Ray tracing - intersection + shading

COMP575

Overview

- Triangle intersection
- Ray tracing overview
- Normals
- Light source
- Surface modeling

Triangle intersection

What is a triangle?
Plane with bounded region. A triangle's region is defined by its 3 vertices.

Triangle intersection

- Plane intersection
  - Does the ray even hit the plane the triangle is on?
- We need an equation for a plane...
  - Point in the plane
  - Direction away from plane (called the normal)

Triangle intersection

- Plane intersection

Ray $p = \mathbf{e} + t\mathbf{d}$ Plane $(\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$

Substitute:

$$(\mathbf{e} + t\mathbf{d} - \mathbf{a}) \cdot \mathbf{n} = 0$$

Solve for t:

$$t = \frac{(\mathbf{a} - \mathbf{e}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

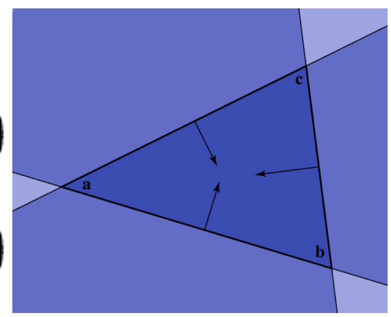
Triangle intersection

Vector break...

Triangle intersection

- Check if plane hit point is 'inside' triangle
- Use cross product

$$(\mathbf{b} - \mathbf{a}) \times (\mathbf{x} - \mathbf{a}) \cdot \mathbf{n} > 0$$
$$(\mathbf{c} - \mathbf{b}) \times (\mathbf{x} - \mathbf{b}) \cdot \mathbf{n} > 0$$
$$(\mathbf{a} - \mathbf{c}) \times (\mathbf{x} - \mathbf{c}) \cdot \mathbf{n} > 0$$



Triangle intersection

- How to get normal?
    - Only have 3 vertices
- The cross product can do this!
    - Form vectors from the vertices
    - Take cross product of vetices
    - Direction of the vectors = orientation of normal

Ray tracing overview

```
scene = loadScene()
image = createImage(x,y)
foreach x,y in image
        ray = generateRay(x, y, scene.camera)
```

```
        hit = traceRay(ray, scene)
        color = getHitColor(ray, hit, scene)

        image[x,y] = color
saveImage(image)
```

Ray tracing overview

```
traceRay(ray, scene):

closestDis = MAX
hitObject = NULL

foreach object in scene
        hit = intersect(ray, object)
        if hit.hitObject
                if hit.distance < tclose
                        closestDis = hit.distance
                        hitObject = object

return closestDis, hitObject
```

Light break...

Shading an object requires information of the object's surface. The location, the direction, the material, etc.

Normals

Direction of surface is important

Amount of light hitting same surface area changes with direction.

Normals

- Normal - direction of surface

- Vector that points away from surface
- Perpendicular to surface tangent
- Helpful if unit length

- Triangle normal vector?
- Sphere normal vector?

Normals

Sphere normal:

$$\mathbf{n} = \mathbf{p} - \mathbf{c}$$

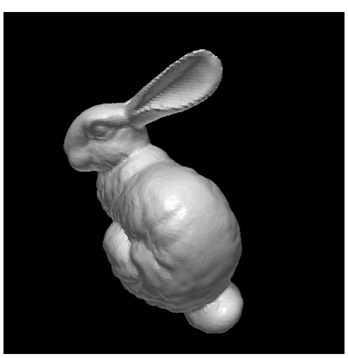
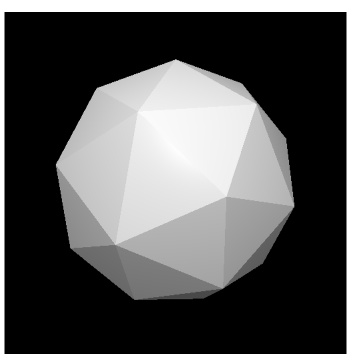
Unit normal:

$$\mathbf{n} = \frac{\mathbf{p} - \mathbf{c}}{|\mathbf{p} - \mathbf{c}|}$$

There are many other intersection algorithms for ray-triangle intersection.

