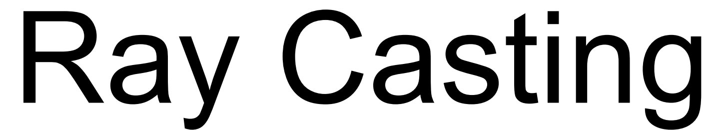


Ray Casting

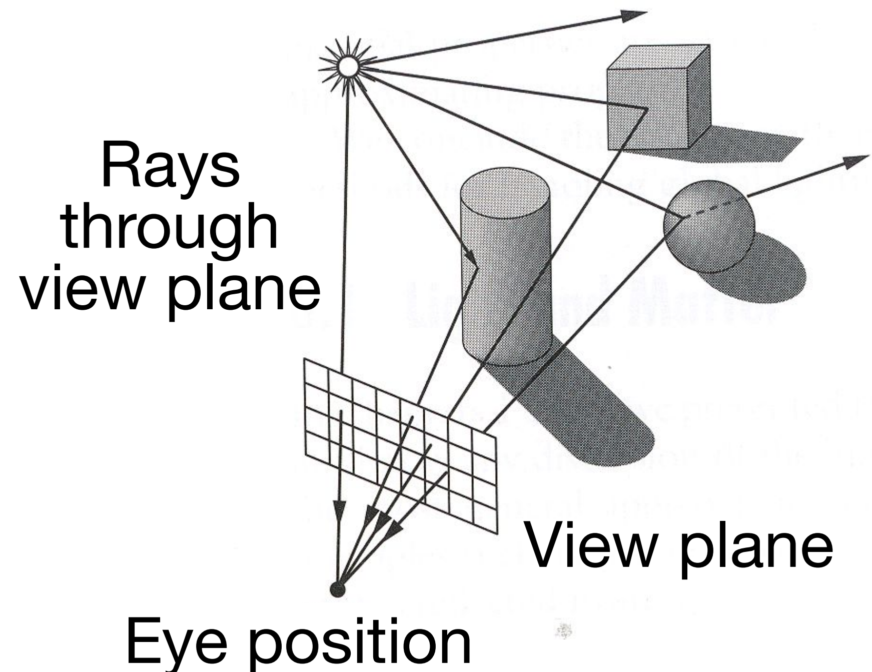
Aaron Bloomfield

CS 445: Introduction to Graphics

Fall 2006

3D Rendering

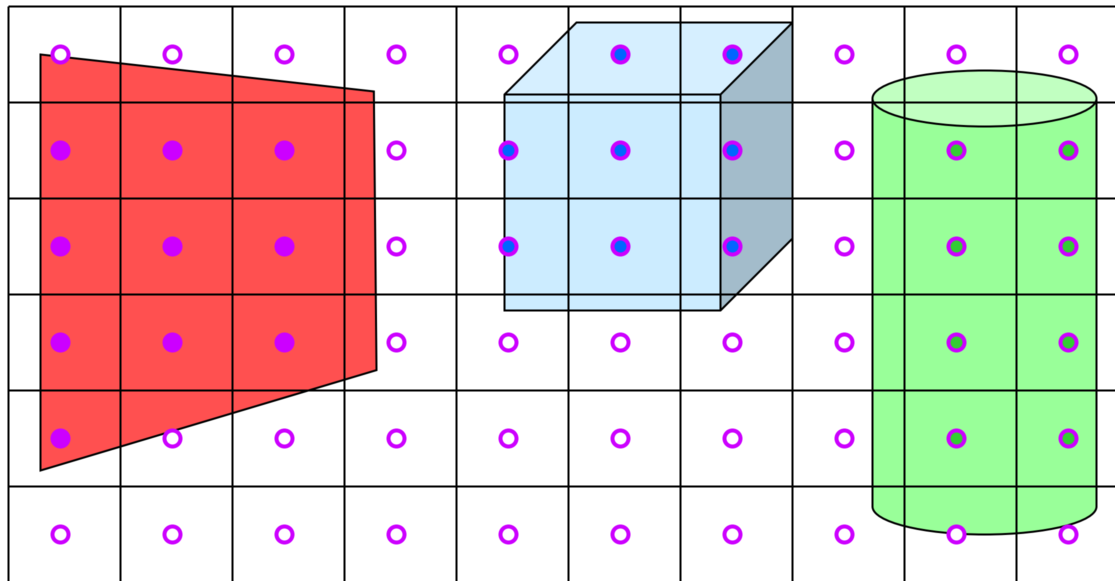
- The color of each pixel on the view plane depends on the radiance emanating from visible surfaces



Simplest method is ray casting

Ray Casting

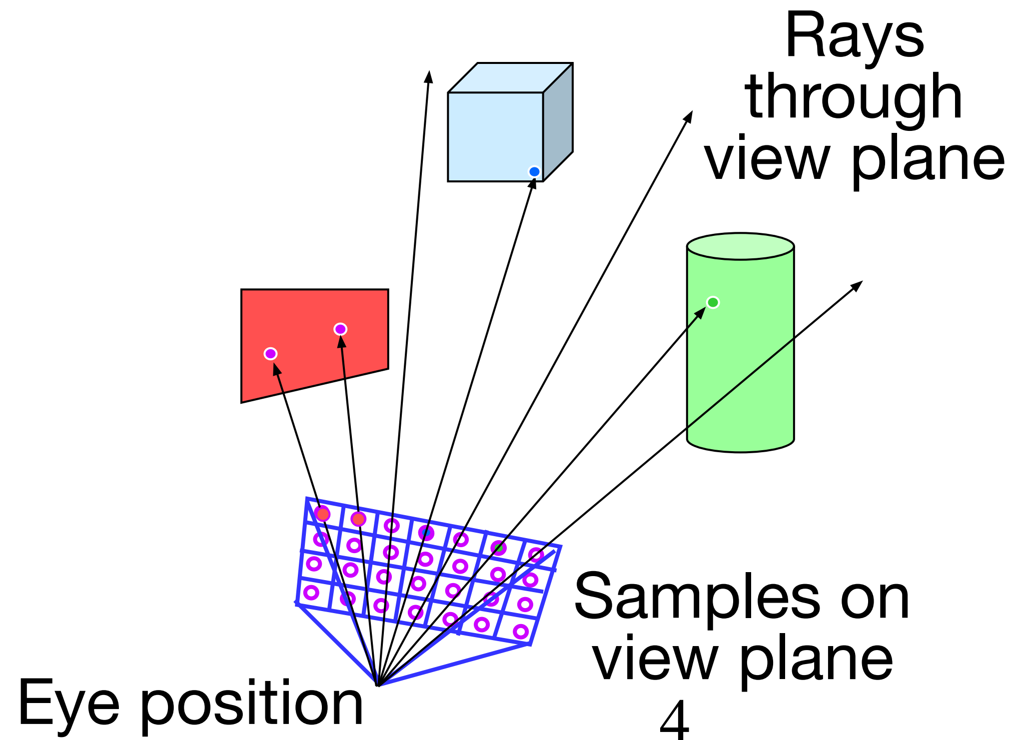
- For each sample …
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute color sample based on surface radiance

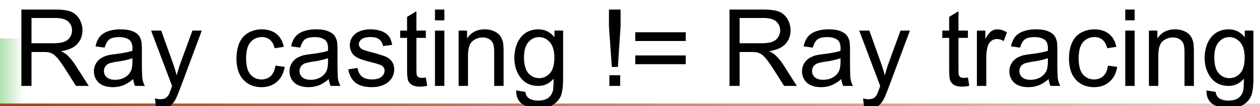


Ray Casting

WHY?

- For each sample …
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute color sample based on surface radiance





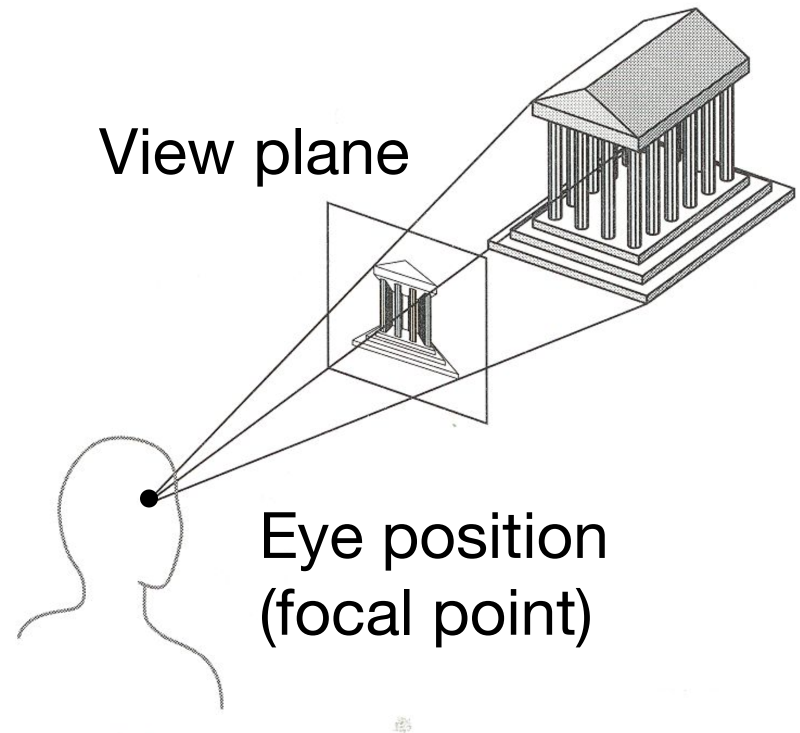
Ray casting != Ray tracing

- Ray casting does not handle reflections
  - These can be “faked” by environment maps
  - This speeds up the algorithm
- Ray tracing does
  - And is thus much slower

