



3D Rendering Intro & Ray Casting

COS 426, Spring 2022

Felix Heide

Princeton University

Computer Graphics

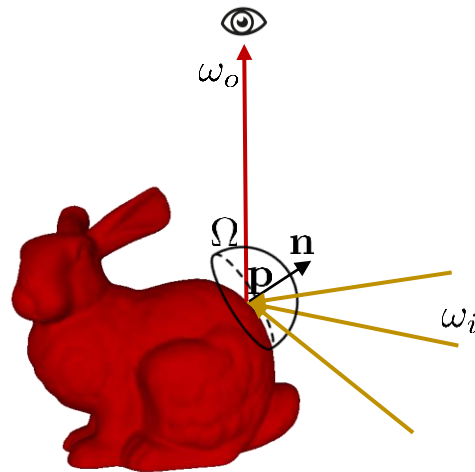


Source: (Project Sol Part 2) <https://www.youtube.com/watch?v=pNmHx8yPLk>

Rendering Equation

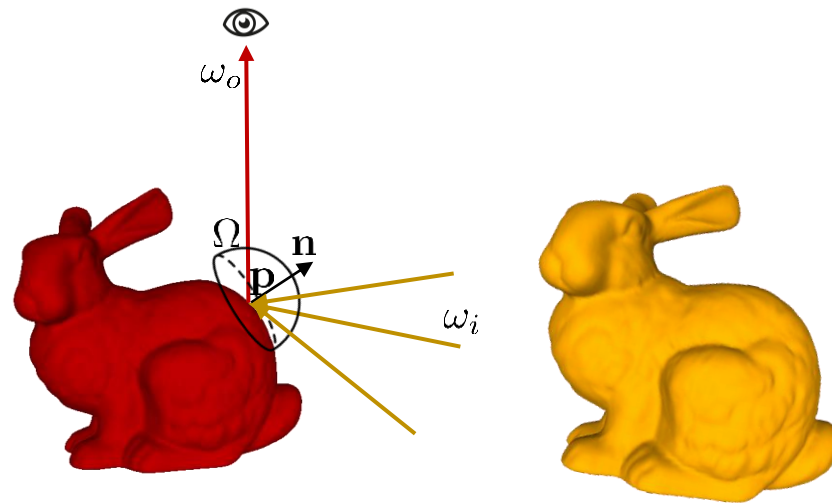
$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{\Omega} L_i(\mathbf{p}, \omega_i) f_r(\mathbf{p}, \omega_i, \omega_o) (\omega_i \cdot \mathbf{n}) d\omega_i$$

Outgoing radiance Incident radiance BRDF



Rendering Equation

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{\Omega} \boxed{L_i(\mathbf{p}, \omega_i)} f_r(\mathbf{p}, \omega_i, \omega_o) (\omega_i \cdot \mathbf{n}) d\omega_i$$



Syllabus



I. Image processing

II. Modeling

III. Rendering

IV. Animation

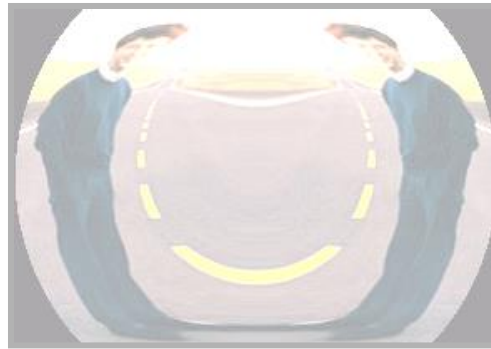
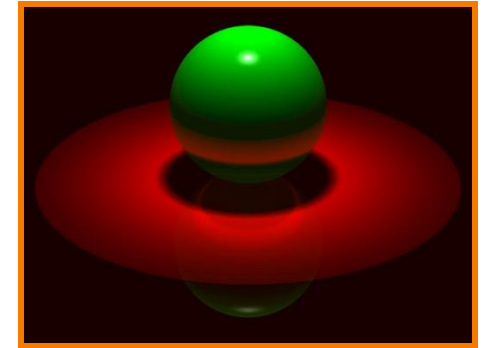


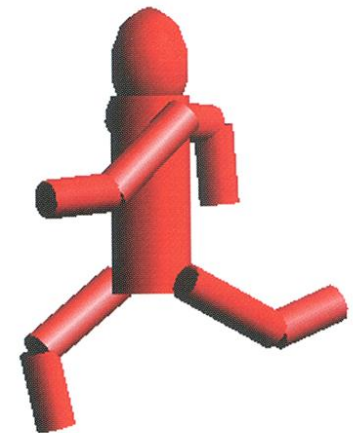
Image Processing
(Rusty Coleman, CS426, Fall99)



Rendering
(Michael Bostock, CS426, Fall99)



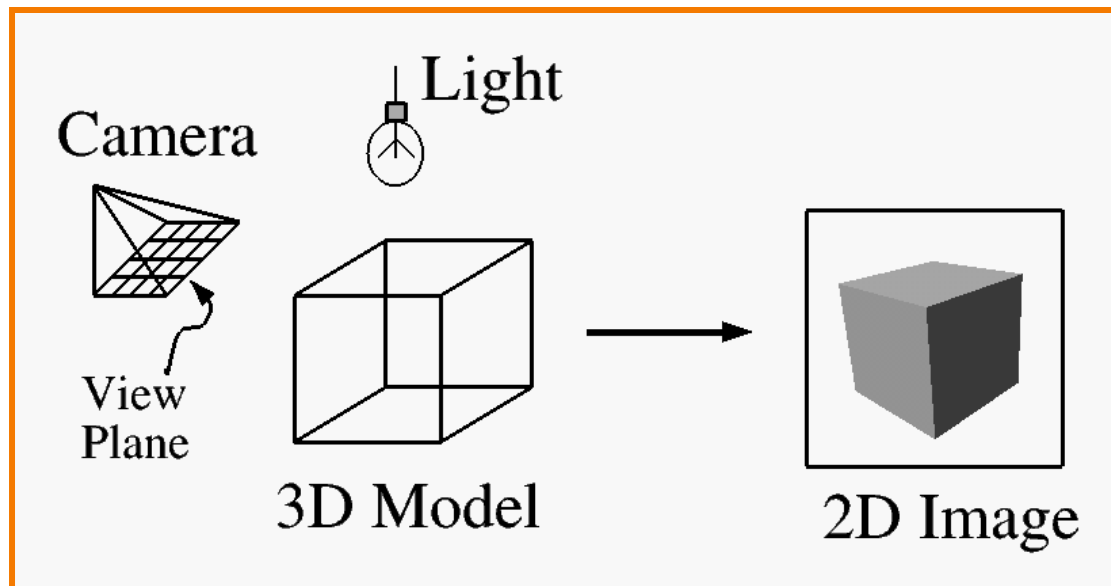
Modeling
(Dennis Zorin, CalTech)



Animation
(Angel, Plate 1)

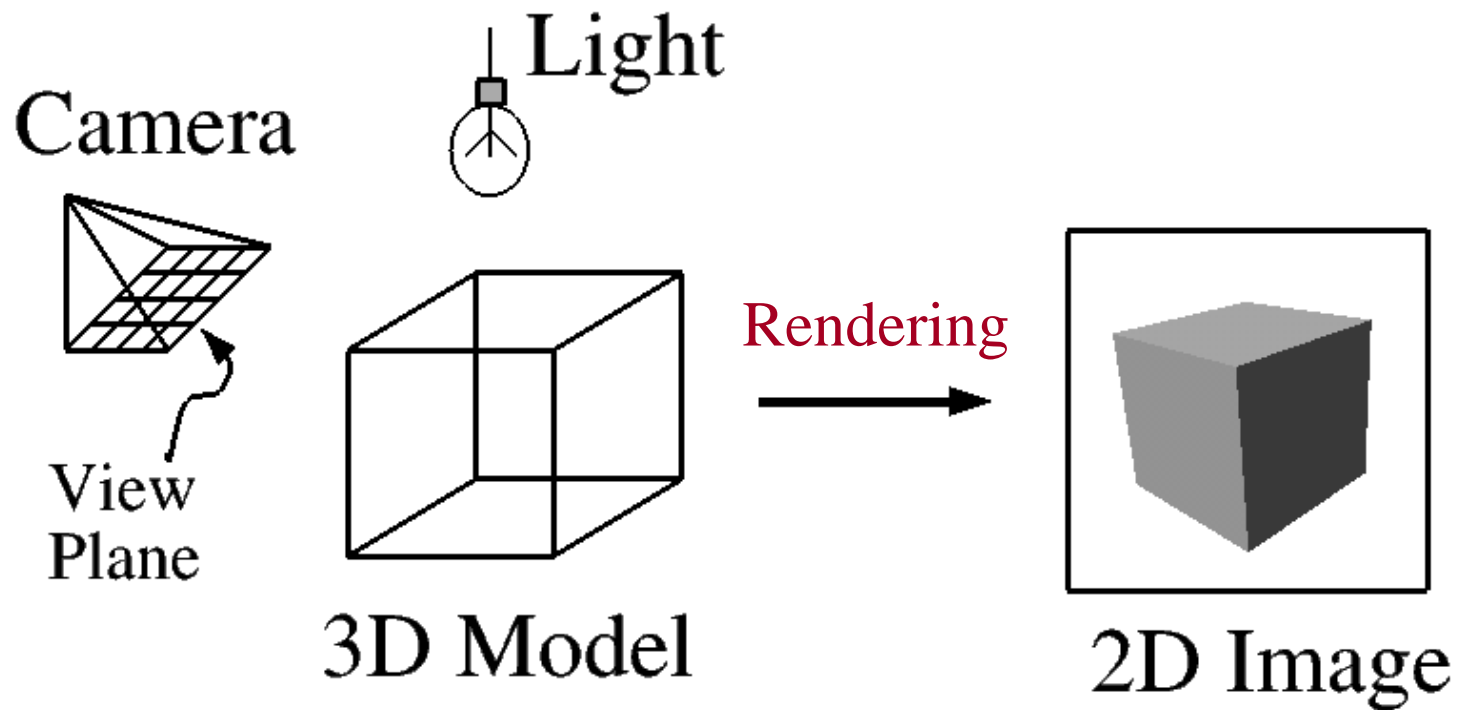
What is 3D Rendering?

- Topics in computer graphics
 - Imaging = *representing 2D images*
 - Modeling = *representing 3D objects*
 - Rendering = *constructing 2D images from 3D models*
 - Animation = *simulating changes over time*



What is 3D Rendering?

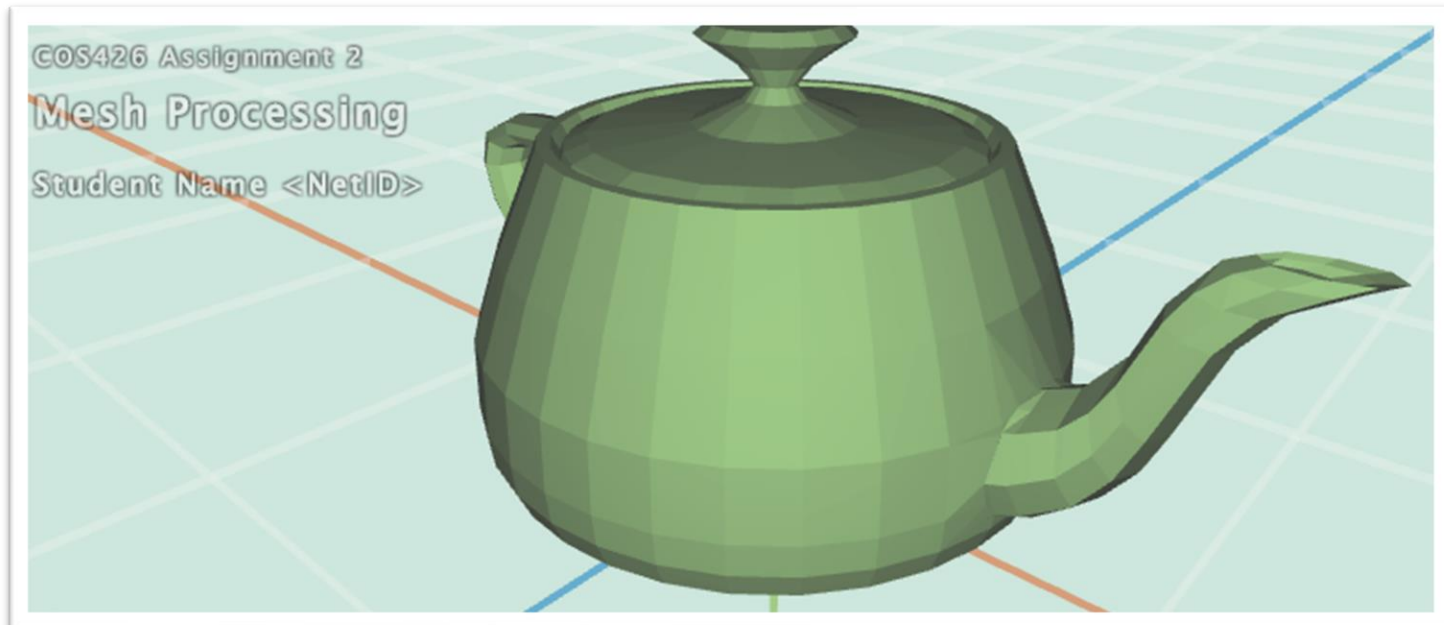
- Construct image from 3D model



Interactive 3D Rendering



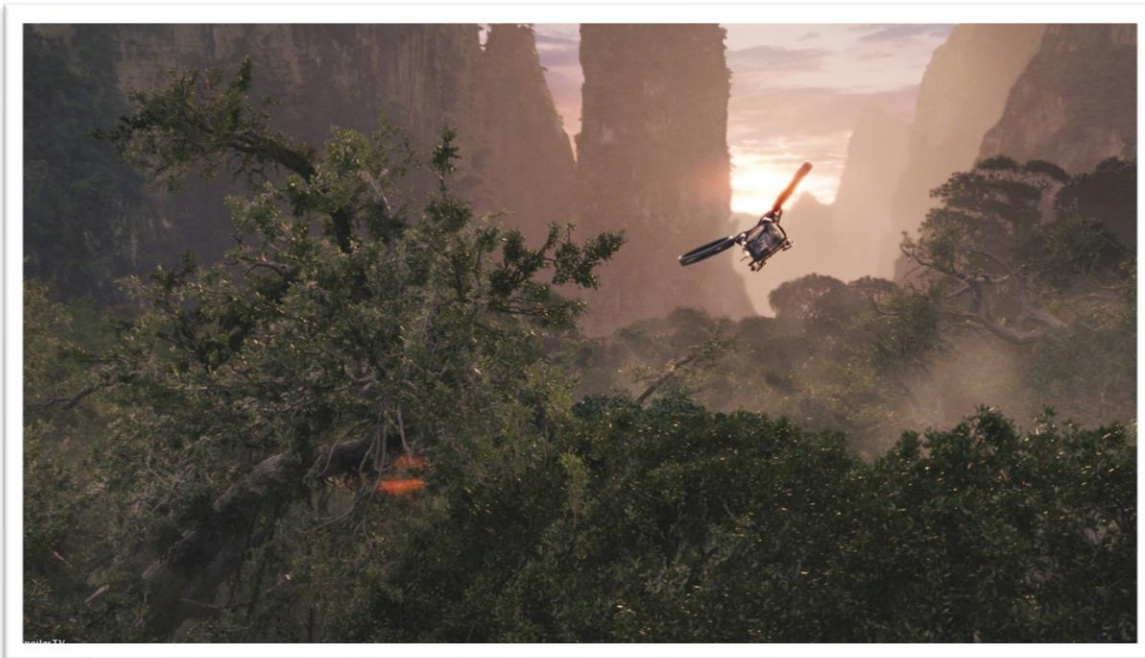
- Images generated in fraction of a second (e.g., 1/30) as user controls rendering parameters (e.g., camera)
 - » Achieve highest quality possible in given time
 - » Useful for visualization, games, etc.



Offline 3D Rendering



- One image generated with as much quality as possible for a particular set of rendering parameters
 - » Take as much time as is needed (minutes, hours...)
 - » Photorealism: movies, cut scenes, etc.



Avatar

3D Rendering Issues



- What issues must be addressed by a 3D rendering system?

Pixar





3D Rendering Issues

- What issues must be addressed by a 3D rendering system?
 - Camera
 - Visible surface determination
 - Lights
 - Reflectance
 - Shadows
 - Indirect illumination
 - Sampling
 - etc.

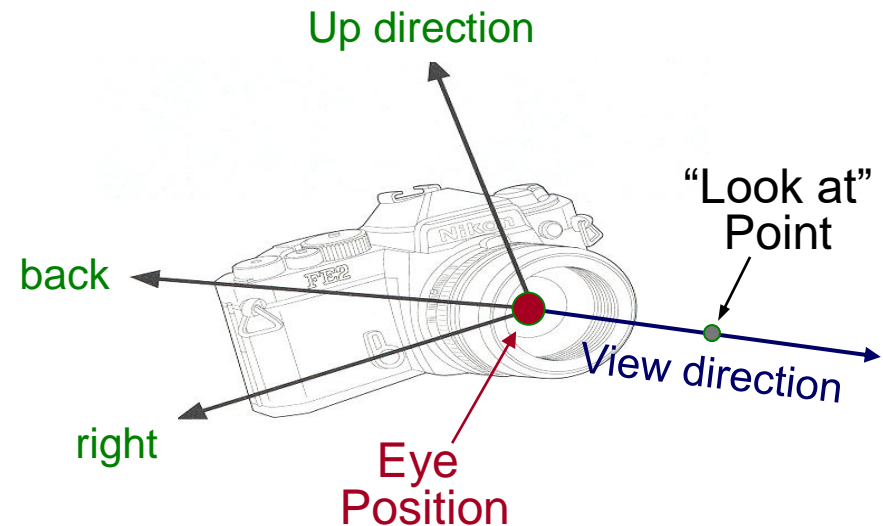


3D Rendering Issues

- What issues must be addressed by a 3D rendering system?
 - Camera
 - Visible surface determination
 - Lights
 - Reflectance
 - Shadows
 - Indirect illumination
 - Sampling
 - etc.

Pinhole Camera Parameters

- Position
 - Eye position (p_x, p_y, p_z)
- Orientation
 - View direction (d_x, d_y, d_z) or “look at” point
 - Up direction (u_x, u_y, u_z)
- Coverage
 - Field of view (fov_x, fov_y)
- Resolution
 - x and y





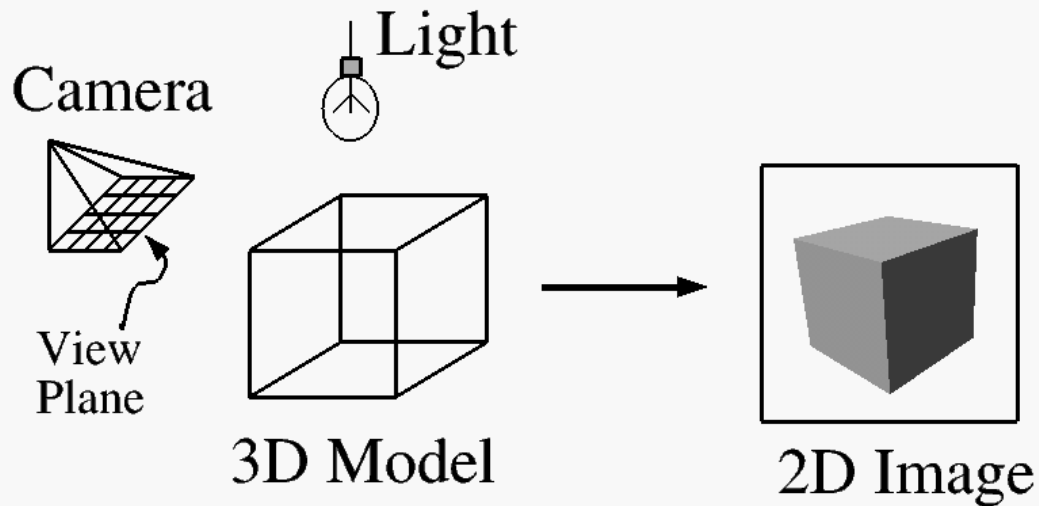
3D Rendering Issues

- What issues must be addressed by a 3D rendering system?
 - Camera
 - Visible surface determination
 - Lights
 - Reflectance
 - Shadows
 - Indirect illumination
 - Sampling
 - etc.

Visible Surface Determination

- The color of each pixel on the view plane depends on the radiance (“amount of light”) emanating from **visible** surfaces

How find visible surfaces?



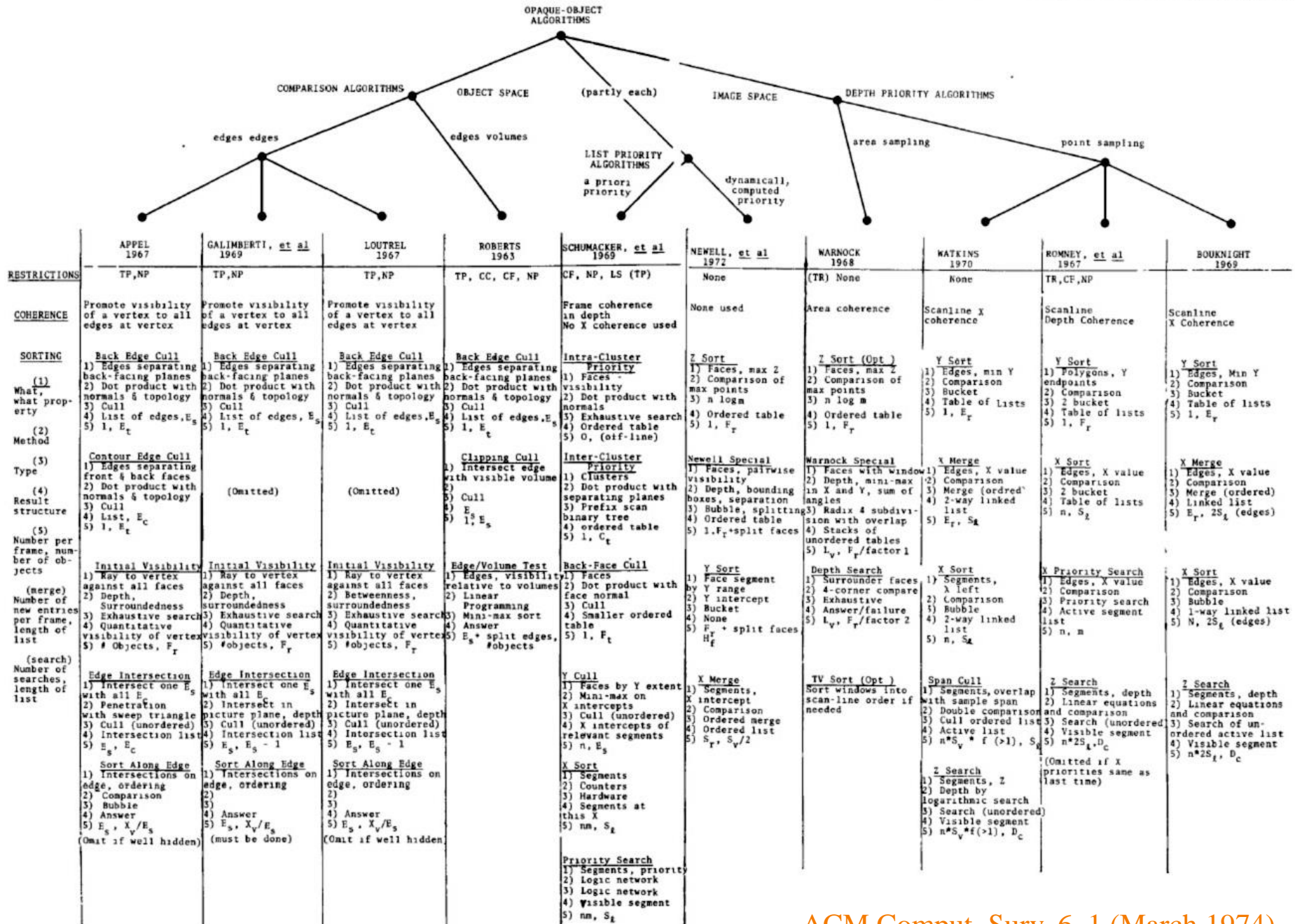


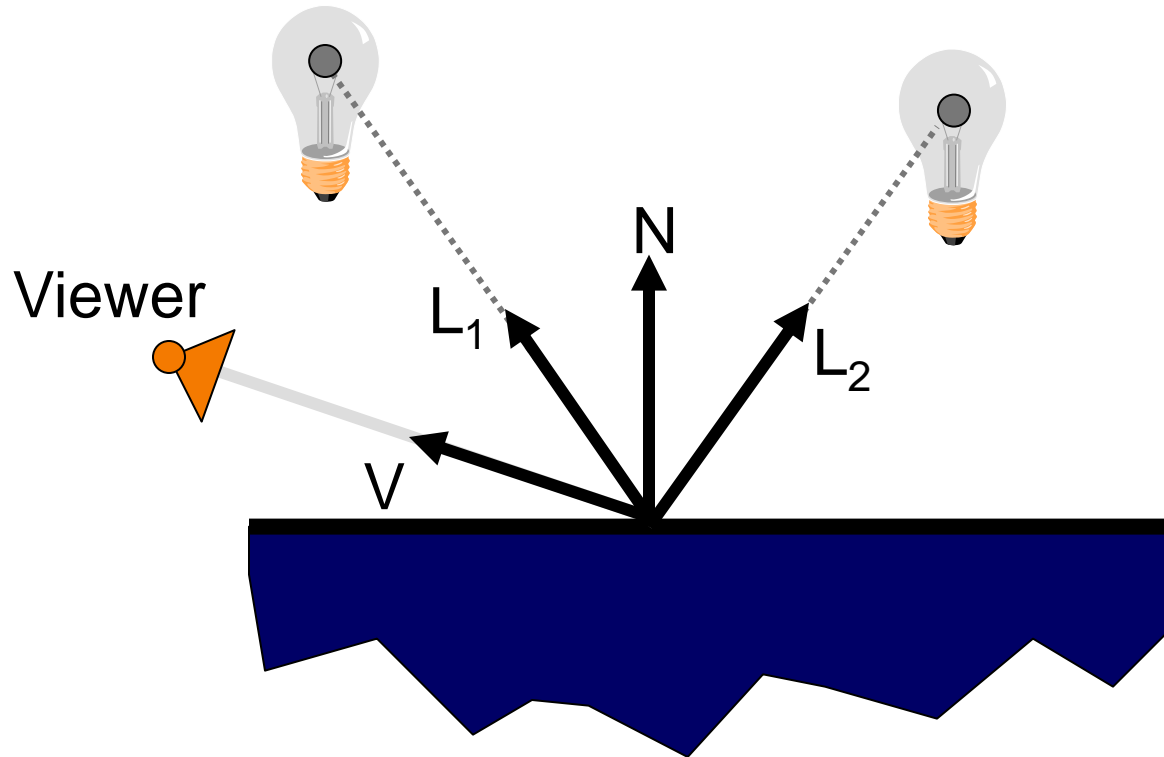
Figure 29. Characterization of ten opaque-object algorithms & Comparison of the algorithms.



