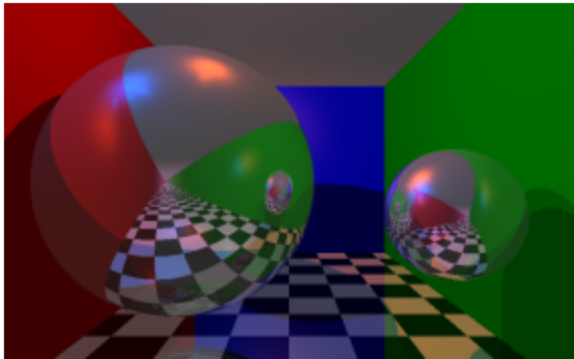
CSI 4105 Computer Graphics

Spring 2017

Lecture 8: Illumination (Part II, Shading)

Seon Joo Kim
Yonsei University



Objectives

- Discuss shading models
  - Flat, Gouraud, Phong
- Discuss how to compute normals per vertex
- Discuss setting up the light(s) in OpenGL
  - Up to eight lights
  - The lights must be specified in the “world” frame

Shading Approaches
Flat, Gouraud, Phong

The Phong Illumination Model

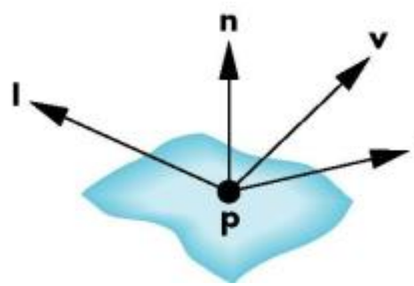
Recall that our illumination model has three terms

- **Ambient Light**
  - fixed value
- **Diffuse Reflection**
  - depends on (light direction **L** and surface normal **N**)
- **Specular Reflection**
  - (depends reflected light direction (**R** – computed from **L** and **N**) and viewer **V** direction)

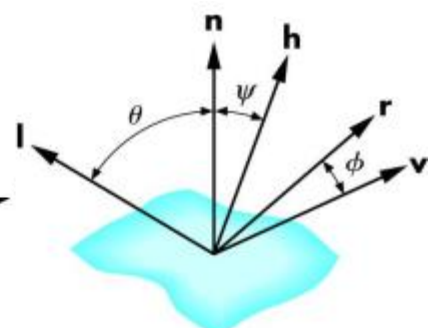
$$I = \underbrace{\boxed{k_a I_a}}_{\text{ambient}} + \underbrace{\boxed{k_d I_l (N \cdot L)}}_{\text{diffuse}} + \underbrace{\boxed{k_s I_l (V \cdot R)^{n_s}}}_{\text{specular}}$$

Modified Phong Illumination Model

- OpenGL uses what is known as a "Modified Phong Illumination Model"
  - The idea is to make the specular computation more efficient
  - Instead of computing the ideal reflector - *r*
  - They estimate a "half-way vector" called "H"
    - This vector is half way between the viewer, *v*, and light direction, *l*
    - Think of this as an "approximation" of *r*
- This model was proposed by Jim Blinn (also called the Blinn-Phong model)



Phong Model



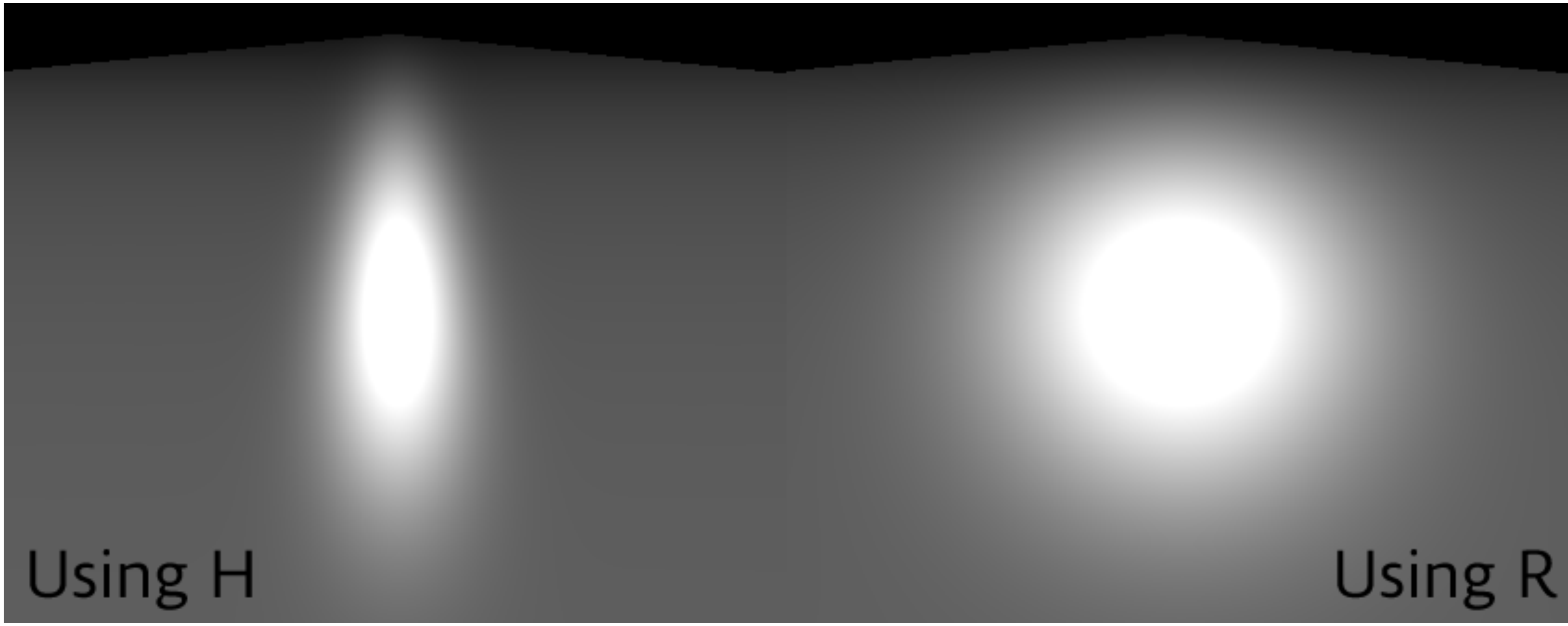
Modified Phong

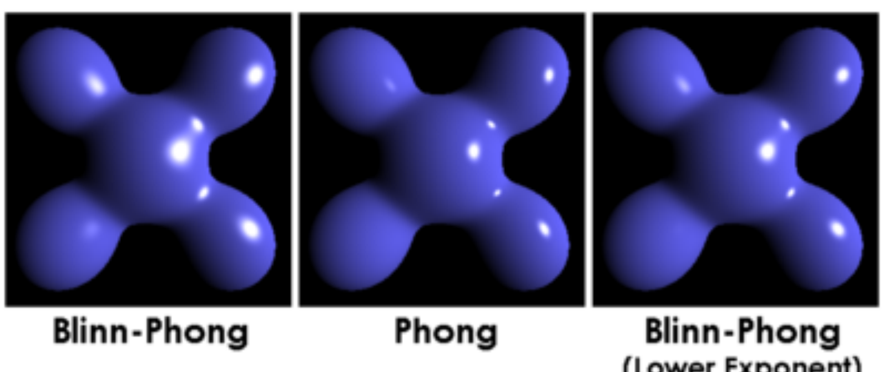
$$\mathbf{h} = (\mathbf{l} + \mathbf{v}) / |\mathbf{l} + \mathbf{v}|$$

$$I = k_a I_a + k_d I_l (N \cdot L) + \boxed{k_s I_l (V \cdot H)^{n_s}}$$

Jim Blinn
PhD Utah, 1978

Example of Modified Phong vs. Phong



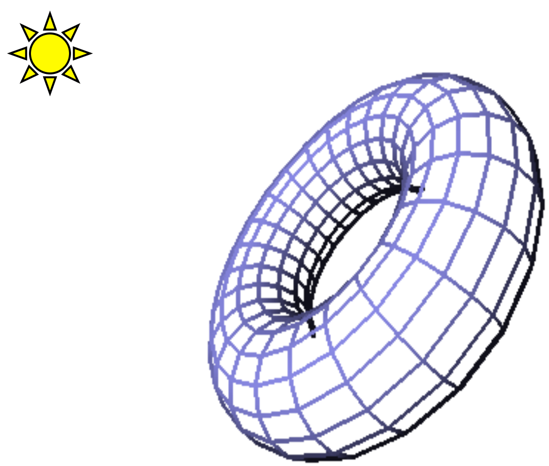


http://en.wikipedia.org/wiki/Blinn–Phong_shading_model

Shading the Polygons . . .

