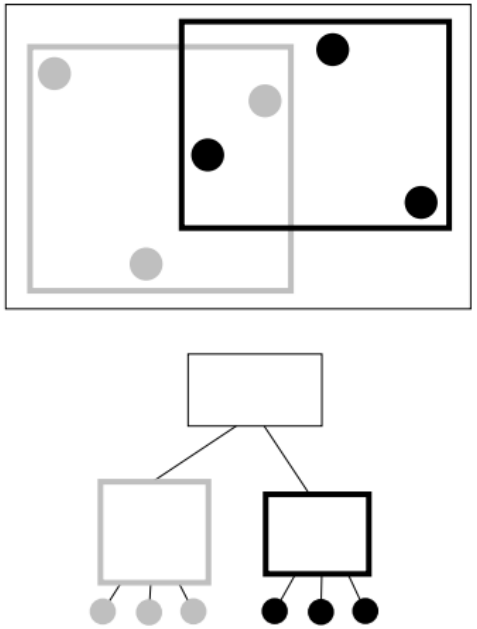
$t \in [\, t_{\text{xmin}}, t_{\text{xmax}} \,] \cap [\, t_{\text{ymin}}, t_{\text{ymax}} \,]$

Hierarchical Bounding Boxes

(a.k.a. Bounding Volume Hierarchies)





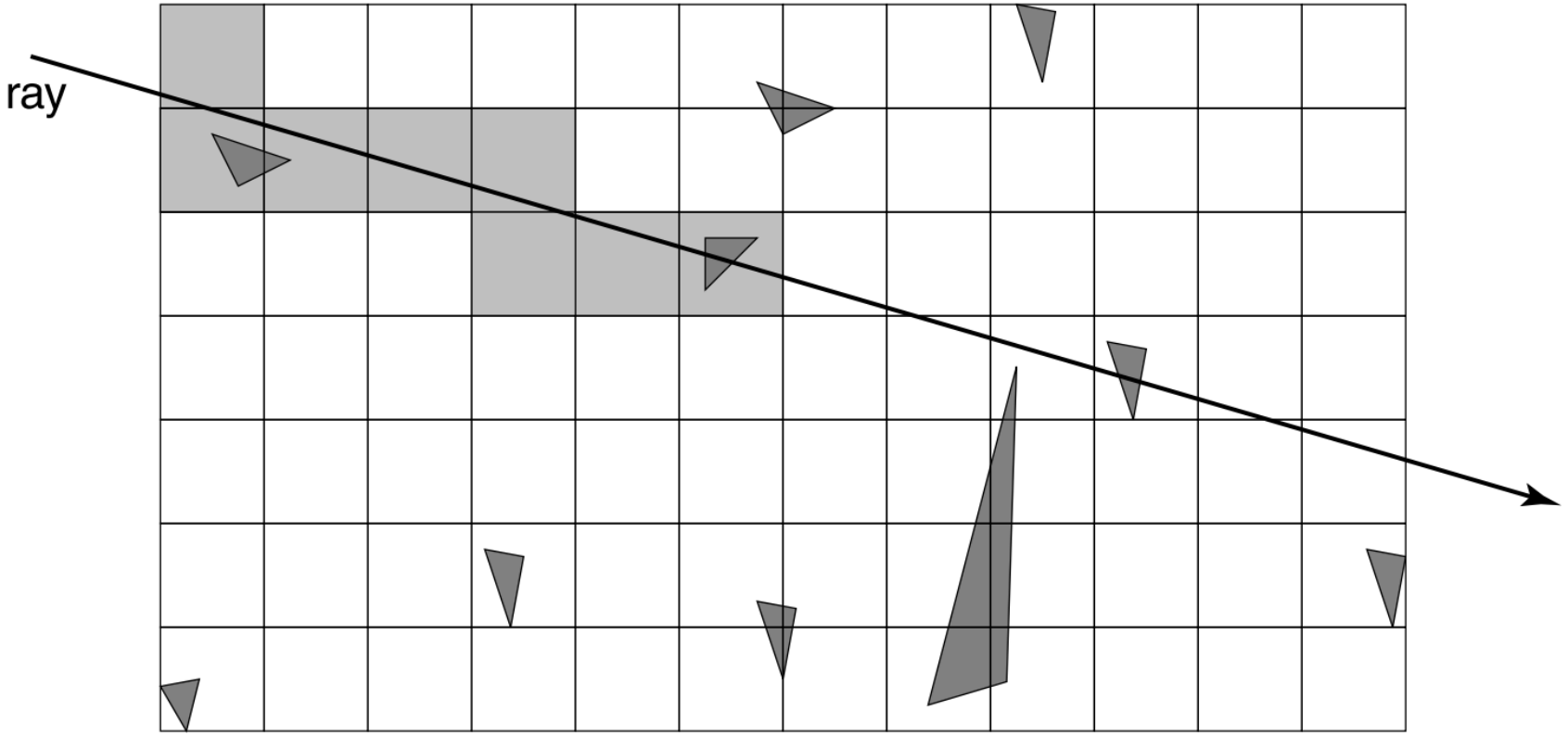


Note: There is no geometric ordering. A ray may hit both sub-trees.

There is no clear way to group objects, but we typically try to minimize the total volume of the bounding boxes.

How do you find the closest object hit by a ray?

