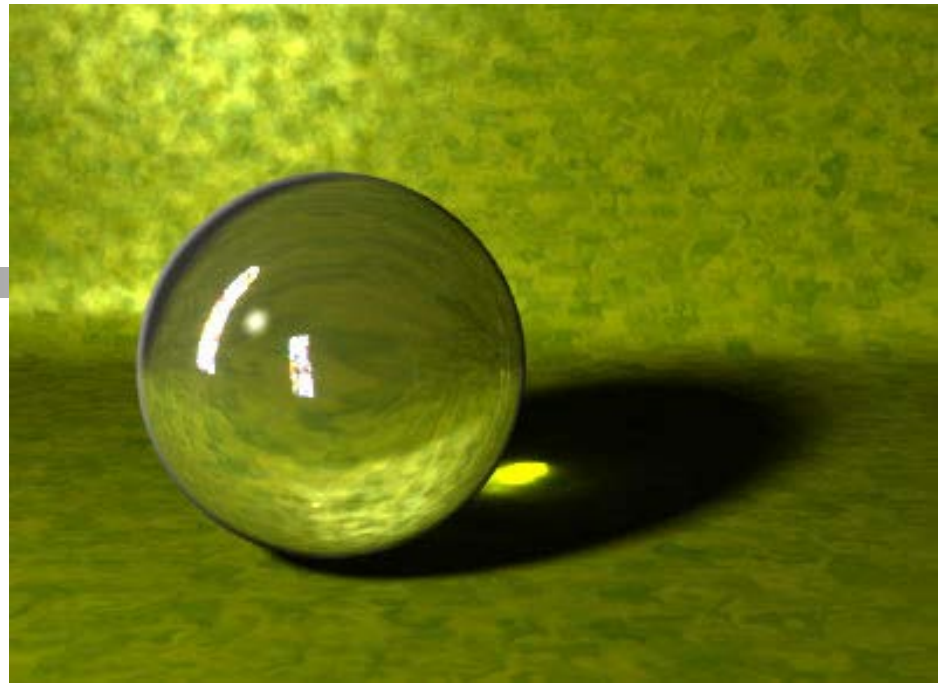
Ray Tracing



Today

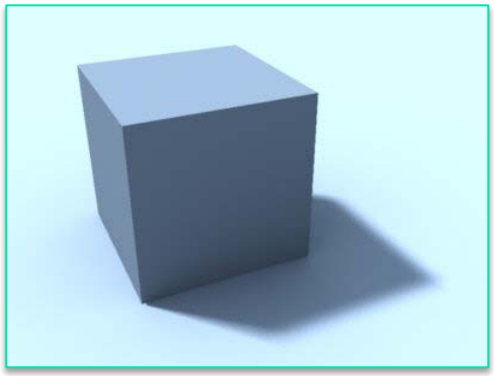
- Part 1: Ray casting
- Part 2: Lighting
- Part 3: Recursive Ray tracing
- Part 4: Acceleration Techniques

Example 1

Example 2

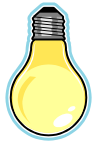
What is 3D Rendering?

- Construct an image from a 3D model

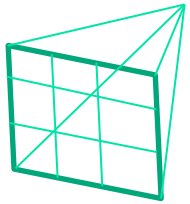


Rendering

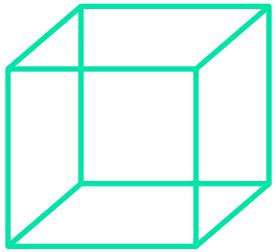
Light



Camera



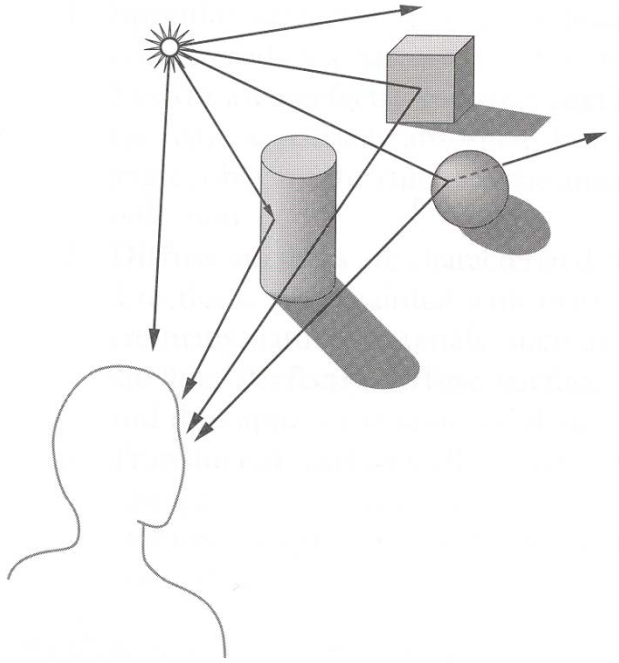
3D Model



View Plane

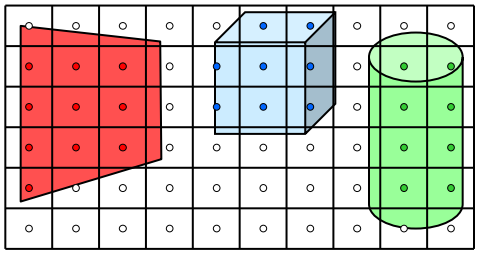
In the Physical World How Do We See?

- Option 1: shoot rays from lights and see how they move in space until they reach the eye.
- Problem?
- When tracing rays from light sources to the eye many are wasted since they never reach the eye
- Solution: reverse the order – shoot rays from the eye until they hit the light?



Part I:
Ray Casting

- How to find the color of a pixel in an image?

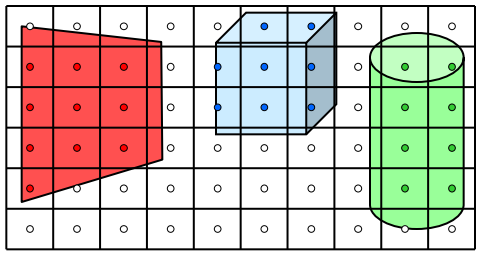


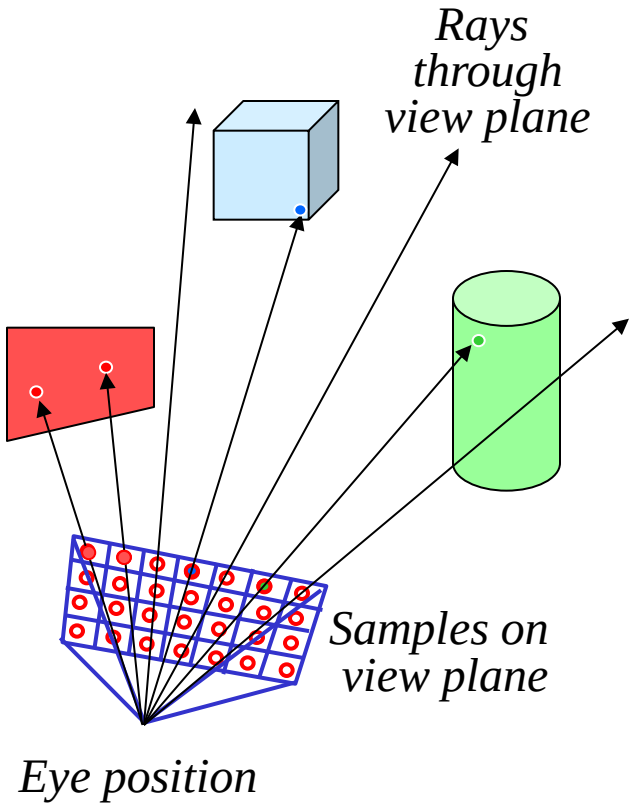
First Approximation: Ray Casting

1. Shoot Rays from center of projection through pixels
2. Find intersection with scene objects
3. Calculate an approximated color at intersection point

Geometry

Lighting





Ray Casting

```
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(scene, ray, hit);
        }
    }
    return image;
}
```

Stages Per Image Pixel

1. Shoot Rays from center of projection through pixels:

```
Ray ray = ConstructRayThroughPixel(camera, i, j);
```

2. Find intersection with scene objects

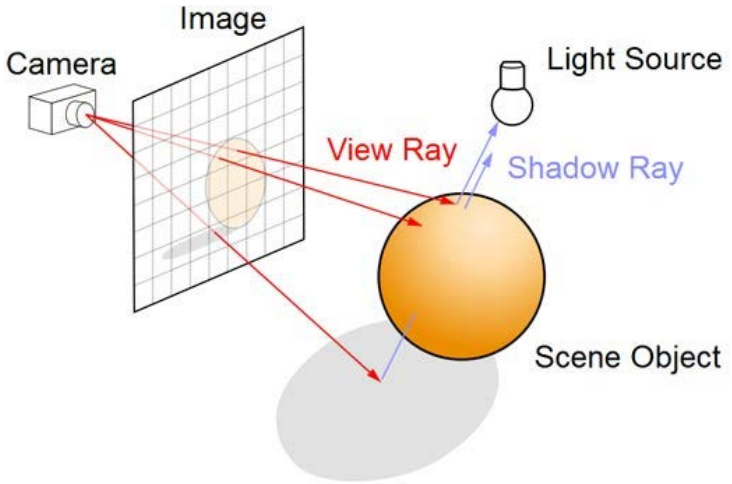
```
Intersection hit = FindIntersection(ray, scene);
```

3. Calculate an approximated color at intersection point

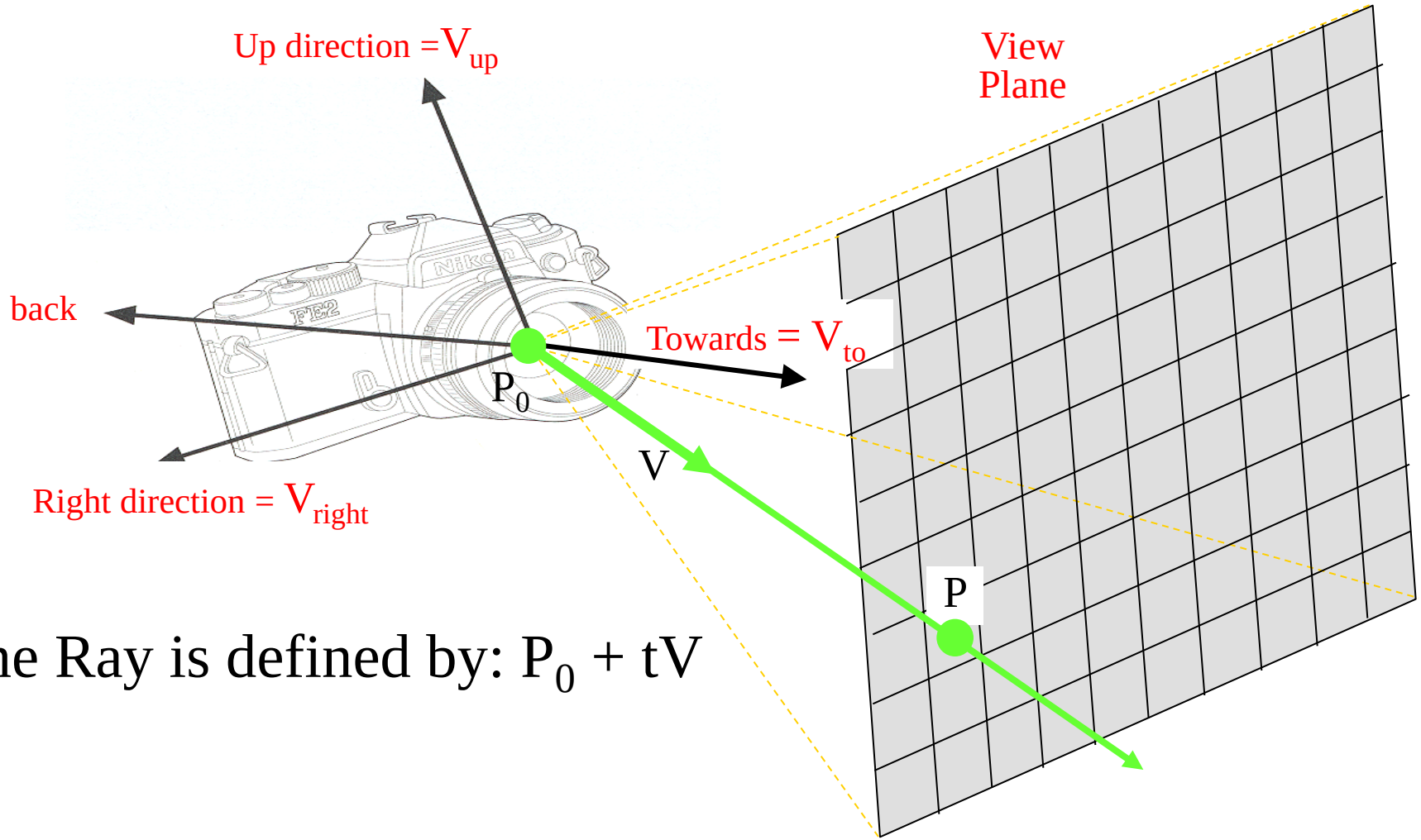
```
image[i][j] = GetColor(scene, ray, hit);
```

Part I: Geometry

Ray Intersections

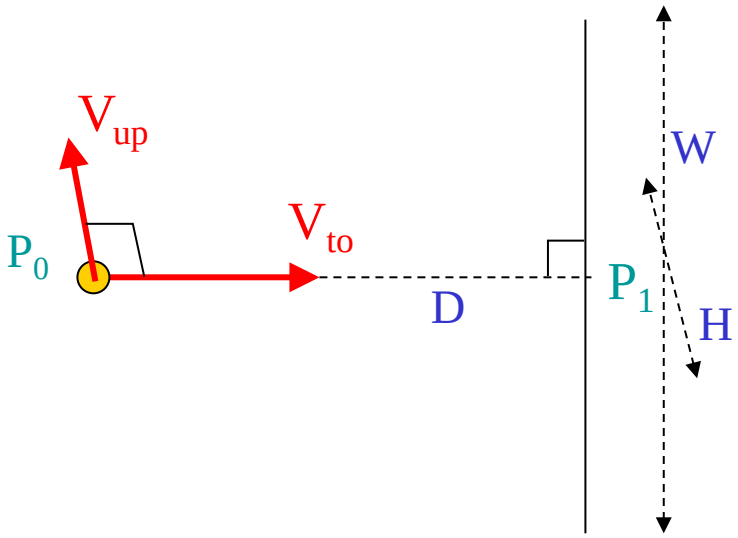


Constructing Ray Through a Pixel



The Ray is defined by: $P_0 + tV$

Example



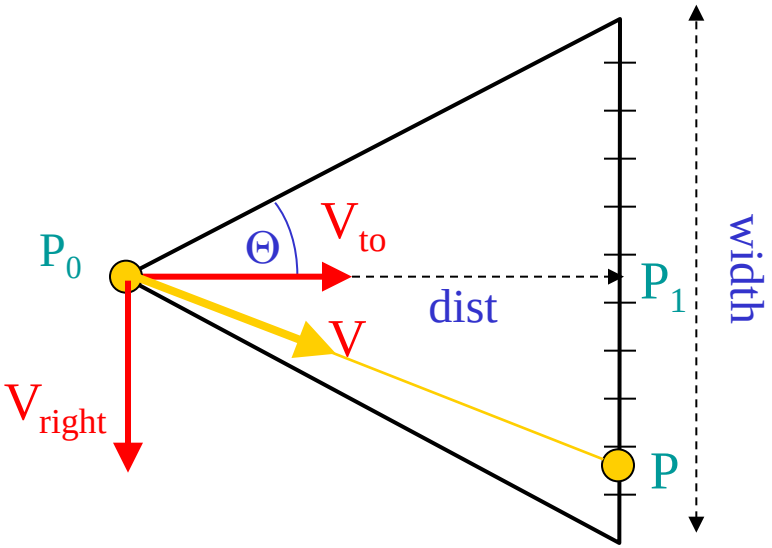
1D Example – shoot through the (i,0) pixel

$V_{right} = V_{to} \times V_{up}$

Θ = frustum half-angle

dist = distance to view plane

width = 2*dist*tan(Θ)



Ray is defined by:

P(t) = P0 + tV

To define the ray we must find V

To find V we find P = pixel intersection point

P1 = P0 + dist*Vto

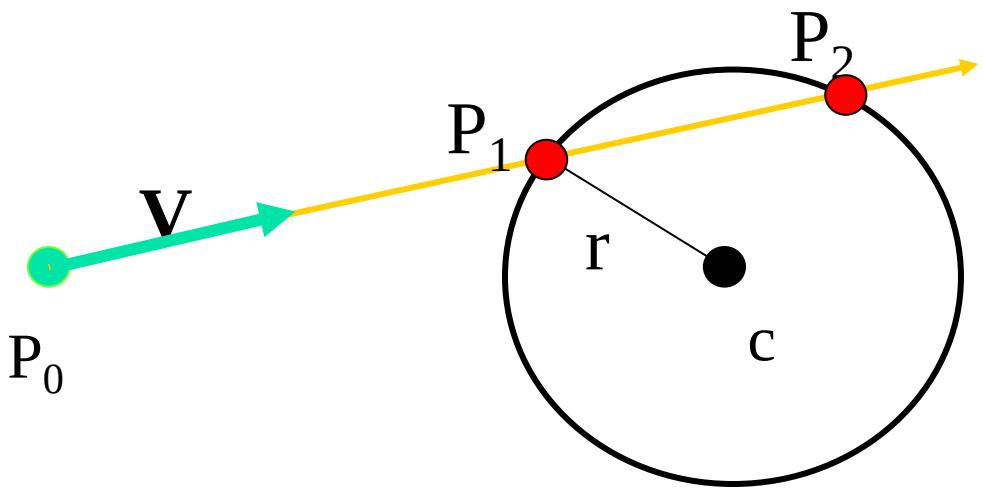
P = P1 ± i * Vright for i ∈(0,width/2)

Now we can find V = (P - P0) / ||P - P0 ||

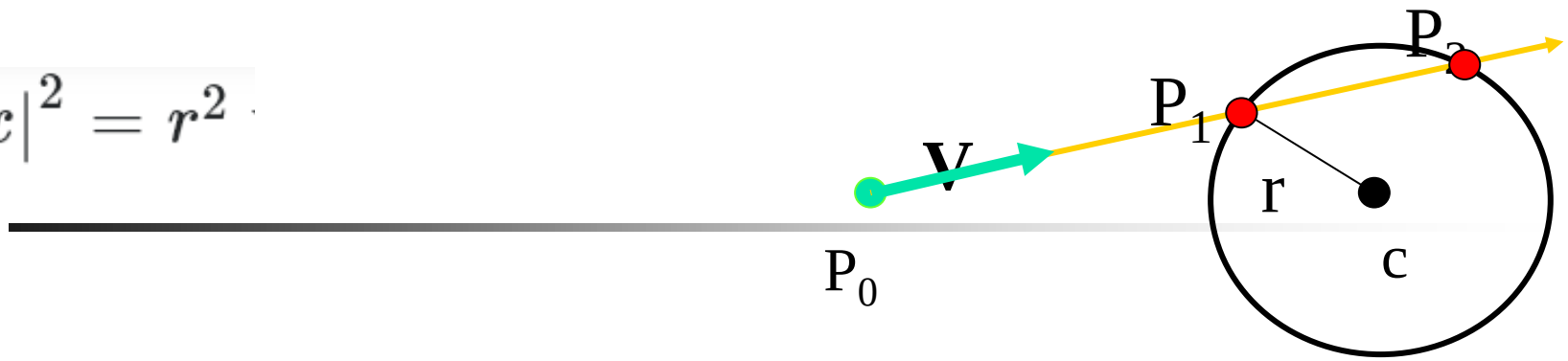
Ray-Scene Intersection

1. Finding the intersection point with a geometric primitive
2. Find closest intersection point in a group of objects (scene)
3. Acceleration techniques
    1. Bounding volume hierarchies
    2. Spatial partitions

Sphere Intersection



$$|c - x|^2 = r^2$$

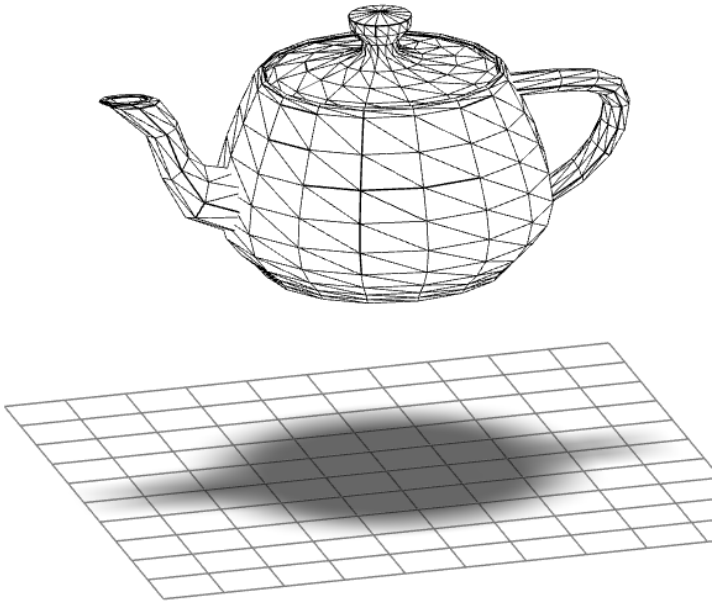


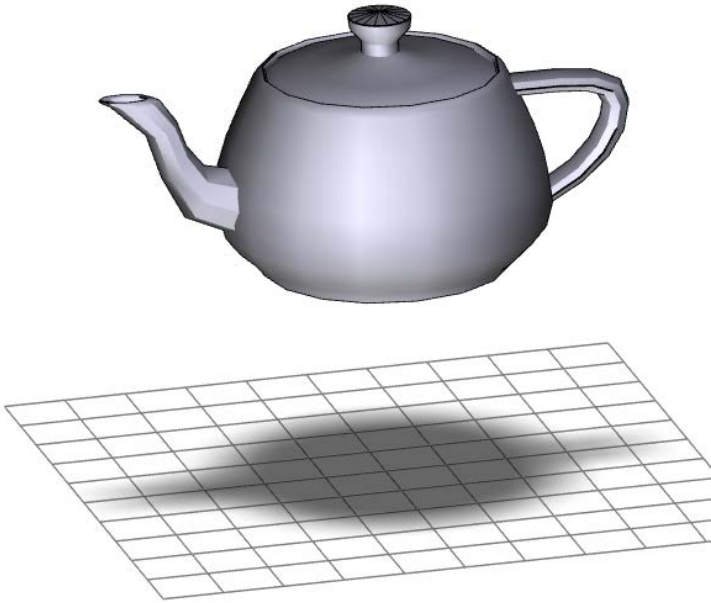
$$x(t) = a + tb$$ (P(t) = P0 + tV)

$$(tb + (a - c)) \cdot (tb + (a - c)) = r^2$$

$$t = \frac{-2(a - c) \cdot b \pm \sqrt{[2(a - c) \cdot b]^2 - 4|b|^2(|a - c|^2 - r^2)}}{2|b|^2}$$

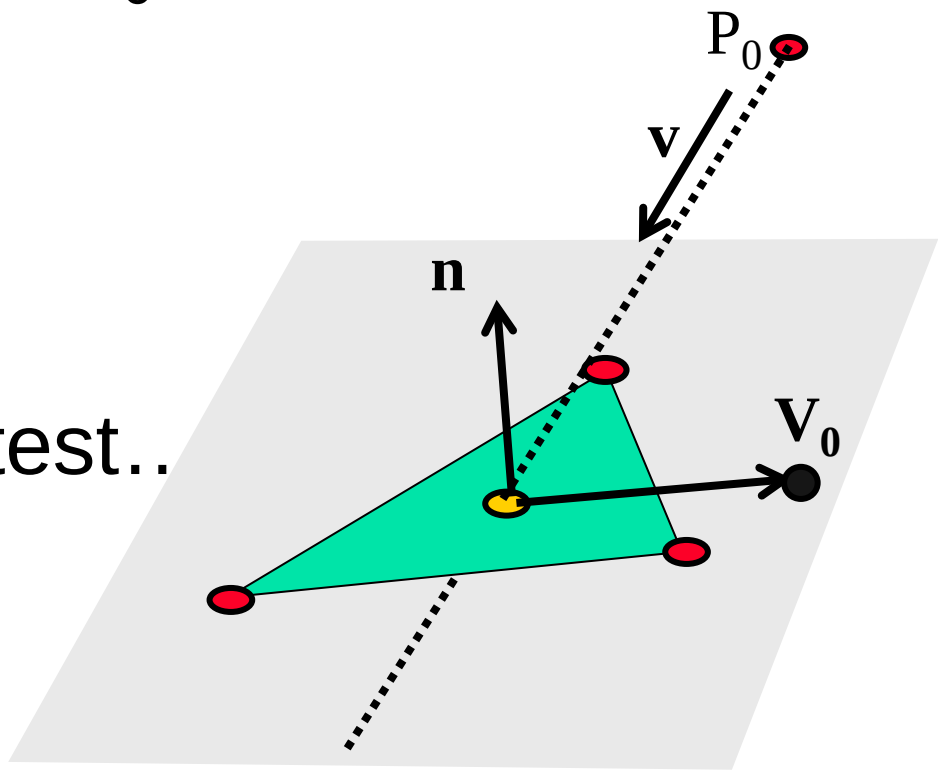
Meshes?





