Lighting Continued

CS452/CS552: Computer Graphics

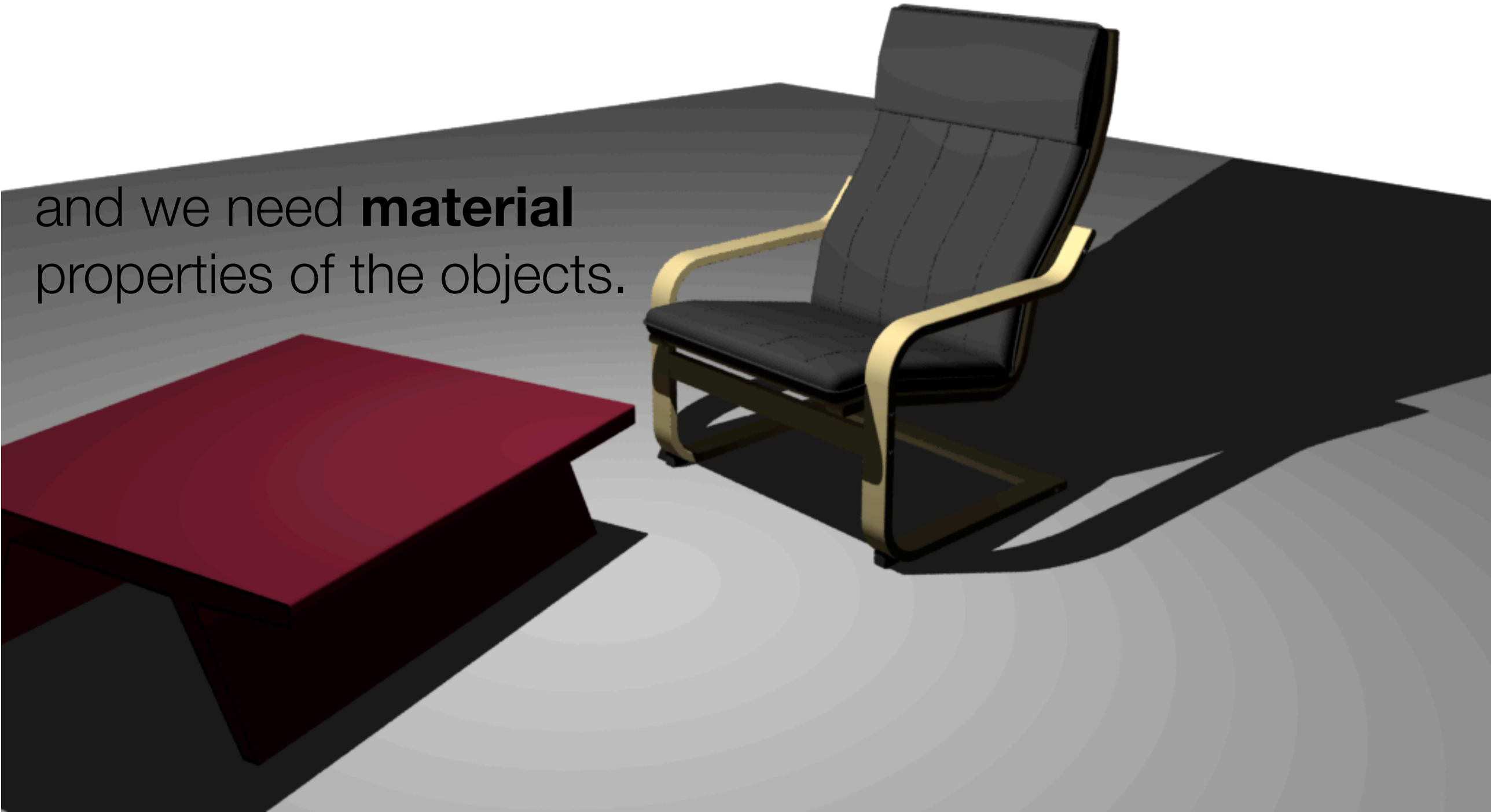
Chapter 6 of Angel & Shreiner

Recap of 10 Lighting

To light an object,

We need **light sources**,

and we need **material** properties of the objects.



Types of Light Sources

Ambient Light:
Specify intensity $\mathbf{I}_a$.

Point Light Sources:
Specify point $\mathbf{p_0}$ and intensity $\mathbf{I}(\mathbf{p_0})$ at point $\mathbf{p_0}$.

Spotlight:
Specify point $\mathbf{p_0}$, intensity $\mathbf{I}(\mathbf{p_0})$ at point $\mathbf{p_0}$, direction of spotlight axis $\mathbf{u}$, and exponent e.

Directional (Distant) Light Sources:
Specify direction $\mathbf{i}$ and intensity along direction $\mathbf{I}(\mathbf{i})$.

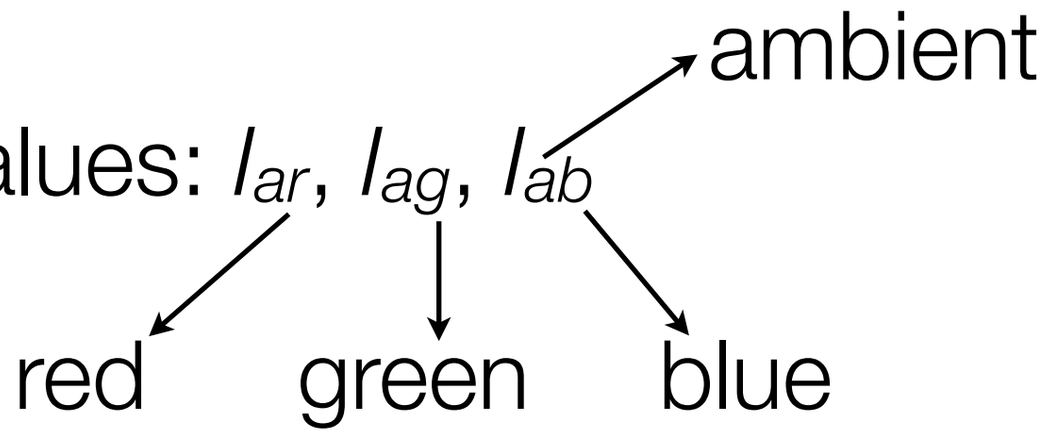
Types of Light Sources

Ambient Light: Uniform Light Present ‘Everywhere’

Most of this room has yellow lighting everywhere.

Or you could just have a single light quantity that represents ambient yellow light everywhere.

Ambient light has three values: I_{ar}, I_{ag}, I_{ab}



Types of Light Sources

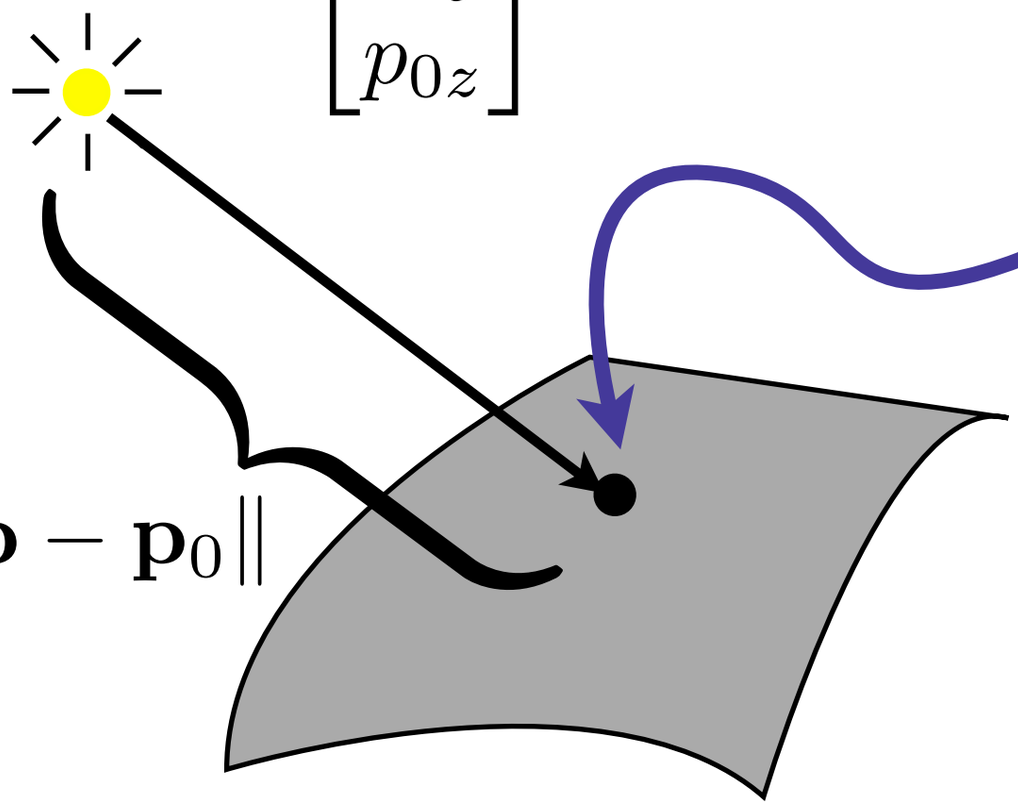
Point Light Source: Brightness falls off as **square** of distance

Source location

$$\mathbf{p}_0 = \begin{bmatrix} p_{0x} \\ p_{0y} \\ p_{0z} \end{bmatrix}$$

Location of point on surface

$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$



Distance: $\|\mathbf{p} - \mathbf{p}_0\|$

Brightness at source location $\mathbf{p}_0$:

$$\mathbf{I}(\mathbf{p}_0) = \begin{bmatrix} I_r(\mathbf{p}_0) \\ I_g(\mathbf{p}_0) \\ I_b(\mathbf{p}_0) \end{bmatrix}$$

Brightness at point $\mathbf{p}$:

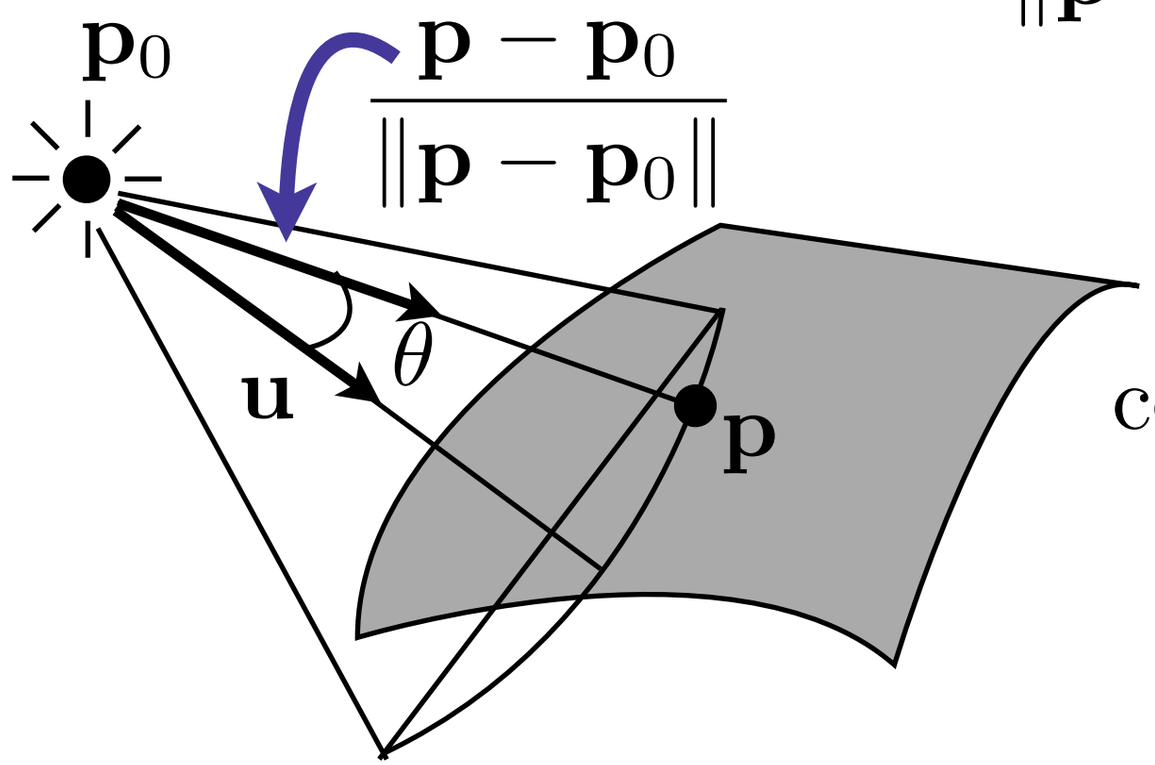
$$\mathbf{I}(\mathbf{p}, \mathbf{p}_0) = \frac{1}{\|\mathbf{p} - \mathbf{p}_0\|^2} \mathbf{I}(\mathbf{p}_0)$$