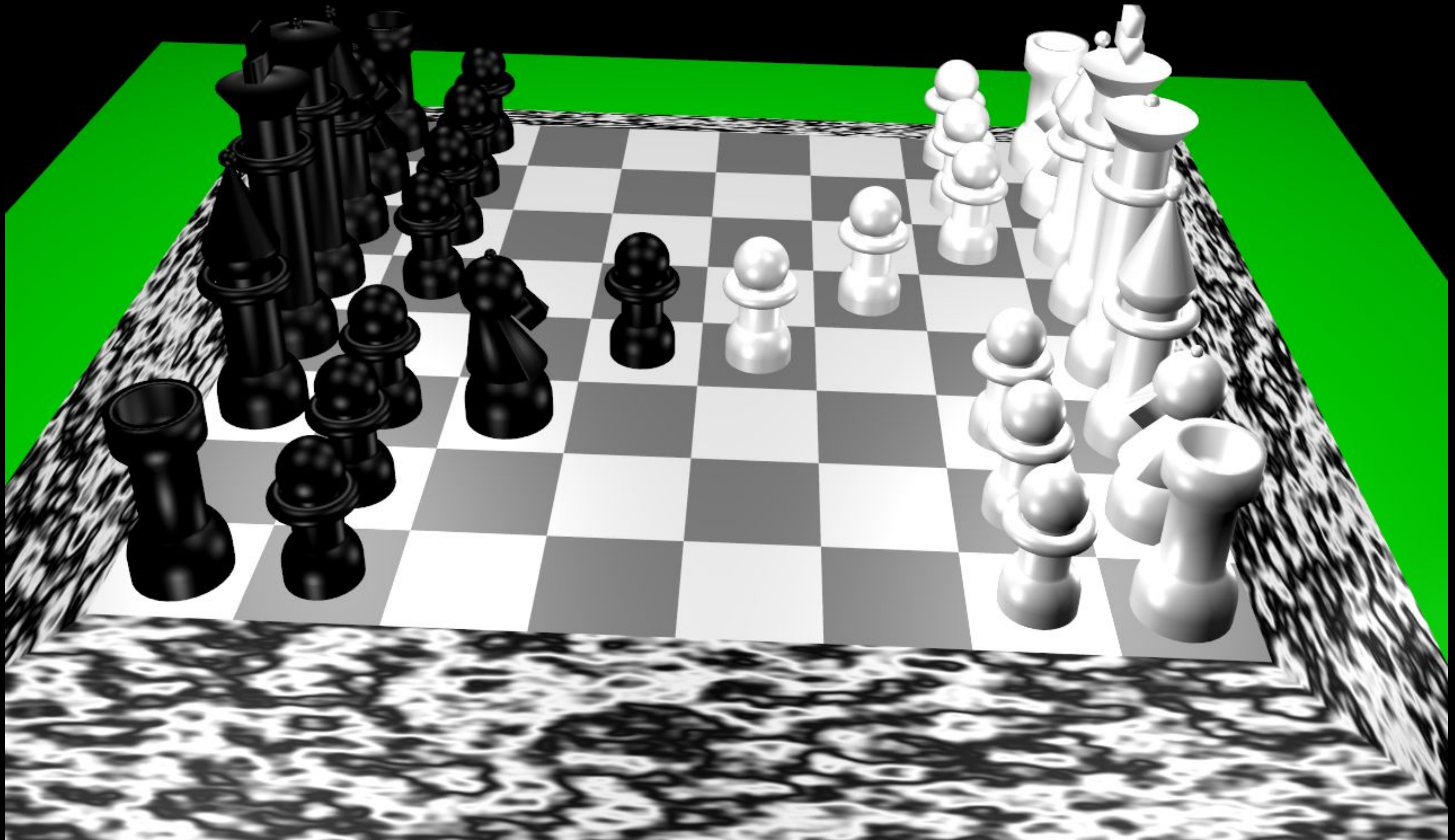
- We will generally be vague about the difference

Compare to "real-time" graphics

- The 3-D scene is "flattened" into a 2-D view plane
- Ray tracing is MUCH slower
  - But can handle reflections much better
- Some examples on the next few slides

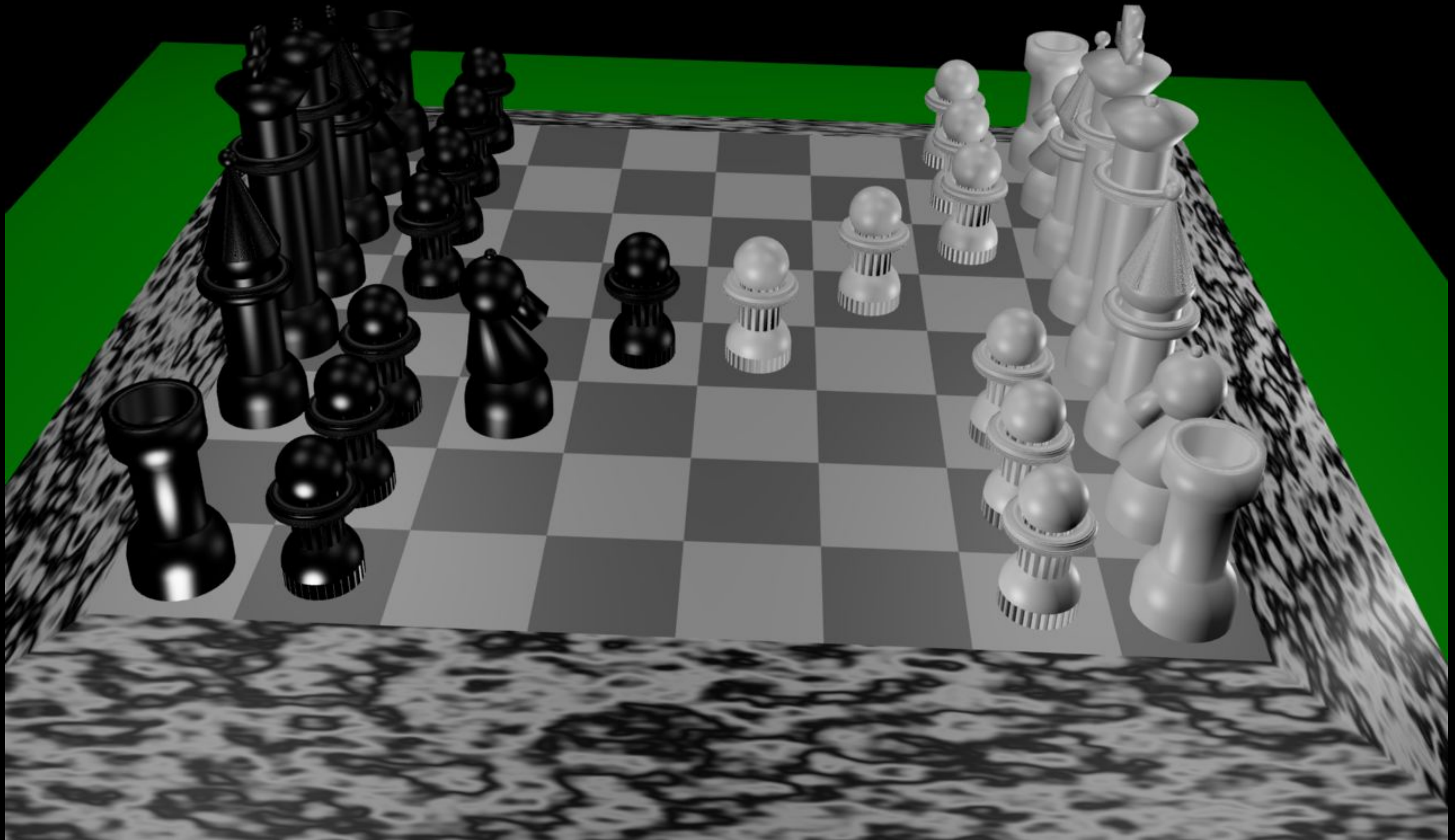


Rendered without raytracing



Rendered with raytracing

blender



WIKIPEDIA
The Free Encyclopedia

WIKIPEDIA

WIKIPEDIA

Ray Casting

- Simple implementation:

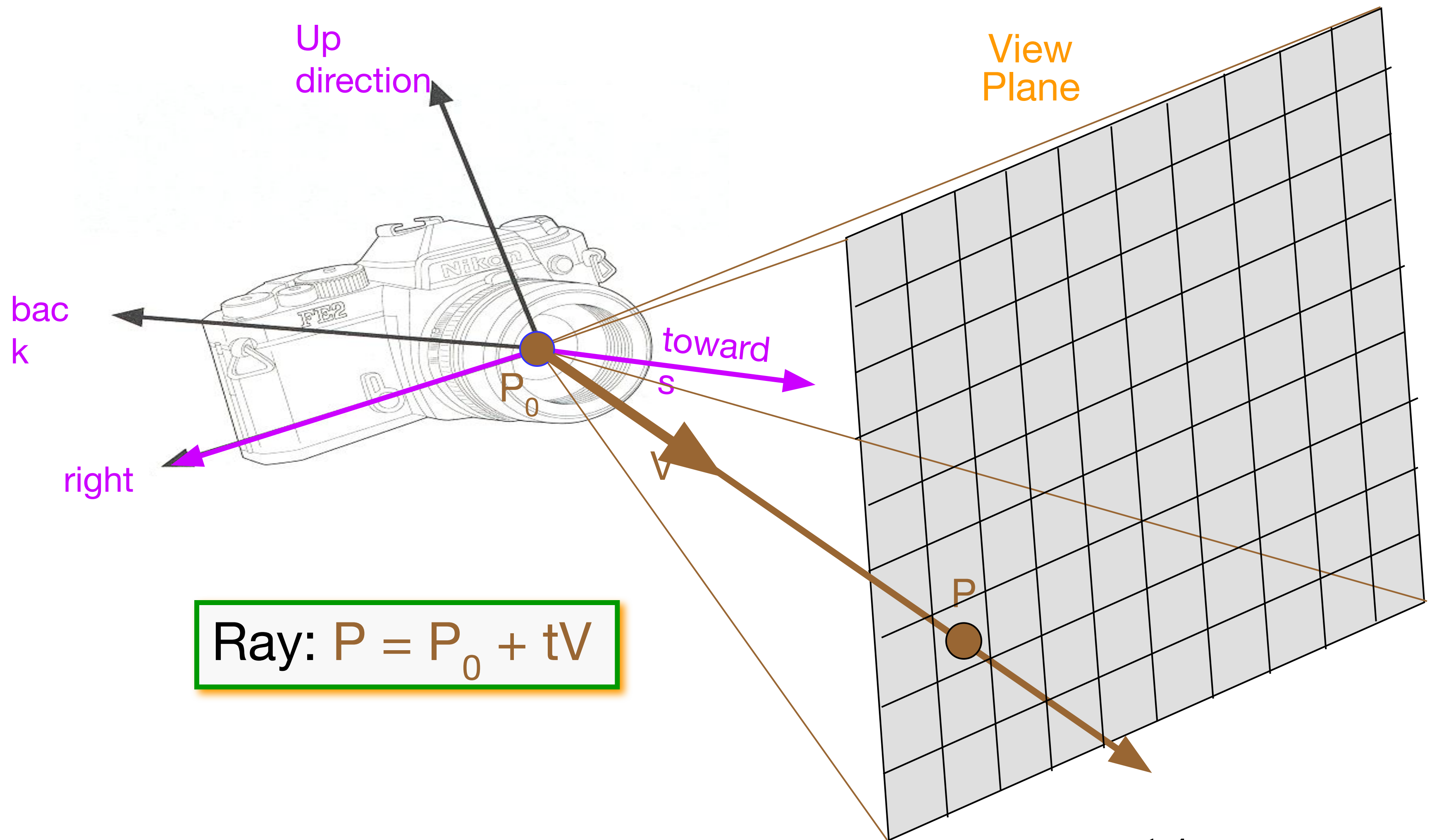
```
Image RayCast(Camera camera, Scene scene, int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(hit);
        }
    }
    return image;
}
```

Ray Casting

- Simple implementation:

```
Image RayCast(Camera camera, Scene scene, int width, int height)
{
	Image image = new Image(width, height);
	for (int i = 0; i < width; i++) {
		for (int j = 0; j < height; j++) {
			Ray ray = ConstructRayThroughPixel(camera, i, j);
			Intersection hit = FindIntersection(ray, scene);
			image[i][j] = GetColor(hit);
		}
	}
	return image;
}
```

Constructing Ray Through a Pixel

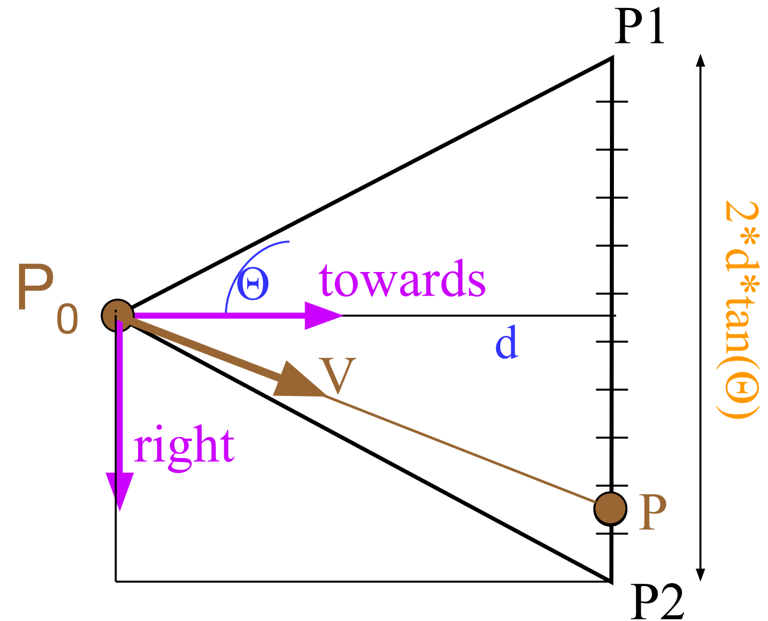


Constructing Ray Through a Pixel

- 2D Example

Θ = frustum half-angle
d = distance to view plane

right = towards x up



$P1 = P_0 + d*towards - d*tan(\Theta)*right$
$P2 = P_0 + d*towards + d*tan(\Theta)*right$