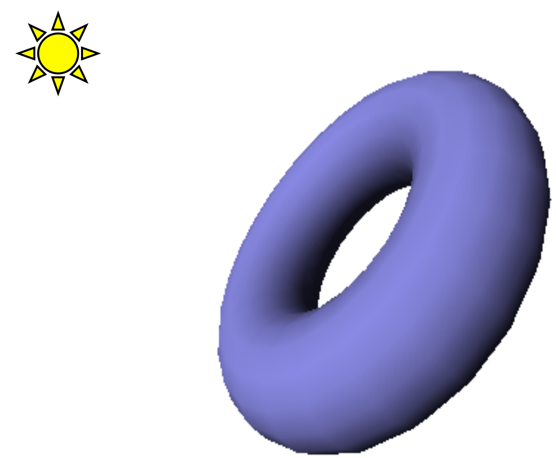
- Recall that interactive computer graphics hardware can only draw polygons
- We need to use the lighting models to “shade” these polygons

Light Source



Polygon Model

Light Source



Polygon model shaded

Shading Approaches

Three main surface rendering approaches are used to render the polygons:

(1) Flat Shading

(2) Gouraud Shading

(3) Phong Shading

Interpolative Shading Techniques

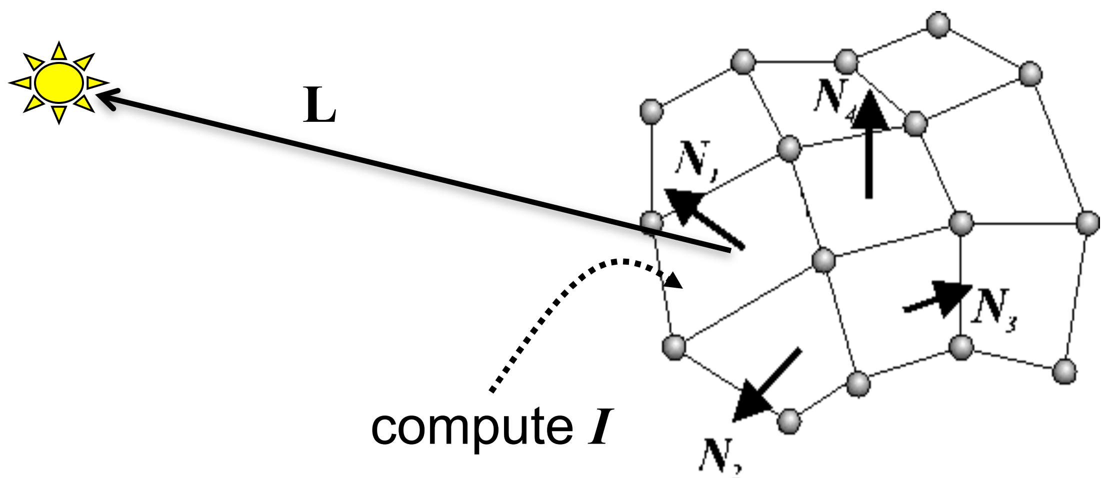
These are ordered in terms of speed and quality. “Flat” is the fastest, but lowest quality, “Phong” is the slowest but highest quality.

Method 1: Flat Shading

- Simplest approach. Perform one computation of the illumination model to obtain an intensity I.

- Assign a *single* intensity to entire polygon/triangle.

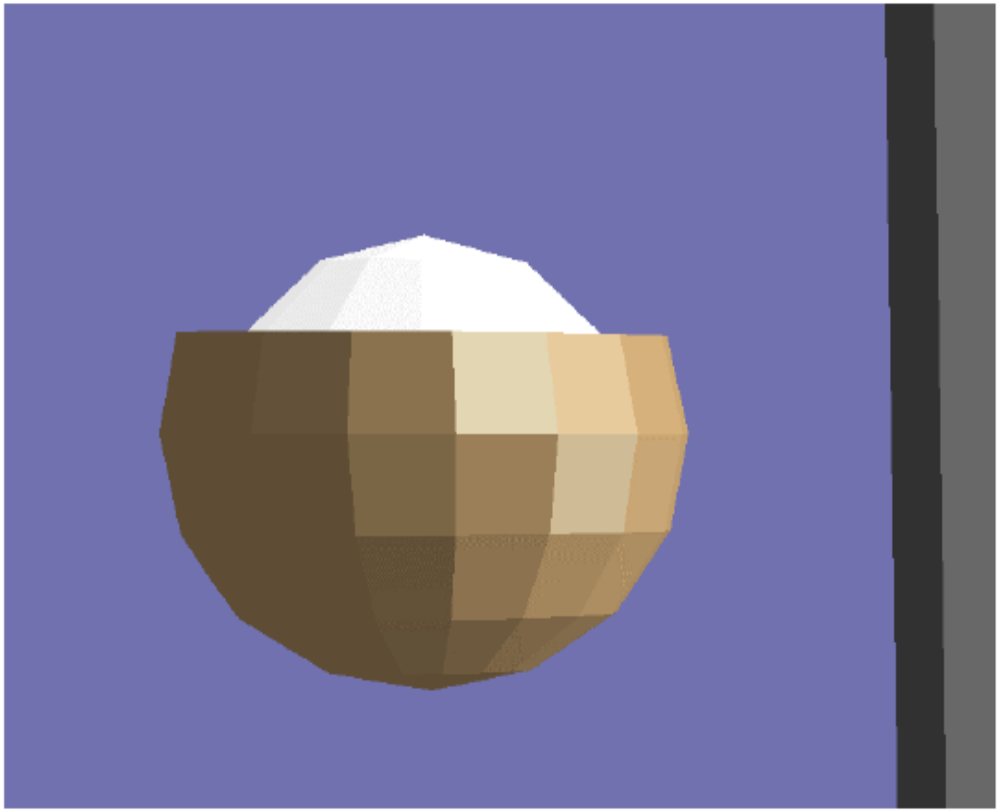
- We call this "Flat Shading"



all points in the polygon will be assigned intensity "***I***"

$$I = k_a I_a + k_d I_l (N \cdot L) + k_s I_l (V \cdot R)^{n_s}$$

Flat Shading Example



The surface is not smooth when displayed. Very low realism.

Note: **Sometimes we desire this effect.**

Lets the user see the underlying model clearly.

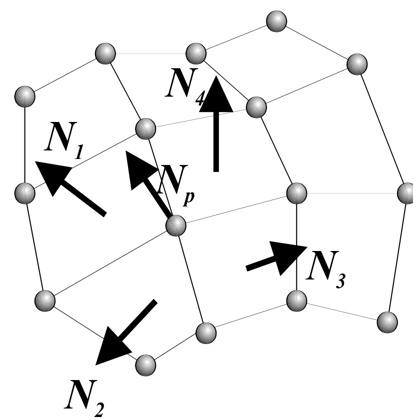
Interpolative Shading

Interpolative shading

To recover (approximately) the visual appearance of curved surface which is now represented by flat polygons.

Assumptions/Input Data:

1) An approximate normal to the original smooth surface is *given (or is computed) at* ***each vertex***



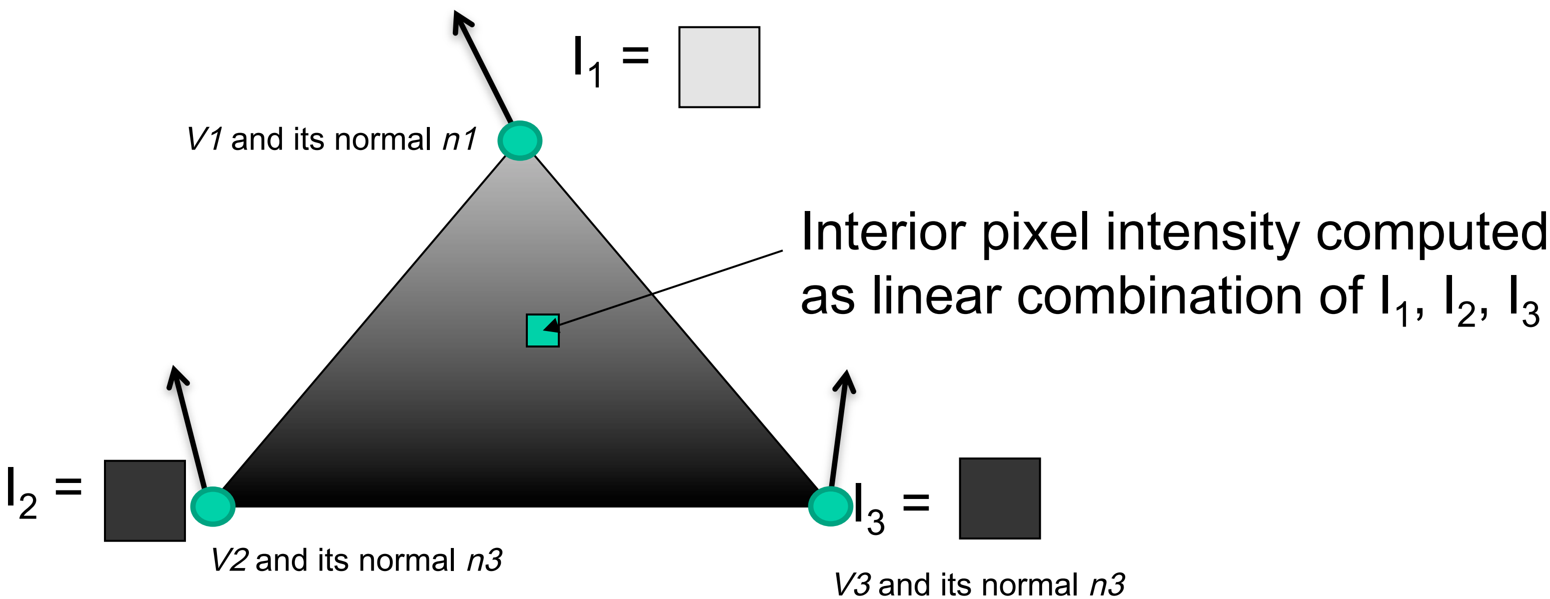
$$N_p=(N_1+N_2+N_3+N_4)/4$$

2) The shading of a particular pixel can be obtained by a bilinear interpolation of the appropriate quantities from adjacent vertices.