Dependence on square of distance

Types of Light Sources

Need source location $\mathbf{p}_0$ and direction of maximum intensity $\mathbf{u}$

Unit vector along direction from $\mathbf{p}_0$ to $\mathbf{p}$: $\dfrac{\mathbf{p}-\mathbf{p}_0}{\|\mathbf{p}-\mathbf{p}_0\|}$



$$\cos\theta = \mathbf{u} \cdot \left(\frac{\mathbf{p}-\mathbf{p}_0}{\|\mathbf{p}-\mathbf{p}_0\|} \right)$$

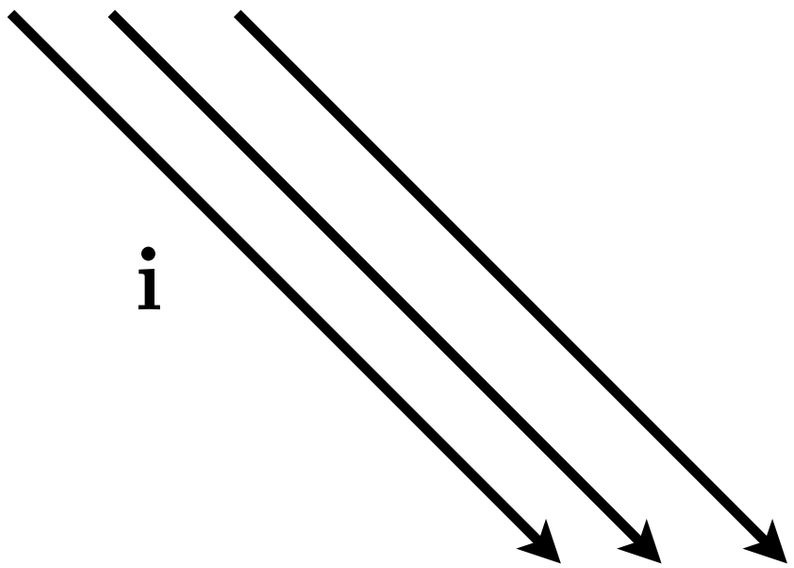
Dot product

Brightness at point $\mathbf{p}$: $\mathbf{I}(\mathbf{p}, \mathbf{p}_0) = \dfrac{1}{\|\mathbf{p}-\mathbf{p}_0\|^2} \mathbf{I}(\mathbf{p}_0) \cos^e \theta$

Types of Light Sources

Distant Light Sources (Directional Light)

The point of origin is at infinity. Which means all we know is the direction **i** the light is coming from.

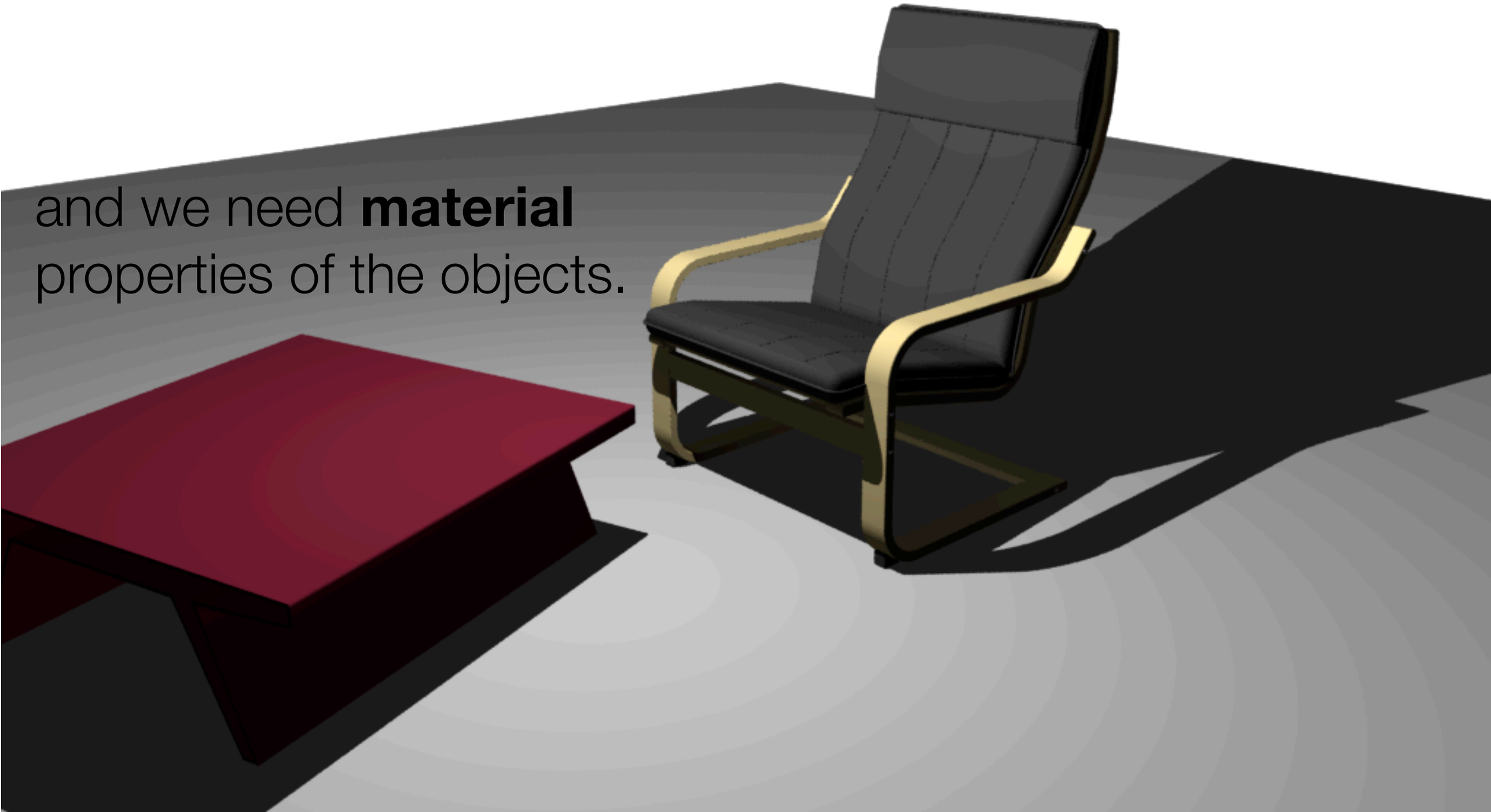


$$\mathbf{I}(\mathbf{i}) = \begin{bmatrix} I_r(\mathbf{i}) \\ I_g(\mathbf{i}) \\ I_b(\mathbf{i}) \end{bmatrix}$$

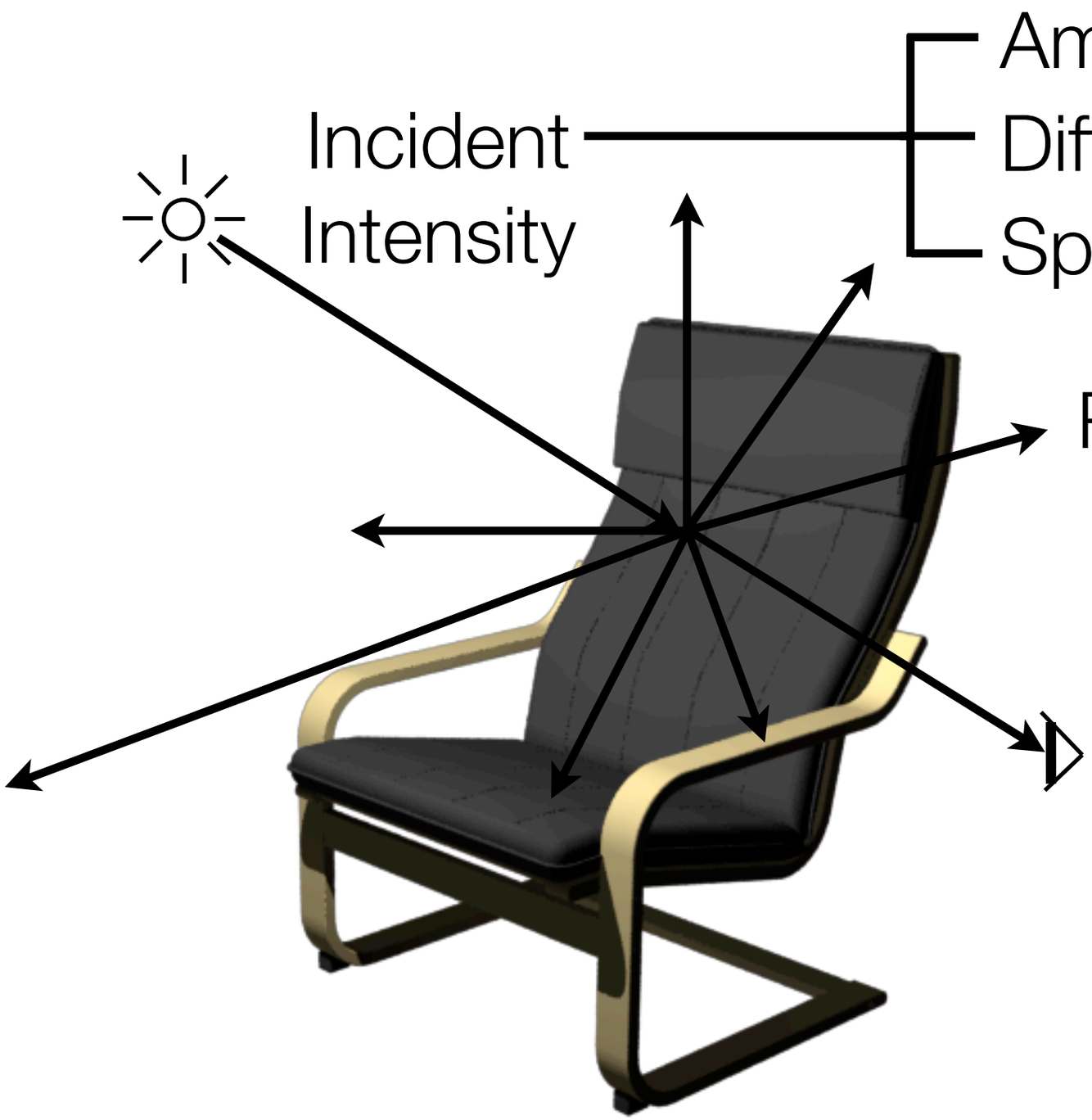
To light an object,

We need **light sources**,

and we need **material** properties of the objects.



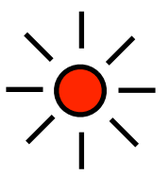
Phong Reflection Model



Ambient Reflection

Part of the incident light is reflected
without any directional dependence.

Ambient term in incident light
has high amount of red



$$R_{ar} = k_{ar} I_{ar},\ 0 \leq k_{ar} \leq 1$$
$$R_{ag} = k_{ag} I_{ag},\ 0 \leq k_{ag} \leq 1$$
$$R_{ab} = k_{ab} I_{ab},\ 0 \leq k_{ab} \leq 1$$

Object has high
amount of white

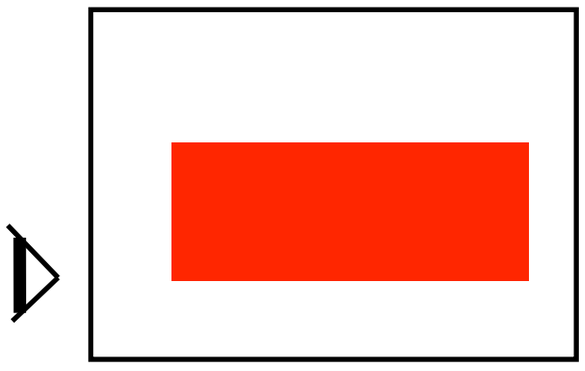
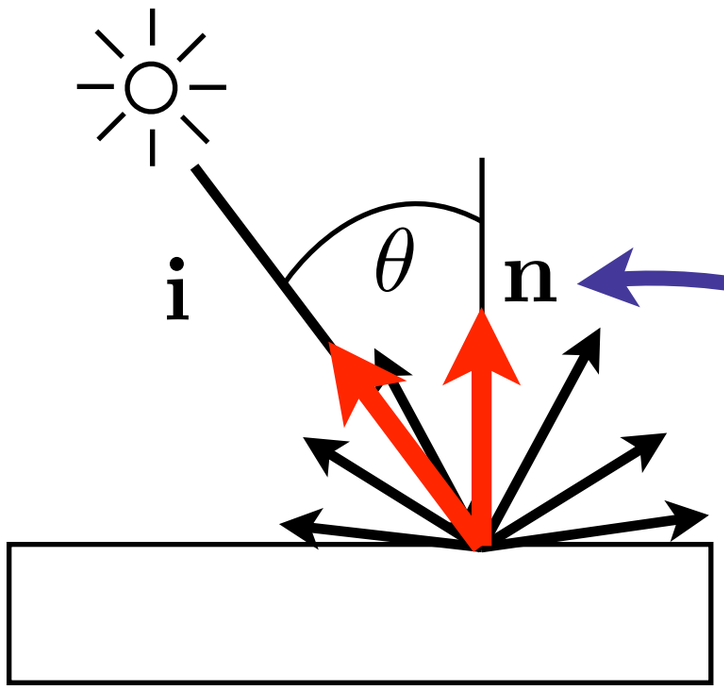


Image of object is red

Ambient reflection tells you why on average a white box illuminated by red light appears red.