3D Rendering Issues

- What issues must be addressed by a 3D rendering system?
 - Camera
 - Visible surface determination
 - Lights
 - Reflectance
 - Shadows
 - Indirect illumination
 - Sampling
 - etc.

Lighting Simulation





3D Rendering Issues

- What issues must be addressed by a 3D rendering system?
 - Camera
 - Visible surface determination
 - Lights
 - Reflectance
 - **Shadows**
 - Indirect illumination
 - Sampling
 - etc.

Shadows



- Occlusions from light sources
 - Soft shadows with area light source

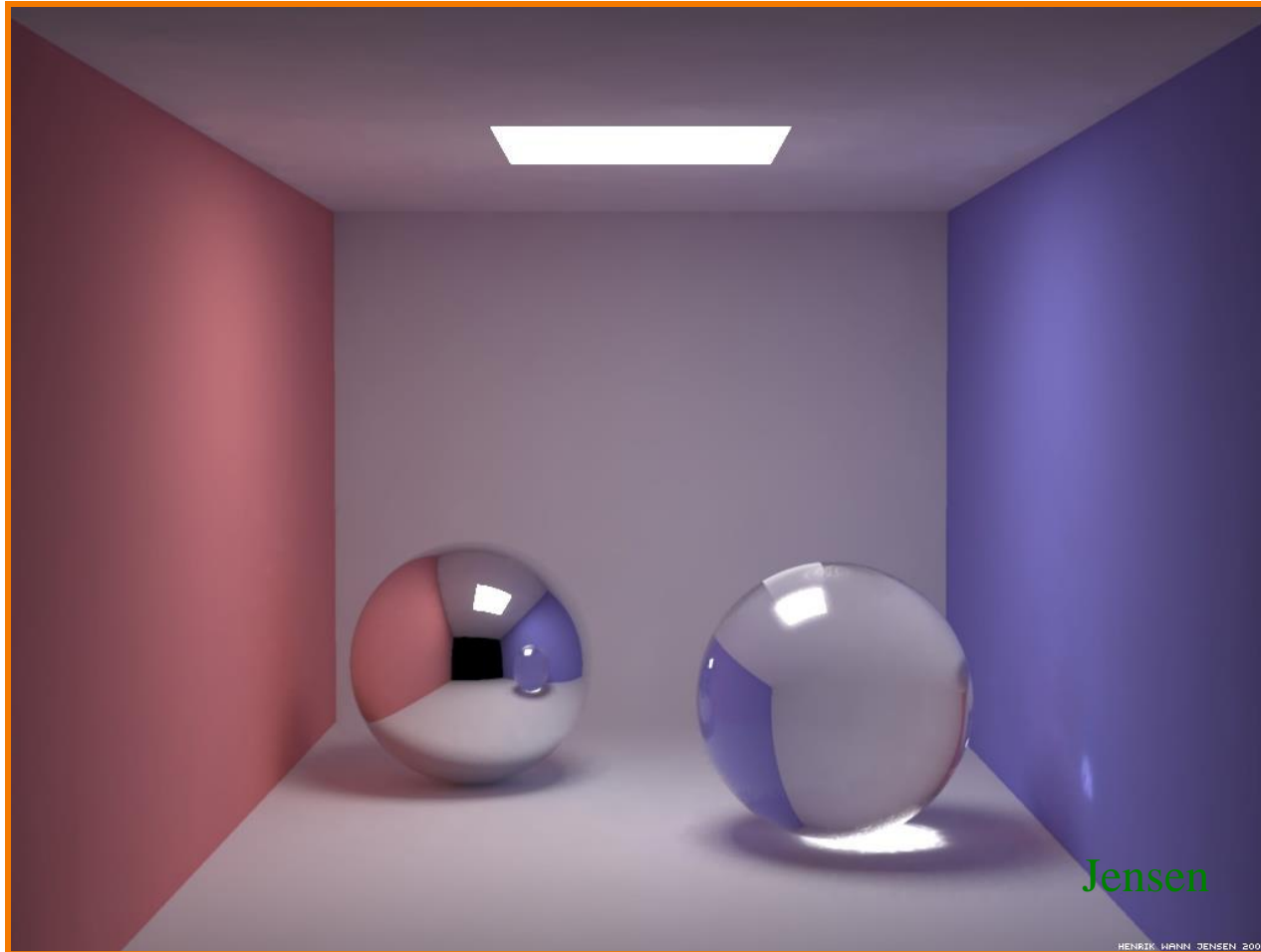




3D Rendering Issues

- What issues must be addressed by a 3D rendering system?
 - Camera
 - Visible surface determination
 - Lights
 - Reflectance
 - Shadows
 - Indirect illumination
 - Sampling
 - etc.

Indirect Illumination





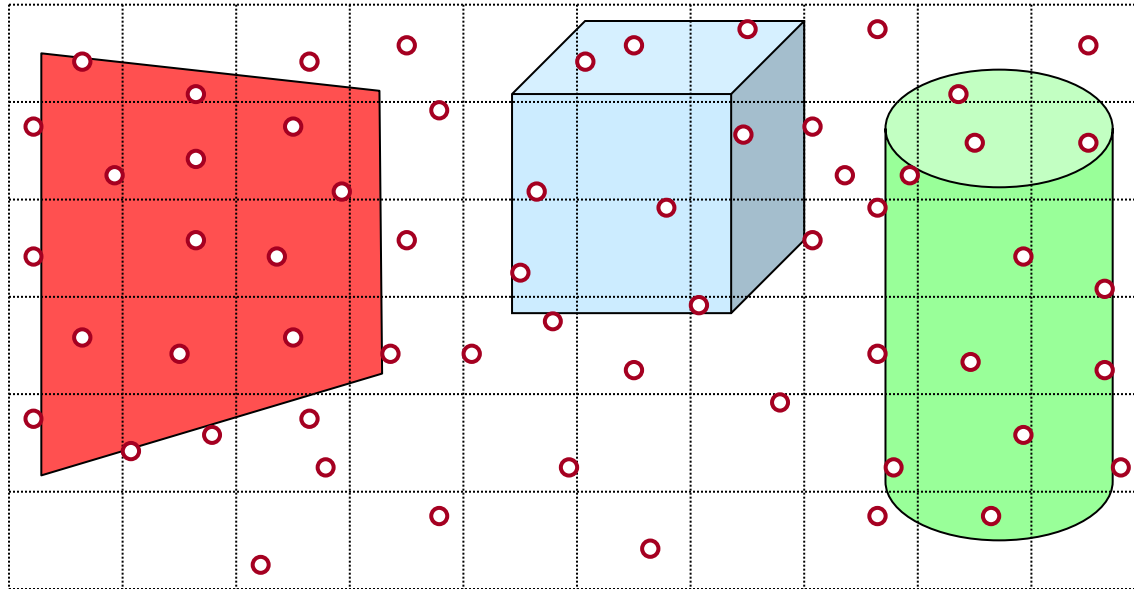
3D Rendering Issues

- What issues must be addressed by a 3D rendering system?
 - Camera
 - Visible surface determination
 - Shadows
 - Reflectance
 - Indirect illumination
 - **Sampling**
 - etc.

Sampling



- Scene can be sampled with any ray
 - Rendering is a problem in sampling and reconstruction





Rendering Method I: Ray Casting

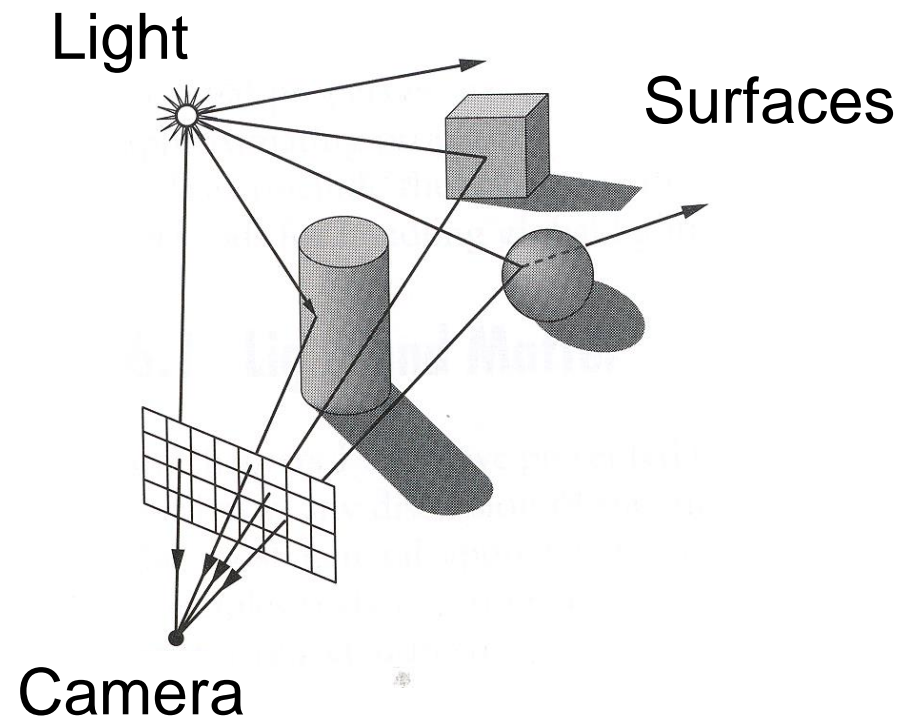
Heckbert's Business Card Ray Tracer



- ```
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
("%0.f %0.f %0.f\n",U);}/*minray!*/
```

# Ray Casting

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

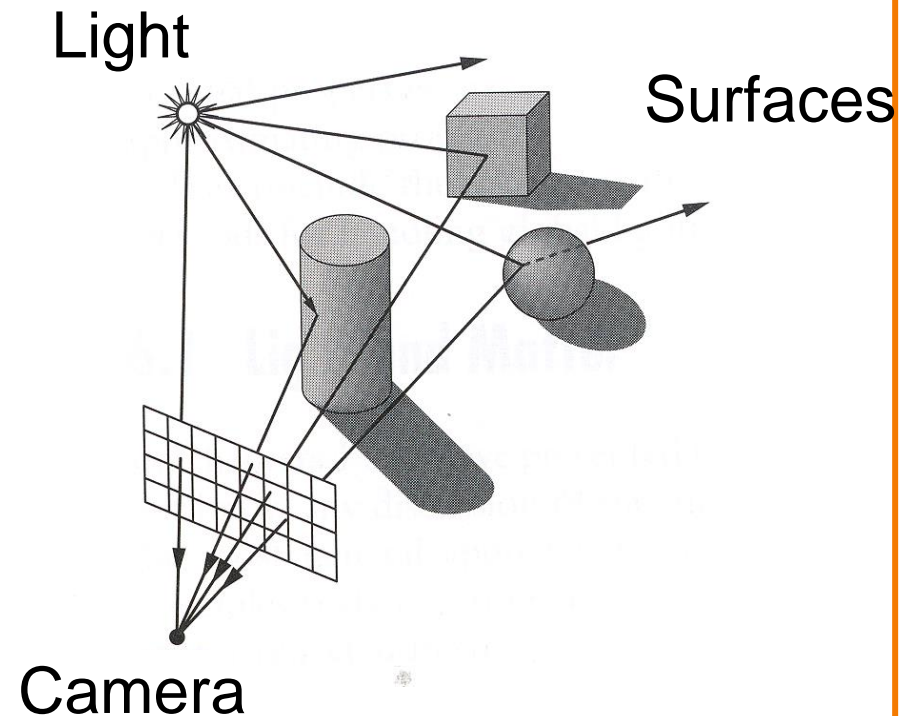


# Scene



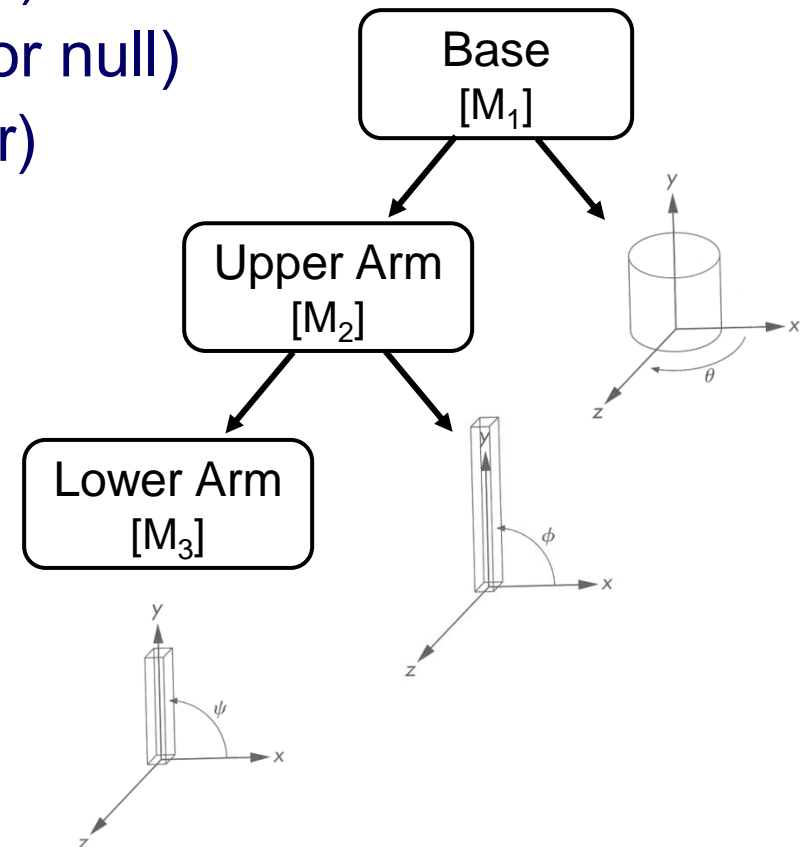
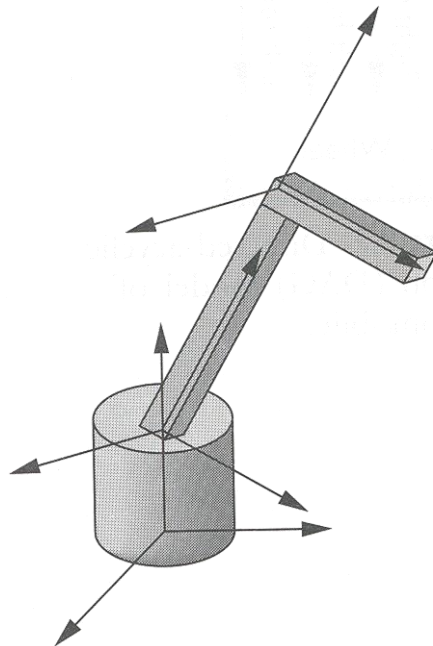
- Scene has:
  - Scene graph with surface primitives
  - Set of lights
  - Camera

```
struct R3Scene {
 R3Node *root;
 vector<R3Light *> lights;
 R3Camera camera;
 R3Box bbox;
 R3Rgb background;
 R3Rgb ambient;
};
```



# Scene Graph

- Scene graph is hierarchy of nodes, each with:
  - Bounding box (in node's coordinate system)
  - Transformation (4x4 matrix)
  - Shape (mesh, sphere, ... or null)
  - Material (more on this later)



# Scene Graph



- Simple scene graph implementation:

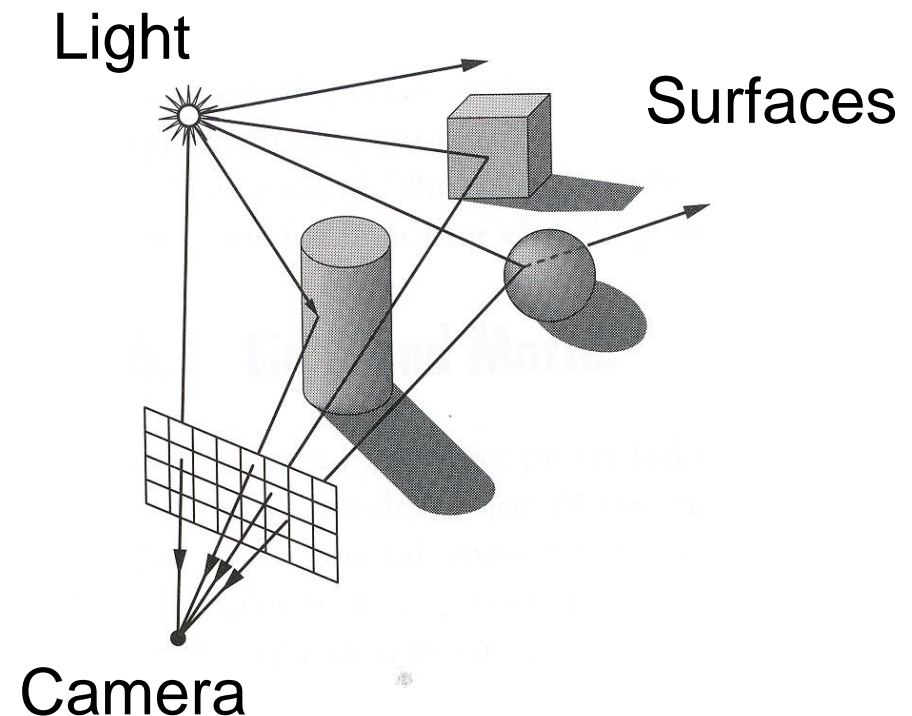
```
struct R3Node {
 struct R3Node *parent;
 vector<struct R3Node *> children;
 R3Shape *shape;
 R3Matrix transformation;
 R3Material *material;
 R3Box bbox;
};
```

```
struct R3Shape {
 R3ShapeType type;
 R3Box *box;
 R3Sphere *sphere;
 R3Cylinder *cylinder;
 R3Cone *cone;
 R3Mesh *mesh;
};
```

# Ray Casting



- For each sample (pixel) ...
  - Construct ray from eye position through view plane
  - Compute radiance leaving first point of intersection between ray and scene



# Ray Casting



- Simple implementation:

```
R2Image *RayCast(R3Scene *scene, int width, int height)
{
 R2Image *image = new R2Image(width, height);
 for (int i = 0; i < width; i++) {
 for (int j = 0; j < height; j++) {

 image->SetPixel(i, j, radiance);
 }
 }
 return image;
}
```



# Ray Casting



- Simple implementation:

```
R2Image *RayCast(R3Scene *scene, int width, int height)
{
 R2Image *image = new R2Image(width, height);
 for (int i = 0; i < width; i++) {
 for (int j = 0; j < height; j++) {
 R3Ray ray = ConstructRayThroughPixel(scene->camera, i, j);

 image->SetPixel(i, j, radiance);
 }
 }
 return image;
}
```

# Ray Casting



- Simple implementation:

```
R2Image *RayCast(R3Scene *scene, int width, int height)
{
 R2Image *image = new R2Image(width, height);
 for (int i = 0; i < width; i++) {
 for (int j = 0; j < height; j++) {
 R3Ray ray = ConstructRayThroughPixel(scene->camera, i, j);
 R3Rgb radiance = ComputeRadiance(scene, &ray);
 image->SetPixel(i, j, radiance);
 }
 }
 return image;
}
```

