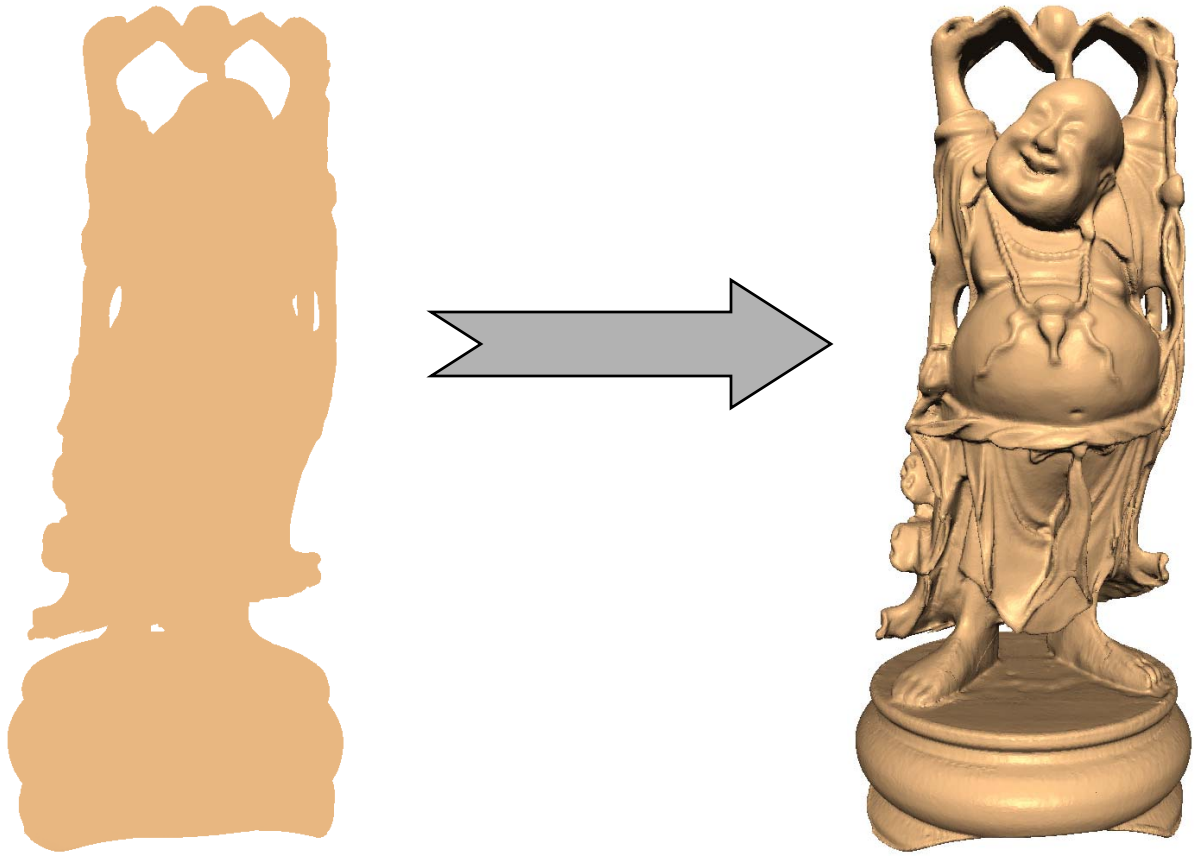
Today: Illumination & Shading



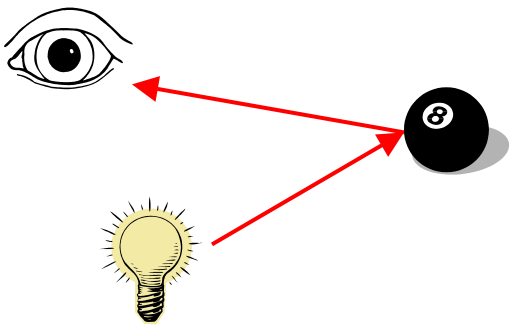
(1 million triangles drawn with 1 color per triangle)

