# Announcements

- PA1 out
  - In pairs: PA0 (find partners), stay after class to find partners, or post on piazza, or contact the TAs, ...
  - capped cylinder, cone

- Staff list
  - cs4620-staff-l@cornell.edu

---

# CS4620/5620: Lecture 8

## Ray Tracing Basics

### Image so far

- With eye ray generation and scene intersection

```java
for (0 <= iy < ny) {
  for (0 <= ix < nx) {
    ray = camera.getRay(ix, iy);
    c = scene.trace(ray, 0, +inf);
    image.set(ix, iy, c);
  }
}
```

```java
Scene.trace(ray, tMin, tMax) {
  surface, t = surfs.intersect(ray, tMin, tMax);
  if (surface != null) return surface.color();
  else return black;
}
```

---

### Shading

- Compute light reflected toward camera

**Inputs:**
- eye direction
- light direction (for each of many lights)
- surface normal
- surface parameters (color, shininess, …)

**Shading independent of view direction**

- Lambertian cosine law

```
L_d = k_d \cdot I \cdot \max(0, n \cdot l)
```

---

### Diffuse reflection

- Light is scattered uniformly in all directions
  - the surface color is the same for all viewing directions
- Lambert's cosine law

- 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 \( \cos \theta = l \cdot n \)
Lambertian shading

- Produces matte appearance

[Image of Lambertian shading]

Diffuse shading

Light

- Local light
  - Position
- Directional light (e.g., sun)
  - Direction, no position

Image so far

```java
Scene.trace(Ray ray, tMin, tMax) {
    surface, t = hit(ray, tMin, tMax);
    if surface is not null {
        point = ray.evaluate(t);
        normal = surface.getNormal(point);
        return surface.shade(ray, point, normal, light);
    }
    else return backgroundColor;
}
```

Shadows

- Surface is only illuminated if nothing blocks its view of the light
- With ray tracing it’s easy to check
  - just intersect a ray with the scene!

```java
Surface.shade(ray, point, normal, light) {
    shadRay = (point, light.pos - point);
    if (shadRay not blocked) {
        v = -normalize(ray.direction);
        l = normalize(light.pos - point);
        // compute shading
    }
    return black;
}
```
Shadow rounding errors

• Sounds like it should work, but hmm....

• What's going on?

• Solution: shadow rays start a tiny distance from the surface

• Do this by moving the start point, or by limiting the t range

Multiple lights

• Just loop over lights, add contributions

• Important to fill in black shadows

• Ambient shading
  – black shadows are not really right
  – one solution: dim light at camera
  – alternative: add a constant “ambient” color to the shading...

Specular shading (Phong)

• Intensity depends on view direction
  – bright near mirror configuration
  – measure “near” by dot product of unit vectors

\[ \cos(\alpha) = \mathbf{v} \cdot \mathbf{r} \]
Specular shading (Phong)
- Intensity depends on view direction
  - bright near mirror configuration

$$L_s = k_s I_{\max}(0, \cos \alpha)^n$$
$$\cos \alpha = v \cdot r$$
$$L_s = k_s I_{\max}(0, v \cdot r)^n$$

Reflected direction
- Intensity depends on view direction
  - reflects incident light from mirror direction

$$r = 2(n \cdot l)n - l$$

Specular shading (Blinn-Phong)
- Close to mirror ⇔ half vector near normal

$$h = \text{bisector}(v, l) = \frac{v + l}{\|v + l\|}$$

$$L_s = k_s I_{\max}(0, \cos \alpha)^n$$
$$= k_s I_{\max}(0, n \cdot h)^n$$

Phong model—plots
- Increasing n narrows the lobe

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

Specular shading

Diffuse + Phong shading
Ambient shading

- Shading that does not depend on anything
  - add constant color to account for disregarded illumination and fill in black shadows

\[
L_a = k_a I_a
\]

Putting it together

- Usually include ambient, diffuse, Phong in one model

\[
L = L_a + L_d + L_s = k_a I_a + k_d I \max(0, n \cdot l) + k_s I \max(0, n \cdot h)^n
\]

- The final result is the sum over many lights

\[
L = k_a I_a + \sum_{i=1}^{N} [k_d I_i \max(0, n \cdot l_i) + k_s I_i \max(0, n \cdot h_i)]
\]

Mirror reflection

- Consider perfectly shiny surface (a mirror)
  - there isn’t a highlight
  - instead there’s a reflection of other objects
- Can render this using recursive ray tracing
  - to find out mirror reflection color, ask what color is seen from surface point in reflection direction
  - already computing reflection direction for Phong…
- “Glazed” material has mirror reflection and diffuse
  - \( L = L_a + L_d + L_m \)
  - where \( L_m \) is evaluated by tracing a new ray

\[
\mathbf{r} = \mathbf{v} + 2((\mathbf{n} \cdot \mathbf{v})\mathbf{n} - \mathbf{v}) = 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n} - \mathbf{v}
\]

Diffuse + mirror reflection (glazed)

Ray tracer architecture 101

- You want a class called Ray
  - point and direction; evaluate(t)
  - possible: tMin, tMax
- Some things can be intersected with rays
  - individual surfaces, groups of surfaces (acceleration goes here), the whole scene
  - make these all subclasses of Surface
  - limit the range of valid t values (e.g. shadow rays)
- Once you have the visible intersection, compute the color
  - may want to separate shading code from geometry
  - separate class: Material (each Surface holds a reference to one)
  - its job is to compute the color
Architectural practicalities

• Return values
  – surface intersection tends to want to return multiple values
    • t, surface or shader, normal vector, maybe surface point
  – in many programming languages (e.g. Java) this is a pain
  – typical solution: an intersection record
    • a class with fields for all these things
    • keep track of the intersection record for the closest intersection

• Efficiency
  – in Java the (or, a) key to being fast is to minimize creation of objects
  – what objects are created for every ray? try to find a place for them where you can reuse them.
  – Shadow rays can be cheaper (any intersection will do, don’t need closest)
  – but: “First Get it Right, Then Make it Fast”