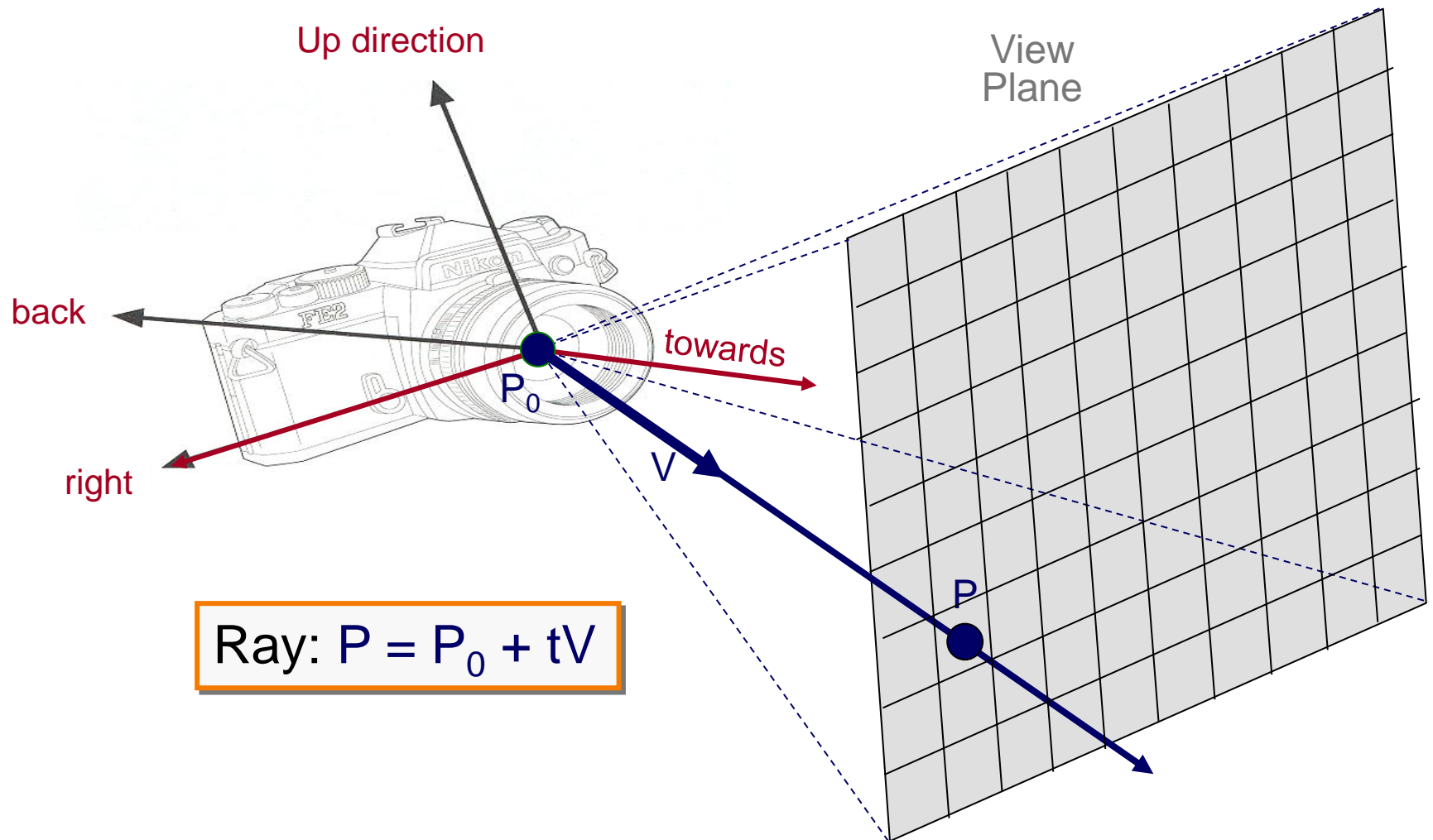
# Ray Casting



- Simple implementation:

```
R2Image *RayCast(R3Scene *scene, int width, int height)
{
 R2Image *image = new R2Image(width, height);
 for (int i = 0; i < width; i++) {
 for (int j = 0; j < height; j++) {
 R3Ray ray = ConstructRayThroughPixel(scene->camera, i, j);
 R3Rgb radiance = ComputeRadiance(scene, &ray);
 image->SetPixel(i, j, radiance);
 }
 }
 return image;
}
```

# Constructing Ray Through a Pixel



# Constructing Ray Through a Pixel



- 2D Example

$\Theta$  = frustum **half**-angle

$d$  = distance to view plane

$\text{right} = \text{towards} \times \text{up}$

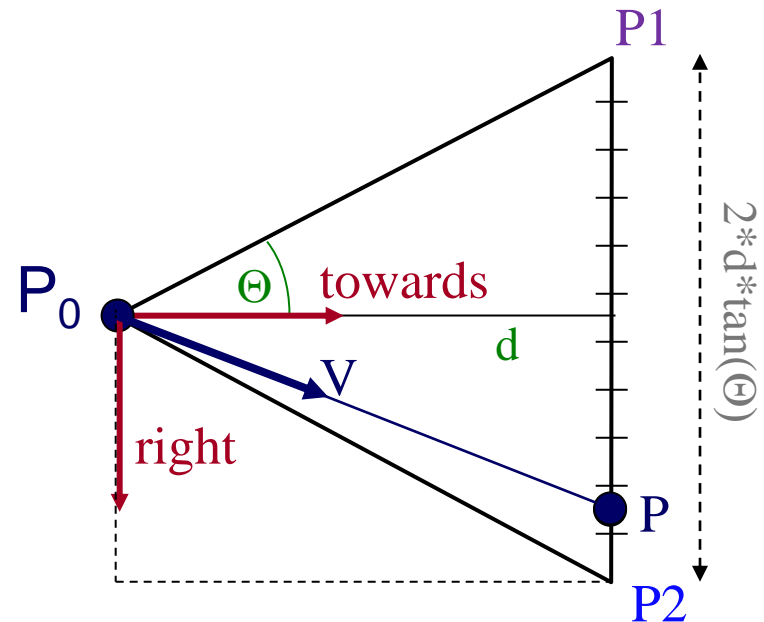
$$P1 = P_0 + d * \text{towards} - d * \tan(\Theta) * \text{right}$$

$$P2 = P_0 + d * \text{towards} + d * \tan(\Theta) * \text{right}$$

$$P = P1 + ((i + 0.5) / \text{width}) * (P2 - P1)$$

$$V = (P - P_0) / \|P - P_0\|$$

( $d$  cancels out...)



$$\text{Ray: } P = P_0 + tV$$

# Ray Casting



- Simple implementation:

```
R2Image *RayCast(R3Scene *scene, int width, int height)
{
 R2Image *image = new R2Image(width, height);
 for (int i = 0; i < width; i++) {
 for (int j = 0; j < height; j++) {
 R3Ray ray = ConstructRayThroughPixel(scene->camera, i, j);
 R3Rgb radiance = ComputeRadiance(scene, &ray);
 image->SetPixel(i, j, radiance);
 }
 }
 return image;
}
```

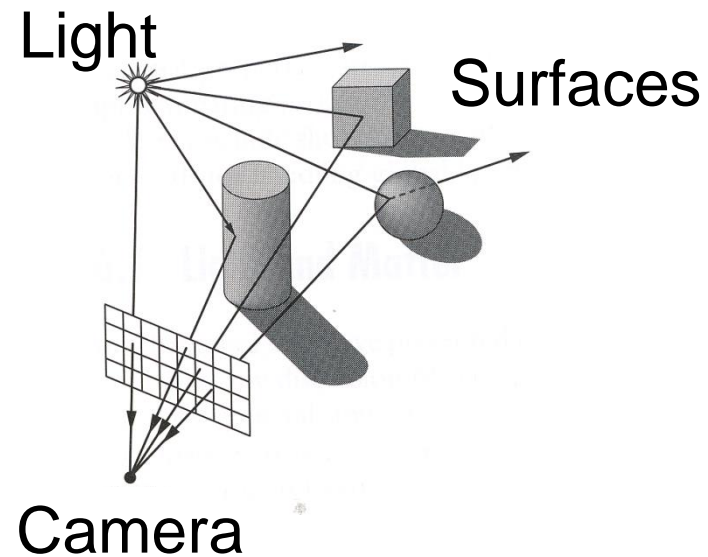


# Ray Casting

- Simple implementation:

```
R3Rgb ComputeRadiance(R3Scene *scene, R3Ray *ray)
{
 R3Intersection intersection = ComputeIntersection(scene, ray);
}
```

```
struct R3Intersection {
 bool hit;
 R3Node *node;
 R3Point position;
 R3Vector normal;
 double t;
};
```



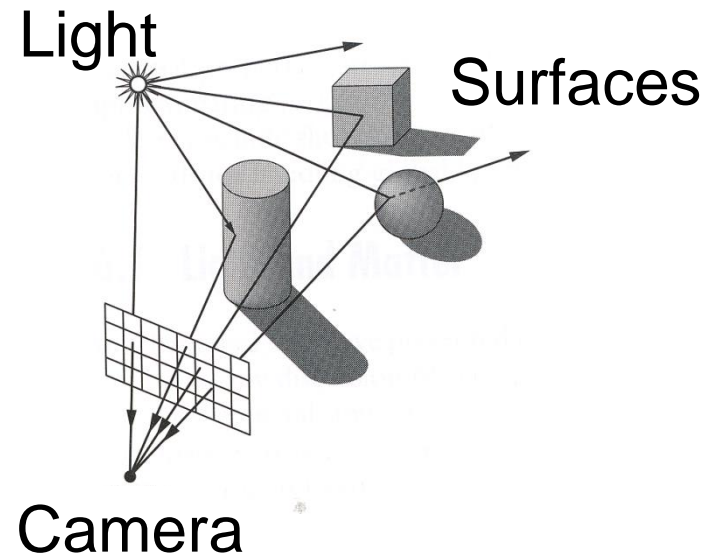


# Ray Casting

- Simple implementation:

```
R3Rgb ComputeRadiance(R3Scene *scene, R3Ray *ray)
{
 R3Intersection intersection = ComputeIntersection(scene, ray);
 return ComputeRadiance(scene, ray, intersection);
}
```

```
struct R3Intersection {
 bool hit;
 R3Node *node;
 R3Point position;
 R3Vector normal;
 double t;
};
```





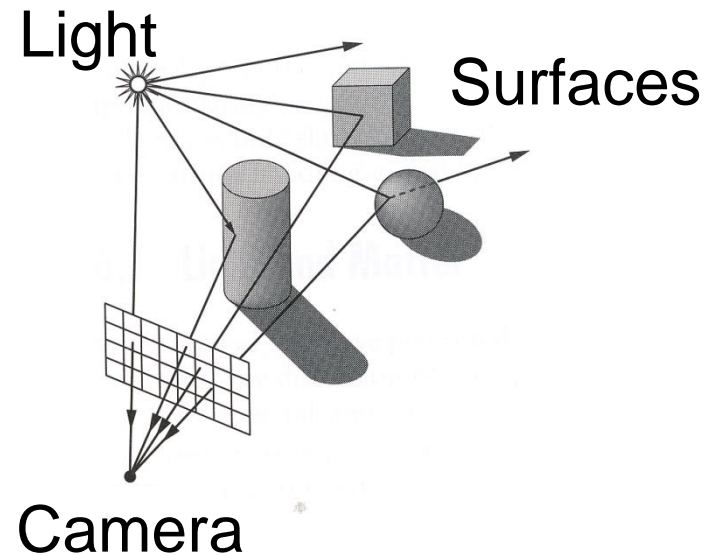


# Ray Casting

- Simple implementation:

```
R3Rgb ComputeRadiance(R3Scene *scene, R3Ray *ray)
{
 R3Intersection intersection = ComputeIntersection(scene, ray);
 return ComputeRadiance(scene, ray, intersection);
}
```

```
struct R3Intersection {
 bool hit;
 R3Node *node;
 R3Point position;
 R3Vector normal;
 double t;
};
```





# Ray Intersection

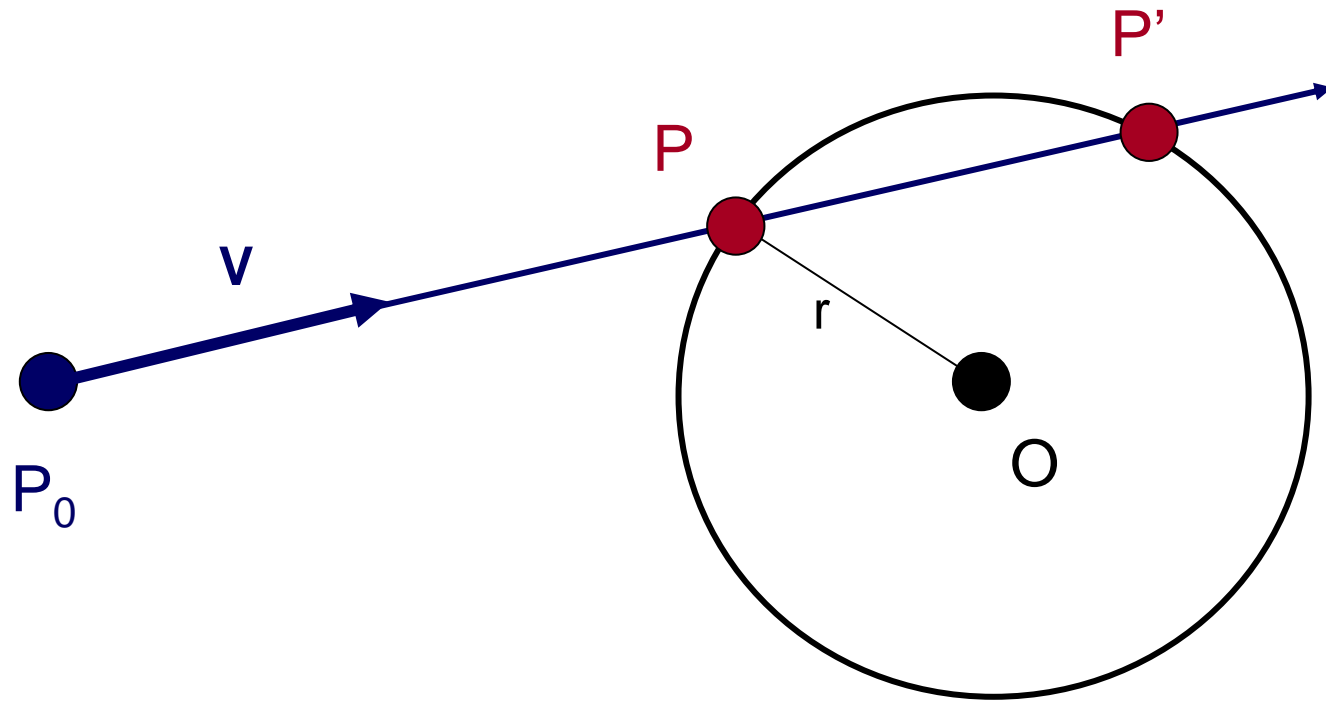
- Ray Intersection
  - Sphere
  - Triangle
  - Box
  - Scene
  
- Ray Intersection Acceleration
  - Bounding volumes
  - Uniform grids
  - Octrees
  - BSP trees



# Ray Intersection

- Ray Intersection
  - Sphere
  - Triangle
  - Box
  - Scene
- Ray Intersection Acceleration
  - Bounding volumes
  - Uniform grids
  - Octrees
  - BSP trees

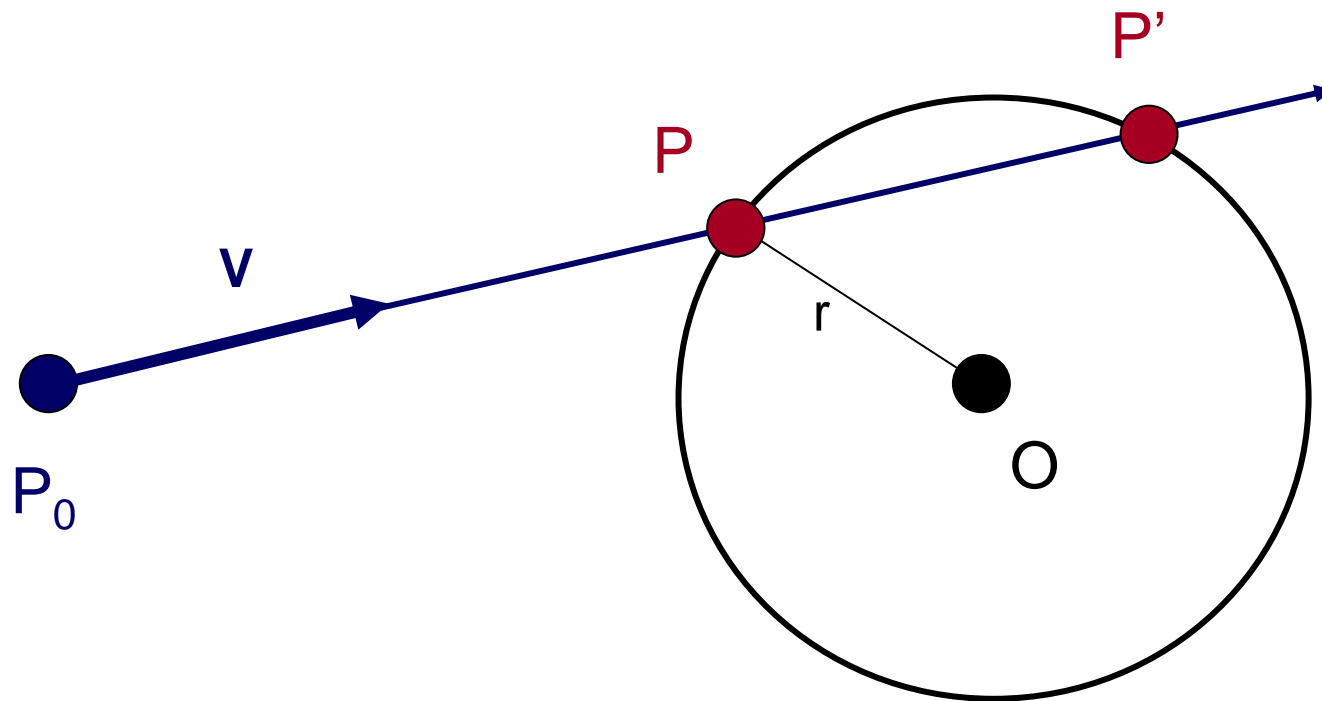
# Ray-Sphere Intersection



# Ray-Sphere Intersection

Ray:  $P = P_0 + tV$

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





# Ray-Sphere Intersection I

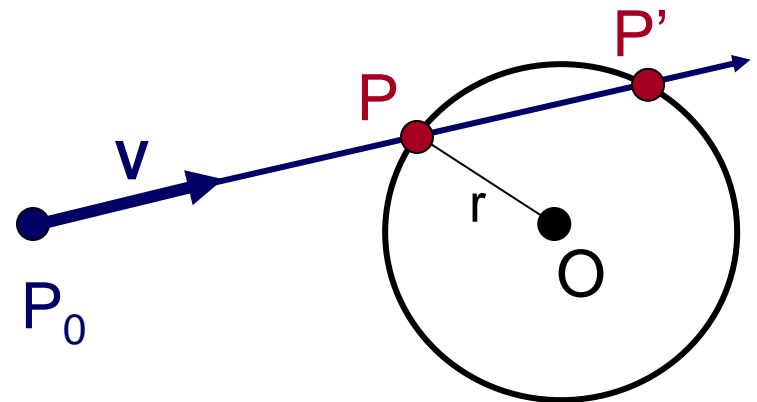
Ray:  $P = P_0 + tV$

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

Algebraic Method

Substituting for  $P$ , we get:

$$|P_0 + tV - O|^2 - r^2 = 0$$



$$P = P_0 + tV$$



# Ray-Sphere Intersection I

Ray:  $P = P_0 + tV$

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

Algebraic Method

Substituting for  $P$ , we get:

$$|P_0 + tV - O|^2 - r^2 = 0$$

Solve quadratic equation:

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

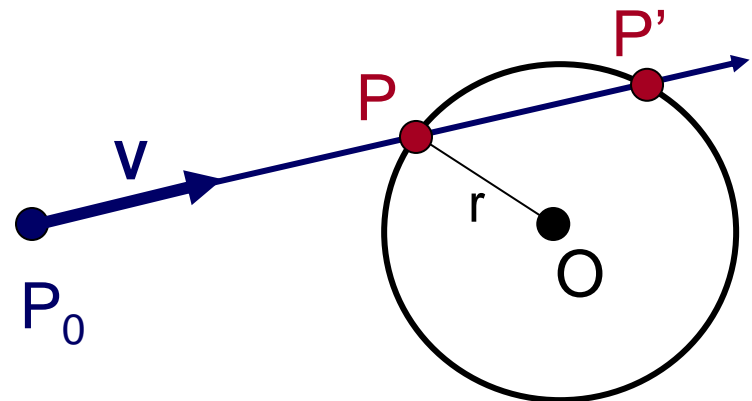
where:

$$a = V^2$$

$$b = 2V \cdot (P_0 - O)$$

$$c = |P_0 - O|^2 - r^2 = 0$$

$$P = P_0 + tV$$





# Ray-Sphere Intersection II

Ray:  $P = P_0 + tV$

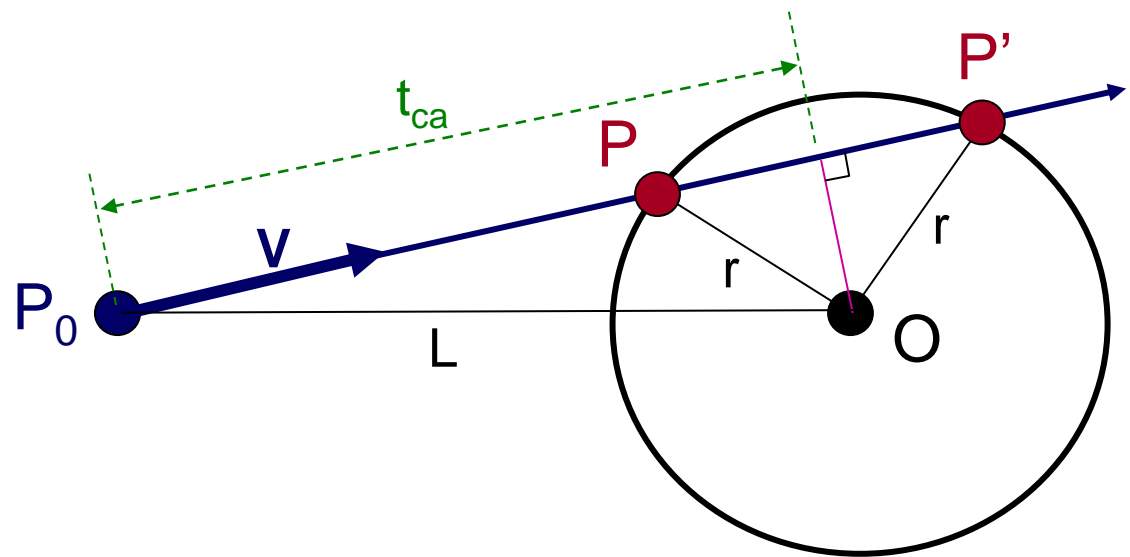
Sphere:  $|P - O|^2 - r^2 = 0$

Geometric Method

$L = O - P_0$

$t_{ca} = L \cdot V$

if ( $t_{ca} < 0$ ) return INF



$P = P_0 + tV$





# Ray-Sphere Intersection II

Ray:  $P = P_0 + tV$

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

Geometric Method

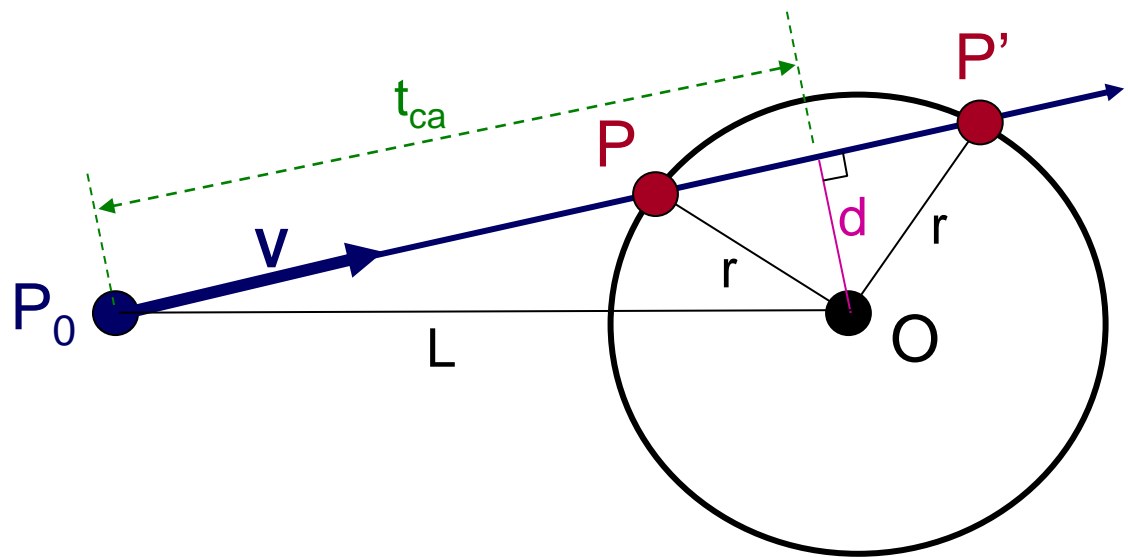
$L = O - P_0$

$t_{ca} = L \cdot V$

if ( $t_{ca} < 0$ ) return INF

$d^2 = L \cdot L - t_{ca}^2$

if ( $d^2 > r^2$ ) return INF



$P = P_0 + tV$



# Ray-Sphere Intersection II

$$\text{Ray: } P = P_0 + tV$$

$$\text{Sphere: } |P - O|^2 - r^2 = 0$$

Geometric Method

$$L = O - P_0$$

$$t_{ca} = L \cdot V$$

if ( $t_{ca} < 0$ ) return INF

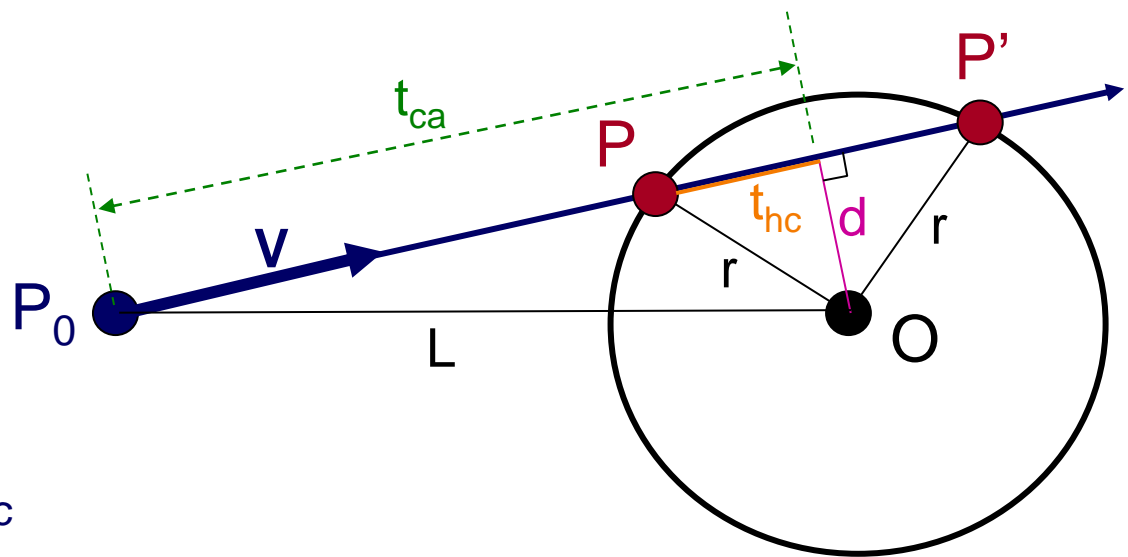
$$d^2 = L \cdot L - t_{ca}^2$$

if ( $d^2 > r^2$ ) return INF

$$t_{hc} = \text{sqrt}(r^2 - d^2)$$

$$t = t_{ca} - t_{hc} \text{ and } t_{ca} + t_{hc}$$

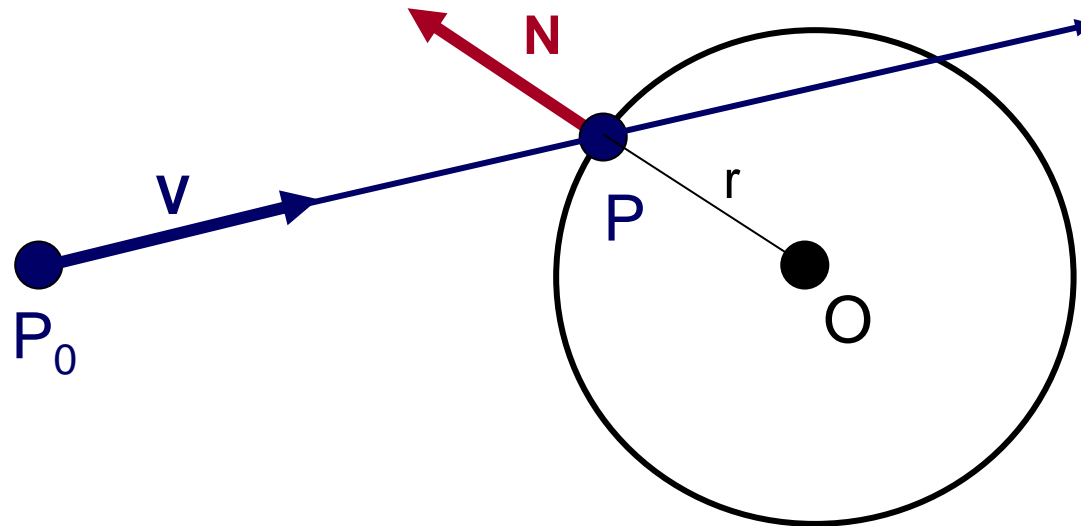
$$P = P_0 + tV$$



# Ray-Sphere Intersection

- Need normal vector at intersection for lighting calculations (next lecture)

$$N = (P - O) / \|P - O\|$$

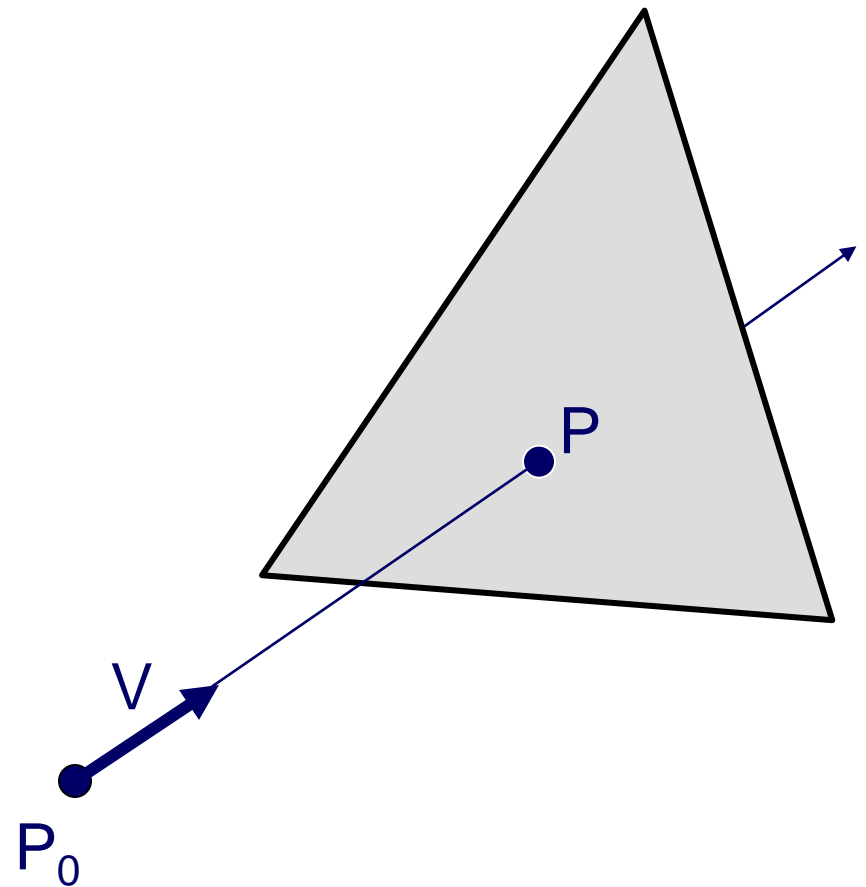




# Ray Intersection

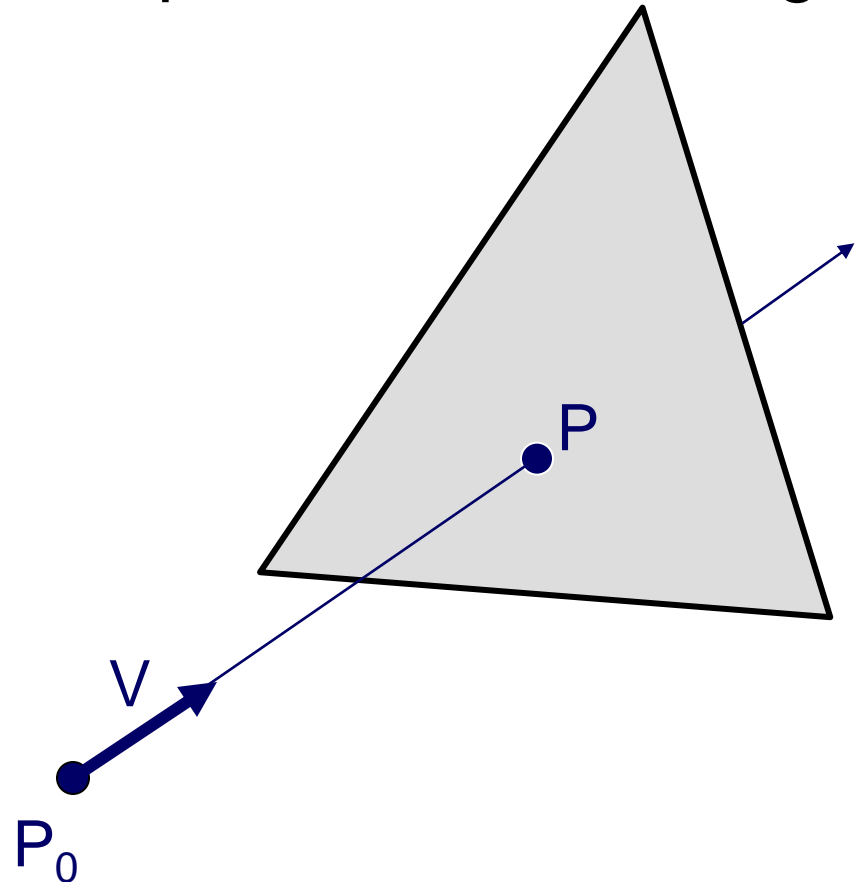
- Ray Intersection
  - Sphere
  - Triangle
  - Box
  - Scene
- Ray Intersection Acceleration
  - Bounding volumes
  - Uniform grids
  - Octrees
  - BSP trees

# Ray-Triangle Intersection



# Ray-Triangle Intersection

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





# Ray-Plane Intersection

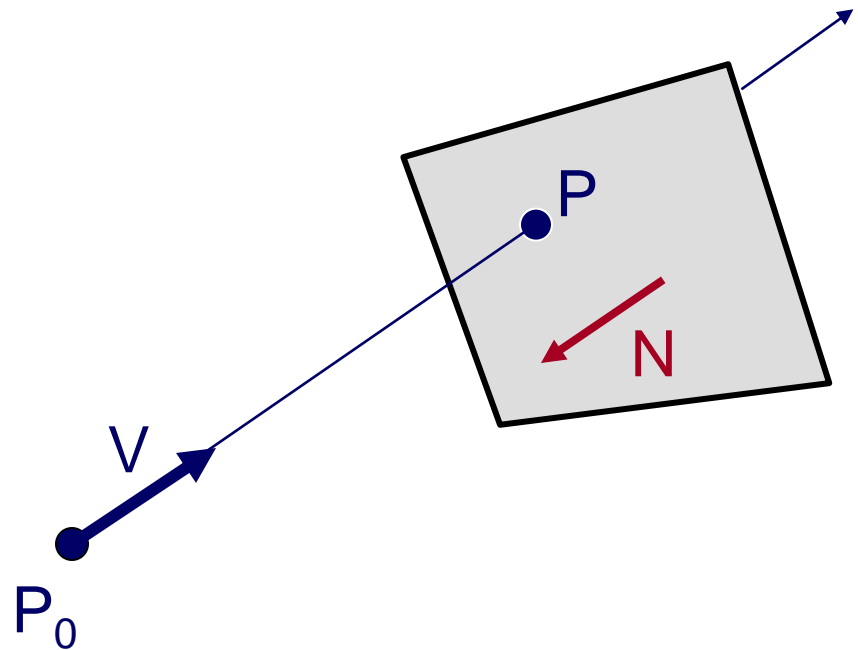
Ray:  $P = P_0 + tV$

Plane:  $P \cdot N + d = 0$

Substituting for  $P$ , we get:

$$(P_0 + tV) \cdot N + d = 0$$

Algebraic Method





# Ray-Plane Intersection

$$\text{Ray: } P = P_0 + tV$$

$$\text{Plane: } P \cdot N + d = 0$$

Algebraic Method

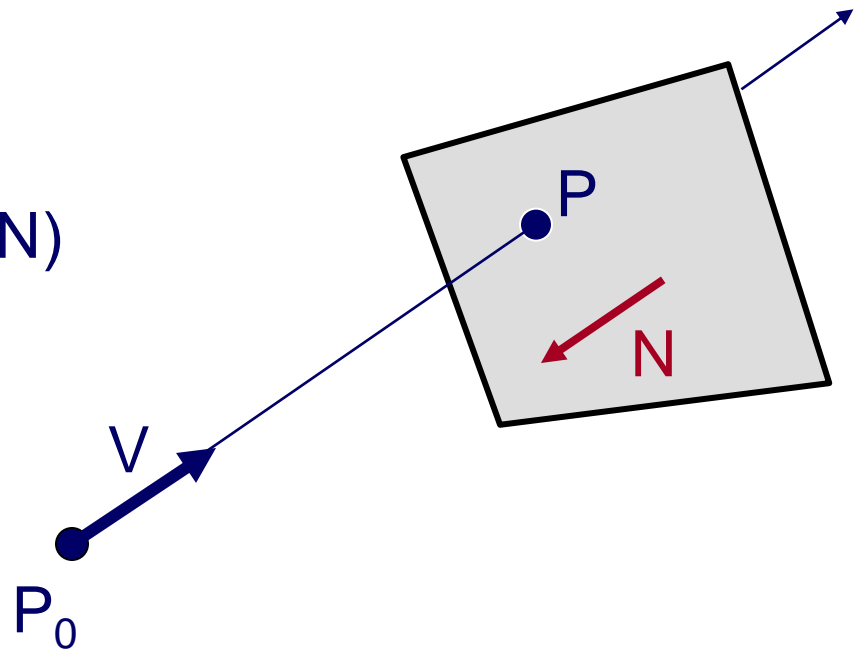
Substituting for  $P$ , we get:

$$(P_0 + tV) \cdot N + d = 0$$

Solution:

$$t = -(P_0 \cdot N + d) / (V \cdot N)$$

$$P = P_0 + tV$$





# Ray-Triangle Intersection I

- Check if point is inside triangle algebraically

For each side of triangle

$$V_1 = T_1 - P_0$$

$$V_2 = T_2 - P_0$$

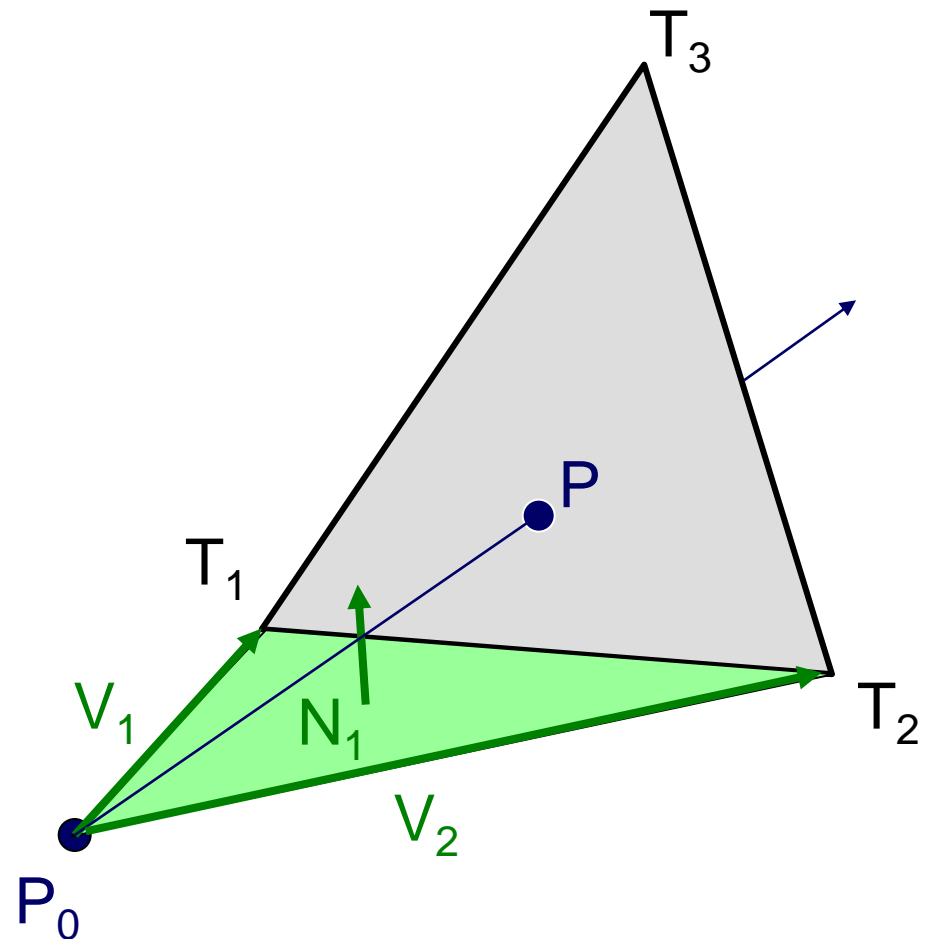
$$N_1 = V_2 \times V_1$$

Normalize  $N_1$

Plane  $p(P_0, N_1)$

end

return TRUE



# Ray-Triangle Intersection I

- Check if point is inside triangle algebraically

For each side of triangle

$$V_1 = T_1 - P_0$$

$$V_2 = T_2 - P_0$$

$$N_1 = V_2 \times V_1$$

Normalize  $N_1$

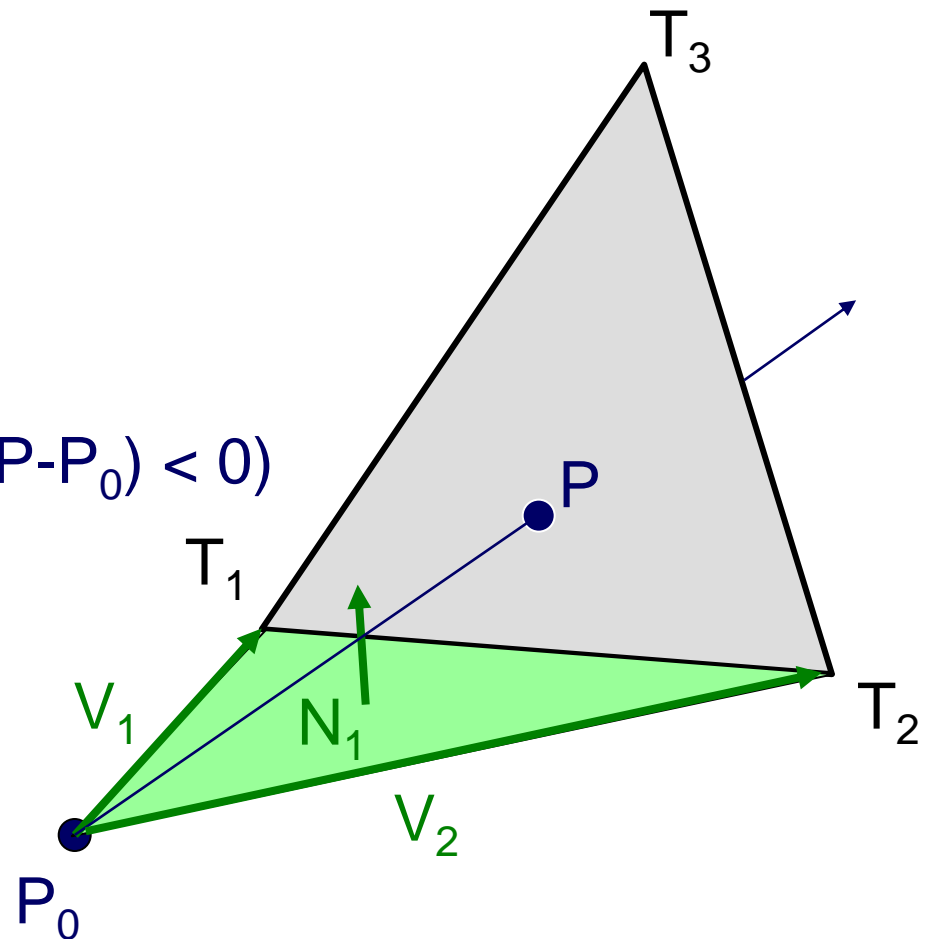
Plane  $p(P_0, N_1)$

if (SignedDistance( $p$ ,  $P - P_0$ ) < 0)

return FALSE

end

return TRUE





# Ray-Triangle Intersection II

- Check if point is inside triangle algebraically

For each side of triangle

$$V_1 = T_1 - P$$

$$V_2 = T_2 - P$$

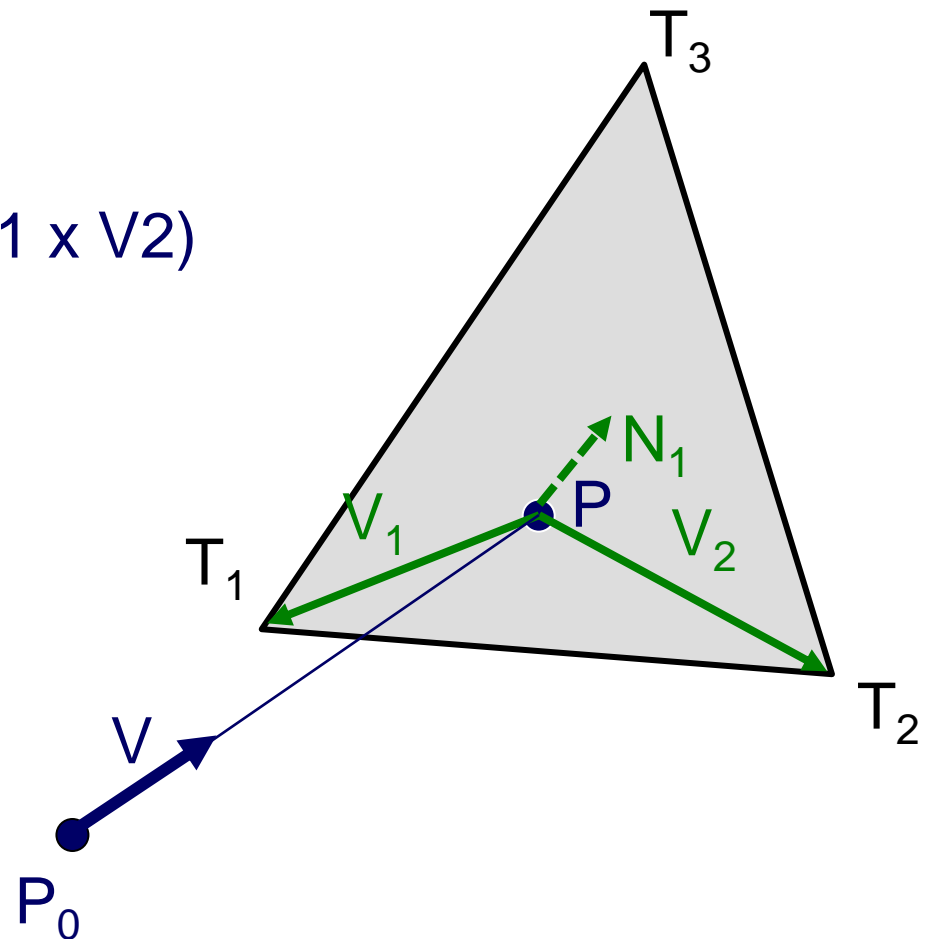
$$N_1 = V_2 \times V_1 \text{ (but not } V_1 \times V_2)$$