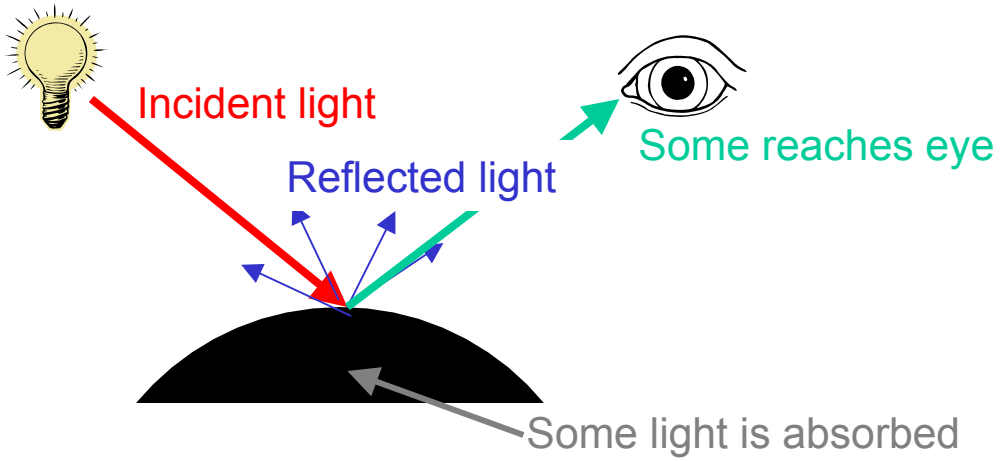
Determining an Object's Appearance

Ultimately, we're interested in modeling light transport in scene

- Light is emitted from light sources and interacts with surfaces
- on impact with an object, some is reflected and some is absorbed
- distribution of reflected light determines "finish" (matte, glossy, …)
- composition of light arriving at eye determines what we see

Let's focus on the local interaction of light with single surface point





Modeling Light Sources

In general, light sources have a very complex structure

- incandescent light bulbs, the sun, CRT monitors, …

To simplify things, we'll focus on point light sources for now

- light source is a single infinitesimal point
- emits light equally in all directions (isotropic illumination)
- outgoing light is set of rays originating at light point

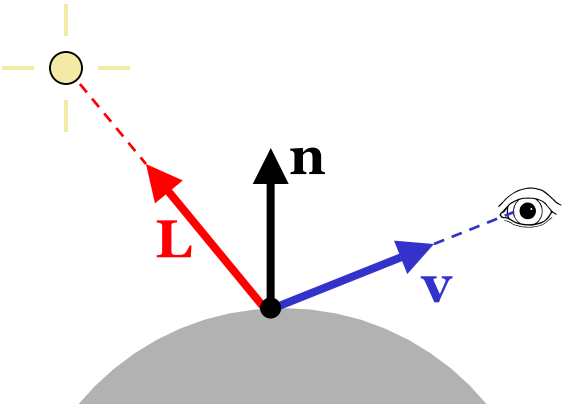
Creating lights in OpenGL

- glEnable(GL_LIGHTING) — turn on lighting of objects
- glEnable(GL_LIGHT0) — turn on specific light
- glLight(…) — specify position, emitted light intensity, …

Basic Local Illumination Model

We're only interested in light that finally arrives at view point

- a function of the light & viewing positions
- and local surface reflectance



Characterize light using RGB triples

- can operate on each channel separately

Given a point, compute intensity of reflected light

Diffuse Reflection

This is the simplest kind of reflection

- also called Lambertian reflection
- models dull, matte surfaces — materials like chalk