$P = P1 + (i + 0.5) / width * (P2 - P1)$
$\quad = P1 + (i + 0.5) / width * 2*d*tan(\Theta)*right$
$V = (P - P_0) / | P - P_0 |$

Ray: $P = P_0 + tV$

Ray Casting

- Simple implementation:

```
Image RayCast(Camera camera, Scene scene, int width, int height)
{
	Image image = new Image(width, height);
	for (int i = 0; i < width; i++) {
		for (int j = 0; j < height; j++) {
			Ray ray = ConstructRayThroughPixel(camera, i, j);
			Intersection hit = FindIntersection(ray, scene);
			image[i][j] = GetColor(hit);
		}
	}
	return image;
}
```

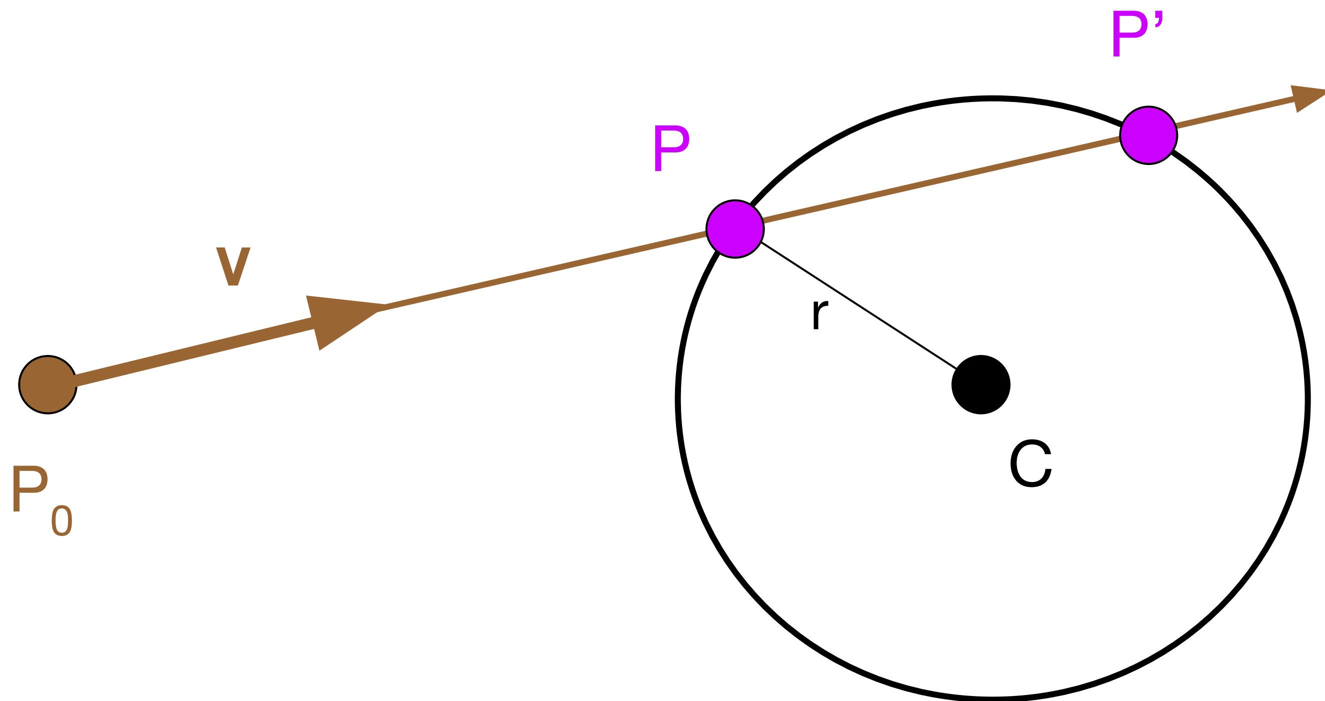
Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)
- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - Uniform grids
    - Octrees
    - BSP trees

Ray-Sphere Intersection

Ray: $P = P_0 + tV$

Sphere: $|P - C|^2 - r^2 = 0$



Ray-Sphere Intersection

Ray: $P = P_0 + tV$

Sphere: $(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2$

$$|P - C|^2 - r^2 = 0$$

Substituting for P, we get:

$$|\mathbf{P_0} + \mathbf{tV} - C|^2 - r^2 = 0$$

Solve quadratic equation:

$$at^2 + bt + c = 0$$

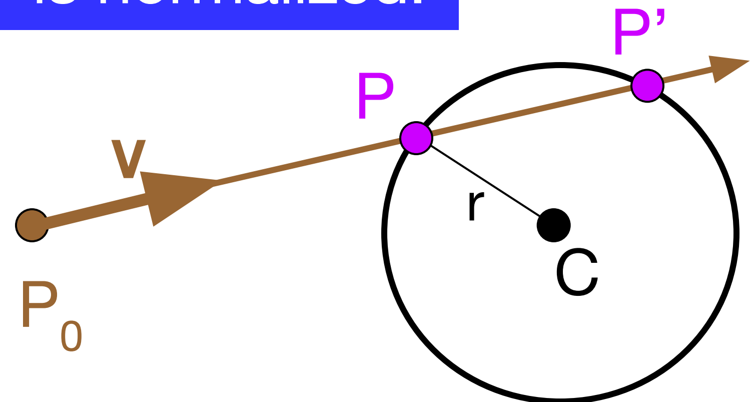
where:

$a = |V|^2 = 1$

$b = 2\, V \bullet (P_0 - C)$

$c = |P_0 - C|^2 - r^2$

If ray direction is normalized!

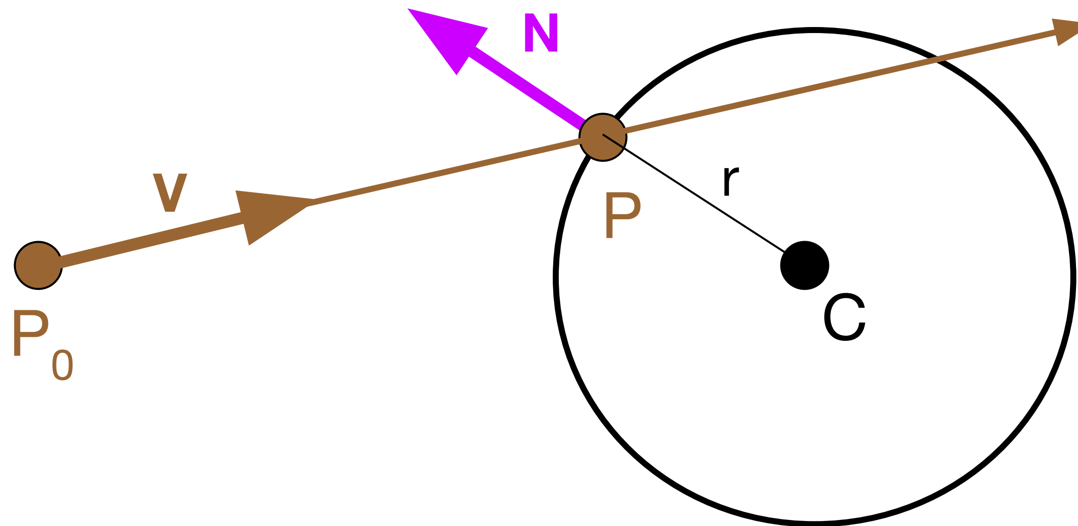


$P = P0 + tV$

Ray-Sphere Intersection

- Need normal vector at intersection for lighting calculations

$$N = (P - C) / |P - C|$$

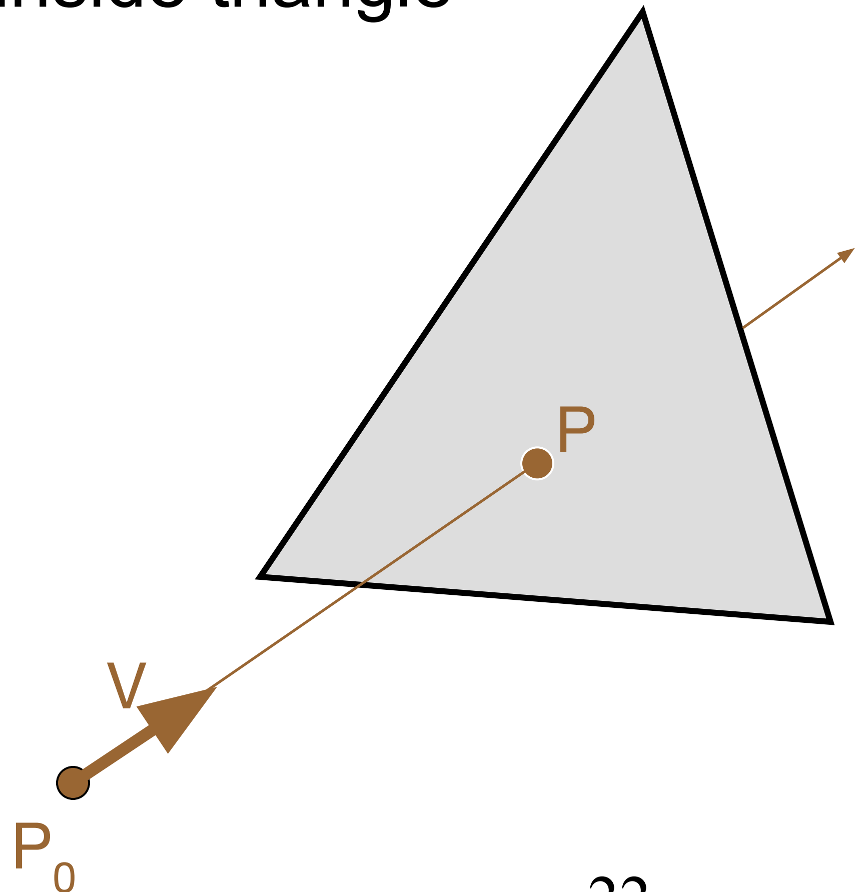


Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - **Triangle**
  - Groups of primitives (scene)
- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - Uniform grids
    - Octrees
    - BSP trees