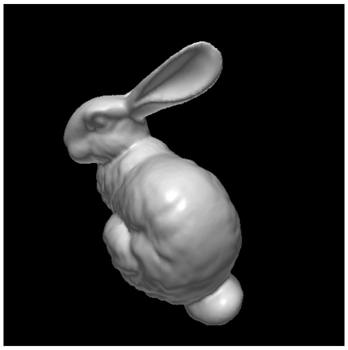
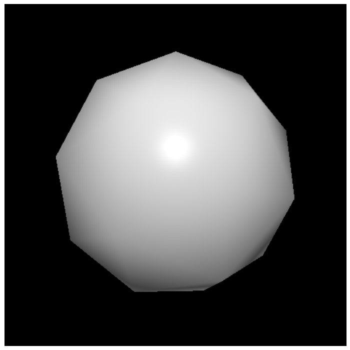
Normals

- Triangle normal:
  - Just use the plane's normal (**n**)!
  - Or extract from vertex vectors (cross product)

Normals

- Triangle vertex normals
  - Normal for each vertex
  - Blend between normals

Overview

- Triangle intersection
- Ray tracing overview
- Normals
- Light source
- Surface modeling

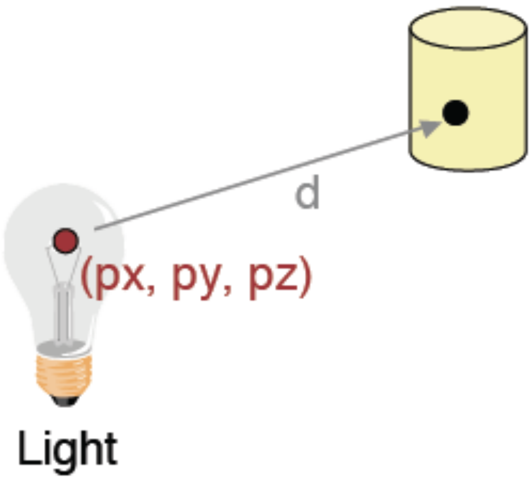
Light sources

- Point source
- Directional source
- Spotlight source
- Many other (more correct) models...

- Attenuation - larger distance results in less light

Light sources

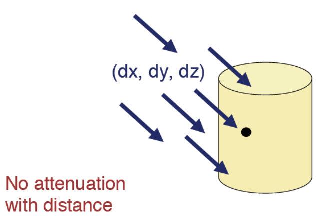
- Point source
  - Omni-directional point
  - Intensity I_0 (rgb)
  - Position (px py pz)
  - Attenuation k_c k_l k_q



$$I_L = \frac{\mathrm{I}_0}{\mathrm{k_c} + \mathrm{k_l} d + \mathrm{k_q} d^2}$$

Light sources

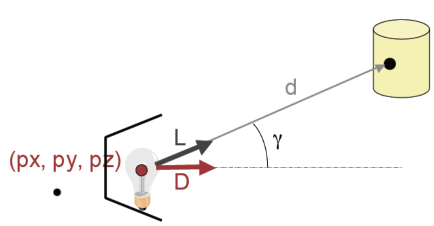
- Directional source
  - Point light at infinity
  - Intensity I_0 (rgb)
  - Direction



$$I_L = I_0$$

Light sources

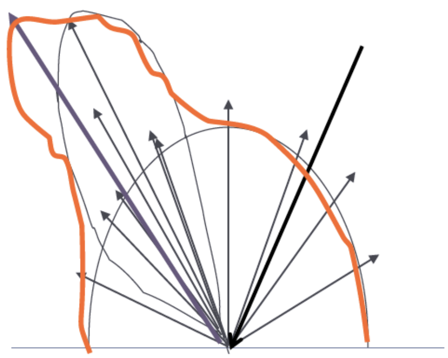
- Spotlight source
  - Point light with directionality
  - Intensity I_0 (rgb)
  - Position (px py pz)
  - Direction (dx dy dz)
  - Attenation



$$I_L = \begin{cases} \dfrac{\mathrm{I}_0}{\mathrm{k_c} + \mathrm{k_l} d + \mathrm{k_q} d^2} & \text{if inside light cone} \\ 0 & \text{otherwise} \end{cases}$$