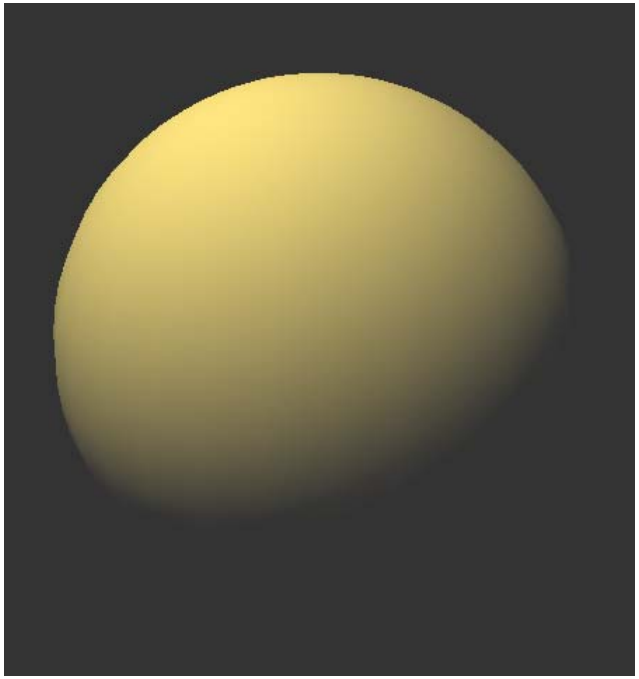
Ideal diffuse reflection

- scatters incoming light equally in all directions
- identical appearance from all viewing directions
- reflected intensity depends only on direction of light source

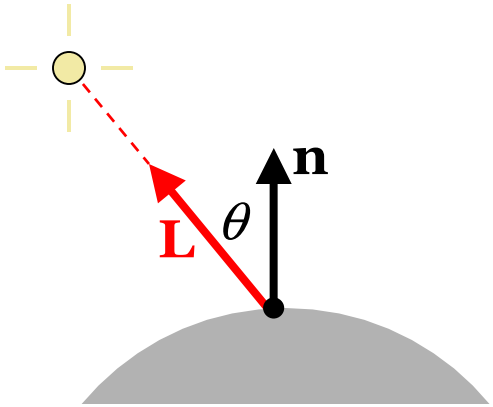
Light is reflected according to Lambert's Law

Lambert's Law for Diffuse Reflection

Purely diffuse object



$$I_d = I_L k_d \max(\cos\theta, 0)$$
$$= I_L k_d \max(\mathbf{n} \cdot \mathbf{L}, 0)$$



I_d : resulting intensity (diffuse)

I_L : light source intensity

k_d : (diffuse) surface reflectance coefficient

$k_d \in [0,1]$

θ : angle between normal & light direction

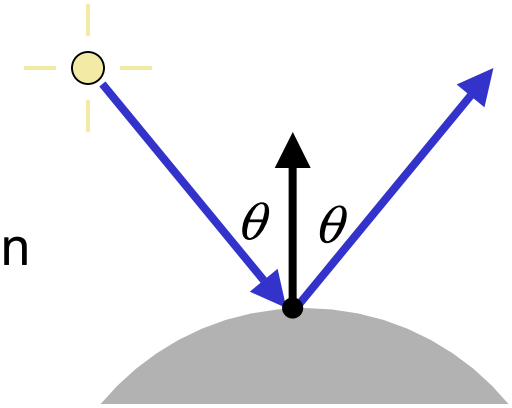
Specular Reflection

Diffuse reflection is nice, but many surfaces are shiny

- their appearance changes as the viewpoint moves
- they have glossy specular highlights (or specularities)
- because they reflect light coherently, in a preferred direction

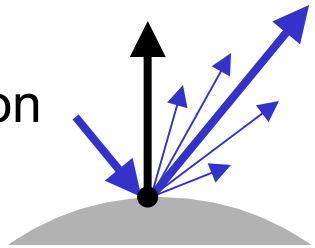
A mirror is a perfect specular reflector

- incoming ray reflected about normal direction
- nothing reflected in any other direction



Most surfaces are imperfect specular reflectors

- reflect rays in cone about perfect reflection direction

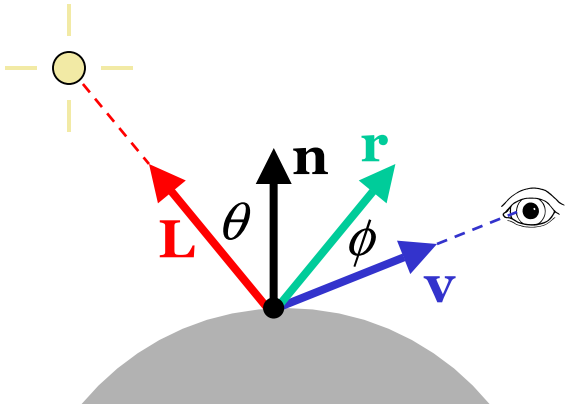


Phong Specular Illumination Model

$$I_s = I_L k_s \max(\cos\phi, 0)^n$$
$$= I_L k_s \max(\mathbf{r}\cdot\mathbf{v}, 0)^n$$