Diffuse Reflection

$$R_{dr} = k_{dr} I_{dr} \max(\cos\theta, 0),\ 0 \le k_{dr} \le 1$$
$$R_{dg} = k_{dg} I_{dg} \max(\cos\theta, 0),\ 0 \le k_{dg} \le 1$$
$$R_{db} = k_{db} I_{db} \max(\cos\theta, 0),\ 0 \le k_{db} \le 1$$



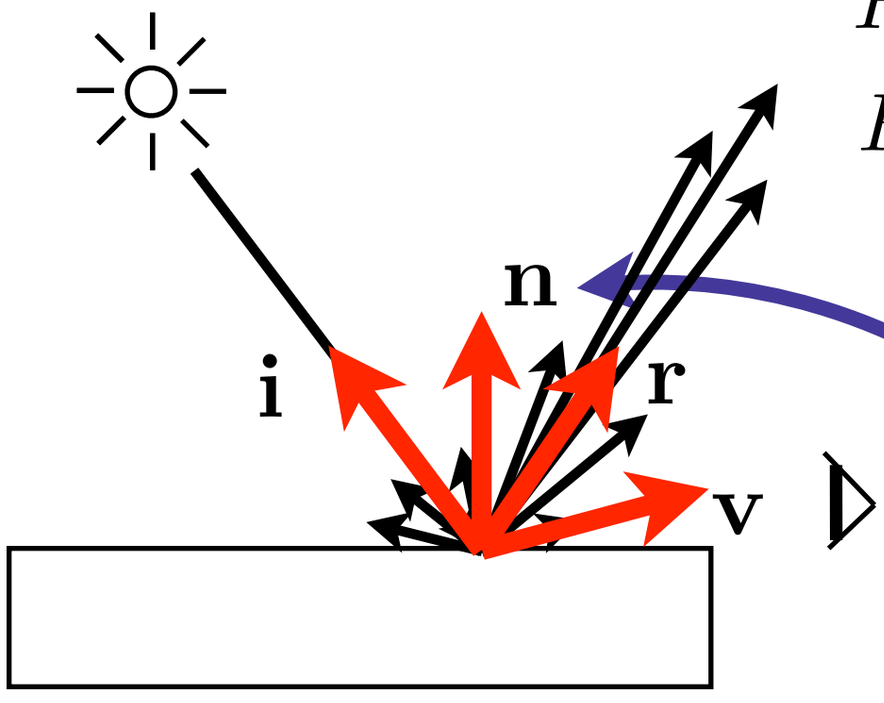
Normal: unit vector perpendicular to surface.

$$\cos\theta = \mathbf{i} \cdot \mathbf{n}$$

Why use max(cosθ,0)?

Specular Reflection

$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$
$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$
$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$



Normal: unit vector perpendicular to surface.

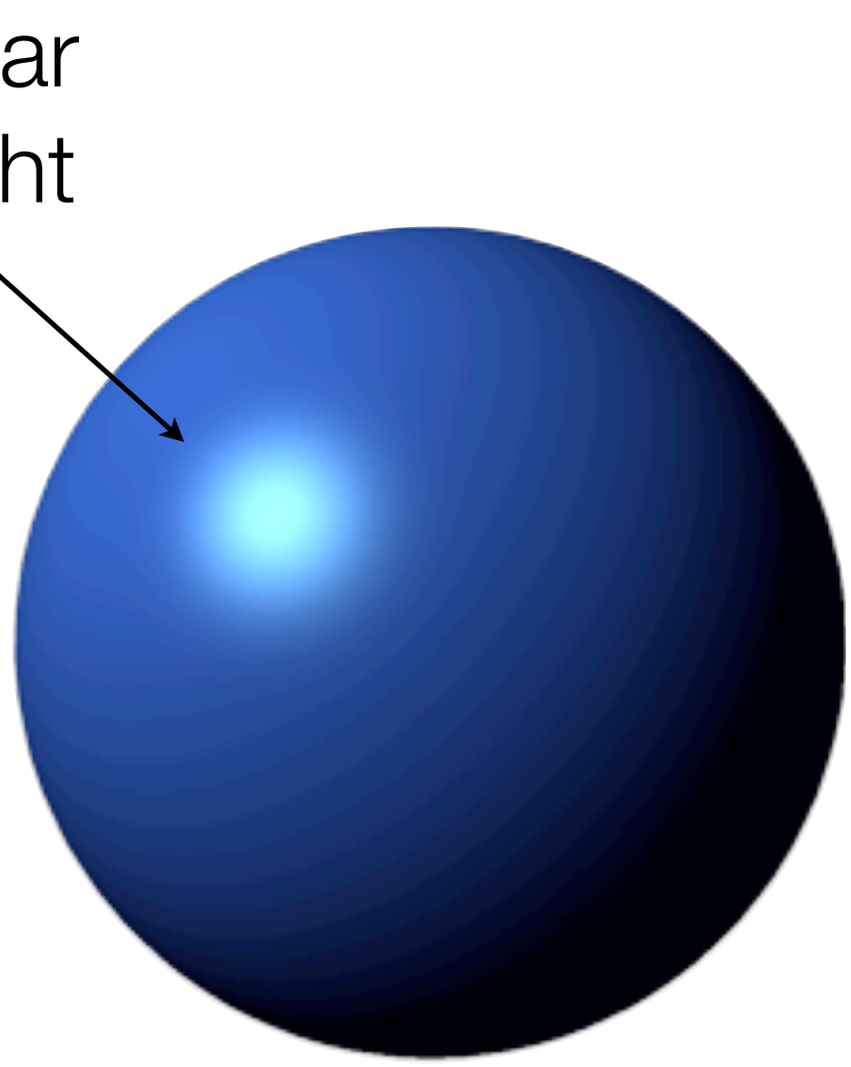
r is just **i** mirrored around **n**.

What is alpha?

$$\mathbf{r} = 2\left(\mathbf{i} \cdot \mathbf{n}\right)\mathbf{n} - \mathbf{i}$$

Diffuse versus Specular Reflections

Diffuse Only



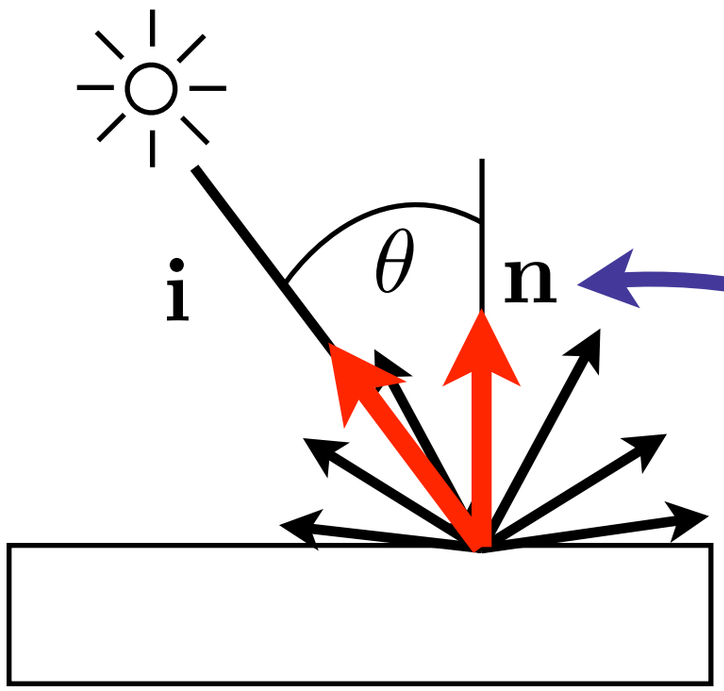
Diffuse + Specular

Normals and Shading

Normals

We need them! For diffuse reflection,

$$R_{dr} = k_{dr} I_{dr} \max(\cos\theta, 0),\ 0 \le k_{dr} \le 1$$
$$R_{dg} = k_{dg} I_{dg} \max(\cos\theta, 0),\ 0 \le k_{dg} \le 1$$
$$R_{db} = k_{db} I_{db} \max(\cos\theta, 0),\ 0 \le k_{db} \le 1$$



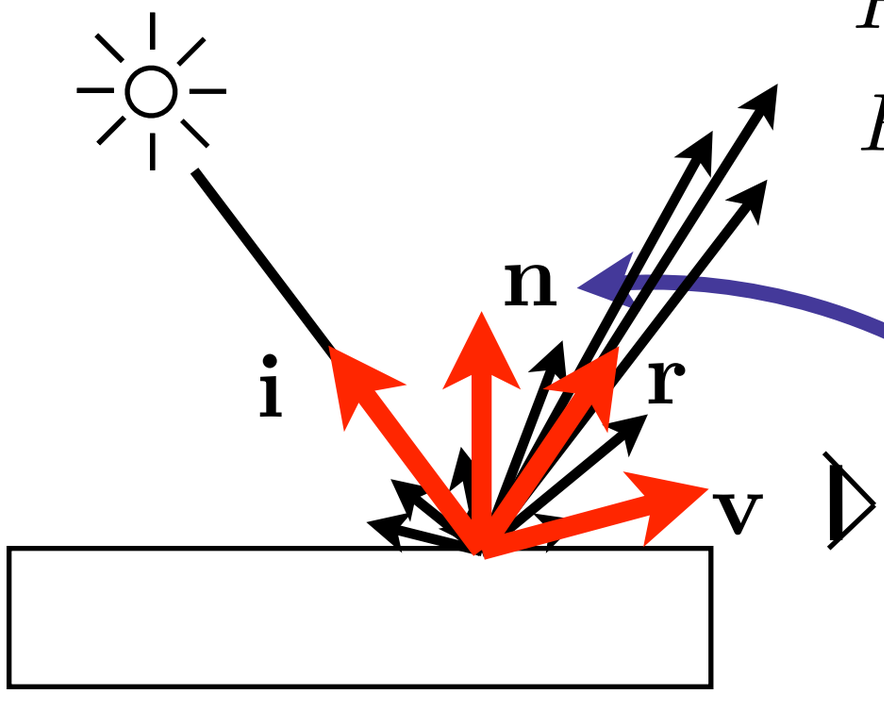
Normal: unit vector perpendicular to surface.

$$\cos\theta = \mathbf{i} \cdot \mathbf{n}$$

Normals

We need them! For diffuse reflection,
and for specular reflection.

$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$
$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$
$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$

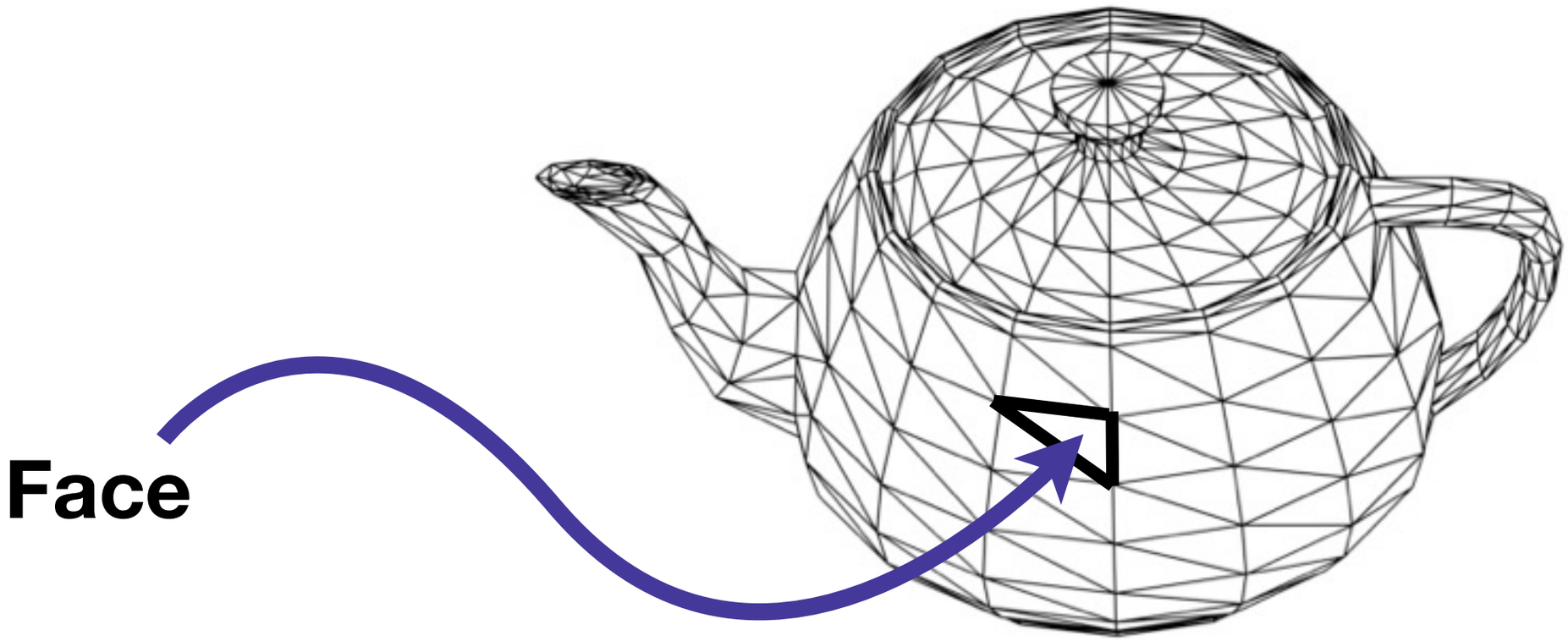


Normal: unit vector
perpendicular to surface.

