



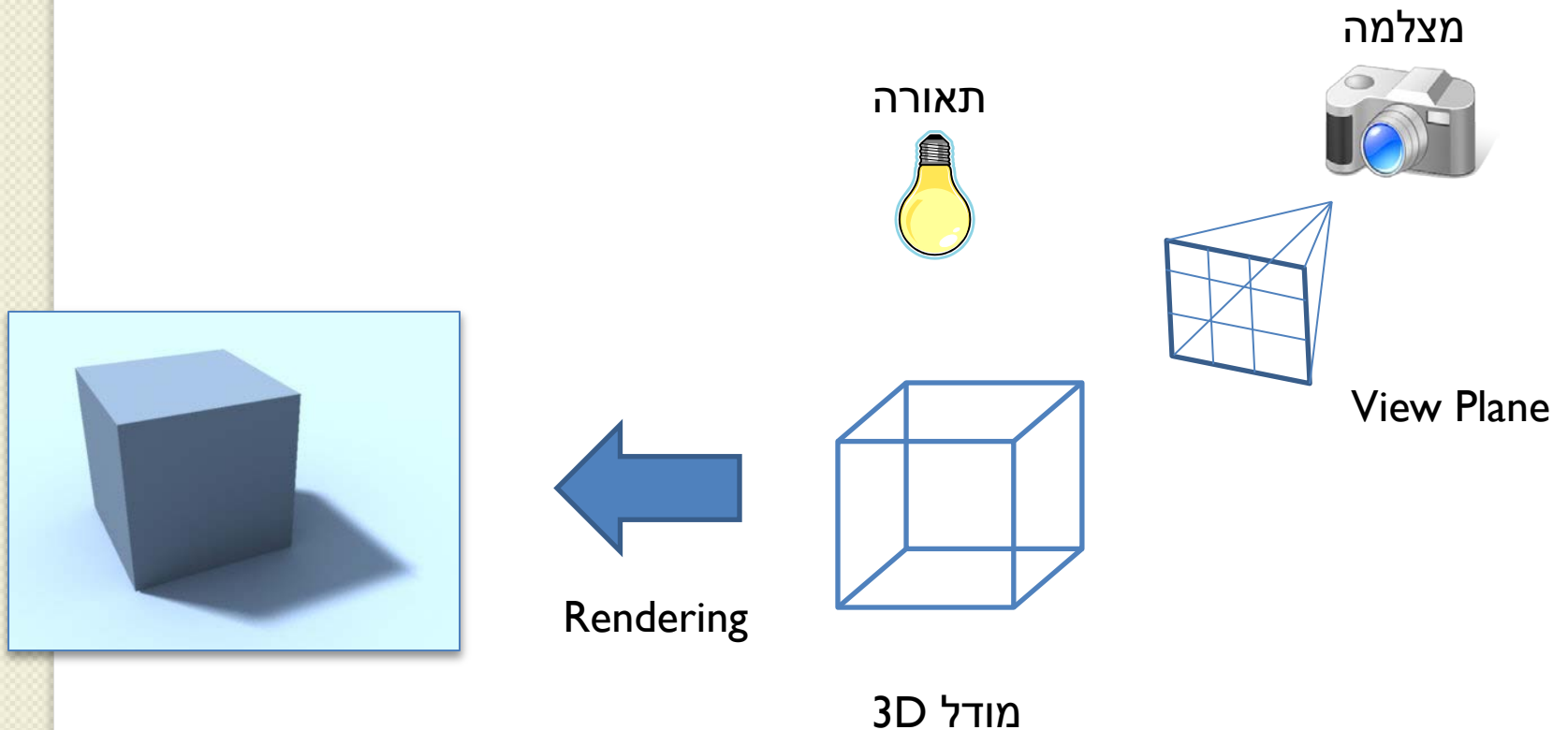
קורס גרפיקה ממוחשבת

2011/2012 סמסטר א'

Rendering

What is 3D rendering?

- Construct an image from a 3D model



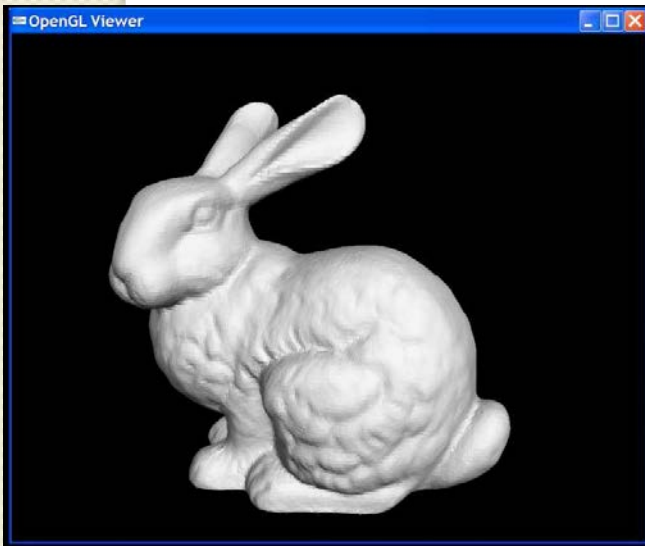
Rendering Scenarios

- אינטראקטיבי

- מייצרים תמונות בשבריר שנייה (לפחות 10 בשנייה) כאשר המשתמש שולט בפרמטרים של הרינדור

- יש צורך להשיג את האיכות הגבוהה ביותר בהתחשב בזמן הנתון (הקצב הנדרש)

- שימושי לויזואליזציות, משחקים וכו'



Rendering Scenarios

- אצווה (batch)

- כל תמונה מיוצרת ברמת פירוט גבוהה ככל
האפשר עבור סט ספציפי של פרמטרים

- לוקח כמה זמן שצריך

- שימושי לפוטוריאליזם, סרטים וכו'



3D Rendering Issues

- What does a 3D rendering system have to do?
 - Camera
 - Visible surface determination
 - Lights
 - Reflectance
 - Shadows
 - Indirect Illumination
 - Sampling
 - Etc.

Camera Models

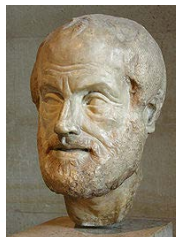
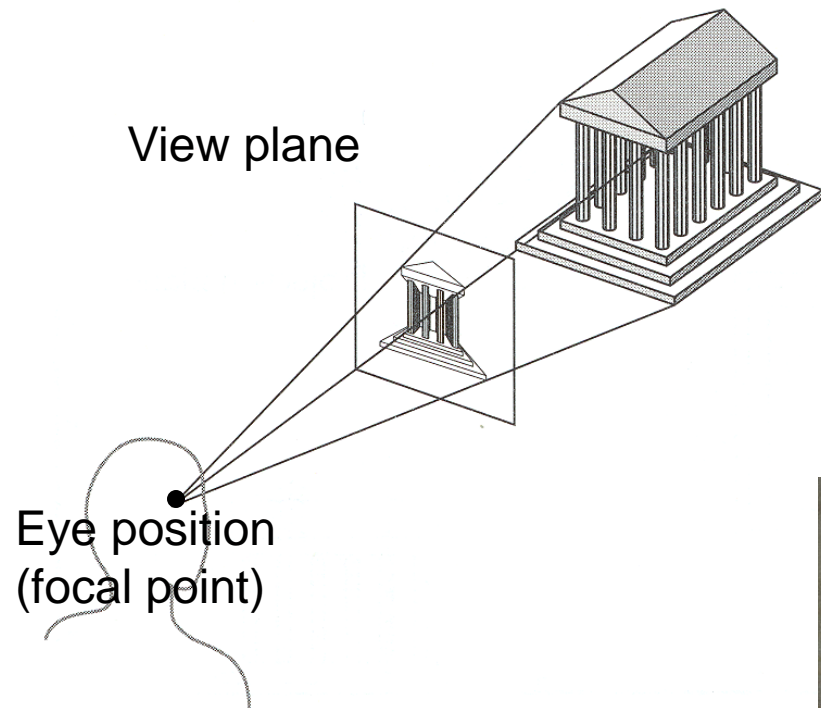
- The most common model is pin-hole camera
 - All captured light rays arrive along paths toward focal point without lens distortion (everything is in focus)
 - Sensor response proportional to radiance

Other models consider ...

Depth of field

Motion blur

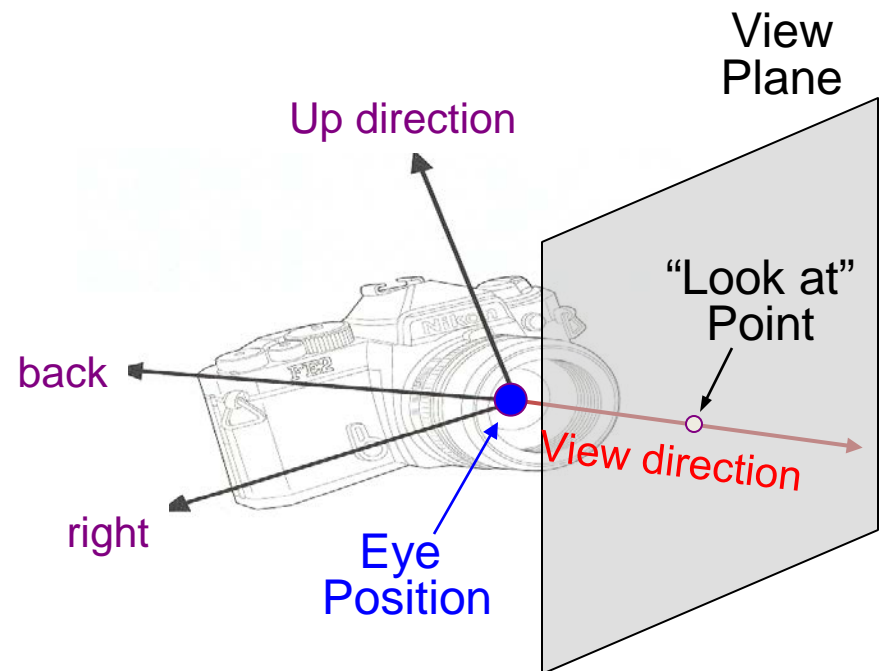
Lens distortion



aristotle

Camera Parameters

- Position
 - Eye position (p_x, p_y, p_z)
- Orientation
 - View direction (dx, dy, dz)
 - Up direction (ux, uy, uz)
- Aperture
 - Field of view ($xfov, yfov$)
- Film plane
 - “Look at” point
 - View plane normal

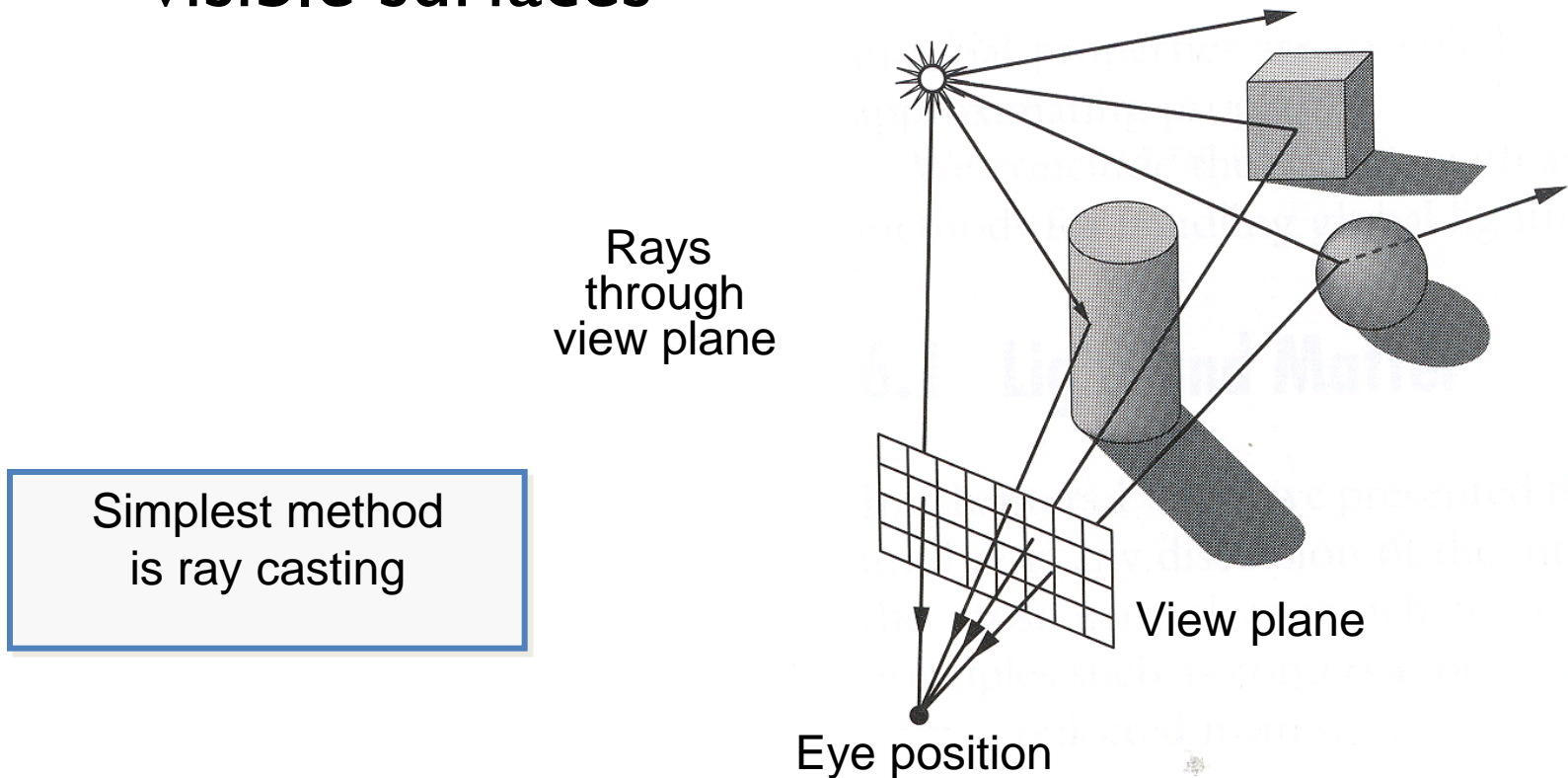


3D Rendering Issues

- What does a 3D rendering system have to do?
 - Camera
 - Visible surface determination
 - Lights
 - Reflectance
 - Shadows
 - Indirect Illumination
 - Sampling
 - Etc.