Method 2: Gouraud Shading

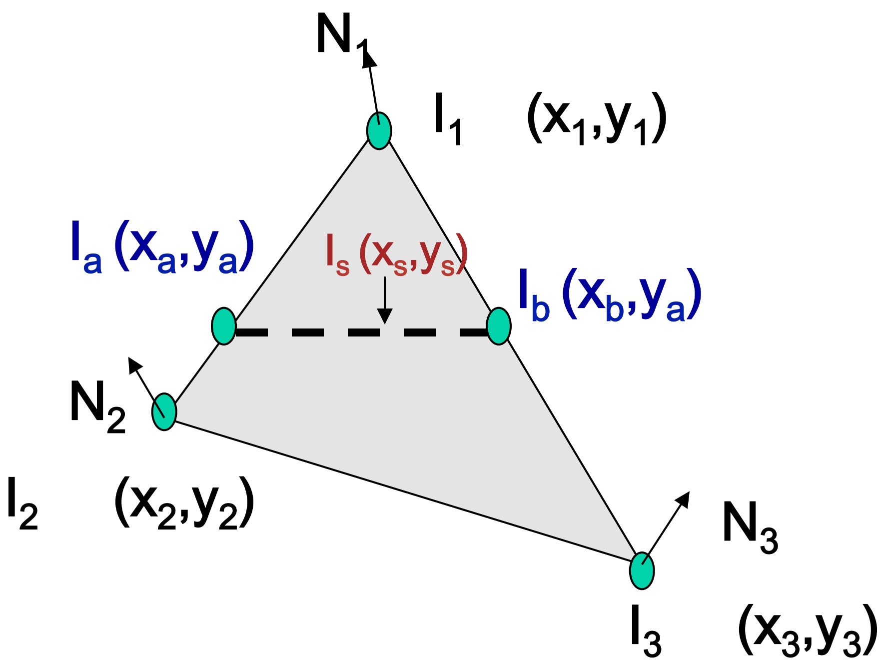
Henri Gouraud
PhD 1971
Utah

- (1) Calculate the intensity at each vertex using the illumination model
- (2) Intensities of interior pixels are determined by linearly interpolating the vertices' intensities
- Gives a smooth intensity change across the polygon



Gouraud Shading

$$I = k_a I_a + k_d I_l (N \cdot L) + k_s I_l (V \cdot R)^{n_s}$$



Step 1:
For each triangle vertex:
compute intensities I_1, I_2, I_3

Step 2:
Interpolate intensities at each pixel location

Scan line edge intensity

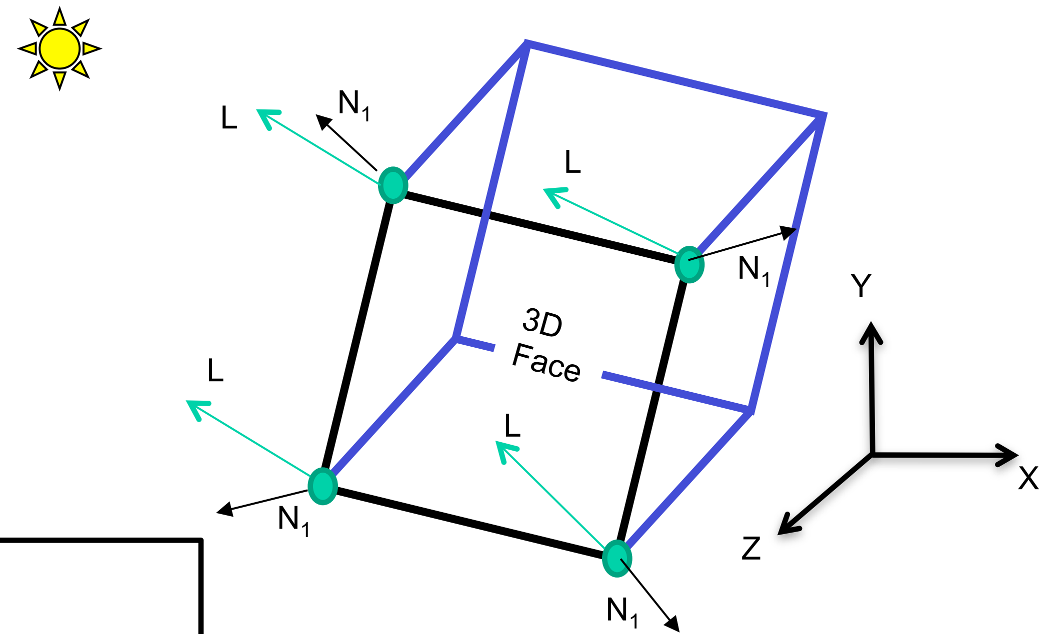
$I_a = [I_1 (y_s - y_2) + I_2(y_1 - y_s)] / (y_1 - y_2)$

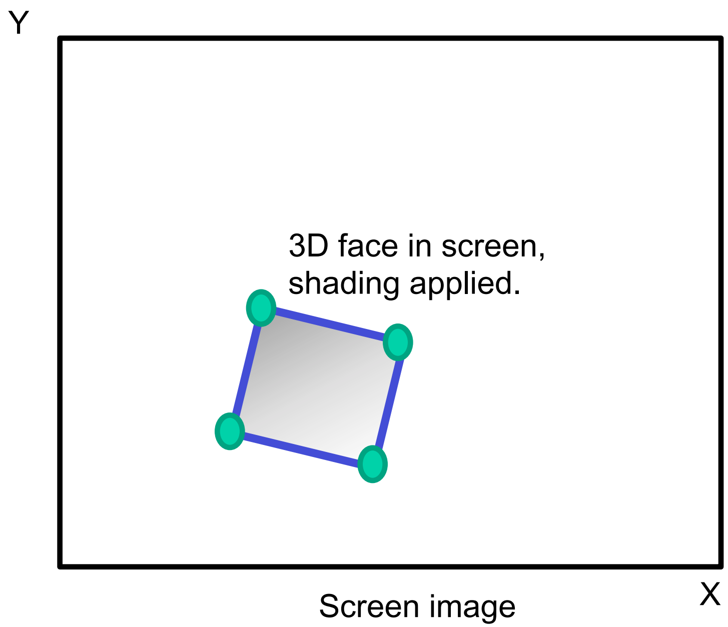
$I_b = [I_1 (y_s - y_3) + I_3(y_1 - y_s)] / (y_1 - y_3)$

Interior point intensity

$I_s = [I_a (x_b - x_s) + I_b(x_s - x_a)] / (x_b - x_a)$

Aside: When does OpenGL perform shading?



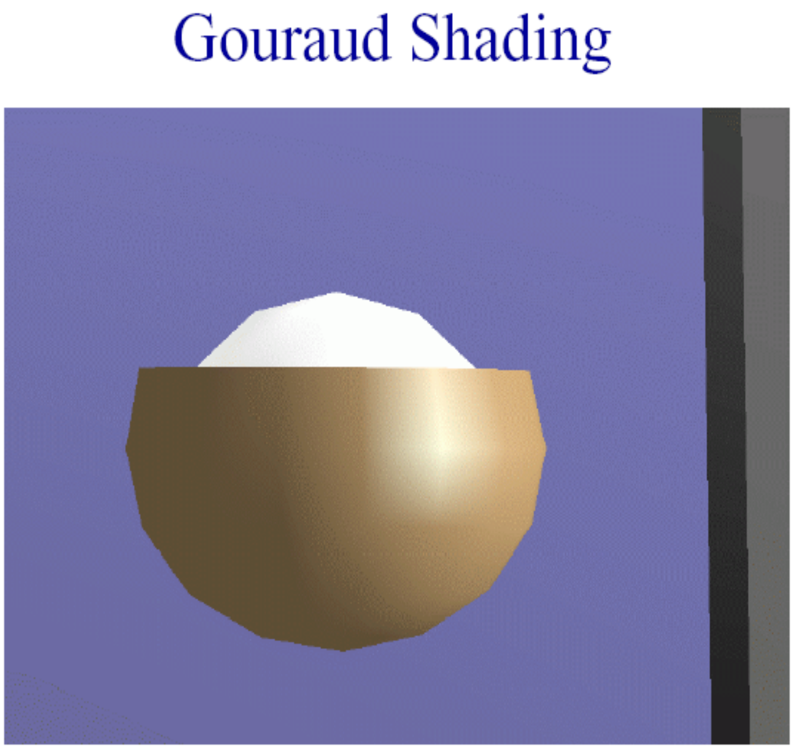


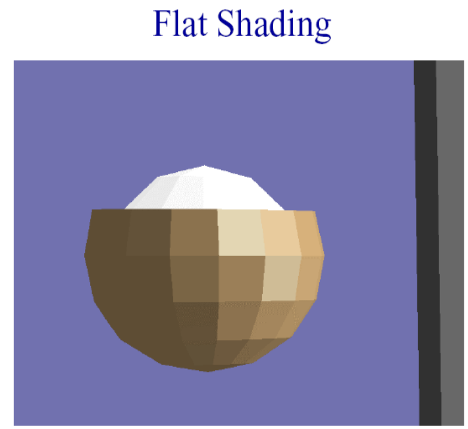
Shading is performed **after** the 3D face has been projected and mapped to the view-port (pixels).