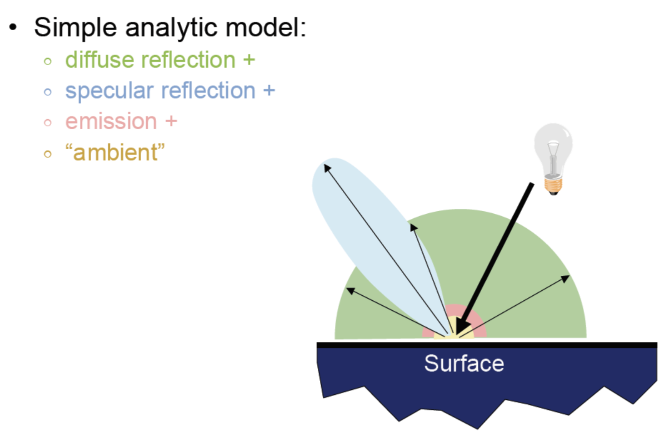
Surfaces

- Real surfaces are complicated
    - Bidirectional reflectance distribution function (BRDF)



Surfaces

- Let's use a simplified model



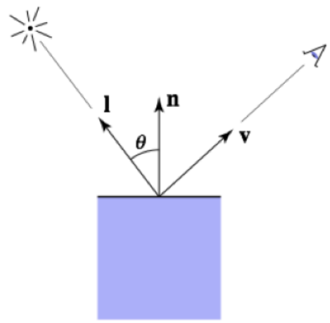
Surfaces

- Diffuse - flat reflectance material
- Specular - shiny reflectance material
- Emissive - glowing material
- Ambient - covers other complicated lighting
    - Material to material reflections
    - High order reflections reflections

Surfaces

- Useful vectors

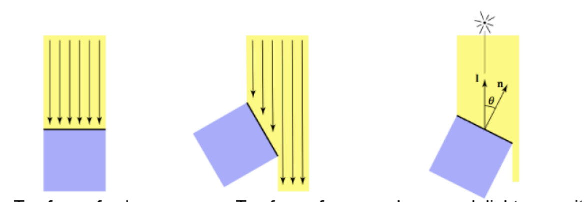
- **l** light direction (light - hit)
- **n** surface normal
- **v** view direction (eye - hit)



Surfaces

Diffuse

- Varies with light direction
- Use surface normal and light direction
- Dot product of unit vectors
- Check if negative!



Top face of cube receives a certain amount of light | Top face of 60° rotated cube intercepts half the light | In general, light per unit area is proportional to cos θ = **l** • **n**

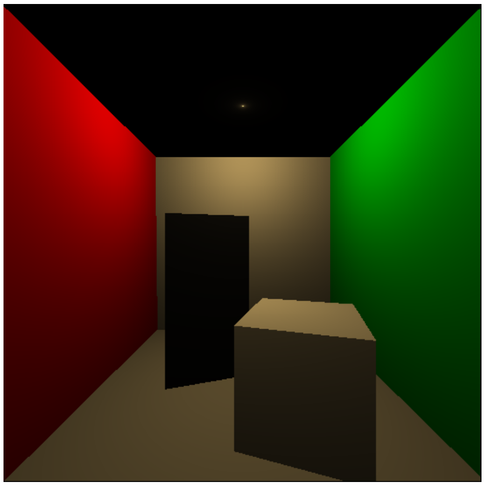
Surfaces

Diffuse

- K_d is surface's diffuse reflectance
- I_L is light intensity
- $I = K_d(\mathbf{l} \cdot \mathbf{n})I_L$

Surfaces

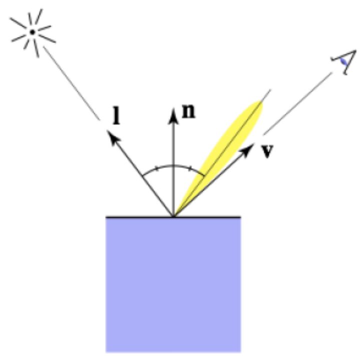
Diffuse



Surfaces

Specular

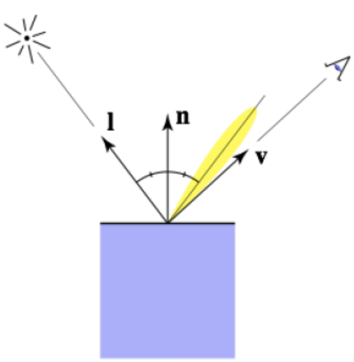
- Varies with light direction
- Varies with view direction
- Use surface normal, light reflection