$$\text{if } (V \cdot N_1 < 0)$$

return FALSE

end

return TRUE





# Ray-Triangle Intersection II

- Check if point is inside triangle algebraically

For each side of triangle

$$V_1 = T_1 - P$$

$$V_2 = T_2 - P$$

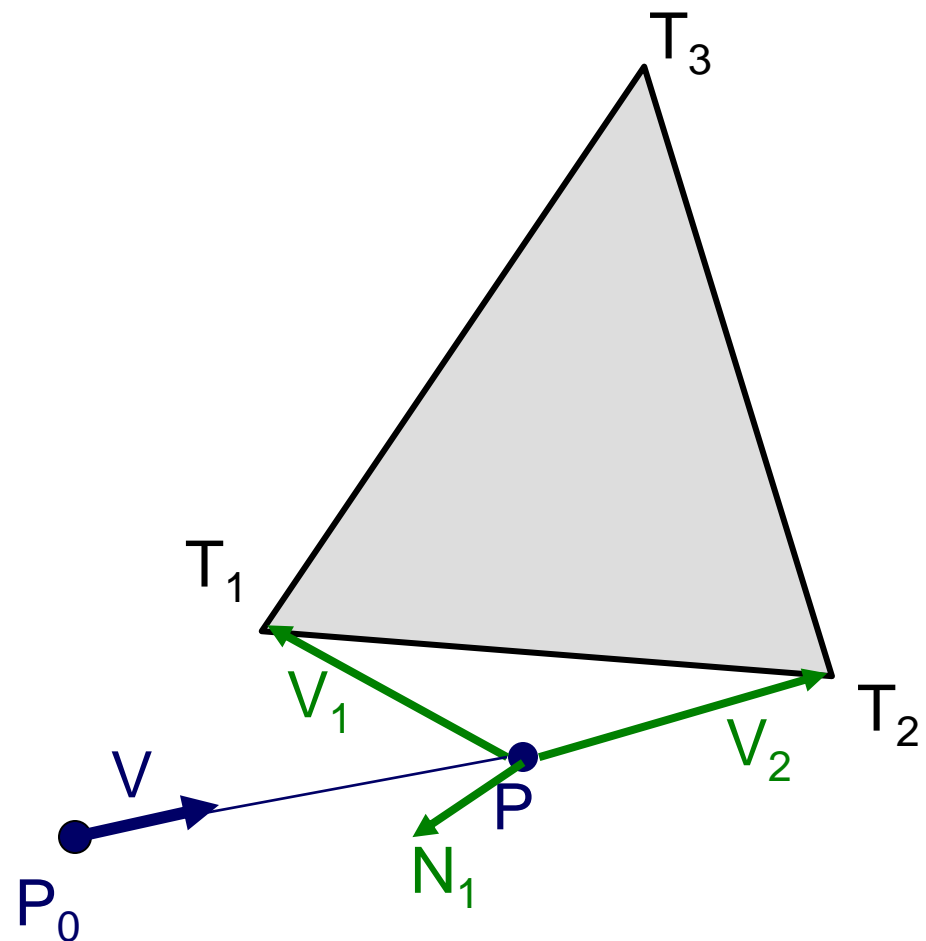
$$N_1 = V_2 \times V_1$$

$$\text{if } (V \cdot N_1 < 0)$$

return FALSE

end

return TRUE





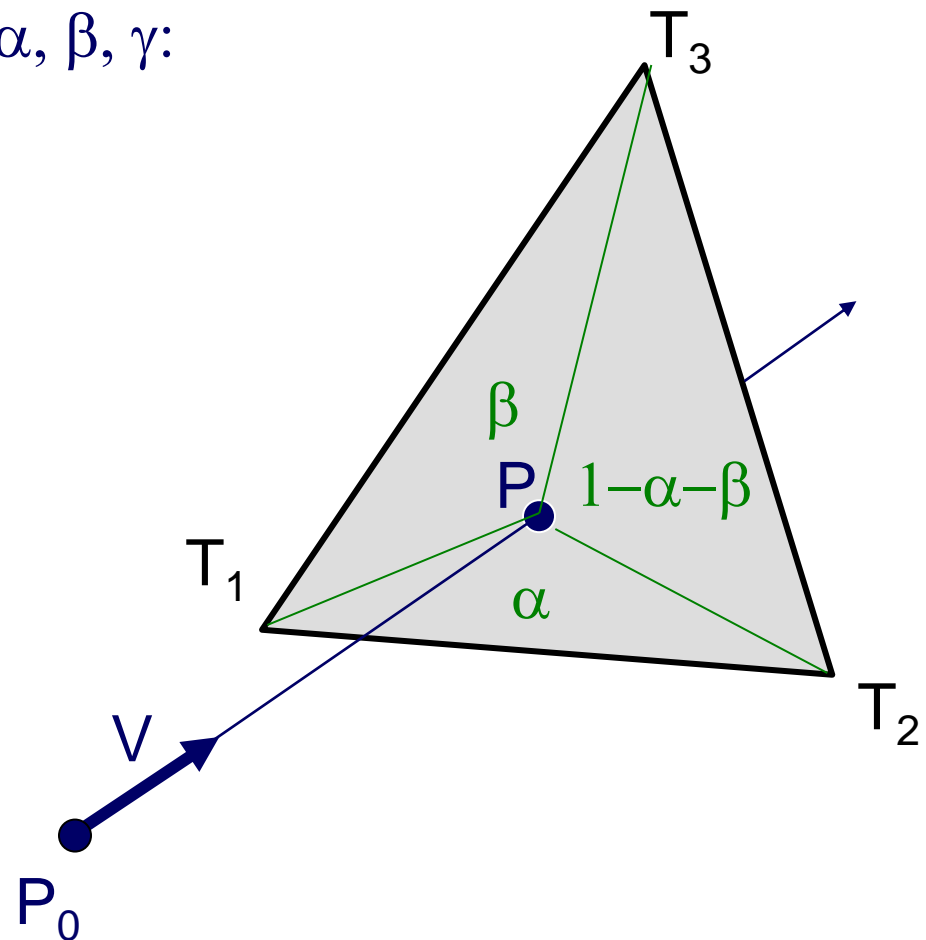
# Ray-Triangle Intersection III

- Check if point is inside triangle parametrically

“Barycentric coordinates”  $\alpha, \beta, \gamma$ :

$$P = \alpha T_3 + \beta T_2 + \gamma T_1$$

where  $\alpha + \beta + \gamma = 1$





# Ray-Triangle Intersection III

- Check if point is inside triangle parametrically

“Barycentric coordinates”  $\alpha, \beta, \gamma$ :

$$P = \alpha T_3 + \beta T_2 + \gamma T_1$$

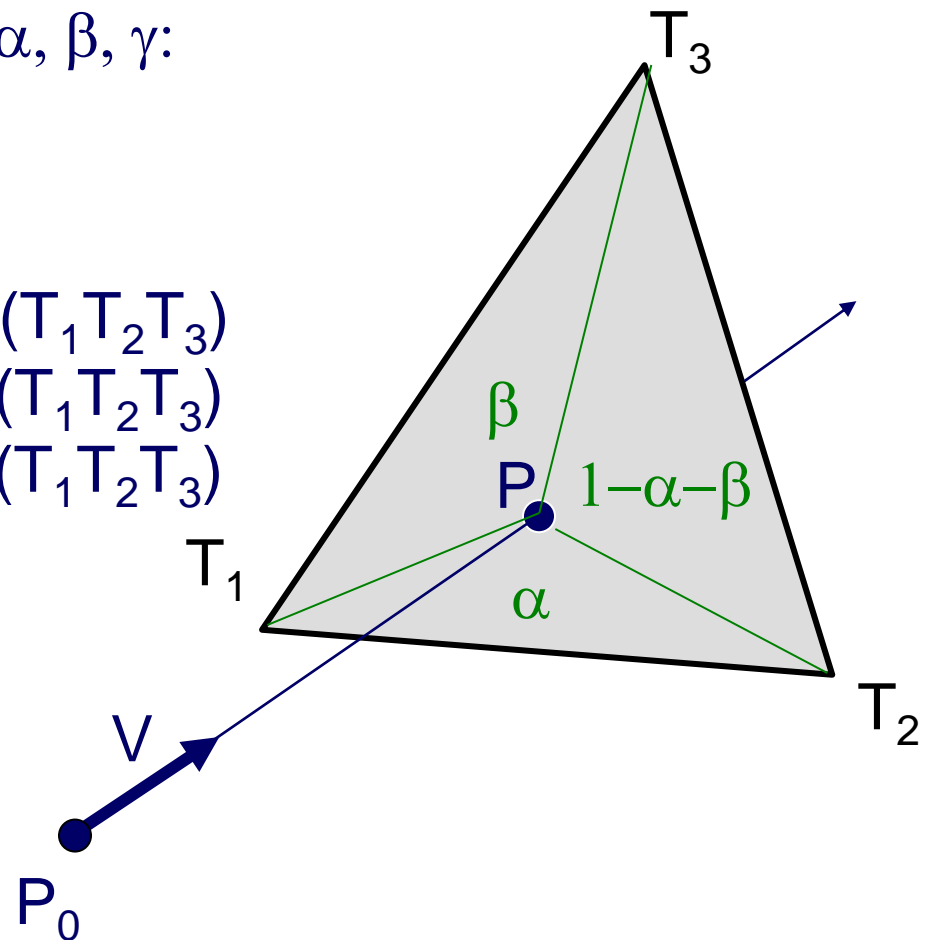
where  $\alpha + \beta + \gamma = 1$

$$\alpha = \text{Area}(T_1 T_2 P) / \text{Area}(T_1 T_2 T_3)$$

$$\beta = \text{Area}(T_1 P T_3) / \text{Area}(T_1 T_2 T_3)$$

$$\gamma = \text{Area}(P T_2 T_3) / \text{Area}(T_1 T_2 T_3)$$

$$= 1 - \alpha - \beta$$





# Ray-Triangle Intersection III

- Check if point is inside triangle parametrically

Compute “barycentric coordinates”  $\alpha$ ,  $\beta$ :

$$\alpha = \text{Area}(T_1T_2P) / \text{Area}(T_1T_2T_3)$$

$$\beta = \text{Area}(T_1PT_3) / \text{Area}(T_1T_2T_3)$$

$$\text{Area}(T_1T_2T_3) = \frac{1}{2} \|\ (T_2 - T_1) \times (T_3 - T_1) \|\$$

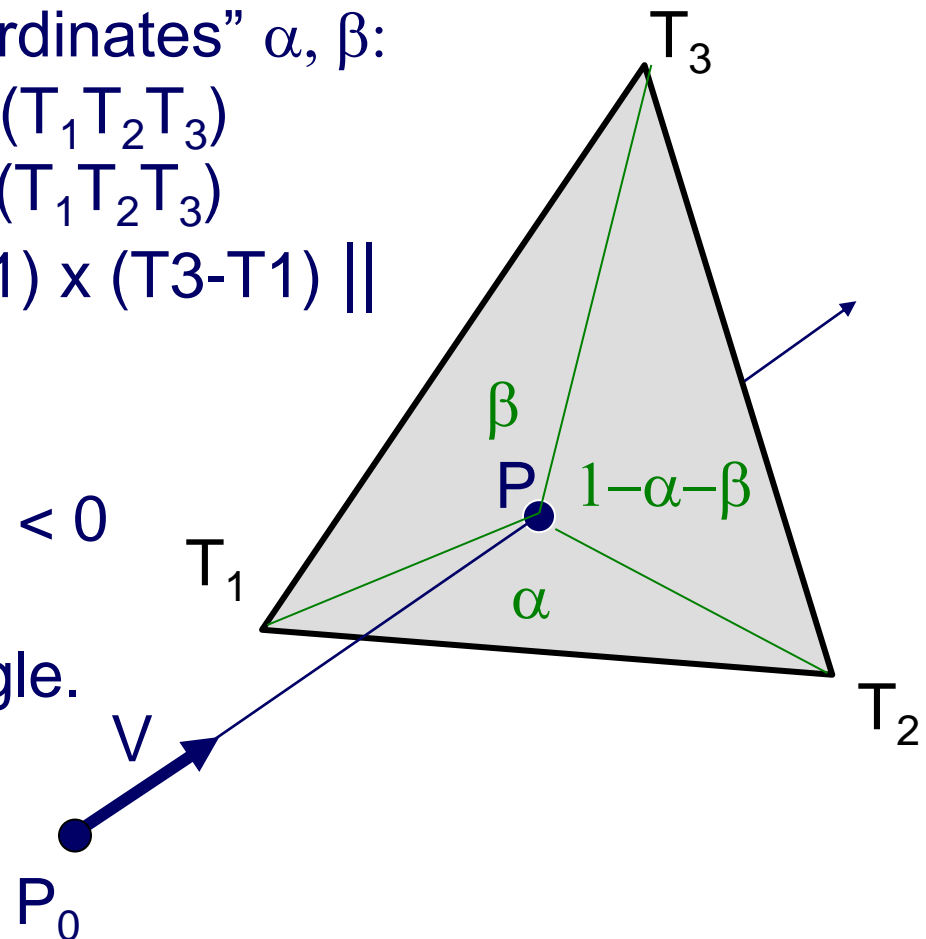
check if backfacing:

$$\left( (T_2 - T_1) \times (T_3 - T_1) \right) \cdot V < 0$$

Check if point inside triangle.

$$0 \leq \alpha \leq 1 \text{ and } 0 \leq \beta \leq 1$$

$$\text{and } \alpha + \beta \leq 1$$





# Ray Intersection

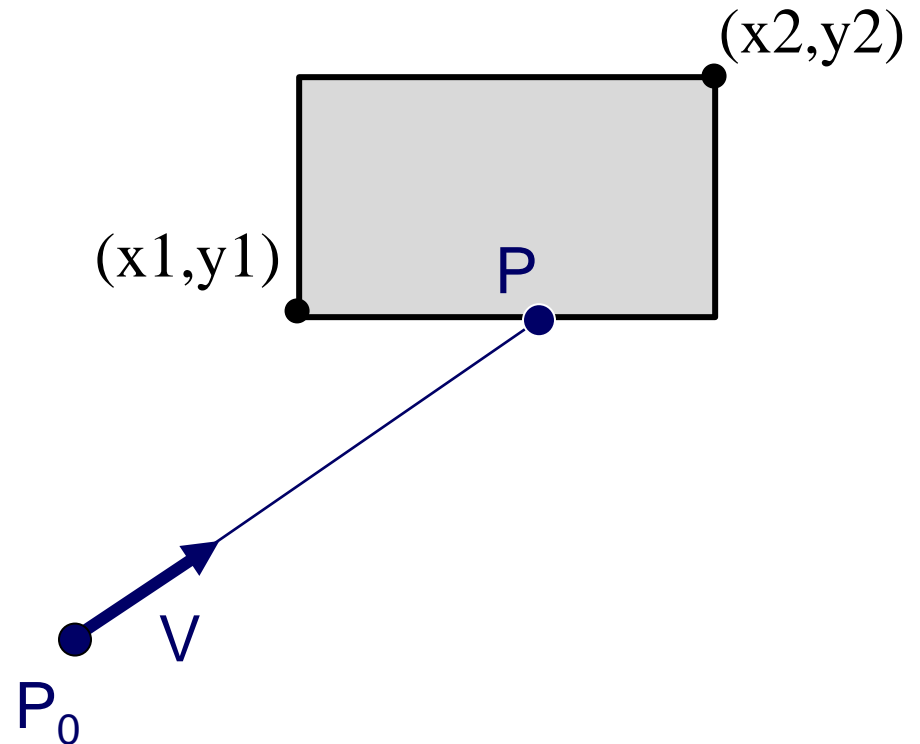
- Ray Intersection
  - Sphere
  - Triangle
  - **Box**
  - Scene
  
- Ray Intersection Acceleration
  - Bounding volumes
  - Uniform grids
  - Octrees
  - BSP trees





# Ray-Box Intersection

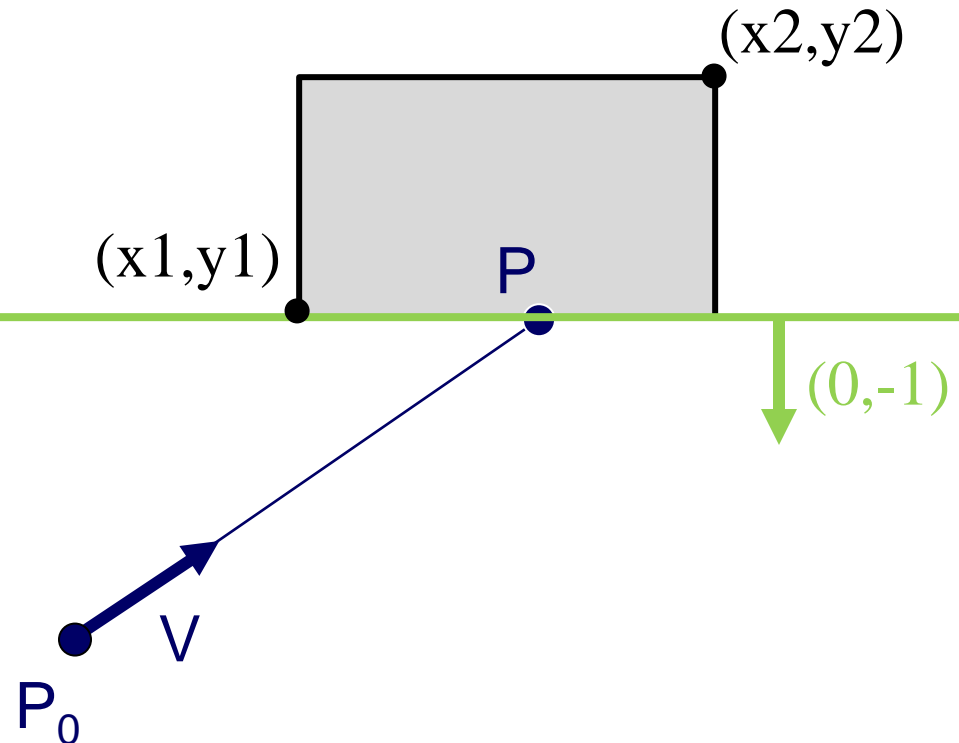
- Check front-facing sides for intersection with ray and return closest intersection (least  $t$ )





# Ray-Box Intersection

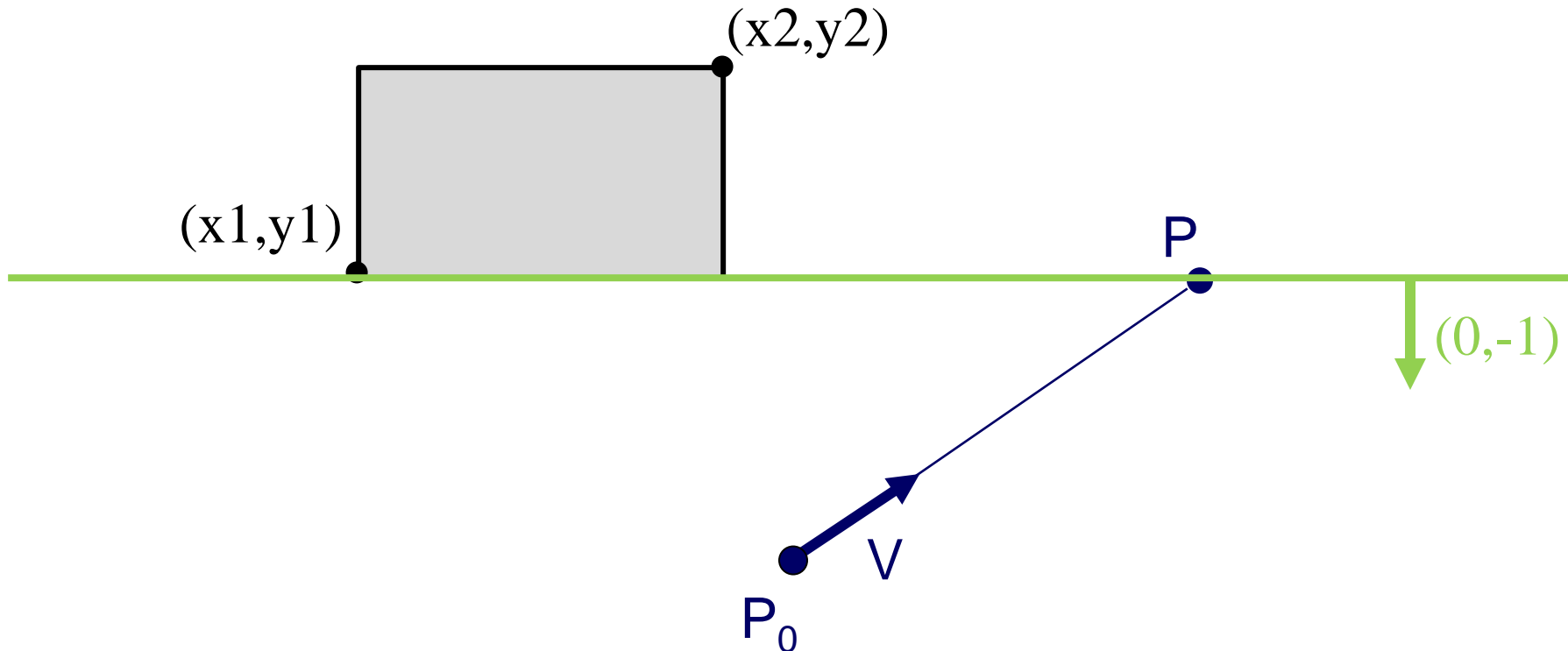
- Check front-facing sides for intersection with ray and return closest intersection (least  $t$ )
  - Find intersection with plane
  - Check if point is inside rectangle





# Ray-Box Intersection

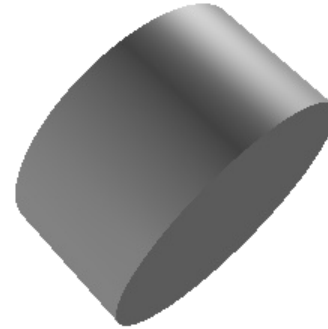
- **Check** front-facing sides for intersection with ray and return closest intersection (least  $t$ )
  - Find intersection with plane
  - Check if point is inside rectangle



# Other Ray-Primitive Intersections



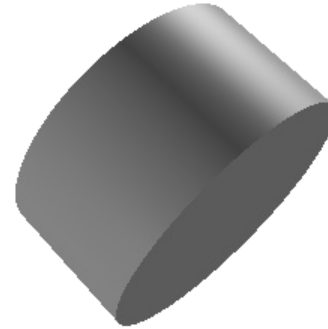
- Cone, cylinder:
  - Similar to sphere
  - Must also check end caps



# Other Ray-Primitive Intersections



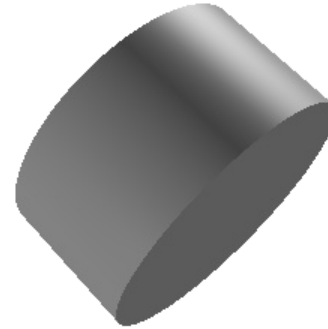
- Cone, cylinder:
  - Similar to sphere
  - Must also check end caps
- Convex polygon
  - Same as triangle (check point-in-polygon algebraically)
  - Or, decompose into triangles, and check all of them



# Other Ray-Primitive Intersections



- Cone, cylinder:
  - Similar to sphere
  - Must also check end caps
- Convex polygon
  - Same as triangle (check point-in-polygon algebraically)
  - Or, decompose into triangles, and check all of them
- Mesh
  - Compute intersection for all polygons
  - Return closest intersection (least  $t$ )



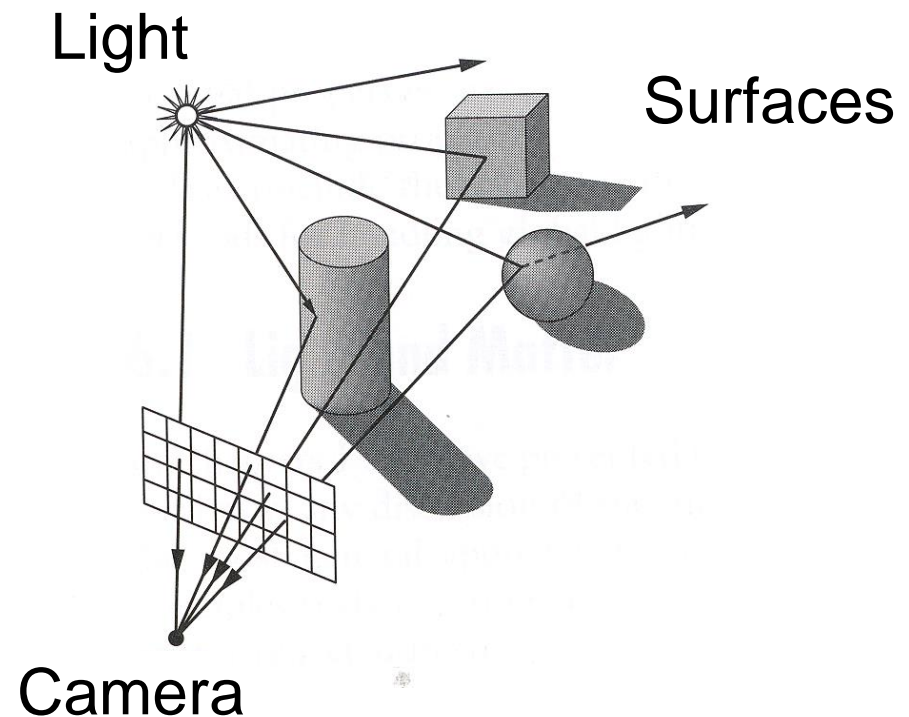


# Ray Intersection

- Ray Intersection
  - Sphere
  - Triangle
  - Box
  - Scene
  
- Ray Intersection Acceleration
  - Bounding volumes
  - Uniform grids
  - Octrees
  - BSP trees