Calculating Normals

Ideally, every little point on a surface should have a normal.

In practice, we compute normals over **faces**
of our 3D shapes. Here, a **face** refers to a triangle.

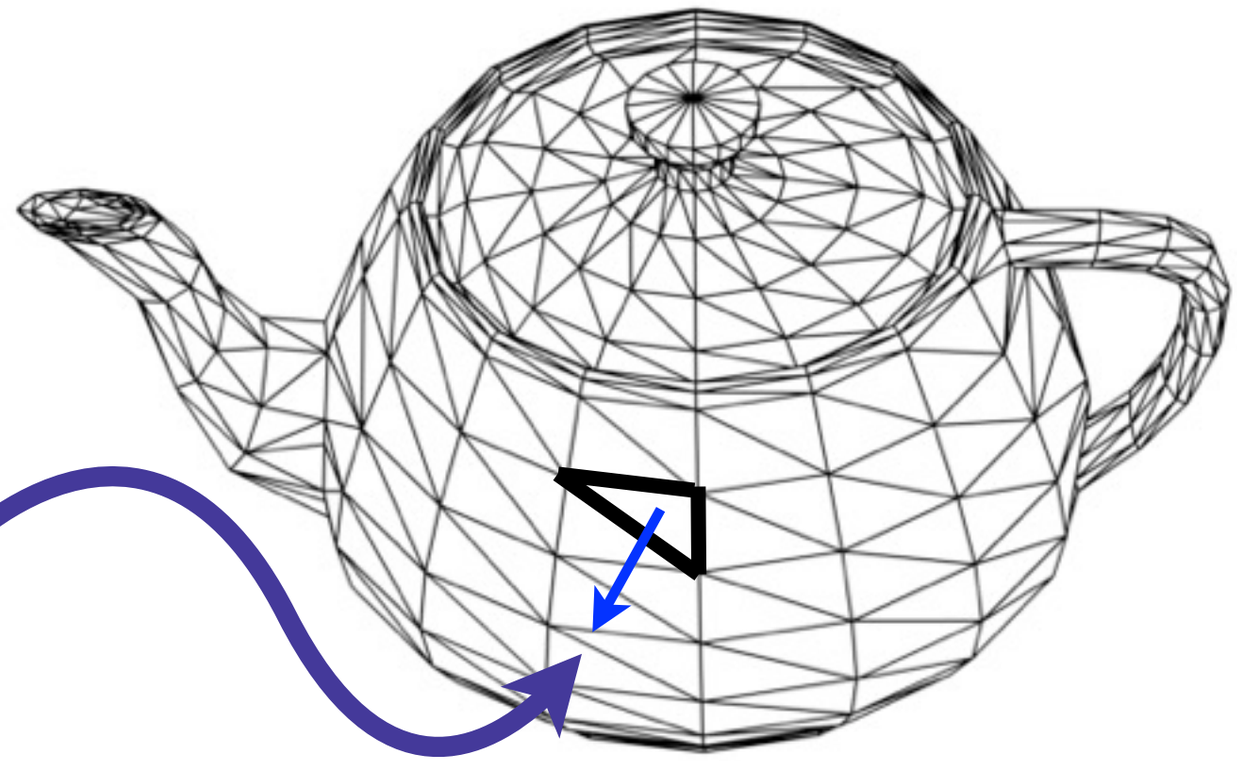


Calculating Normals

Ideally, every little point on a surface should have a normal.

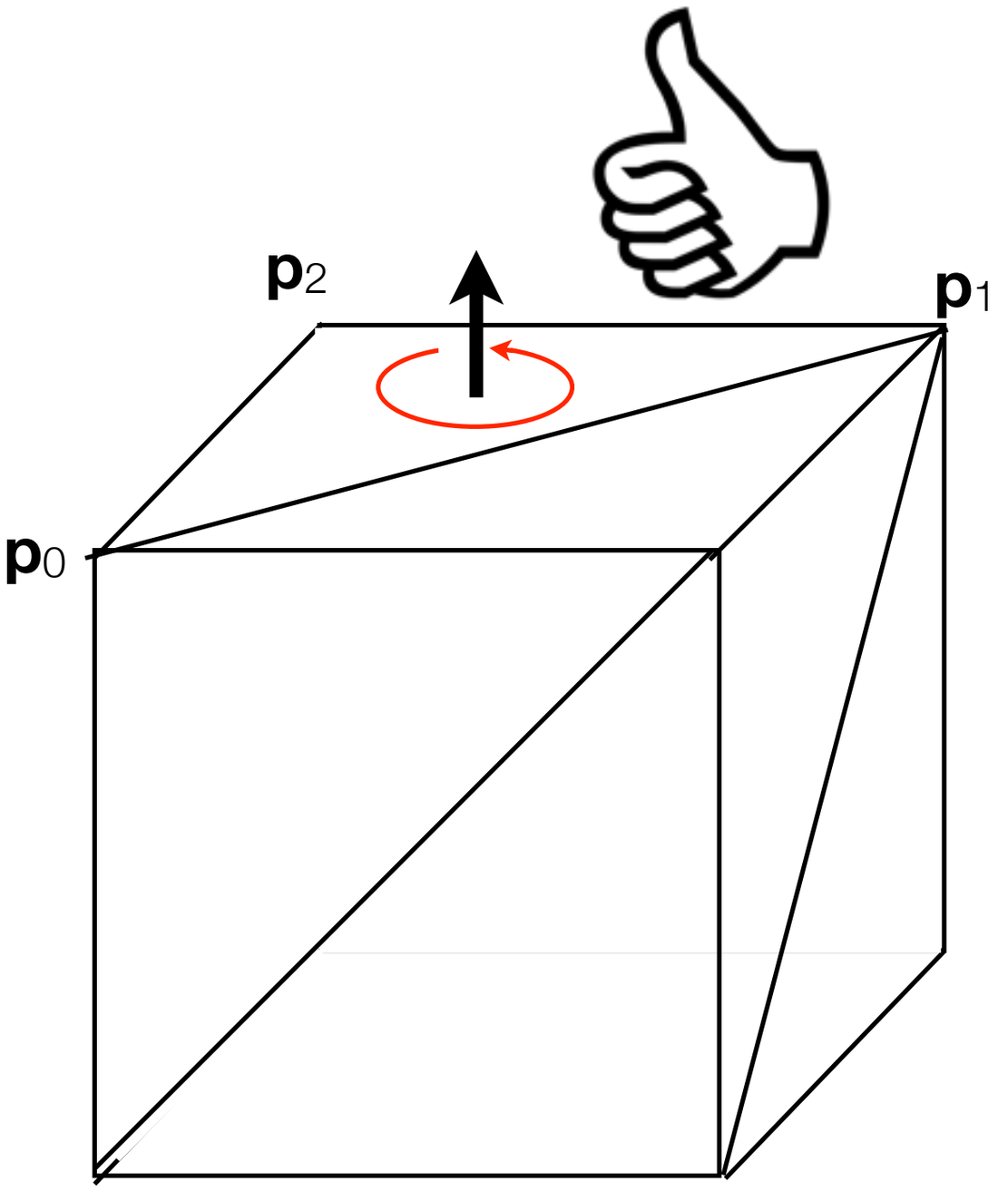
In practice, we compute normals over **faces**
of our 3D shapes. Here, a **face** refers to a triangle.

Normal: unit vector perpendicular to face pointed outwards.



Calculating Face Normals

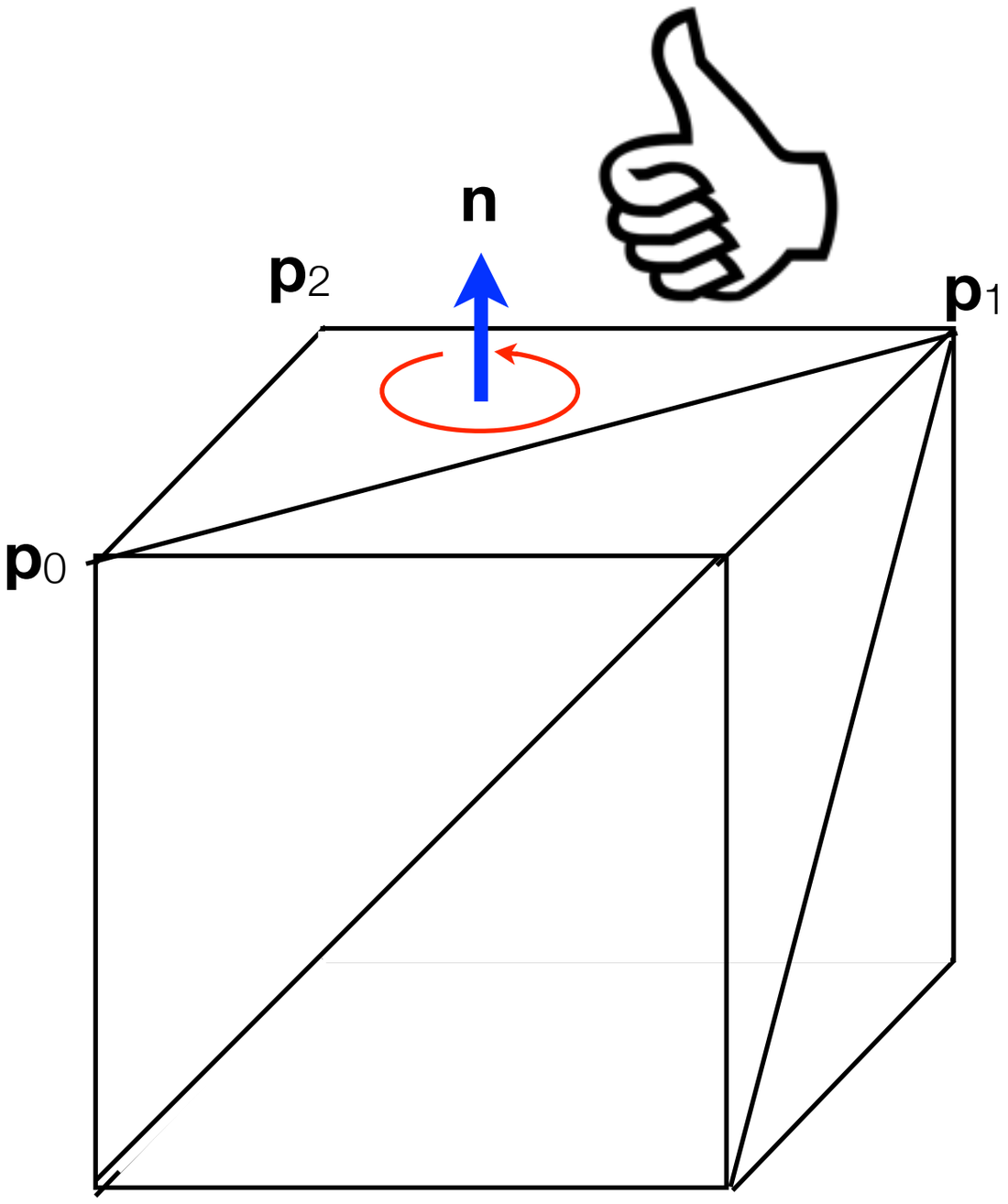
When we did 3D transformations, we ensured all vertices of a 3D shape are taken in counterclockwise order.



If you curl the fingers of your **right hand** around the order of vertices, the thumb points **outwards**.