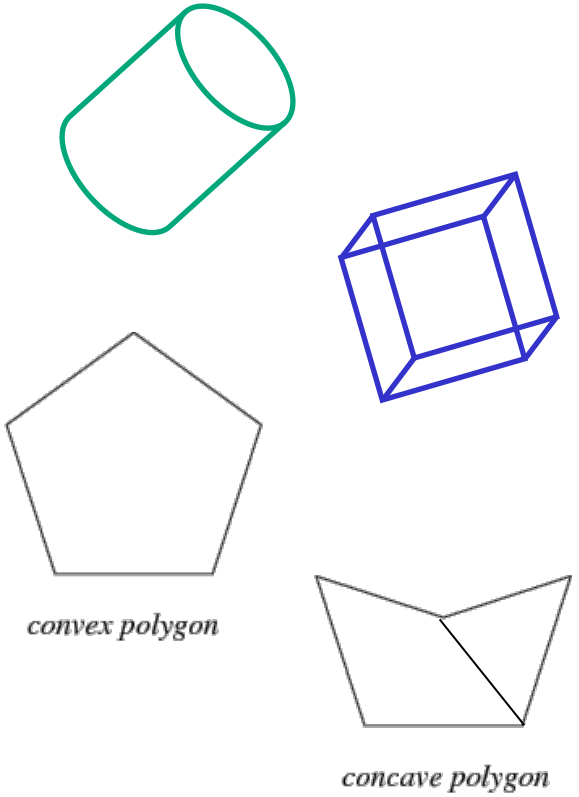
Triangle intersection

- Plane through V_0 with normal n
- Parametric line: $P(t) = P_0+vt$
- $(V_0 - P(t))n=0$
- $(V_0 - P_0-vt)n=0$
- $t = (V_0 - P_0)n/vn$
- Point inside triangle test…



Other 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

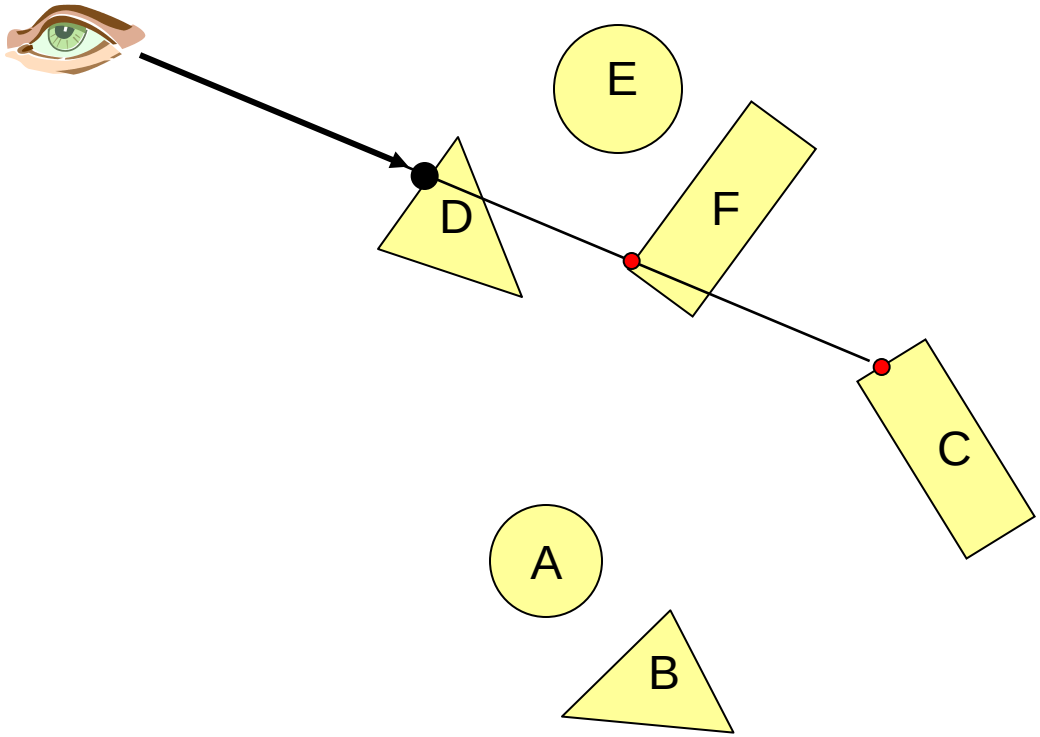


Algorithms for 3D object intersection:

http://www.realtimerendering.com/intersections.html

Intersecting a Group of Objects

- Must find the nearest intersection



Intersecting a Scene

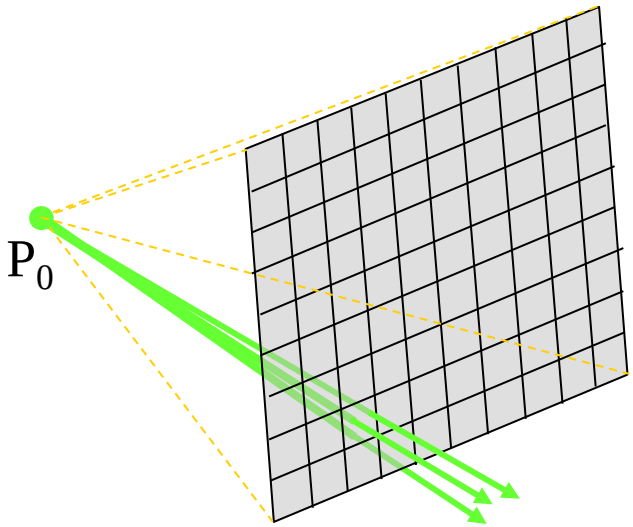
A Brute Force Approach:

```
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)
}
```

Alisaing?

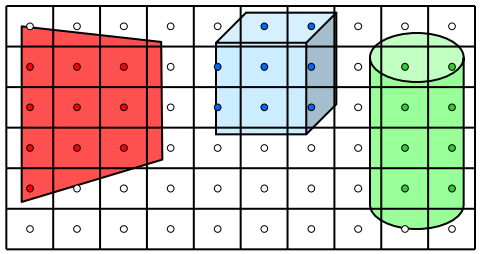
Aliasing?

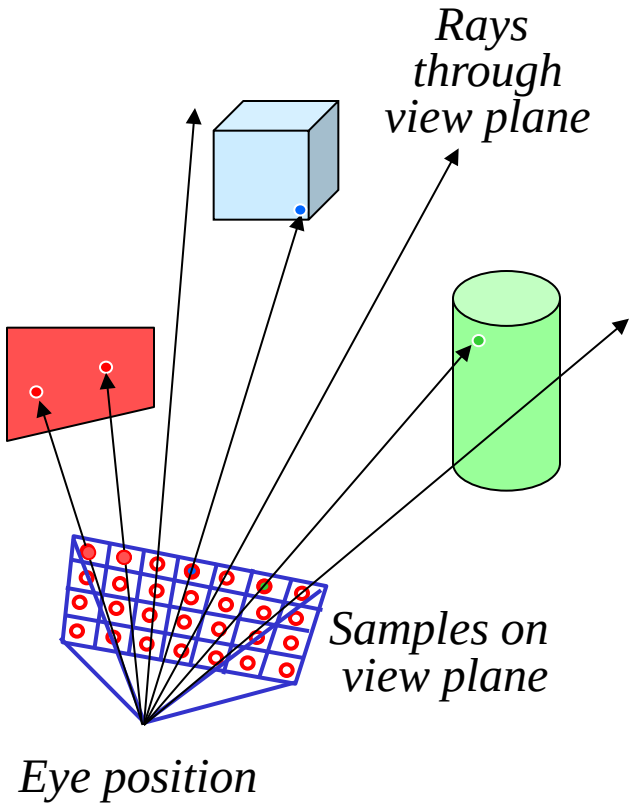
- Yet again we are sampling a continuous world and get sampling artifacts.
- Better results: we shoot more than one ray trough each pixel and average their color!
- (again: smoothing vs. aliasing)



Ray Casting

1. Shoot Rays from center of projection through pixels
2. Find intersection with scene objects
3. Calculate an approximated color at intersection point



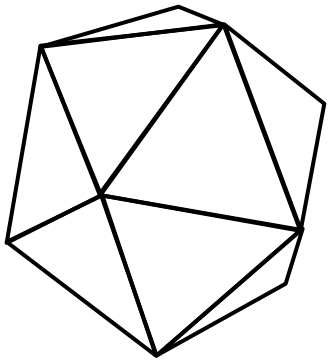


Part II: Lighting

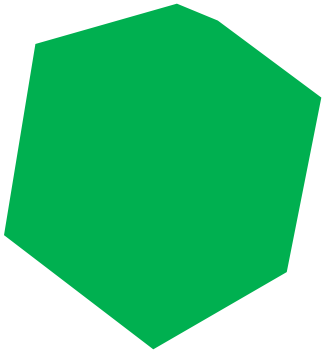
Determining Pixel Color

From Model to Picture

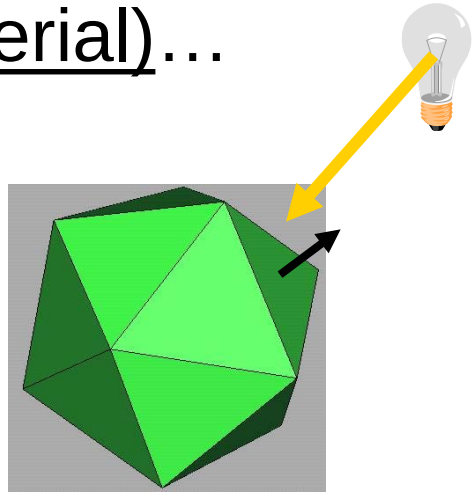
- Using just color creates a flat image
- Real color is created by the interaction of light with surface (normal and material)…



Wireframe Model



Without Illumination

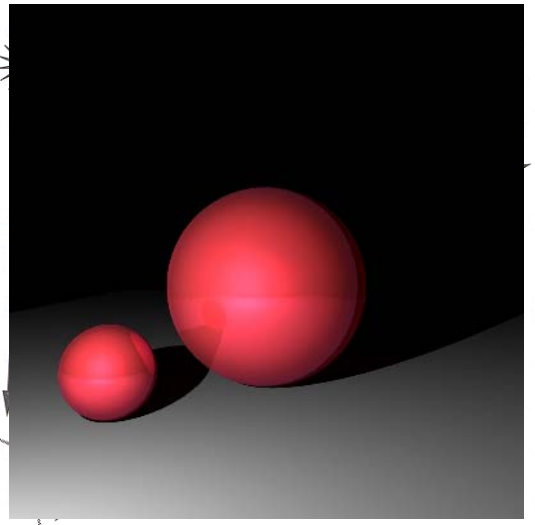


With Illumination

What we see is not just color!

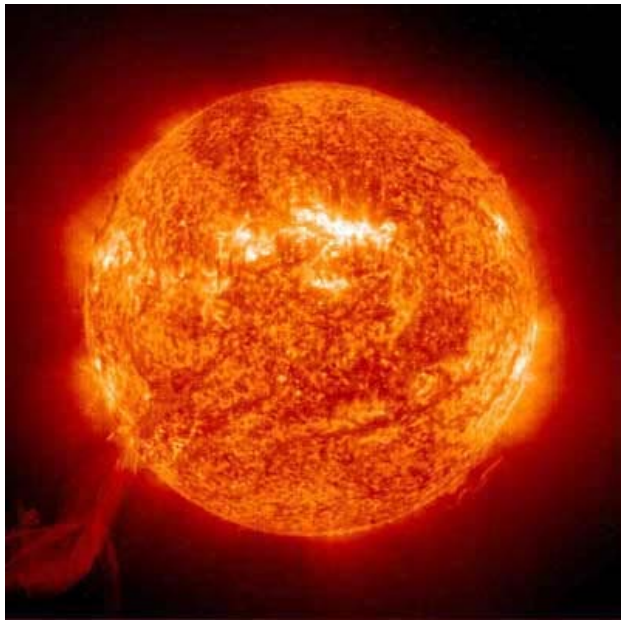
```
image[i][j] = GetColor(scene, ray, hit);
```

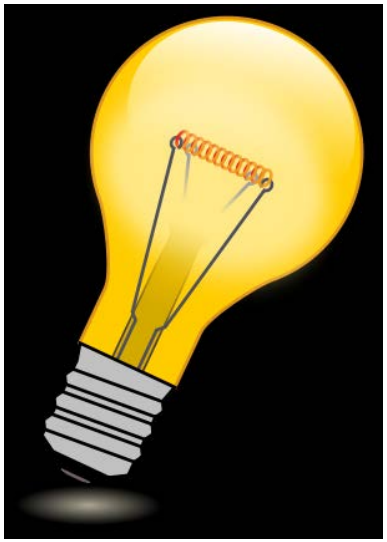
- How do we compute **radiance** for a sample ray?



- Depends on:
  - Light properties
  - Surface material properties
  - Geometry (interrelations)
  - Other factors: shadows, atmosphere, smoke, fog

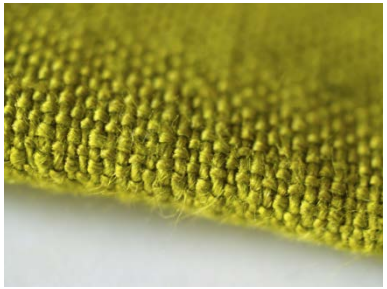
Different Types of Lights

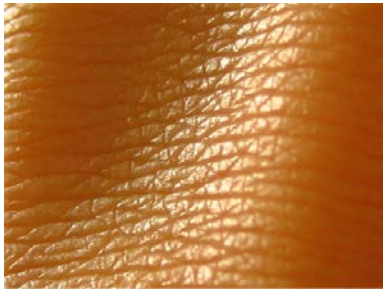
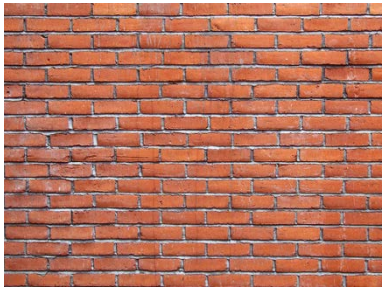


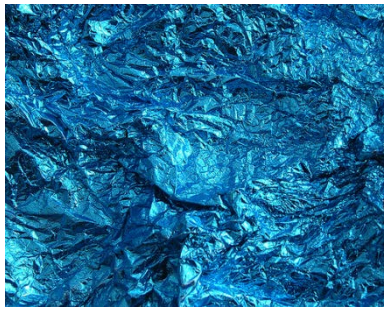


Different Types of Materials





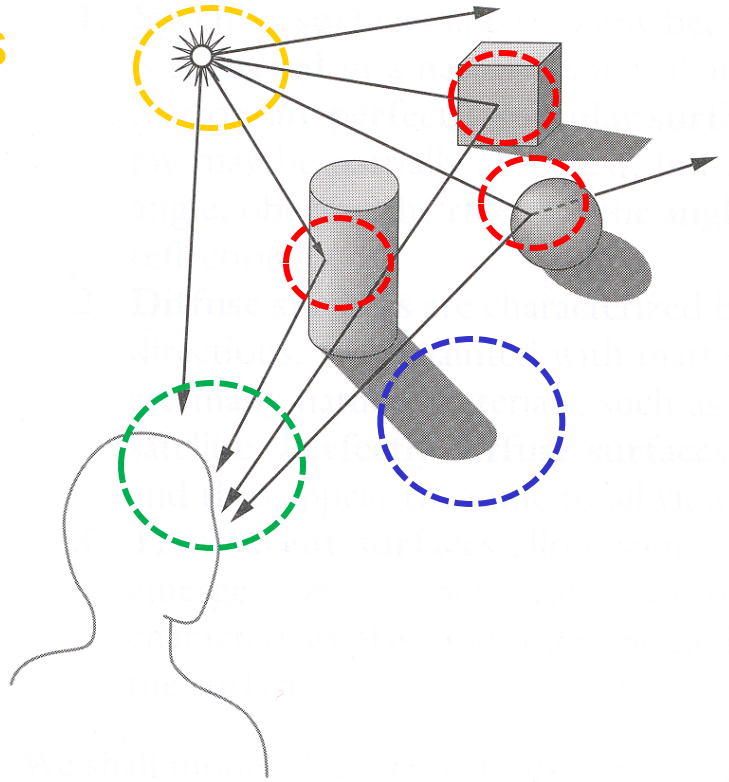




Other Factors

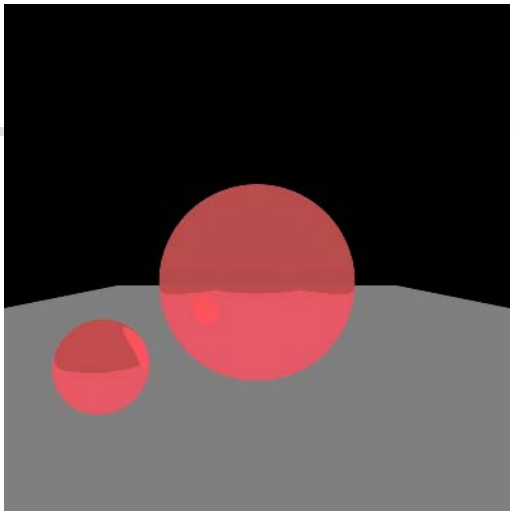
Goal

- Must derive computer models for:
  - Emission at light sources
  - Scattering at surfaces
  - Reception at the camera
  - Global Effects
- Desirable features:
  - Accurate
  - Concise
  - Efficient to compute

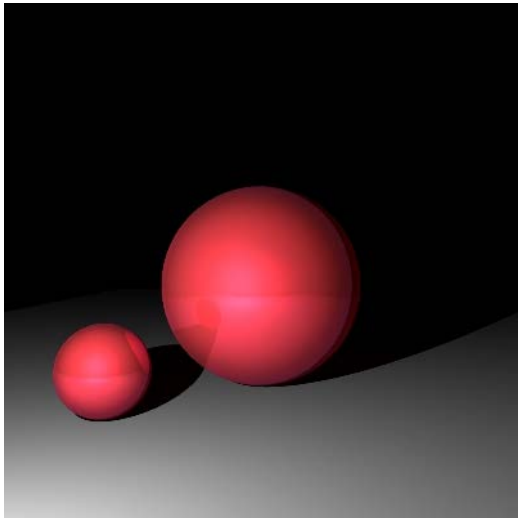


Direct Illumination

- Two issues:
  - Emission at light sources
  - Scattering at surfaces

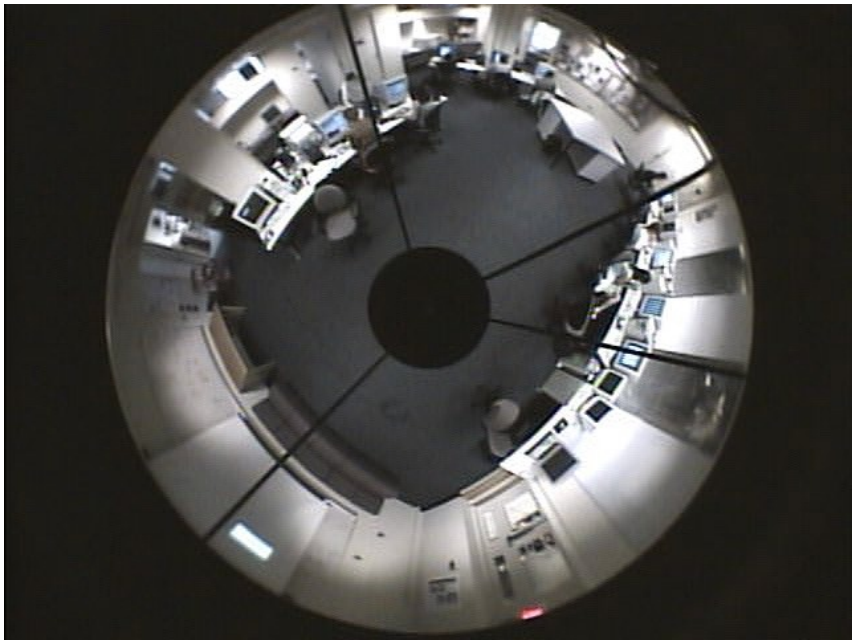


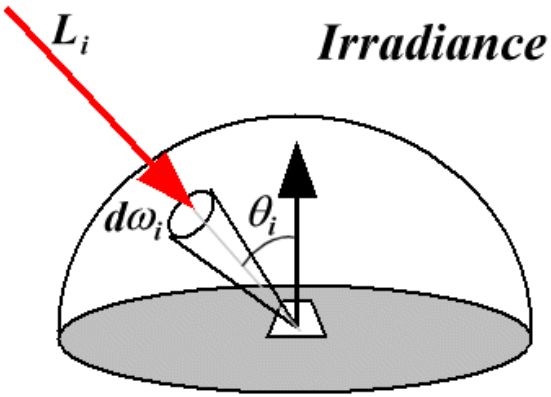
- Later: global illumination
  - Shadows, reflections refractions



Irradiance

- The irradiance is a two dimensional function describing the incoming light at a given point.
- In radiometry, irradiance is the radiant flux (power) received by a surface per unit area

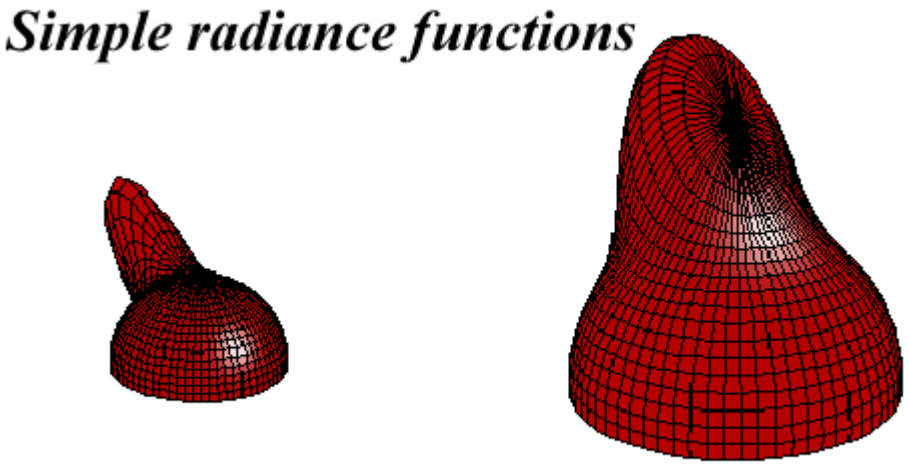




$$E_i = \int_{\Omega_i} L_i \cos\theta_i d\omega_i$$

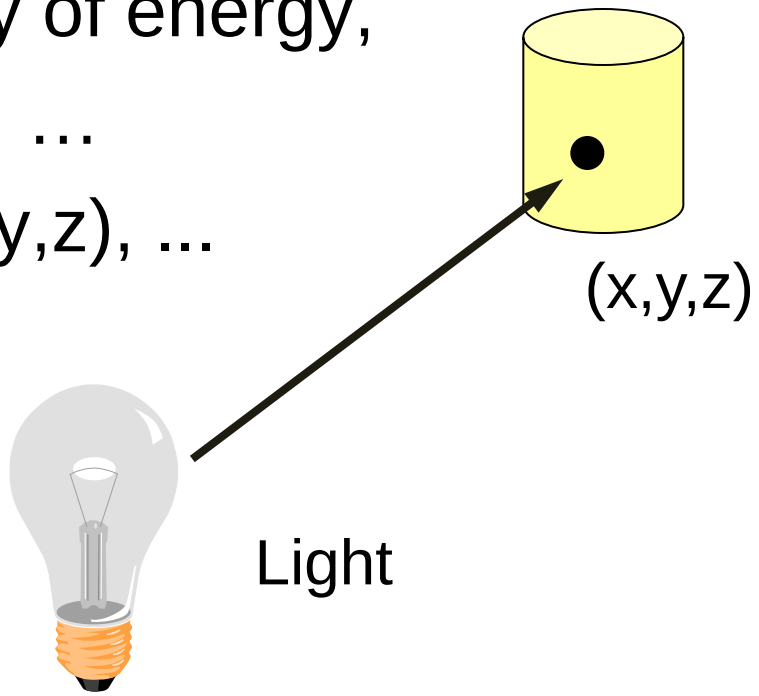
Radiance

- The radiance is a two dimensional function representing the light reflected from a surface at a given point.