# Ray-Scene Intersection

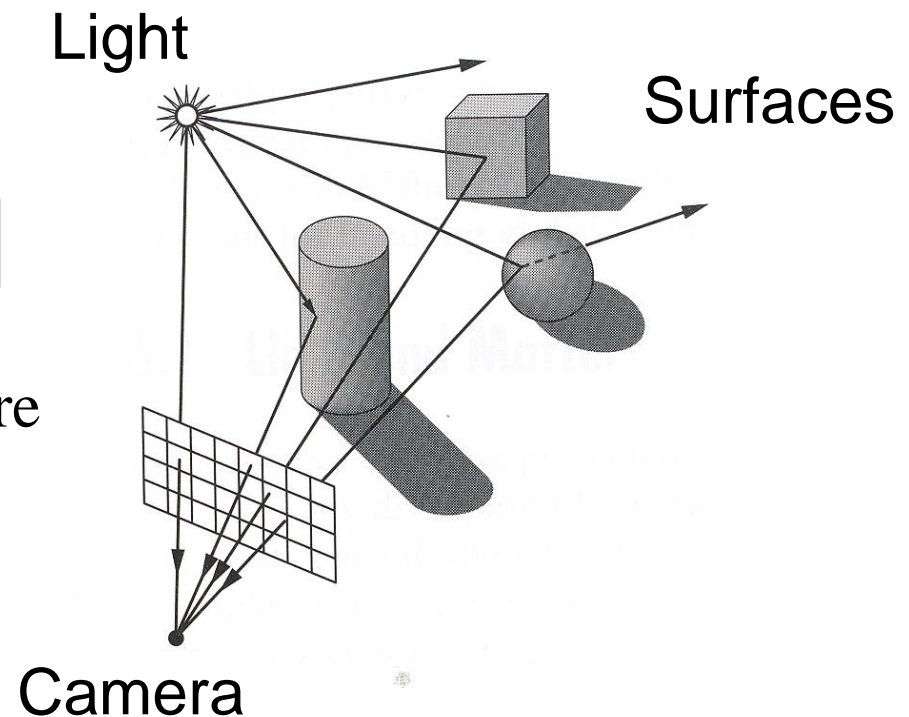
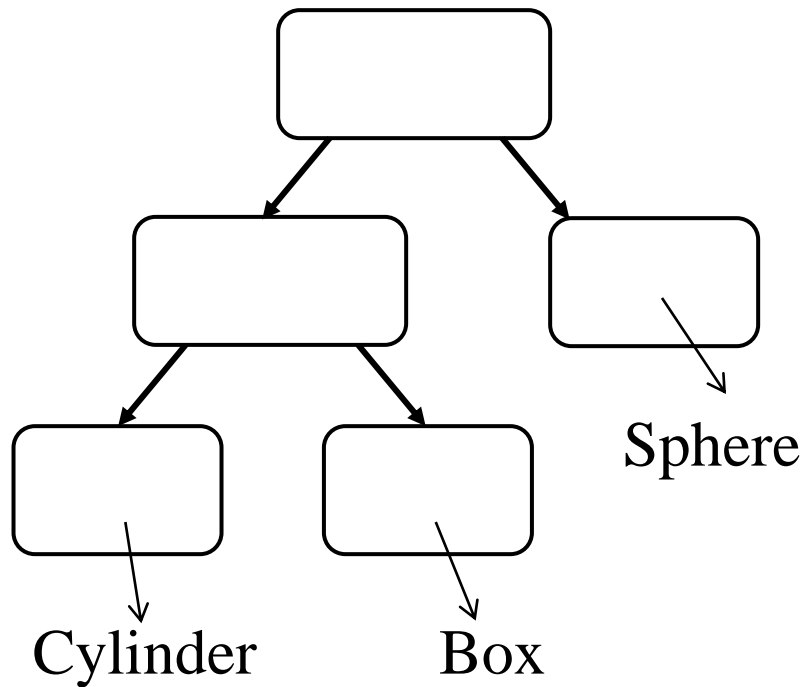
- Intuitive method
  - Compute intersection for **all** nodes of scene graph
  - Return closest intersection (least  $t$ )





# Ray-Scene Intersection

- Scene graph is a DAG
  - Traverse with recursion



# Ray-Scene Intersection I



```
R3Intersection ComputeIntersection(R3Scene *scene, R3Node *node, R3Ray *ray)
{
 // Check for intersection with shape
 shape_intersection = Intersect node's shape with ray
 if (shape_intersection is a hit) closest_intersection = shape_intersection
 else closest_intersection = infinitely far miss
}
```

# Ray-Scene Intersection I



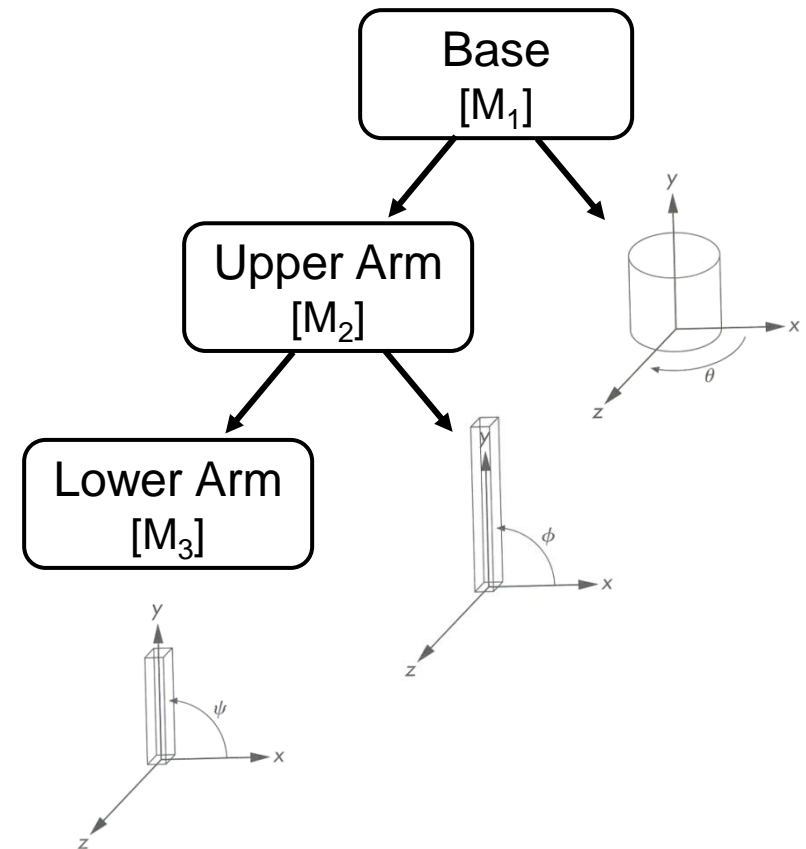
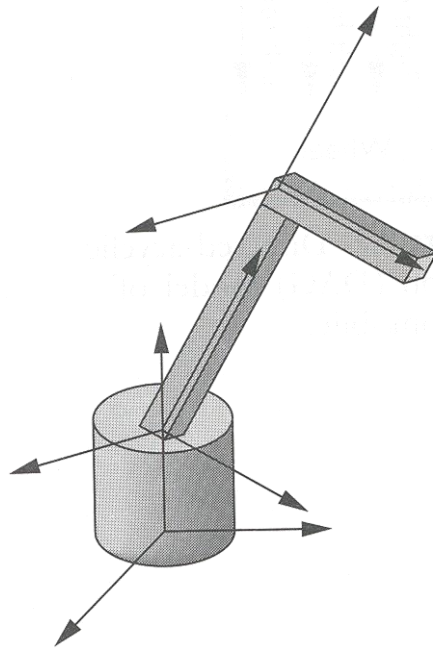
```
R3Intersection ComputeIntersection(R3Scene *scene, R3Node *node, R3Ray *ray)
{
 // Check for intersection with shape
 shape_intersection = Intersect node's shape with ray
 if (shape_intersection is a hit) closest_intersection = shape_intersection
 else closest_intersection = infinitely far miss

 // Check for intersection with children nodes
 for each child node
 // Check for intersection with child contents
 child_intersection = ComputeIntersection(scene, child, ray);
 if (child_intersection is a hit and is closer than closest_intersection)
 closest_intersection = child_intersection;

 // Return closest intersection in tree rooted at this node
 return closest_intersection
}
```

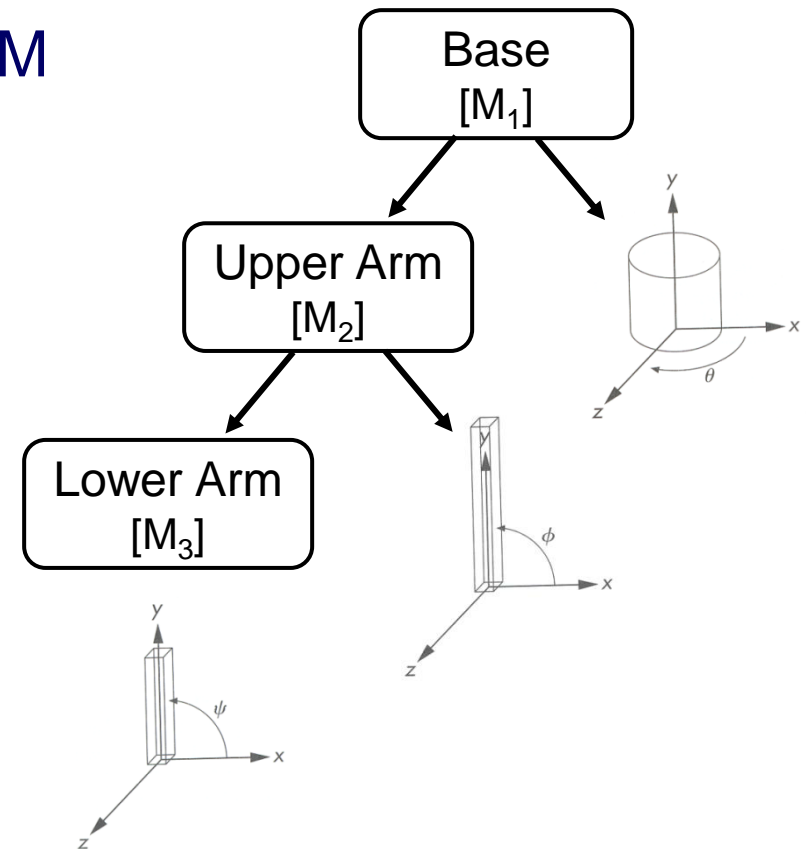
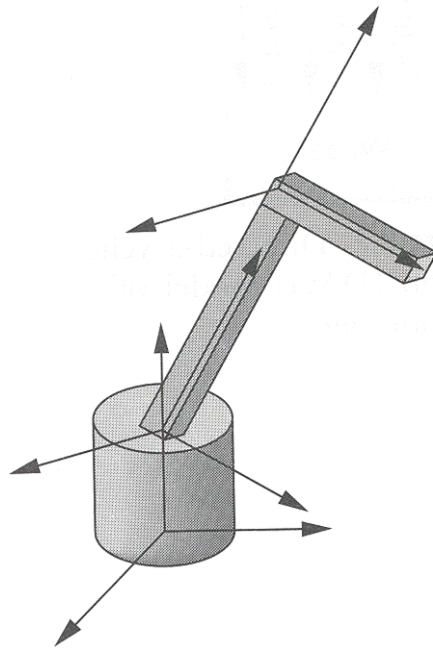
# Ray-Scene Intersection

- Scene graph can have transformations



# Ray-Scene Intersection

- Scene graph node can have transformations
  - Transform **ray** (not primitives) by **inverse** of  $M$
  - Intersect in coordinate system of node
  - Transform intersection by  $M$



# Ray-Scene Intersection II



```
R3Intersection ComputeIntersection(R3Scene *scene, R3Node *node, R3Ray *ray)
{
 // Transform ray by inverse of node's transformation

 // Check for intersection with shape

 // Check for intersection with children nodes

 // Transform intersection by node's transformation

 // Return closest intersection in tree rooted at this node
}
```

# Ray-Scene Intersection II



```
R3Intersection ComputeIntersection(R3Scene *scene, R3Node *node, R3Ray *ray)
{
 // Transform ray by inverse of node's transformation

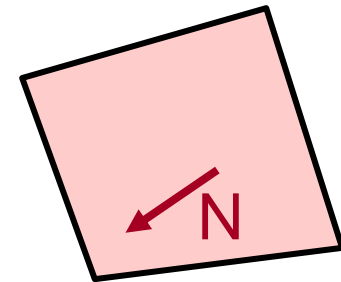
 // Check for intersection with shape

 // Check for intersection with children nodes

 // Transform intersection by node's transformation

 // Return closest intersection in tree rooted at this node
}
```

**Note:** directions (including ray direction and surface normal  $N$ ) must be transformed by inverse transpose of  $M$





# Ray Intersection

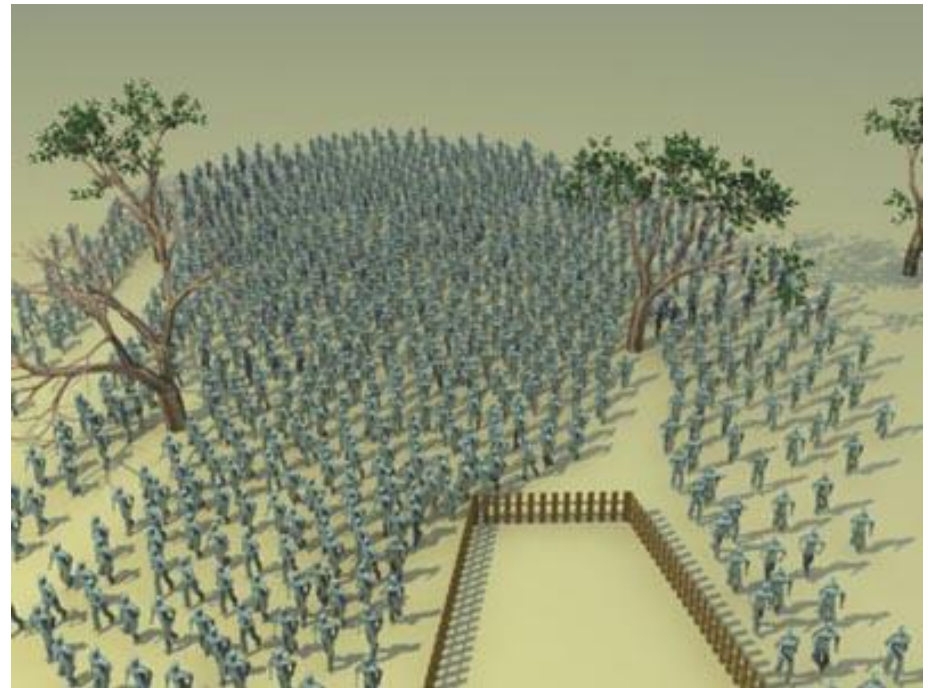
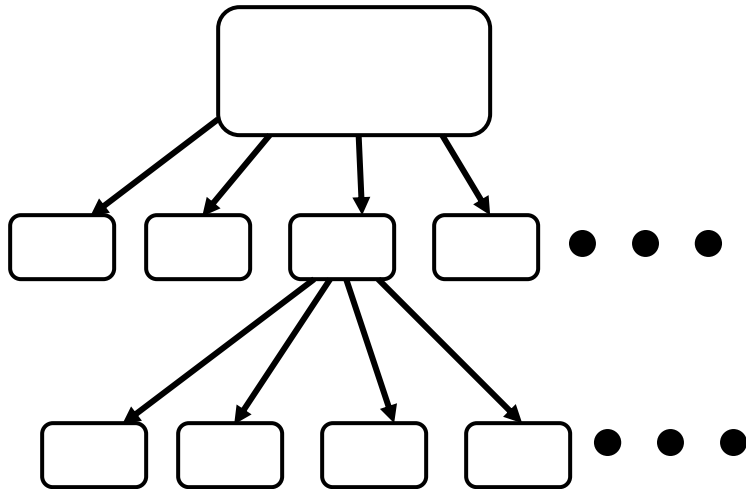
- Ray Intersection
  - Sphere
  - Triangle
  - Box
  - Scene
  
- Ray Intersection Acceleration
  - Bounding volumes
  - Uniform grids
  - Octrees
  - BSP trees



# Ray Intersection Acceleration



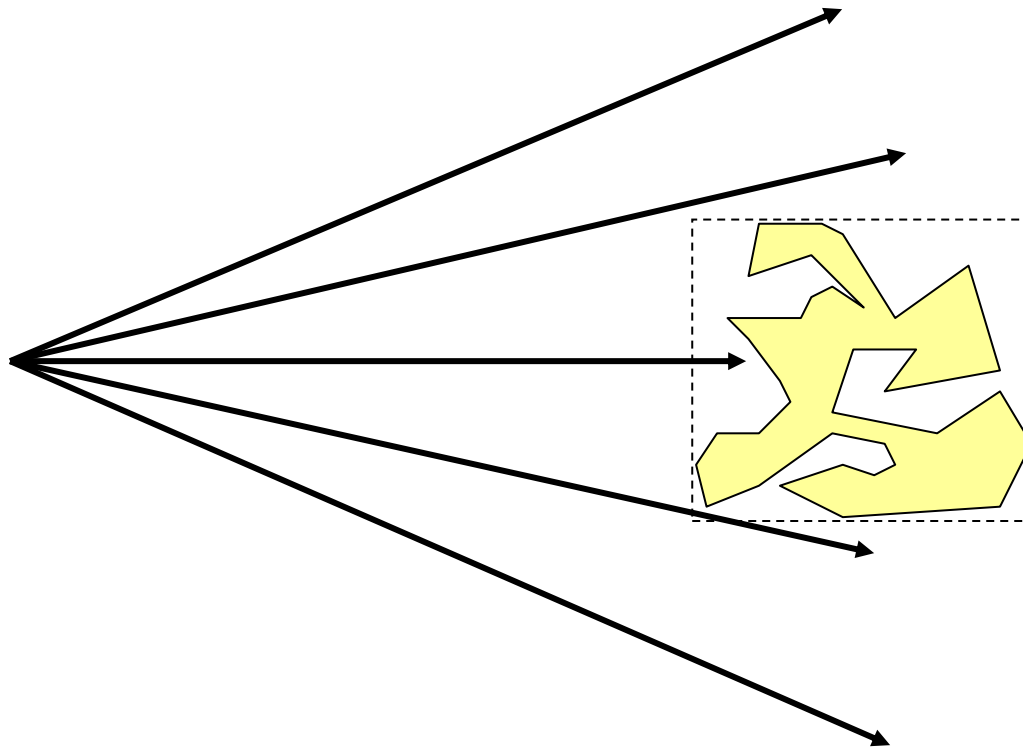
- What if there are a lot of nodes?



# Bounding Volumes



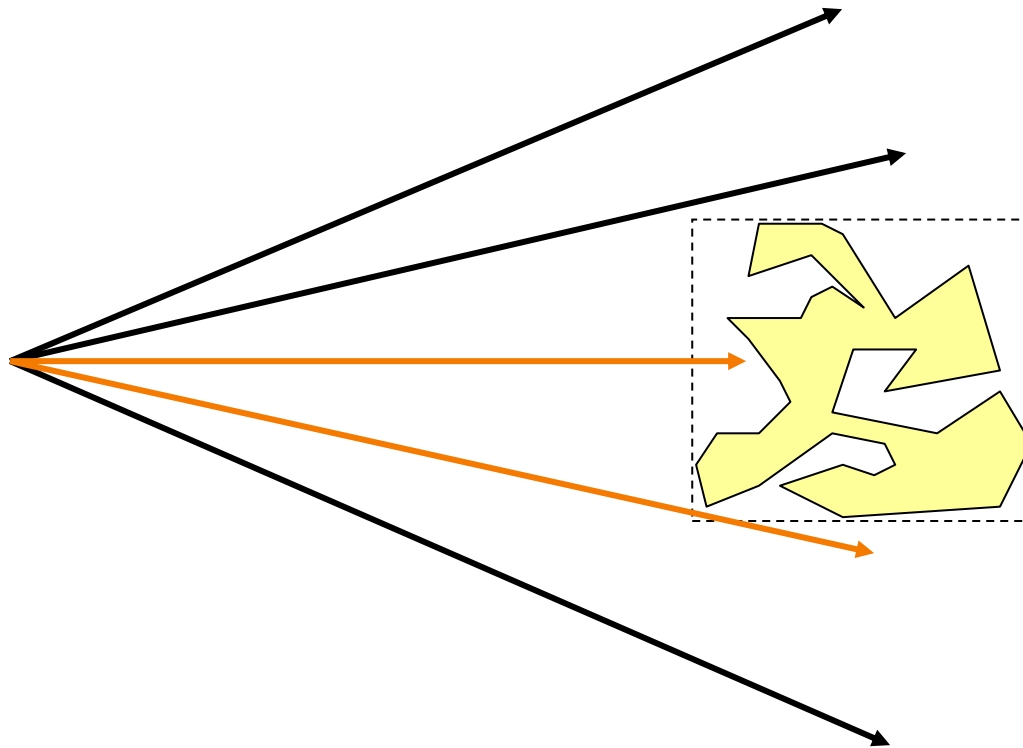
- Check for intersection with simple bounding volume first