We call this “rasterization” – the geometry is being converted into pixels (or the raster).

So, after projection, OpenGL uses the normals per vertex to shade.

Gouraud Shading





Compare w/ flat shading

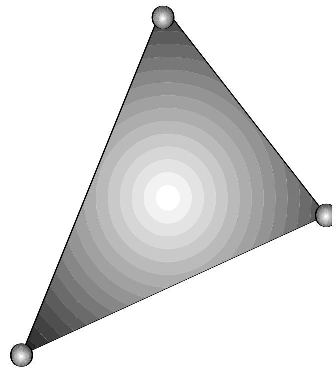
The surface looks much smoother.

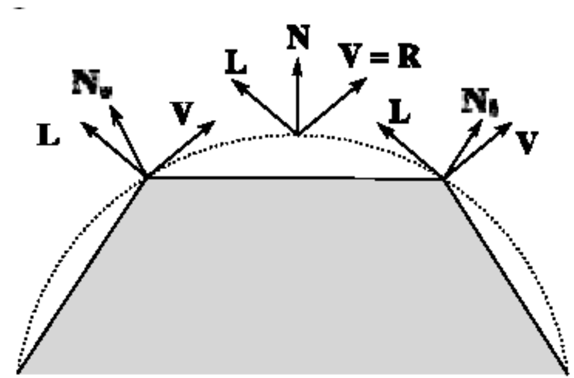
*This of course “costs” more to compute than flat shading.. .

-Have to compute intensity at *each* vertex, need to interpolate these values.

Flaws in Gouraud Shading . . .

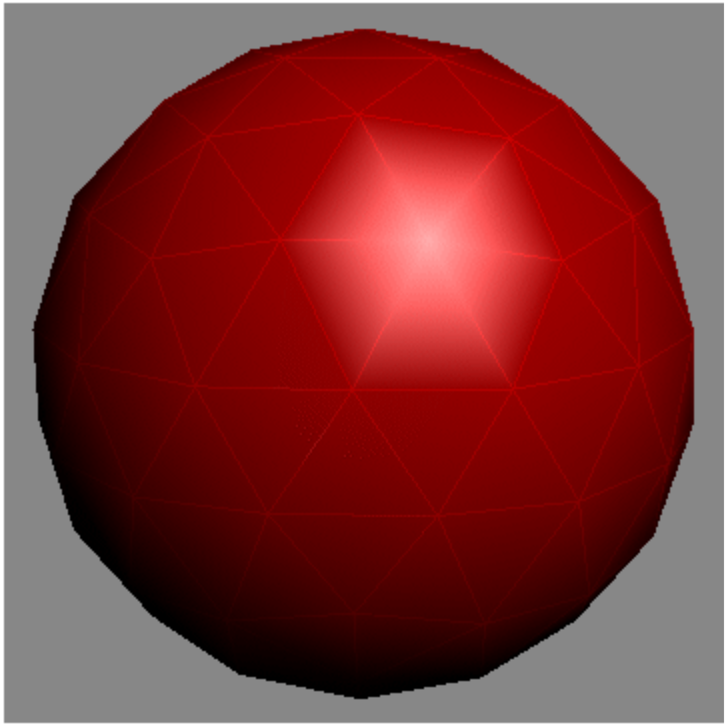
Highlight anomalies



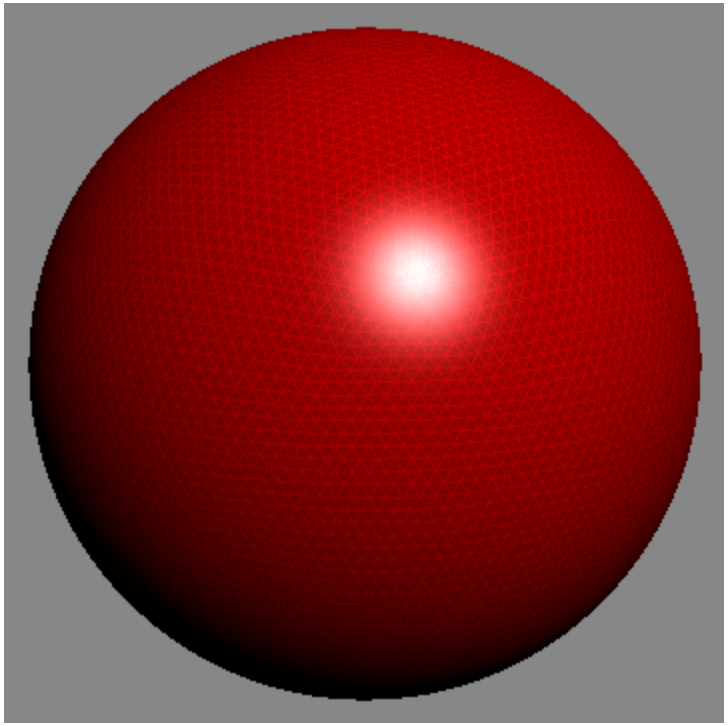


- If the highlight appears in the interior of the polygon, Gouraud may fail to shade this highlight because no highlighted intensities are recorded/calculated at the vertices.

Gouraud with Different Resolution Geometry



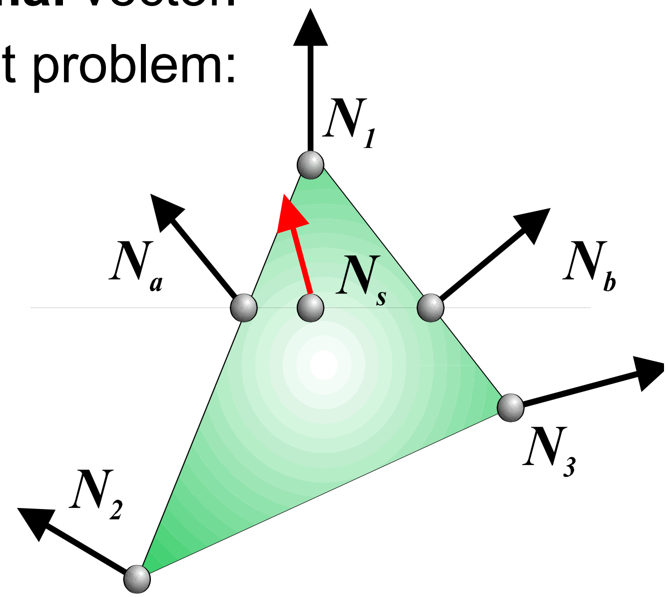
Effects of Gouraud shading with low quality geometry .



Effects of Gouraud shading with high quality geometry . .

Method 3: Phong Shading

- Instead of interpolating the vertex intensities, Phong proposed to interpolate the vertex **normal** vector.
- Solves the interior highlight problem:



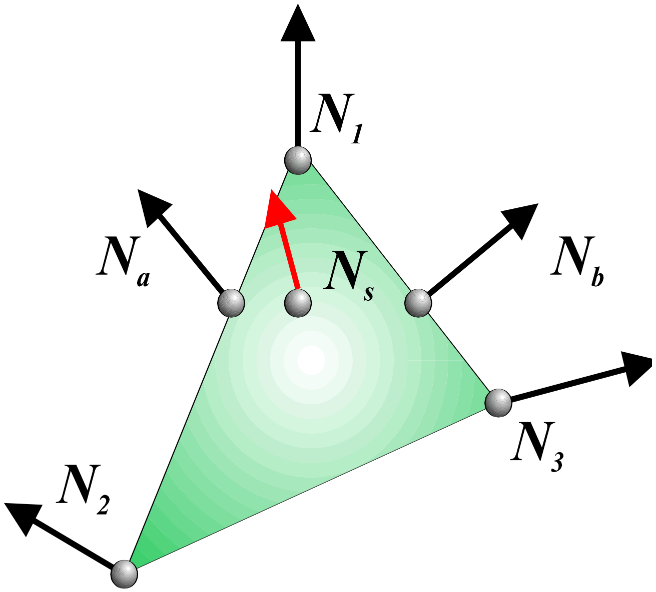
- Interpolation equations are:

$N_a = [N_1 (y_s - y_2) + N_2(y_1 - y_s)] / (y_1 - y_2)$
$N_b = [N_1 (y_s - y_3) + N_3(y_1 - y_s)] / (y_1 - y_3)$
$N_s = [N_a (x_b - x_s) + N_b(x_s - x_a)] / (x_b - x_a)$

At each point inside the polygon, we interpolate its N, then applying illumination model computation to get the value.