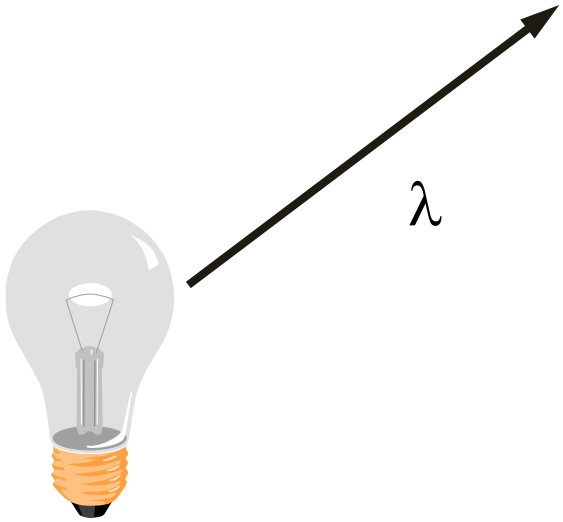
Modeling Light Sources

- $I_L(x,y,z,\theta,\phi,\lambda)$ ...
  - describes the intensity of energy,
  - leaving a light source, …
  - arriving at location (x,y,z), ...
  - from direction (θ,ϕ), ...
  - with wavelength λ



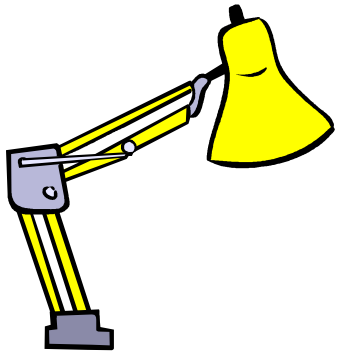
Empirical Models

- Ideally measure irradiant energy for “all” situations
  - Too much storage
  - Difficult in practice



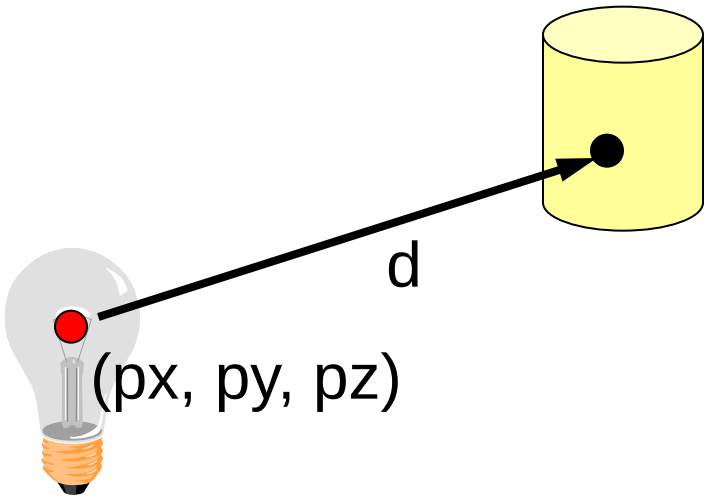
Light Source Models (OpenGL)

- Simple mathematical models:
  - Point light
  - Directional light
  - Spot light



Point Light Source

- Models omni-directional point source (e.g., bulb)
  - Intensity (I_0),
  - Position (px, py, pz),
  - Factors (k_c, k_l, k_q) for attenuation with distance (d)

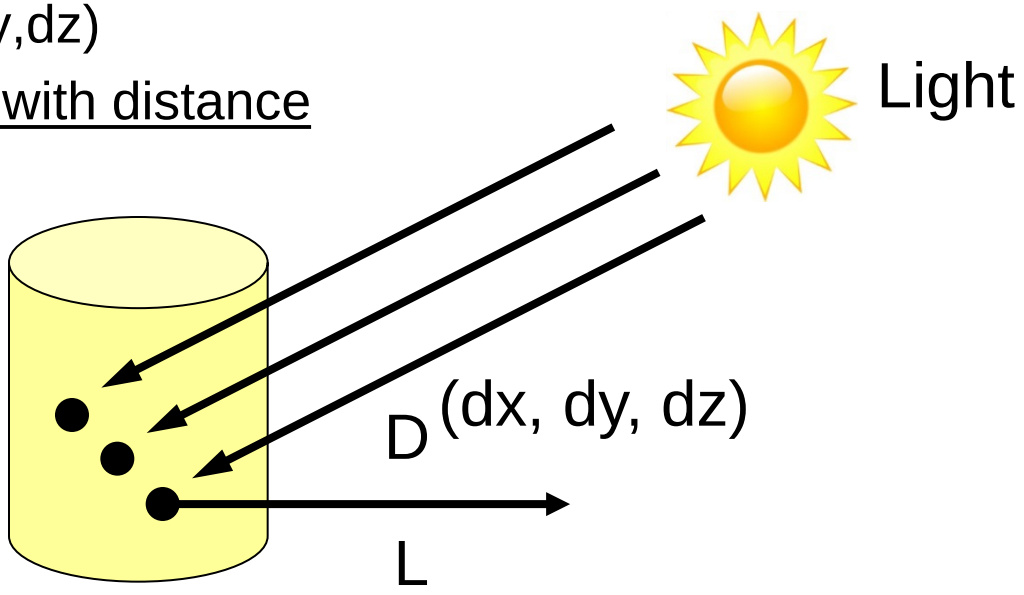


$$I_L = \frac{I_0}{k_c + k_l d + k_q d^2}$$

Directional Light Source

- Models point light source at infinity (e.g., sun)
  - Intensity (I_0),
  - Direction (dx,dy,dz)
  - No attenuation with distance

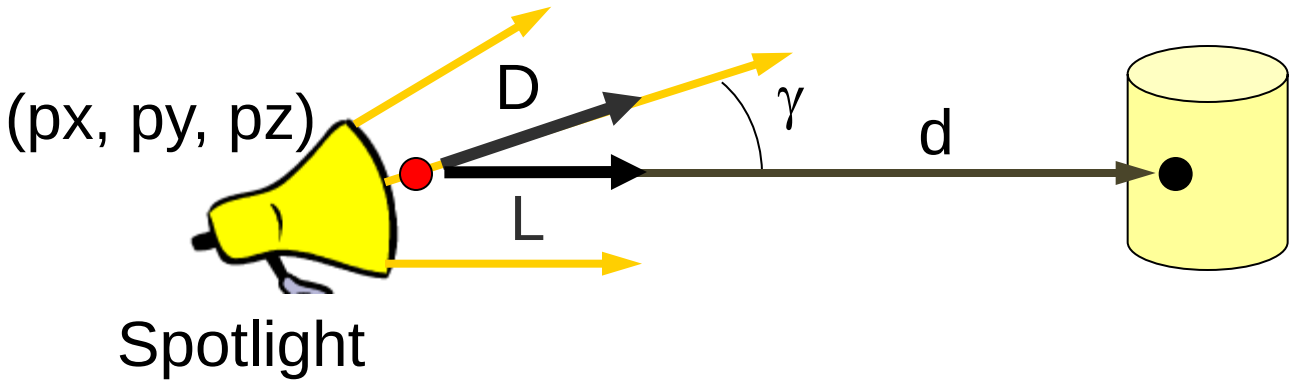
$$I_L = I_0(D \cdot L)$$



Spot Light Source

- Models point light source with direction (e.g., Luxo)
  - Intensity (I_0),
  - Position (px, py, pz),
  - Direction D=(dx, dy, dz)
  - Attenuation

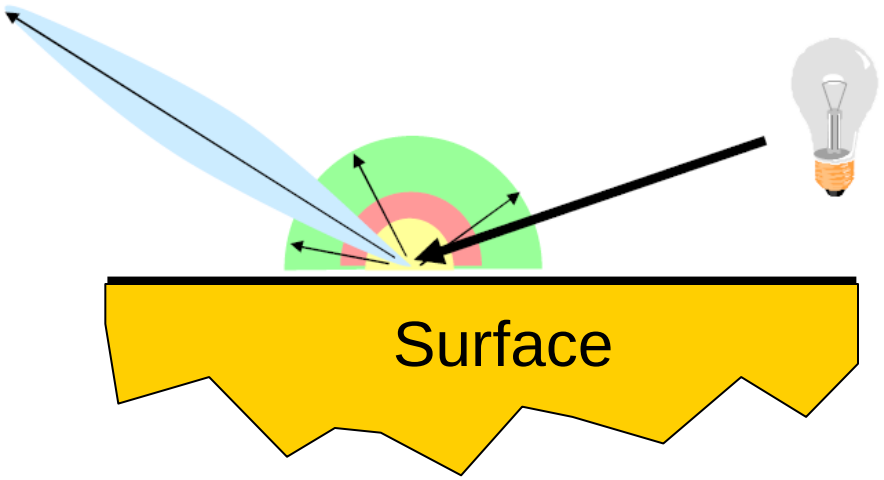
$$I_L = \frac{I_0 (D \cdot L)}{k_c + k_l d + k_q d^2}$$



Scattering at surfaces

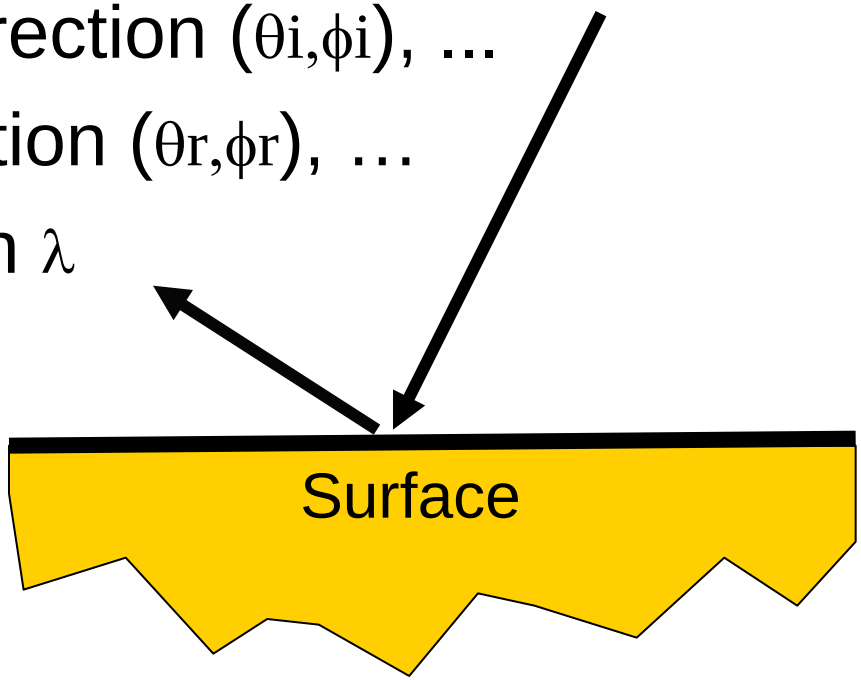
- Basic approach is energy conservation:

Light_out = Light_in - Light_absorbed



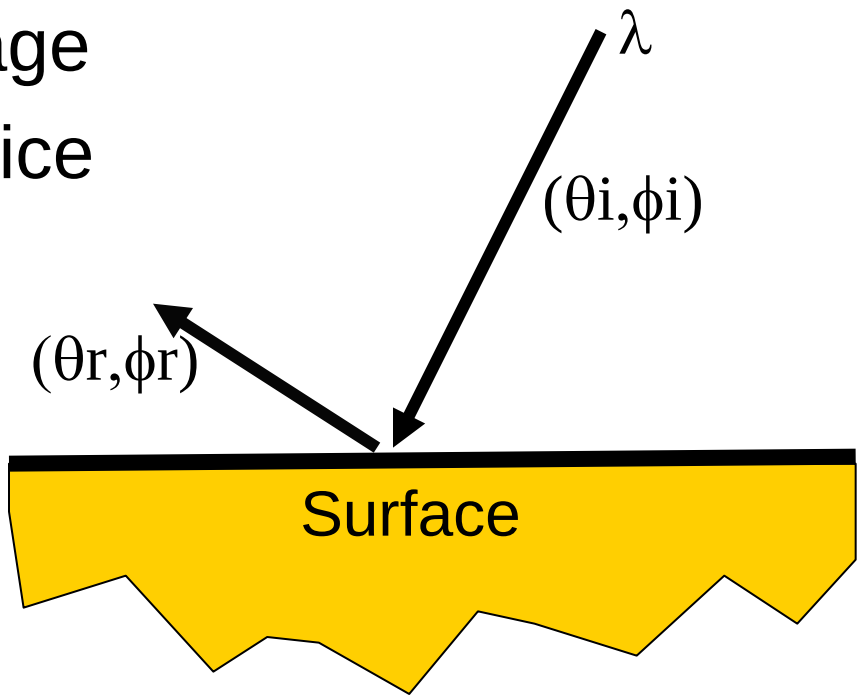
Modeling Surface Reflectance

- Rs($\theta i, \phi i, \theta r, \phi r, \lambda$) …
  - describes the amount of incident energy,
  - arriving from direction ($\theta i, \phi i$), ...
  - leaving in direction ($\theta r, \phi r$), …
  - with wavelength λ



Empirical Models

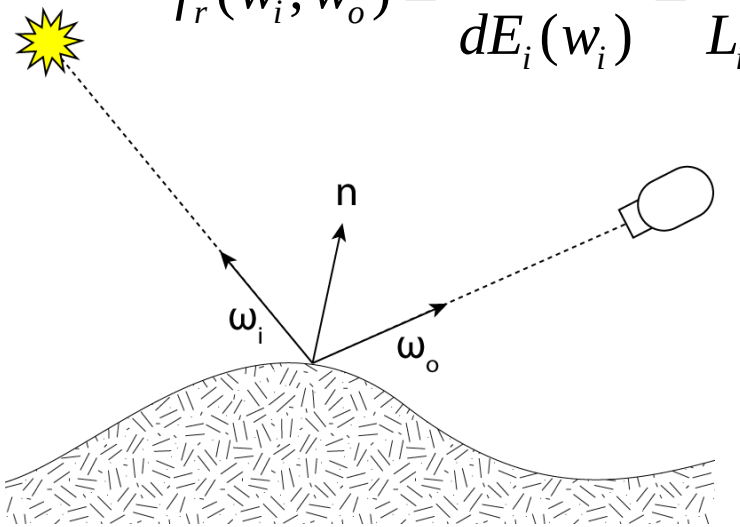
- Ideally measure radiant energy for “all” combinations of incident angles
  - Too much storage
  - Difficult in practice



Bidirectional Reflectance Distribution Function (BRDF)

- 4-dimensional function which defines light reflection at an opaque surface.
- BRDFs can be measured directly from real objects using calibrated cameras and lightsources
- The ratio of reflected radiance exiting along w_0 to the irradiance incident on the surface from direction w_i:

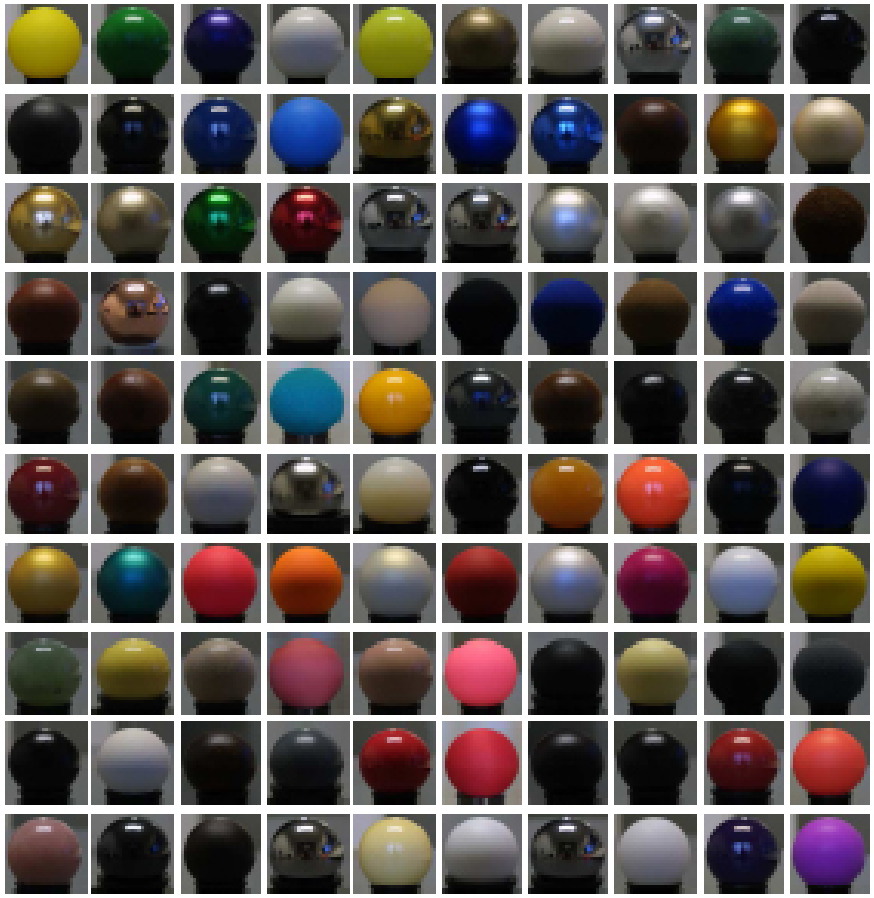
$$f_r(w_i, w_o) = \frac{dL_r(w_0)}{dE_i(w_i)} = \frac{dL_r(w_0)}{L_i(w_i)\cos(\theta_i)dw_i}$$

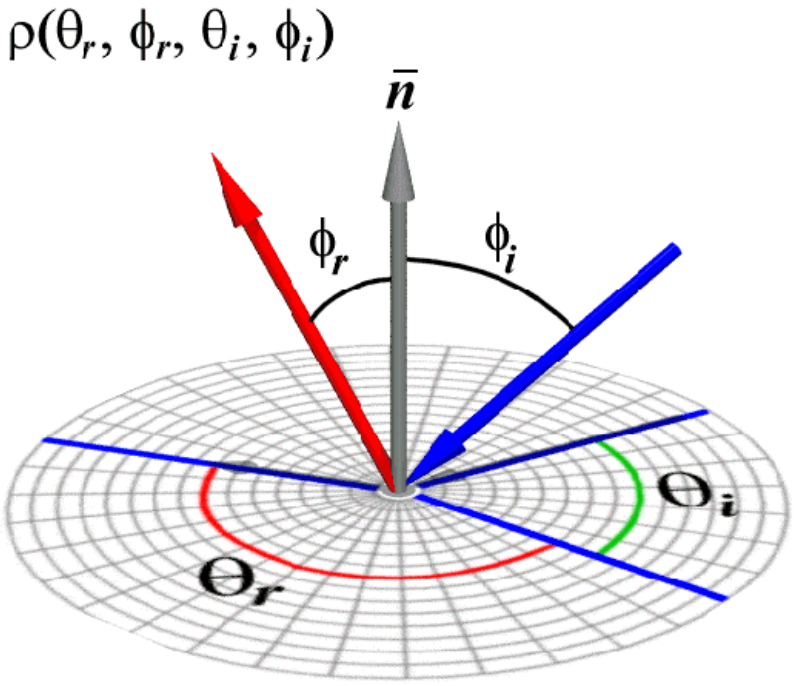


Gonioreflectometer

- A gonioreflectometer is a device for measuring a bidirectional reflectance distribution function (BRDF).

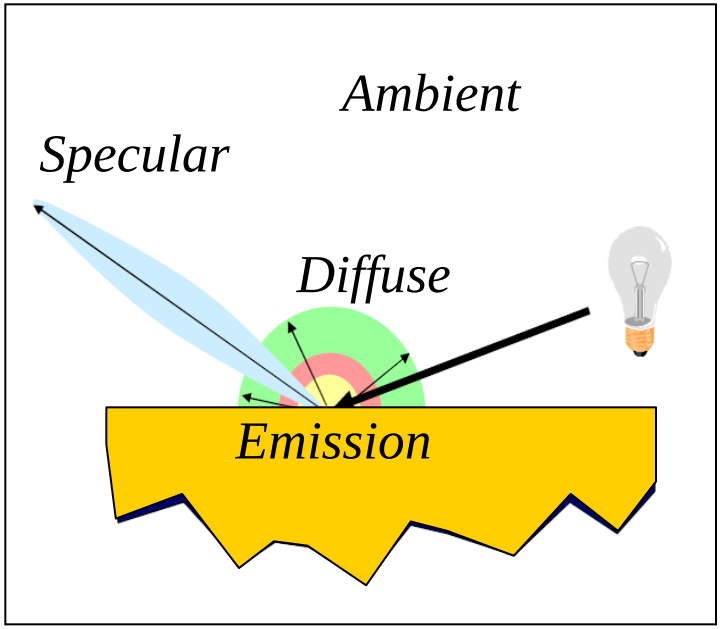
BRDF: 4 Dimensional Function





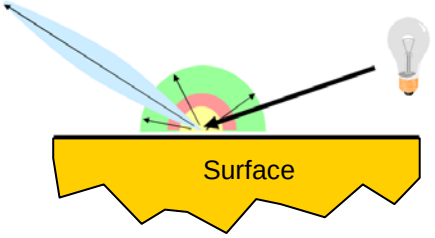
Phong Reflectance Model

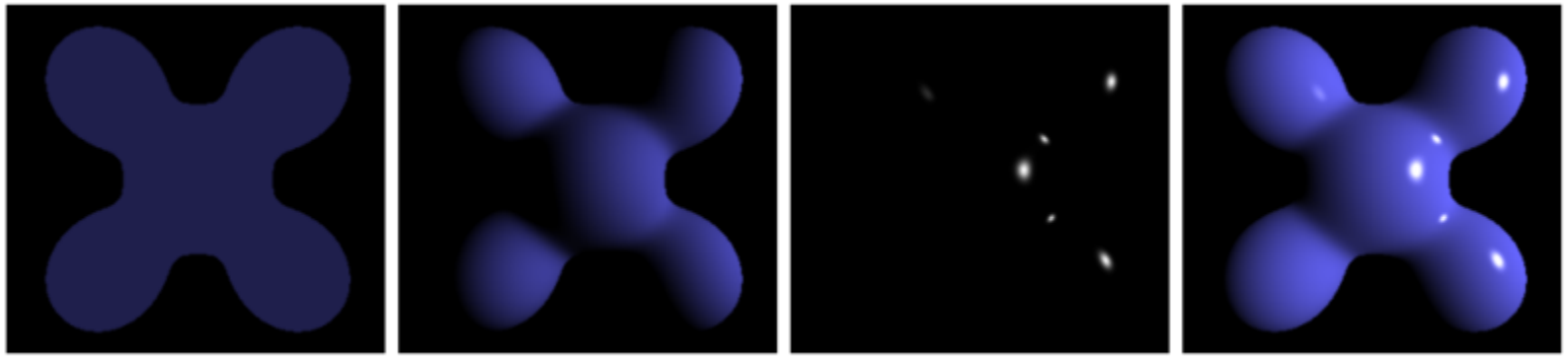
- Based on model proposed by Phong in his PhD dissertation 1973



Phong Reflectance Model

- Simpler analytic approximation model
  - Diffuse reflection +
  - Specular reflection +
  - Emission +
  - Ambient

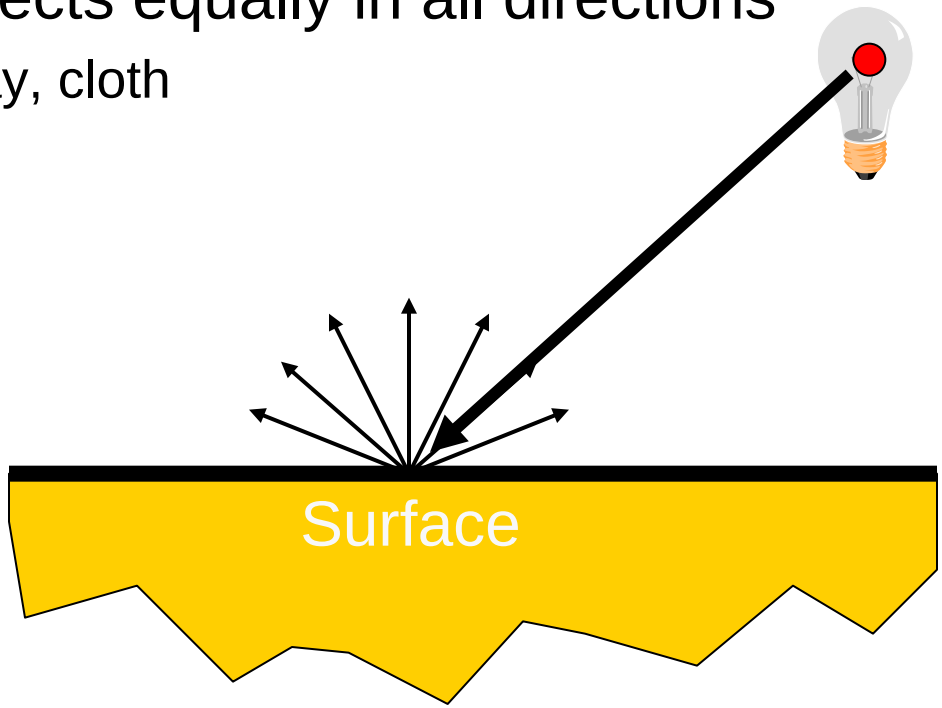




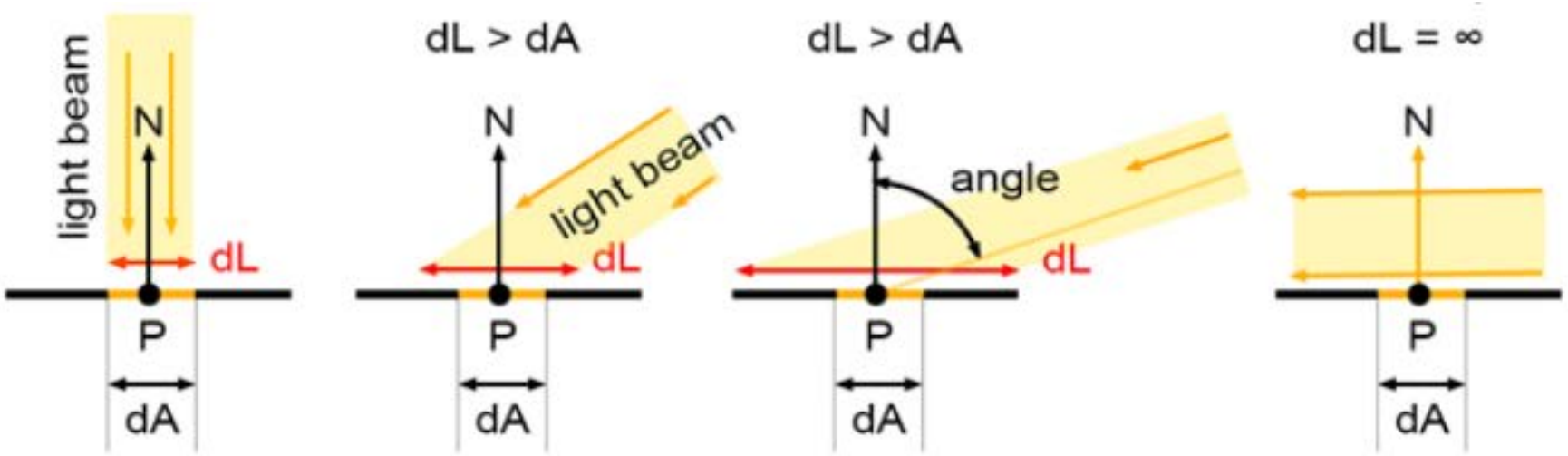
Diffuse Reflection

rough surfaces such as clothi
paper, leads to a type
of **reflection** known as
diffuse reflection

- Diffuse definition = spread-out, extend in all directions
- Assume surface reflects equally in all directions
  - Examples: chalk, clay, cloth



dA/dL = amount of discrete light energy in a point

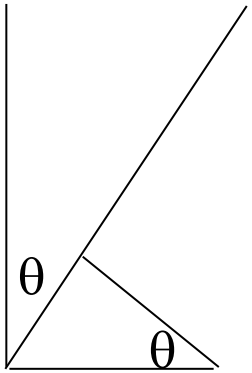


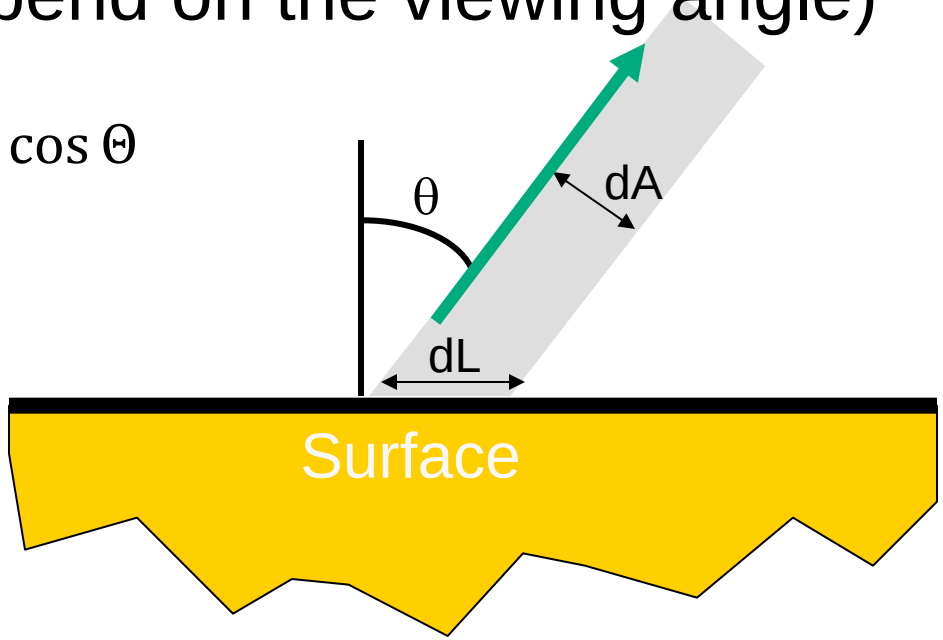
as the angle between the light beam and the normal at P increases, the cross section of the beam with the surface of the object becomes larger