# Bounding Volumes

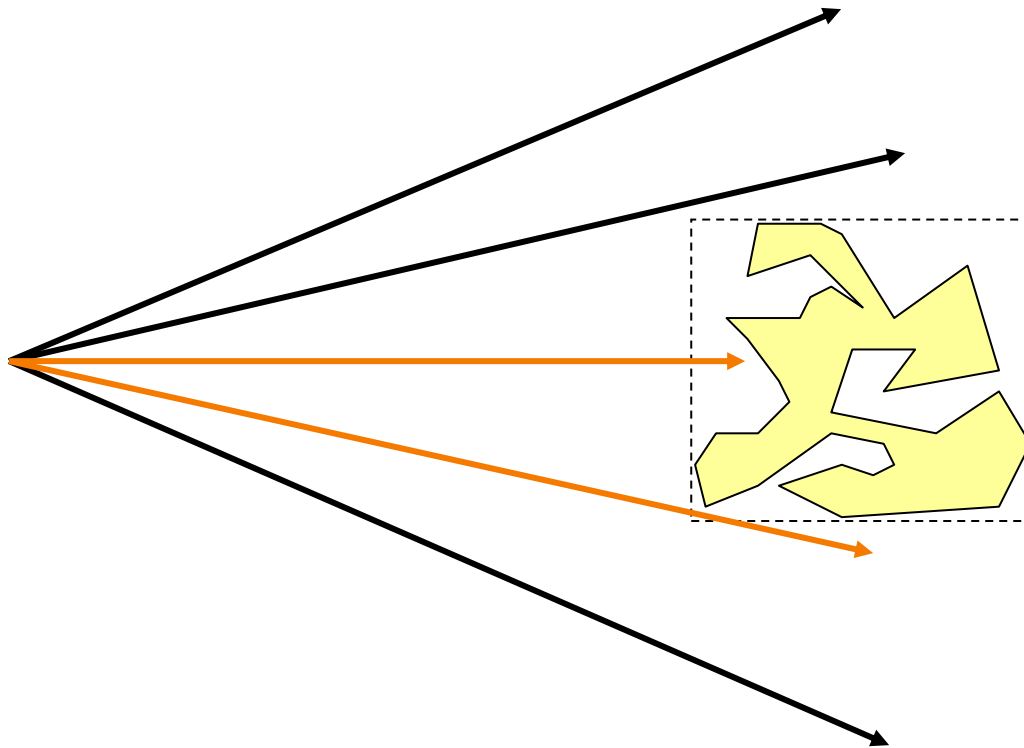


- Check for intersection with bounding volume first



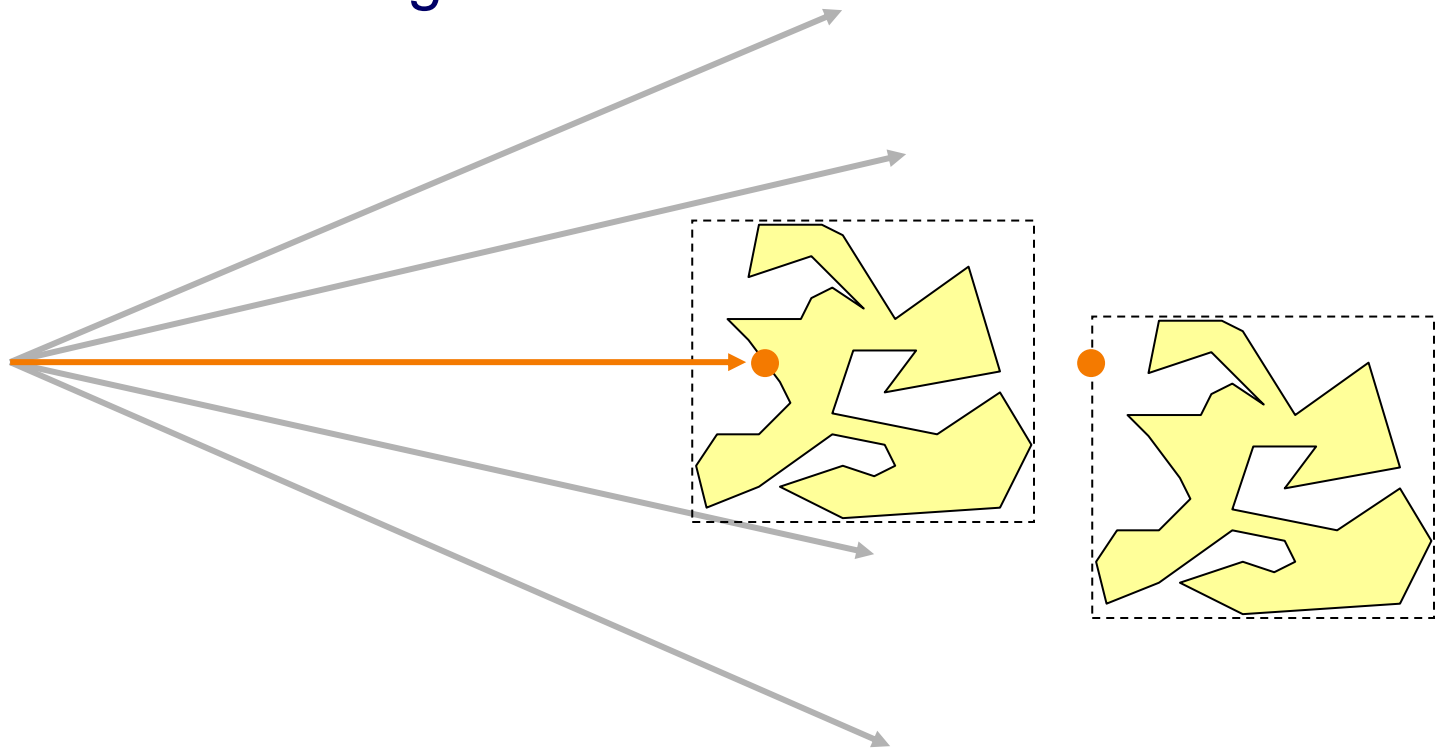
# Bounding Volumes

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



# Bounding Volumes

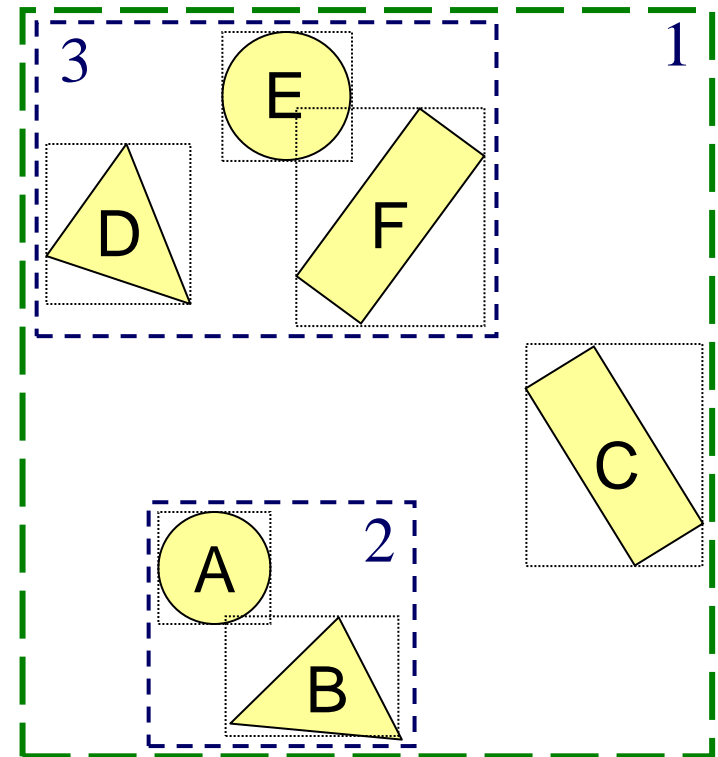
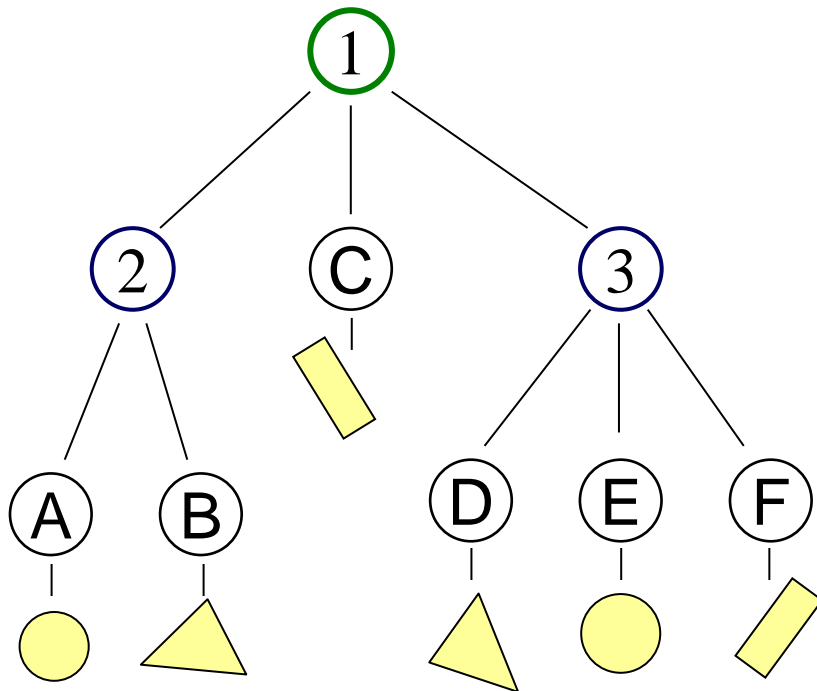
- Check for intersection with bounding volume first
  - If already found a primitive intersection closer than intersection with bounding box, then skip checking contents of bounding box





# Bounding Volume Hierarchies

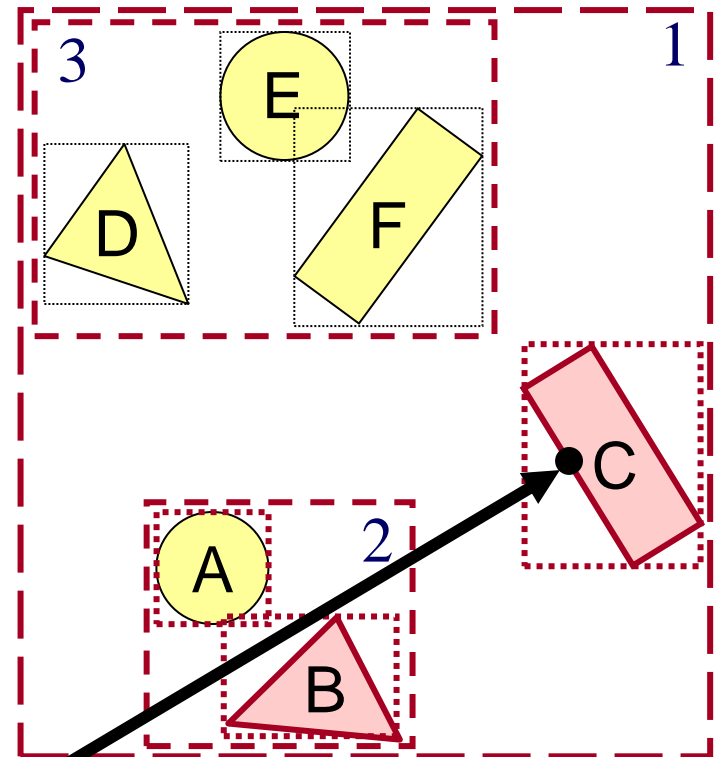
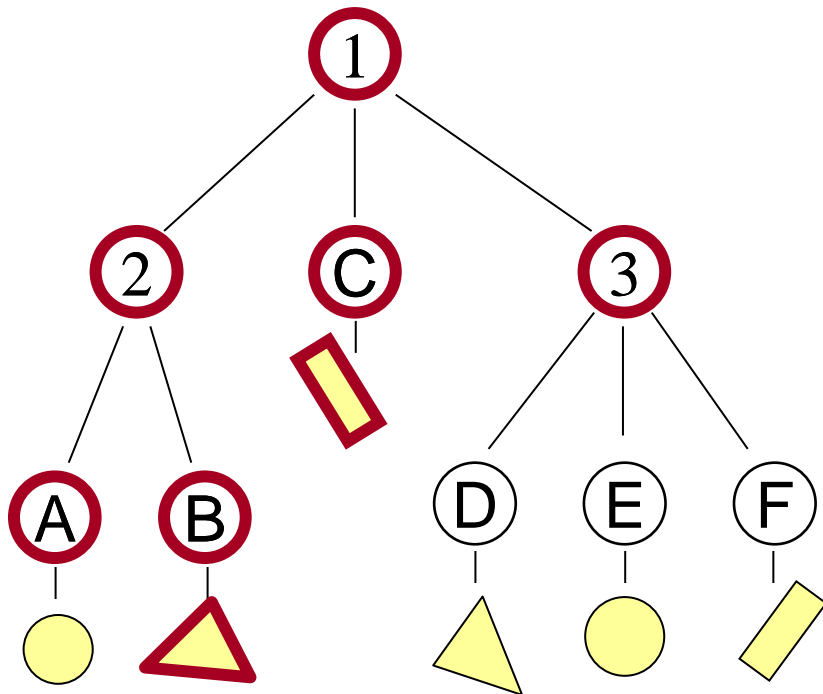
- Scene graph has hierarchy of bounding volumes
  - Bounding volume of interior node contains all children





# Bounding Volume Hierarchies

- Checking bounding volumes hierarchically (within each node) can greatly accelerate ray intersection



# Bounding Volume Hierarchies



```
R3Intersection ComputeIntersection(R3Scene *scene, R3Node *node, R3Ray *ray)
{
 // Transform ray by inverse of node's transformation
 // Check for intersection with shape

 // Check for intersection with children nodes
 for each child node
 // Check for intersection with child bounding box first
 bbox_intersection = Intersect child's bounding box with ray
 if (bbox_intersection is a miss or further than closest_intersection) continue

 // Check for intersection with child contents
 child_intersection = ComputeIntersection(scene, child, ray);
 if (child_intersection is a hit and is closer than closest_intersection)
 closest_intersection = child_intersection;

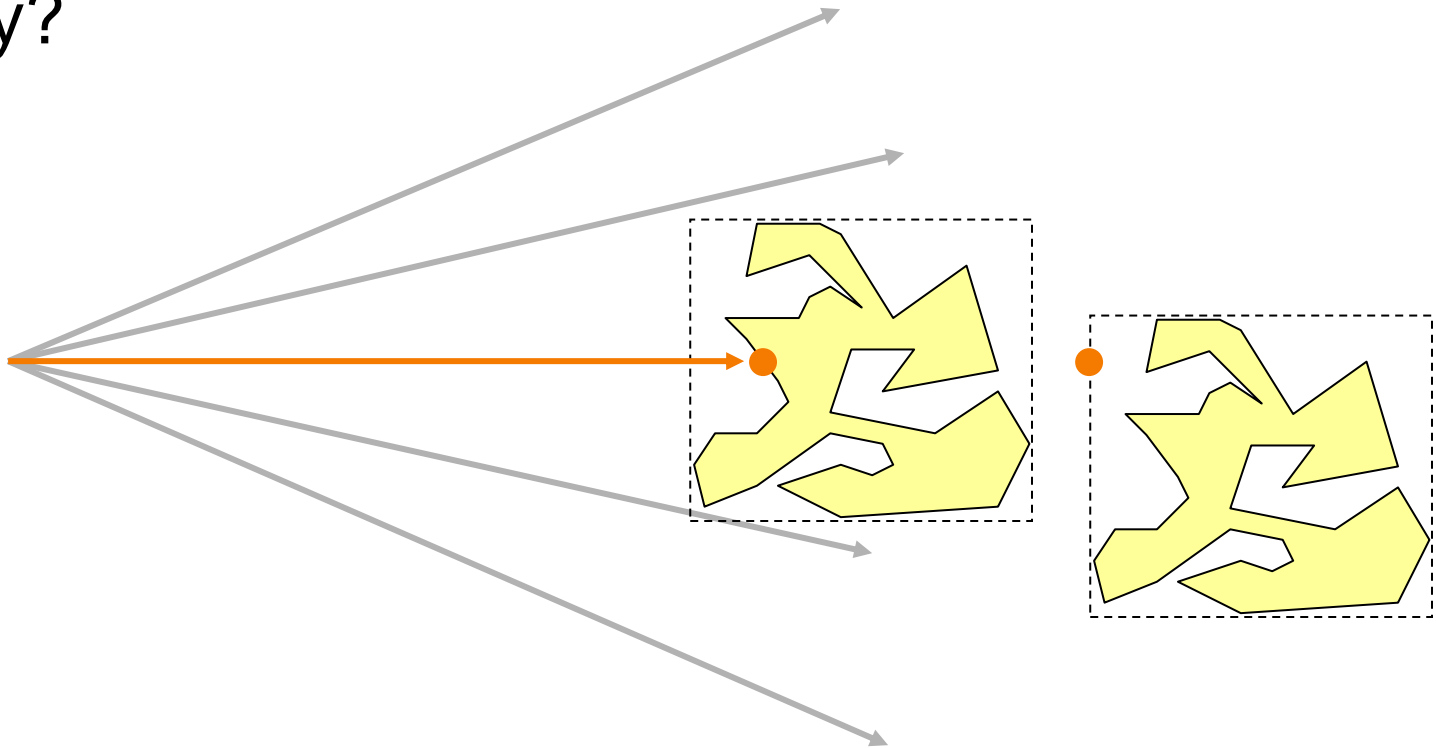
 // Transform intersection by node's transformation
 // Return closest intersection in tree rooted at this node
}
```



# Sort Bounding Volume Intersections

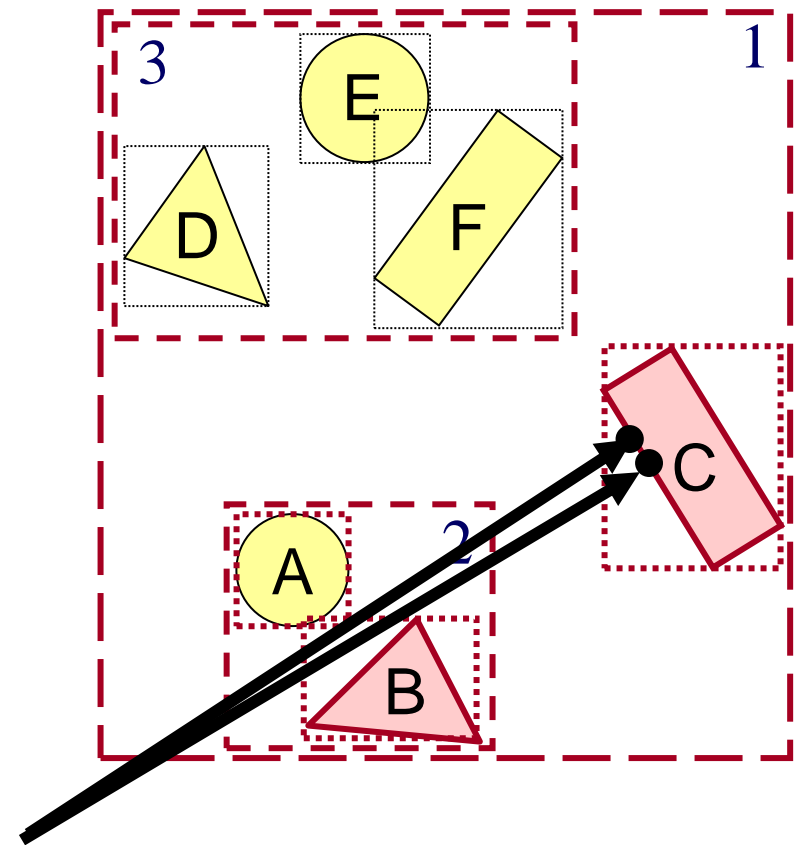
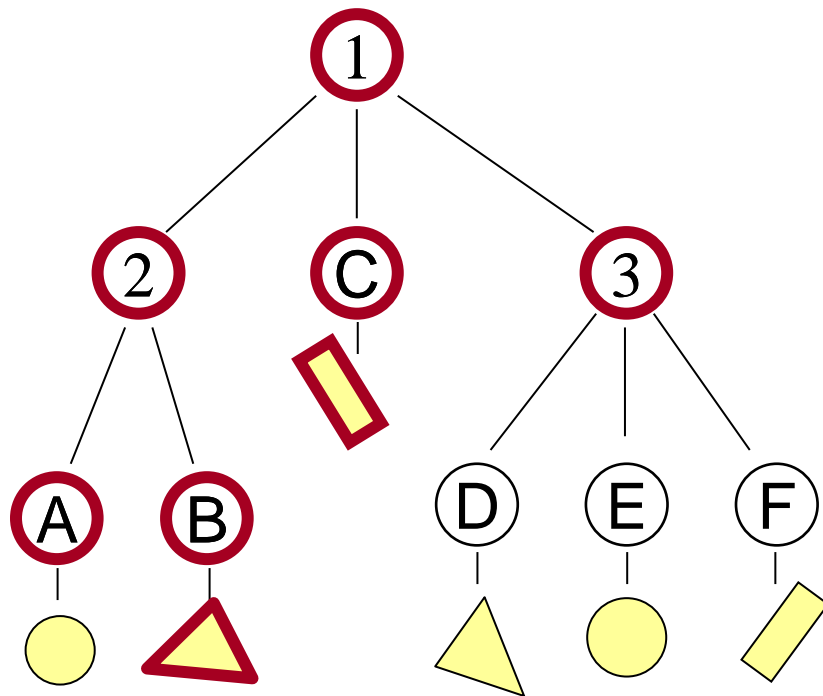


- Sort child bounding volume intersections and then visit child nodes in front-to-back order
- Why?



# Cache Node Intersections

- For each node, store closest child intersection from previous ray and check that node first





# Bounding Volumes

- Common primitives are:
  - Axis-aligned bounding box
  - Sphere
- What are the tradeoffs?
  - Sphere has simple/efficient intersection code
  - Bounding box is generally “tighter”



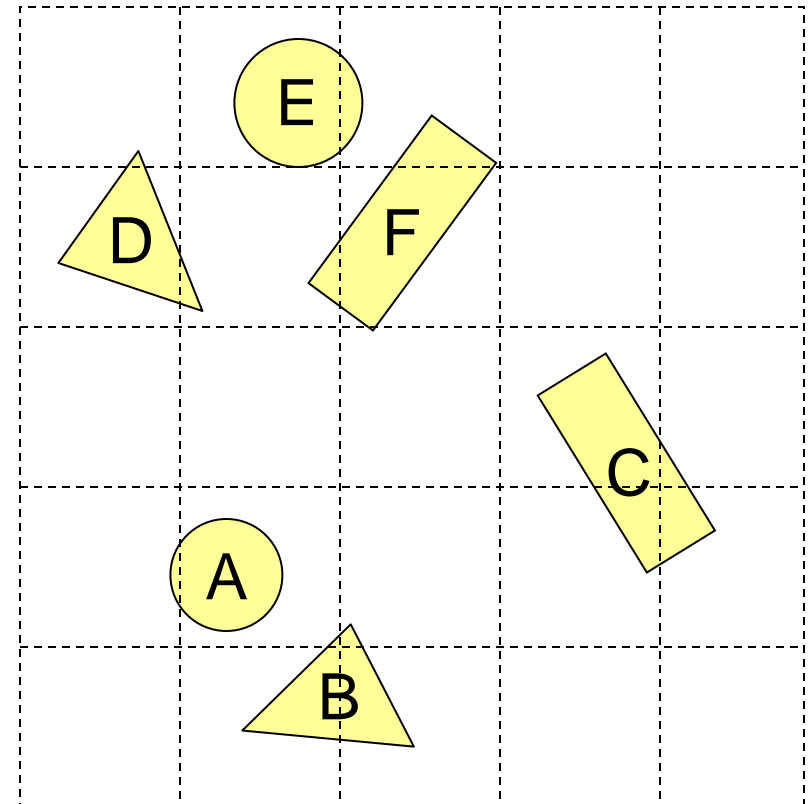
# Ray Intersection

- Ray Intersection
  - Sphere
  - Triangle
  - Box
  - Scene
  
- Ray Intersection Acceleration
  - Bounding volumes
  - **Uniform grids**
  - Octrees
  - BSP trees

# Uniform Grid



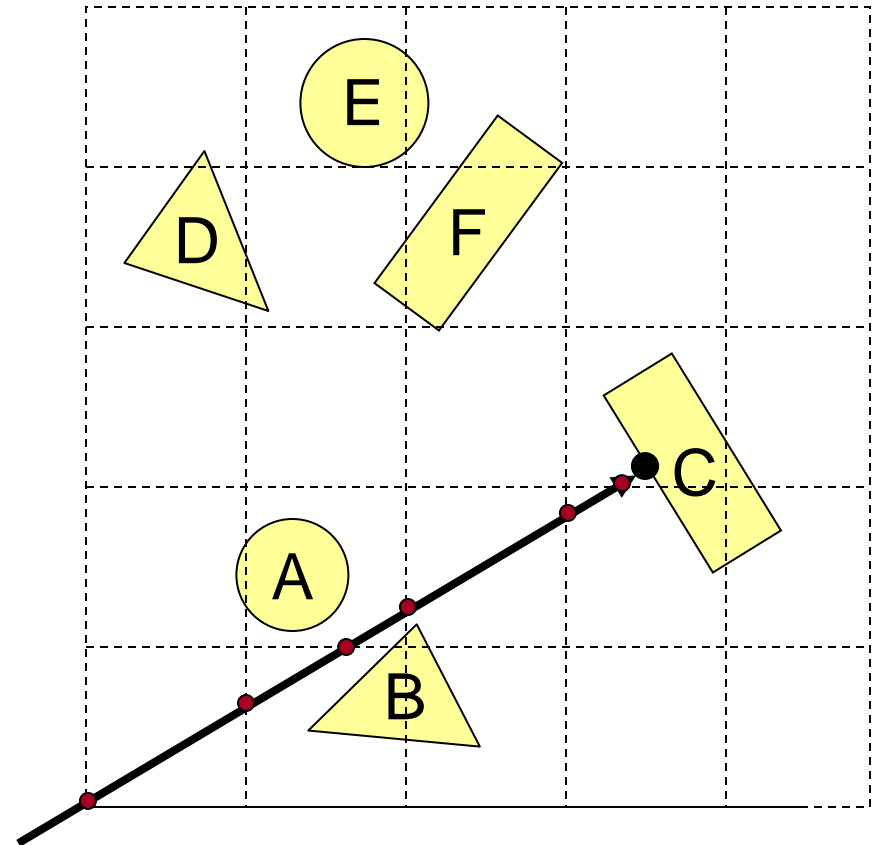
- Construct uniform grid over scene
  - Index primitives according to overlaps with grid cells



# Uniform Grid

- Trace rays through grid cells
  - Fast
  - Incremental

Only check primitives in intersected grid cells

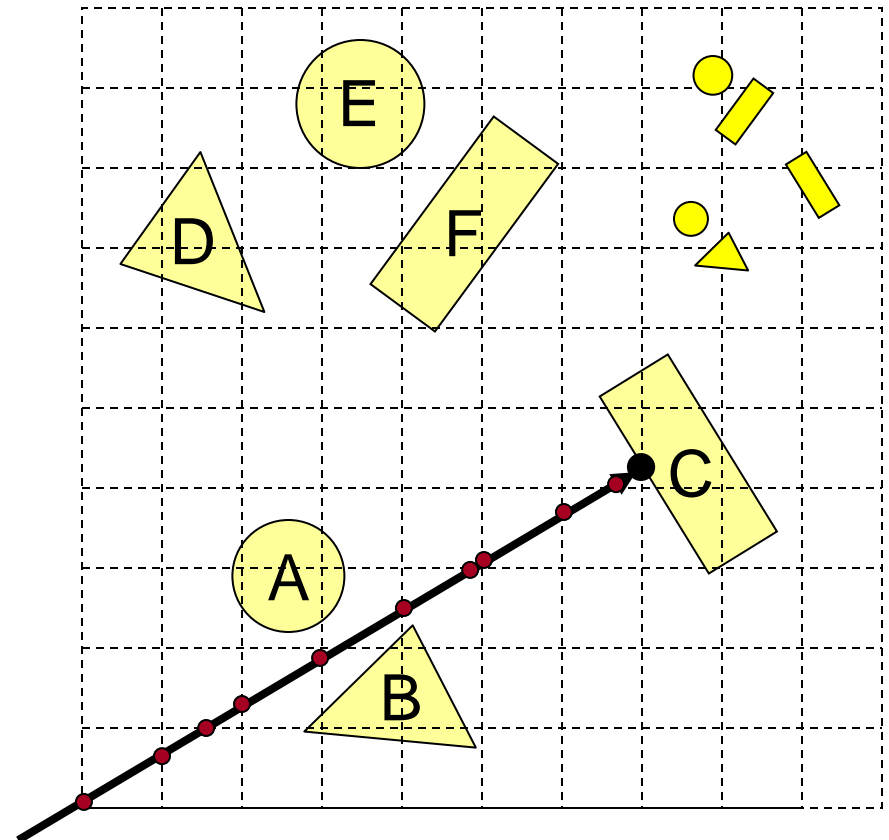


# Uniform Grid

- Potential problem:
  - How choose suitable grid resolution?