Ray-Triangle Intersection

- First, intersect ray with plane
- Then, check if point is inside triangle



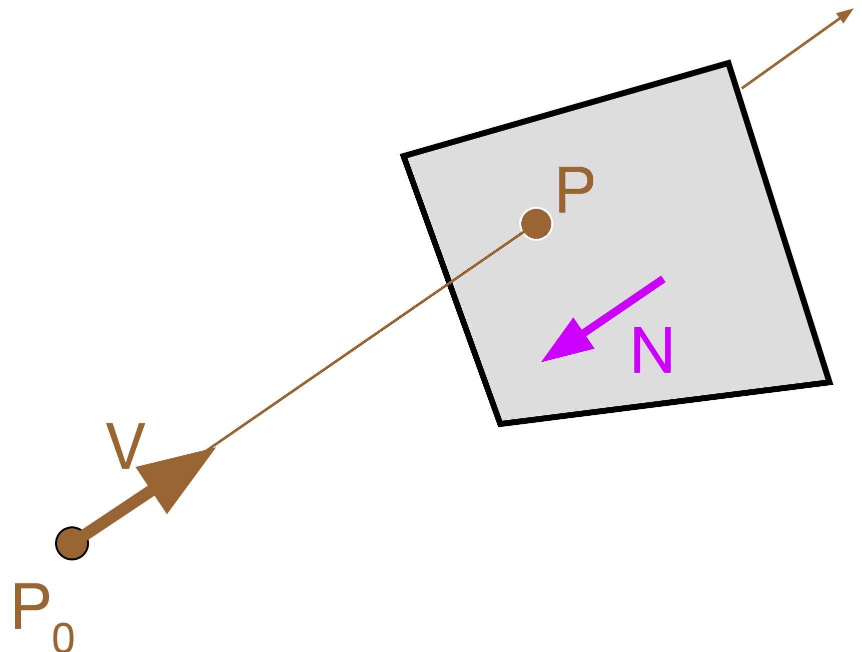
Ray-Plane Intersection

Ray: $P = P_0 + tV$
Plane: $ax + by + cz + d = 0$
$P \bullet N + d = 0$

Substituting for P, we get:
$(\mathbf{P_0 + tV}) \bullet N + d = 0$

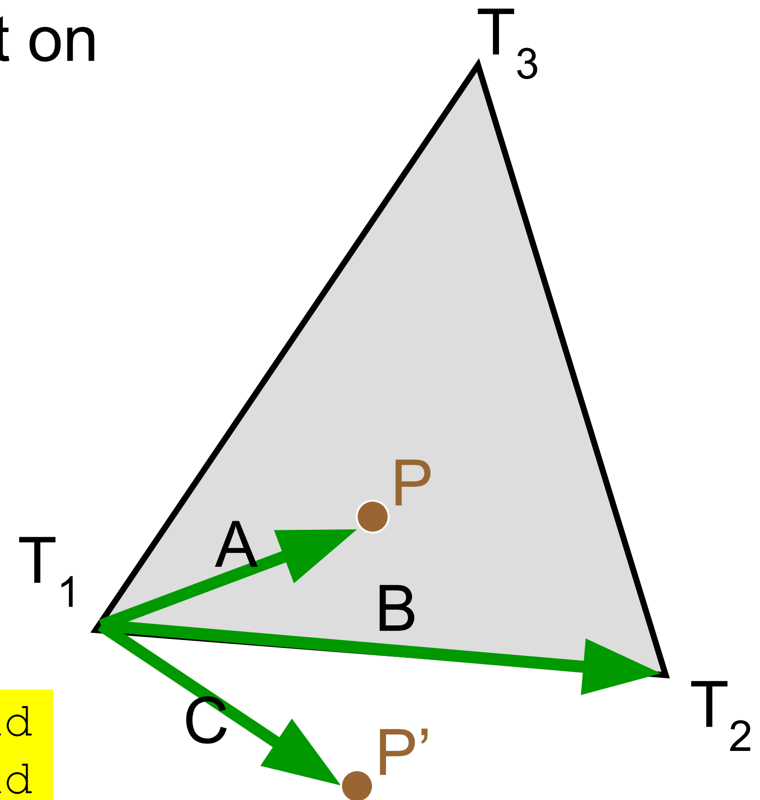
Solution:
$t = -(P_0 \bullet N + d) / (V \bullet N)$

$P = P_0 + tV$



Ray-Triangle Intersection I

- Check if point is inside triangle geometrically
  - First, find ray intersection point on plane defined by triangle
  - AxB will point in the opposite direction from CxB




```
SameSide(p1,p2, a,b):
   cp1 = Cross (b-a, p1-a)
   cp2 = Cross (b-a, p2-a)
   return Dot (cp1, cp2) >= 0
```

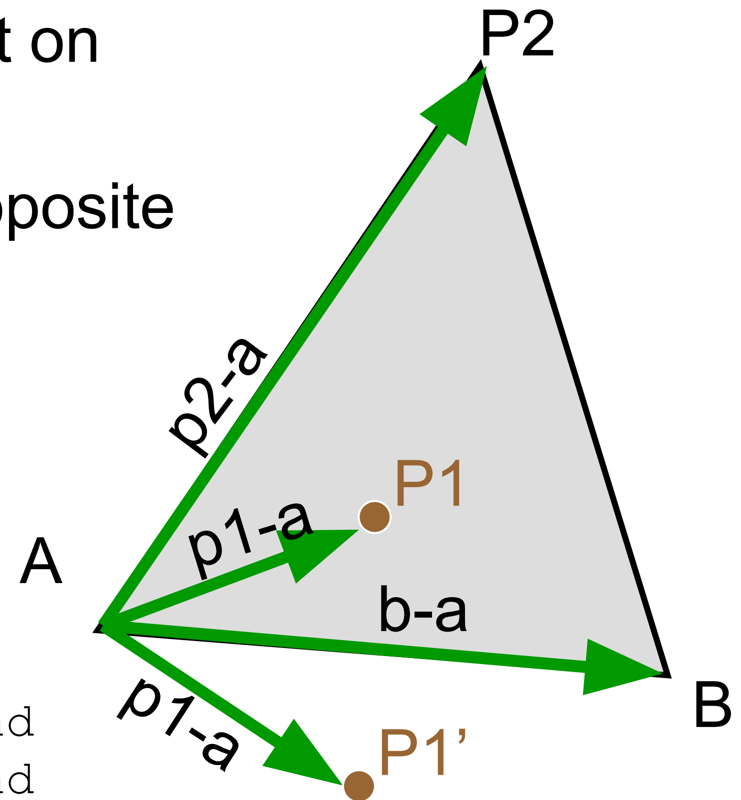
```
PointInTriangle(p, t1, t2, t3):
   return SameSide(p, t1, t2, t3) and
          SameSide(p, t2, t1, t3) and
          SameSide(p, t3, t1, t2)
```

Ray-Triangle Intersection II

- Check if point is inside triangle geometrically
  - First, find ray intersection point on plane defined by triangle
  - (p1-a)x(b-a) will point in the opposite direction from (p1-a)x(b-a)

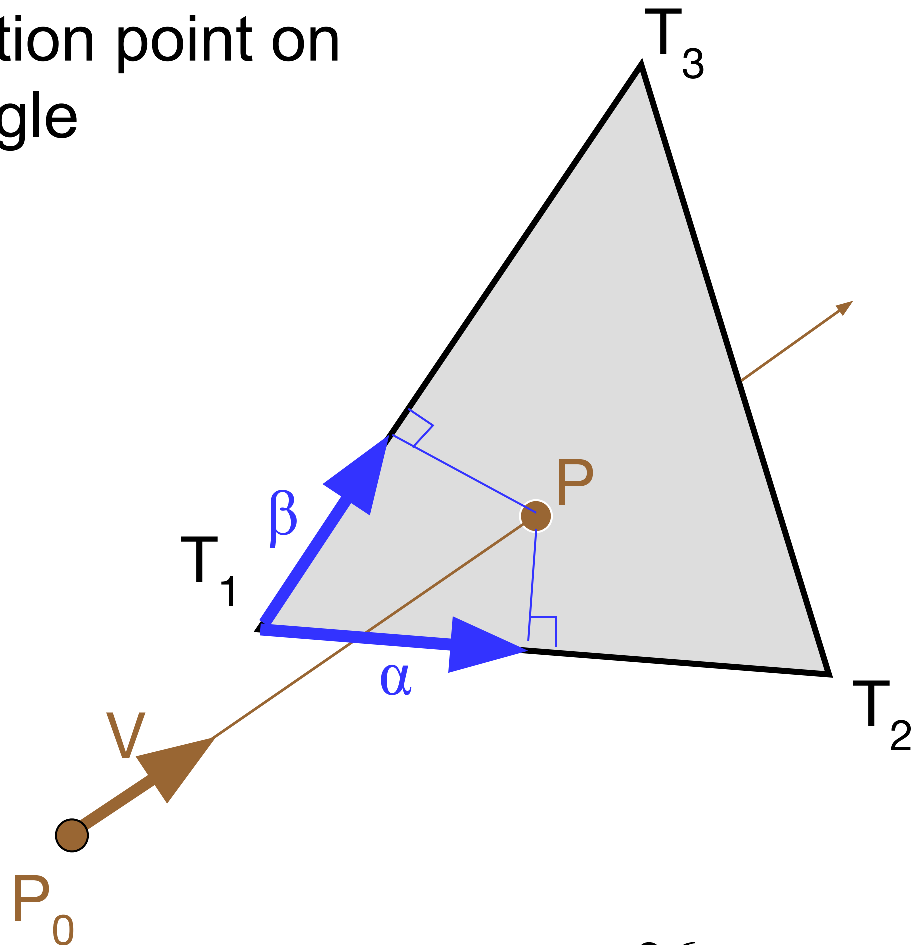
```
SameSide(p1,p2, a,b):
   cp1 = Cross (b-a, p1-a)
   cp2 = Cross (b-a, p2-a)
   return Dot (cp1, cp2) >= 0
```

```
PointInTriangle(p, t1, t2, t3):
   return SameSide(p, t1, t2, t3) and
          SameSide(p, t2, t1, t3) and
          SameSide(p, t3, t1, t2)
```



Ray-Triangle Intersection III

- Check if point is inside triangle parametrically
  - First, find ray intersection point on plane defined by triangle



Compute α, β:

$P = \alpha (T_2 - T_1) + \beta (T_3 - T_1)$

Check if point inside triangle.

$0 \le \alpha \le 1$ and $0 \le \beta \le 1$

$\alpha + \beta \le 1$

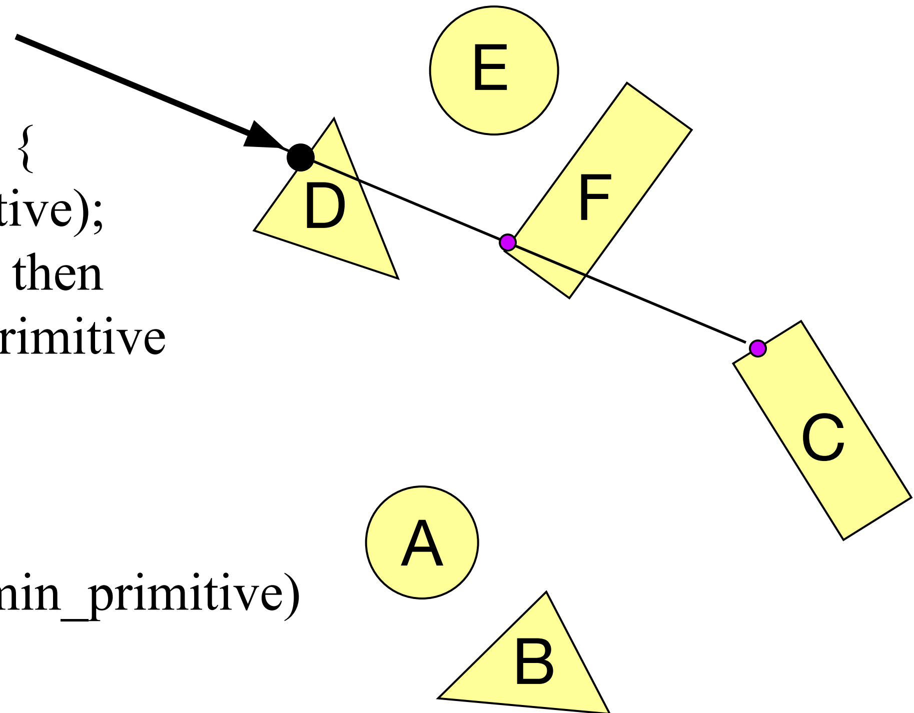
Other Ray-Primitive Intersections

- Cone, cylinder, ellipsoid:
  - Similar to sphere
- Box
  - Intersect front-facing planes (max 3!), return closest
- Convex polygon
  - Same as triangle (check point-in-polygon algebraically)
- Concave polygon
  - Same plane intersection
  - More complex point-in-polygon test

