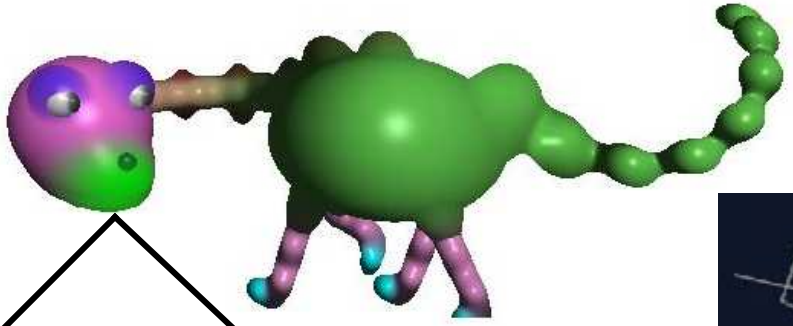
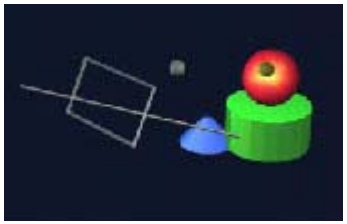
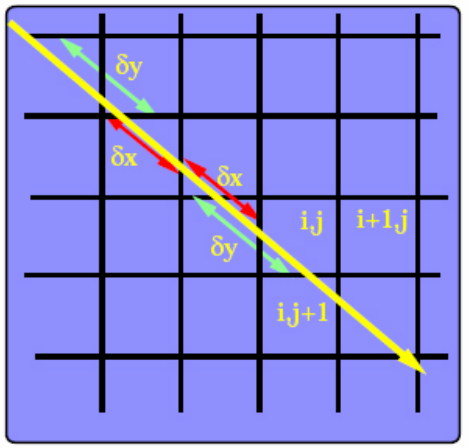


CSC 305
Ray Tracing I

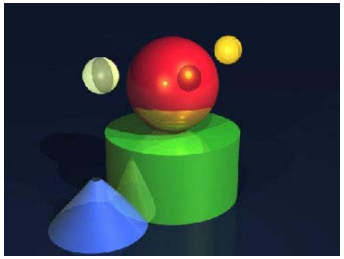
Brian Wyvill

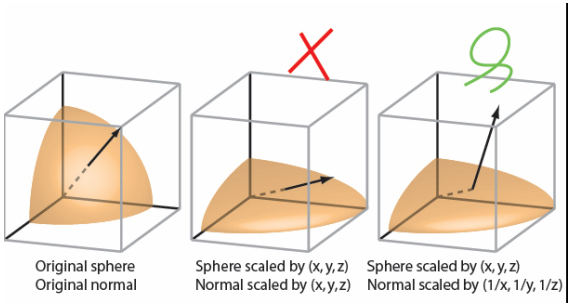


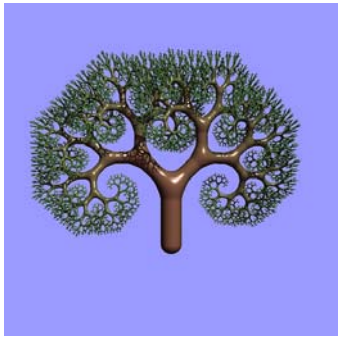


dx is the distance between voxels yz faces

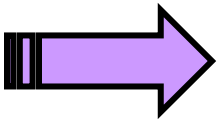
dx and dy record the total distance along the ray.

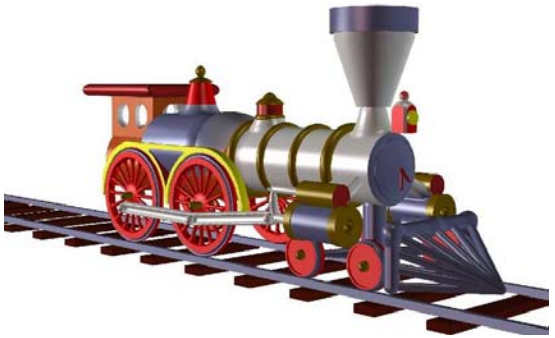




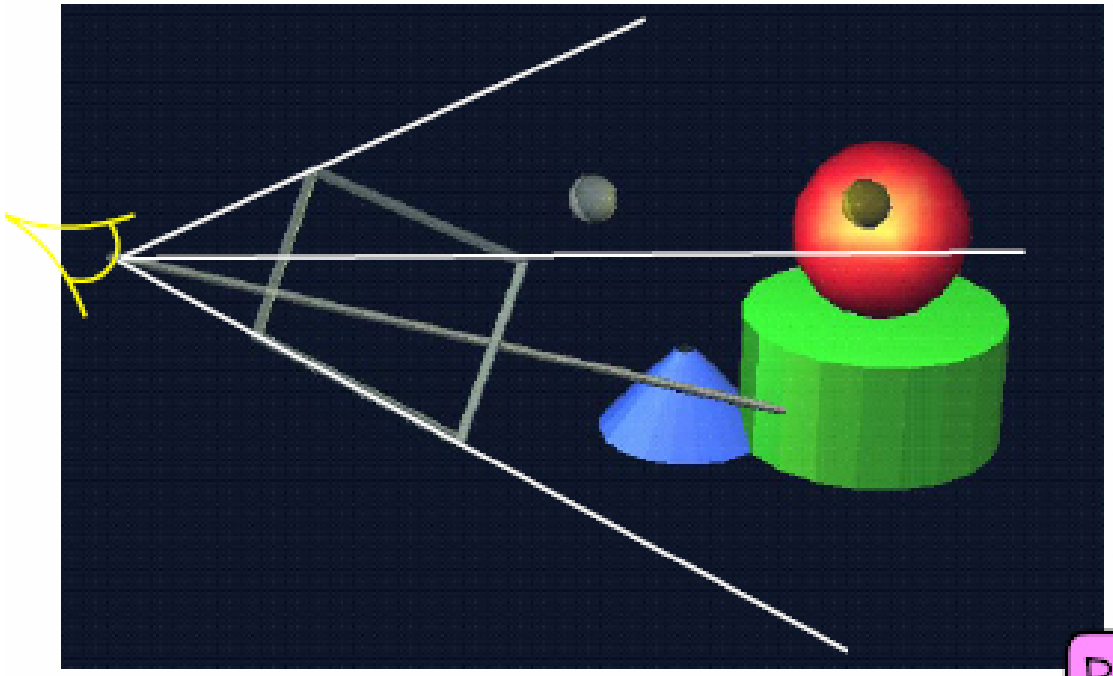


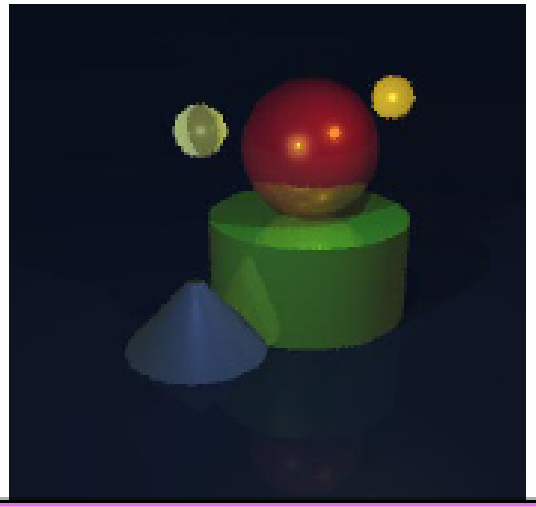
The University of Victoria
Graphics Group





Ray Tracing





Ray Casting

Pinhole camera model.