Too little benefit  
if grid is too coarse

Too much cost  
if grid is too fine





# Ray Intersection

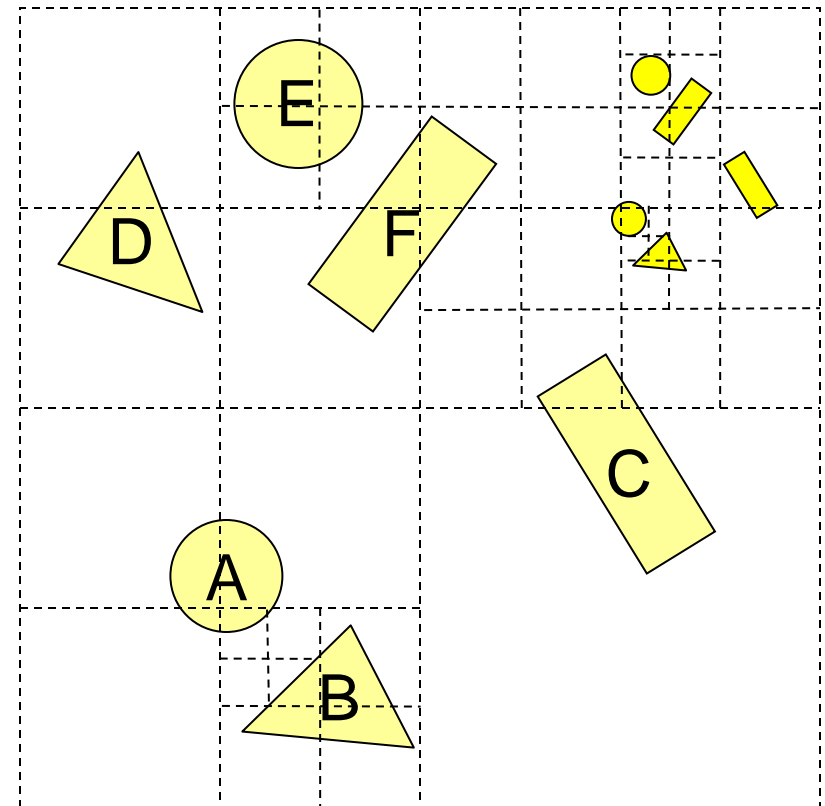
- Ray Intersection
  - Sphere
  - Triangle
  - Box
  - Scene
  
- Ray Intersection Acceleration
  - Bounding volumes
  - Uniform grids
  - **Octrees**
  - BSP trees



# Octree

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

Generally fewer cells

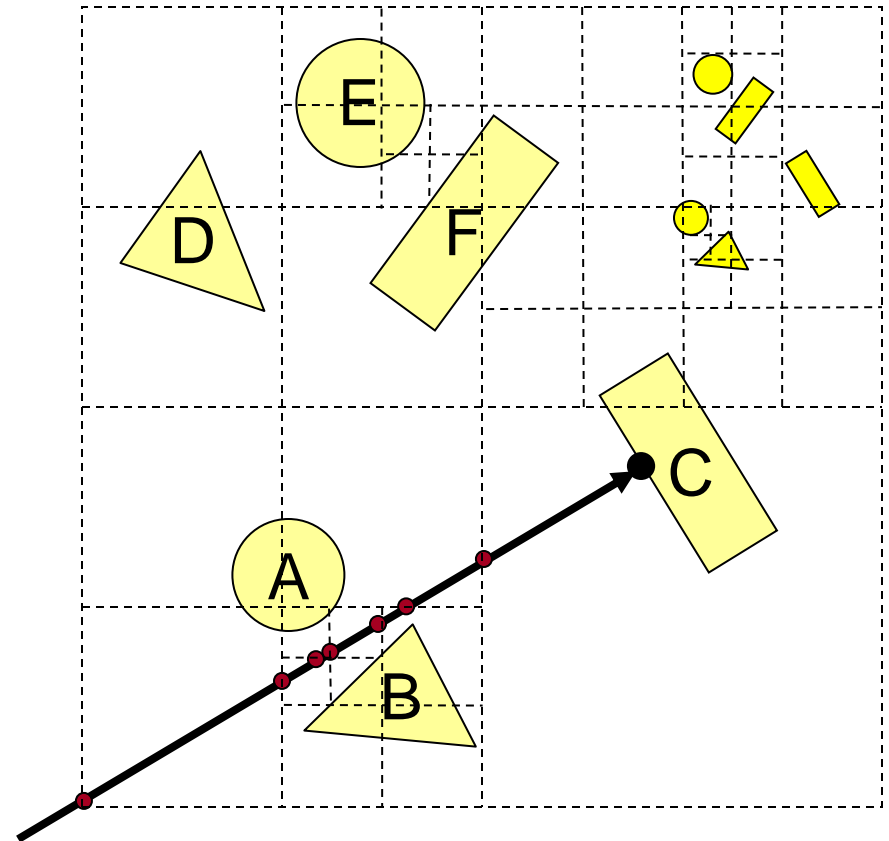




# Octree

- Trace rays through neighbor cells
  - Fewer cells

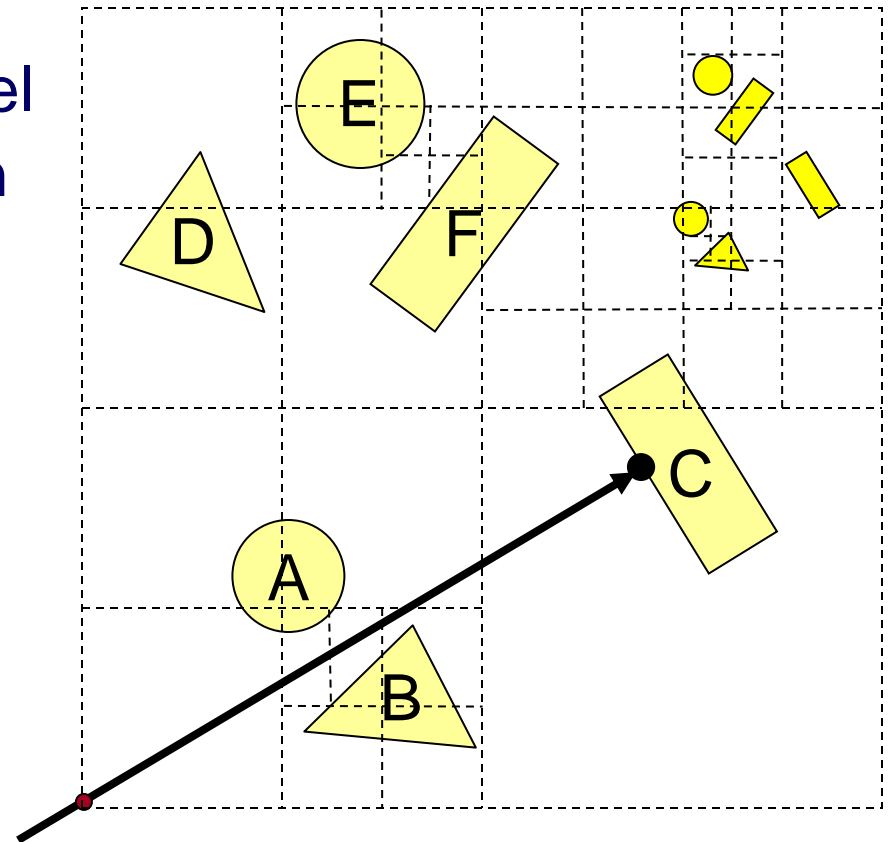
Trade-off fewer cells for more expensive traversal



# Octree



- Or, check rays versus octree boxes hierarchically
  - Computing octree boxes while descending tree
  - Sort eight boxes front-to-back at each level
  - Check primitives/children inside box





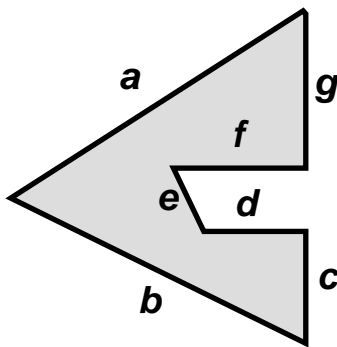
# Ray Intersection

- Ray Intersection
  - Sphere
  - Triangle
  - Box
  - Scene
  
- Ray Intersection Acceleration
  - Bounding volumes
  - Uniform grids
  - Octrees
  - **BSP trees**

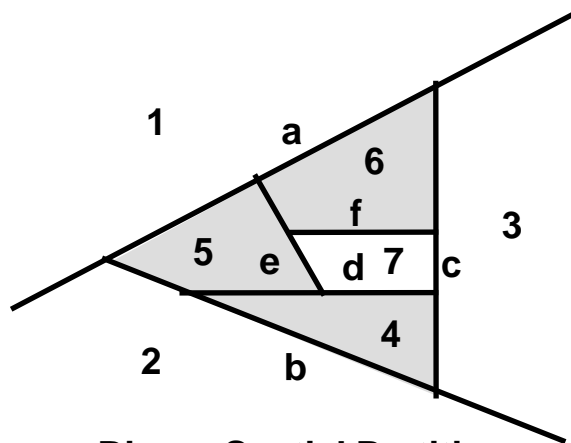


# Binary Space Partition (BSP) Tree

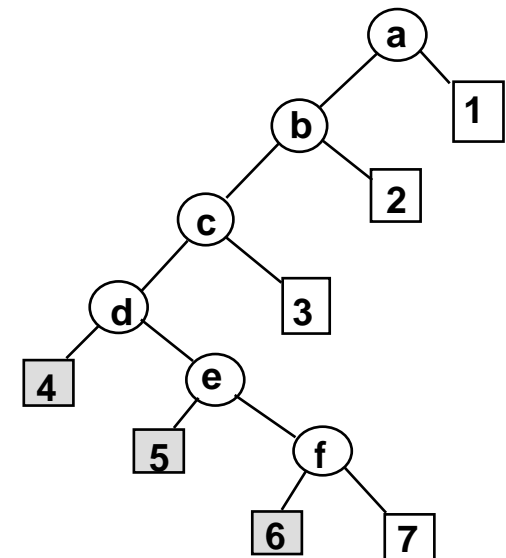
- Recursively partition space by planes
  - BSP tree nodes store partition plane and set of polygons lying on that partition plane
  - Every part of every polygon lies on a partition plane



Object



Binary Spatial Partition

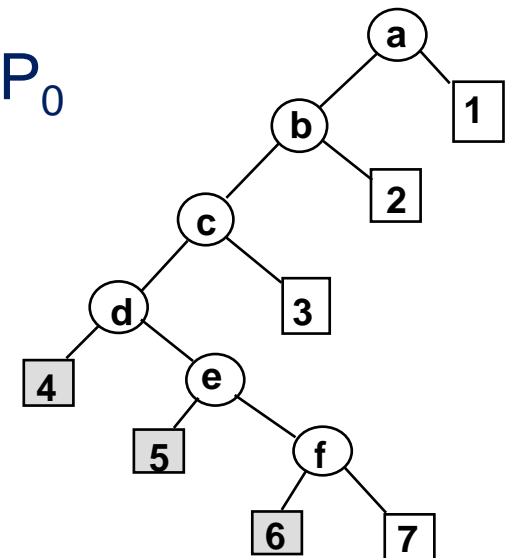
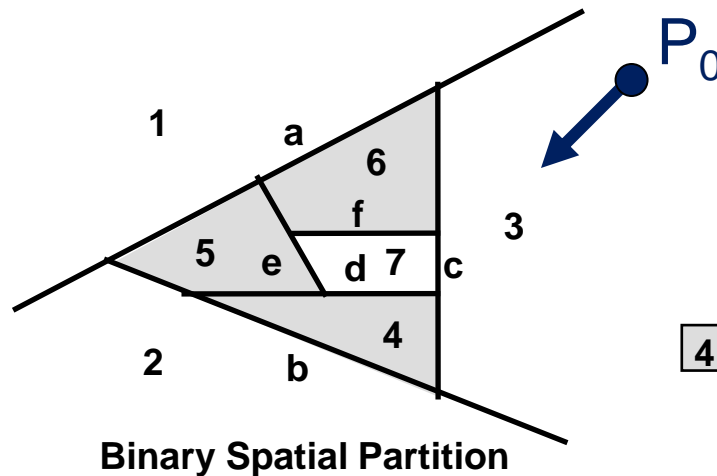
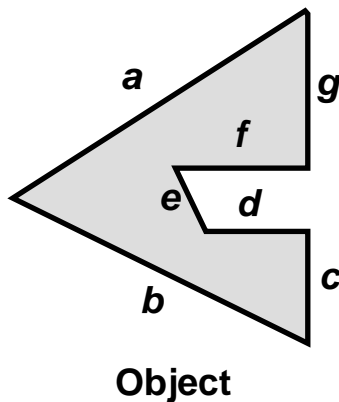


Binary Tree



# Binary Space Partition (BSP) Tree

- Traverse nodes of BSP tree front-to-back
  - Visit halfspace (child node) containing  $P_0$
  - Intersect polygons lying on partition plane
  - Visit halfspace (other child node) not containing  $P_0$



Binary Tree

# Binary Space Partition (BSP) Tree



R3Intersection

```
ComputeBSPIntersection(R3Ray *ray, BspNode *node, double min_t, double max_t)
{
 // Compute parametric value of ray-plane intersection
 t = ray parameter for intersection with split plane of node
 if (t < min_t) || (t < max_t) return no_intersection;

 // Compute side of partition plane that contains ray start point
 int side = (SignedDistance(node->plane, ray.Start()) < 0) ? 0 : 1;
 intersection1 = ComputeBSPIntersection(ray, node->child[side], min_t, t);
}
```

# Binary Space Partition (BSP) Tree



R3Intersection

```
ComputeBSPIntersection(R3Ray *ray, BspNode *node, double min_t, double max_t)
{
 // Compute parametric value of ray-plane intersection
 t = ray parameter for intersection with split plane of node
 if (t < min_t) || (t < max_t) return no_intersection;

 // Compute side of partition plane that contains ray start point
 int side = (SignedDistance(node->plane, ray.Start()) < 0) ? 0 : 1;
 intersection1 = ComputeBSPIntersection(ray, node->child[side], min_t, t);
 if (intersection1 is a hit) return intersection1;
 intersection2 = ComputePolygonsIntersection(ray, node->polygons);
 if (intersection2 is a hit) return intersection2;
}
```



# Binary Space Partition (BSP) Tree



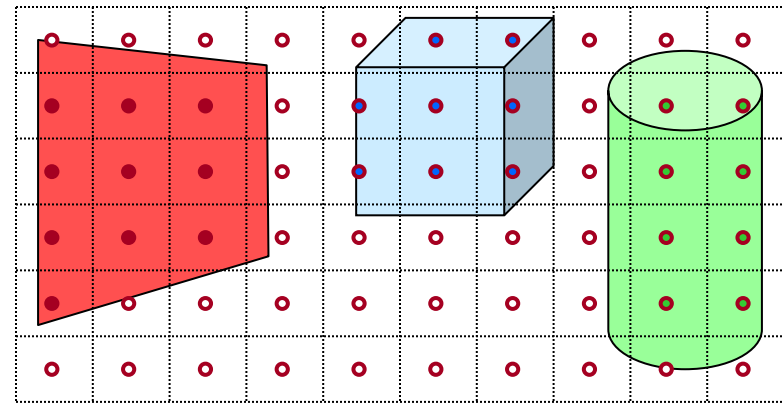
R3Intersection

```
ComputeBSPIntersection(R3Ray *ray, BspNode *node, double min_t, double max_t)
{
 // Compute parametric value of ray-plane intersection
 t = ray parameter for intersection with split plane of node
 if (t < min_t) || (t < max_t) return no_intersection;

 // Compute side of partition plane that contains ray start point
 int side = (SignedDistance(node->plane, ray.Start()) < 0) ? 0 : 1;
 intersection1 = ComputeBSPIntersection(ray, node->child[side], min_t, t);
 if (intersection1 is a hit) return intersection1;
 intersection2 = ComputePolygonsIntersection(ray, node->polygons);
 if (intersection2 is a hit) return intersection2;
 intersection3 = ComputeBSPIntersection(ray, node->child[1-side], t, max_t);
 return intersection 3;
}
```

# Other Accelerations

- Screen space coherence – check  $> 1$  ray at once
  - Beam tracing
  - Pencil tracing
  - Cone tracing
- Memory coherence
  - Large scenes
- Parallelism
  - Ray casting is “embarrassingly parallelizable”
  - Assignment 3 (raytracer) runs program per-pixel
- etc.





# Acceleration

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

Expected time is sub-linear in number of primitives

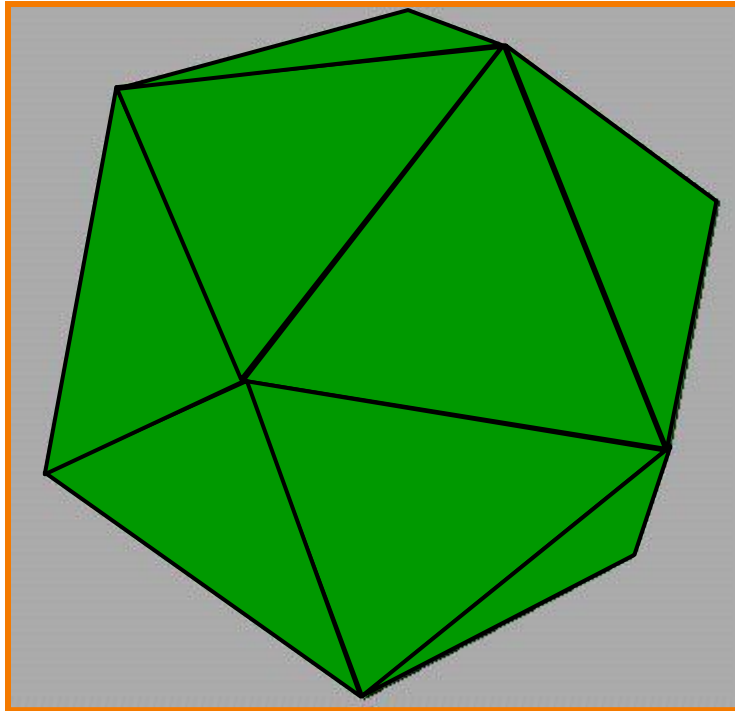
# Summary



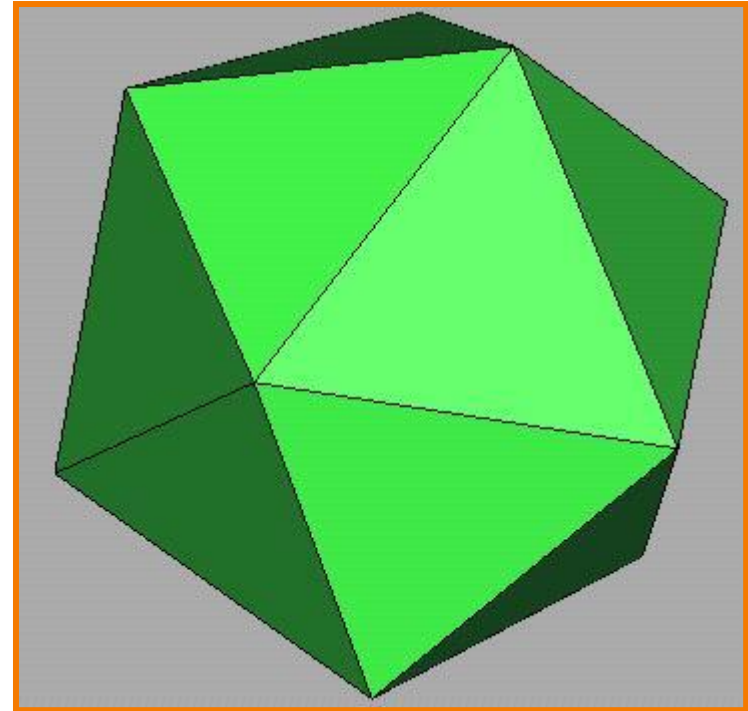
- Writing a simple ray casting renderer is easy
  - Generate rays
  - Intersection tests
  - Lighting calculations

```
R2Image *RayCast(R3Scene *scene, int width, int height)
{
 R2Image *image = new R2Image(width, height);
 for (int i = 0; i < width; i++) {
 for (int j = 0; j < height; j++) {
 R3Ray ray = ConstructRayThroughPixel(scene->camera, i, j);
 R3Rgb radiance = ComputeRadiance(scene, &ray);
 image->SetPixel(i, j, radiance);
 }
 }
 return image;
}
```

# Next Time is Illumination!



Without Illumination



With Illumination