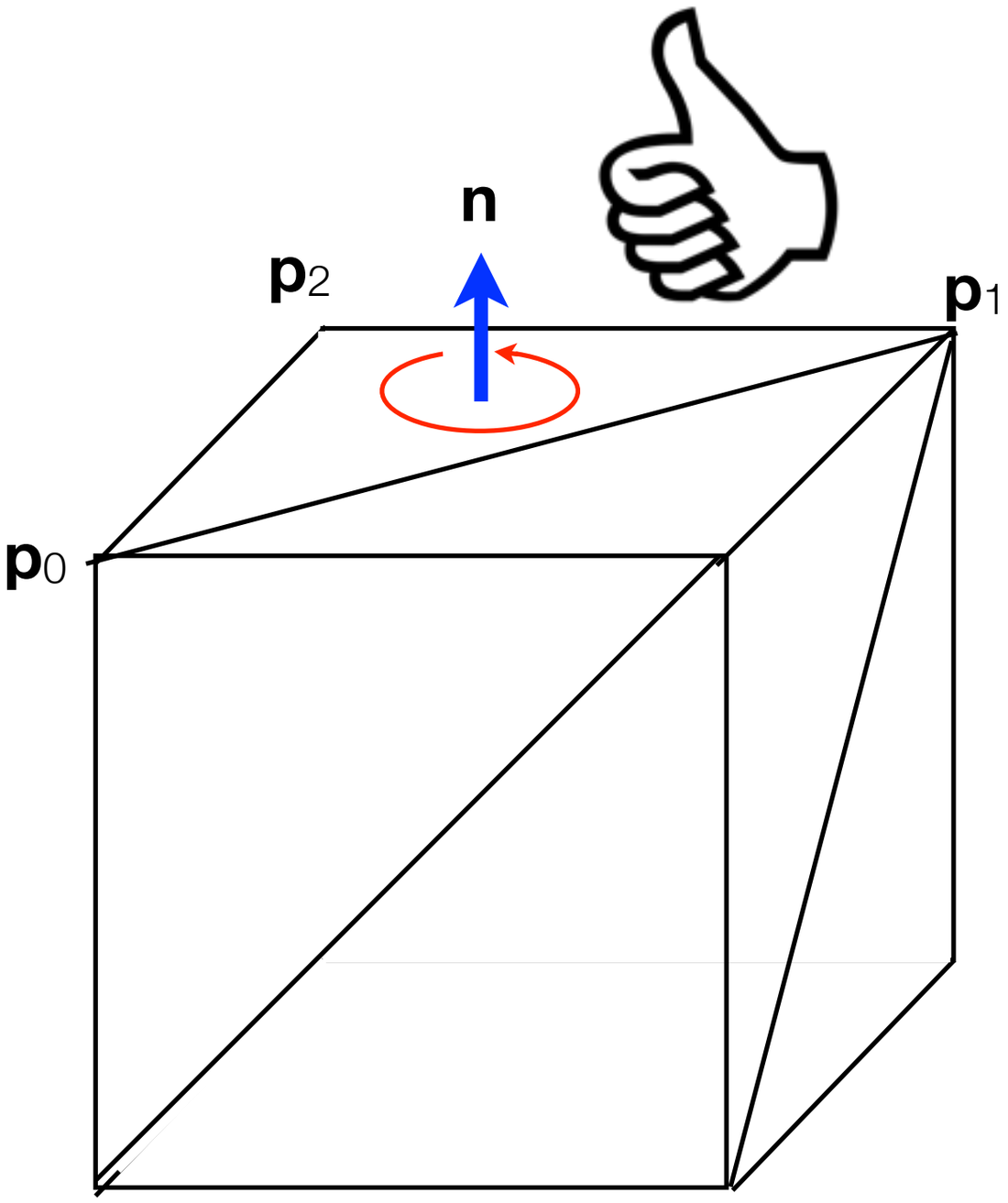
Calculating Face Normals

We will now get a unit normal **n** for the triangle $\mathbf{p}_0$ $\mathbf{p}_1$ $\mathbf{p}_2$



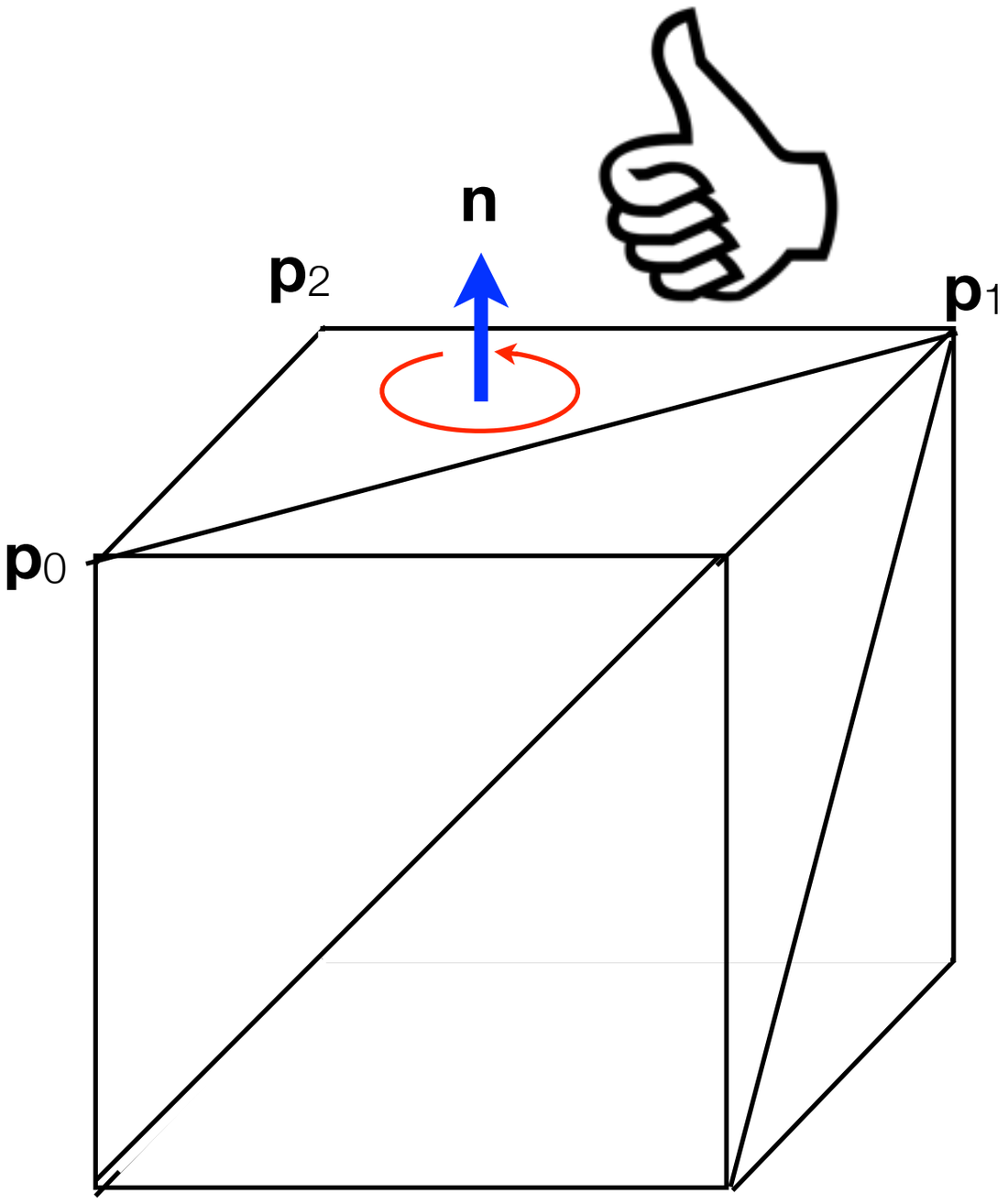
Calculating Face Normals

How can you get **n**?



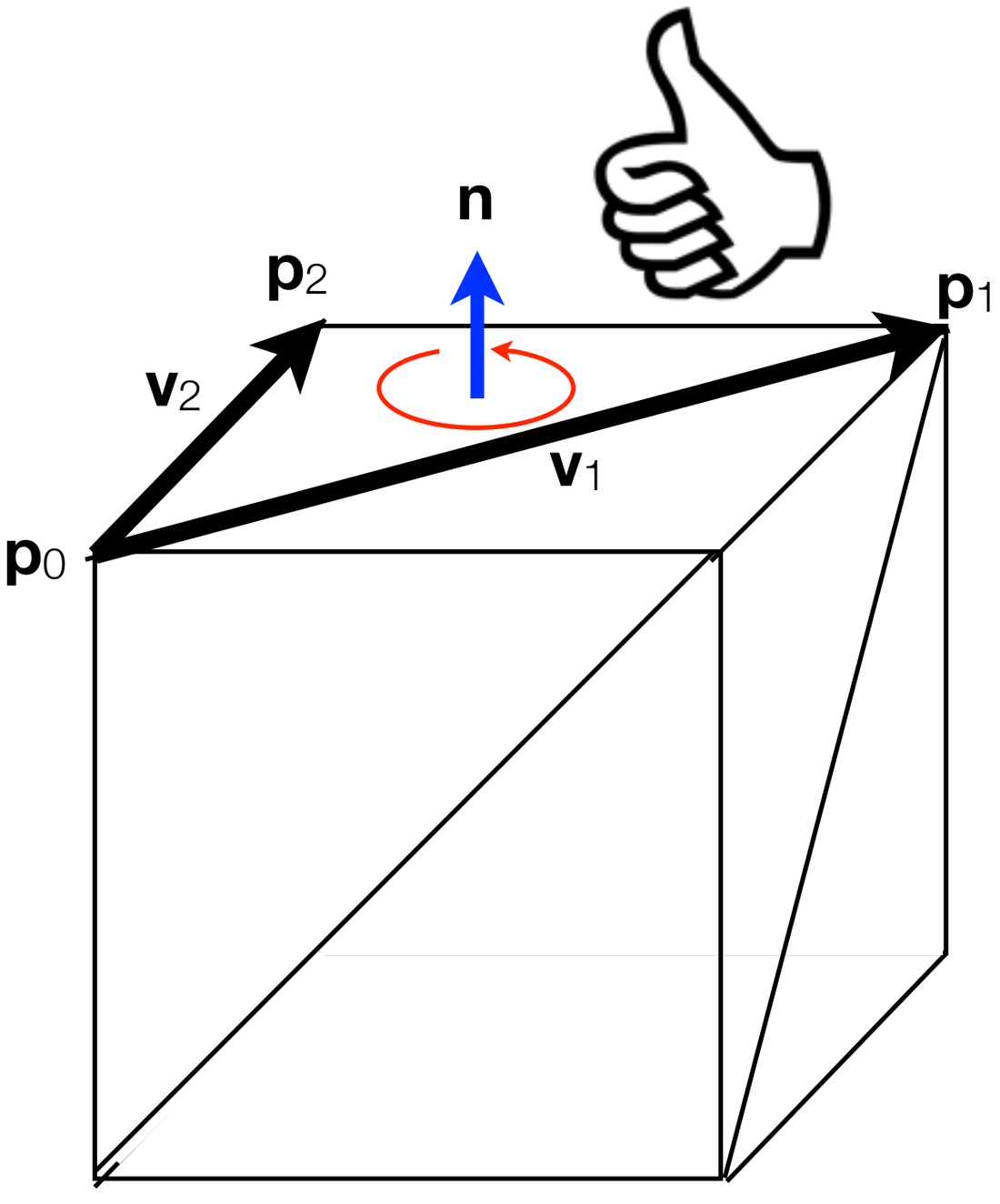
Calculating Face Normals

n is the normal perpendicular to the plane containing points $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$.