For every pixel project a ray into the scene. Compare ray with every object. Find nearest intersection. Compute colour of pixel.

Recursive Ray Tracing

Cast rays as before but at each ray-surface intersection spawn off new ray recursively.

Shadow Feeler
Reflected Ray
Refracted Ray

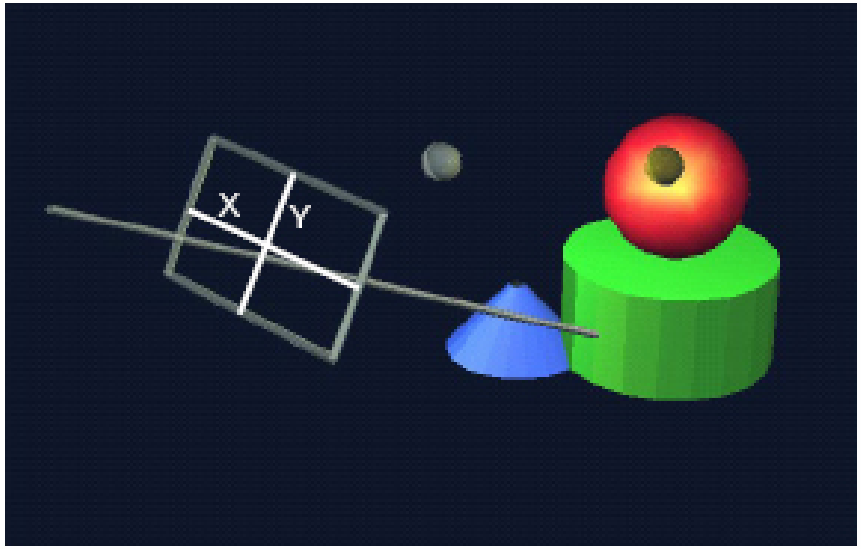
P201 Shirley

A Simplified Ray Tracer

```
Procedure RayTrace()
{
        for y=1 to -1 by 2/(Y-1) do
            for x=1 to -1 by 2/(X-1) do {
                ray.org=[0,0,0];
                ray.dir=[x, y, 1];
                PutPixel(Render(ray));
            }
}

Function Render(ray)
{
        object=QueryScene(ray);
        if (object==NIL) return backgroundColour;
        return Shade(object, ray);
}

Function QueryScene(ray)
{
        closest=NIL;
        distance=INFINITY;
        foreach object in sceneList do {
            if intersect(ray, object)
                if (object.distance<distance) {
                    closest=object;
                    distance=object.distance;
                }
        return closest;
}
```



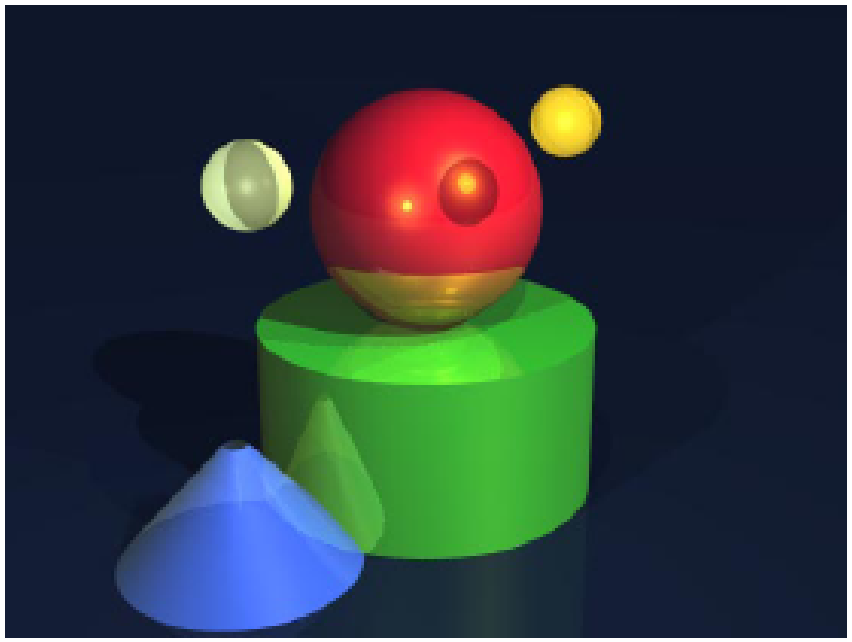
Solves visible surface problem and perspective.

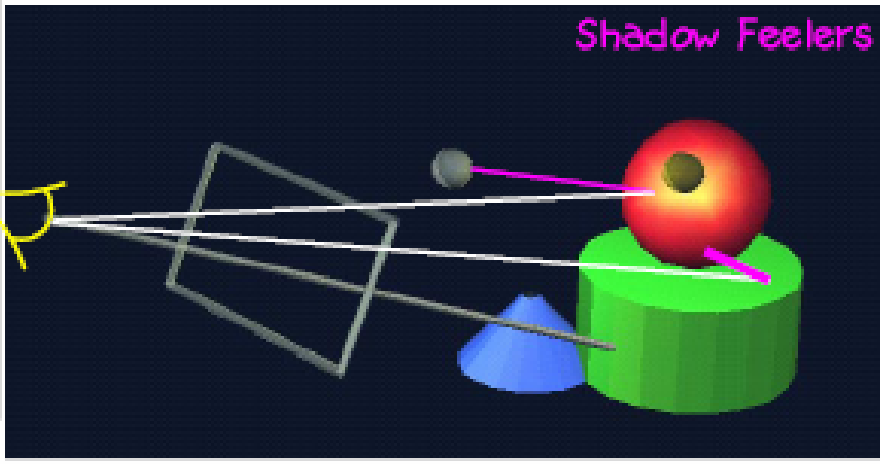
Note this is over simplified and does not take into account the field of view or the fact that rays need to be related to the integer pixel grid. (See notes on sampling).