Ray-Scene Intersection

- Find intersection with front-most primitive in group

```
Intersection FindIntersection(Ray ray, Scene scene)
{
    min_t = infinity
    min_primitive = NULL
    For each primitive in scene {
        t = Intersect(ray, primitive);
        if (t > 0 && t < min_t) then
            min_primitive = primitive
            min_t = t
        }
    }
    return Intersection(min_t, min_primitive)
}
```



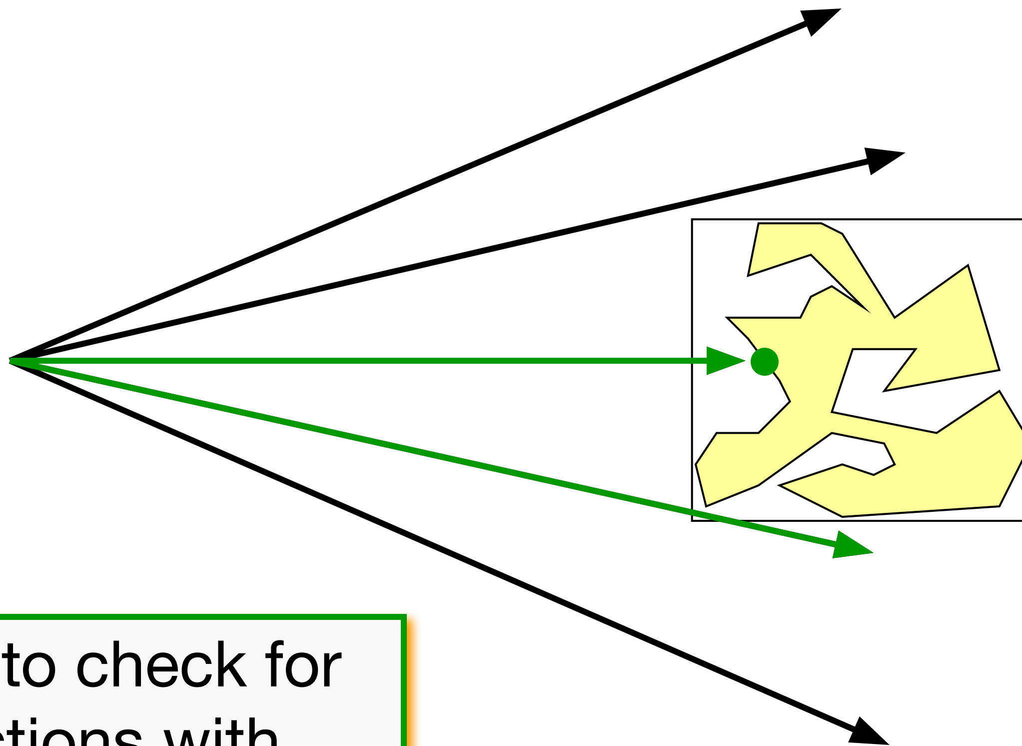
Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)

- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - Uniform grids
    - Octrees
    - BSP trees

Bounding Volumes

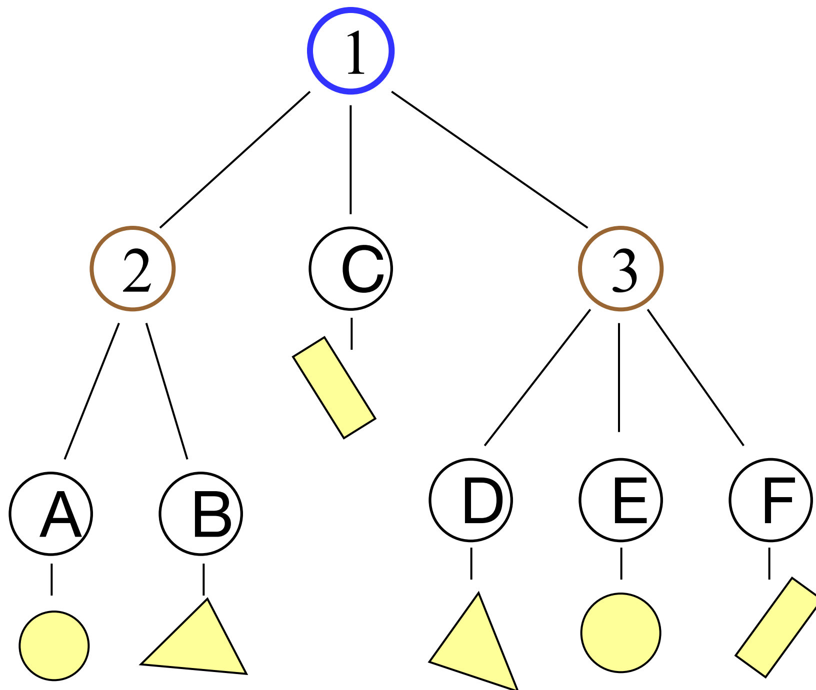
- Check for intersection with simple shape first
  - If ray doesn't intersect bounding volume, then it doesn't intersect its contents

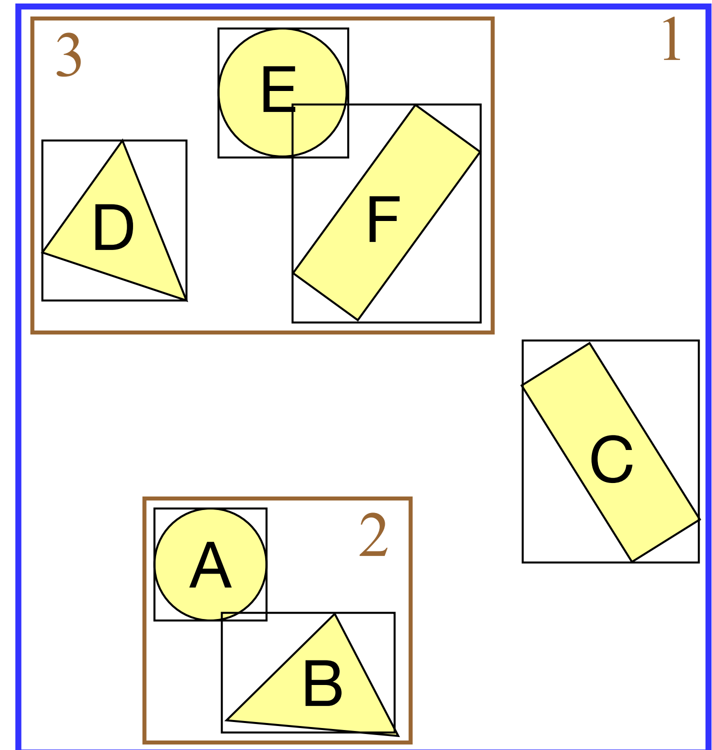


Still need to check for intersections with shape.

Bounding Volume Hierarchies I

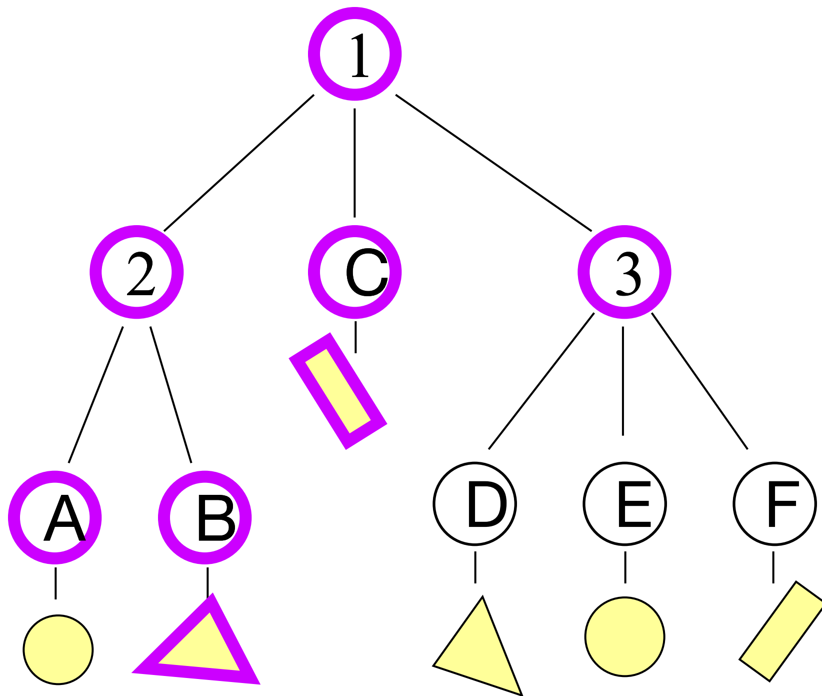
- Build hierarchy of bounding volumes
  - Bounding volume of interior node contains all children

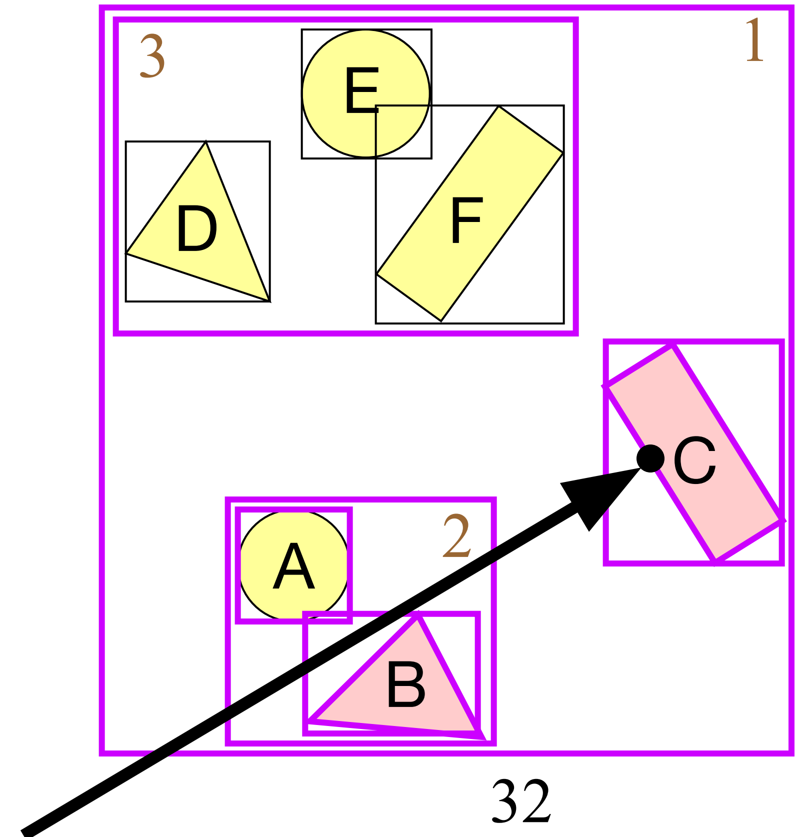




Bounding Volume Hierarchies

- Use hierarchy to accelerate ray intersections
  - Intersect node contents only if hit bounding volume