One particular specular reflection model

- quite common in practice
- it is purely empirical
- there's *no physical basis* for it



I_s : resulting intensity (specular)

I_L : light source intensity

k_s : (specular) surface reflectance coefficient

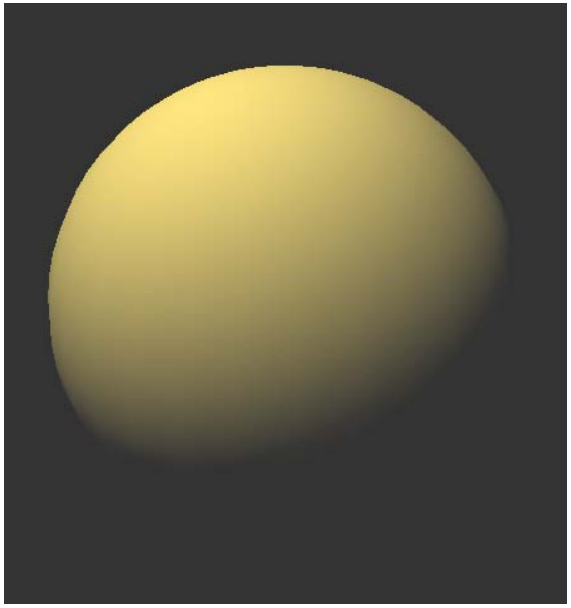
$k_s \in [0,1]$

ϕ : angle between viewing & reflection direction

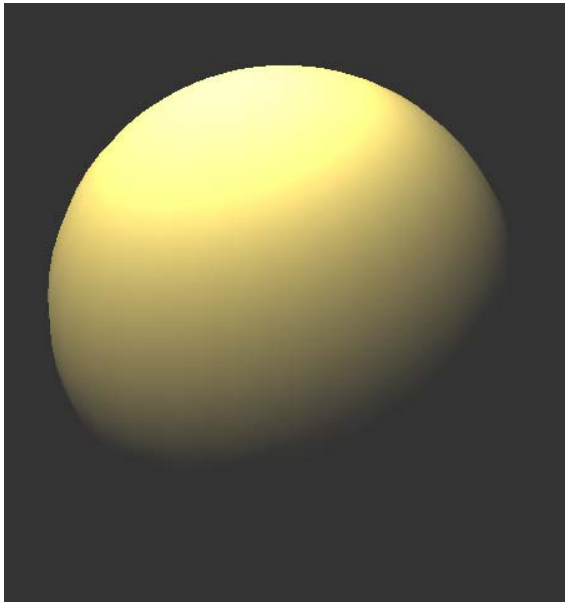
n : "shininess" factor

Examples of Phong Specular Model

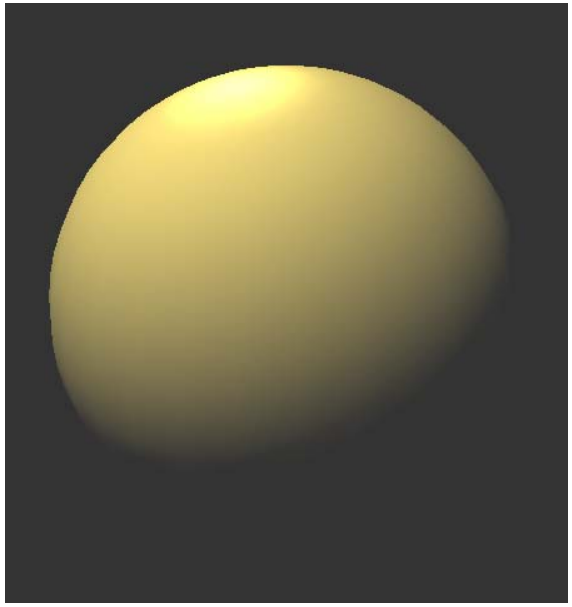
Diffuse only



Diffuse + Specular (shininess 5)