Calculating Face Normals

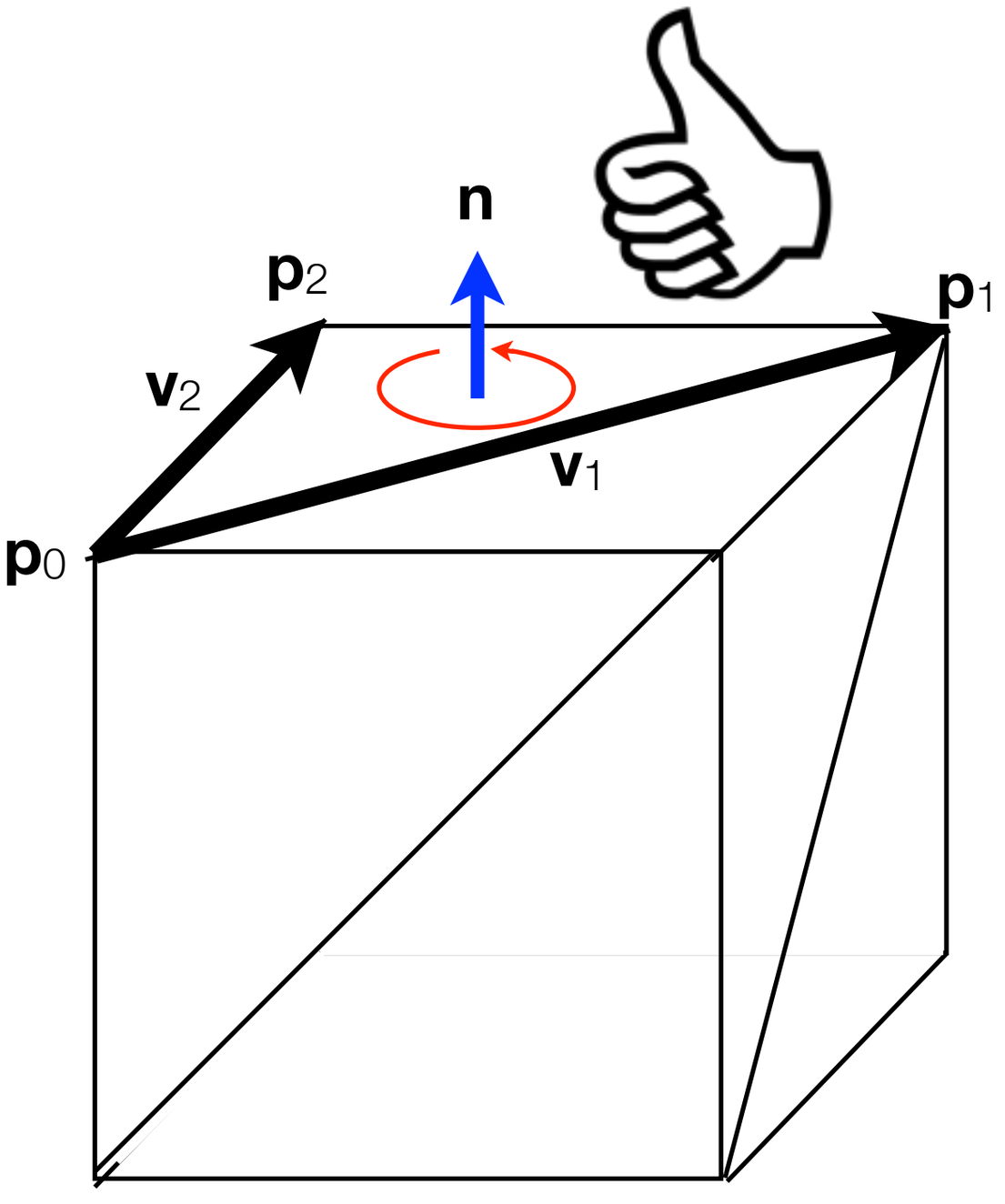
n is the cross product of vectors $\mathbf{v}_1$ and $\mathbf{v}_2$



$$\mathbf{n} = \mathbf{v}_1 \times \mathbf{v}_2$$

Calculating Face Normals

n is the cross product of vectors $\mathbf{v}_1$ and $\mathbf{v}_2$



$$\mathbf{n} = \mathbf{v}_1 \times \mathbf{v}_2$$

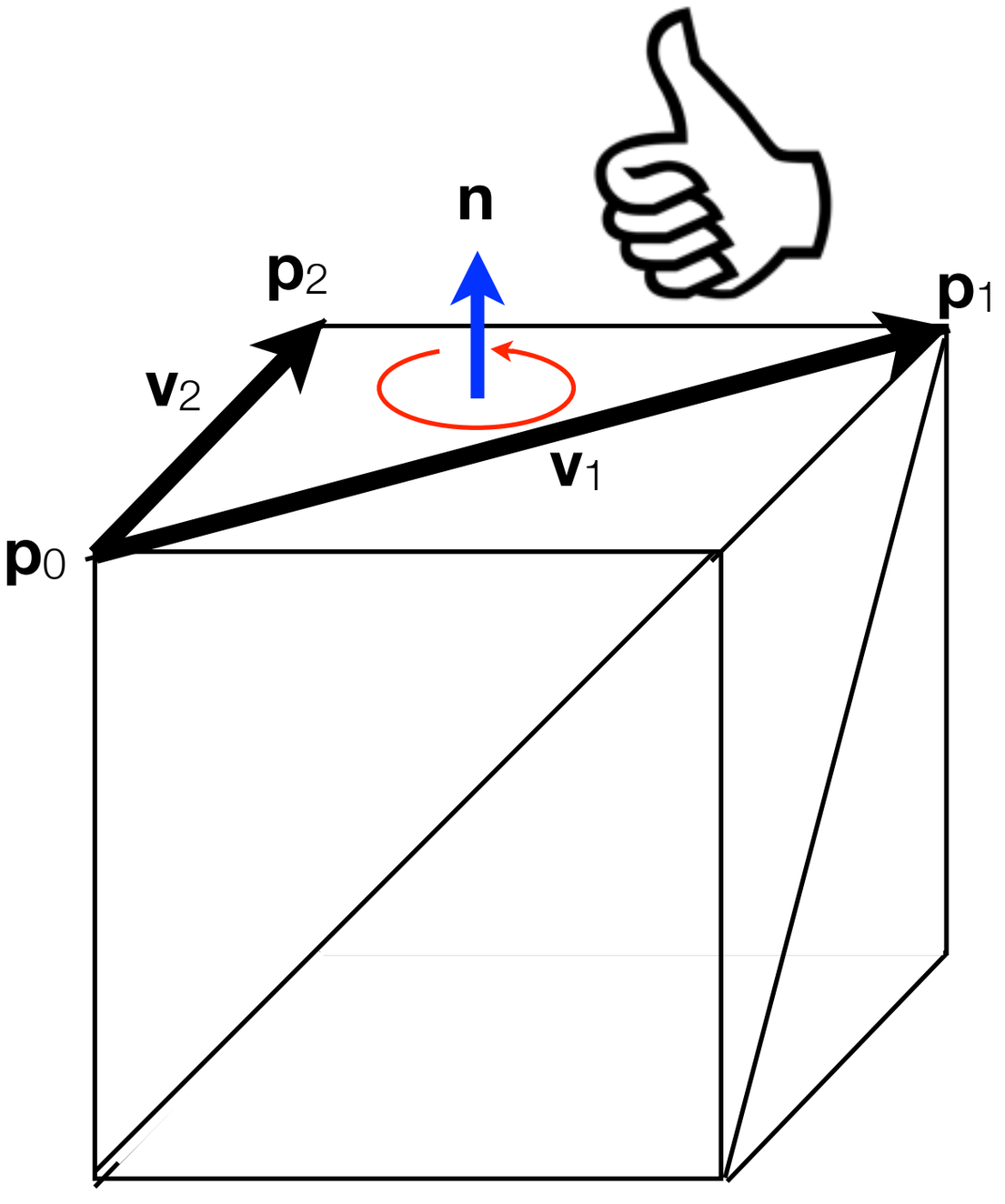
cross() in MV.js

Algebra of cross product:

$$\mathbf{n} = \begin{bmatrix} v_{1y}v_{2z} - v_{1z}v_{2y} \\ -v_{1x}v_{2z} + v_{1z}v_{2x} \\ v_{1x}v_{2y} - v_{1y}v_{2x} \end{bmatrix}$$

Calculating Face Normals

n is the cross product of vectors $\mathbf{v}_1$ and $\mathbf{v}_2$



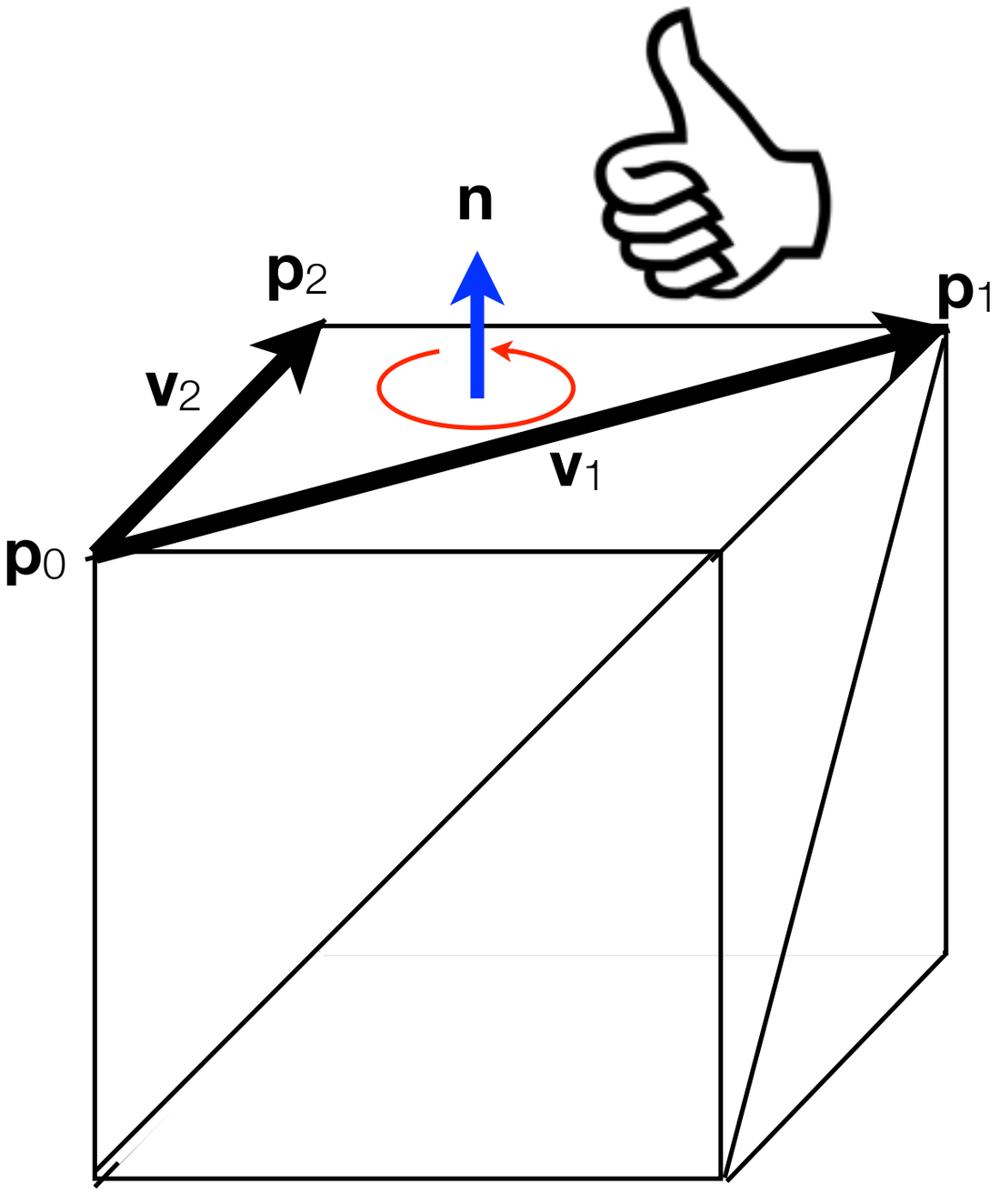
$$\mathbf{n} = \mathbf{v}_1 \times \mathbf{v}_2$$

Geometry of cross product:

n is a vector perpendicular to $\mathbf{v}_1$ and $\mathbf{v}_2$. If you wave your hand from $\mathbf{v}_1$ to $\mathbf{v}_2$, your thumb points in the direction of **n**.