Surfaces

Specular

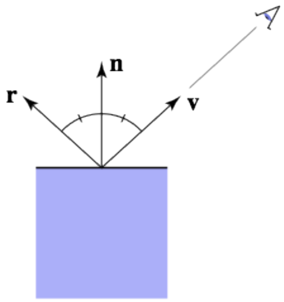
- How to get light reflection vector?



$$\mathbf{l}_r = 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n} - \mathbf{l}$$

Surfaces

- General specular reflection
    - Mirrors



$$\mathbf{r} = 2(\mathbf{v}\cdot\mathbf{n})\mathbf{n} - \mathbf{v}$$

Surfaces

Specular

- K_s is surface's specular reflectance
- I_L is light intensity
- $I = K_s(\mathbf{v}\cdot\mathbf{l_r})^p I_L$

Surfaces

Specular

- *p* is 'shiny' amount
- Higher *p* makes surface more shiny

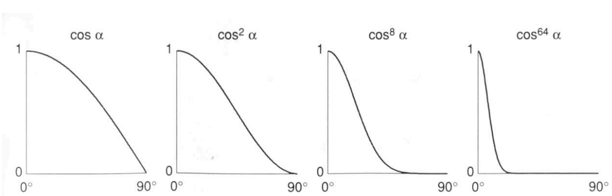
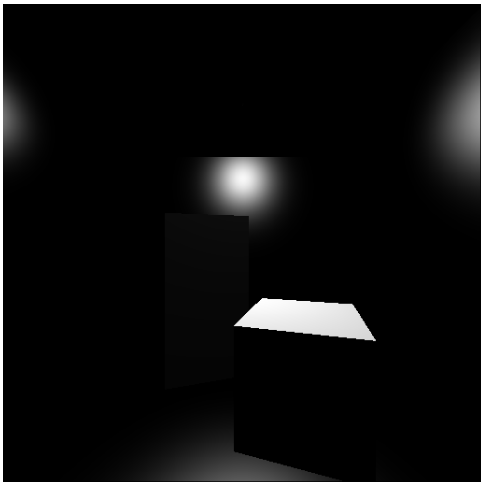


Fig. 16.9 Different values of $\cos^n \alpha$ used in the Phong illumination model.

Surfaces

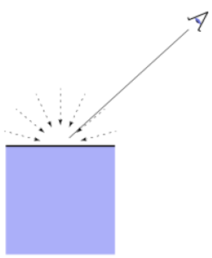
Specular



Surfaces

Ambient

- Covers missing effects
  - Surface to surface reflection
- No direction components



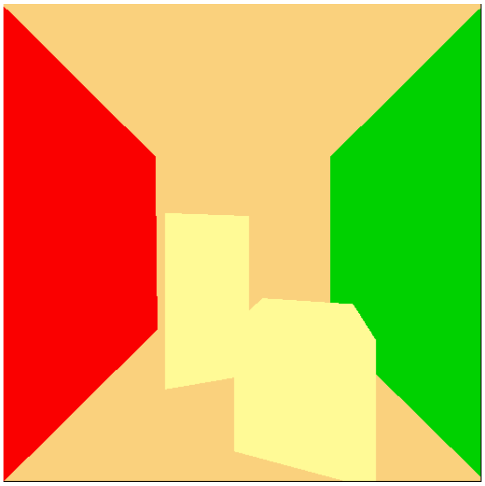
Surfaces

Ambient

- K_a is surface's ambient reflectance
- I_a is light's ambient intensity
  - Could also use diffuse/specular intensity
- $I = K_a I_a$