A Recursive Ray Tracer

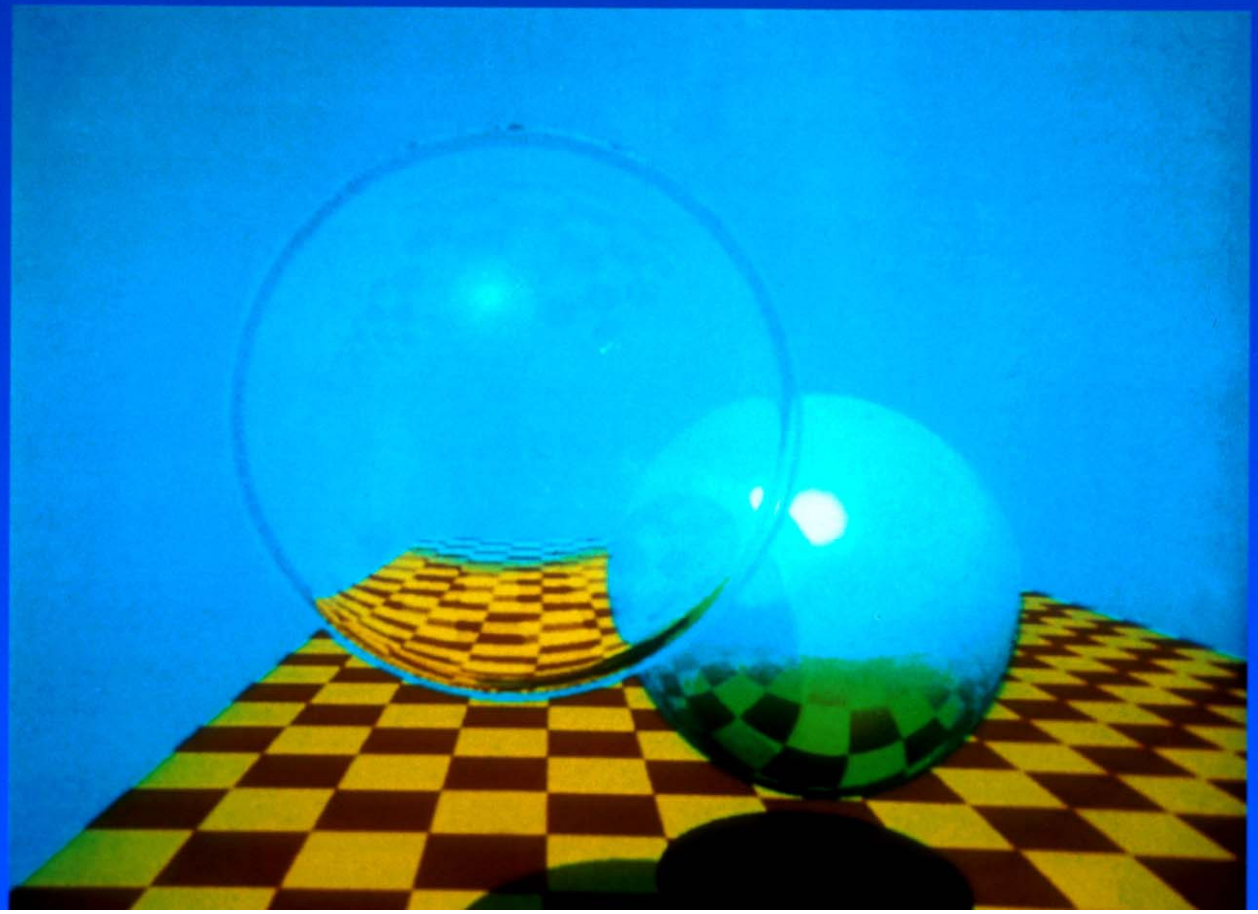
```
Function Render(ray)
{
        object=QueryScene(ray);
        if (object==NIL) return backgroundColour;
        intensity=Shade(object, ray);
        if (object.reflect>0.0) intensity=intensity+
          object.reflect*Render(Reflect(object, ray));
        if (object.refract>0.0) intensity=intensity+
          object.refract*Render(Refract(object, ray));
        return intensity;
}

Function Shade(object, ray)
{
        intensity=AmbientIntensity;
        lightRay.org=object.intersect;
        foreach light in LightList do {
              lightRay.dir=light.dir;
              if (QueryScene(lightRay)=NIL))
                    intensity=intensity+RefFunc(ray, object, light);
        }
        return intensity;
}
```