Modeling Diffuse Reflectance

- How much light is reflected (spread)?
  - Depends on angle of incident light
  - (Does not depend on the viewing angle)

$$dA = dL \cos \Theta$$





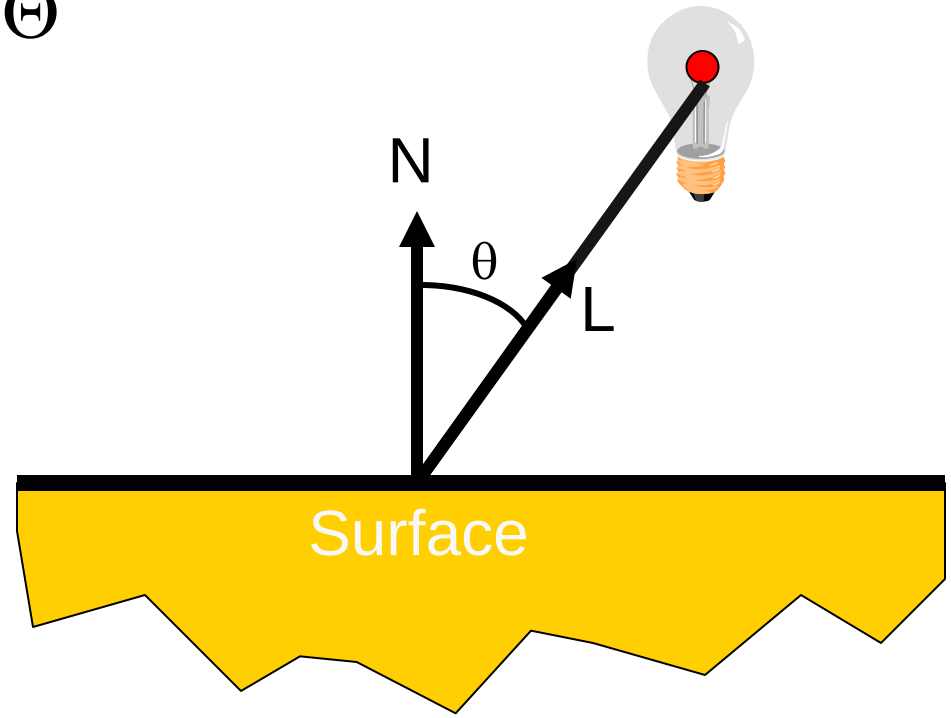
Lambertian Model (Lambert's Cosine Law)

cosine law (dot product)

$$N \cdot L = |N||L| \cos \Theta$$

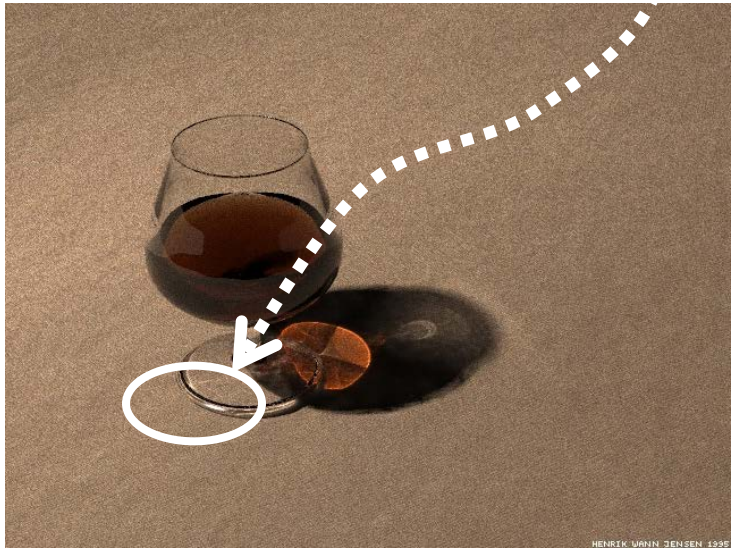
$$\hat{N} \cdot \hat{L} = \cos \Theta$$

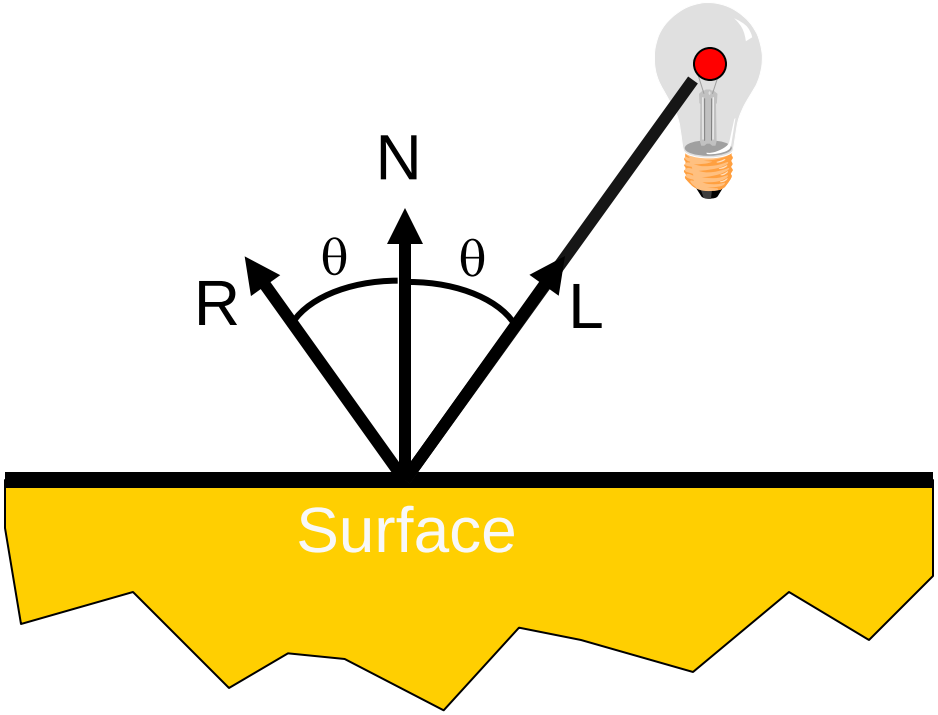
$$I_D = K_D(\hat{N} \cdot \hat{L}) I_L$$



Specular Reflection (mirror)

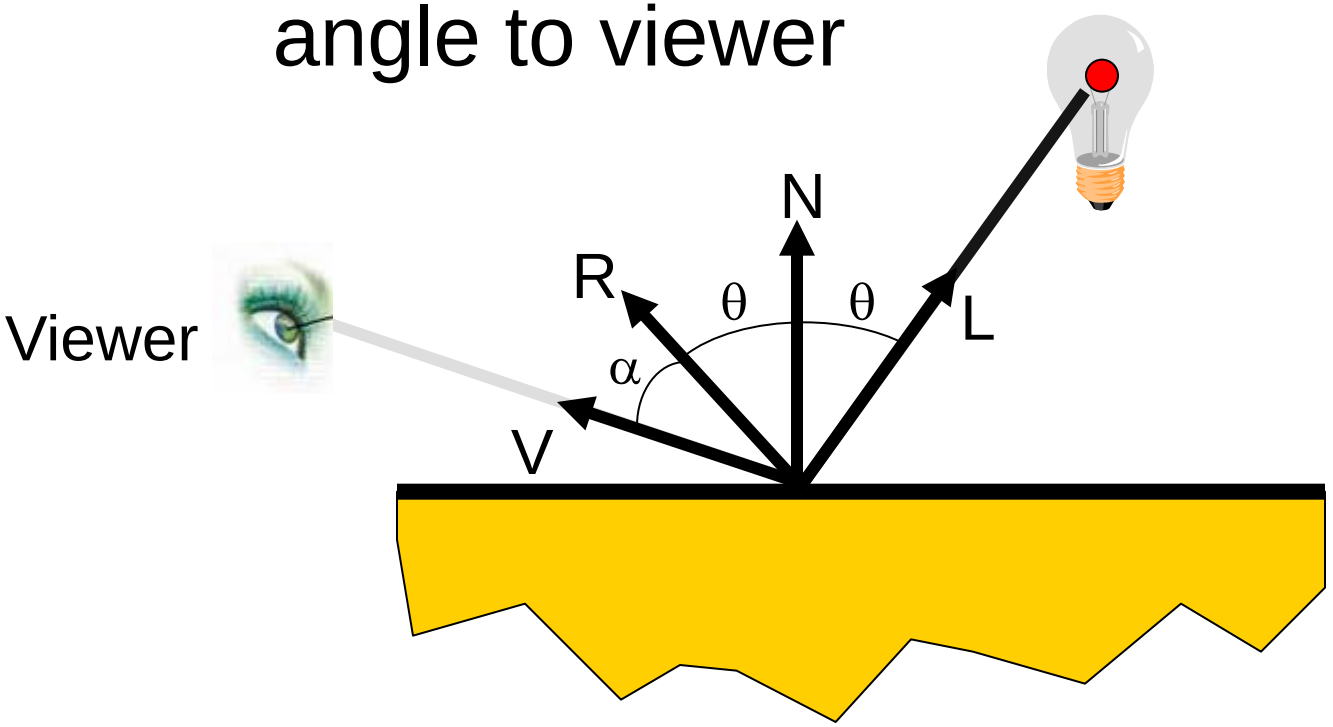
- Reflection is strongest near mirror angle
  - Examples: mirrors, metals, glass





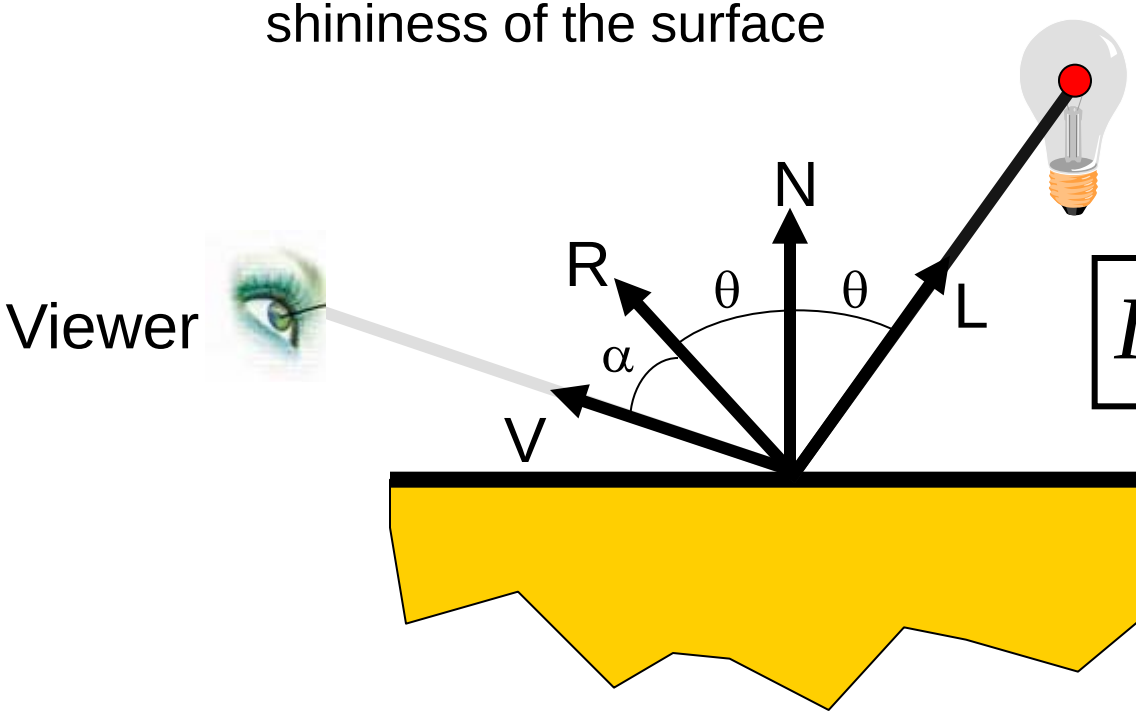
How much light is seen?

- Depends on angle of incident light and angle to viewer



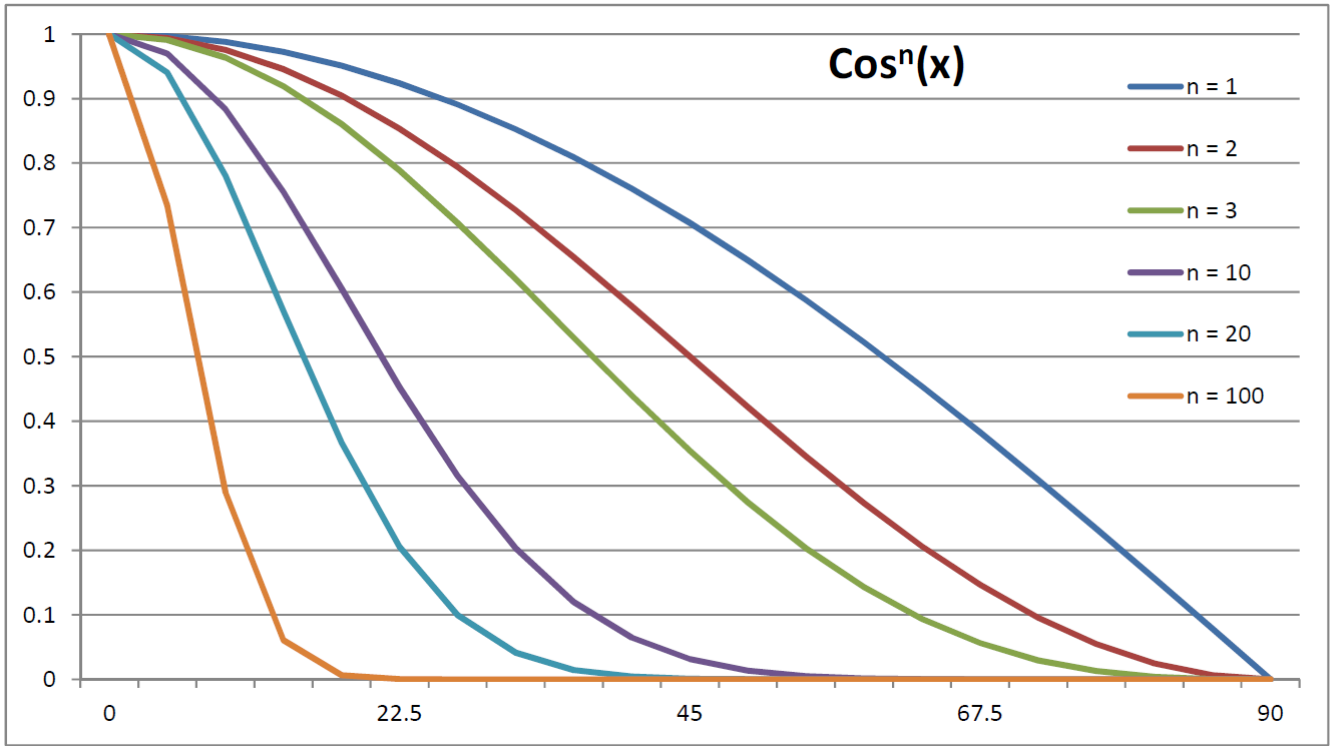
Specular Reflection

- Phong model: reflection depends on $\cos^n(\alpha)$
- Where n is the Phong exponent governing the apparent shininess of the surface

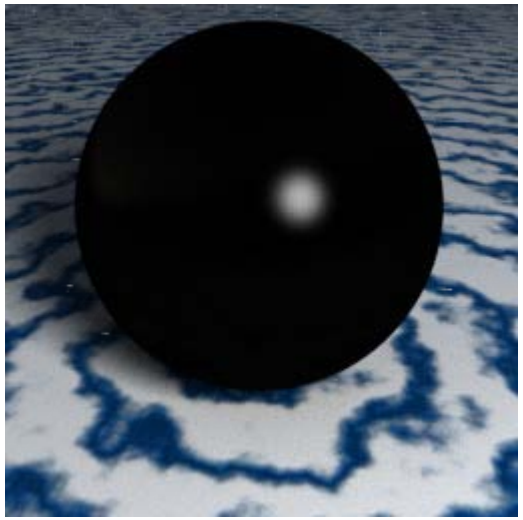


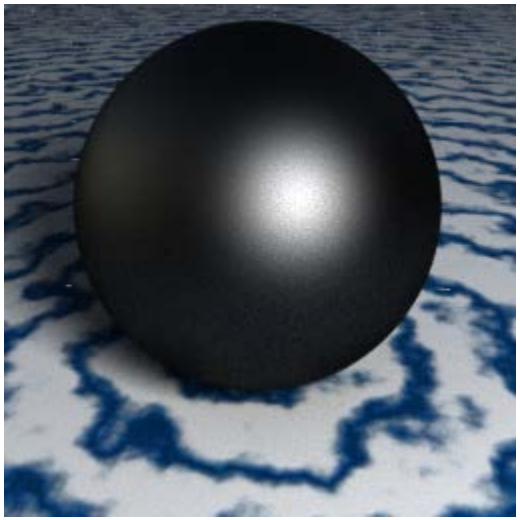
$$I_S = K_S (V \cdot R)^n I_L$$

Behavior of $(V \cdot R)^n$



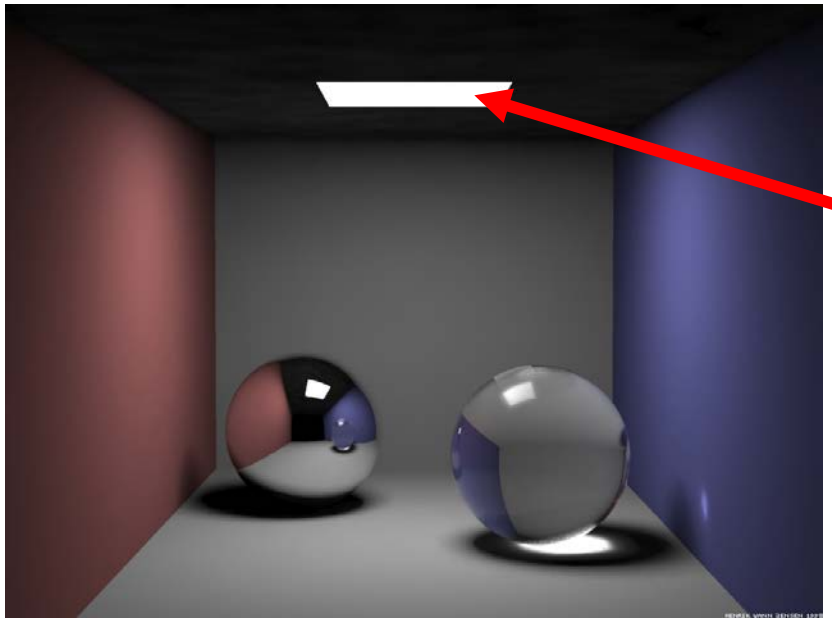
Examples of Different Exponent





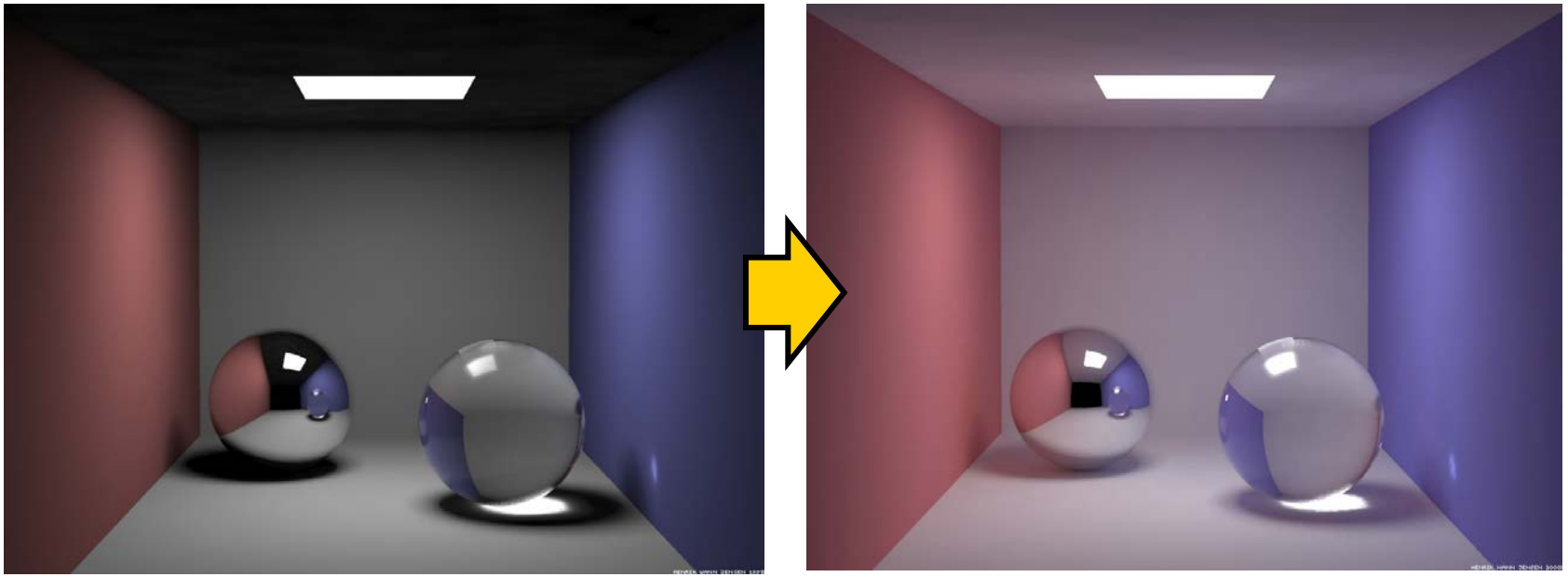
Light Emission

- Represents light emanating directly from object



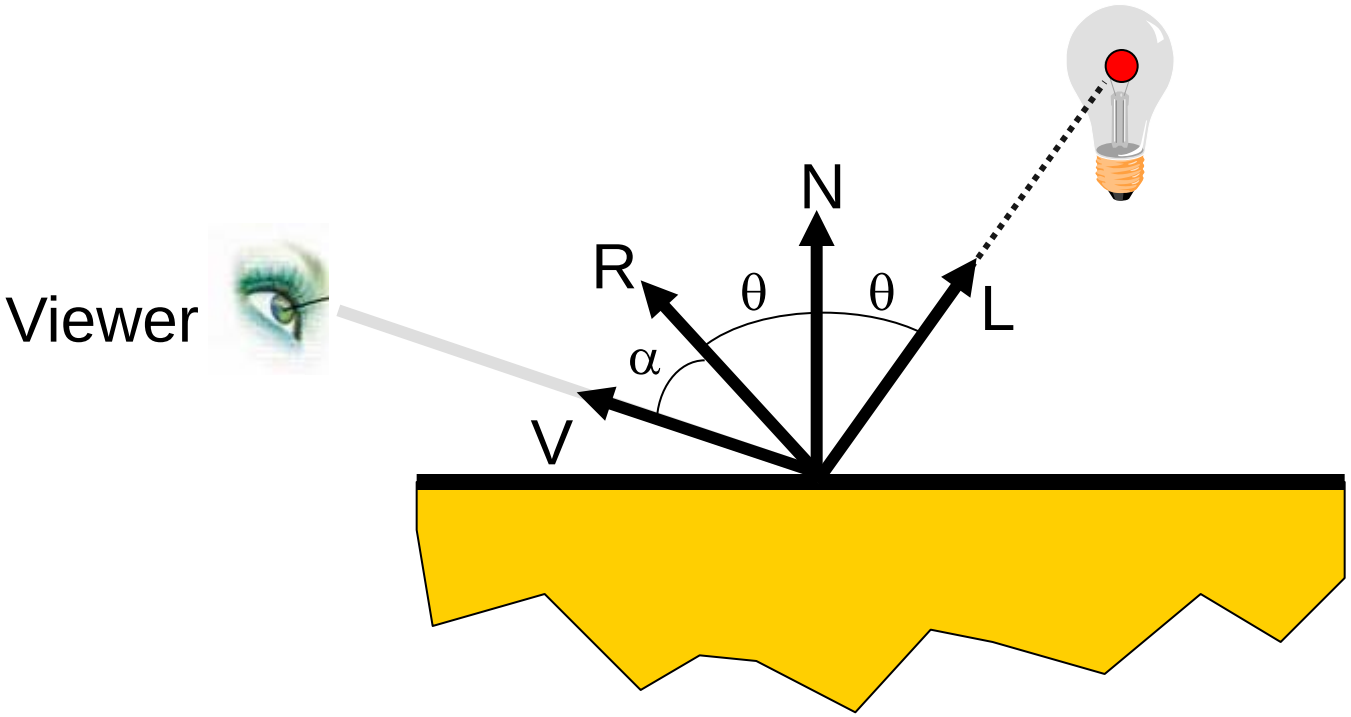
Ambient Term

- Represents reflection of all indirect illumination
- This is a total hack (avoids complexity of global illumination)!