Don't confuse **Phong Shading** with **Phong Illumination** model.
Two different things, invented by the same person: Bui-Tuong Phong

Phong Shading



For each pixel:

(1) Interpolate the normal vector **N**

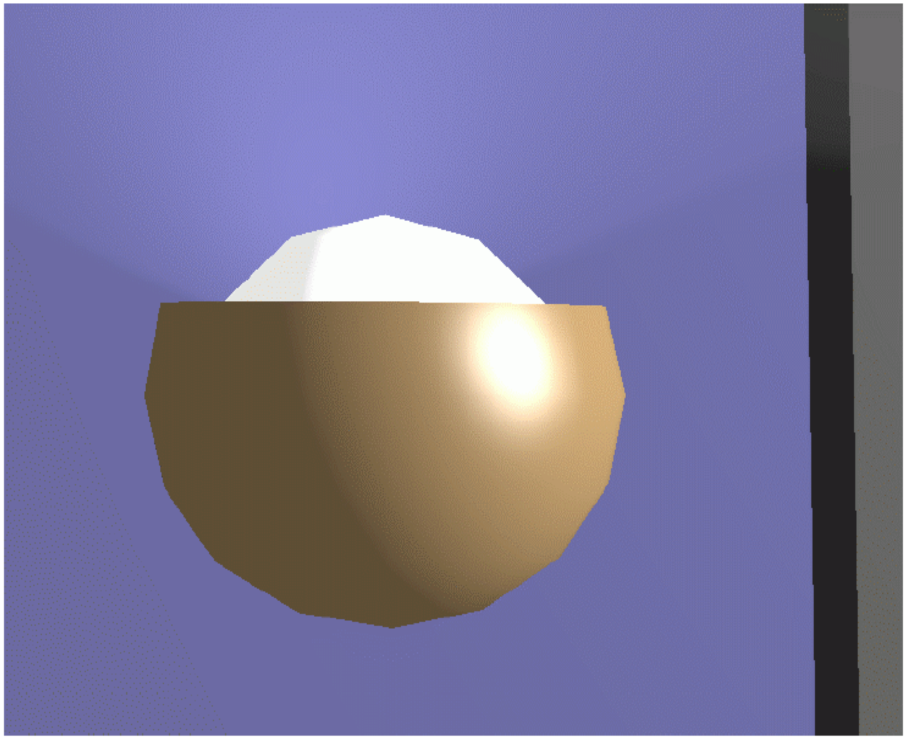
(2) Compute intensity model for this interpolated normal:

$$I = k_a I_a + k_d I_l (\boxed{Ns} \cdot L) + k_s I_l (V \cdot \boxed{R})^{n_s}$$

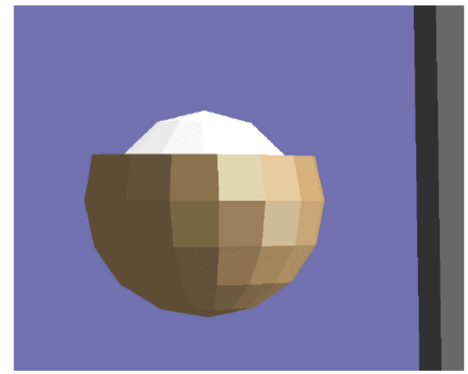
Use the interpolated Ns

(3) Shade this pixel with computed *I*

Phong Shading Example



Flat Shading



Compare w/ flat shading

Appears very smooth even though the underlying geometry is quite low. Better estimates the interior intensities.

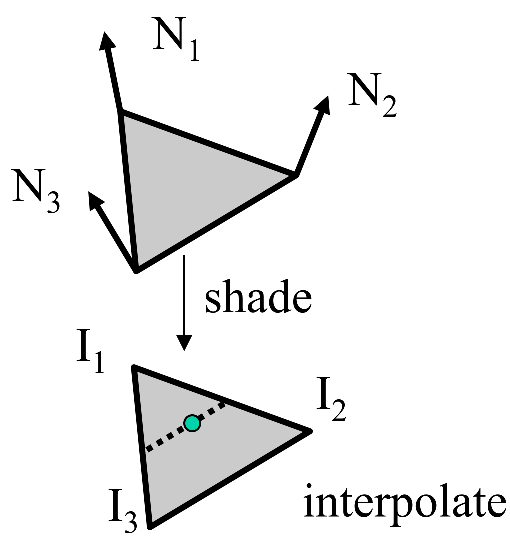
Since illumination calculation has to be invoked at *each* interior surface point, Phong shading is much more expensive than Gouraud shading -- **Phong shading is not supported by OpenGL ☹ -- too slow.**

Its too bad, because the result are very nice – even for low quality geometry! (Phong is awesome!)

Gouraud vs Phong Shading

Gouraud interpolation

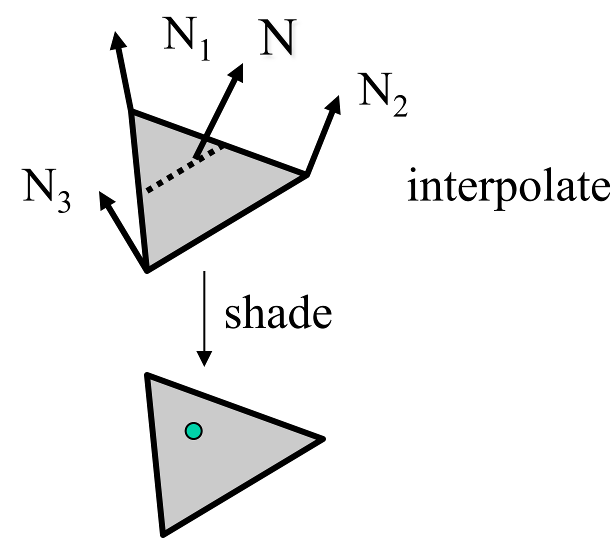
- Compute vertex normals
- Shade only vertices
- Interpolate the resulting vertex colors



Gouraud interpolation

Phong Shading

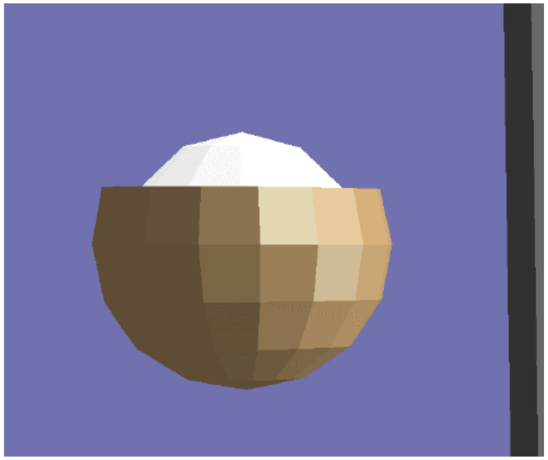
- Compute vertex normals
- Interpolate normals and normalize
- Shade using the interpolated normals