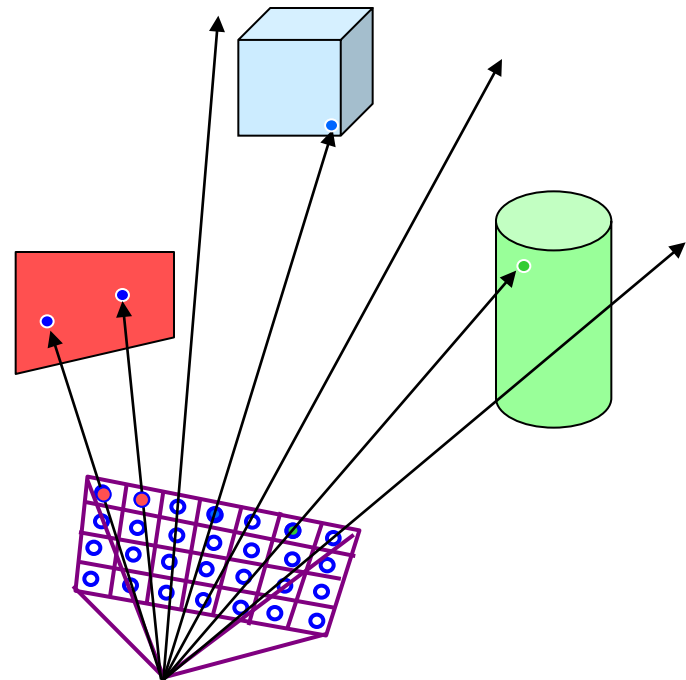
Visible Surface Determination

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



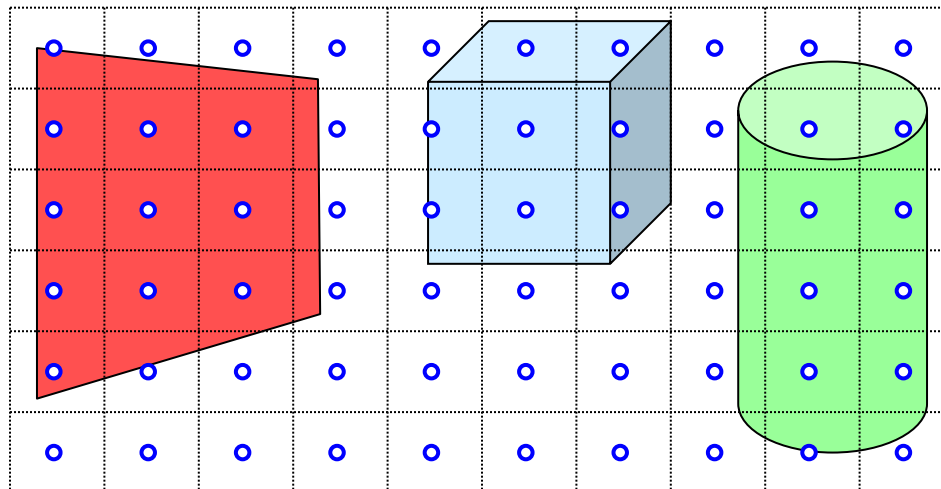
Ray Casting

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



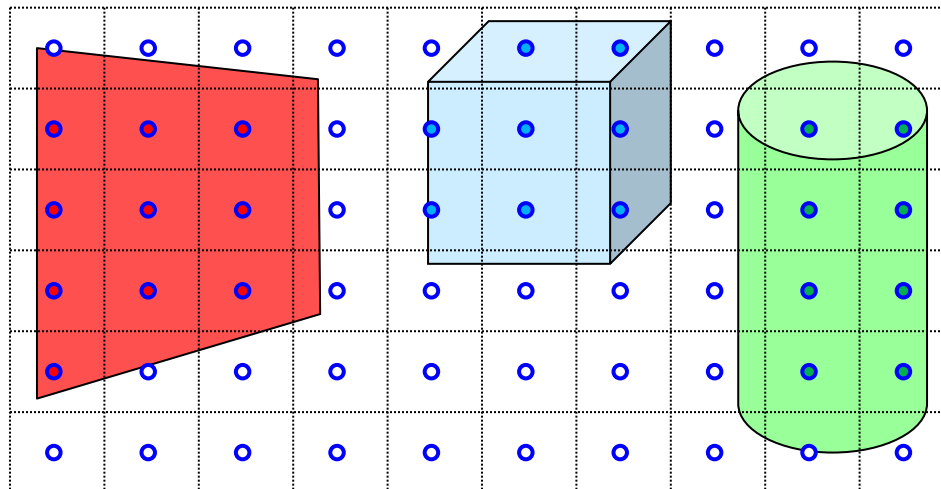
Ray Casting

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



Ray Casting

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

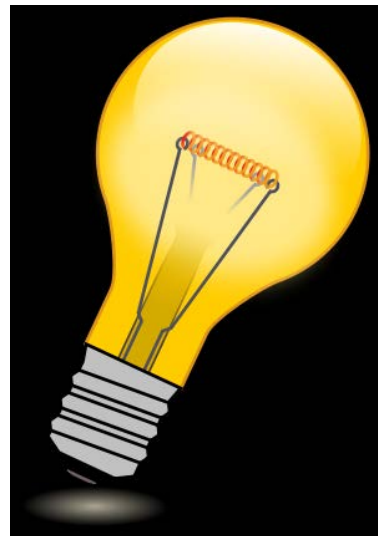
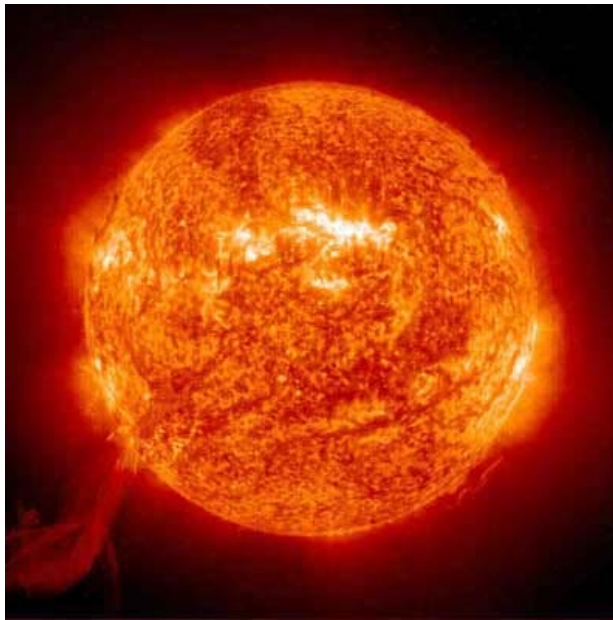


3D Rendering Issues

- What does a 3D rendering system have to do?
 - Camera
 - Visible surface determination
 - Lights
 - Reflectance
 - Shadows
 - Indirect Illumination
 - Sampling
 - Etc.

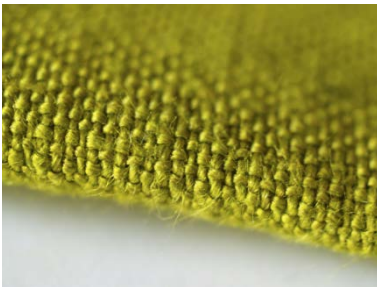
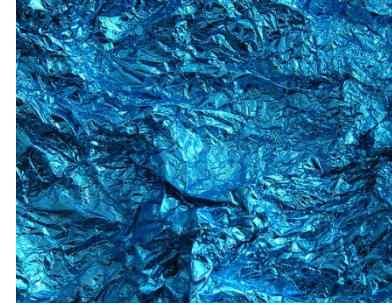
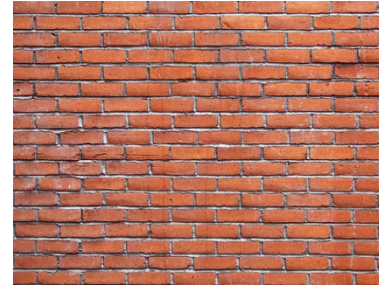
Lights and Surfaces

- Different Lights



Lights and Surfaces

- Different Materials



Other Lighting Parameters

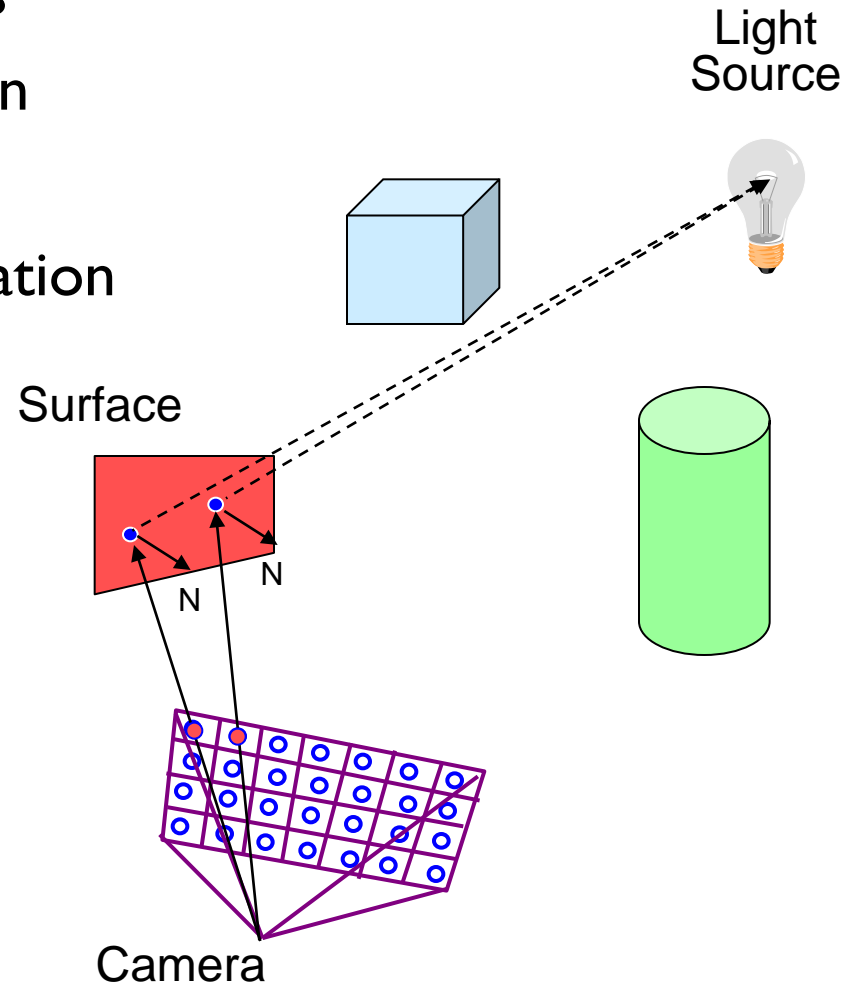
- Atmospheric attenuation



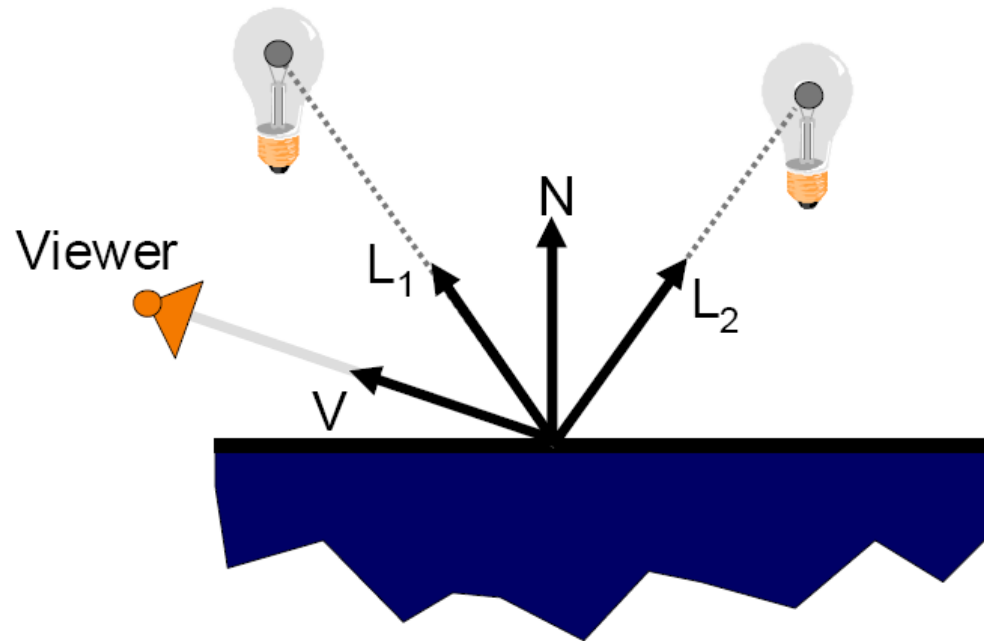
- Camera response

Lighting Simulation

- Lighting parameters
 - Light source emission
 - Surface reflectance
 - Atmospheric attenuation
 - Camera response

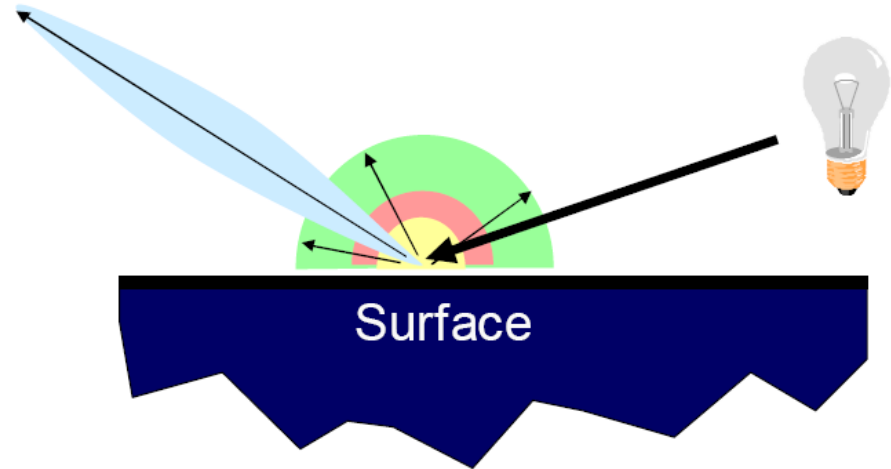


Lighting Simulation

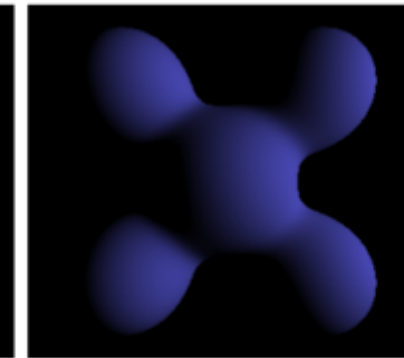


Phong Reflectance Model

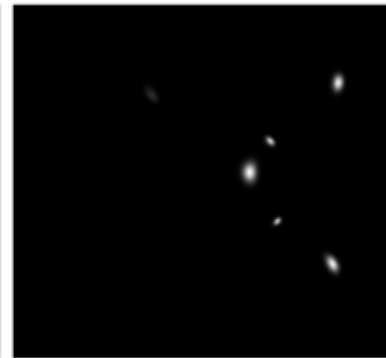
- Simple analytic model
 - Diffuse reflection+
 - Specular reflection+
 - Emission+
 - “ambient”



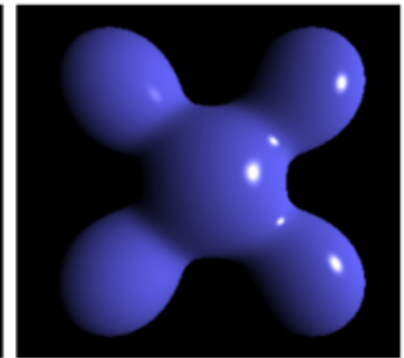
Ambient



Diffuse



Specular



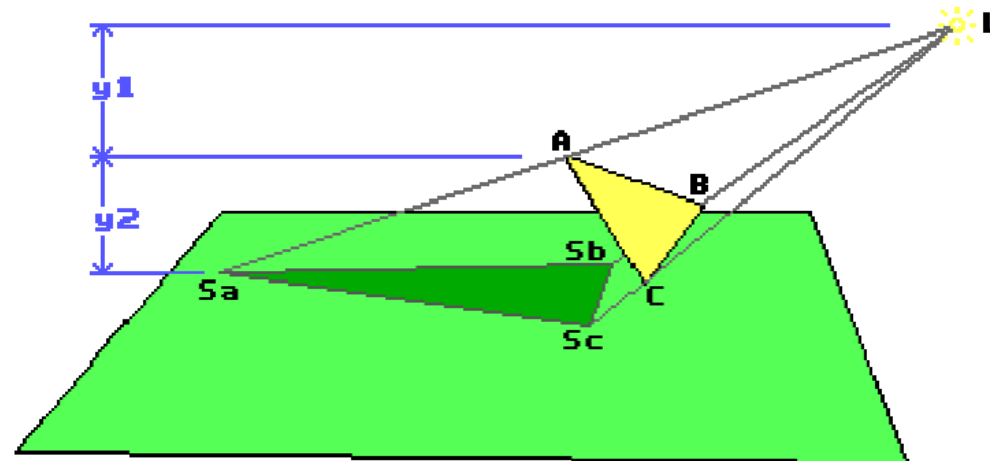
= Phong Reflection

3D Rendering Issues

- What does a 3D rendering system have to do?
 - Camera
 - Visible surface determination
 - Lights
 - Reflectance
 - **Shadows**
 - Indirect Illumination
 - Sampling
 - Etc.