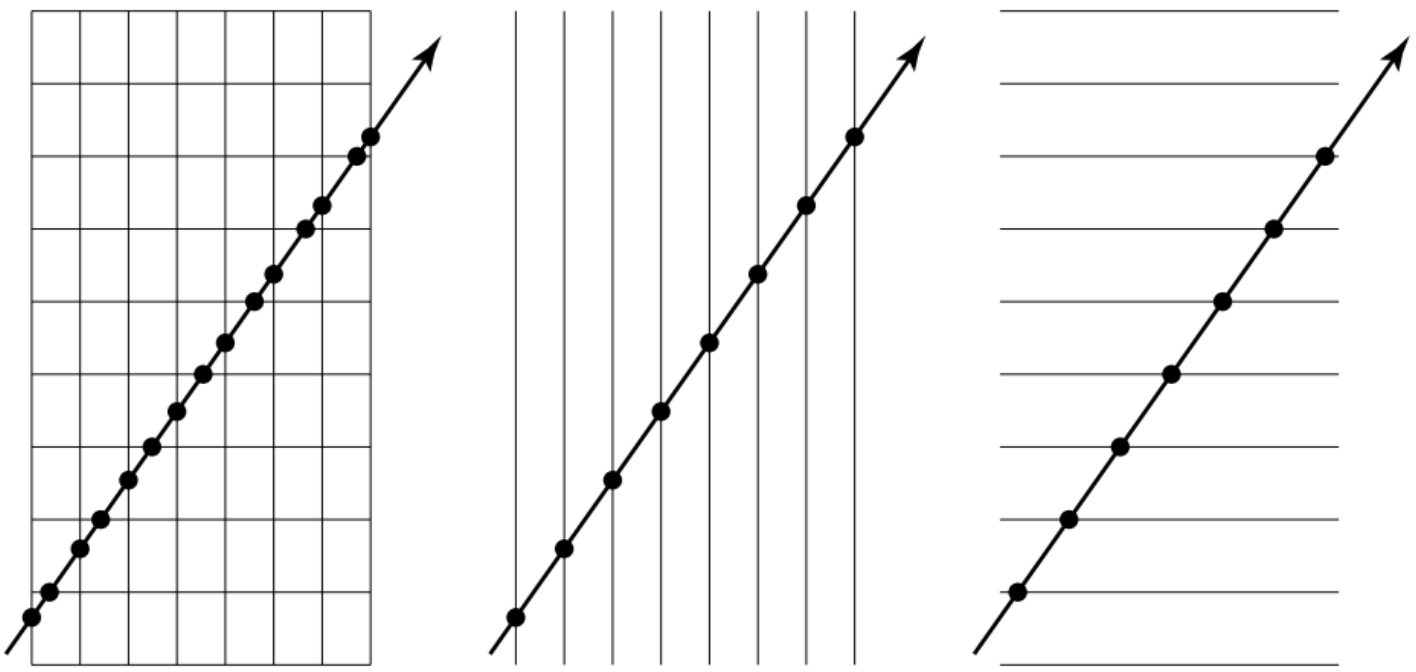
Uniform Spatial Subdivision



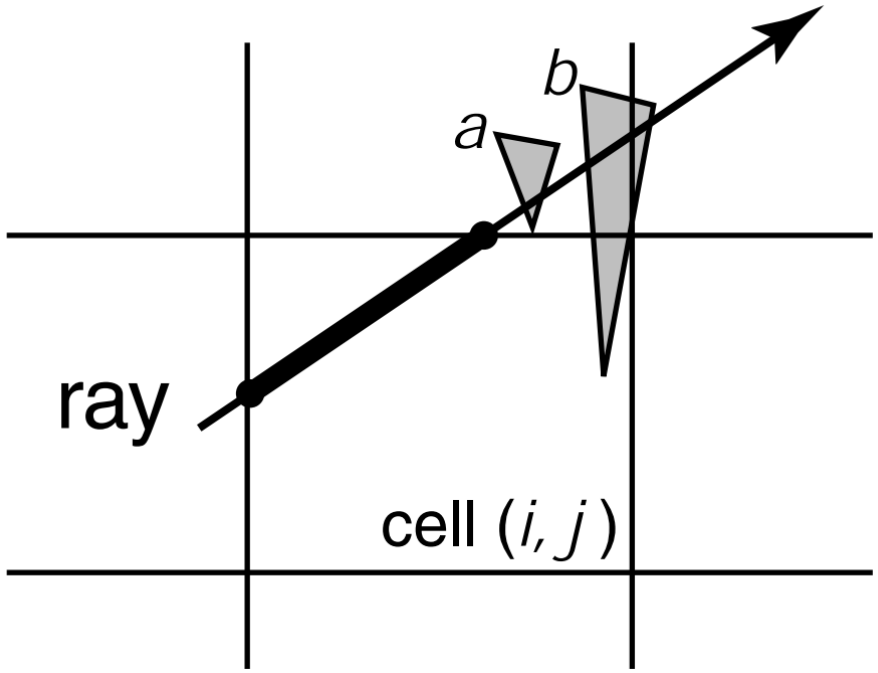
The cells in the grid are of the same size but may not be cubes.

Compute the first cell hit by the ray and then walk along the ray.

Each cell stores the set of objects intersecting the cell.



Hits on the parallel planes are evenly spaced.
This makes it easy to walk along a ray.



Only hits within a cell should be reported.