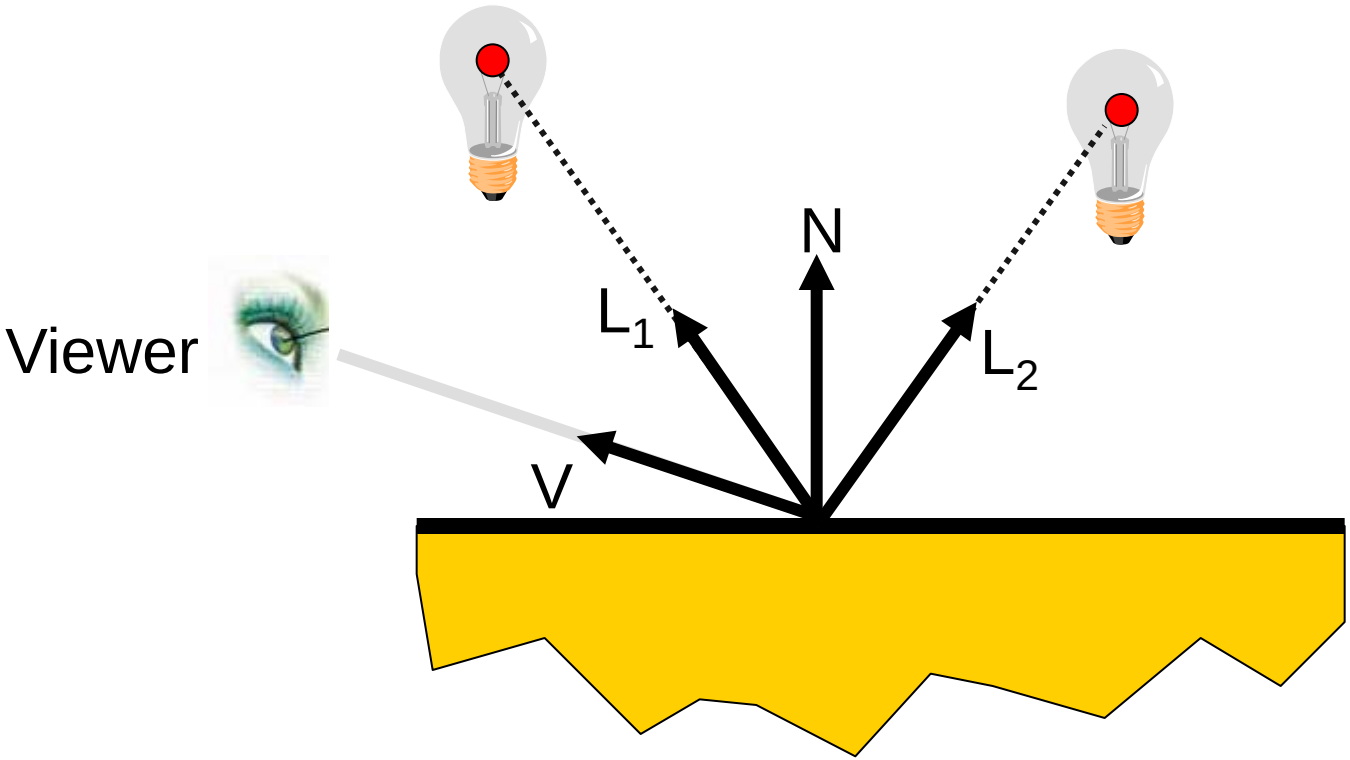
Single Light Illumination Calculation

$$I = I_E + K_A I_{AL} + K_D (N \cdot L) I_L + K_S (V \cdot R)^n I_L$$



Multiple Light Illumination Calculation

$$I = I_E + K_A I_{AMB} + \sum_i (K_D (N \cdot L_i) I_{L_i} + K_S (V \cdot R_i)^n I_{L_i})$$



For Every Light Wavelength?

$$I_\lambda = I_{\lambda E} + K_{\lambda A} I_{\lambda AMB} + \sum_i (K_{\lambda D}(N \cdot L_i) + K_{\lambda S}(V \cdot R_i)^n) I_{\lambda L_i}$$

Enough to calculate for the 3 base colors:

$$I_r = I_{rE} + K_{rA} I_{rAMB} + \sum_i (K_{rD}(N \cdot L_i) + K_{rS}(V \cdot R_i)^n) I_{rL_i}$$

$$I_g = I_{gE} + K_{gA} I_{bAMB} + \sum_i (K_{gD}(N \cdot L_i) + K_{gS}(V \cdot R_i)^n) I_{gL_i}$$

$$I_b = I_{bE} + K_{bA} I_{bAMB} + \sum_i (K_{bD}(N \cdot L_i) + K_{bS}(V \cdot R_i)^n) I_{bL_i}$$

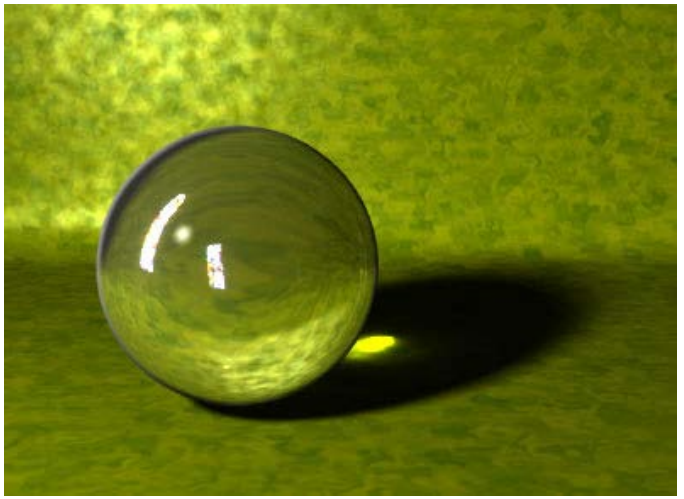
Get Color for Ray Casting

```
RGB GetColor(Scene scene, Ray in_ray, Point hit)
{
    // Ambient and Emission calculations
    RGB color = calcEmissionColor(scene) +
                calcAmbientColor(scene);
    // Diffuse & Specular calculations
    for (int i = 0; i < getNumLights(scene); i++) {
        Light light = getLight(i,scene);
        color += calcDiffuseColor(scene,hit,light) +
                 calcSpecularColor(scene,hit,light);
    }
    return color;
}
```

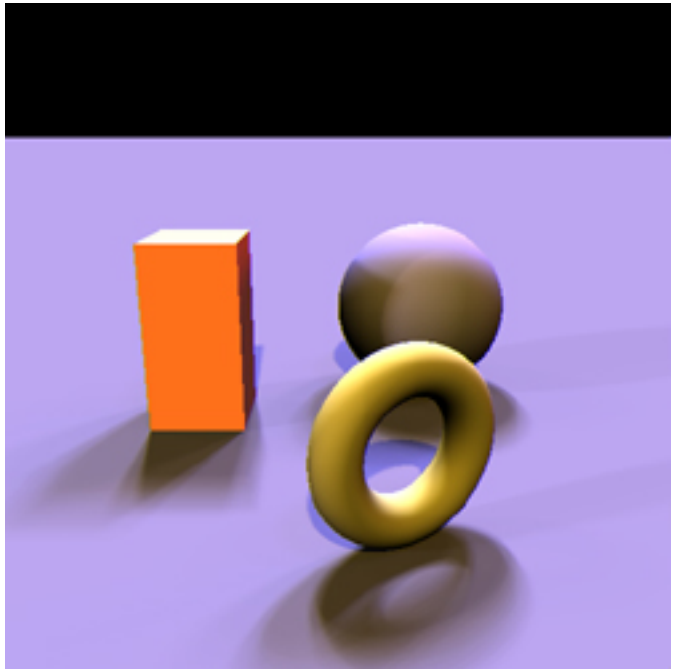
in_ray

Part III:
Recursive Ray Tracing

- Shadows
- Refractions
- Inter-object reflections



Shadows

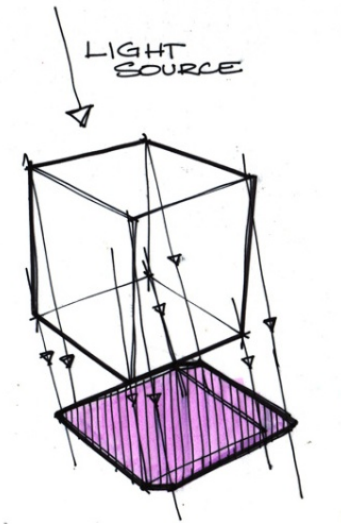


Shadows

- Occlusions from light sources by other objects
- Point light source creates hard edges
- Area light source create soft edges

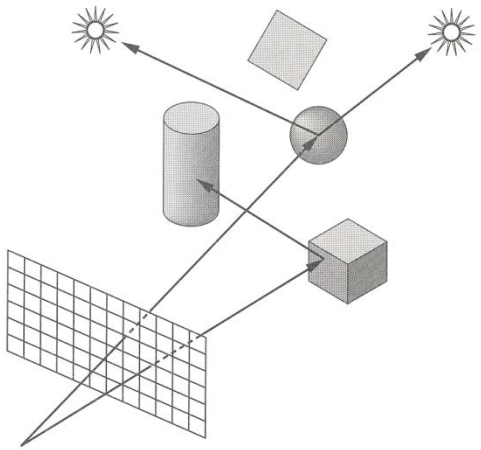
Shadow Calculation?

- Instead of calculating the shadow of each object explicitly - do an implicit calculation
  - For each **resulting intersection point** with ray
  - Shoot **“shadow rays”** to each light source
  - Check if a light source is blocked/occluded by other objects
  - Set if is in shadow.



Shadows Rays

- Shadow terms tell which light sources are blocked
  - Cast ray towards each light source L_i
  - $S_i = 0$ if ray is blocked, $S_i = 1$ otherwise



Shadow Term

$$I = I_E + K_A I_A + \sum_i (K_D (N \cdot L) + K_S (V \cdot R)^n) \boldsymbol{S_i I_i}$$

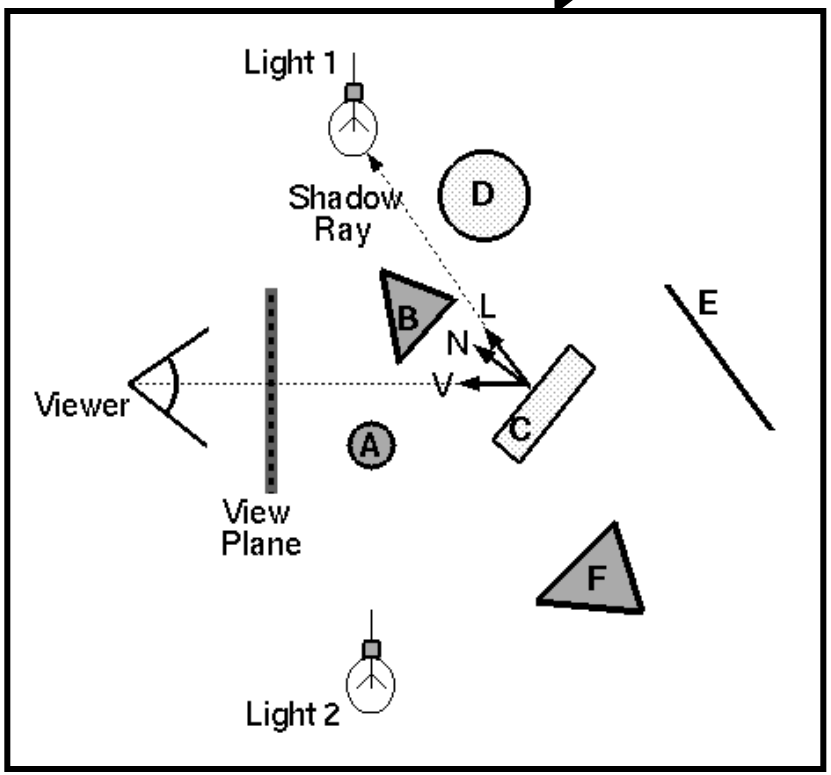
Get Color for Ray Casting with Shadows

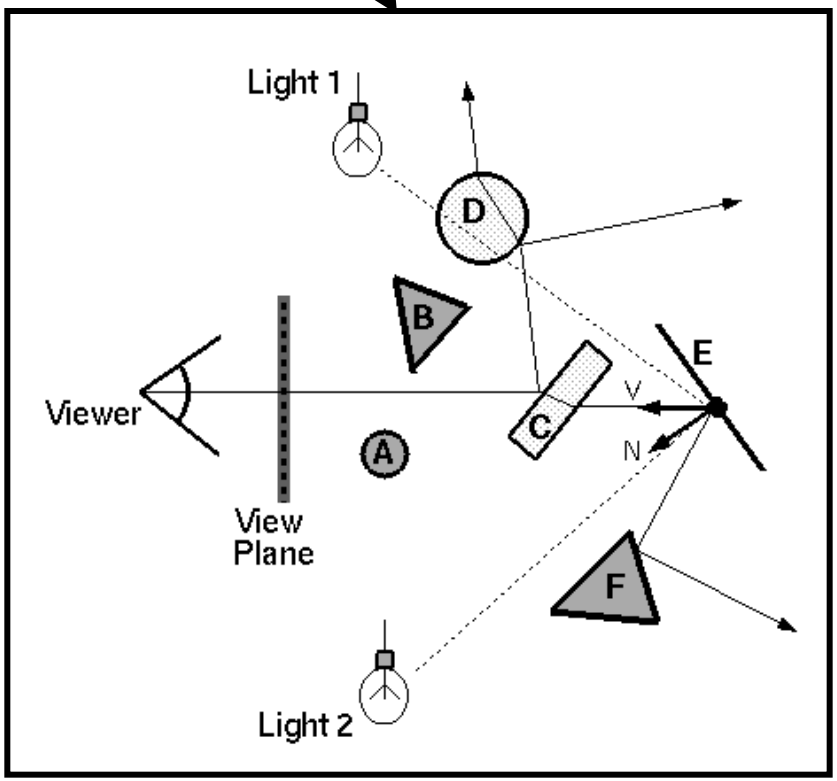
```
RGB GetColor(Scene scene, Ray in_ray, Point hit)
{
    // Ambient and Emission calculations
    RGB color = calcEmissionColor(scene) +
                calcAmbientColor(scene);
    // Diffuse and Specular calculations
    for (int i = 0; i < getNumLights(scene); i++) {
        Light light = getLight(i,scene);
        Ray light_ray = ConstructRaytoLight(hit, light)
        // Add color only if light is not occluded
        if !occluded(light_ray, scene, light) {
            color += calcDiffuseColor(scene,hit,light) +
                     calcSpecularColor(scene,hit,light);
        }
    }
    return color;
}
```

Recursive Ray Tracing

- Ray casting cannot account for global effects of interactions between objects such as reflections (השתקפות) and refractions (שבירה) of light.
- Ray tracing models these by recursively following the (reverse) secondary rays from the intersection points

Ray Casting vs. Tracing





Casting vs. Tracing Fromulas

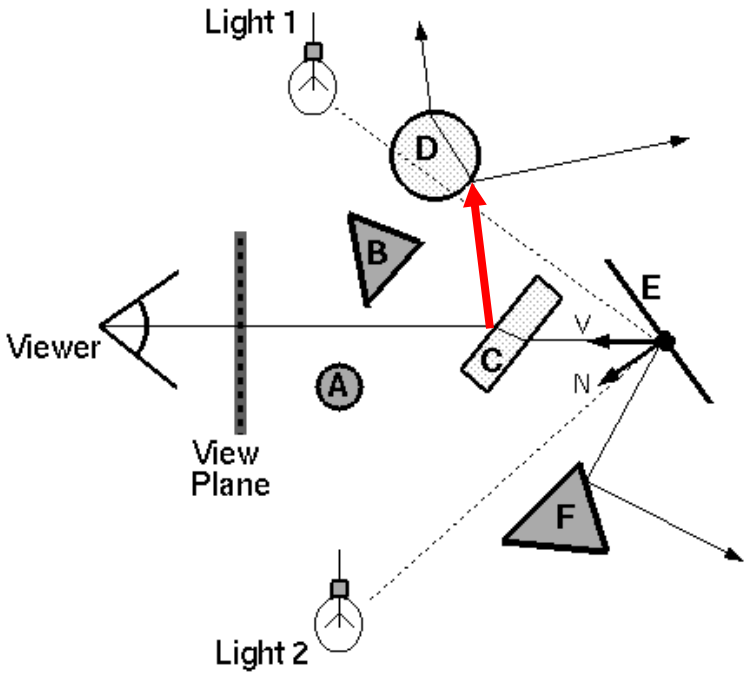
$$I = I_E + K_A I_A + \sum_L (K_D (N \cdot L) + K_S (V \cdot R)^n) S_L I_L$$

- Trace primary rays from camera: direct illumination from unblocked lights only

$$I = I_E + K_A I_A + \sum_L (K_D (N \cdot L) + K_S (V \cdot R)^n) S_L I_L + K_R I_R + K_T I_T$$

- Also trace secondary rays from hit surfaces - global illumination from
  - Reflection R
  - Transparency T

Mirror reflections

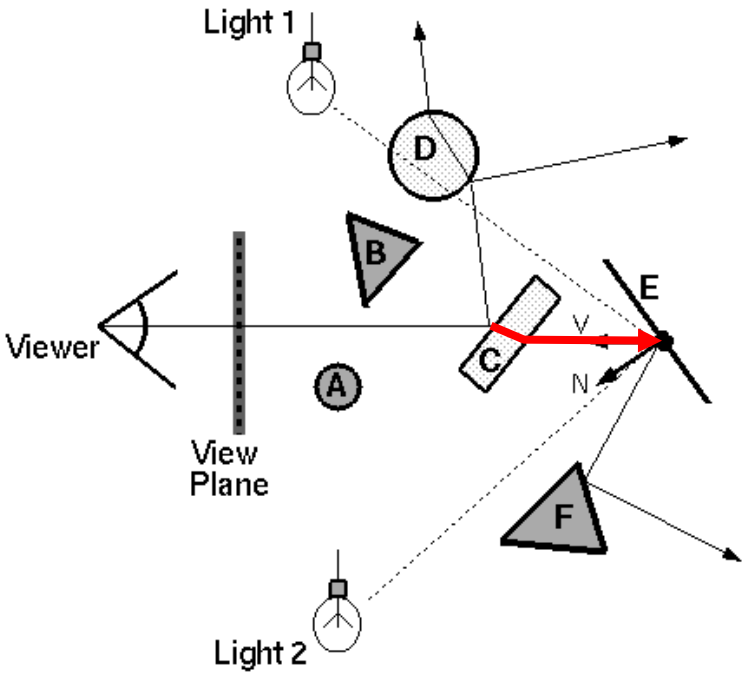


- Trace secondary ray in direction of mirror reflection
- Evaluate radiance along secondary ray and include it into illumination model

Radiance for mirror reflection ray

$$I = I_E + K_A I_A + \sum_L (K_D (N \bullet L) + K_S (V \bullet R)^n) S_L I_L + K_S I_R + K_T I_T$$

Transparency



- Trace secondary ray in direction of refraction and include it into illumination model
- Transparency coefficient is fraction transmitted
  - $K_T = 1$ if object is translucent,
  - $K_T = 0$ if object is opaque
  - $0 < K_T < 1$if object is semi-translucent

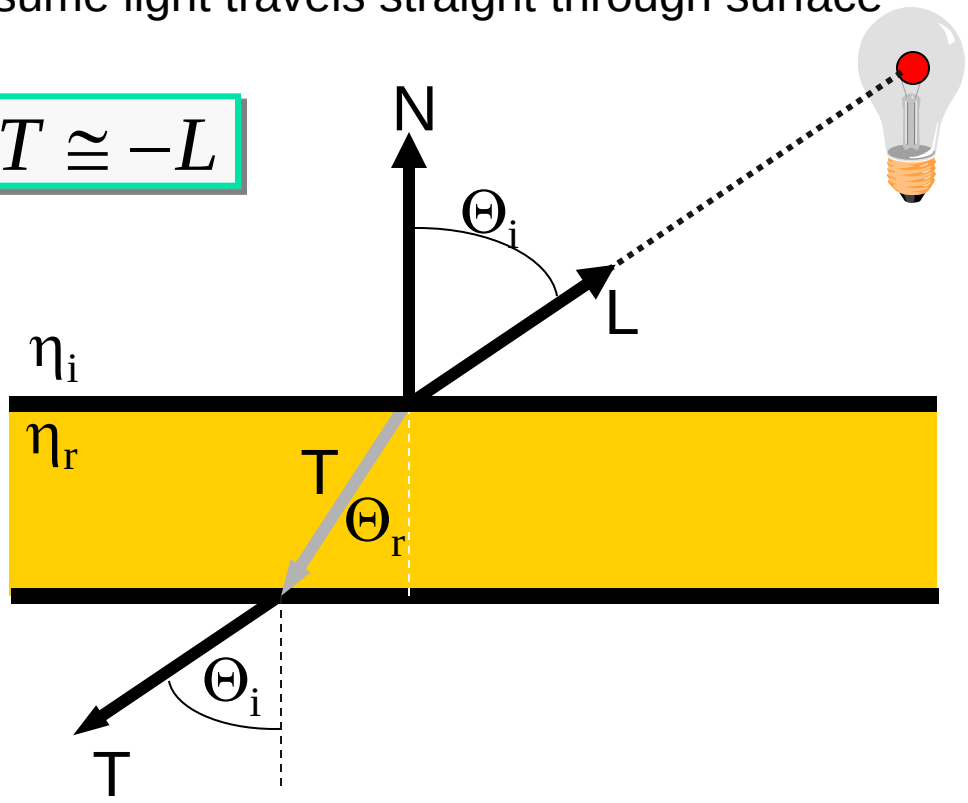
Radiance for refraction ray

$$I = I_E + K_A I_A + \sum_L (K_D (N \bullet L) + K_S (V \bullet R)^n) S_L I_L + K_S I_R + K_T I_T$$

Refractive Transparency

- For thin surfaces, can ignore change in direction
  - Assume light travels straight through surface

$$T \cong -L$$

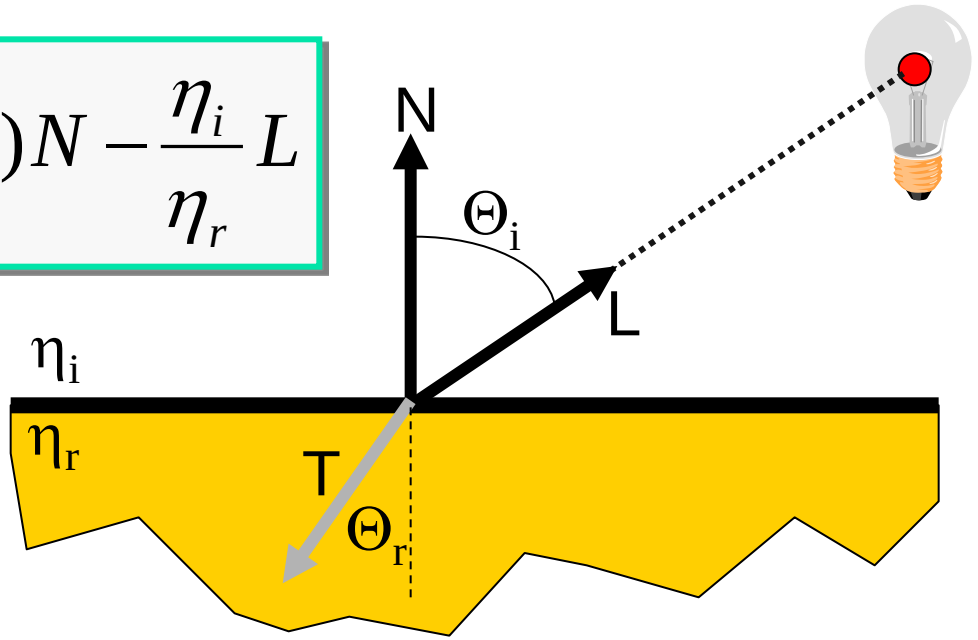


Refractive Transparency

For solid objects, apply Snell's law:

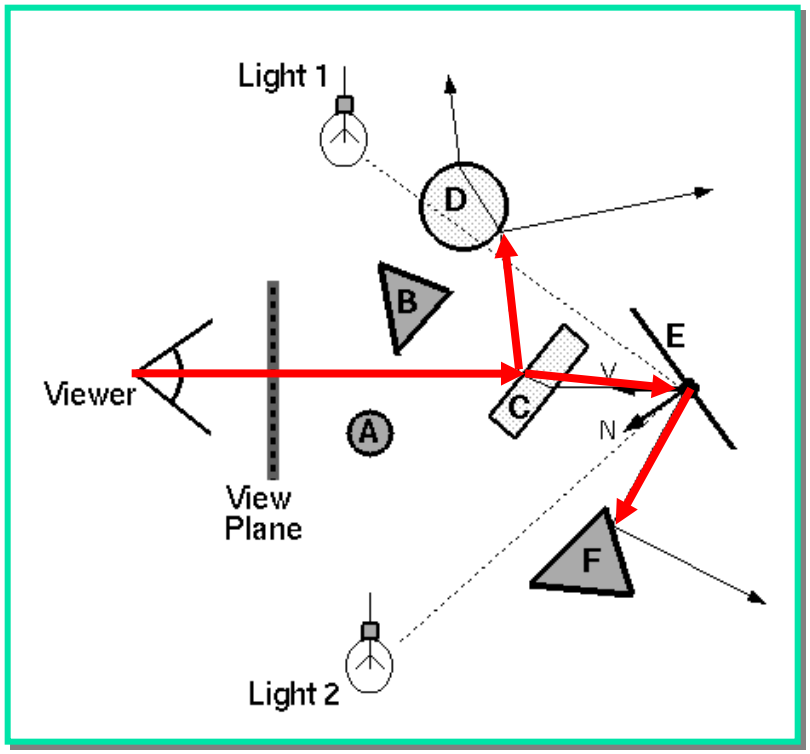
$$\eta_r \sin\Theta_r = \eta_i \sin\Theta_i$$

$$T = \left(\frac{\eta_i}{\eta_r}\cos\Theta_i - \cos\Theta_r\right)N - \frac{\eta_i}{\eta_r}L$$