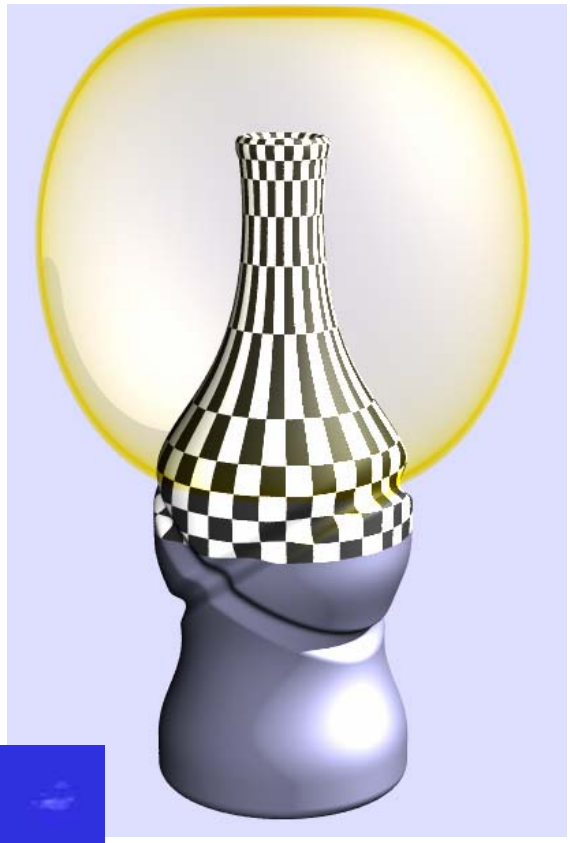


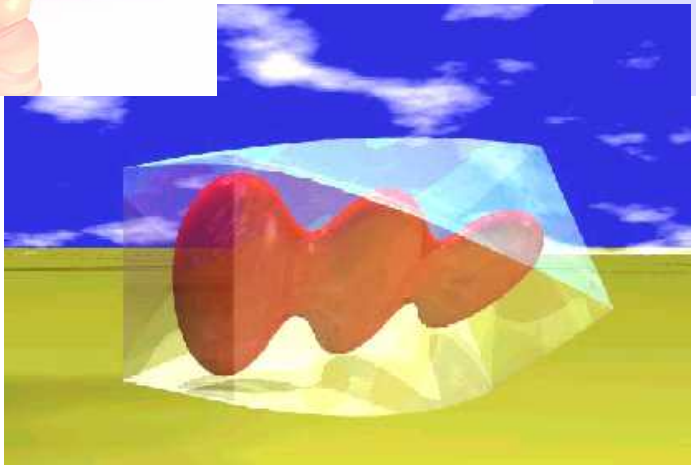


Ray Tracing
Turner Whitted, 1979



Refraction



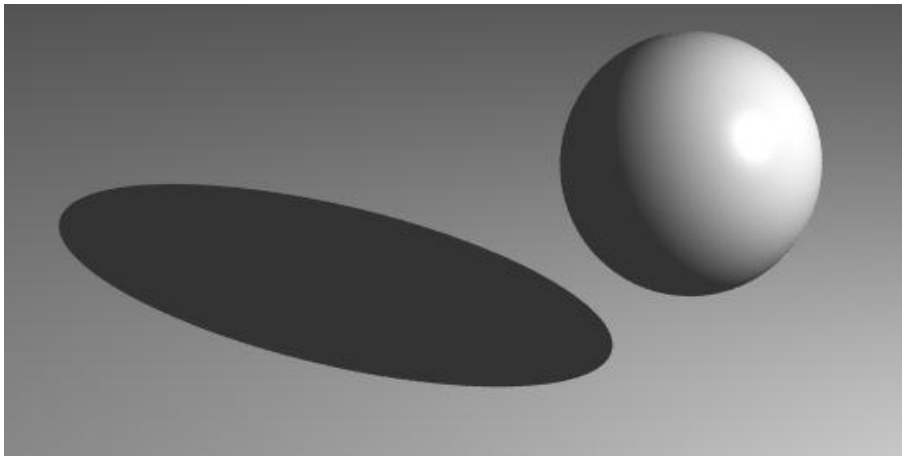


Reflections and Transparency

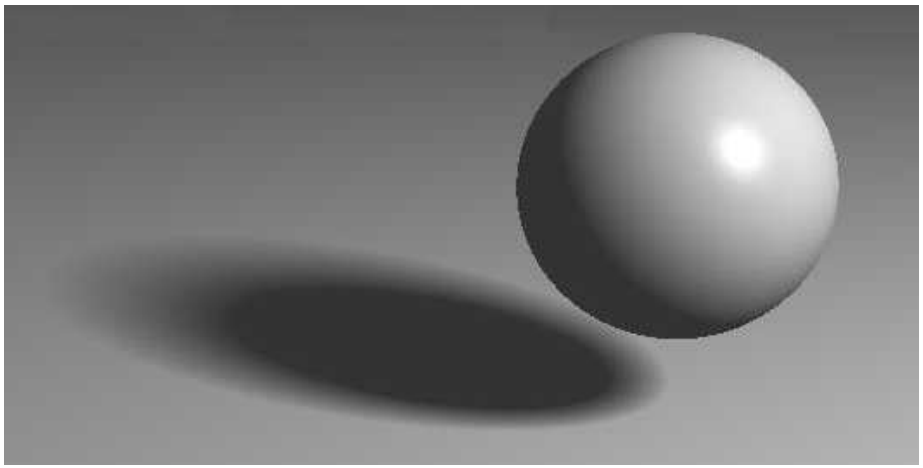
Ray Traced Shadows

Ray Traced Images

Sphere with hard shadow
point sampling source

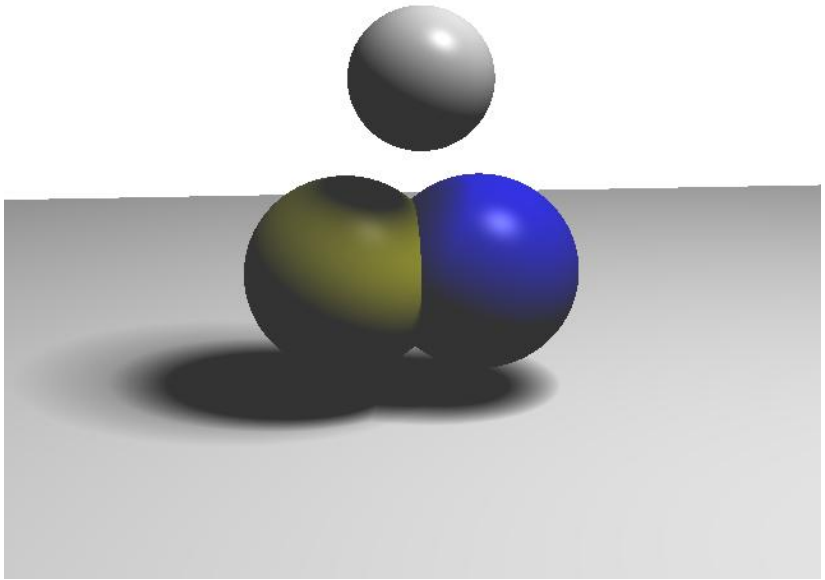


Sphere with soft shadow
sampling wide light source



Shadows
(hard edged)

Shadows (soft edged)



Realism

depth of field

Ray – sphere intersection test

We know: ray origin E
ray direction **V**
O, r for the sphere.
Find P.

$$e^2 + b^2 = c^2$$

$$d^2 + b^2 = r^2$$

$$d = \sqrt{r^2 - (c^2 - e^2)}$$

$$disc = r^2 - (c^2 - e^2)$$

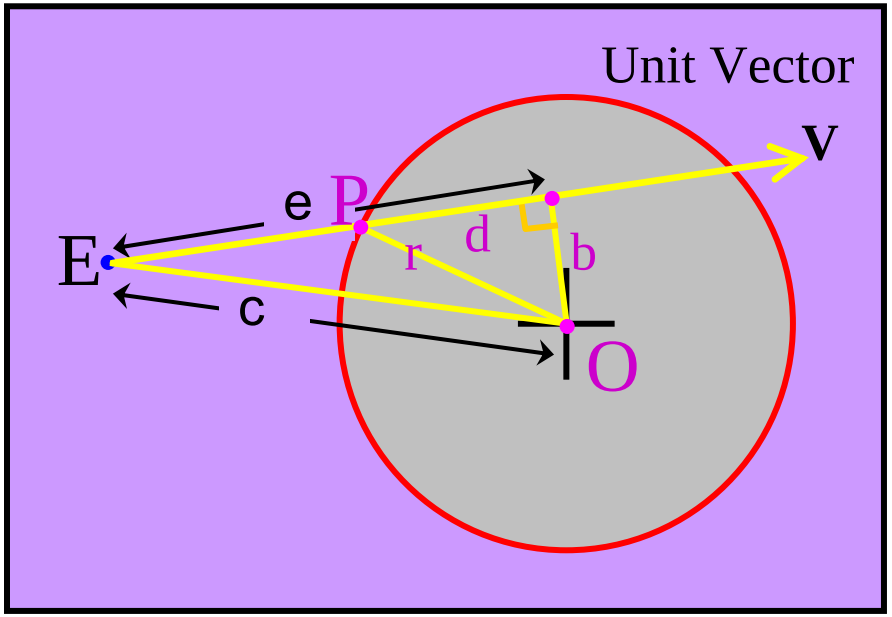
$$e = EO \bullet V$$

if (disc<=0) then no intersection;
else {

$$d = \sqrt{disc};$$

$$P = E + (e - d)\mathbf{V};$$

}



V is the unit vector in the direction of the ray origin E.

Test is designed to quickly eliminate the rays that miss the sphere.

Intersecting A Ray With A Triangle

(See Shirley for better method)

Normal to the plane

$$\mathbf{n} = (\mathbf{b}-\mathbf{a}) \times (\mathbf{b}-\mathbf{c})$$

For a point p in the plane:

$$\mathbf{n}.(\mathbf{p}-\mathbf{b}) = (\mathbf{b}-\mathbf{a}) \times (\mathbf{b}-\mathbf{c}).(\mathbf{p}-\mathbf{b})=0$$

(i.e the angle between the plane and the normal should be 90)

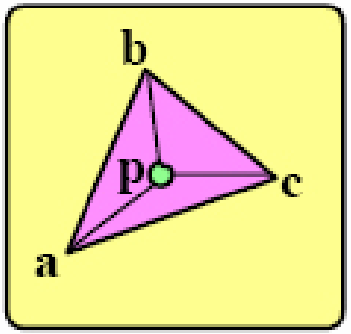
For the intersection tests the ray is presented as an origin plus a ray direction scaled by distance along the ray. The point of intersection is given by:

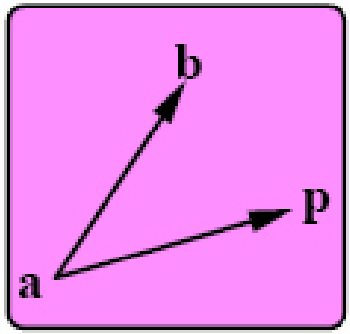
$$\mathbf{p} = \mathbf{u} + \mathbf{v}t$$

substituting this into the plane equation:

$$\mathbf{n}.(\mathbf{u} + \mathbf{v}t-\mathbf{b}) = 0$$

$$t = \mathbf{n}.(\mathbf{b}-\mathbf{u})/\mathbf{n}.\mathbf{v}$$





If $\mathbf{n}.\mathbf{v} = 0$ then ray is parallel to the plane

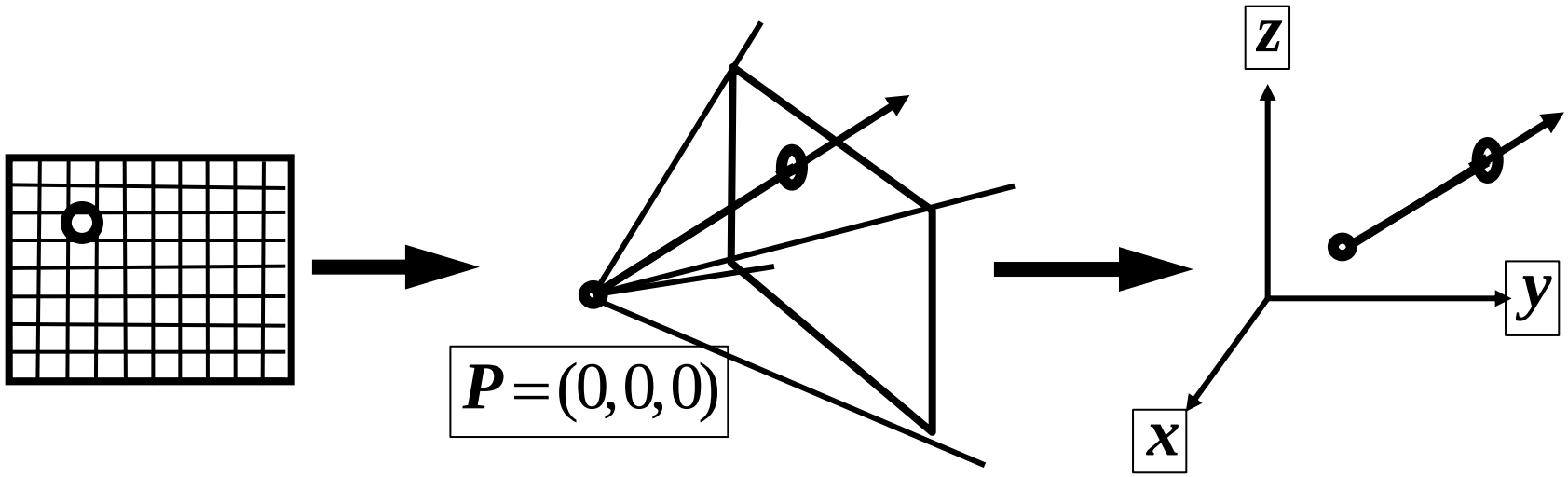
otherwise we can find **p** but still have to test if it is inside the triangle.

This is easily done using a cross product to tell us on which side of each edge the point lies.

In other words the following cross products must have the same sign:

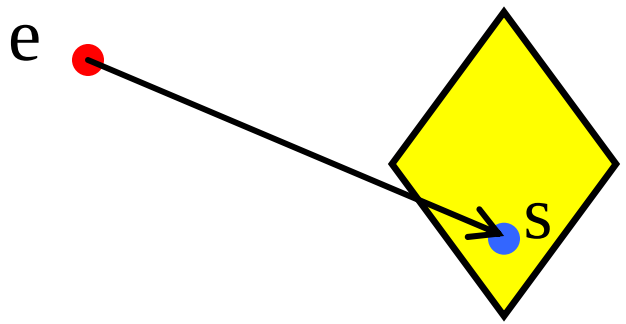
$$(\mathbf{b}-\mathbf{a})\mathbf{x}(\mathbf{p}-\mathbf{a}).\mathbf{n}$$
$$(\mathbf{c}-\mathbf{b})\mathbf{x}(\mathbf{p}-\mathbf{b}).\mathbf{n}$$
$$(\mathbf{a}-\mathbf{c})\mathbf{x}(\mathbf{p}-\mathbf{c}).\mathbf{n}$$

Ray Tracing Basics for Implementation



Screen-space point	Film plane point in canonical view volume	World-space ray

Eye Ray



Ray Equation:

$P(t) = e + t(s - e)$ parametric line equation

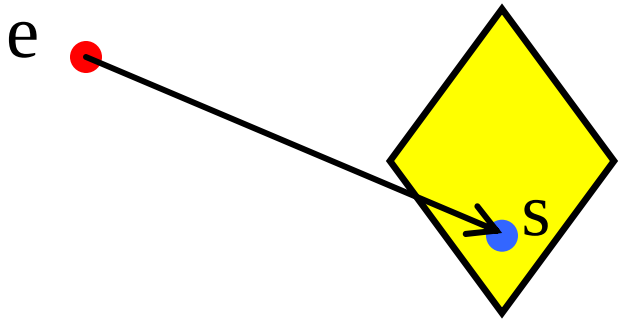
v = s-e p(0)=e p(1)=s

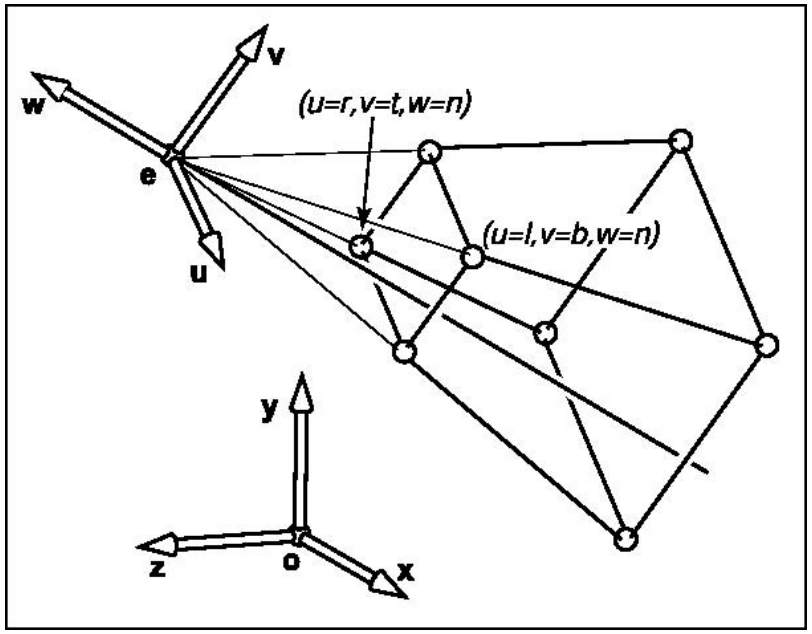
For +(ve) t going from $t_1 < t_2$

then $p(t_1)$ closer to the eye than $p(t_2)$

(-)ve t behind the eye

Computing s



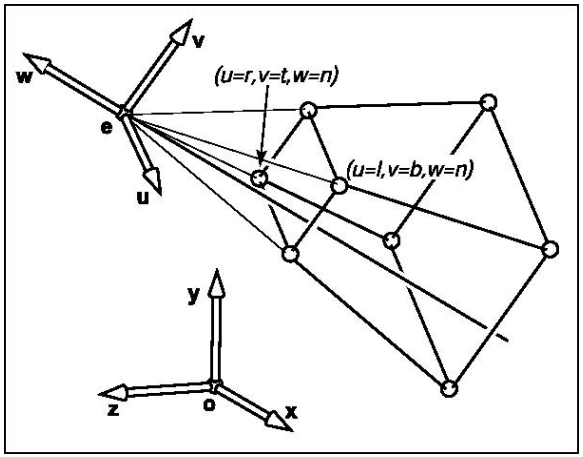


In our viewing coordinate system (**u,v,w**) (w_s=n)
We wish to place the eye at an arbitrary viewing point, e.
A window transform:
[-0.5, n_x-0.5]x[-0.5,n_y-0.5] to [l,r]x[b,t]

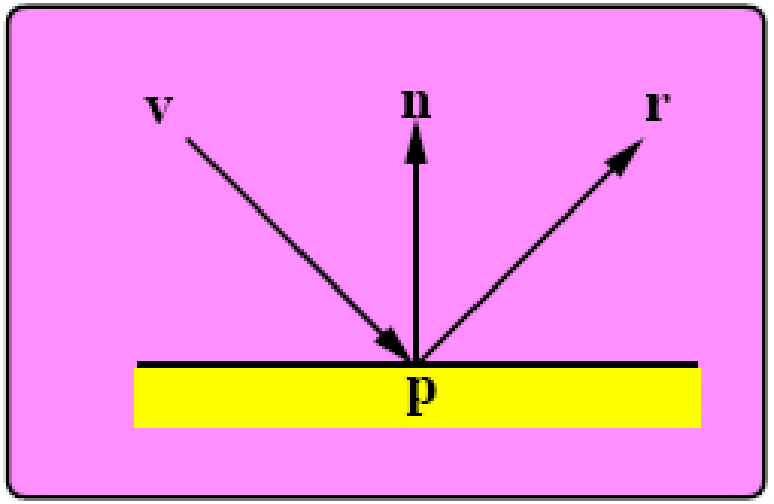
Arbitrary view point

In matrix form:

$$\begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_e \\ 0 & 1 & 0 & y_e \\ 0 & 0 & 1 & z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & x_w & 0 \\ y_u & y_v & y_w & 0 \\ z_u & z_v & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_s \\ v_s \\ w_s \\ 1 \end{bmatrix}$$