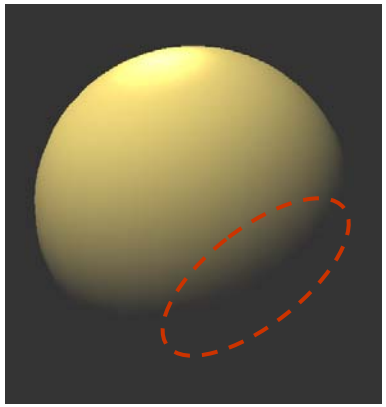
Diffuse + Specular (shininess 50)



The Ambient Glow

So far, areas not directly illuminated by any light appear black

- this tends to look rather unnatural
- in the real world, there's lots of ambient light



To compensate, we invent new light source

- assume there is a constant ambient "glow"
- this ambient glow is *purely fictitious*

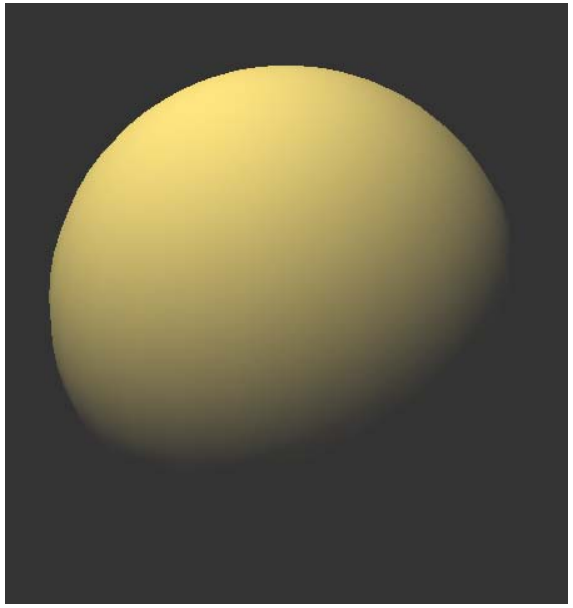
Just add in another term to our illumination equation

$$I = I_d + I_s + I_a k_a$$

I_a : ambient light intensity

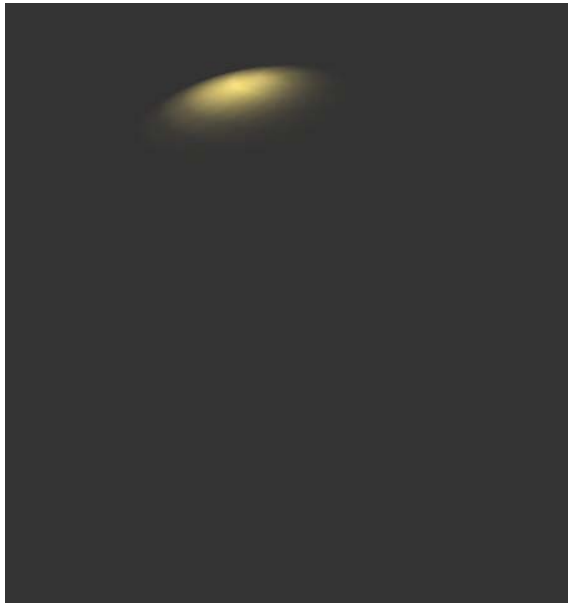
k_a : (ambient) surface reflectance coefficient