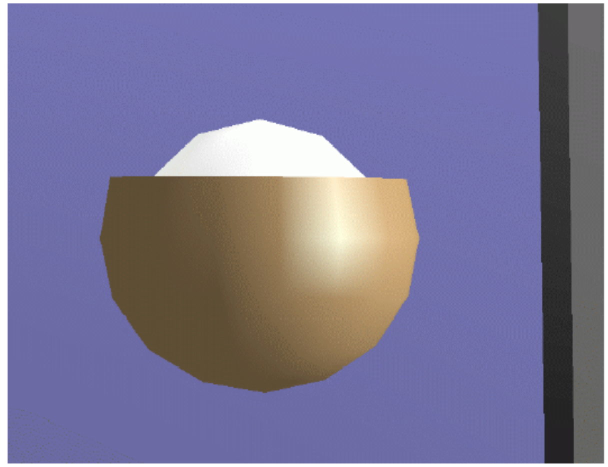
Phong interpolation

Comparisons

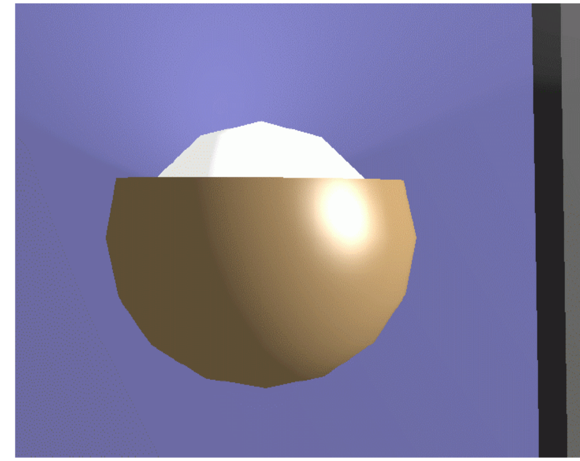
Flat

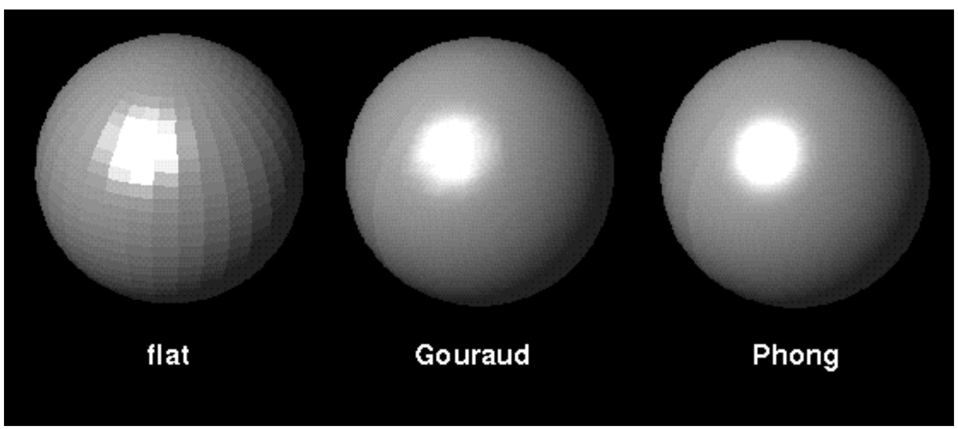


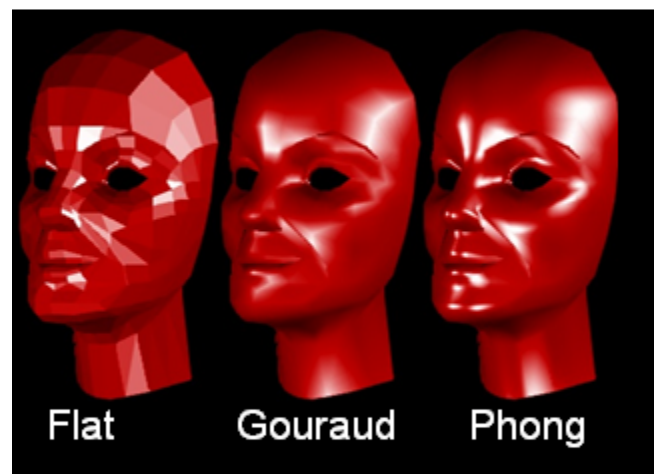
Gouraud



Phong







OpenGL Shading and Normals

Additional resources:
OpenGL Lighting

http://glprogramming.com/red/chapter05.html (Chapter 5 in the "RED BOOK")
http://www.opengl.org/resources/faq/technical/lights.htm (Info on Lighting)
http://www.videotutorialsrock.com/opengl_tutorial/lighting/video.php (video quality not so good, but a good lecture)

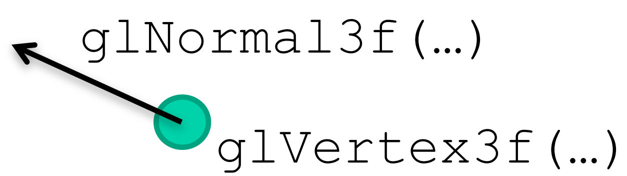
Computing Normals
http://www.codeguru.com/cpp/g-m/opengl/article.php/c2681

See lecture code:
OpenGL Lighting
light1.c, objectMaterials.c

Computing Normals
loadMeshFlat.c, loadMeshSmooth.c (and *.mesh files)

Normals

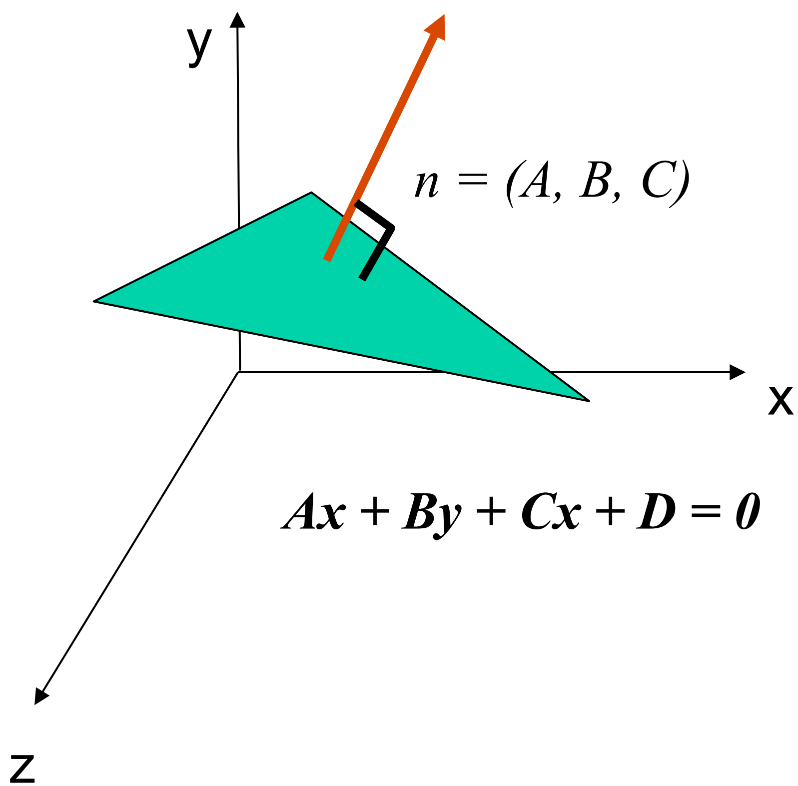
- Normals are crucial for OpenGL to perform shading
- Unfortunately, OpenGL does not compute normals! ☹
  - We must compute the normal ourselves
  - Each vertex must have an associated normal
    - That normal must be normalized to have length “1”
    - `glNormal3f(…)` is the command we use
    - We call this before we call `glVertex3f(…)`



- Note: glutTeapot, glutSphere, etc . . includes normals in the geometry

Surface Normal

The *normal* describes the orientation of the surface of a plane. It is fundamentally part of the plane equation.



Plane Equation:
$Ax + By + Cz + D = 0$

All (x,y,z) points on the plane must satisfy the above equation.

Normal:
Vector perpendicular to the plane
(the A, B, C components of the plane equation are the normal)

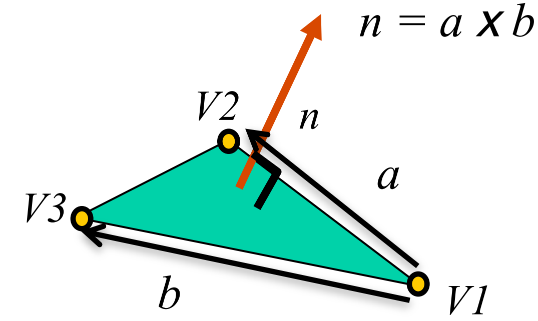
Given a 3D point , P=(x,y,z) , plug it into the equation and see if we get zero. The same as P dot N = D.

Computing the Normal

Consider we are given 3 points on a plane (ie, points from a triangle).

T = [(x1, y1,z1), (x2,y2,z2), (x3, y3, z3)]

V1 V2 V3



[here we assume the vertices are in counter-clockwise order]

```
/* A = v2 - v1 */
Ax = x2 - x1;
Ay = y2 - y1;
Az = z2 - z1;

/* B = v3 - v1 */
Bx = x3 - x1;
By = y3 - y1;
Bz = z3 - z1;

/* n = A x B */
nx = Ay*Bz - Az*By;
ny = Az*Bx - Ax*Bz;
nz = Ax*By - Ay*Bx;
```

STEPS:
Compute two vectors: A and B

$a = V2 - V1;$

$b = V3 - V1;$

$n = a \times b;$

Specifying the Normals with the Geometry

- Normals are specified with your geometry
  - OpenGL is a state machine, so what ever the current "normal" is, that is what is used for the following vertex calls
- Two Example:

Example 1:

```
glBegin(GL_TRIANGLE);
 glNormal3f(nx1, ny1, nz1 );
 glVertex3f(x1 ,  y1,  z1 );
 glNormal3f(nx2, ny2, nz2 );
 glVertex3f(x2 ,  y2,  z2 );
 glNormal3f(nx3, ny3, nz3 );
 glVertex3f(x3 ,  y3,  z3 );
glEnd();
```

This would be used for **Smooth shading**, we need a normal per vertex.