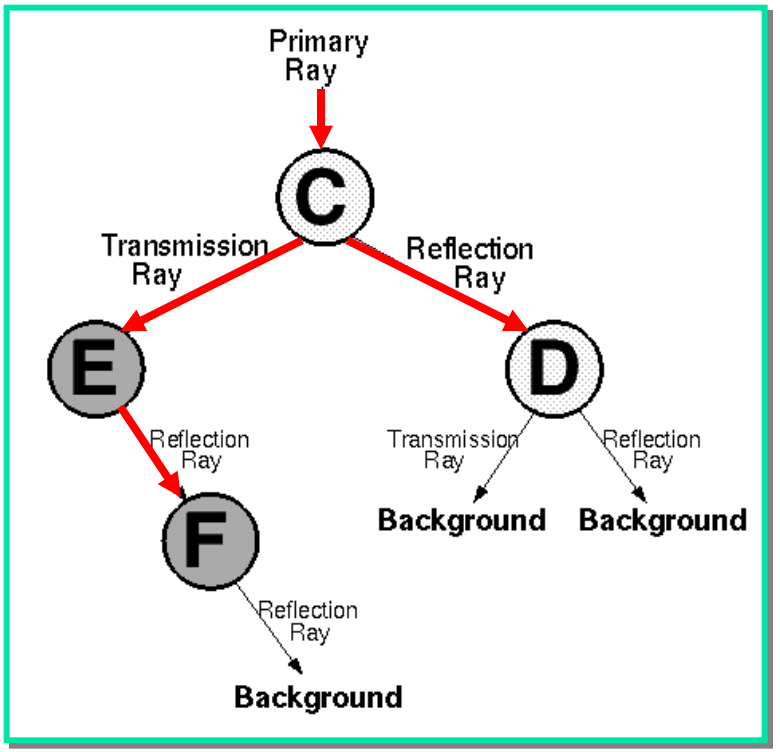


Recursion Ray Tree

- Ray tree represents illumination computation



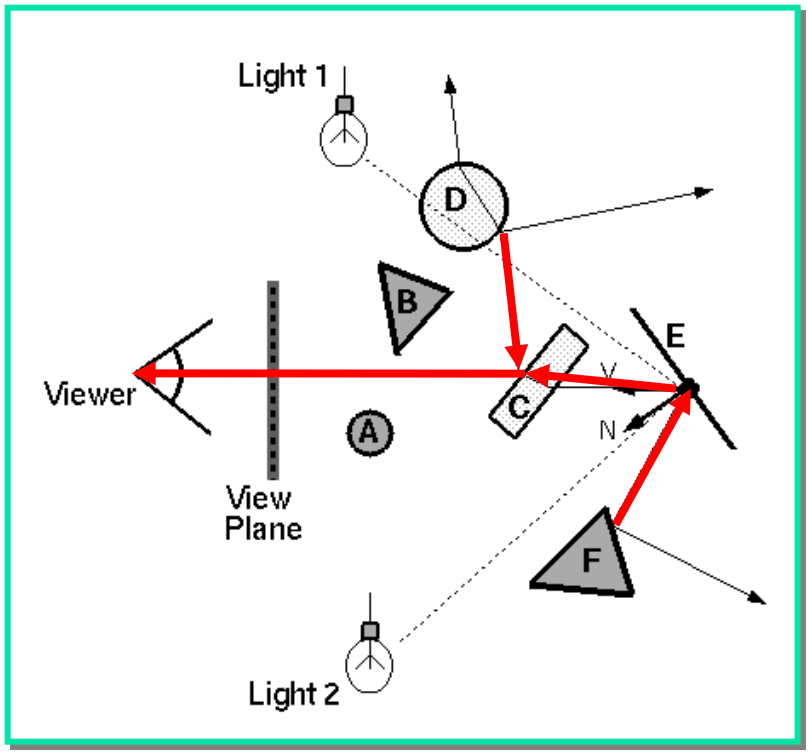
Ray traced through scene



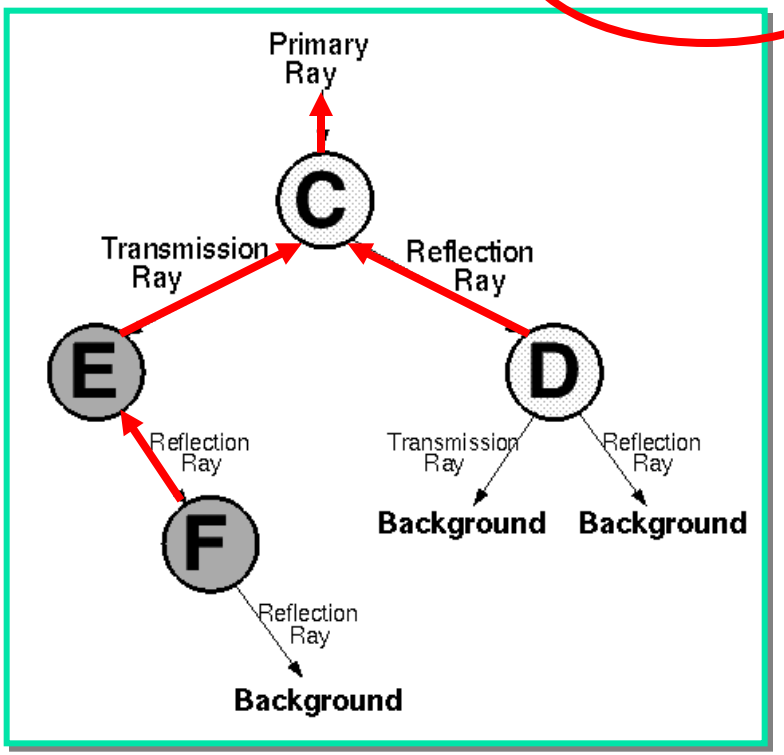
Ray tree

Gather Components

$$I = I_E + K_A I_A + \sum_L (K_D (N \bullet L) + K_S (V \bullet R)^n) S_L I_L + K_S I_R + K_T I_T$$



Ray traced through scene



Ray tree

Ray Tracing

```
Image RayTrace(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);
            image[i][j] = calcColor(scene, ray, hit,0);
        }
    }
    return image;
}
```

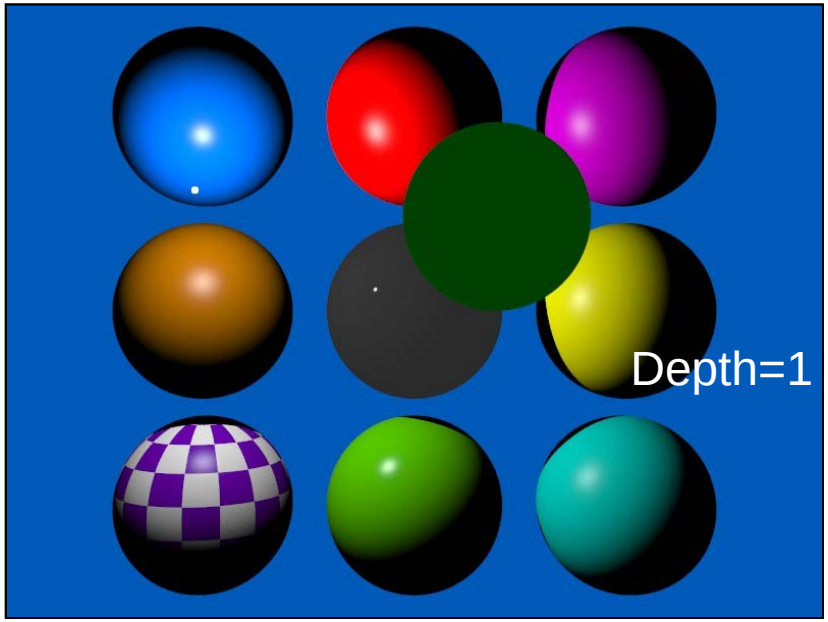
Recursive Calculation of Color for Ray Tracing

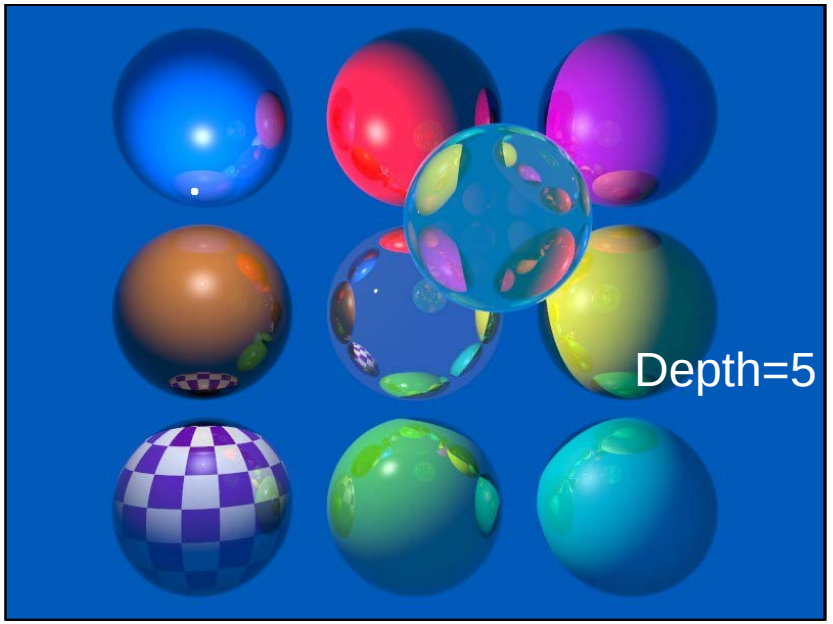
Regular part like ray casting (Note: without shadows!)

Recursive part

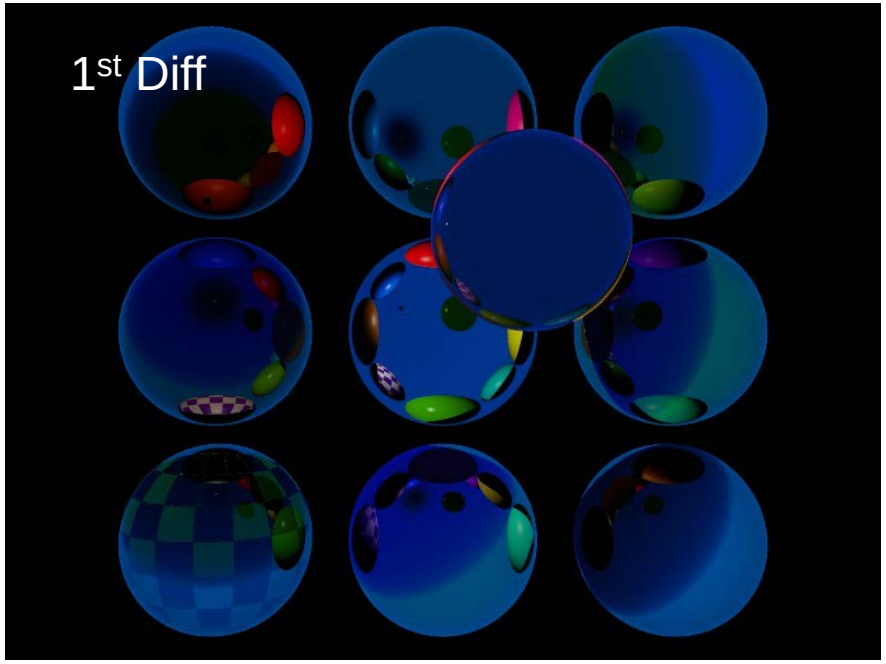
```
RGB CalcColor(Scene scene, Ray in_ray, int level)
{
    Point hit = FindIntersection(in_ray, scene);
    RGB color = calcEmissionColor(scene) +
                calcAmbientColor(scene);
    for (int i = 0; i < getNumLights(scene); i++) {
        Light light = getLight(i,scene);
        color += calcDiffuseColor(scene,hit,light) +
                 calcSpecularColor(scene,hit,light);
    }
    if (level == MAX_LEVEL)
        return rgb(0,0,0);
    Vector normal = getNormalAtPoint(hit);
    Ray out_ray = ConstructOutRay (in_ray, normal);
    color += K_s * CalcColor(scene, out_ray, level+1);
    return color;
        Here should come a similar part for
            a Refractive Ray K_r!!!
}
```

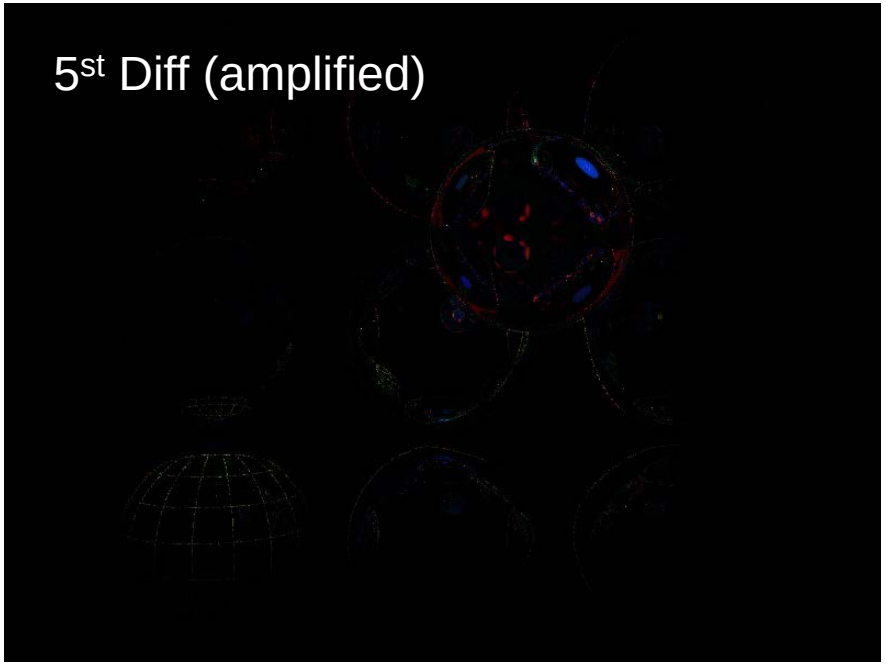
Effect of Recursion Depth



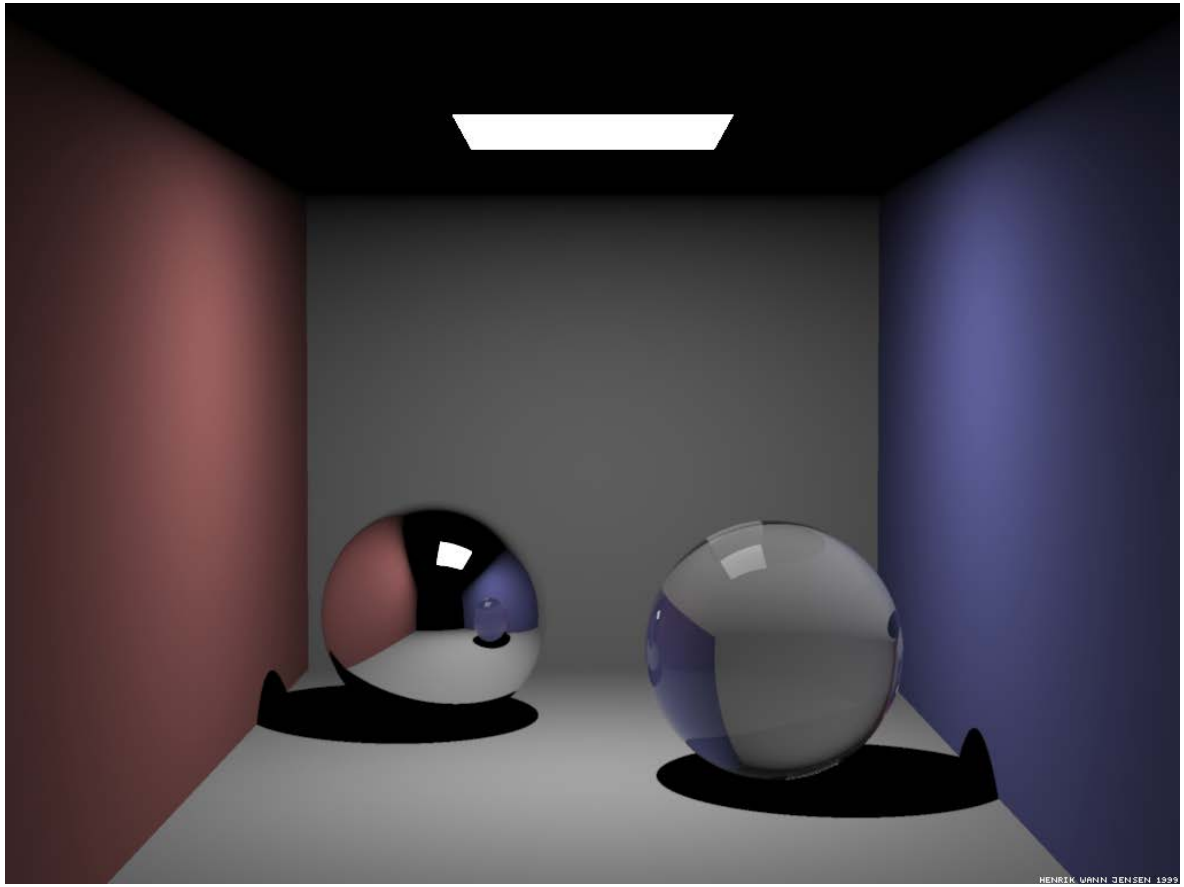


Difference Between Levels

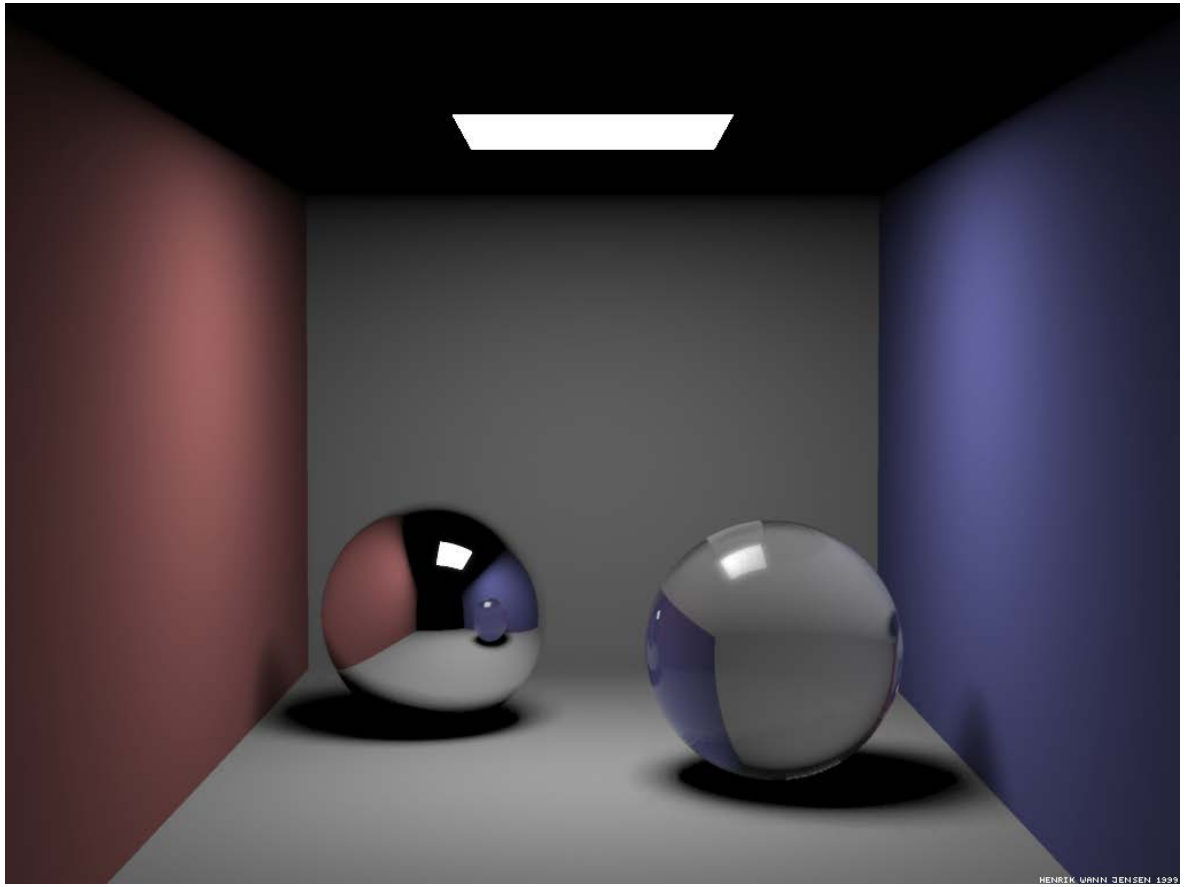




Direct Diffuse + Indirect Specular and transmission

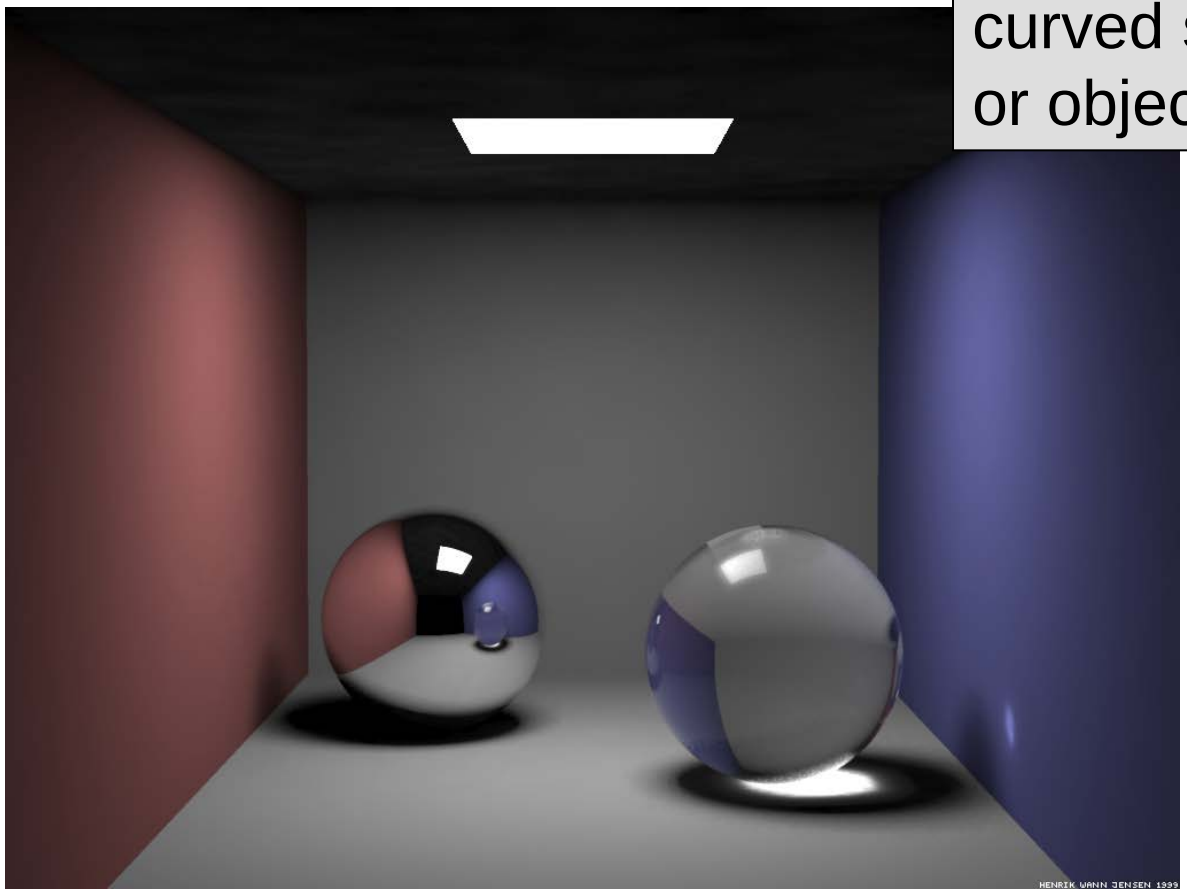


+ Soft Shadows

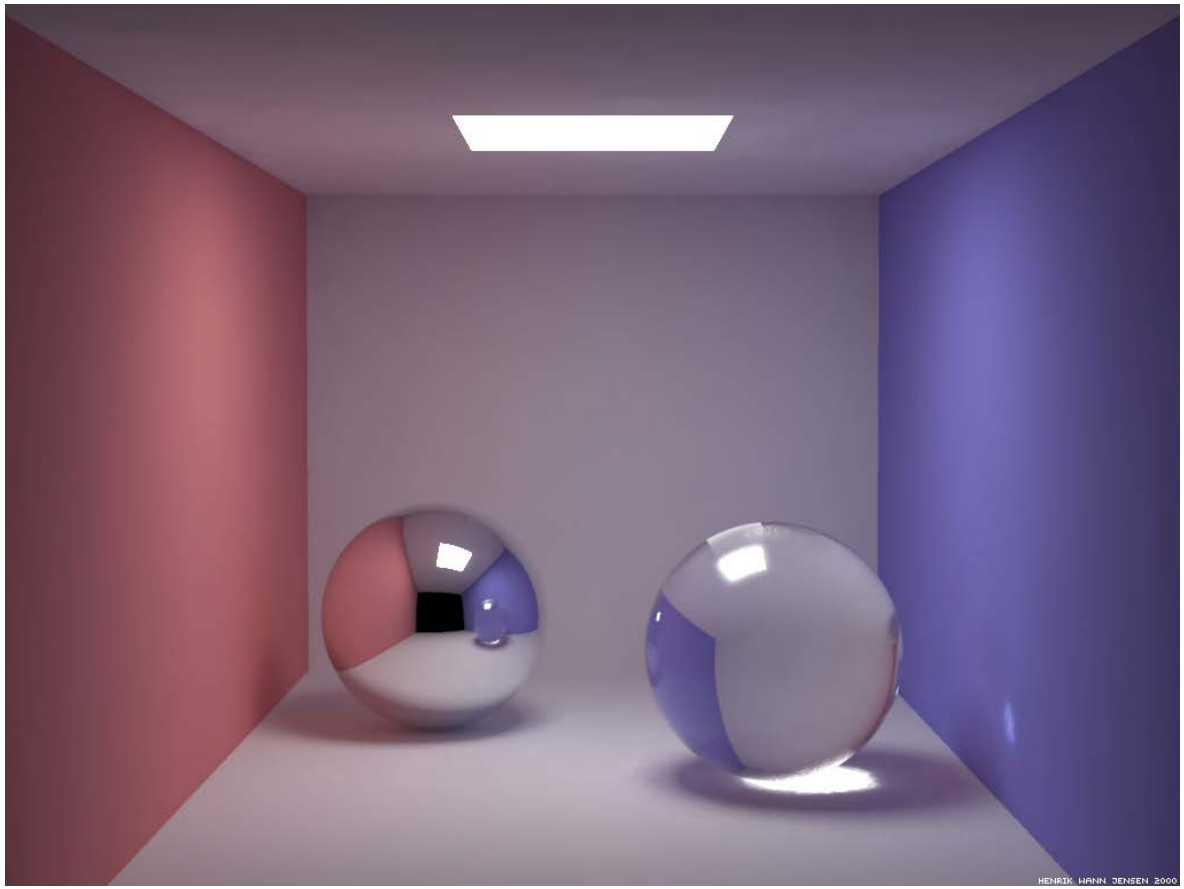


+ Caustics

= the envelope of light rays reflected or refracted by a curved surface or object



+ Indirect Diffuse Illumination



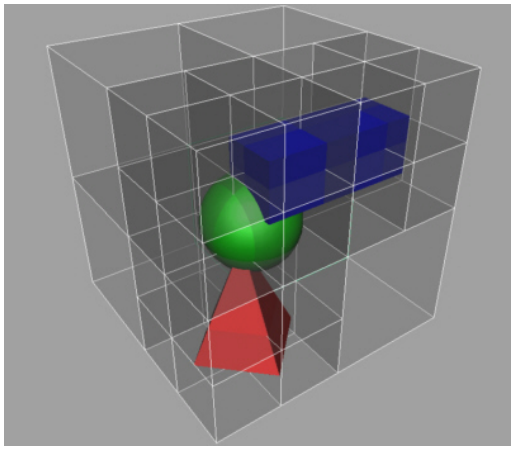
Example

Example

"Balanza" © Jaime Vives Piqueres (2002)