Our Three Basic Components of Illumination



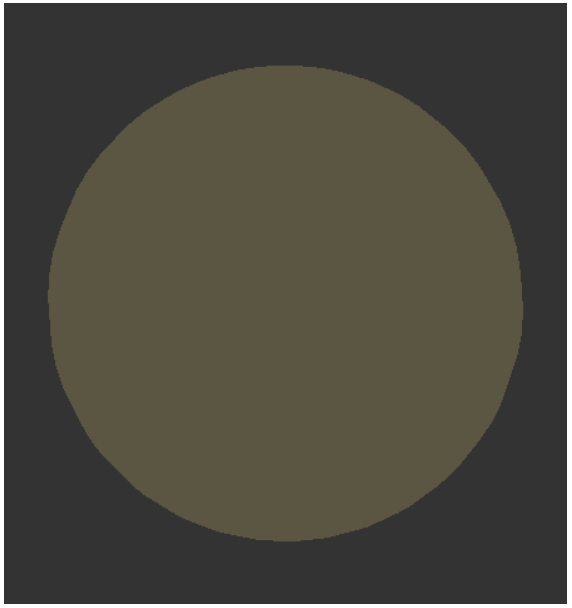
Diffuse

$I = I_d$



Specular

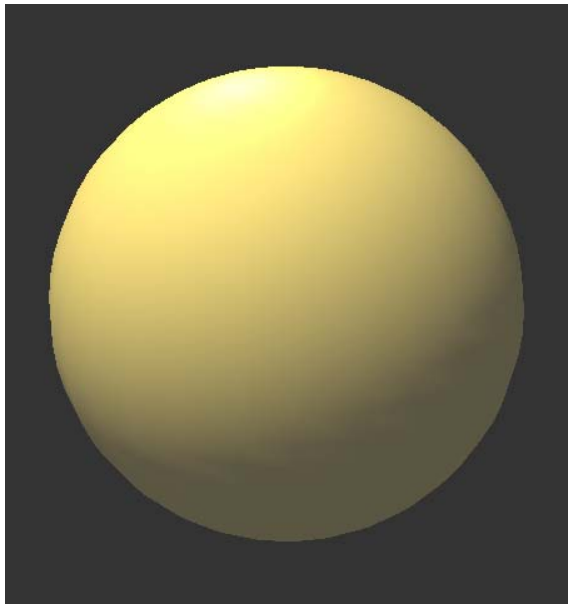
$I = I_s$



Ambient

$I = I_a k_a$

Combined for the Final Result



$$I = I_d + I_s + I_a k_a$$

Recall How to Color Polygons

Hard-coded colors on surface of model

- maybe from pre-computed illumination (e.g., radiosity)
- explicitly specify 1 color per face/vertex

Flat Shaded:

```
glBegin(GL_TRIANGLES);
  for(int j=0; j<n; j++)
  {
    glColor3fv(c);
    glVertex3fv(v1);
    glVertex3fv(v2);
    glVertex3fv(v3);
  }
glEnd();
```

Smooth Shaded:

```
glBegin(GL_TRIANGLES);
  for(int j=0; j<n; j++)
  {
    glColor3fv(c1);
    glVertex3fv(v1);
    glColor3fv(c2);
    glVertex3fv(v2);
    glColor3fv(c3);
    glVertex3fv(v3);
  }
glEnd();
```

Drawing Polygons with Lighting

We usually want OpenGL to infer colors via illumination model

- specify 1 normal per face/vertex

Flat Shaded:

```
glBegin(GL_TRIANGLES);
  for(int j=0; j<n; j++)
  {
    glNormal3fv(n);
    glVertex3fv(v1);
    glVertex3fv(v2);
    glVertex3fv(v3);
  }
glEnd();
```

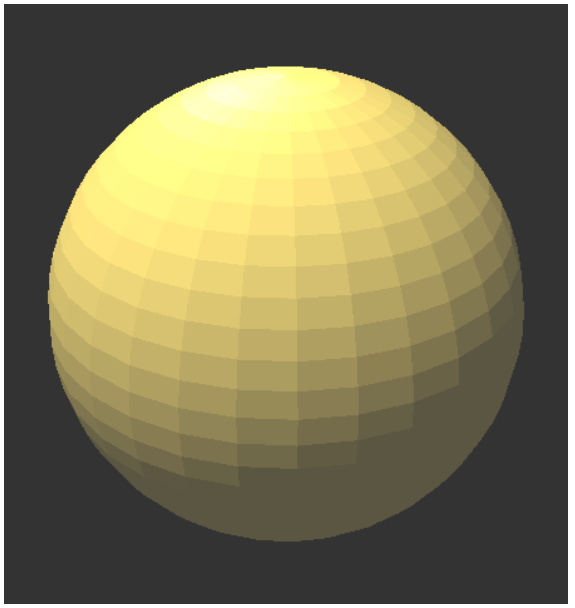
Smooth Shaded:

```
glBegin(GL_TRIANGLES);
  for(int j=0; j<n; j++)
  {
    glNormal3fv(n1);
    glVertex3fv(v1);
    glNormal3fv(n2);
    glVertex3fv(v2);
    glNormal3fv(n3);
    glVertex3fv(v3);
  }
glEnd();
```