Example 2:

```
glBegin(GL_TRIANGLE);
 glNormal3f(nxf, nyf, nzf );
 glVertex3f(x1 ,  y1,  z1 );
 glVertex3f(x2 ,  y2,  z2 );
 glVertex3f(x3 ,  y3,  z3 );
glEnd();
```

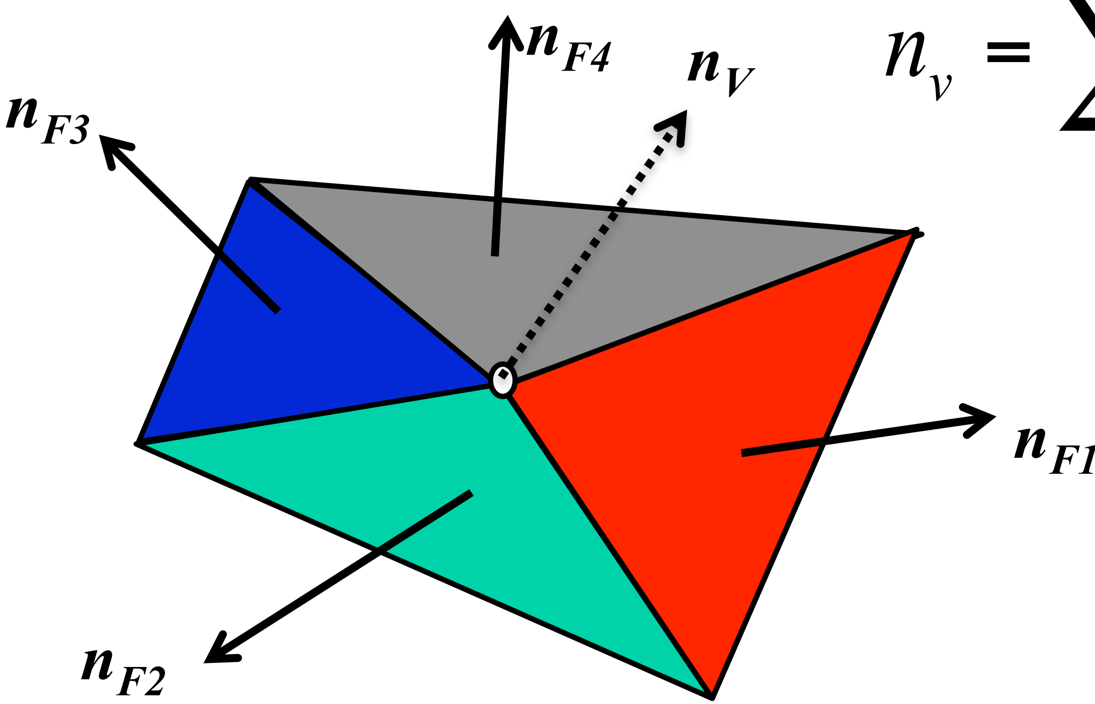
This would be used for **Flat shading**, we only need one normal per face. The normal call establishes the normal, then all subsequent vertex calls use this normal..

See CODE: loadMeshFlat.c, loadMeshSmooth.c

Normal per Vertex

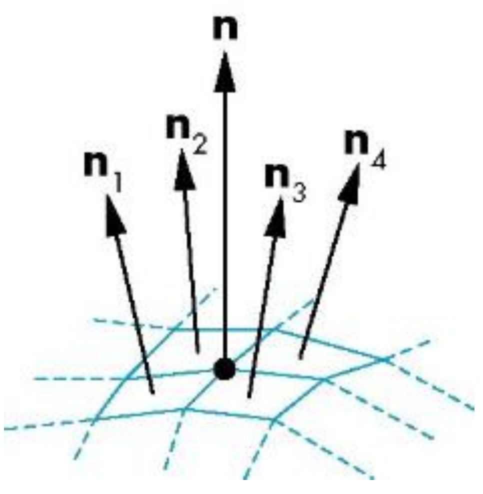
- If we requires a normal per vertex? (the most common case)
- What to do?
  - Average the normals of all the shared faces for a vertex.
    - This idea was proposed by Herni Gouraud

$$n_v = \sum_{i}^{m} \frac{n_{Fi}}{m}$$



Normalizing Your Normal

- OpenGL assumes that your specified normal is normalized
- Be sure to normalize it after computation – otherwise your lighting will be incorrect

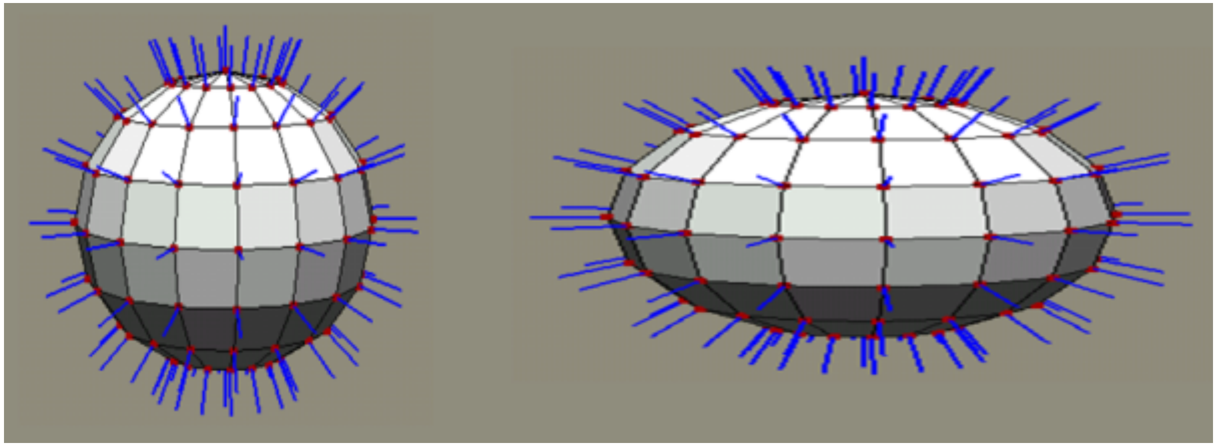


$$\mathbf{n} = (\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4)/\ |\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4|$$

See CODE: loadMeshSmooth.c --- the normals are averaged.

Normals Are Transformed Too!

- When you transform your geometry by matrix M, your normals are transform by matrix M too.
- This can cause your normals not to stay unit length



Original Normals

After glScale(2, 1, 1);
Normals lengths have been affected.

Translation won't effect a *normal*.
Rotation won't change a *normal's* length, but will change its direction.
Scale will affect a *normal's* length (and direction for non-uniform scale)

To Avoid Normal Change in Length

- Enable “normalize”
  - `glEnable(GL_NORMALIZE);`
- This will normalize your normal *after* transformation
  - Note, you still need to specify your normal as a normalized vector in `glNormal3f(…)`! OpenGL expects the original specified normal to be unit length

Steps to Perform Shading in OpenGL

1. Enable Lighting, Light properties, and the Lighting Model (Flat or Smooth)
2. Specify object's material properties
3. Specify the light's position

Order of 2. and 3. doesn't really matter.

4. Specify the geometry with its associate normals

-- this will draw the geometry immediately with corresponding shading

Enabling Lighting and the Lights

- Lighting is an option in OpenGL – you must "turn it on"
  - `glEnable(GL_LIGHTING);`
- OpenGL supports up to 8 light sources - you must "turn them on"
  - `glEnable(GL_LIGHT0);`
  - `glEnable(GL_LIGHT1); . . Up to GL_LIGHT7`
  - *come on, do you really need more than 8?*
- We can select the type of shading we want
  - `glShadeModel(GL_FLAT);         /* Flat     */`
  - `glShadeModel(GL_SMOOTH);       /* Gouraud */`
- IMPORTANT NOTE
  - Once lighting is enabled, `glColor3*()` is ignored!!!

OpenGL Practices

- Typically we place the enabling of lights in the initialization function

```
void init()
{
   GLfloat light_ambient[]={0.25, 0.25, 1.0, 1.0};
   GLfloat light_diffuse[]={0.1, 0.5, 0.5, 1.0};
   GLfloat light_specular[]={1.0, 1.0, 1.0, 1.0};

   /* set up ambient, diffuse, and specular components for light 0 */
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

    glShadeModel(GL_SMOOTH); /*enable smooth shading */
    glEnable(GL_LIGHTING); /* enable lighting */
    glEnable(GL_LIGHT0);  /* enable light 0 */
    glEnable(GL_DEPTH_TEST); /* enable z buffer */

    glEnable(GL_NORMALIZE);   /* this will normalize normals after transformation */
}
```

“Basic” Properties of a Light Source

- Four properties per light

```
glLightv(GL_LIGHT0, GL_AMBIENT, . . .);
glLightv(GL_LIGHT0, GL_DIFUSSE, . . .);
glLightv(GL_LIGHT0, GL_SPECULAR, . . .)
glLightv(GL_LIGHT0, GL_POSITION, . . .);
```

- For light position, there is a “trick*”

```
glLightf(GL_LIGHT0, GL_POSITION, x, y, z, 0.0 or 1.0);
```

If 0 then x,y,z is a direction (directional light)

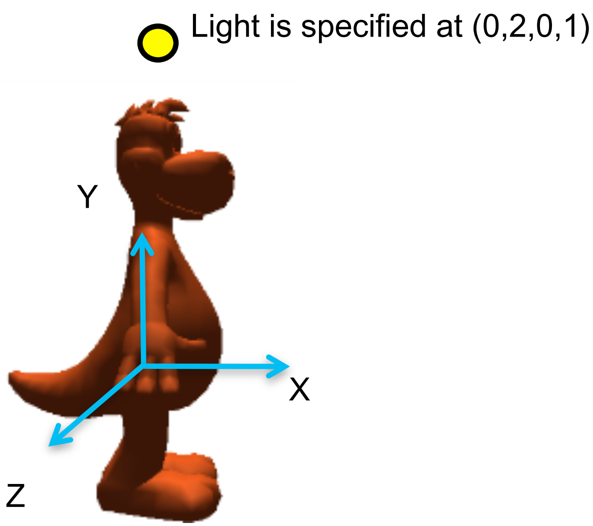
If 1 then x,y,z is a point   (point light)

*If you remember our homogenous representation, this isn’t really a trick . . w=0 is s a vector, w=1 is a point.