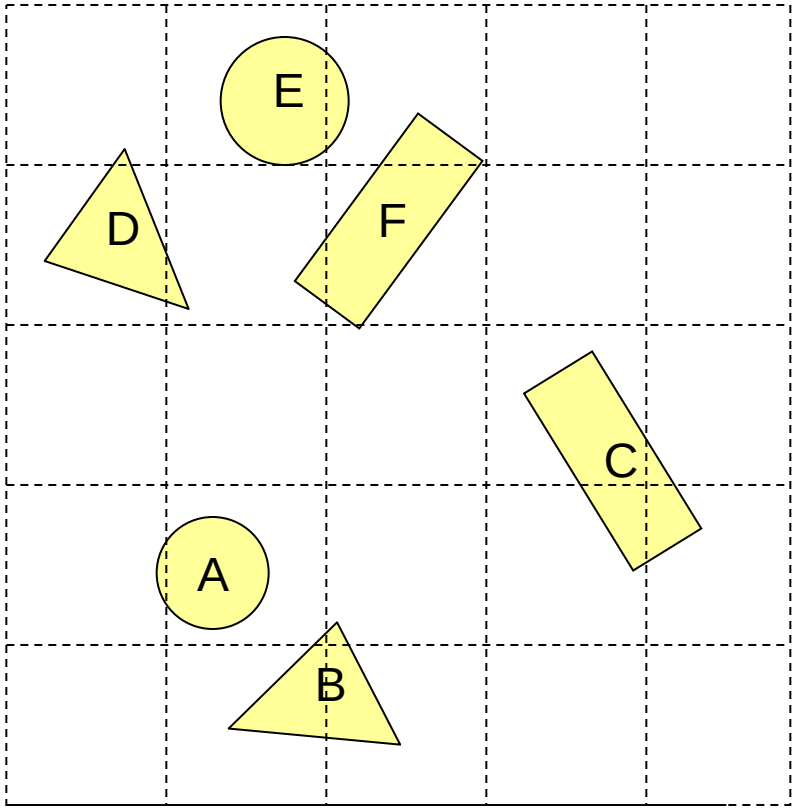
Part IV: Acceleration Techniques

1. Bounding volume hierarchies
2. Spatial partitions:
   - Uniform grids
   - Octrees
   - BSP trees



Uniform Grid

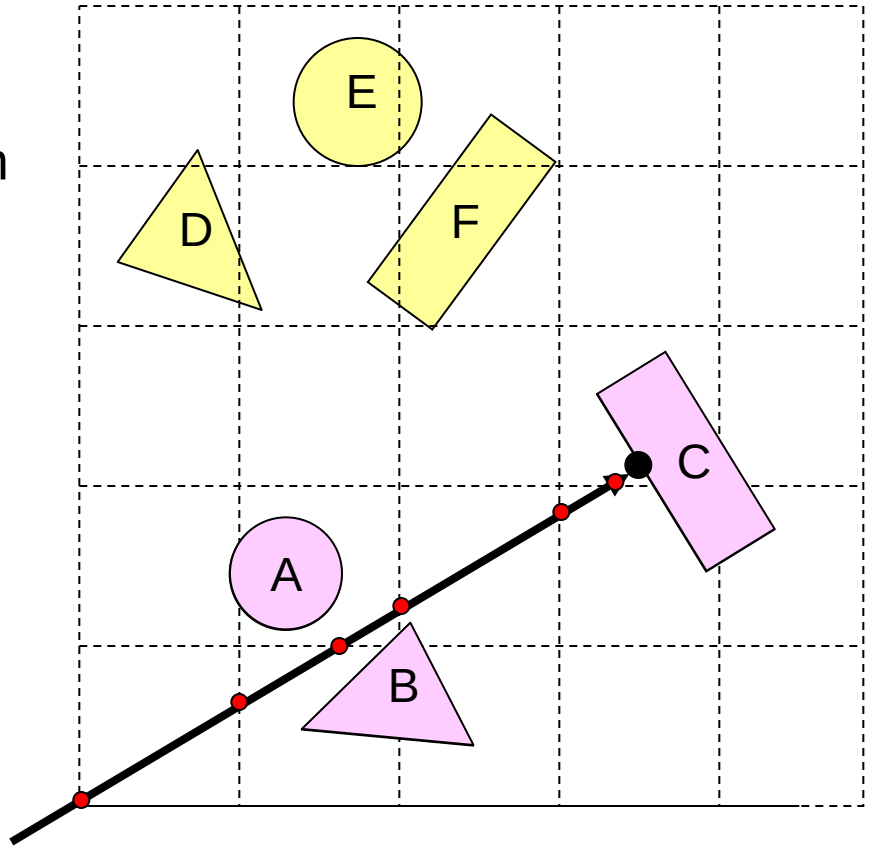
- Construct uniform grid over scene
- Index primitives according to overlaps with grid cells



Uniform Grid

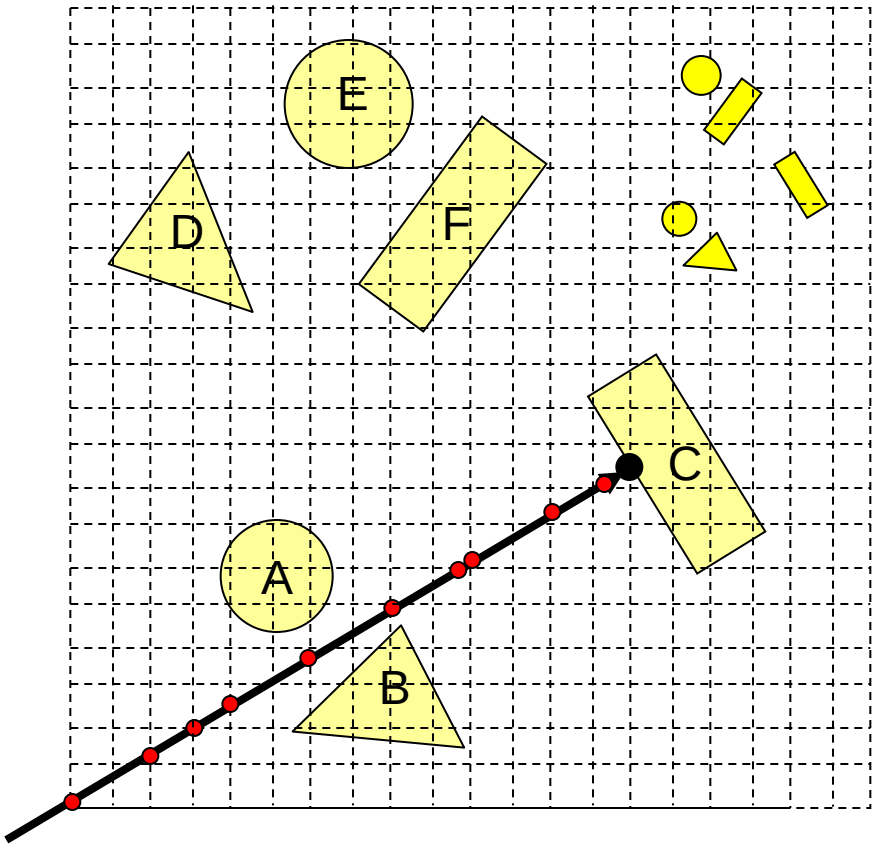
$p(t)=p_0+v(t)$
$p_{0x}+1=p_{0x}+v_xt$
$1/v_x=t$
$p_y = p_{0y}+v_y/v_x$

- Trace rays through grid cells.
  - Fast & Incremental: given an entry point into a cell and a vector, its easy to calculate exit point
- Only check primitives in intersected grid cells



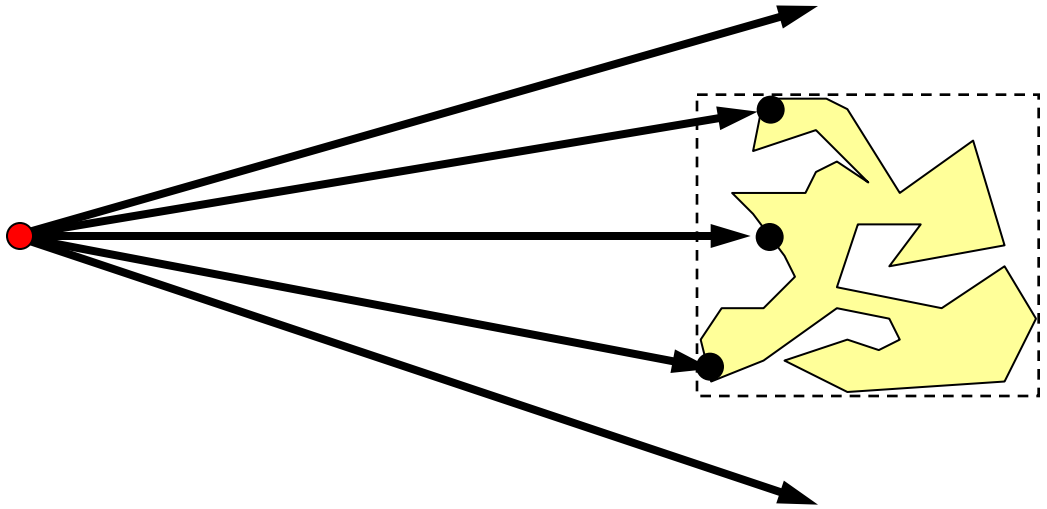
Uniform Grid

- Potential problem:
  - How choose suitable grid resolution?
- Too little benefit if grid is too coarse
- Too much cost if grid is too fine



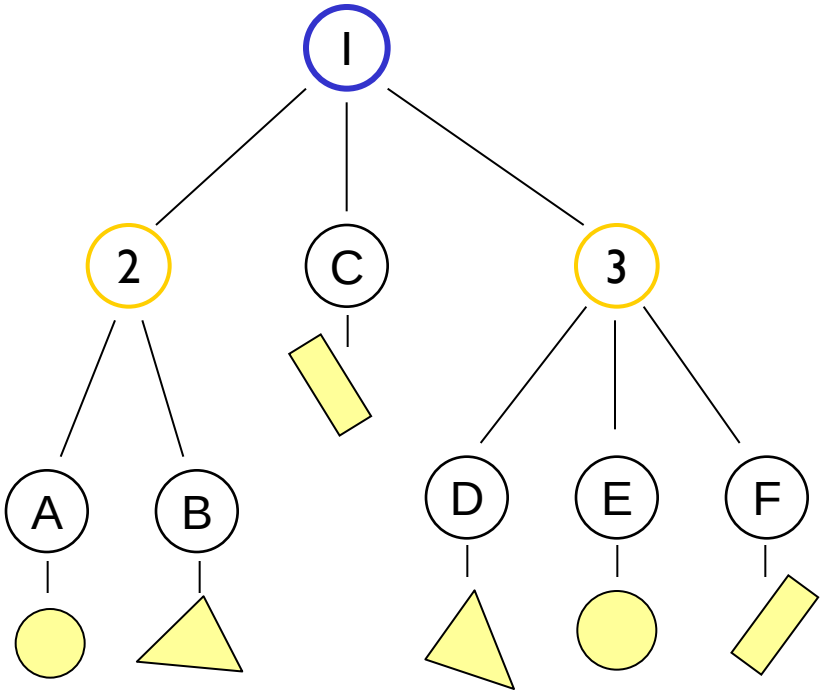
Bounding Volumes

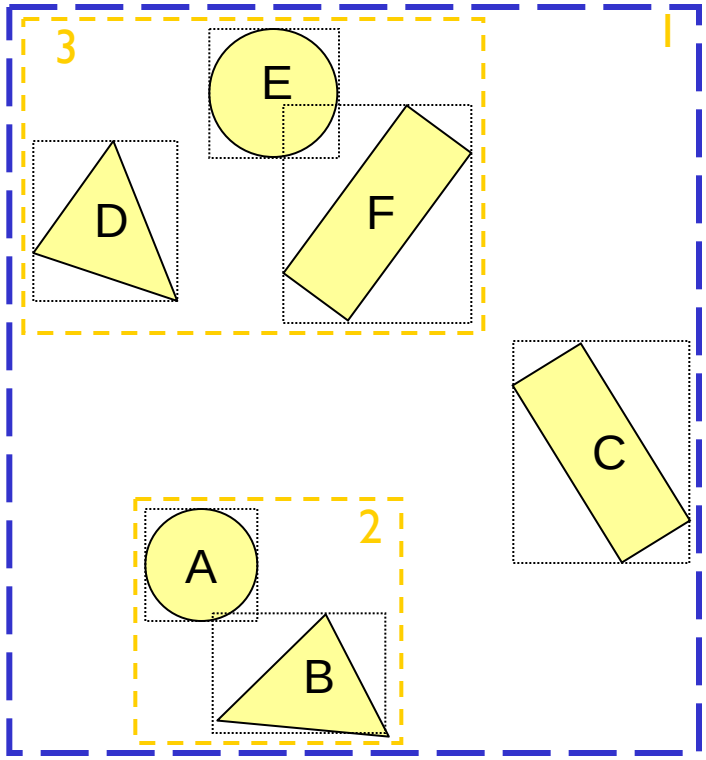
- Key idea: check for intersection with simple shape first
- Use bounding volume for shapes
- If ray doesn’t intersect bounding volume, then it doesn’t intersect its contents!



Bounding Volume Hierarchies

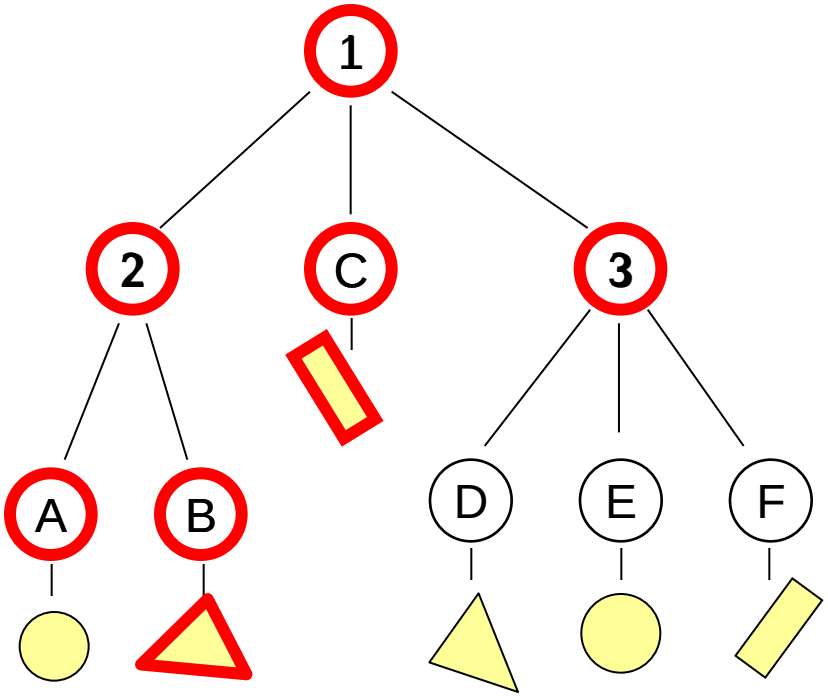
- Build hierarchy of bounding volumes
- Bounding volume of interior node contains all children

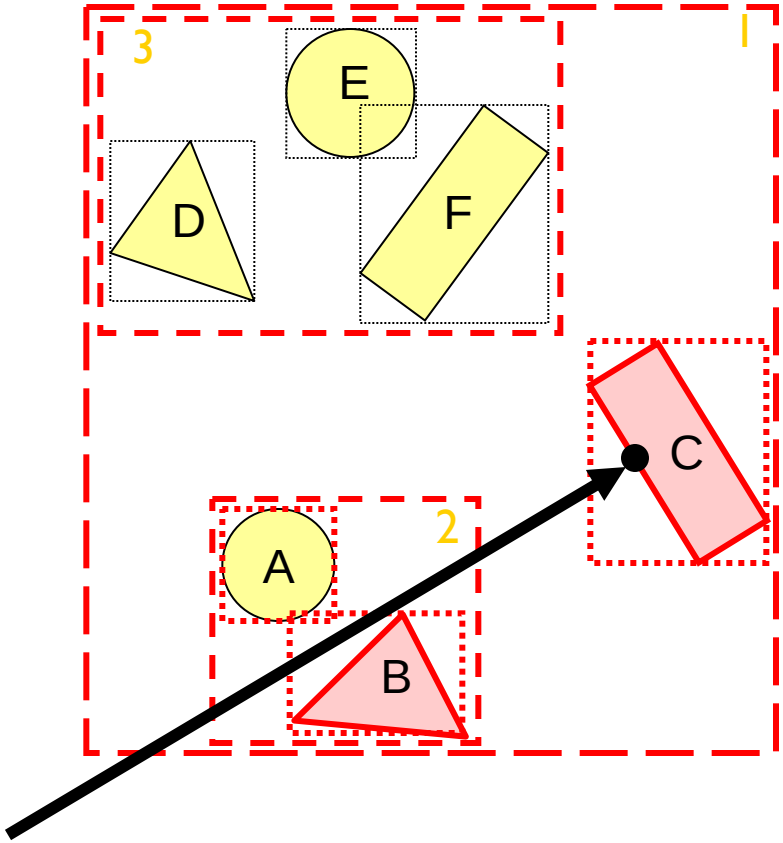




Ray Cast Using BVH

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

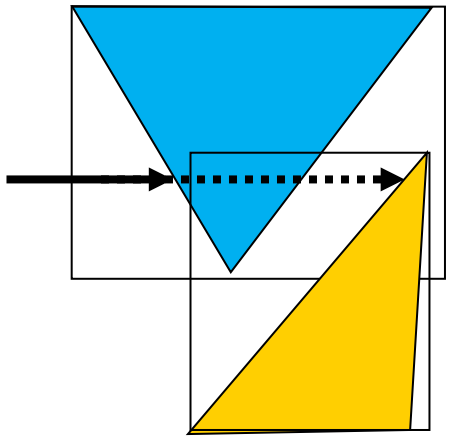




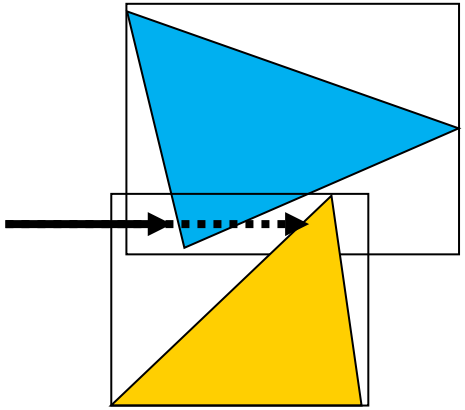
Sort Hits & Detect Early Termination

```
FindIntersection(Ray ray, Node node)
{
    // Find intersections with bounding volumes
    // of child node
    ...
    // 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;
}
```

Sort Hits & Detect Early Termination



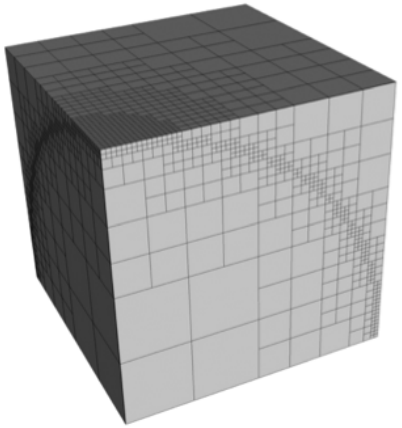
yes

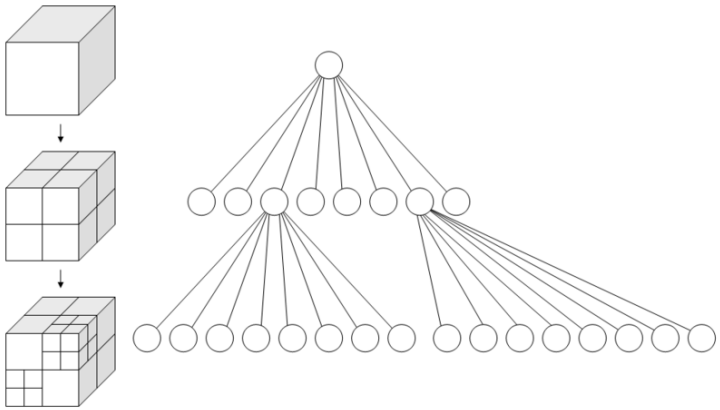


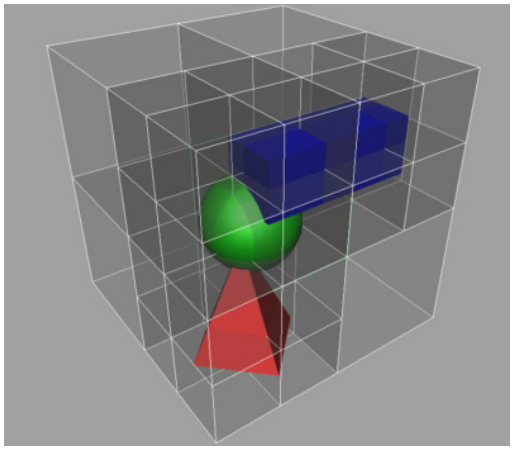
no

Octree

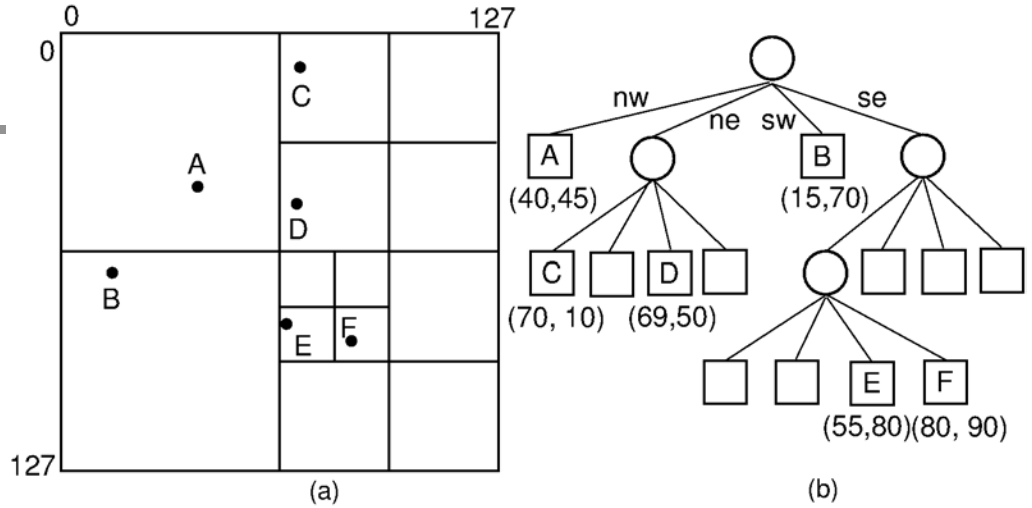
- An octree is a tree data structure that represents a recursive, hierarchical subdivision of 3-dimensional space into cubes.
- Each internal node represents a cube that is divided by 3 axis aligned planes into 8 equal size sub-cubes.



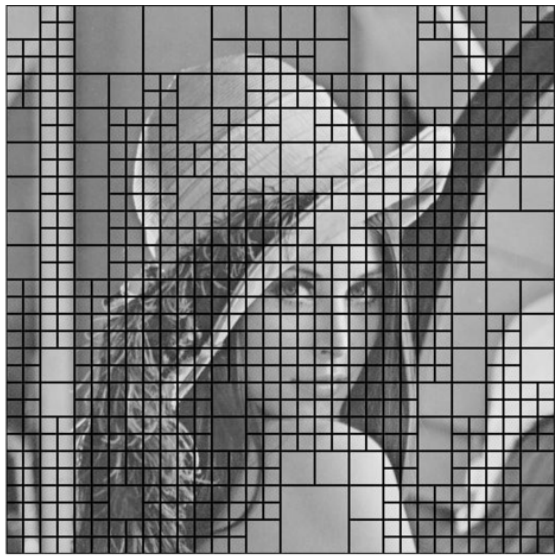




Quadtree in 2D



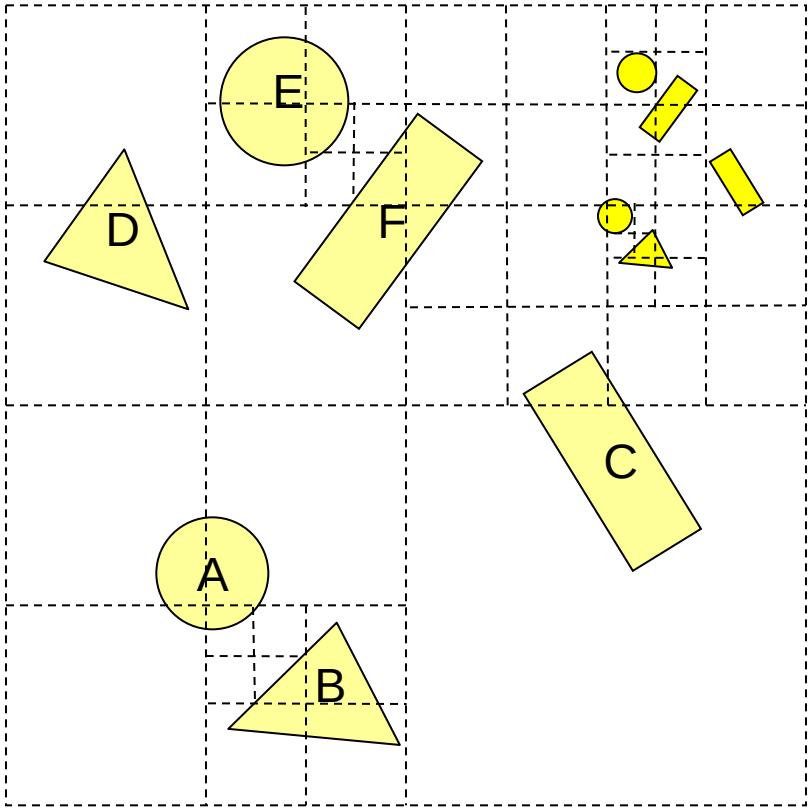
(a)



(b)