Bounding Volume Hierarchies III

- Sort hits & detect early termination

```
FindIntersection(Ray ray, Node node)
{
    // Find intersections with child node bounding volumes
    ...
    // Sort intersections front to back
    ...
    // Process intersections (checking for early termination)
    min_t = infinity;
    for each intersected child i {
        if (min_t < bv_t[i]) break;
        shape_t = FindIntersection(ray, child);
        if (shape_t < min_t) { min_t = shape_t;}
    }
    return min_t;
}
```

Ray-Scene Intersection

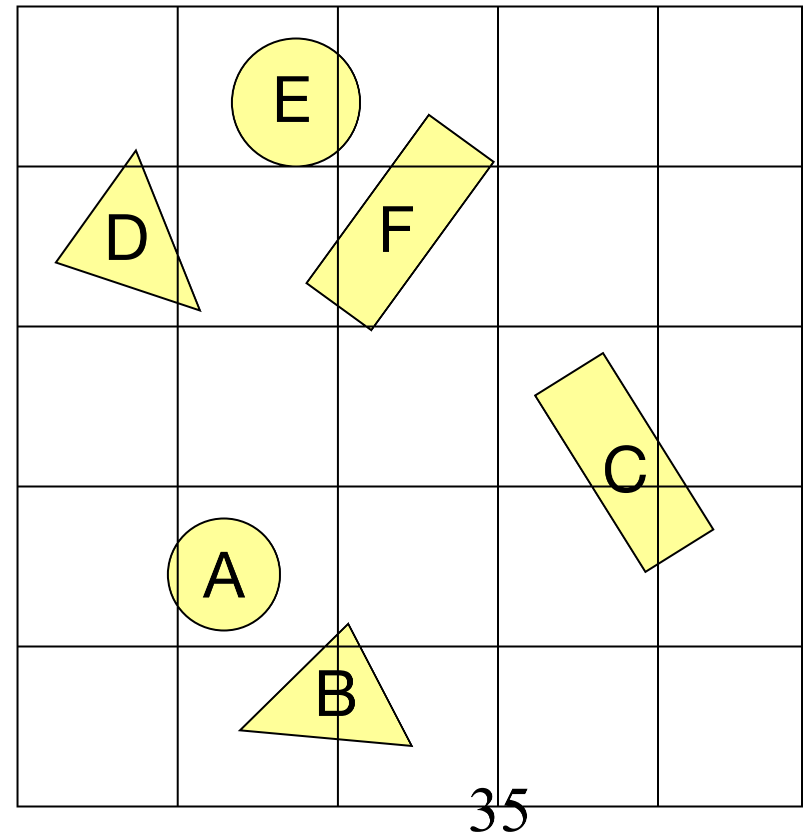
- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)

Acceleration techniques

  - Bounding volume hierarchies
  - Spatial partitions
    - Uniform grids
    - Octrees
    - BSP trees

Uniform Grid

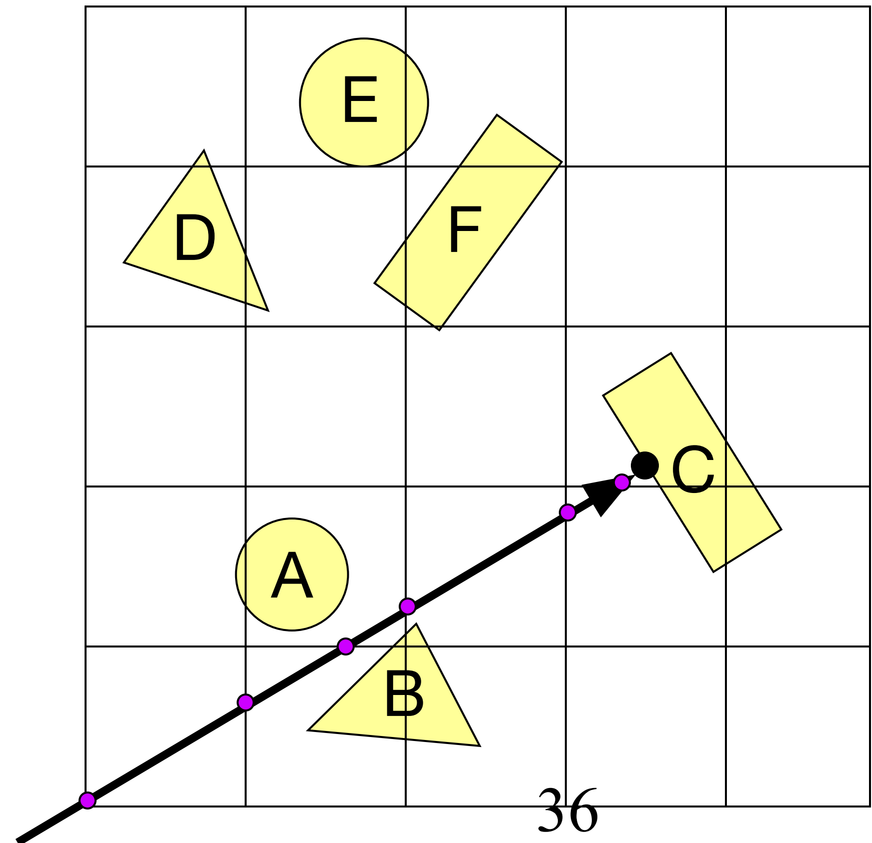
- Construct uniform grid over scene
  - Index primitives according to overlaps with grid cells



Uniform Grid

- Trace rays through grid cells
  - Fast
  - Incremental

Only check primitives in intersected grid cells

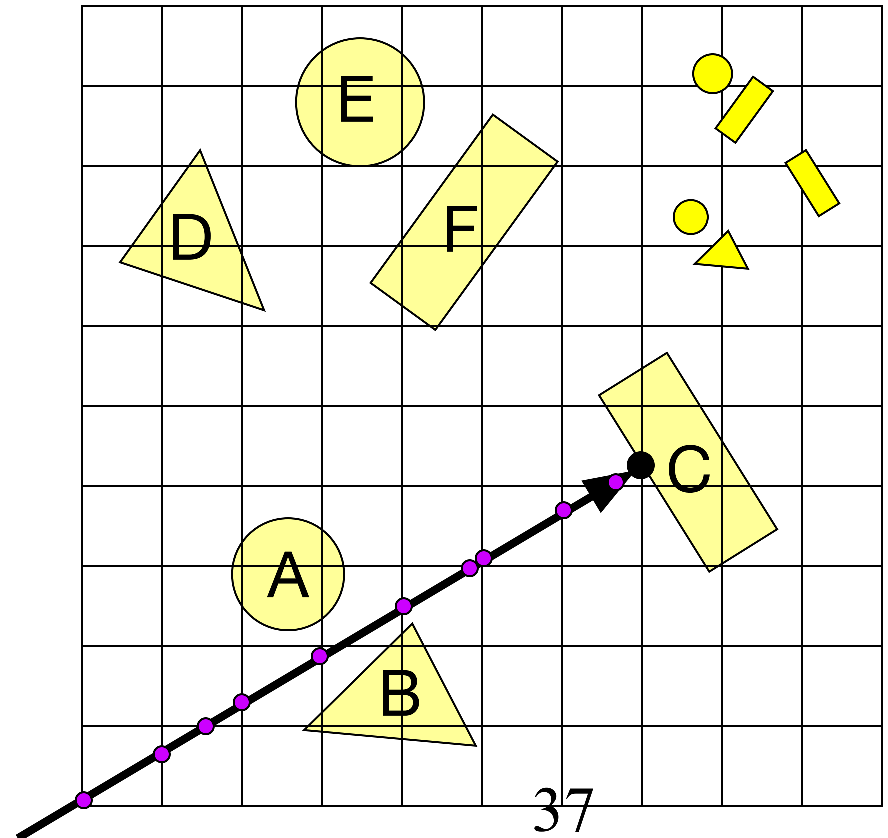


Uniform Grid

- Potential problem:
  - How choose suitable grid resolution?

Too little benefit if grid is too coarse

Too much cost if grid is too fine



Ray-Scene Intersection

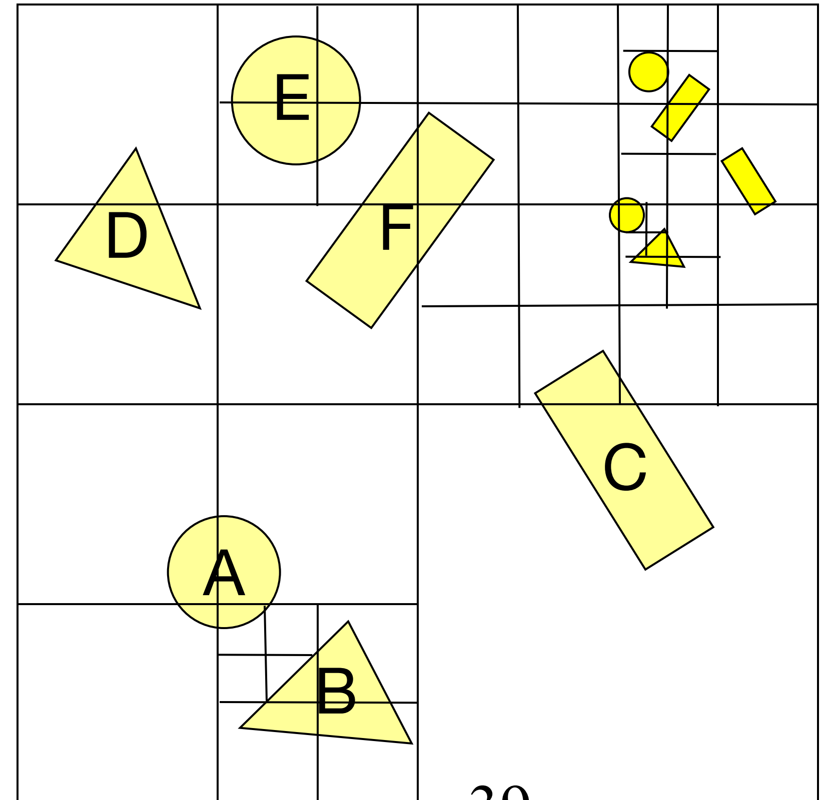
- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)

- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - Uniform grids
    - Octrees
    - BSP trees

Octree

- Construct adaptive grid over scene
  - Recursively subdivide box-shaped cells into 8 octants
  - Index primitives by overlaps with cells