Calculating Reflections



The point of intersection along a ray is given by:

$$\mathbf{p} = \mathbf{u} + \mathbf{v}t$$

The ray direction can be split into components parallel to the surface and the normal:

$$\mathbf{v} = \mathbf{v}_n + \mathbf{v}_s$$

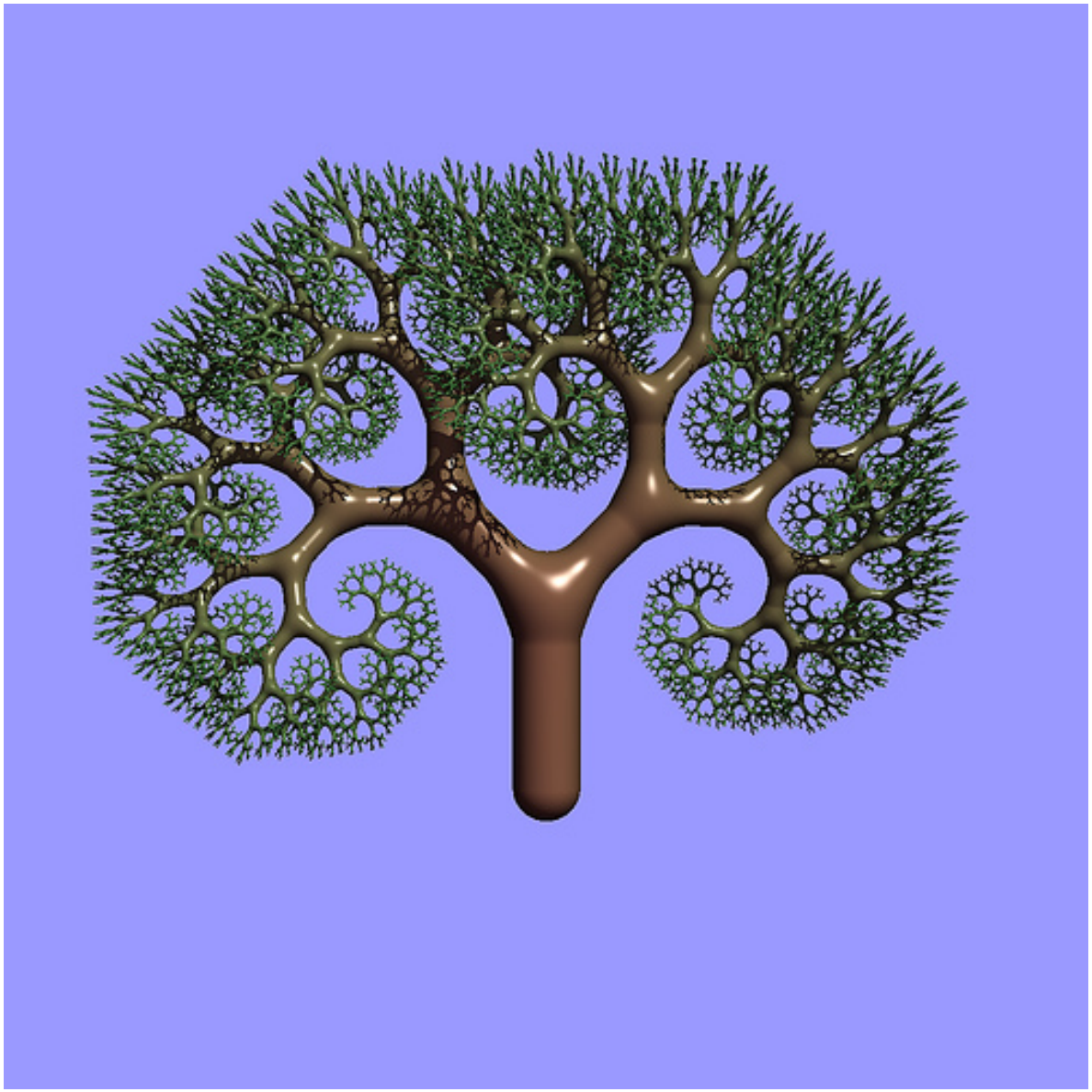
$\mathbf{v}_n$ has magnitude $\mathbf{v}.\mathbf{n}$ and direction $\mathbf{n}$

$$\mathbf{v}_n = (\mathbf{v}.\mathbf{n})\mathbf{n}$$

$\mathbf{r}$ has components $\mathbf{v}_s$ and $-\mathbf{v}_n$

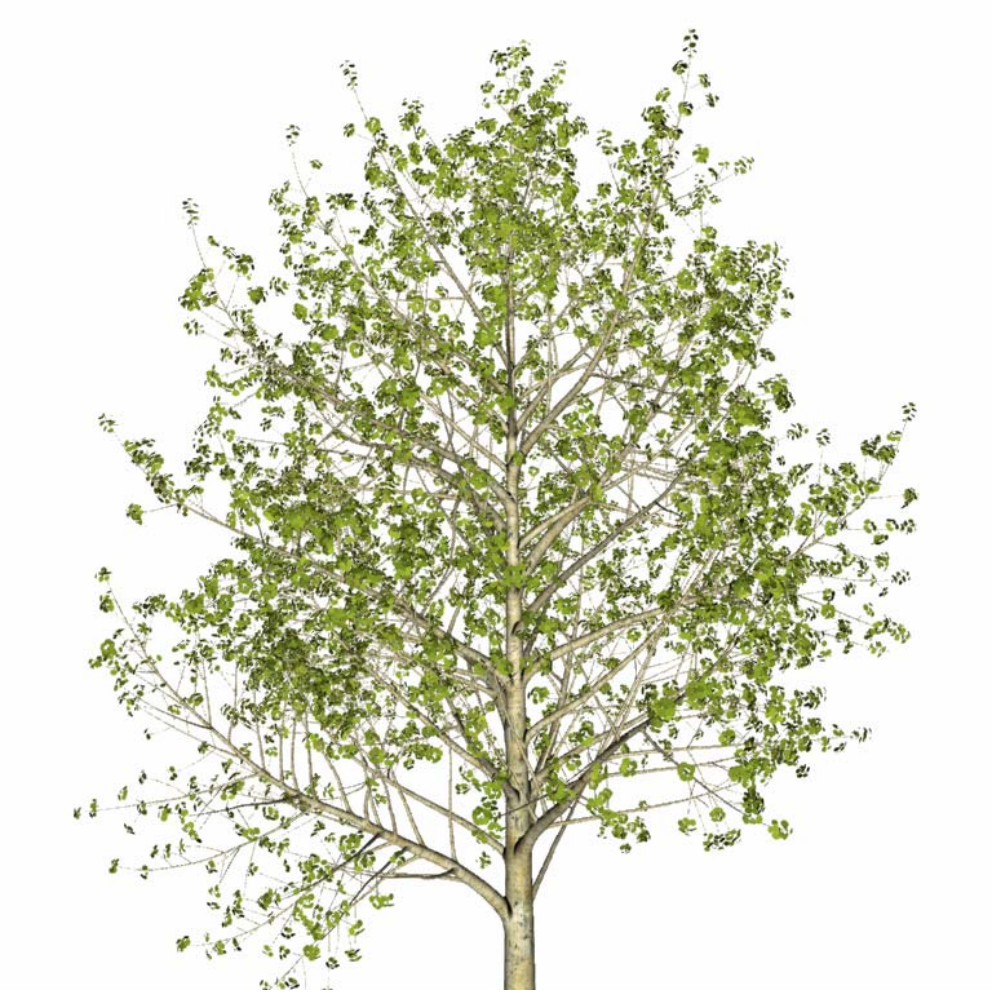
$$\mathbf{v}_s = \mathbf{v} - \mathbf{v}_n$$

$$\mathbf{r} = \mathbf{v}_s - \mathbf{v}_n = \mathbf{v} - 2\mathbf{v}_n = \mathbf{v} - 2(\mathbf{v}.\mathbf{n})\mathbf{n}$$



Level 15
Pythagorus Tree

Need to make
ray tracing faster!