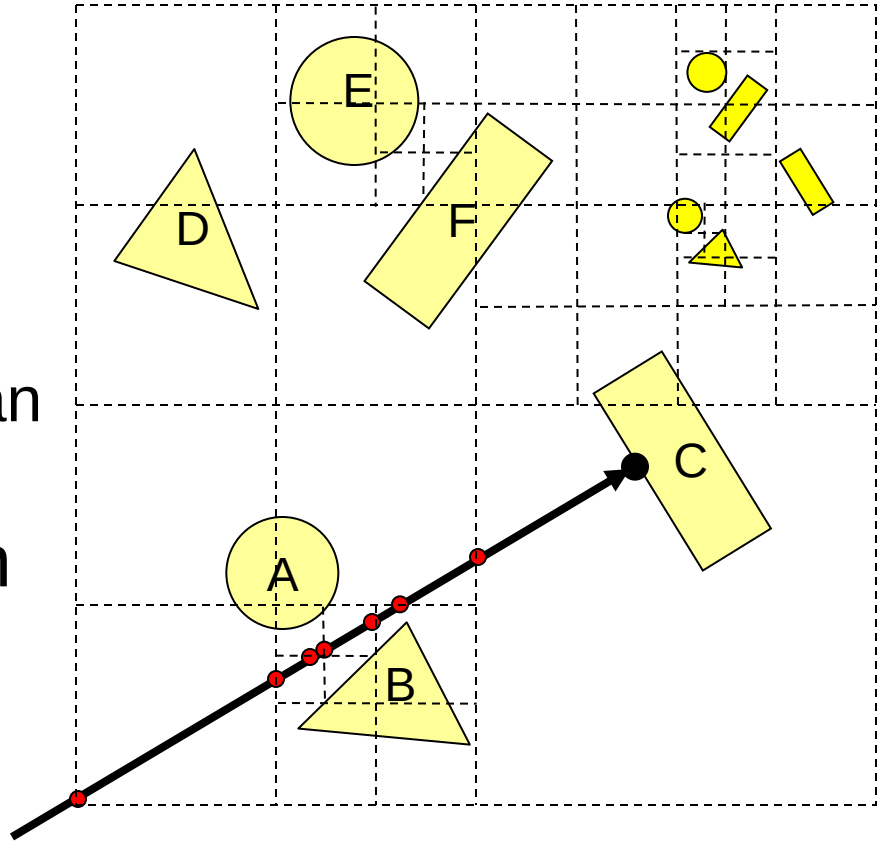
Octree Scene Construction

- Constructs adaptive grid over scene:
  - Recursively subdivide box-shaped cells into 8 octants
  - Index primitives by overlaps with cells
- Fewer cells than a grid



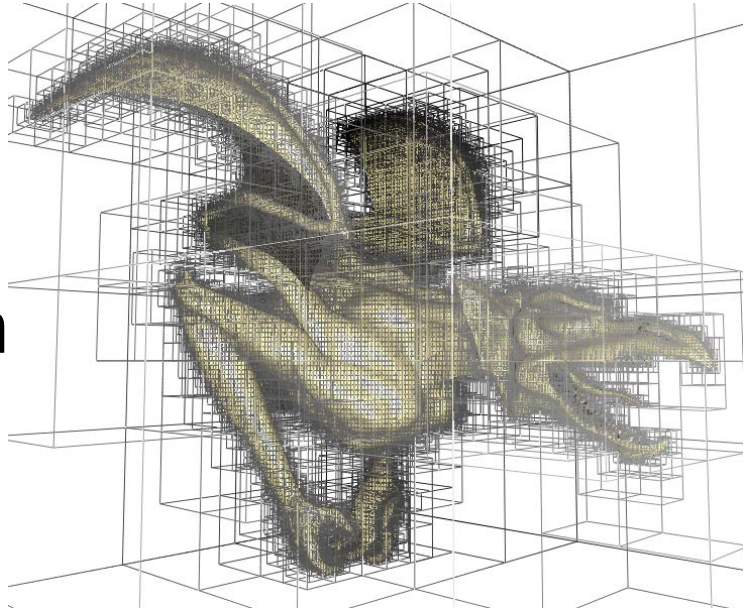
Ray Cast with Octree

- Trace rays through neighbor cells
  - Fewer cells but…
  - More complex neighbor finding than a grid
- A trade-off between fewer cells and more expensive traversal



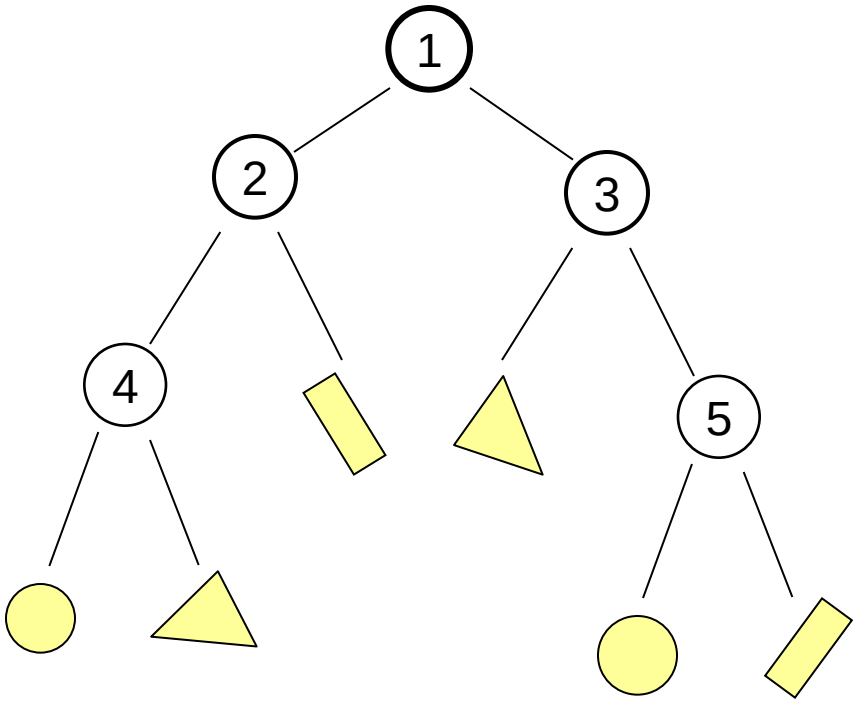
Other Uses of Octree

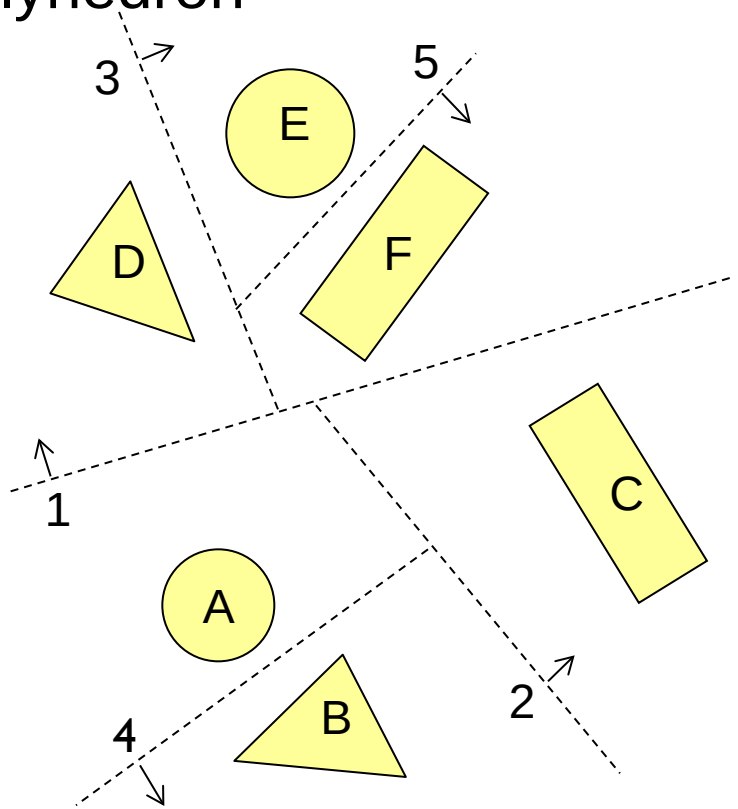
- Very useful in computer graphics for:
  - Intersections
  - Collisions
  - Color quantization
  - Surface reconstruction
  - …



Binary Space Partition (BSP) Tree

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

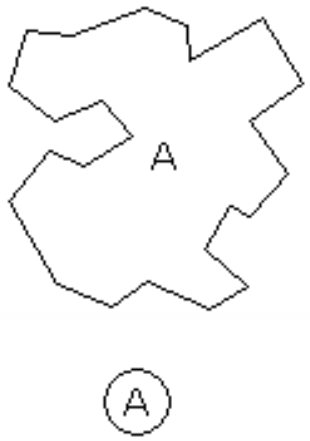




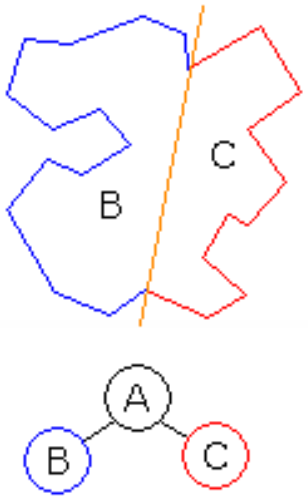
HOW DO YOU BUILD A BSP TREE?

- Select a partition plane.
- Partition the set of polygons with the plane.
- Recurse with each of the two new sets.

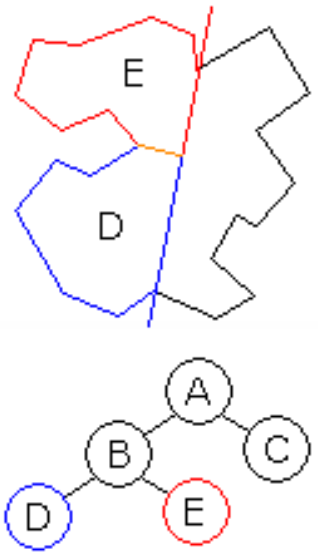
1.



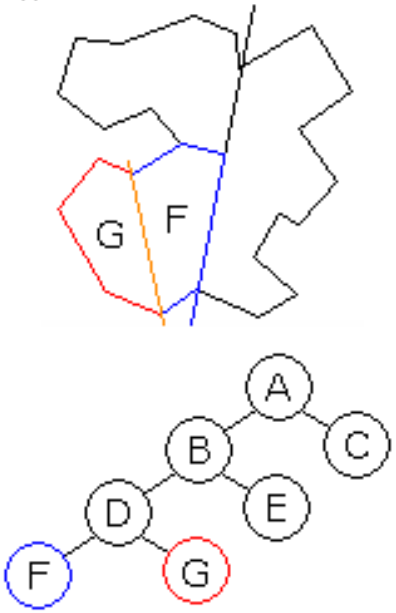
2.



3.



4.

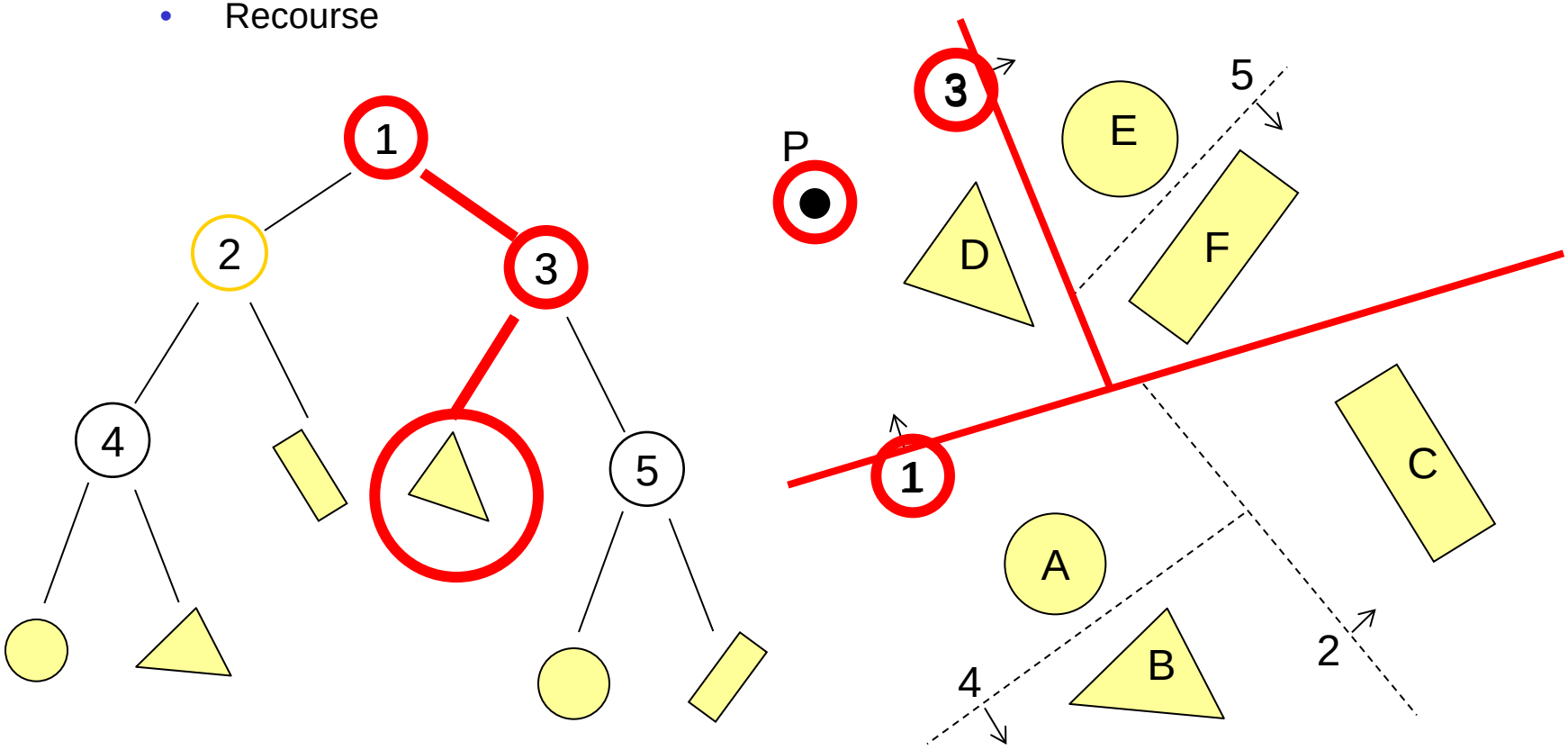


Choosing the partition plane

- Partition plane from the input set of polygons (called an autopartition).
- Axis aligned orthogonal partitions
- Balance tree, where each leaf contains roughly the same number of polygons

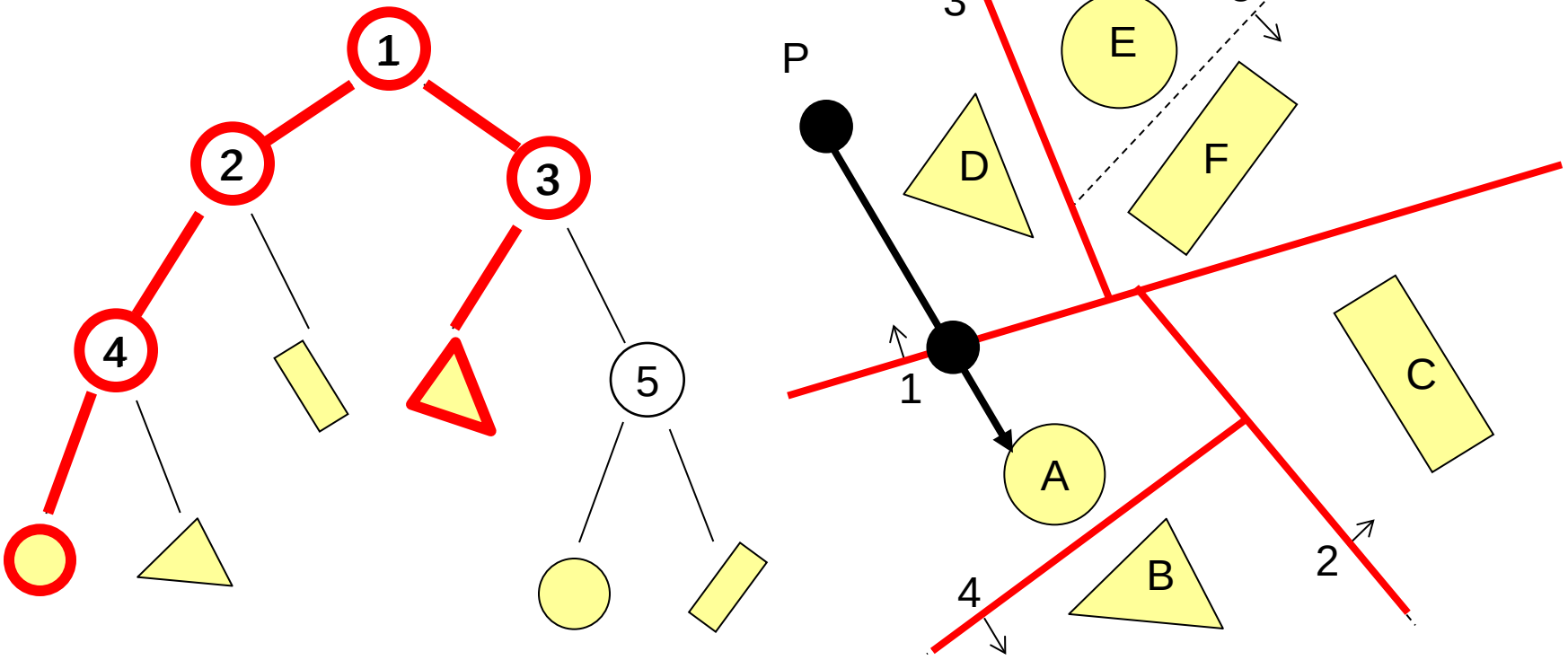
BSP Point Finding

- Start at root
- Go –left/right by plane-point relation
- Recourse

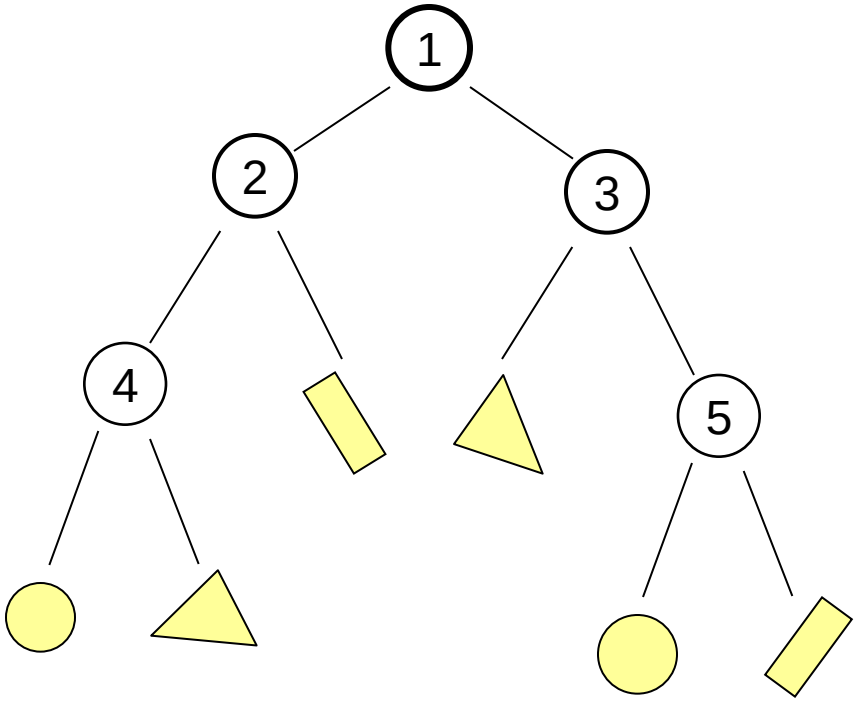


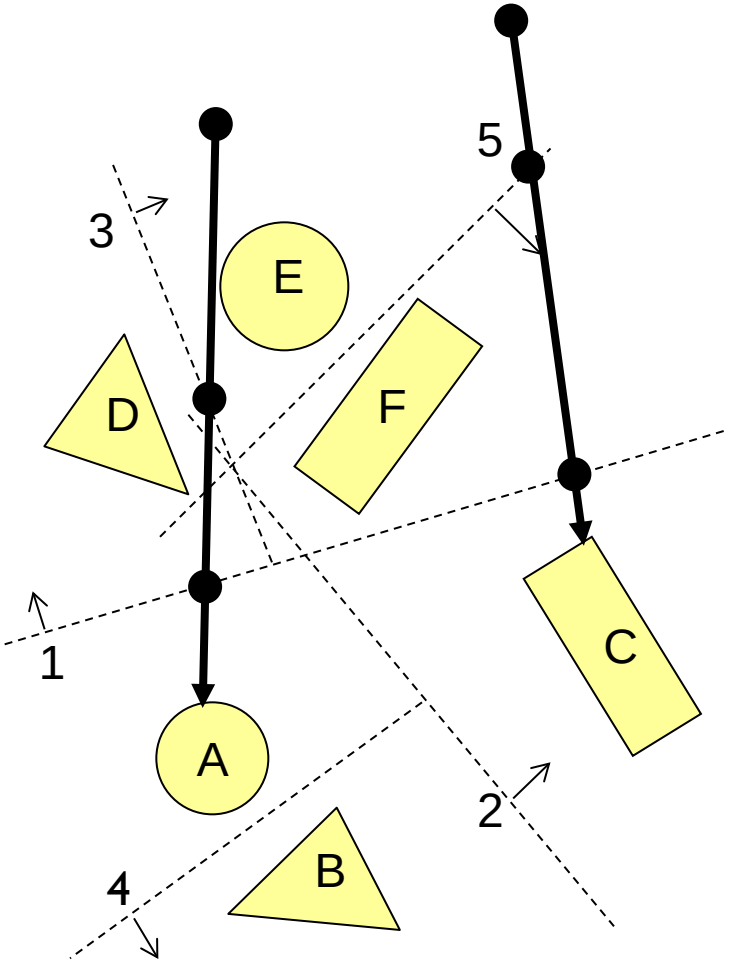
BSP Ray Casting

- Recursion on BSP tree enables simple front-to-back traversal



Ray Cast with BSP Tree





Ray Cast with BSP Tree

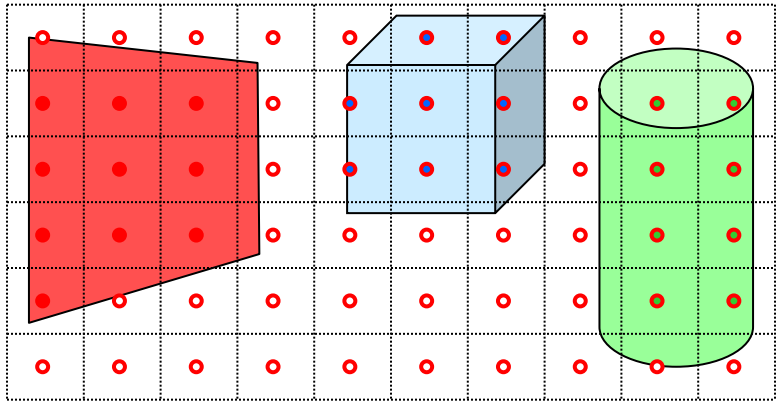
```
Object*
RayCasting( ray, BSP)
{
if (a leaf of ray origin is not known) {
        entryPoint = intersection of ray and BSP bounding box;
        if (no entry point exists) return NULL;
        currentLeaf = LocateLeaf( BSP, entryPoint);
}
// traverse through whole BSP tree
while (currentLeaf != NULL) {
        // exit-face determination
        nextExitFace = GetExitFace( currNode, ray, exitPoint);
        if (currentLeaf is not empty)
        if (ray intersects an object in currentLeaf) {
                ray = terminationLeaf = currentLeaf;
                return object;
        }
        if (currentLeaf [nextExitFace] == NULL)
                return NULL; // the ray is leaving the BSP
        if (currentLeaf [nextExitFace] is leaf)
                currentLeaf = currentLeaf[nextExitFace];
        else
                currentLeaf = LocateLeaf( currentLeaf [nextExitFace], exitPoint);
}
}
```

Ray Cast with 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 = nearest distance of the ray point to split plane of node*
        *(find closest split plane node by traversing tree)
        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)
}
```

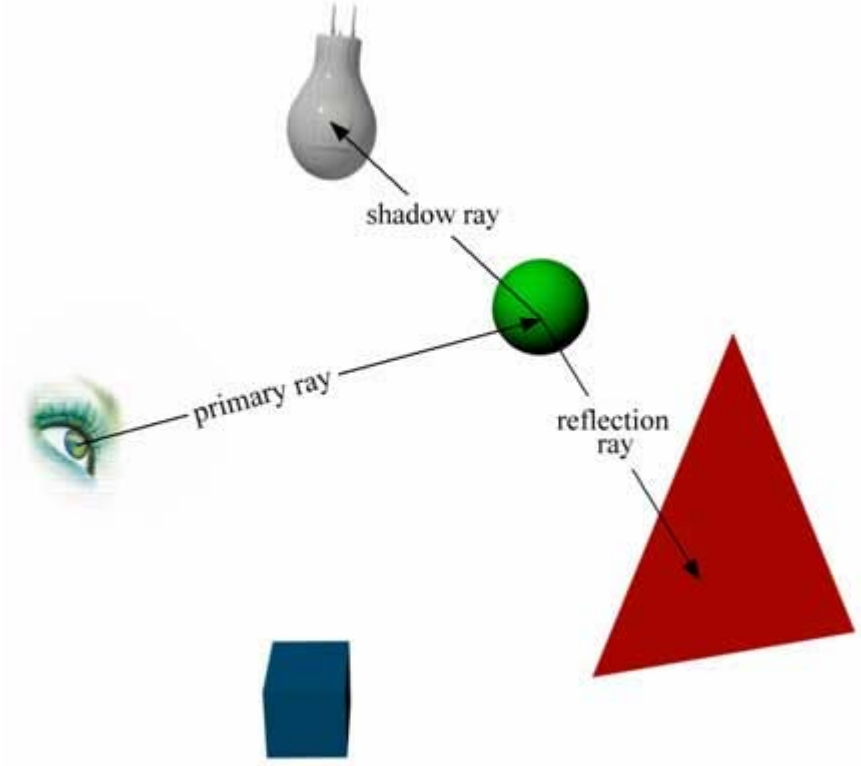
Other Accelerations

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



Simple Model for Exercise

- Use only reflections and shadow rays



Summary

- Ray casting (direct Illumination)
  - Usually use simple analytic approximations for light source emission and surface reflectance
- Recursive ray tracing (global illumination)
  - Incorporate shadows, mirror reflections, and pure refractions

Note: remember these are approximation so that it will be practical to compute

Appendix: Illumination Terminology

- Radiant power [flux] (Φ)
  - Rate at which light energy is transmitted (in Watts).
- Radiant Intensity (I)
  - Power radiated onto a unit solid angle in direction (in Watts/sr)
    - e.g.: energy distribution of a light source (inverse square law)
- Radiance (L)
  - Radiant intensity per unit projected surface area (in Watts/m^2sr)
    - e.g.: light carried by a single ray (no inverse square law)
- Irradiance (E)
  - Incident flux density on a locally planar area (in Watts/m^2)
    - e.g.: light hitting a surface along a
- Radiosity (B)
  - Exitant flux density from a locally planar area (in Watts/ m^2)