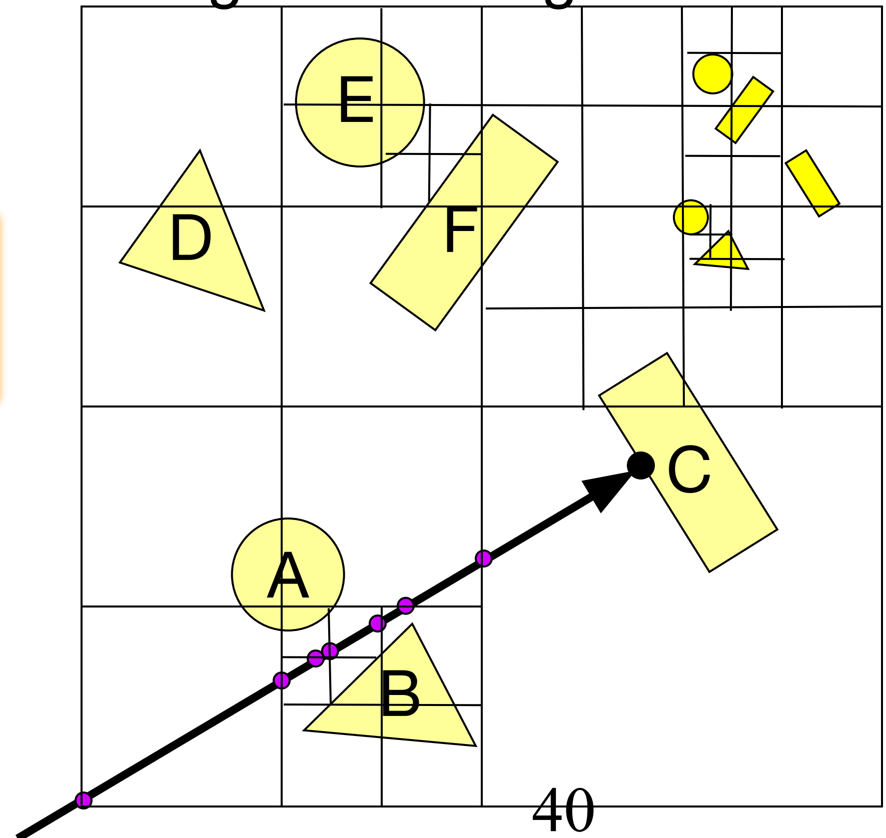
Generally fewer cells



Octree

- Trace rays through neighbor cells
  - Fewer cells
  - Recursive descent – don't do neighbor finding…

Trade-off fewer cells for more expensive traversal



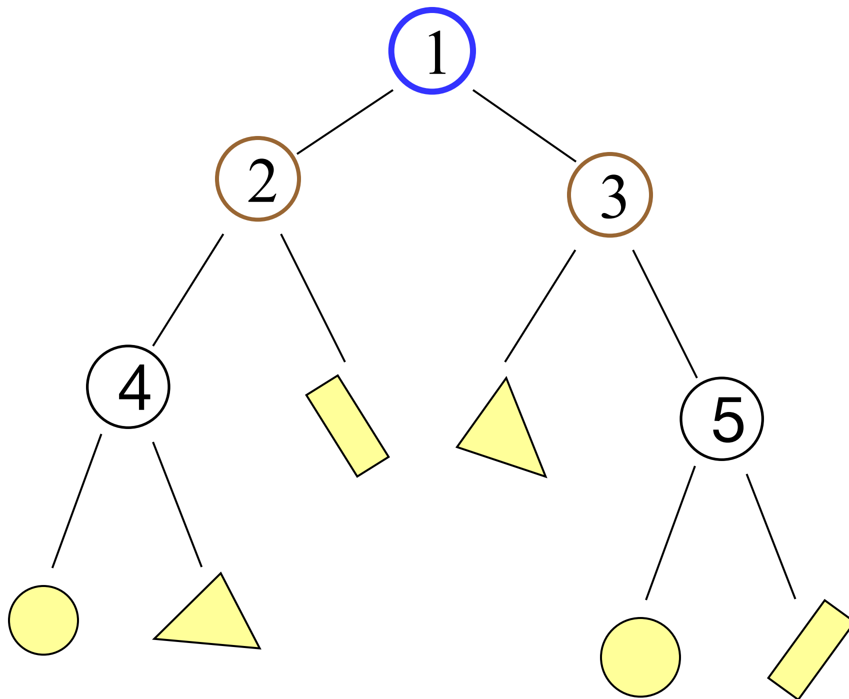
Ray-Scene Intersection

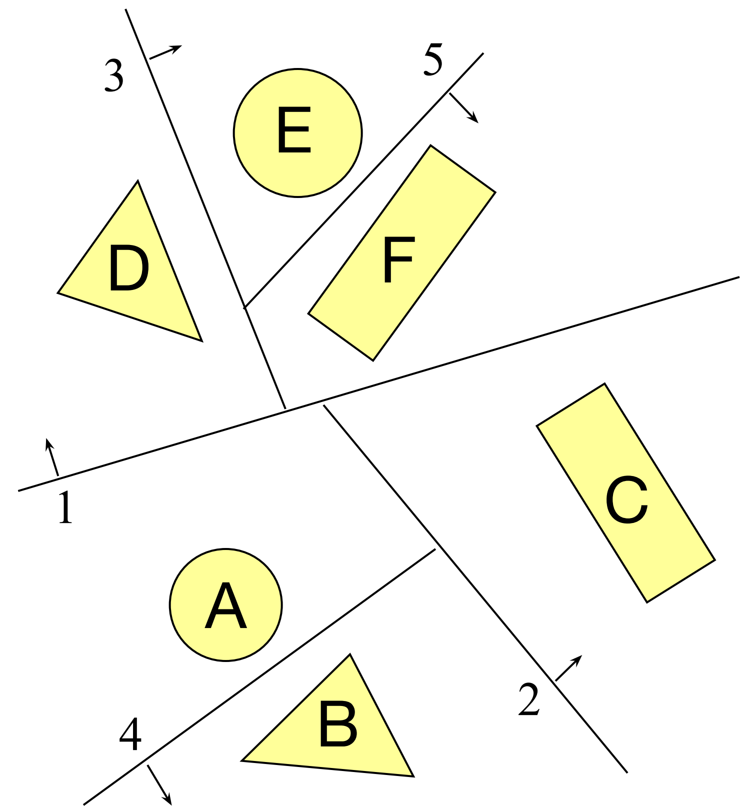
- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)

- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - Uniform grids
    - Octrees
    - BSP trees

Binary Space Partition (BSP) Tree

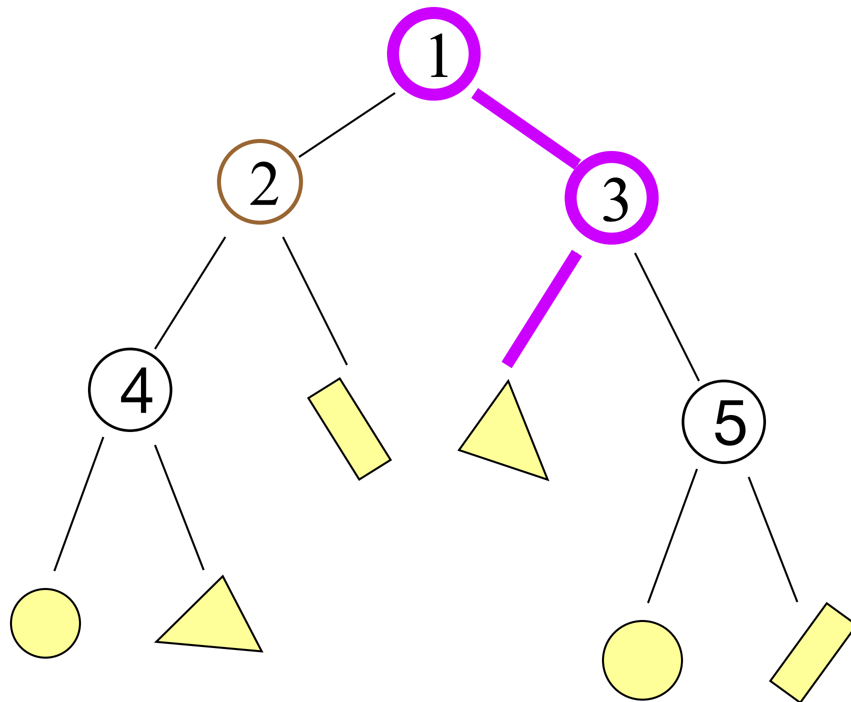
- Recursively partition space by planes
  - Every cell is a convex polyhedron

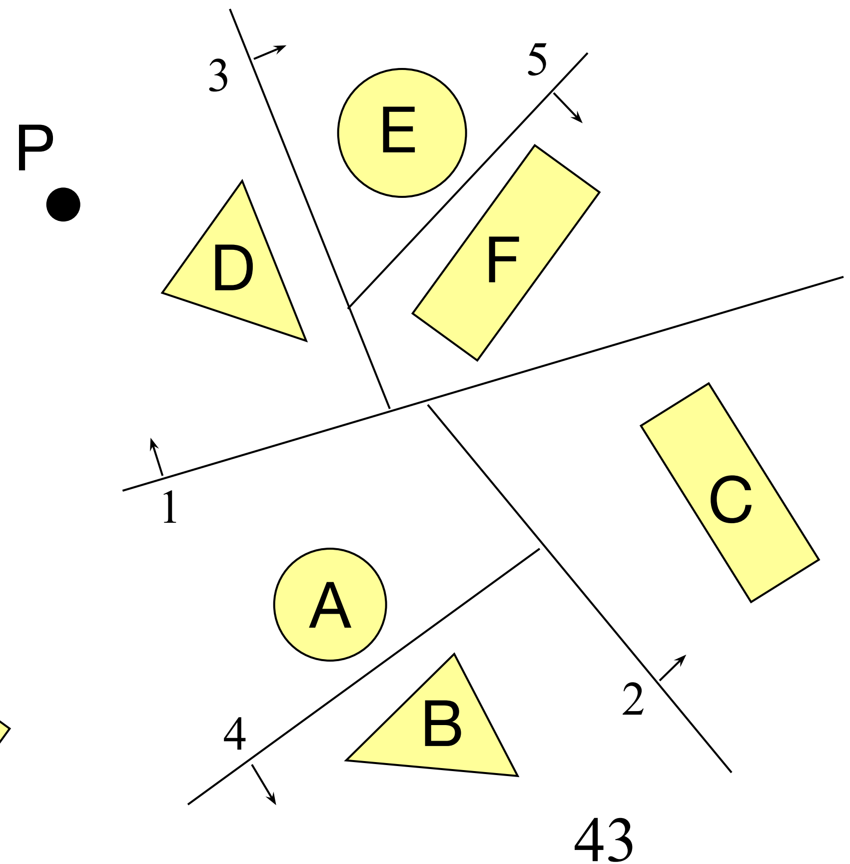




Binary Space Partition (BSP) Tree

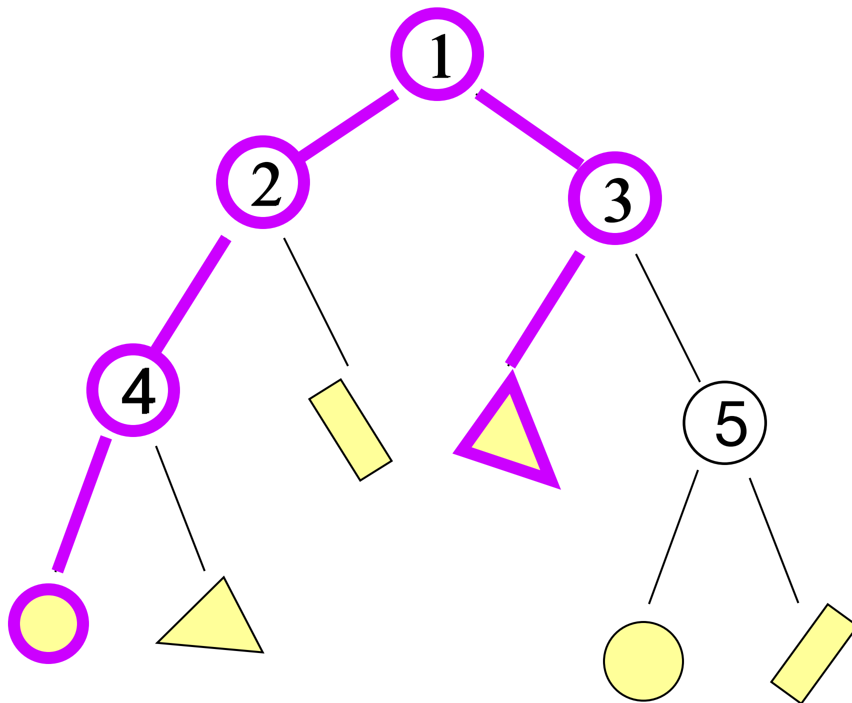
- Simple recursive algorithms
  - Example: point location