This implicit tree could not be rendered with polygons due to insufficient memory.

It had to be ray traced.

Ray Tracing

- **Flexible, accurate, high-quality rendering**
- **Slow**
- **Simplest ray tracer:**
  - Test every ray against every object in the scene
  - N objects, M rays → O(N * M)
- **Using an *acceleration scheme*:**
  - Acceleration scheme = sub-linear complexity of N
  - Grids and hierarchies
  - N objects, M rays → O(log(N) * M)
  - Log(N) is a theoretical estimate, in reality it depends on the scene
  - Speedups of over 100x for complex scenes are possible

Speeding Up Ray Tracing

In the naive algorithm each ray has to be tested against each object.

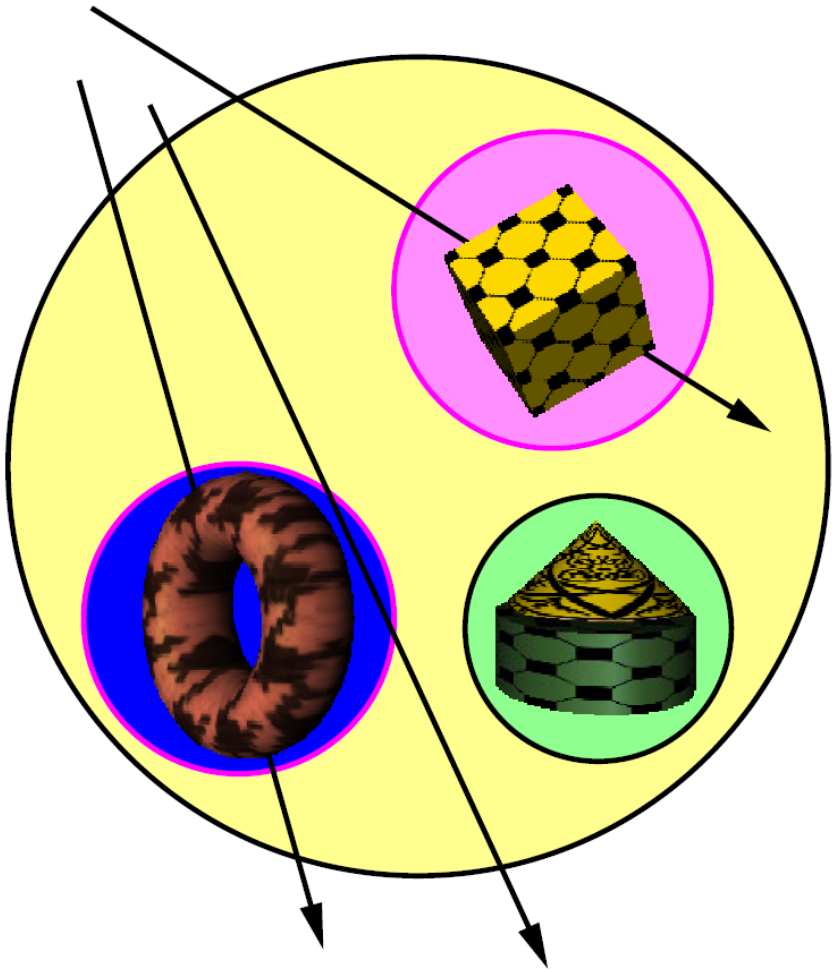
O (m*n) m rays and n objects.

Most rays miss most objects.

Exploit this:

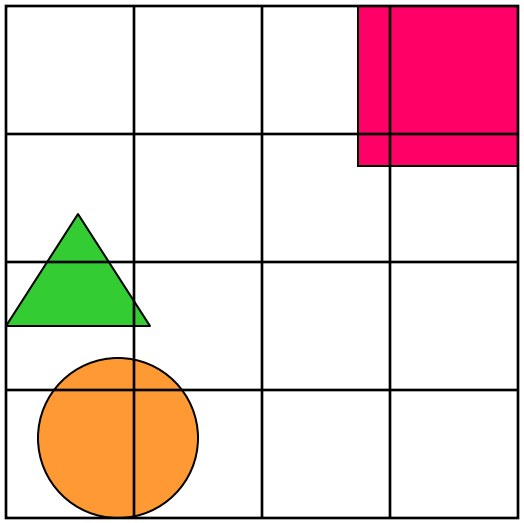
1. Bounding volumes or boxes on hierarchical groups of objects.

2. Space Sub-division.



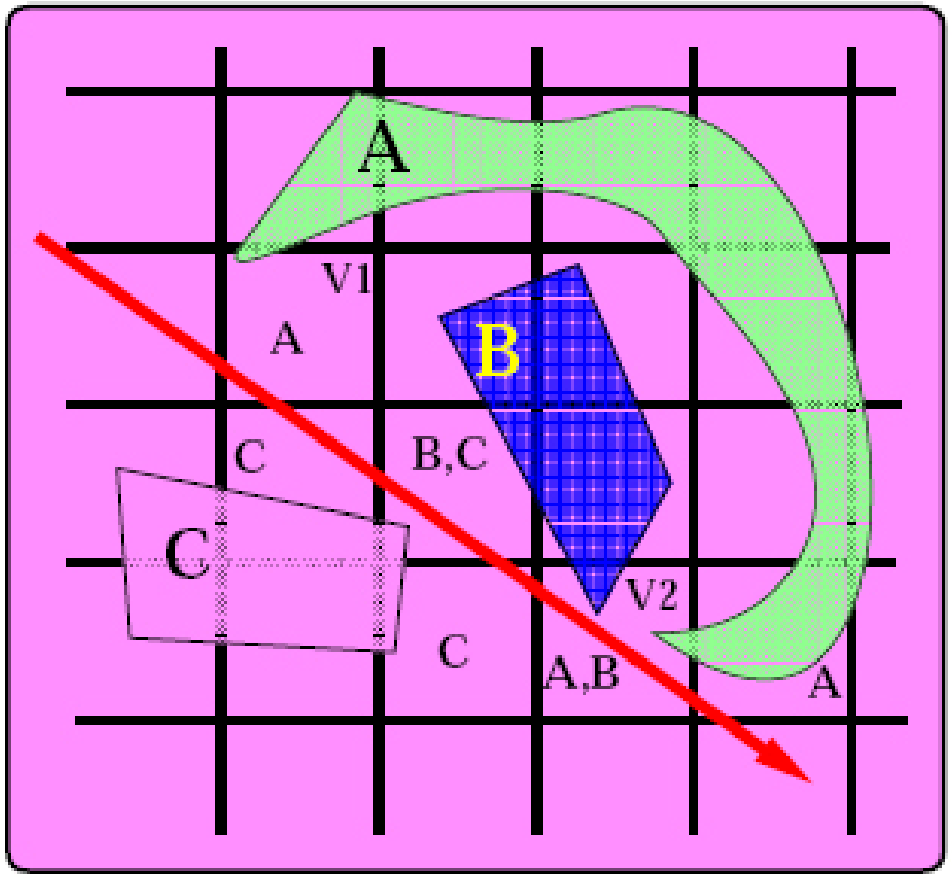
Bounding Spheres

Uniform Grid



- **Ray steps through the grid and is tested against objects in the grid cells along the path of the ray**
- **Can avoid testing the vast majority of the objects for each ray**
- **Grid traversal overhead can negate savings…**