Shading Polygons: Flat Shading

Illumination equations are evaluated at surface locations

- so where do we apply them?



We could just do it once per polygon

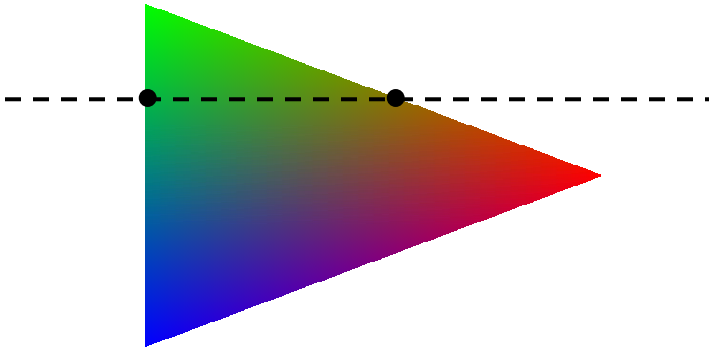
- fill every pixel covered by polygon with the resulting color

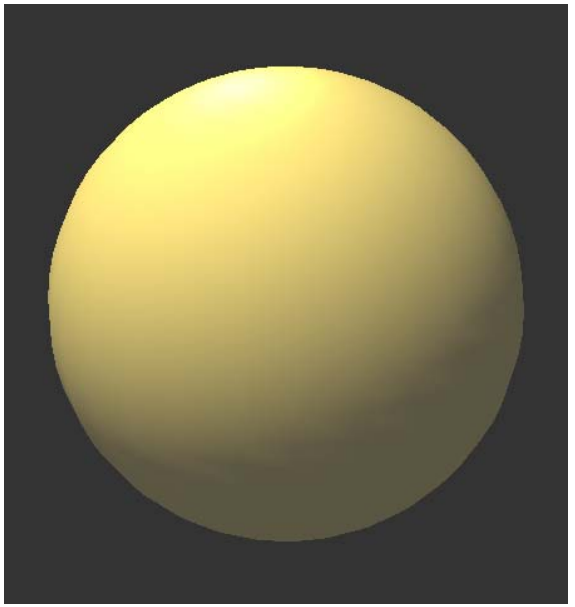
OpenGL — glShadeModel(GL_FLAT)

Shading Polygons: Gouraud Shading

Alternatively, we could evaluate at every vertex

- linearly interpolate color along edges
- linearly interpolate along scan lines
  - *interpolation in screen space varies with viewpoint*