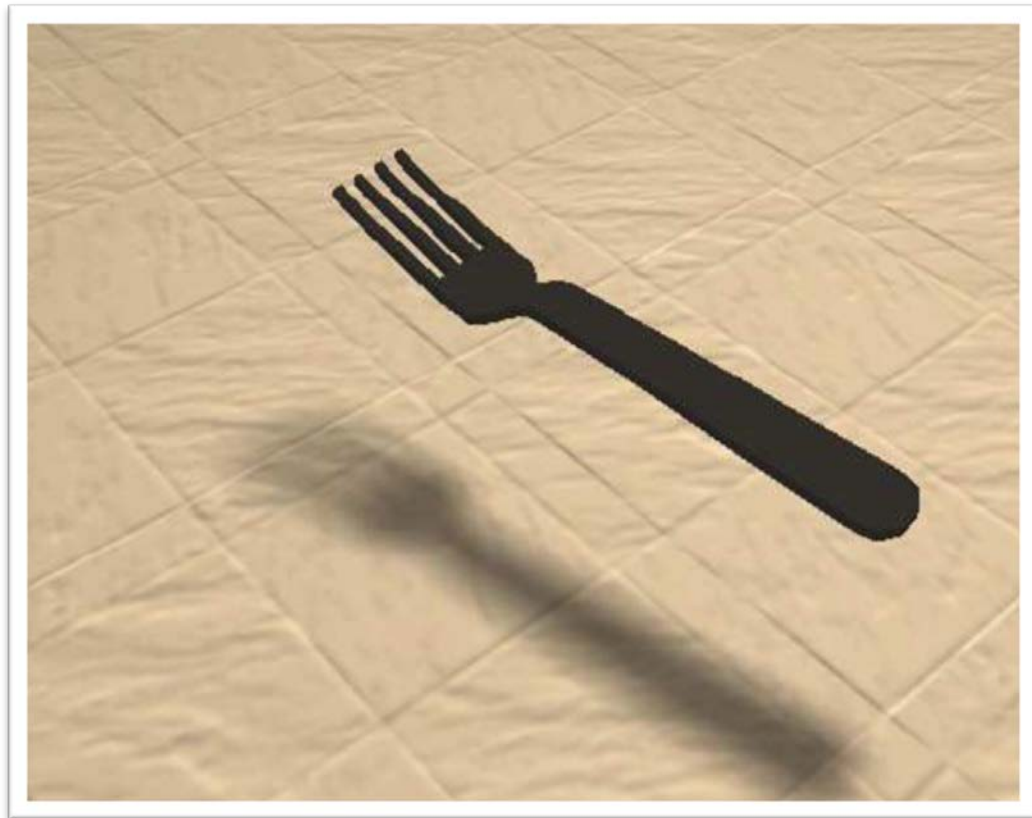
Shadows

- Occlusions from light sources

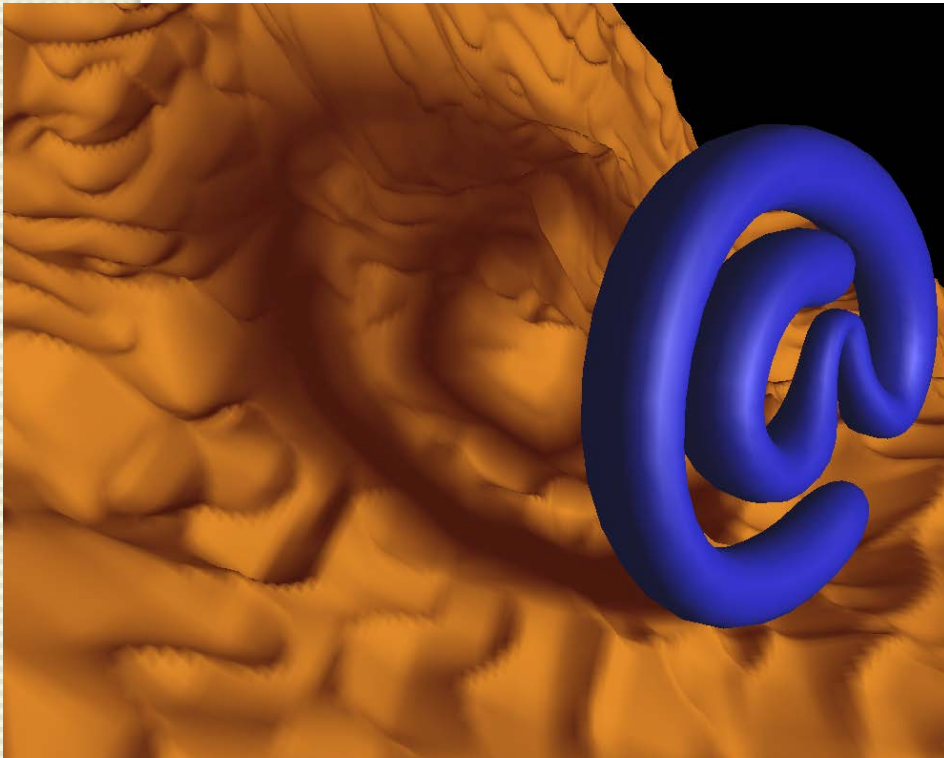
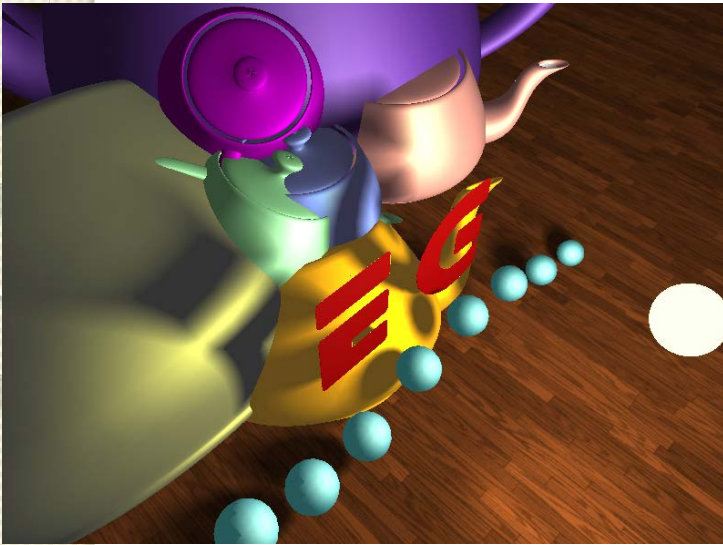


Shadows

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



Shadows

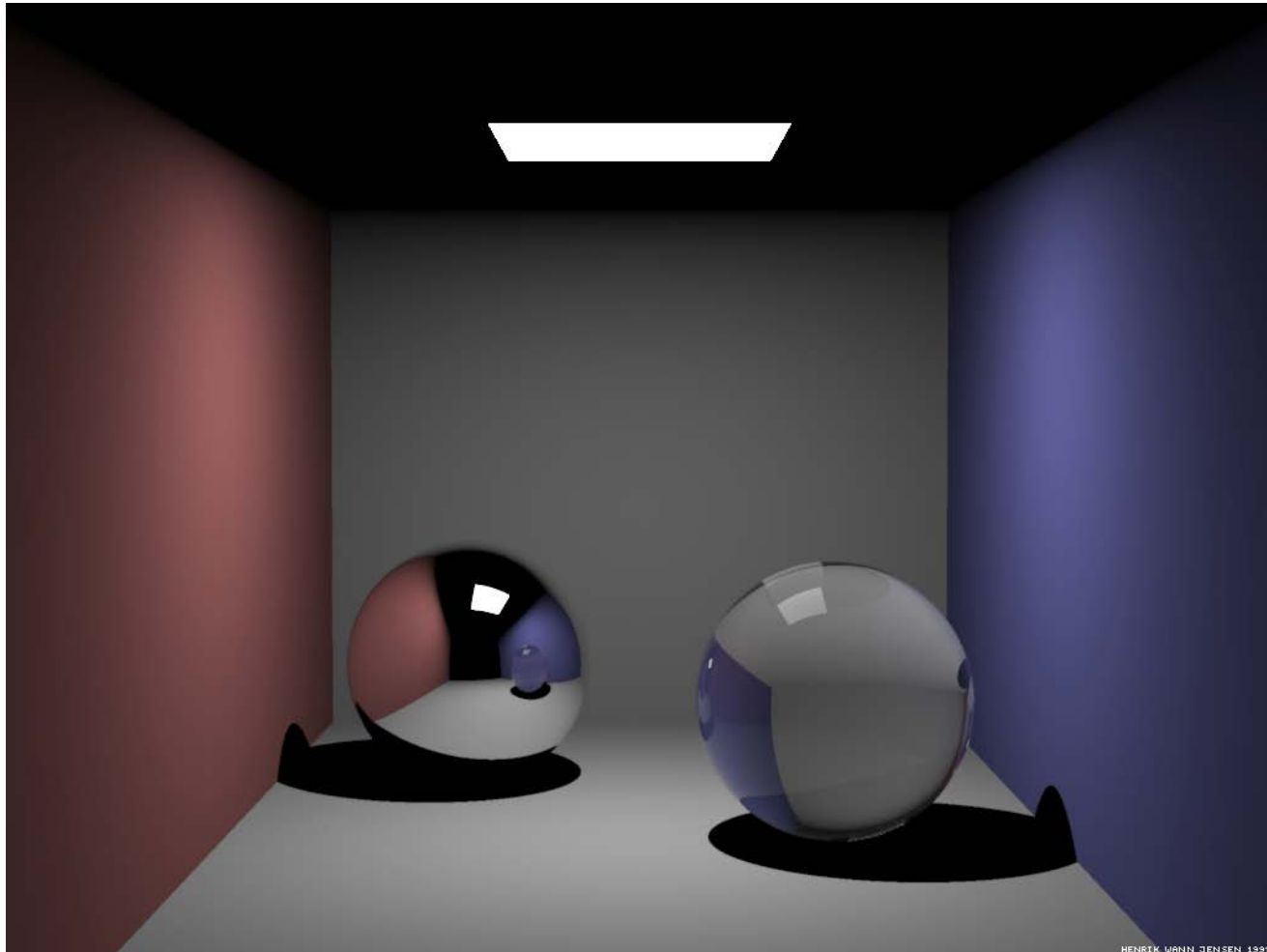


3D Rendering Issues

- What does a 3D rendering system have to do?
 - Camera
 - Visible surface determination
 - Lights
 - Reflectance
 - Shadows
 - Indirect Illumination
 - Sampling
 - Etc.