Calculating Face Normals

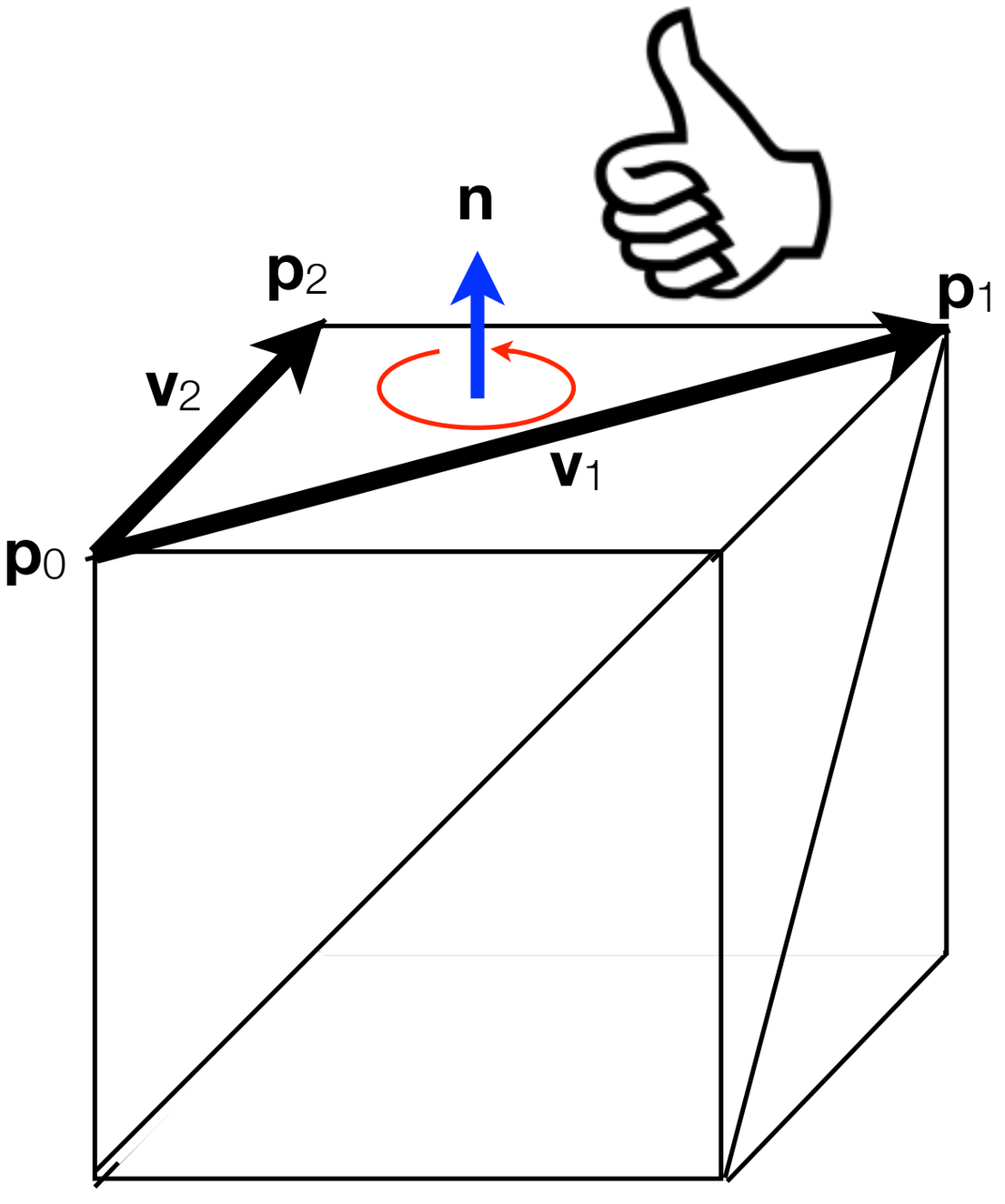
n is the cross product of vectors $\mathbf{v}_1$ and $\mathbf{v}_2$



$$\mathbf{n} = \mathbf{v}_1 \times \mathbf{v}_2$$

Calculating Face Normals

n is the cross product of vectors $\mathbf{v}_1$ and $\mathbf{v}_2$



$$\begin{aligned}\mathbf{n} &= \mathbf{v}_1 \times \mathbf{v}_2 \\ &= (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)\end{aligned}$$

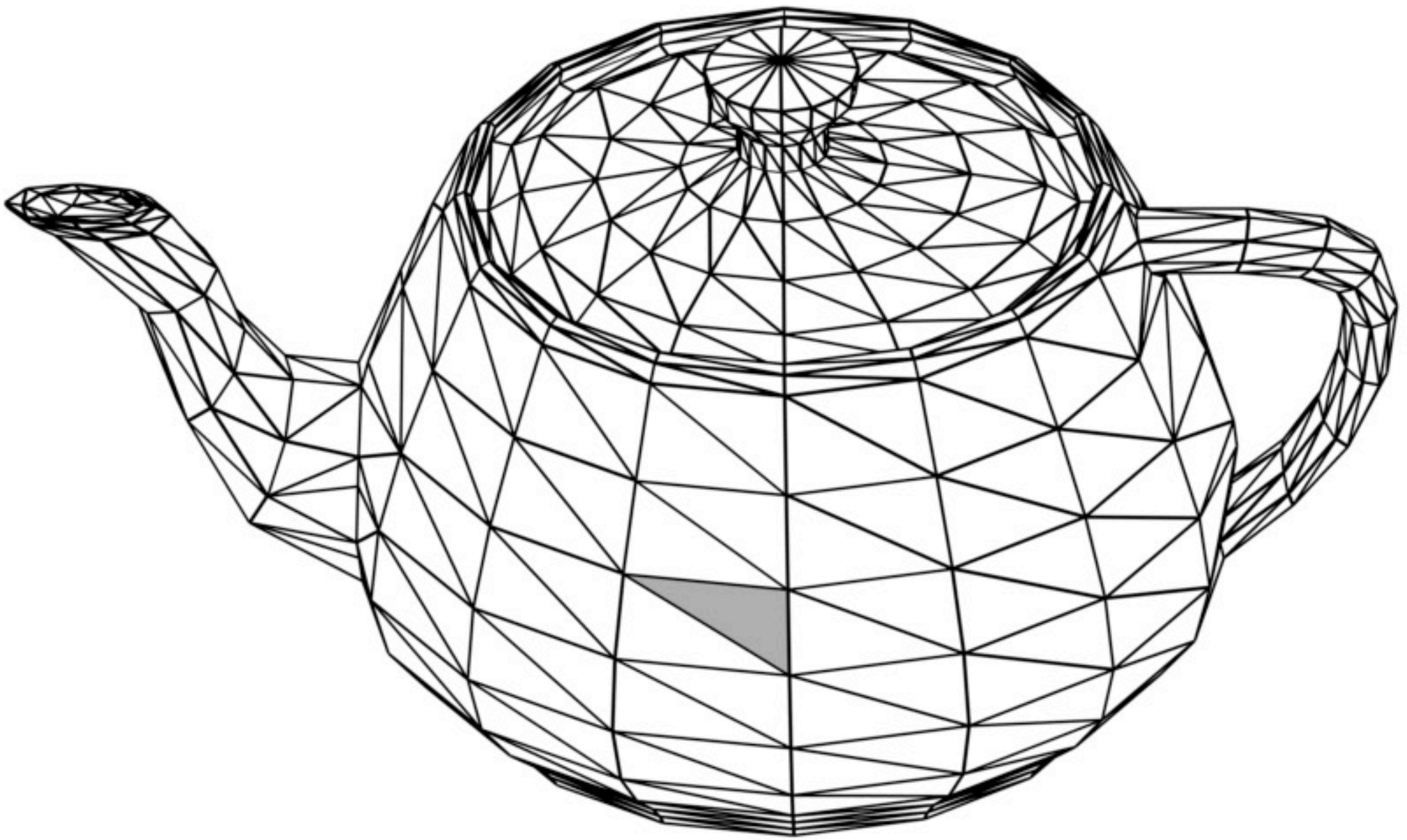
You should normalize **n**.

Once you have the face normals, you can calculate the light output for each face.

The problem is:
using just face normals
gives you **flat shading**.

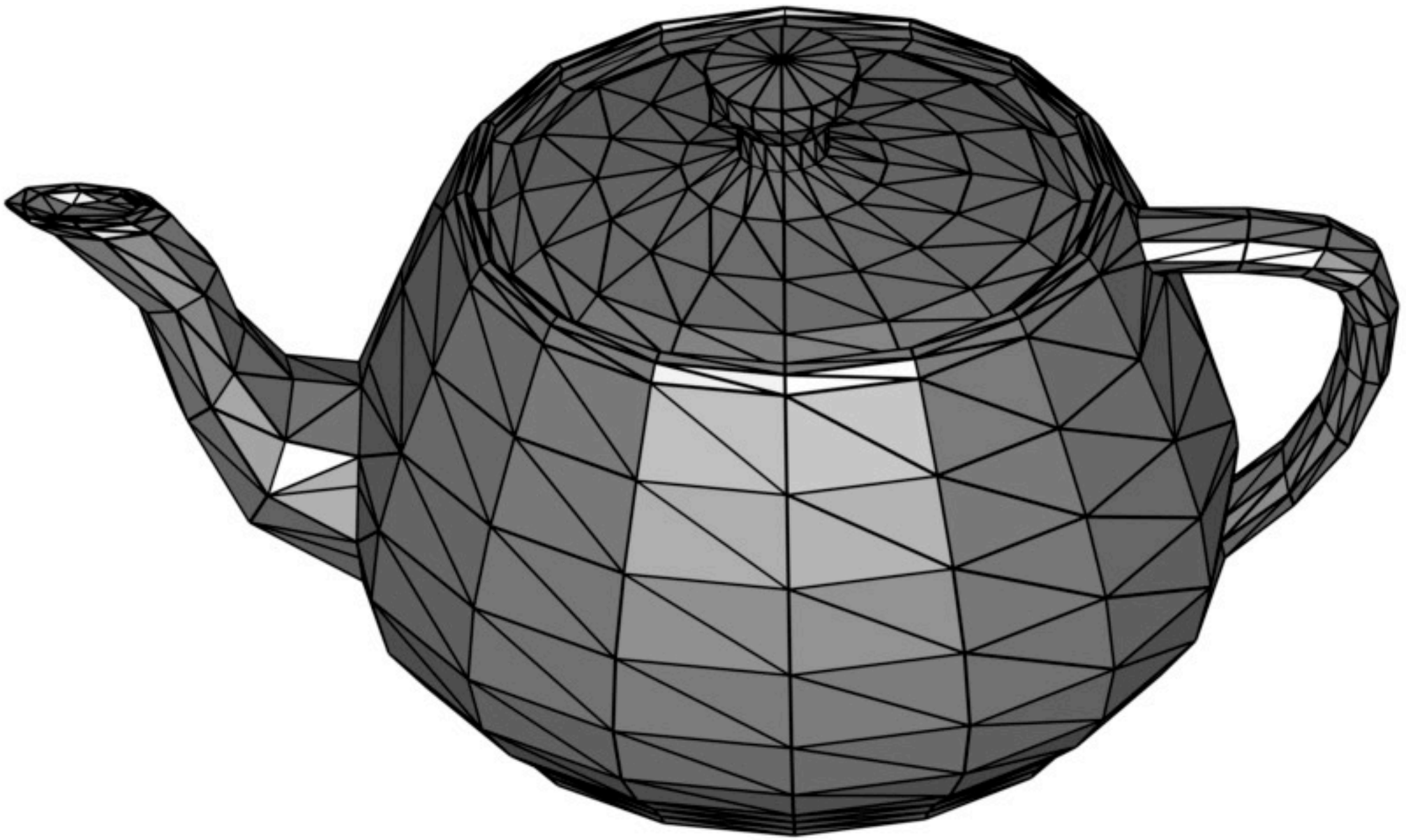
Flat Shading

All points in a face have the same normal, and get similar reflection.
Reflection is specified **per face**.

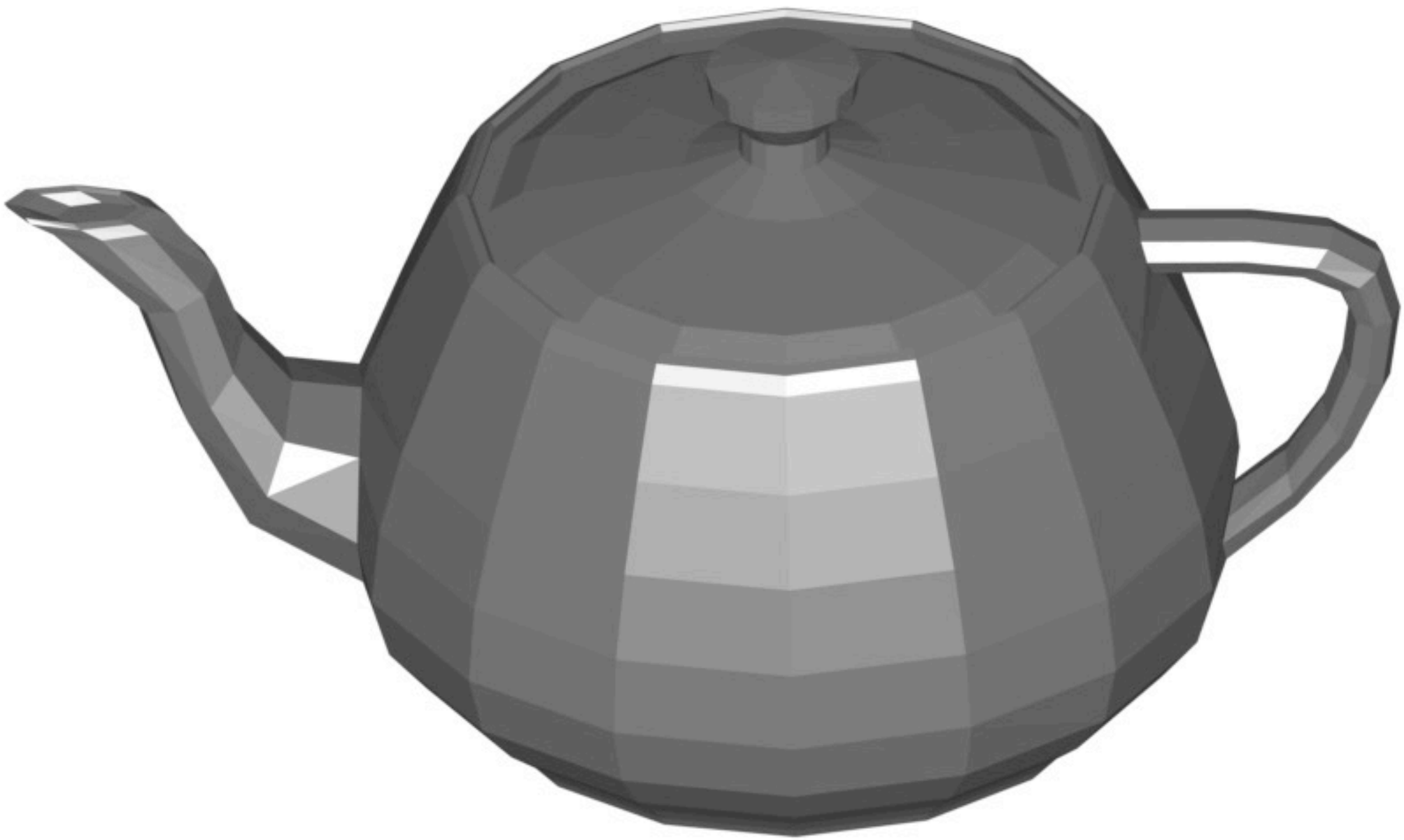


Flat Shading

All points in a face have the same normal, and get similar reflection. Adjacent face has a different normal and the reflection changes drastically across an edge. Result is unnatural.