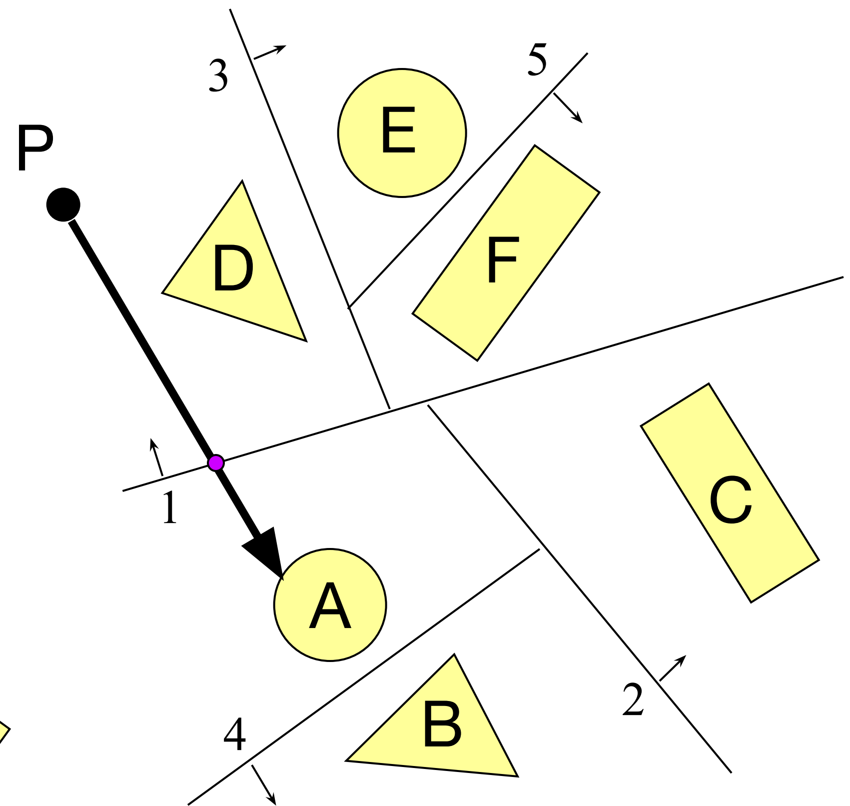


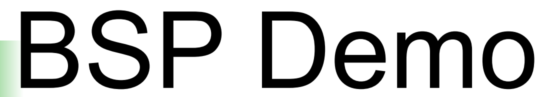


Binary Space Partition (BSP) Tree

- Trace rays by recursion on tree
  - BSP construction enables simple front-to-back traversal





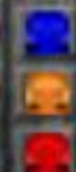


BSP Demo

- http://symbolcraft.com/graphics/bsp/

First game-based use of BSP trees

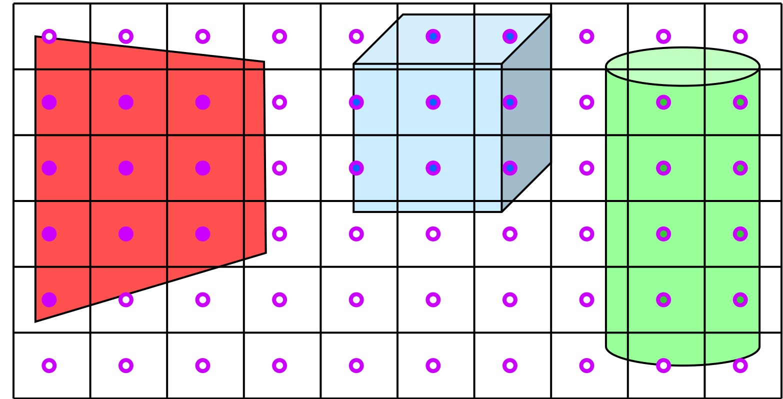
OPCG.net



Doom *(ID Software)*

Other Accelerations

- Screen space coherence
  - Check last hit first
  - Beam tracing
  - Pencil tracing
  - Cone tracing
- Memory coherence
  - Large scenes
- Parallelism
  - Ray casting is “embarrassingly parallel”
- etc.



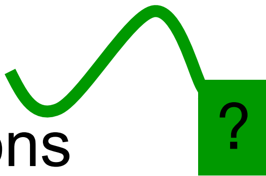
Acceleration

- Intersection acceleration techniques are important
  - Bounding volume hierarchies
  - Spatial partitions
- General concepts
  - Sort objects spatially
  - Make trivial rejections quick
  - Utilize coherence when possible

Expected time is sub-linear in number of primitives

Summary

- Writing a simple ray casting renderer is “easy”
  - Generate rays
  - Intersection tests
  - Lighting calculations

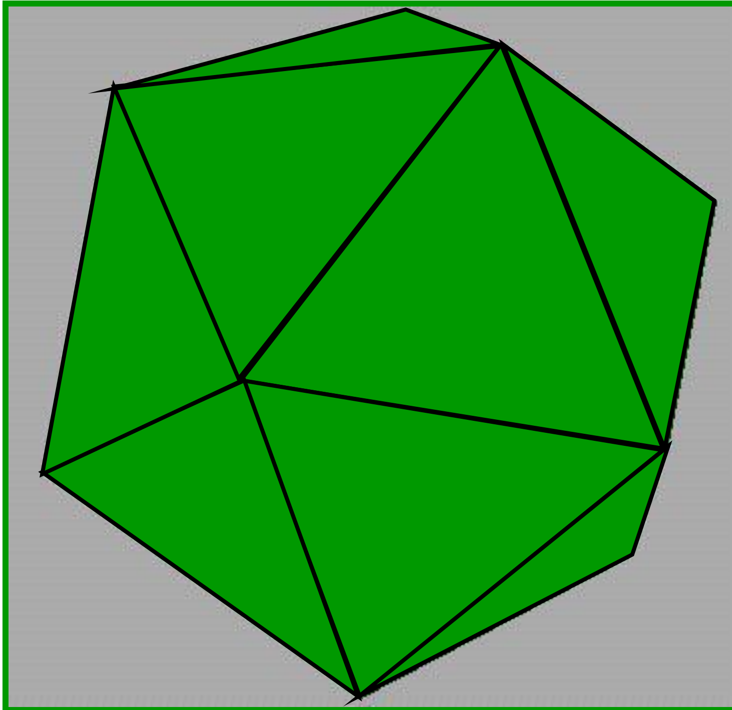



```
Image RayCast(Camera camera, Scene scene, int width, int height)
{
	Image image = new Image(width, height);
	for (int i = 0; i < width; i++) {
		for (int j = 0; j < height; j++) {
			Ray ray = ConstructRayThroughPixel(camera, i, j);
			Intersection hit = FindIntersection(ray, scene);
			image[i][j] = GetColor(hit);
		}
	}
	return image;
}
```

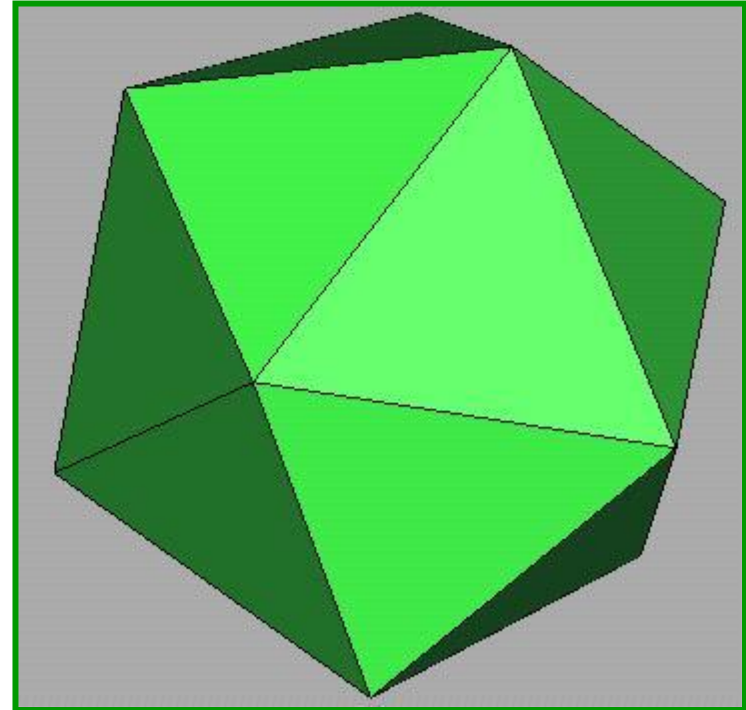
Heckbert's business card ray tracer

```
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{ vec cen,color;
double rad,kd,ks,kt,kl,ir}*s,*best,sph[]={0.,6.,.5,1.,1.,1.,.9, .05,.2,.85,0.,1.7,-1.,8.,-.5,1.,.5,.2,1.,
.7,.3,0.,.05,1.2,1.,8.,-.5,.1,.8,.8, 1.,.3,.7,0.,0.,1.2,3.,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-3.,-3.,12.,
.8,1., 1.,5.,0.,0.,0.,.5,1.5,};yx;double u,b,tmin,sqrt(),tan();double vdot(A,B)vec A ,B;{return A.x
*B.x+A.y*B.y+A.z*B.z;}vec vcomb(a,A,B)double a;vec A,B;{B.x+=a* A.x;B.y+=a*A.y;B.z+=a*A.z;
return B;}vec vunit(A)vec A;{return vcomb(1./sqrt( vdot(A,A)),A,black);}struct sphere*intersect
(P,D)vec P,D;{best=0;tmin=1e30;s= sph+5;while(s-->sph)b=vdot(D,U=vcomb(-1.,P,s->cen)),
u=b*b-vdot(U,U)+s->rad*s ->rad,u=u>0?sqrt(u):1e31,u=b-u>1e-7?b-u:b+u,tmin=u>=1e-7&&
u<tmin?best=s,u: tmin;return best;}vec trace(level,P,D)vec P,D;{double d,eta,e;vec N,color;
struct sphere*s,*l;if(!level--)return black;if(s=intersect(P,D));else return amb;color=amb;eta=
s->ir;d= -vdot(D,N=vunit(vcomb(-1.,P=vcomb(tmin,D,P),s->cen )));if(d<0)N=vcomb(-1.,N,black),
eta=1/eta,d= -d;l=sph+5;while(l-->sph)if((e=l ->kl*vdot(N,U=vunit(vcomb(-1.,P,l->cen))))>0&&
intersect(P,U)==l)color=vcomb(e ,l->color,color);U=s->color;color.x*=U.x;color.y*=U.y;color.z
*=U.z;e=1-eta* eta*(1-d*d);return vcomb(s->kt,e>0?trace(level,P,vcomb(eta,D,vcomb(eta*d-
sqrt (e),N,black))):black,vcomb(s->ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s->kd, color,vcomb
(s->kl,U,black))));}main(){printf("%d %d\n",32,32);while(yx<32*32) U.x=yx%32-32/2,U.z=32/2-
yx++/32,U.y=32/2/tan(25/114.5915590261),U=vcomb(255., trace(3,black,vunit(U)),black),printf
("%.0f %.0f %.0f\n",U);}/*minray!*/
```

Next Time is Illumination!



Without Illumination



With Illumination