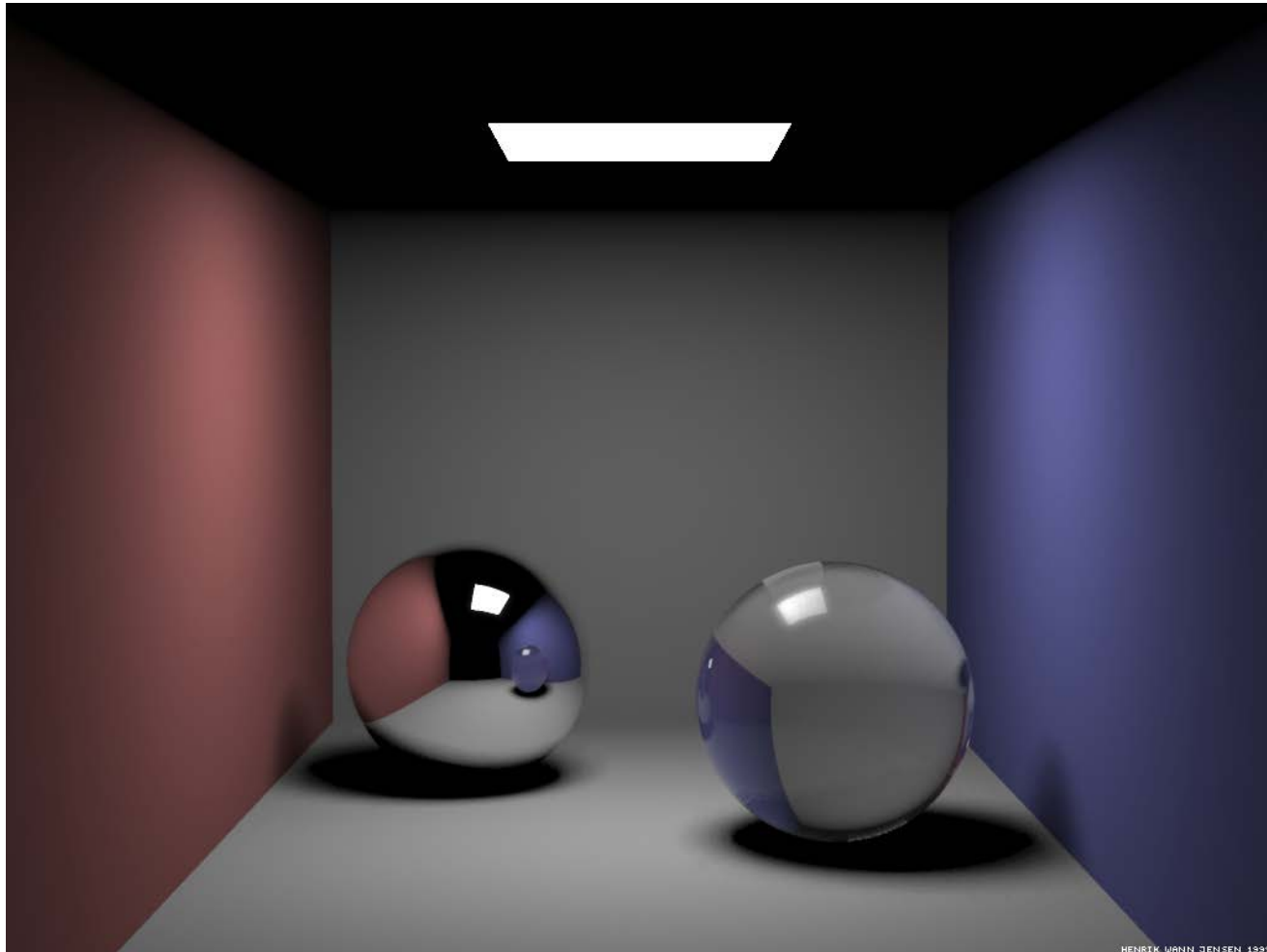
Path Types

Direct diffuse + indirect specular and transmission



Path Types

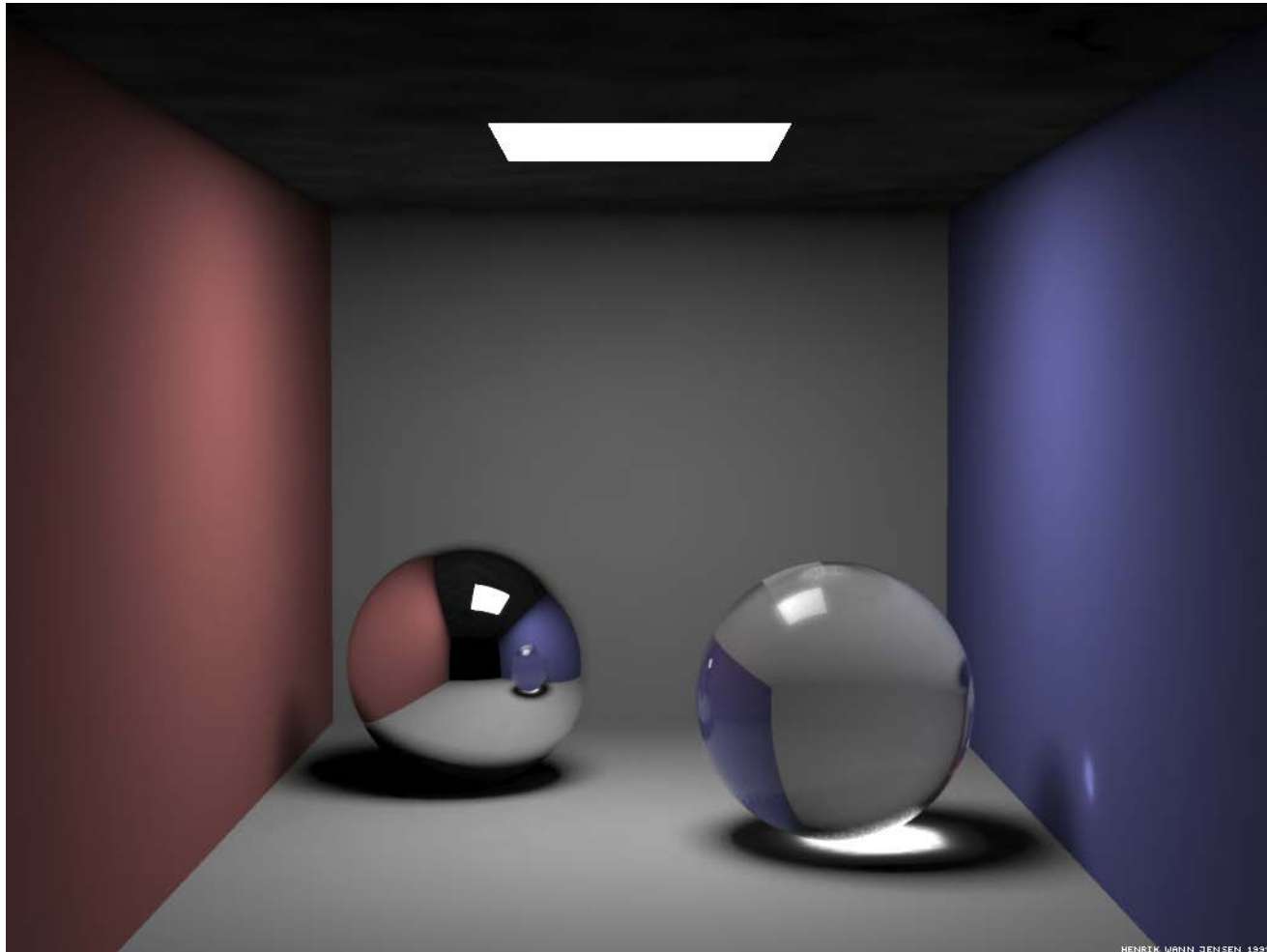
+ Soft Shadows



HENRIK WANN JENSEN 1999

Path Types

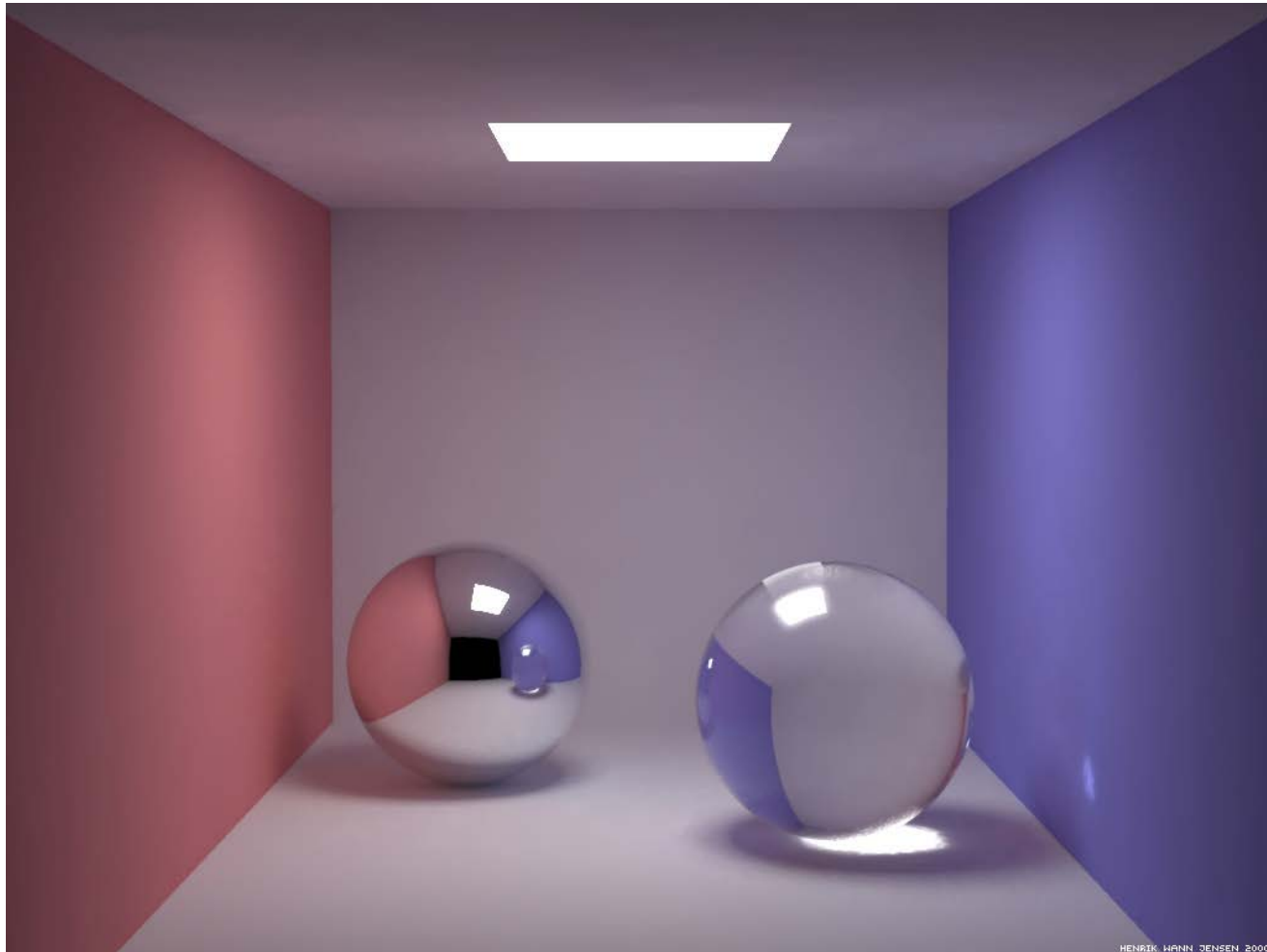
+ caustics



HENRIK WANN JENSEN 1999

Path Types

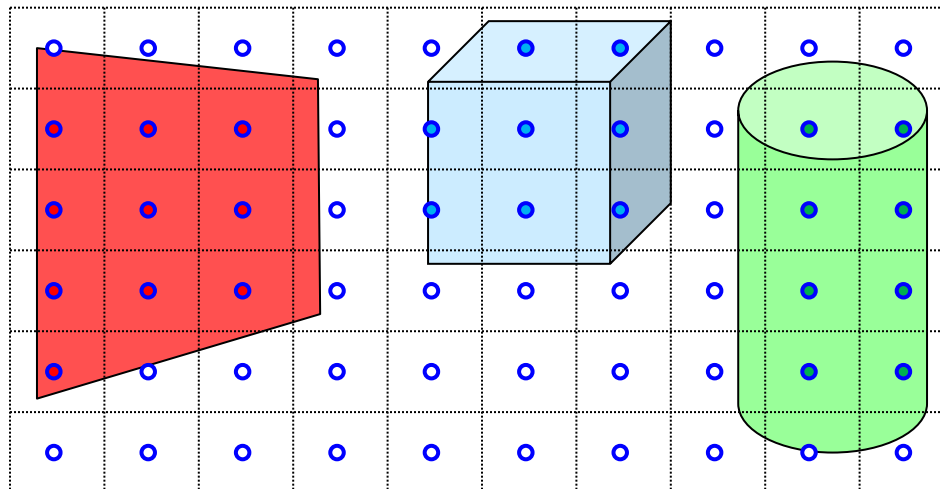
+ indirect diffuse illumination



3D Rendering Issues

- What does a 3D rendering system have to do?
 - Camera
 - Visible surface determination
 - Lights
 - Reflectance
 - Shadows
 - Indirect Illumination
 - Sampling
 - Etc.

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

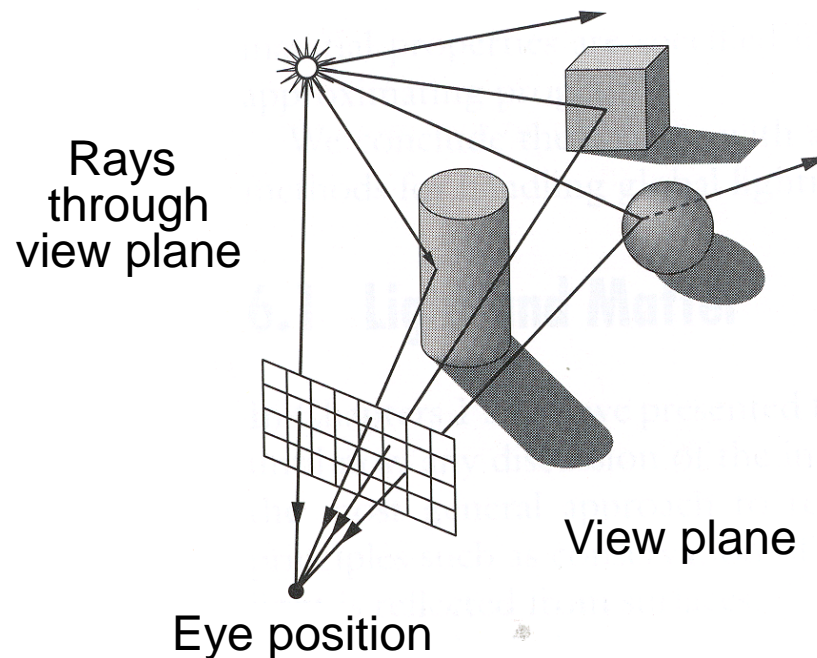




Ray Casting

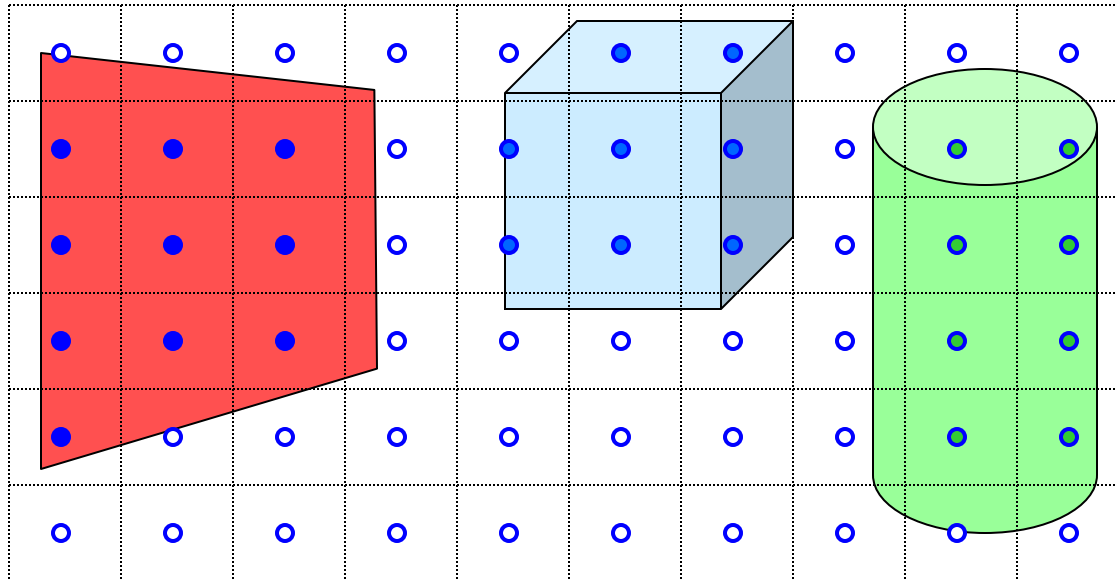
3D Rendering

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



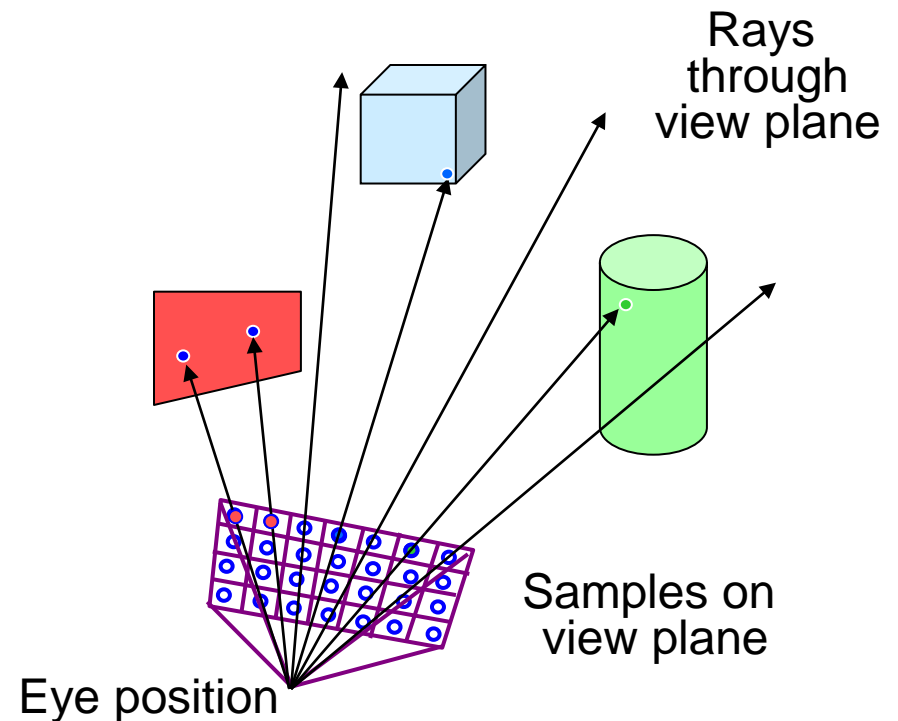
Ray Casting

- 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

- 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

- Simple implementation:

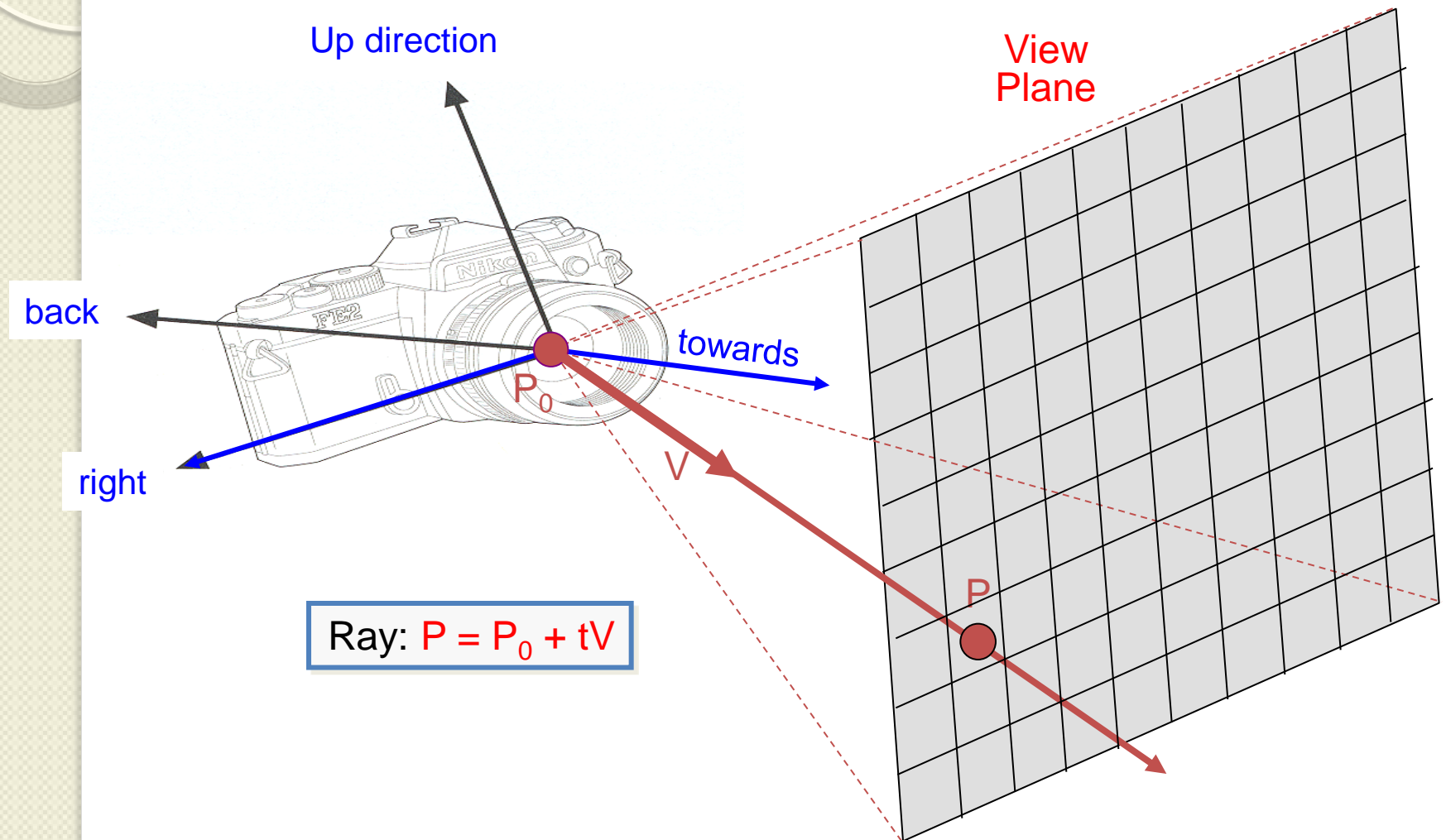
```
Image RayCast(Camera camera, Scene scene, int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(camera, i, j);
            Intersection hit = FindIntersection(ray,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

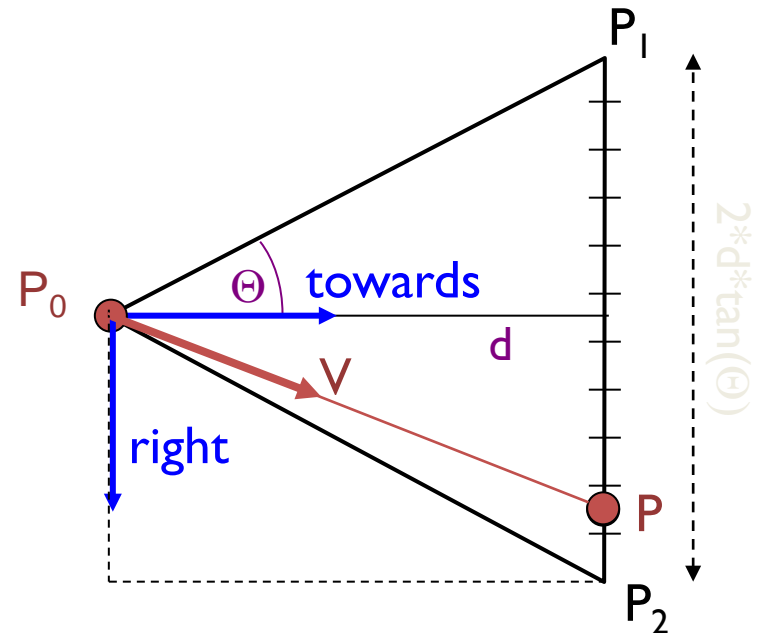
$$P_1 = P_0 + d * \text{towards} - d * \tan(\Theta) * \text{right}$$

$$P_2 = P_0 + d * \text{towards} + d * \tan(\Theta) * \text{right}$$

$$P = P_1 + (i/\text{width} - 0.5) * 2 * d * \tan(\Theta) * \text{right}$$

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

$$i \in \left(-\frac{\text{width}}{2}, \frac{\text{width}}{2} \right)$$



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;
}
```

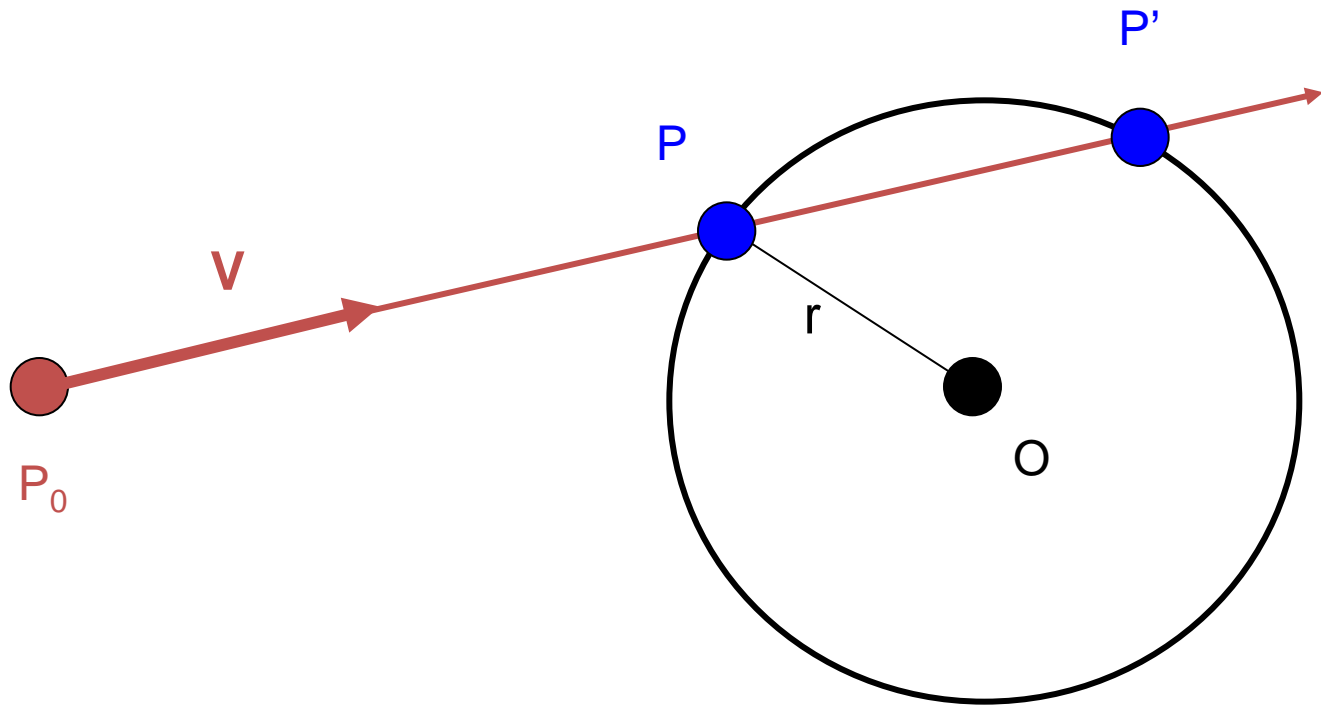
Ray-Scene Intersection

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



Ray-Sphere Intersection I

Ray: $P = P_0 + tV$

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

Algebraic Method

Substituting for P, we get:

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

Solve quadratic equation:

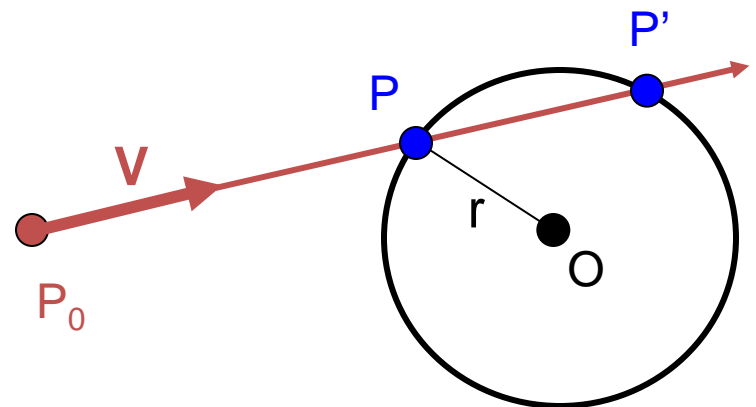
$$at^2 + bt + c = 0$$

where:

$$a = 1$$

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

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



$$P = P_0 + tV$$

Ray-Sphere Intersection II

Ray: $P = P_0 + tV$

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

Geometric Method

$L = O - P_0$

$t_{ca} = L \cdot V$

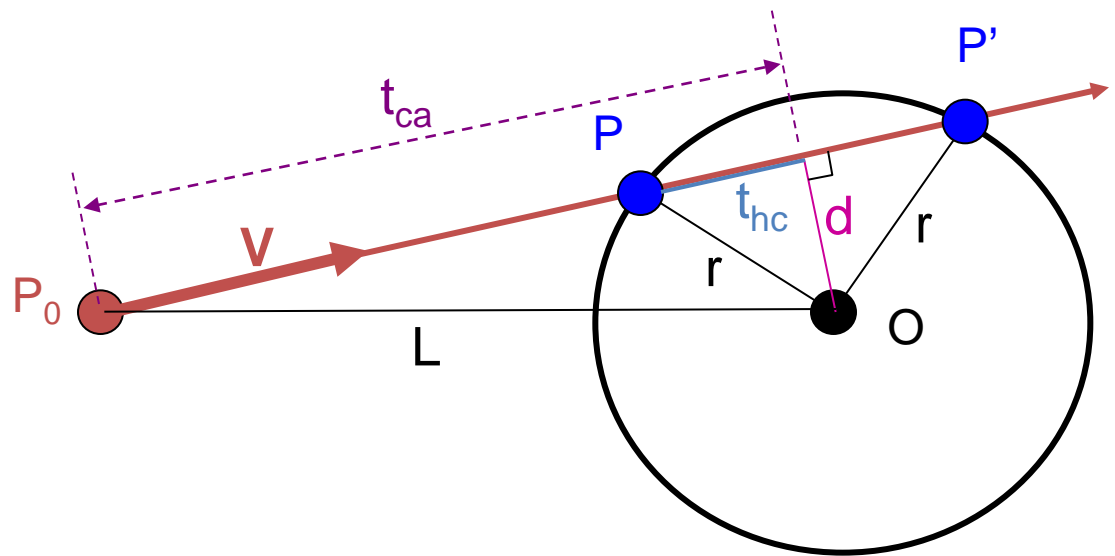
if ($t_{ca} < 0$) return 0

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

if ($d^2 > r^2$) return 0

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

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

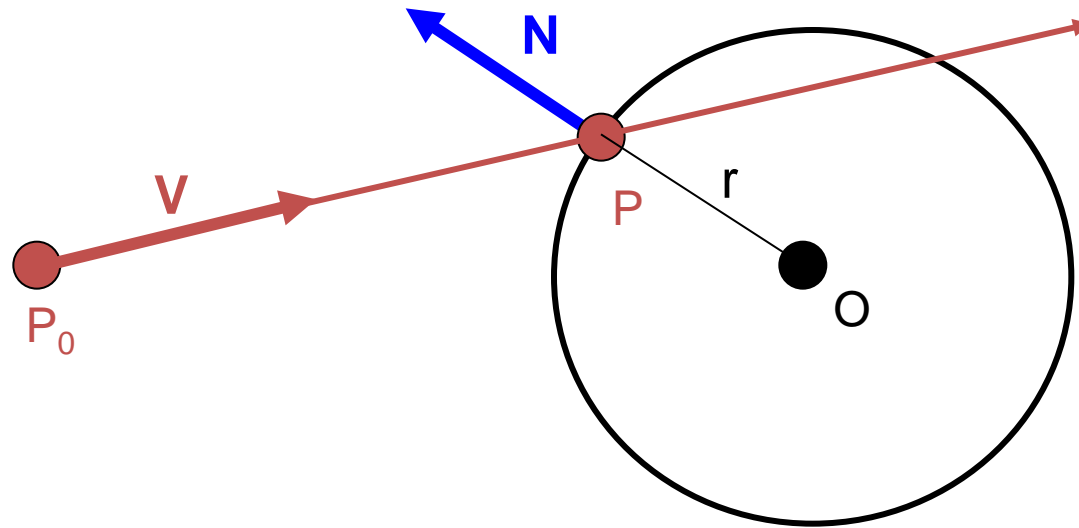


$P = P_0 + tV$

Ray-Sphere Intersection

- Need normal vector at intersection for lighting calculations

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

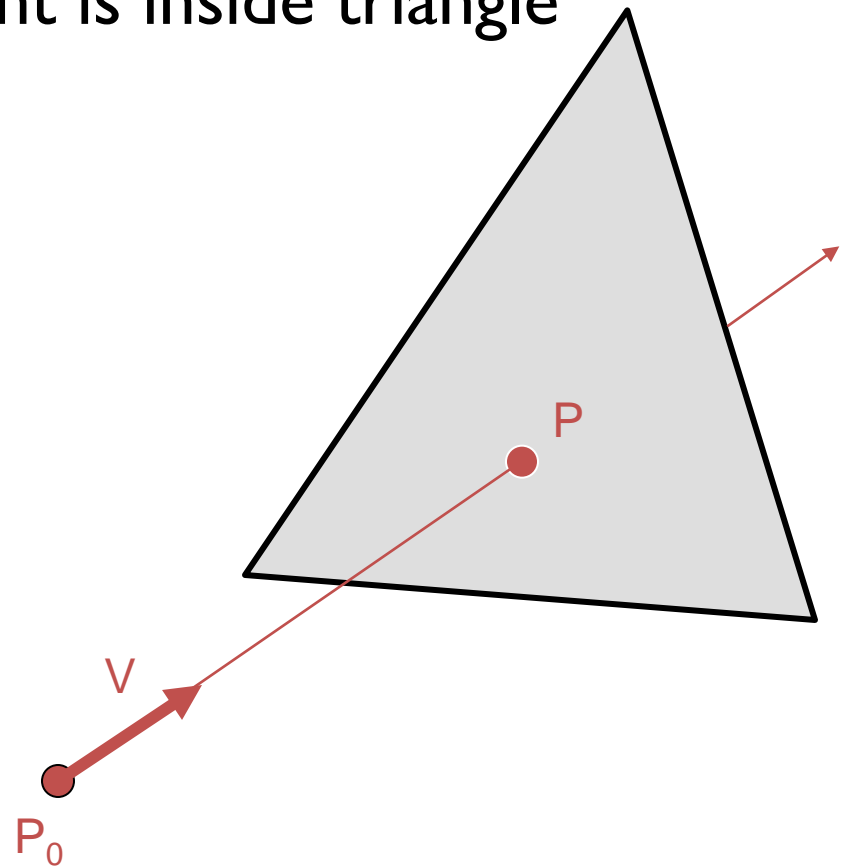


Ray-Scene Intersection

- 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



Algebraic Method

Ray-Plane Intersection

Ray: $P = P_0 + tV$

Plane: $N(P - P_0) = 0 \rightarrow P \cdot N + c = 0$

Substituting for P , we get:

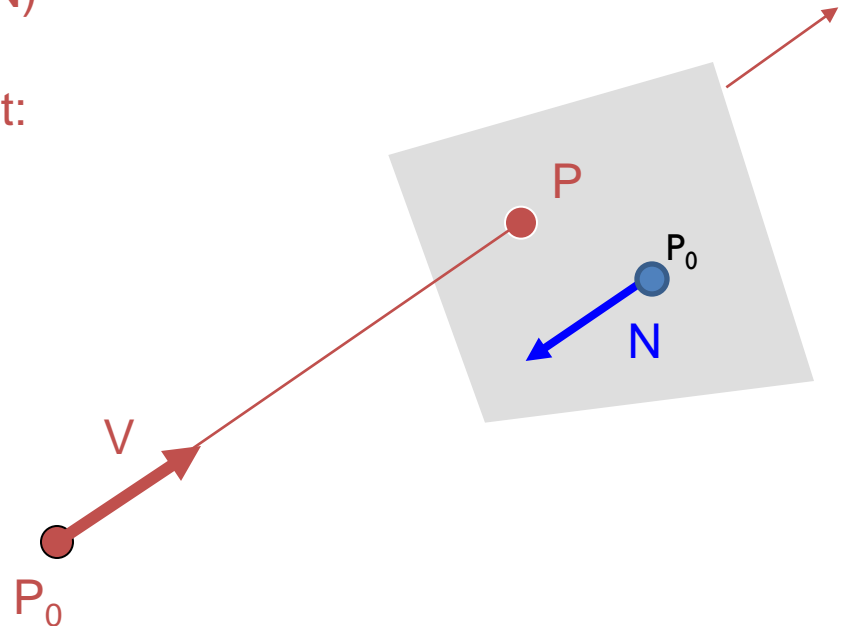
$$(P_0 + tV) \cdot N + c = 0$$

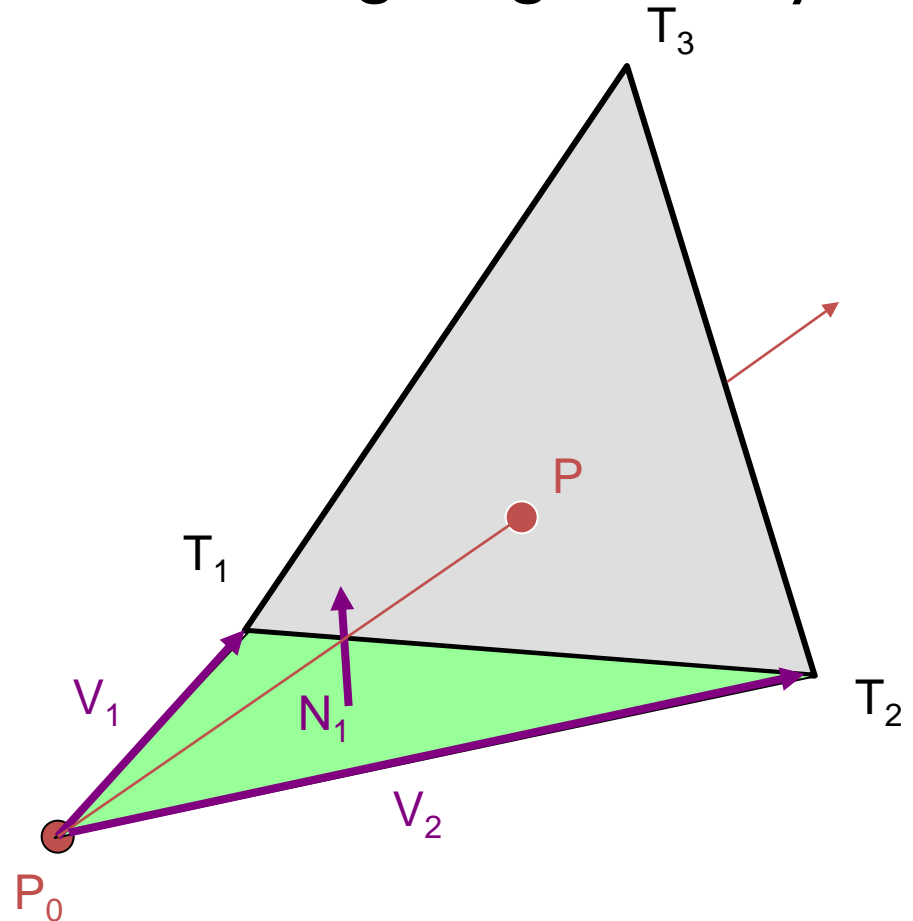
Solution:

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

And the intersection at:

$$P = P_0 + tV$$





Ray-Triangle Intersection II

- Check if point is inside triangle parametrically

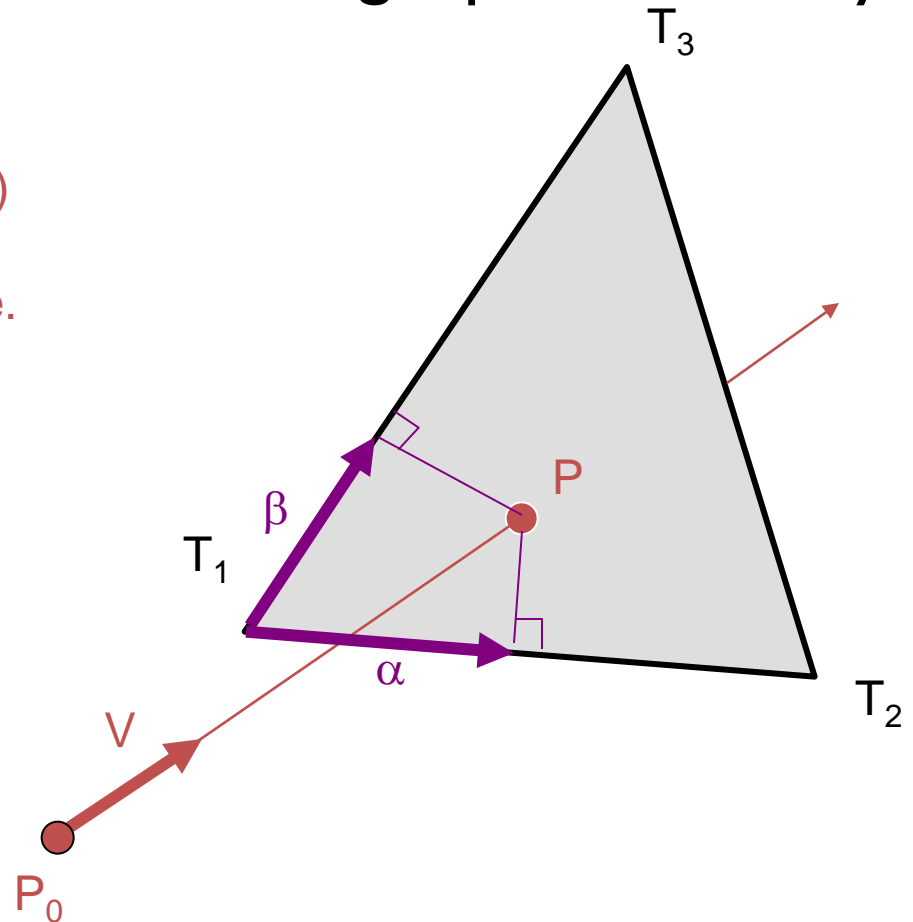
Compute α, β :

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

Check if point inside triangle.

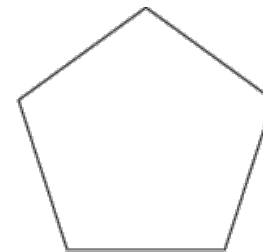
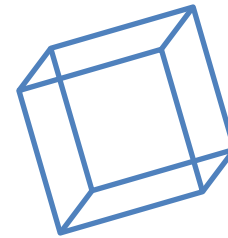
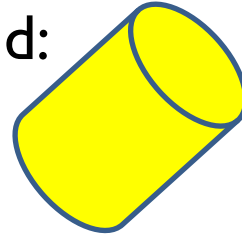
$$0 \leq \alpha \leq 1 \text{ and } 0 \leq \beta \leq 1$$

$$\alpha + \beta \leq 1$$



Other Ray-Primitive Intersections

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



convex polygon



concave polygon

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 < min_t) then
```

```
            min_primitive = primitive
```

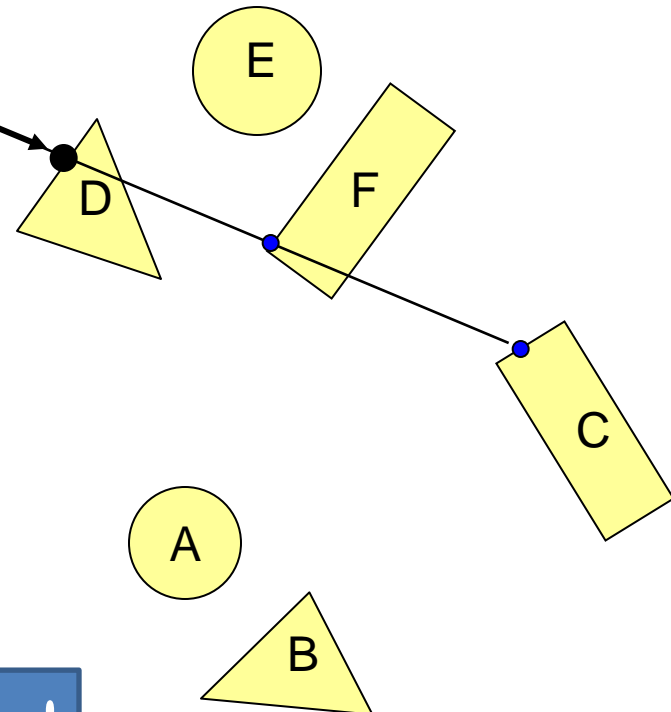
```
            min_t = t
```

```
        }
```

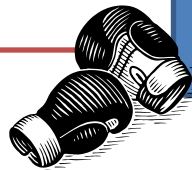
```
    }
```

```
    return Intersection(min_t, min_primitive)
```

```
}
```



Brute Force!

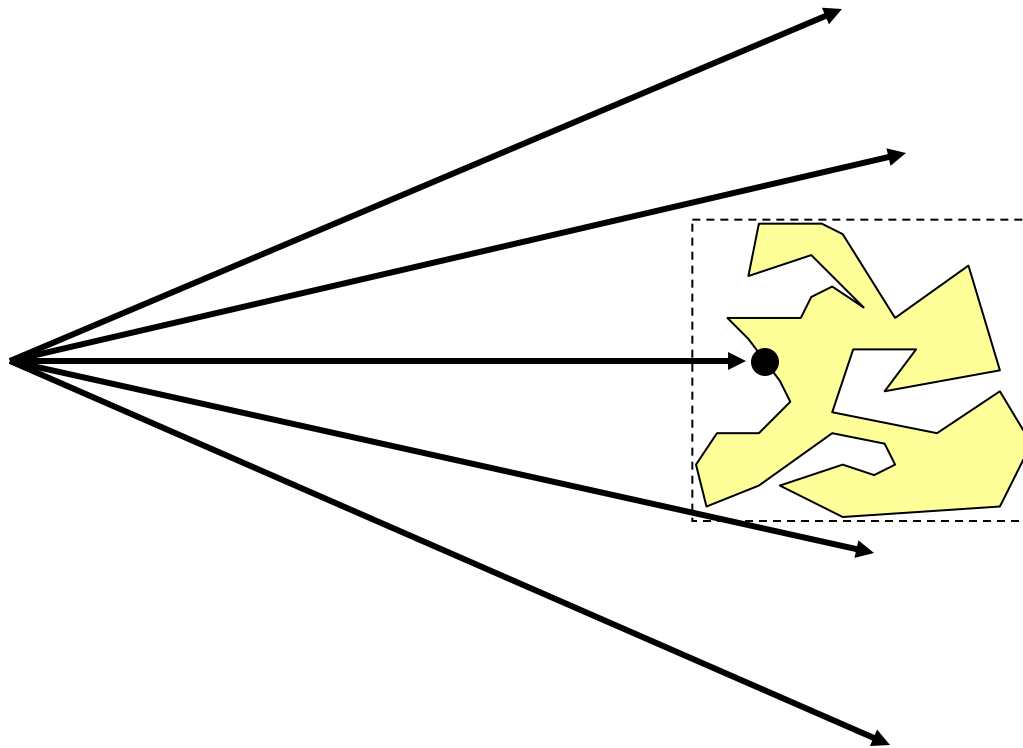


Ray-Scene Intersection

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

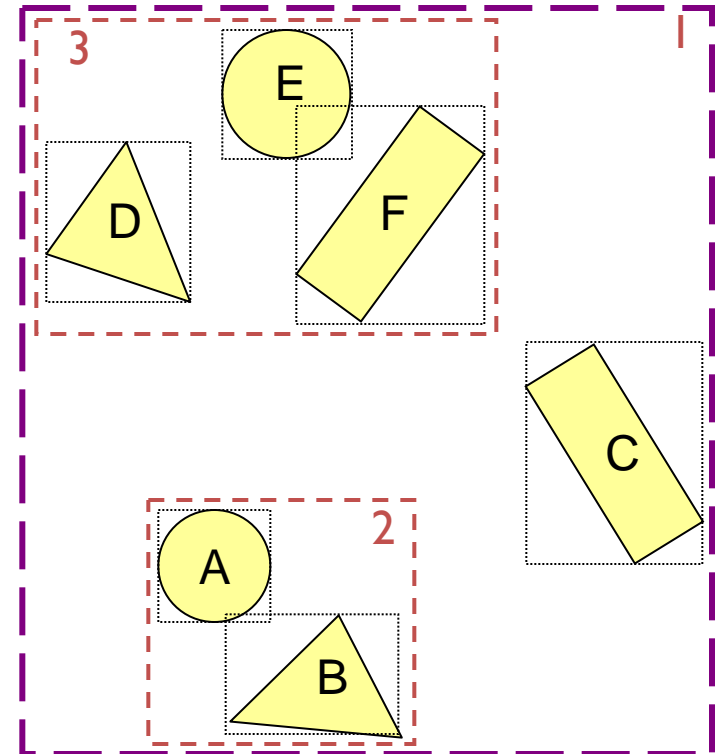
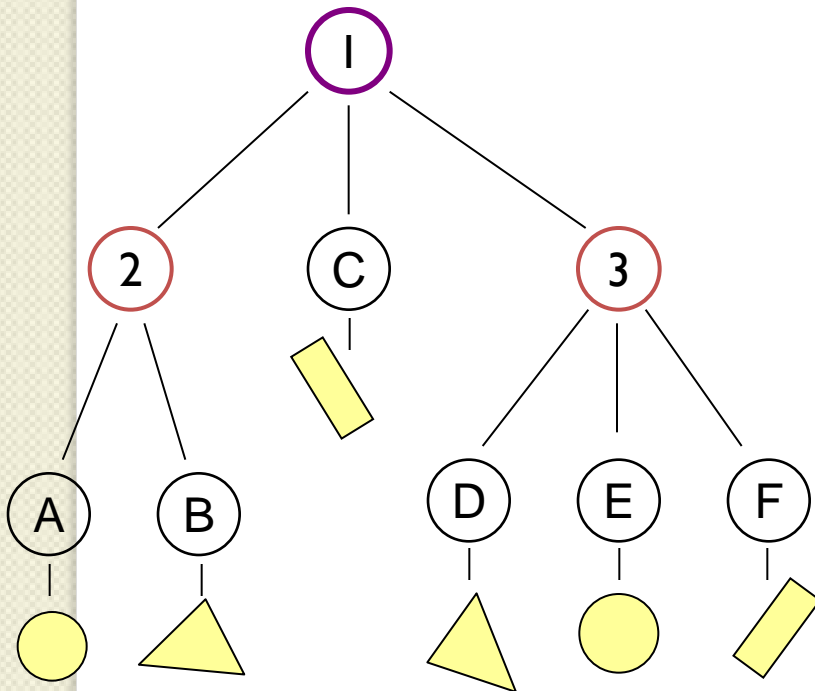
Bounding Volumes

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



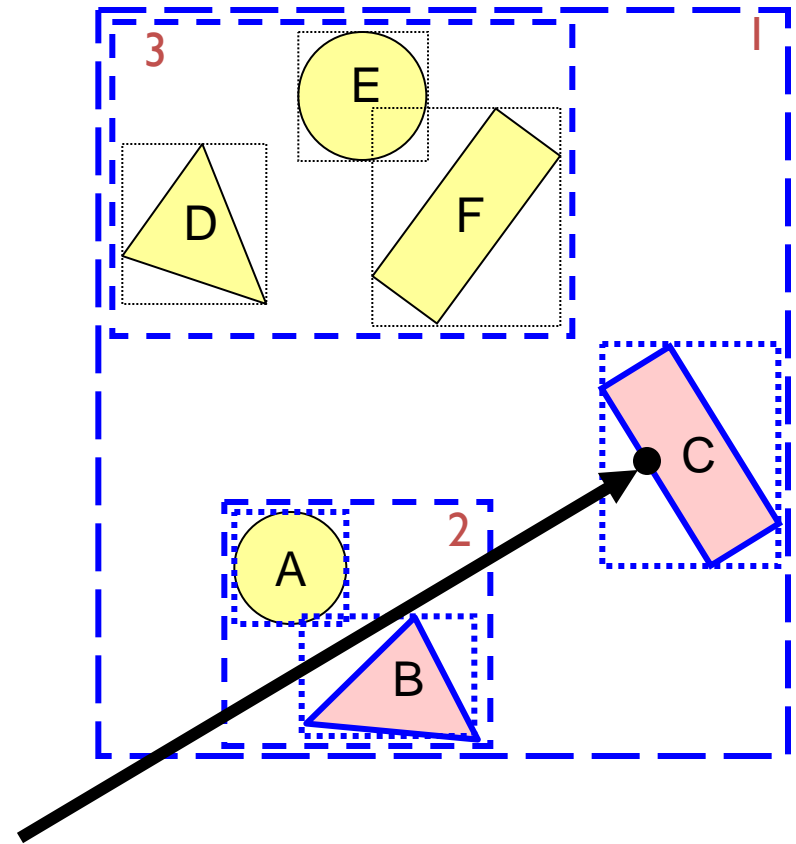
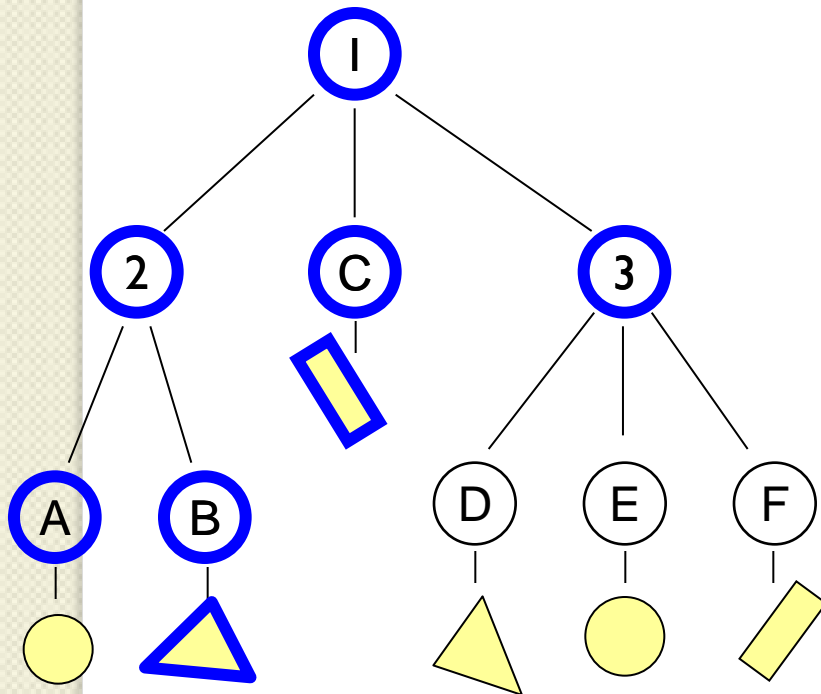
Bounding Volume Hierarchies I

- 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

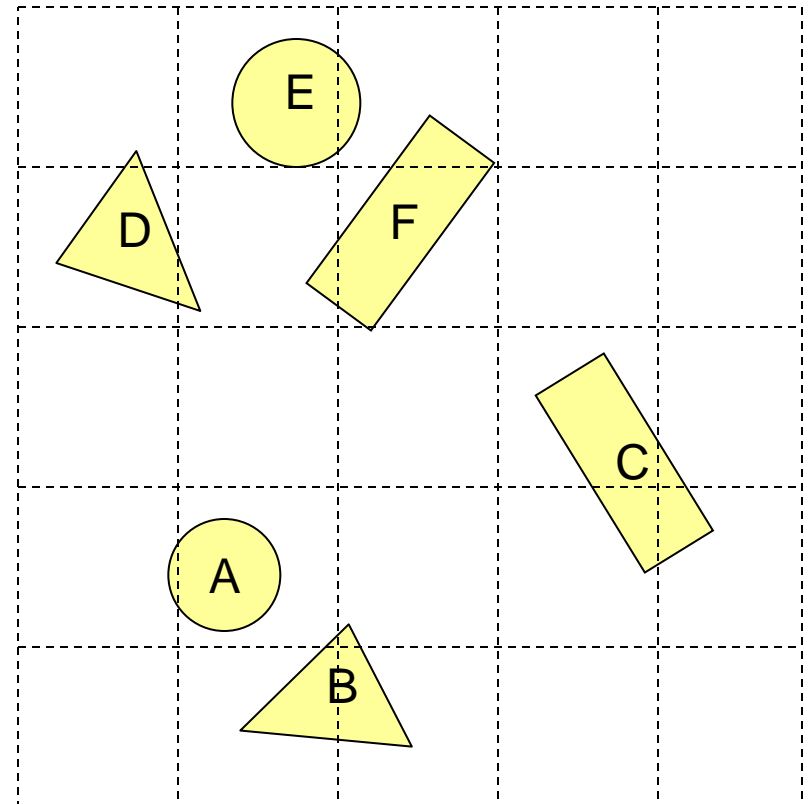
```
FindIntersection(Ray ray, Node node)
{
    // Find intersections with child node bounding volumes
    ...
    // Sort intersections front to back
    ...
    // Process intersections (checking for early termination)
    min_t = infinity;
    for each intersected child i {
        if (min_t < bv_t[i])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