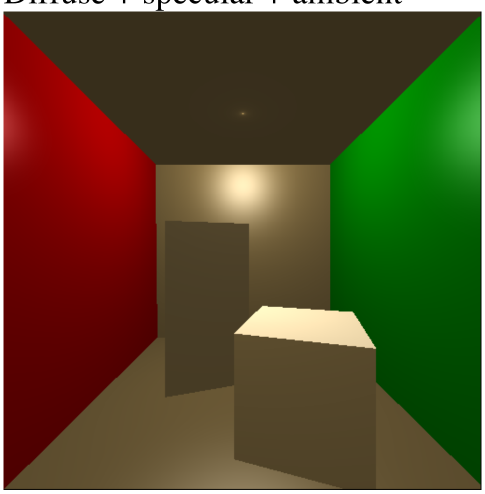
Surfaces

Ambient

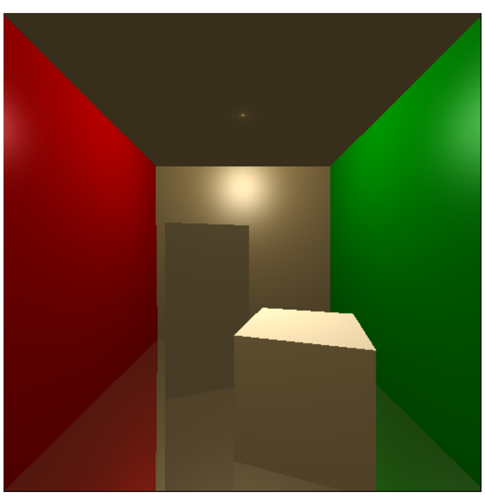


Surfaces

Diffuse + specular + ambient



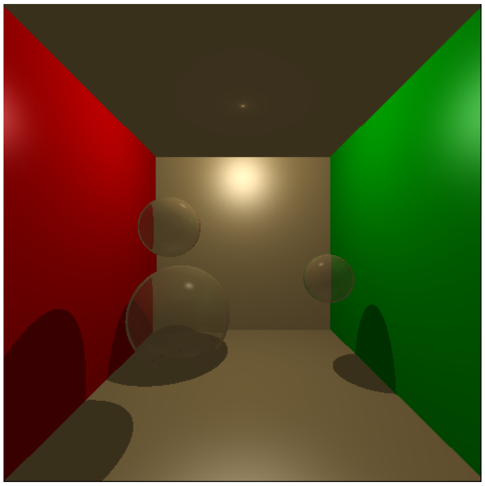
Surfaces



- Mirrors
  - Similar to specular lighting
  - Reflect eye ray (view)
  - Trace reflection ray
  - Add reflected ray color to local surface color
- Recursively reflect rays

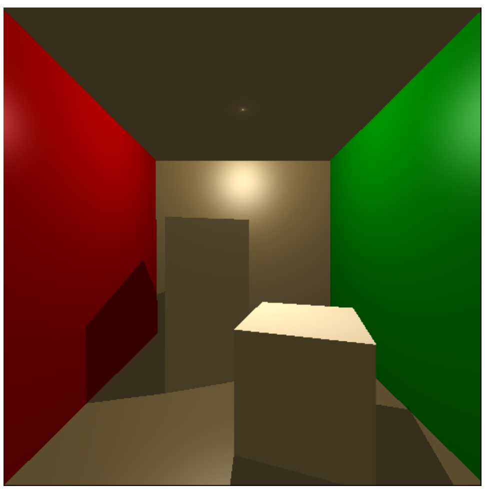
Surfaces

Surfaces



- Transparency
  - Trace ray from hit point through object
  - Add transmitted ray color to local surface color
- Rotate/bend ray based on object material
  - Snell's law
- Recursively transmit rays

Surfaces



- Shadows
  - Get distance from hit to light
  - Trace ray towards light
  - If ray distance < light distance, light is blocked
  - Diffuse and specular become 0

Ray tracing overview

```
getHitColor(ray, hit, scene):

hitColor = 0
material = scene.material[hit.object]
hitPoint = ray.paramenter(hit.distance)

foreach light in scene
        ambientColor = light.ambient * material.ambient
        diffuseColor = light.intensity * material.diffuse * (N dot L)
        specularColor = light.intensity * material.specular * (V dot Lr)^p

        inShadow = doesShadowRayHit(shadowRay, scene)

        hitColor = hitColor+ambient
```

```
        if not inShadow
                hitColor = hitColor+diffuse+specular

reflectColor = traceReflection(reflectRay, scene)
reflectAmount = material.reflectAmount

hitColor = reflectAmount*reflectColor + (1-reflectAmount)hitColor

return hitColor
```