Misses details that don't fall on vertex

- specular highlights, for instance

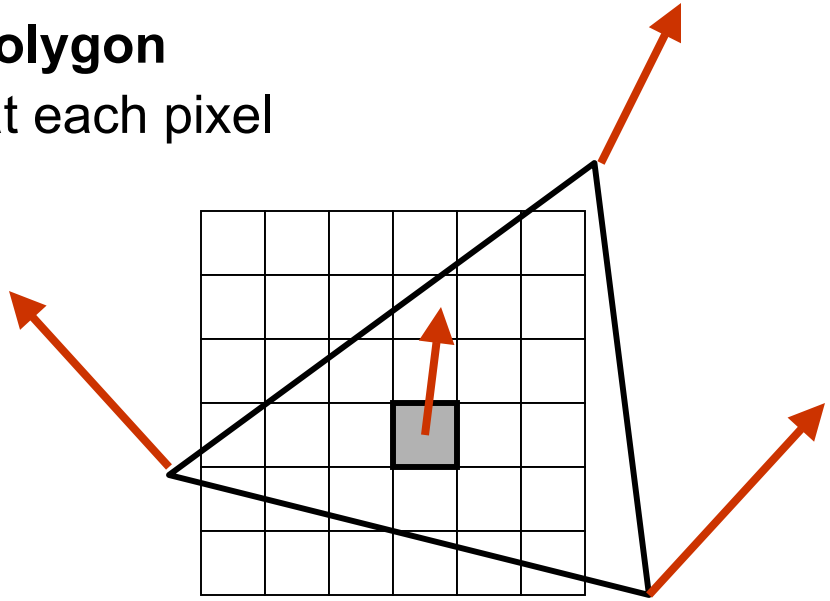
OpenGL — glShadeModel(GL_SMOOTH)

Shading Polygons: Phong Shading

Don’t just interpolate colors over polygons

Interpolate surface normal over polygon

- evaluate illumination equation at each pixel



OpenGL — not supported

Defining Materials in OpenGL

Just like everything else, there is a current material

- specifies the reflectances of the objects being drawn
- reflectances (e.g., k_d) are RGB triples

Set current values with glMaterial(…)

```
GLfloat tan[] = {0.8, 0.7, 0.3, 1.0};
GLfloat tan2[] = {0.4, 0.35, 0.15, 1.0};

glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, tan);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, tan);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, tan2);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 50.0);
```

Defining Lights in OpenGL

A fixed set of lights are available (at least 8)

- turn them on with glEnable(GL_LIGHTx)
- set their values with glLight(…)

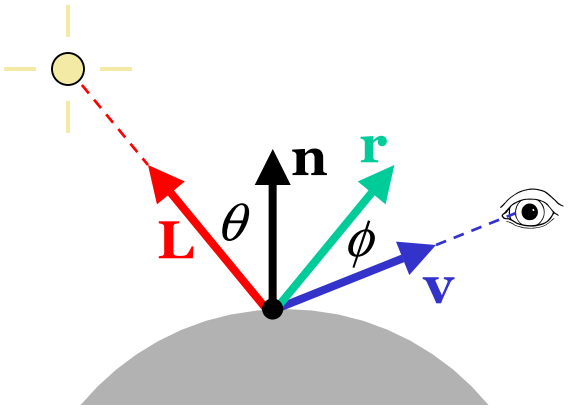
```
GLfloat white[] = {1.0, 1.0, 1.0, 1.0}
GLfloat p[] = {-2.0, -3.0, 10.0, 1.0}; // w=0 for directional light

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

glLightfv(GL_LIGHT0, GL_POSITION, p);
glLightfv(GL_LIGHT0, GL_DIFFUSE, white);
glLightfv(GL_LIGHT0, GL_SPECULAR, white); // can be different

glEnable(GL_NORMALIZE); // guarantee unit normals
```

Summarizing the Shading Model

We describe local appearance with illumination equations

- consists of a sum of set of components — light is additive
- treat each wavelength independently
- currently: diffuse, specular, and ambient terms

$$I = I_L k_d \max(\cos\theta, 0) + I_L k_s \max(\cos\phi, 0)^n + I_a k_a$$



Must shade every pixel covered by polygon

- flat shading: constant color
- Gouraud shading: interpolate corner colors
- Phong shading: interpolate corner normals

What Have We Ignored?

Some local phenomena

- shadows — every point is illuminated by every light source
- attenuation — intensity falls off with square of distance to light
- transparent objects — light can be transmitted through surface

Global illumination

- reflections of objects in other objects
- indirect diffuse light — ambient term is just a hack

Realistic surface detail

- can make an orange sphere
- but it doesn’t have the texture of the real fruit

Realistic light sources

Next Time: Viewing Models & Perspective

We've developed these tools for transforming geometry

- fundamental rotate, translate, scale transforms
- compose them by matrix multiplication

Now we'll apply them to our next big problem

- how do we project 3-D to 2-D?