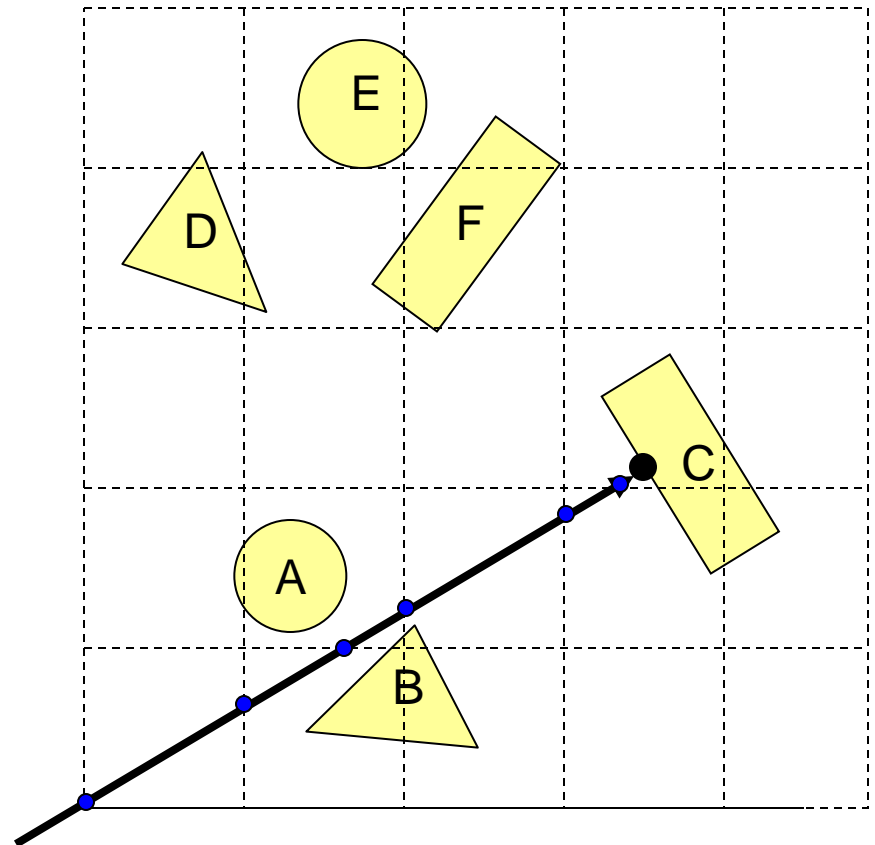


Uniform Grid

- Trace rays through grid cells
 - Fast
 - Incremental

Only check primitives
in intersected grid cells

Given an entry point into a cell
and a vector, its easy to
calculate exit point

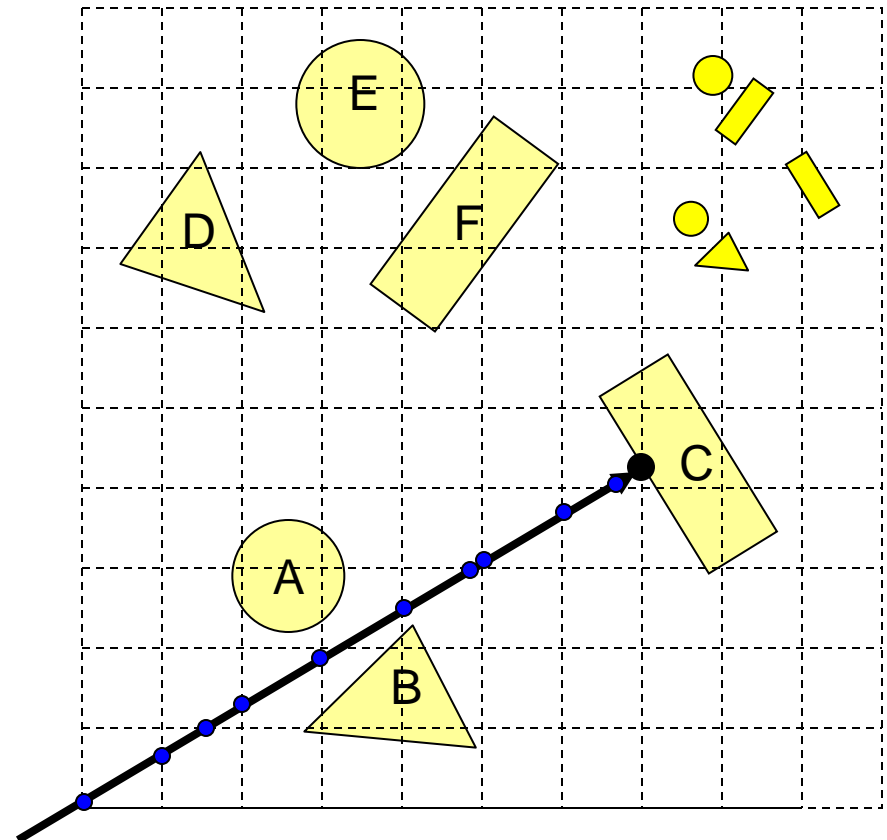


Uniform Grid

- Potential problem:
 - How to choose a suitable grid resolution?

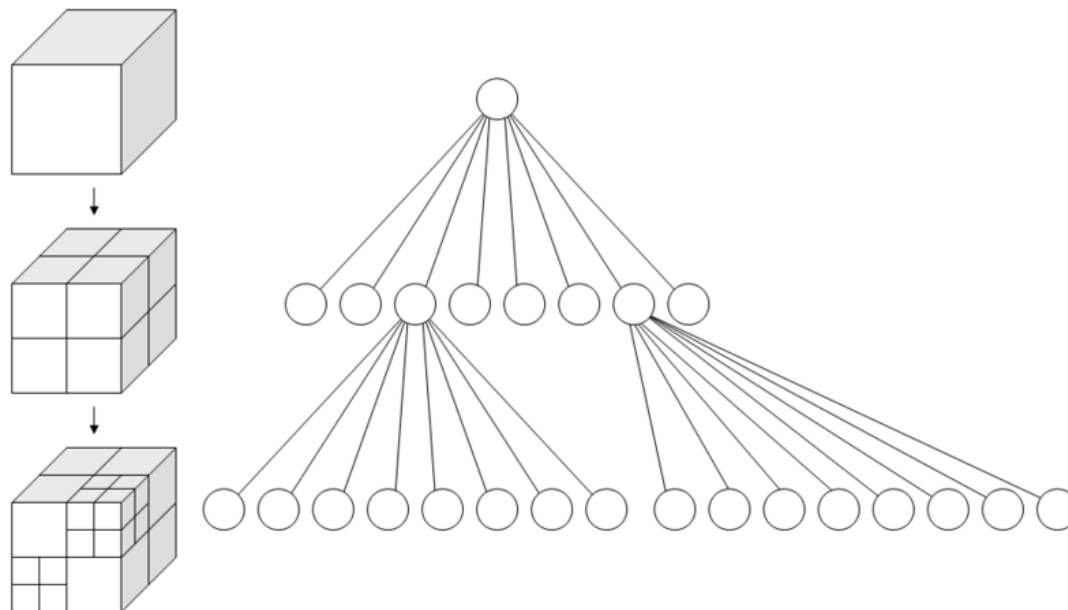
No/Little benefit
if grid is too coarse

Too costly
if grid is too fine



Octree

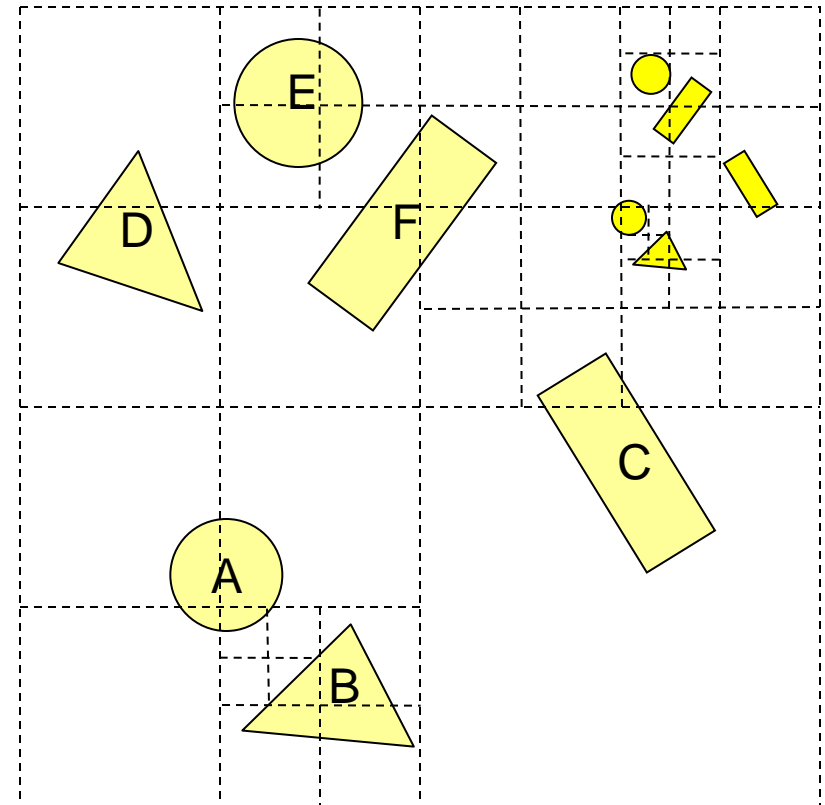
- A tree data structure used to partition three dimensional space
- A 2D version is called a *Quadtree*