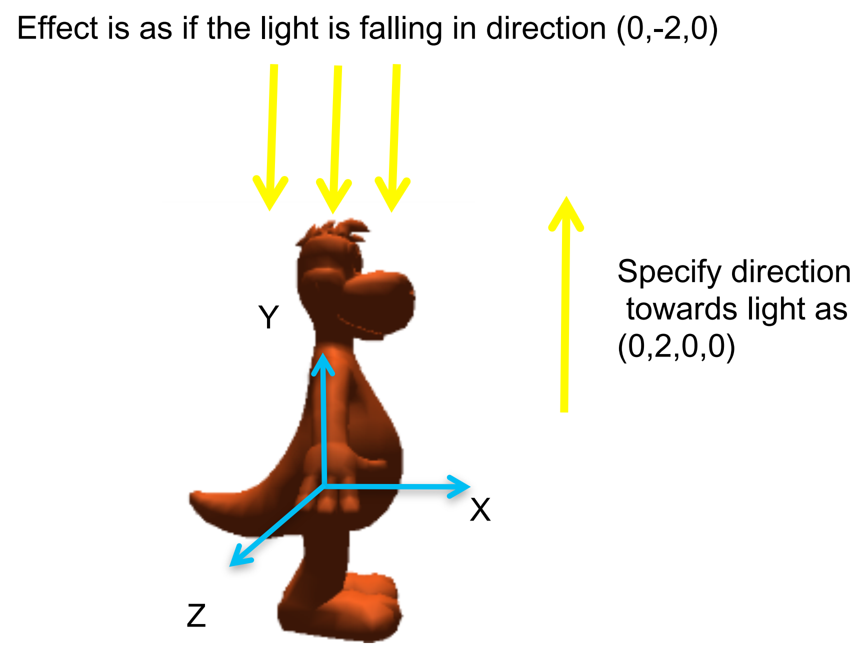
For the *ambient, diffuse, specular* properties we need 4 values representing Red, Green, Blue, and Alpha (right now, let’s ignore alpha). These make up the “color” of the light. Note that the colors of ambient, diffuse, and specular can be different (which is a bit strange)

Specifying the Light's Position/Direction

- Light sources are treated like geometric objects whose positions and directions are affected by the model-view matrix
- You should specify your light source in the world coordinate frame



As a point light source –
it's the point where the light is



As a directional light source –
it's the direction towards the light

Global Ambient Light

- Ambient light depends on color of light sources
  - A red light in a white room will cause a red ambient term that disappears when the light is turned off

- OpenGL also allows a global ambient term that is often helpful for testing
  - **`glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient)`**

  - Note that this is not associate with any particular light source, it's a "global ambient"

Moving Light Sources

- Depending on where we place the position (direction) setting function, we can
  - Move the light source(s) with the object(s)
  - Fix the object(s) and move the light source(s)
  - Fix the light source(s) and move the object(s)
  - Move the light source(s) and object(s) independently

- This is completely up to you as the programmer
  - Again, just think of the light as just geometric coordinate you can manipulate like any other geometry

Object's Material Properties

- Material properties are also part of the OpenGL state and match the terms in the phong model
- Set by **`glMaterialv()`**

```
                     /*  R,    G,    B,    A  */
GLfloat ambient[]  = {0.2, 0.2, 0.2, 1.0};
GLfloat diffuse[]  = {1.0, 0.8, 0.0, 1.0};
GLfloat specular[] = {1.0, 1.0, 1.0, 1.0};
GLfloat shine = 100.0
glMaterialf(GL_FRONT, GL_AMBIENT, ambient);
glMaterialf(GL_FRONT, GL_DIFFUSE, diffuse);
glMaterialf(GL_FRONT, GL_SPECULAR, specular);
glMaterialf(GL_FRONT, GL_SHININESS, shine);
```

$$I = \underbrace{k_a}_{\text{ambient}} I_a + \underbrace{k_d}_{\text{diffuse}} I_l (N \cdot L) + \underbrace{k_s}_{\text{specular}} I_l (V \cdot R)^{n_s}$$

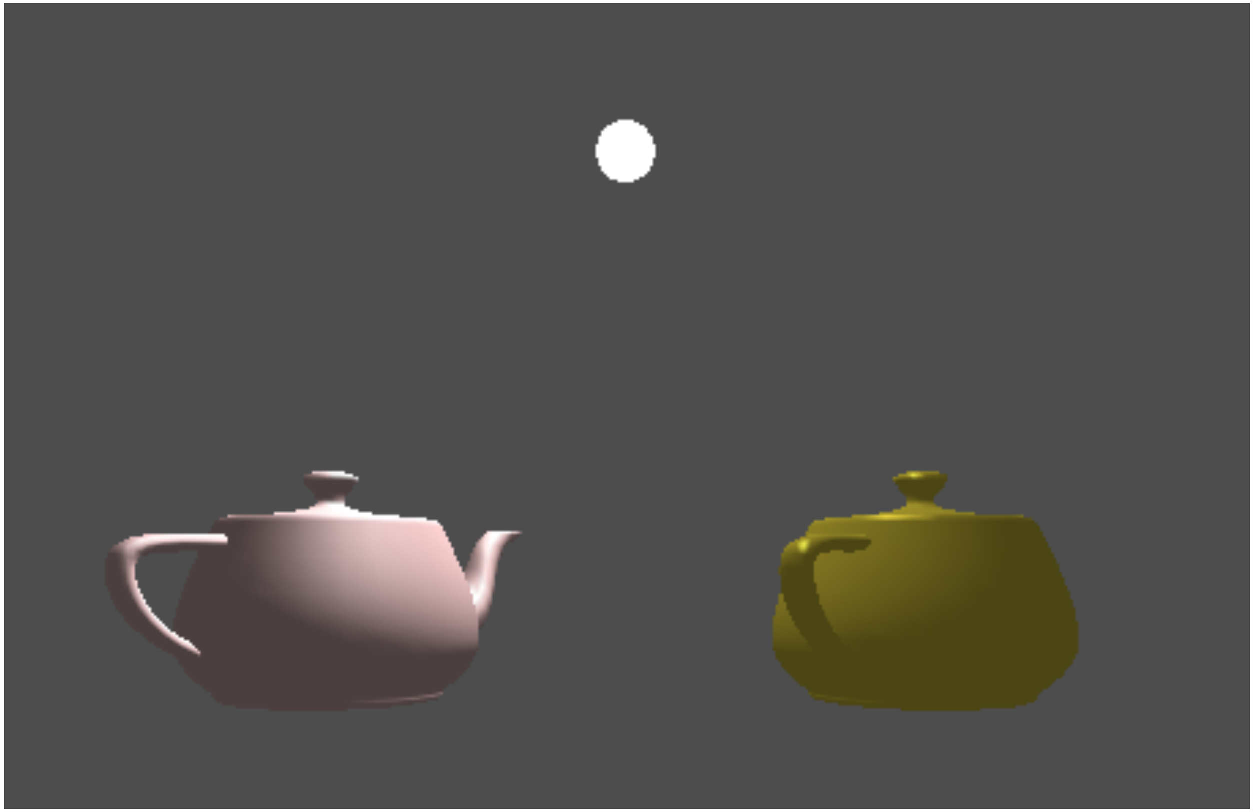
As with light source propeties, we can set the response of the materials to the to the R, G, B, and A.

Material's Emissive Term

- We can simulate an ambient light source in OpenGL by giving a material an emissive component
- This component is unaffected by any sources or transformations

```
GLfloat emission[] = 0.0, 0.3, 0.3, 1.0);
glMaterialf(GL_FRONT, GL_EMISSION, emission);
```

See code example: objectMaterials.c



Some examples of materials properties:
  http://www.cs.utk.edu/~kuck/materials_ogl.htm

Shading Summary

- Three types of shading
  - Flat, Gouraud, Phong
  - Opengl supports Flat and Gouraud
  - New nVidia/ATI hardware can perform Phong
- OpenGL shading
  - Enabling lighting, lights
  - Specifying light properties and object materials
  - Setting up the light
  - (please review sample code)