Uniform Grid



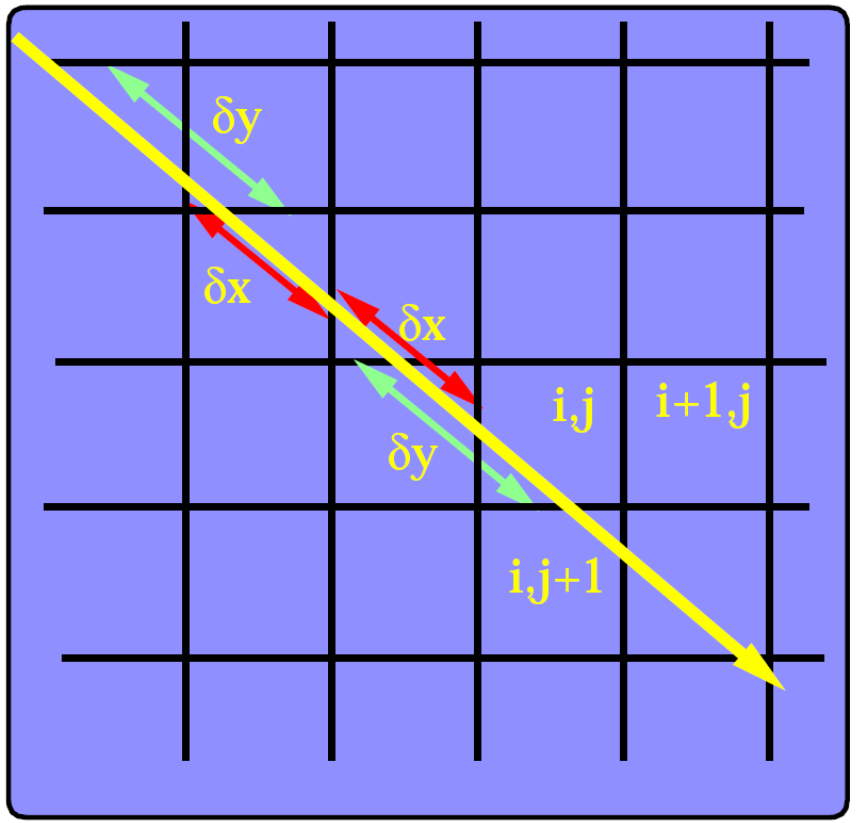
Each ray is checked against each voxel. In the figure the ray intersection for object A is found in voxel v1.

Each ray has a unique number or signature. This is stored with the object so that when the ray is intersected with A in voxel v2 the intersection information is retrieved and the object intersection test is not repeated.

Uniform Grid: problems

- **Grid does not adapt to empty space and local complexity**
  - Works best for uniformly distributed objects (seldom happens in reality)
  - Typical scenes have areas of complex geometry with empty space between them
- **Empty space:**
  - Time is wasted tracing the ray through empty grid cells
- **Local complexity:**
  - Too many objects in each grid cell
  - Could increase grid resolution, but that makes the empty space problem worse
- **Difficult to choose optimal grid resolution that minimizes rendering time: tradeoff between these two problems**
- **Despite this, a grid is still much better than nothing**

Uniform Space Sub-Division Voxel traversal (Cleary et al)



dx is the distance between voxels yz faces

dx and dy record the total distance along the ray.

The Algorithm

dx and dy are the distance from the x and y axis. px,py,pz have values of 1 or -1 depending on direction of ray.

```
repeat
    if dx<=dy {
        i:=i+px
        dx:=dx+δx
    } else {
        j:=j+py
        dy:=dy+δy
    }
Until an intersection is
found in cell i,j
```

Next Voxel Algorithm

Suppose voxels stored as a 3D array n*n*n

voxel[i,j,k] address p = i*n*n + j*n +k

Multiplications in next voxel loop.

But n*n constant

Each time i is incremented p incemented by + or - n^2

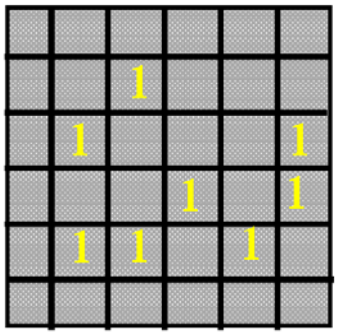
Each time j is incremented p incemented by + or - n

n should be large enough such that most cells are empty

use a hash table instead of 3D array of voxels.
p mod M index into table length M
avoid division by checking p against M at the end of each loop

Termination of Next Voxel Algorithm

Detect when ray leaves bounding volume for scene. Distances to boundary faces are given by sx,sy,sz Compare dx,dy,dz to sx,sy,sz once per loop.



Bit array one bit per voxel indicates if any object overlaps that voxel. Hash table only accessed if bit on.

Voxel Traversal Algorithm in 3D with termination

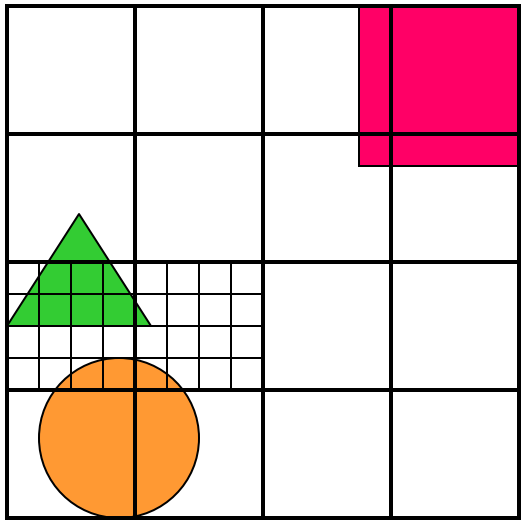
initialize values of px,py,pz, δx,δy,δz,dx,dy,dz and p

```
repeat
        if (dx<=dy) and (dx<=dz) {
                if dx>=sx exit;
                p:=p+px
                dx:=dx+δx
        } else if (dy<=dx) and (dy<=dz) {
                if dy>=sy exit;
                p:=p+py
                dy:=dy+δy
        } else if (dz<=dy) and (dz<=dx) {
                if dz>=sz exit;
                p:=p+pz
                dz:=dz+dz
        }
        if p>M  p:=p−M
until an intersection foundin cell with hash key p
```

Hierarchies

- **Need a scheme that adapts to the distribution of objects in the scene**
- **Build a hierarchy or spatial tree**
- **The scene is recursively subdivided into nodes that enclose space and objects**
  - Empty space is not subdivided
  - Complex areas are subdivided
  - Subdivide until criteria is met: e.g.
    - Number of objects in the node is below a certain threshold (4-8 works well)
    - Tree depth reaches a specified maximum
- **Solves both empty space and local complexity problems**
- **Represented as a tree data structure in memory**
- **Examples…**

Hierarchy of Grids



- **Grid cells/nodes may be empty, contain objects, or contain another grid (e.g. if a cell contains more than 1 object)**
- **An object may span multiple nodes or grid cells**