Octree

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

Generally fewer cells

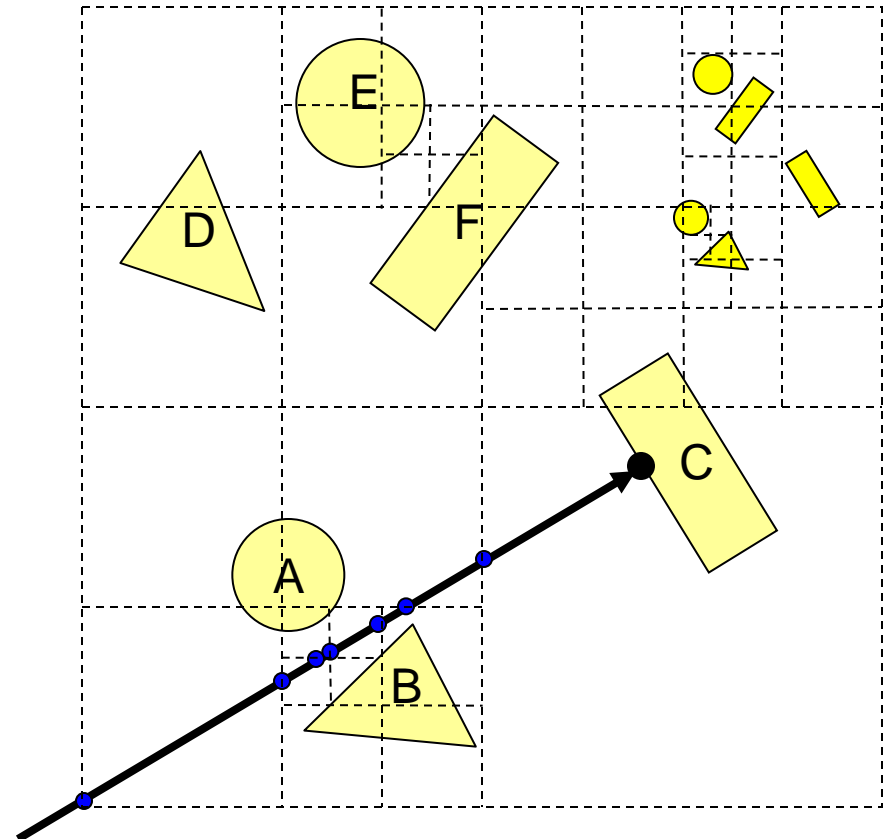


Quadtree

Octree

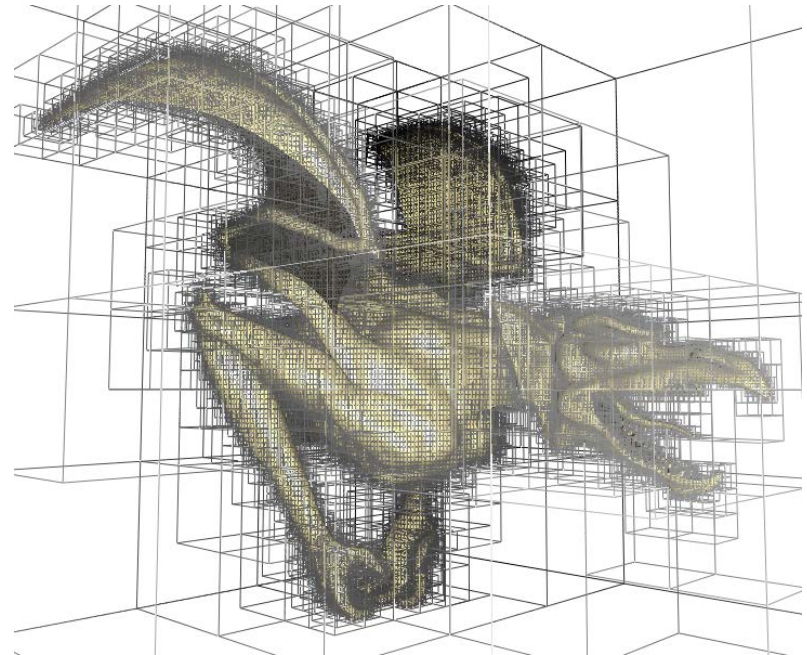
- Trace rays through neighbor cells
 - Fewer cells
 - More complex neighbor finding

Trade-off fewer cells for more expensive traversal



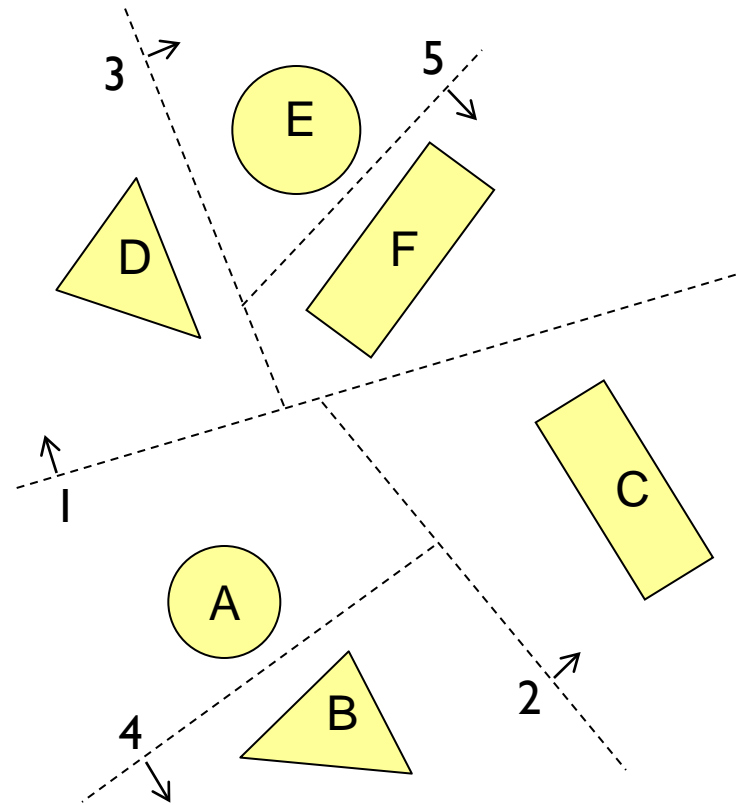
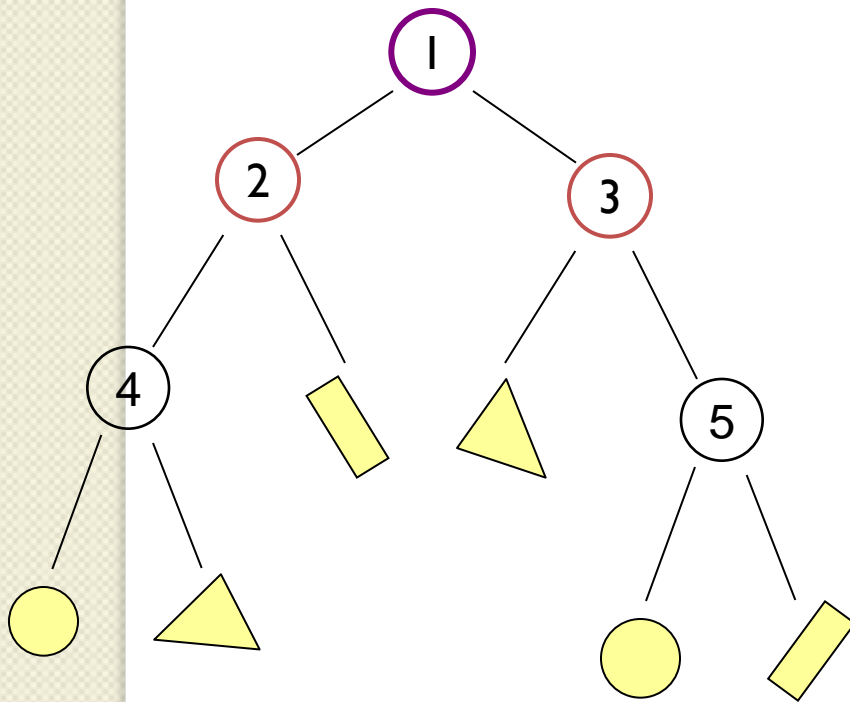
Octree

- Very useful in computer graphics, used for
 - Intersections
 - Collisions
 - Color quantization
 - Surface reconstruction (meshing)
 - ...



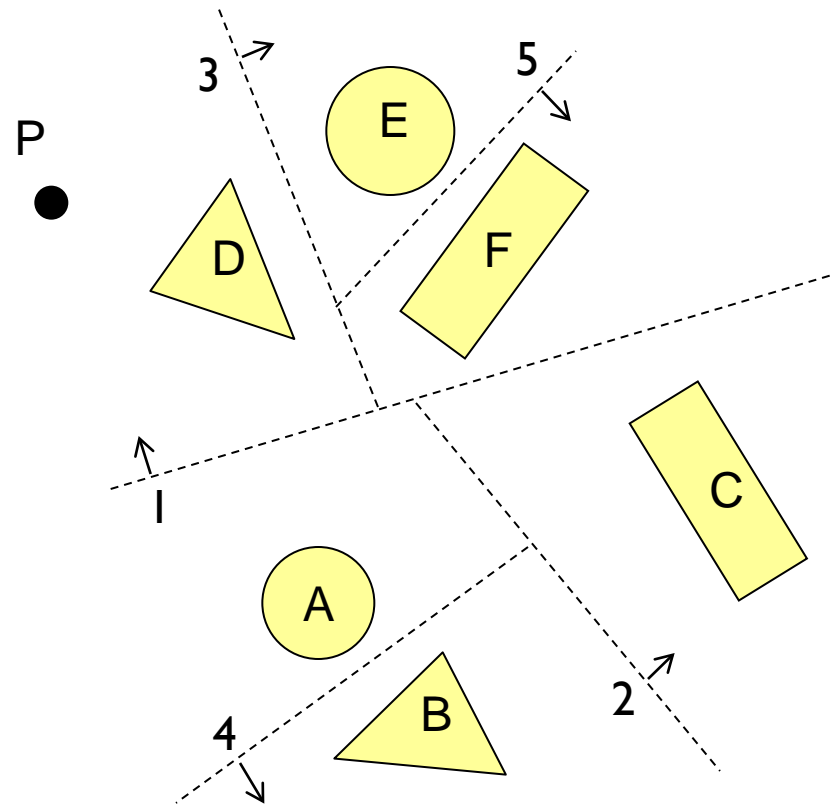
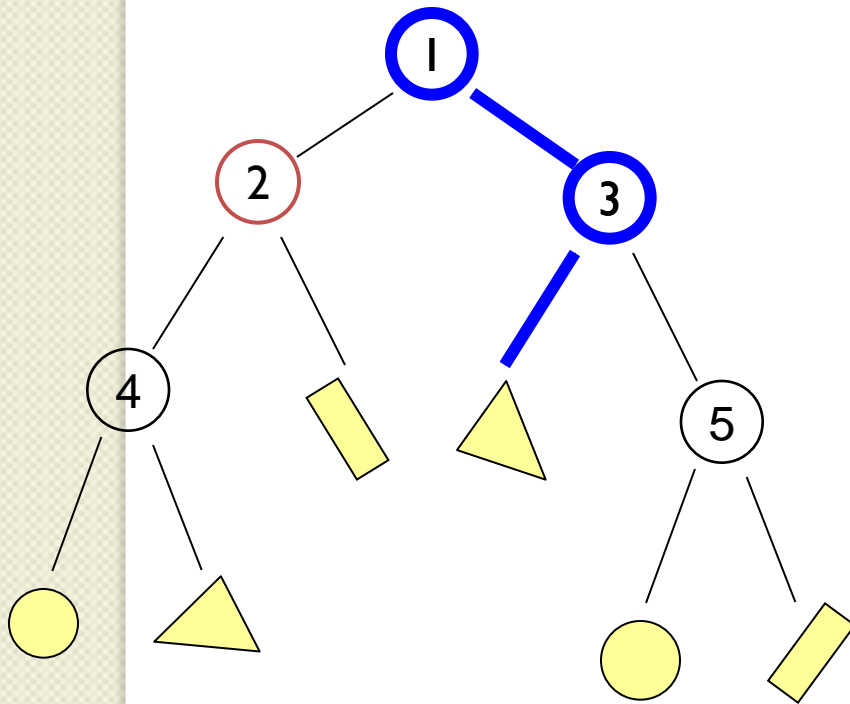
Binary Space Partition (BSP) Tree

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



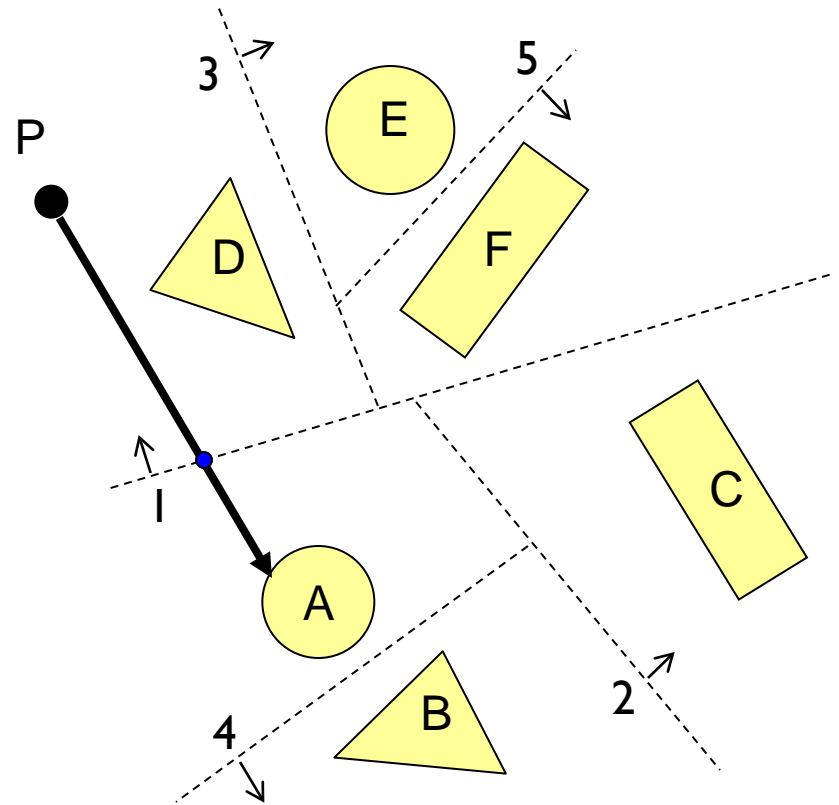
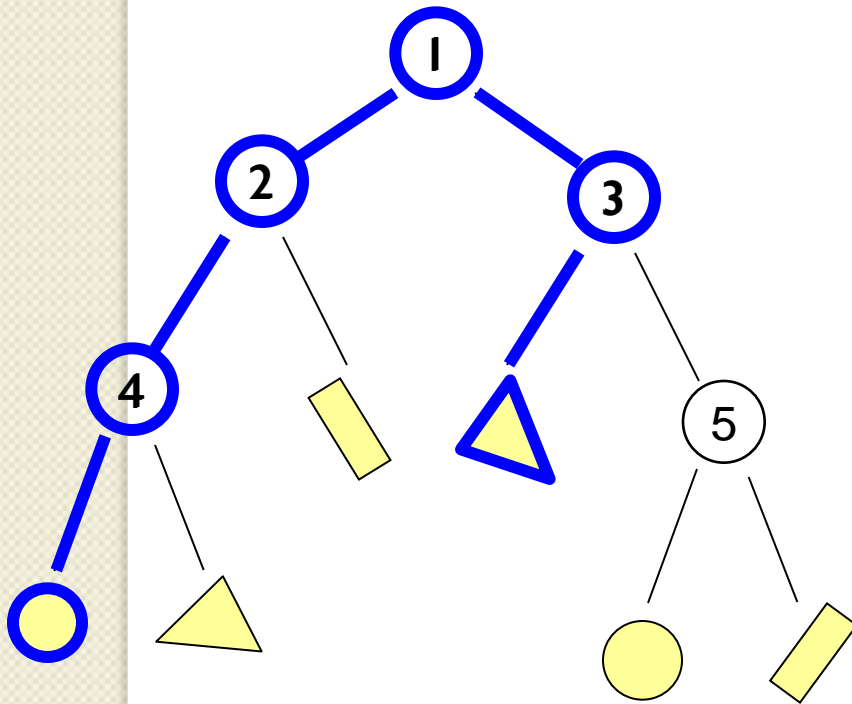
Binary Space Partition (BSP) Tree

- Simple recursive algorithms
 - Example: point finding



Binary Space Partition (BSP) Tree

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



Binary Space Partition (BSP) Tree

```
RayTreeIntersect(Ray ray, Node node, double min, double max)
{
    if (Node is a leaf)
        return intersection of closest primitive in cell, or NULL if none
    else
        dist = distance of the ray point to split plane of node
        near_child = child of node that contains the origin of Ray
        far_child = other child of node
        if the interval to look is on near side
            return RayTreeIntersect(ray, near_child, min, max)
        else if the interval to look is on far side
            return RayTreeIntersect(ray, far_child, min, max)
        else if the interval to look is on both side
            if (RayTreeIntersect(ray, near_child, min, dist)) return ...;
            else return RayTreeIntersect(ray, far_child, dist, max)
}
```

-

Summary

- Writing a simple ray casting renderer is easy
 - Generate rays
 - Intersection tests
 - Lighting calculations
- What's next?
 - **Illumination**