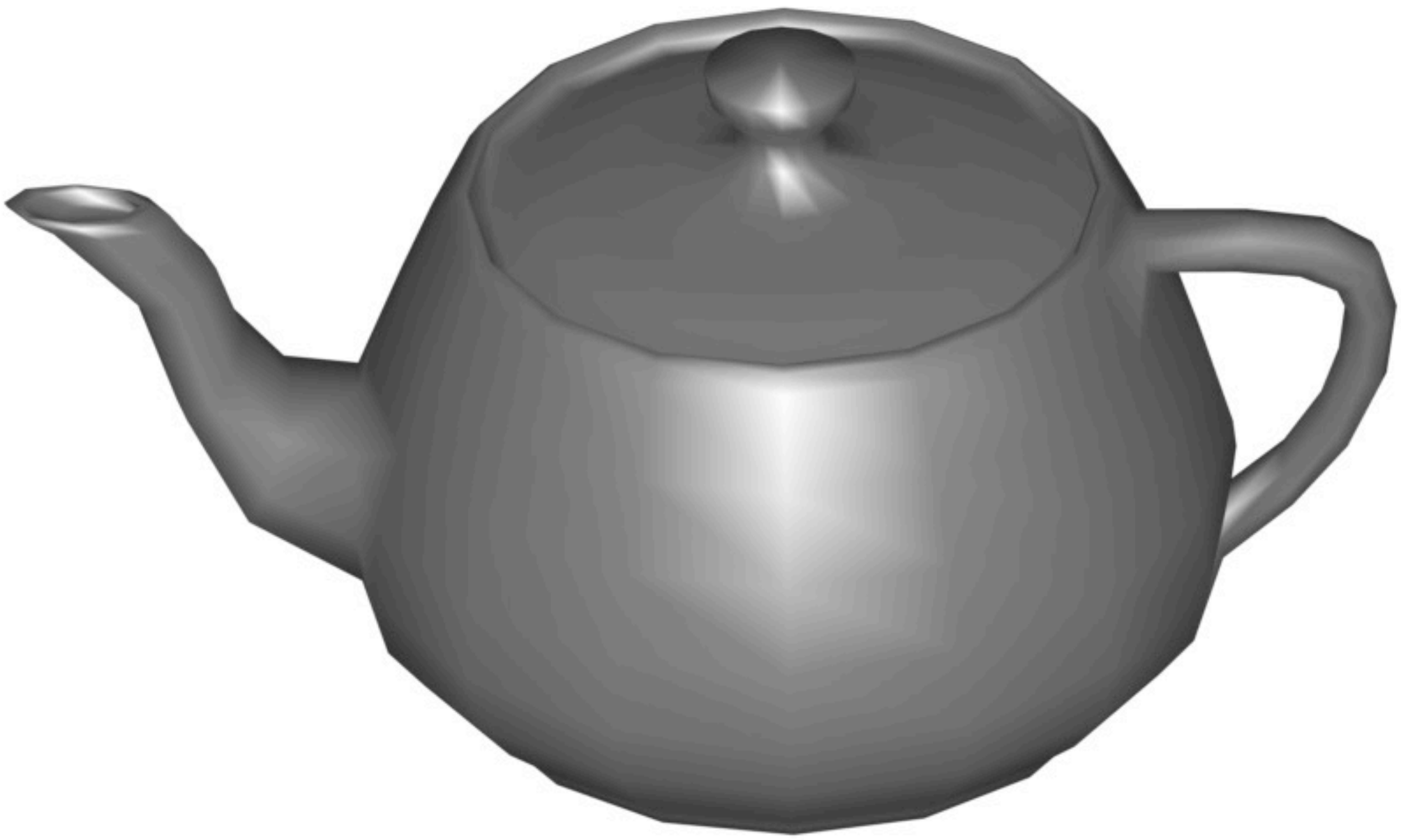


Flat Shading



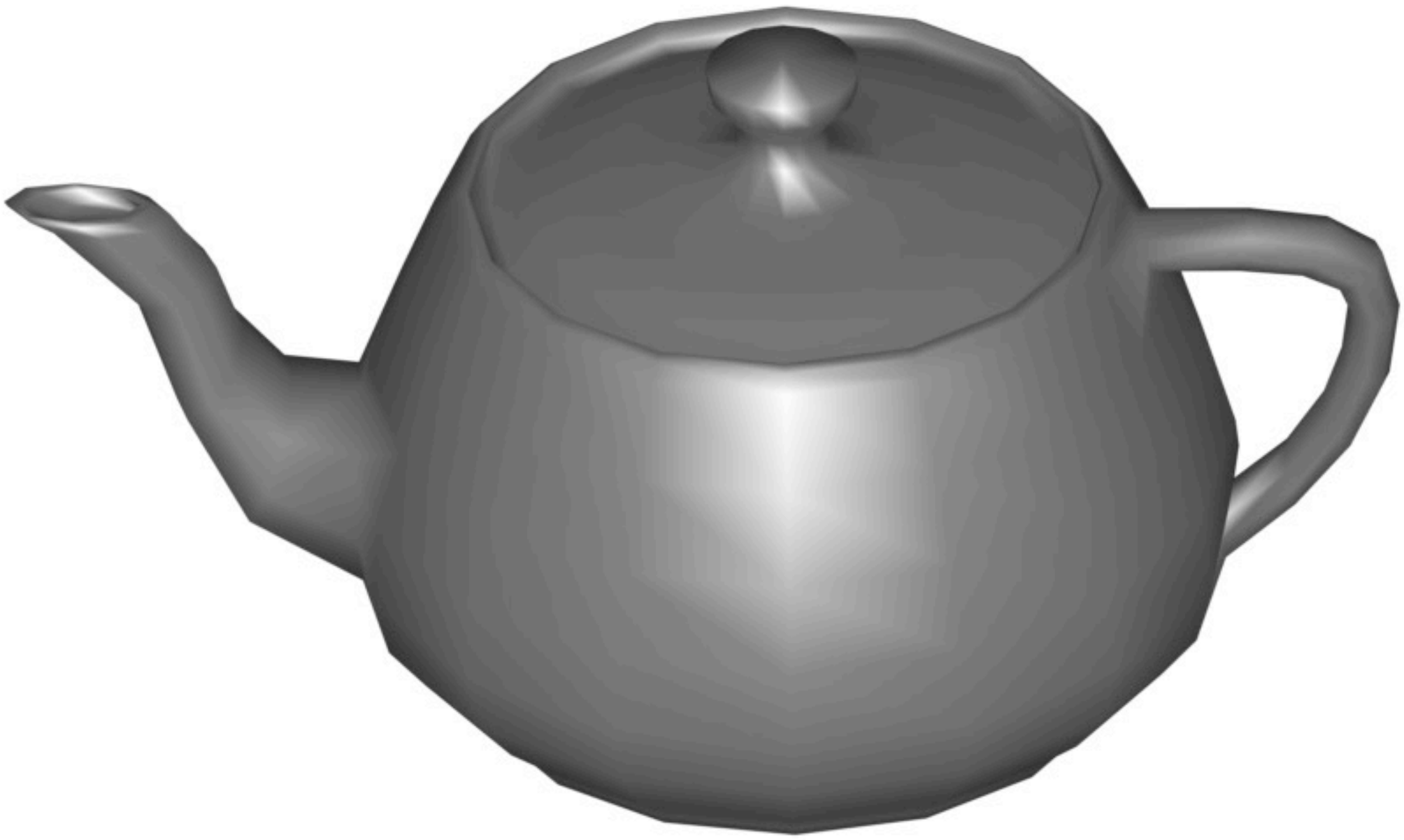
Interpolated Shading

Gives you a smoother result.



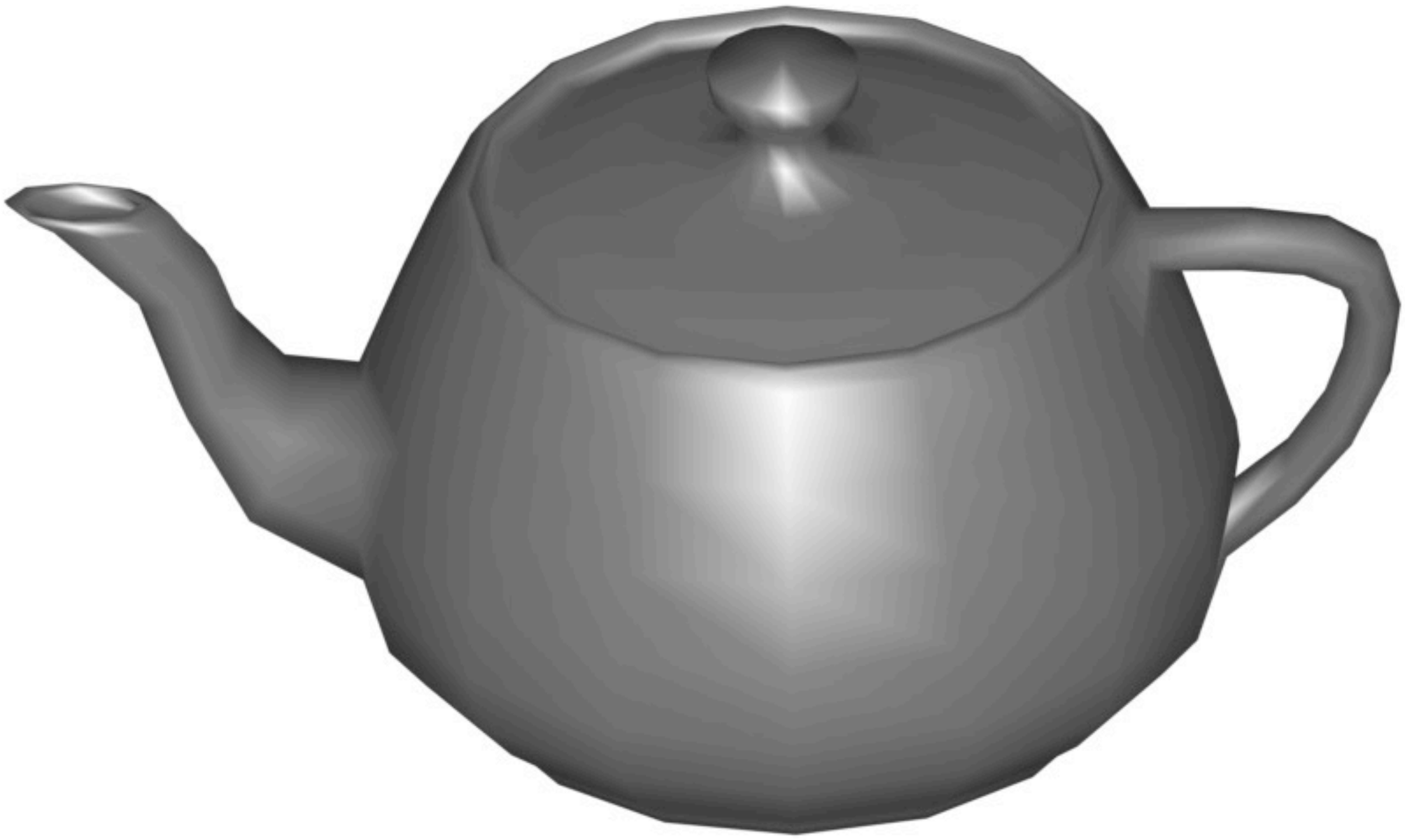
Interpolated Shading

Gives you a smoother result.
Reflection is specified **per vertex**.



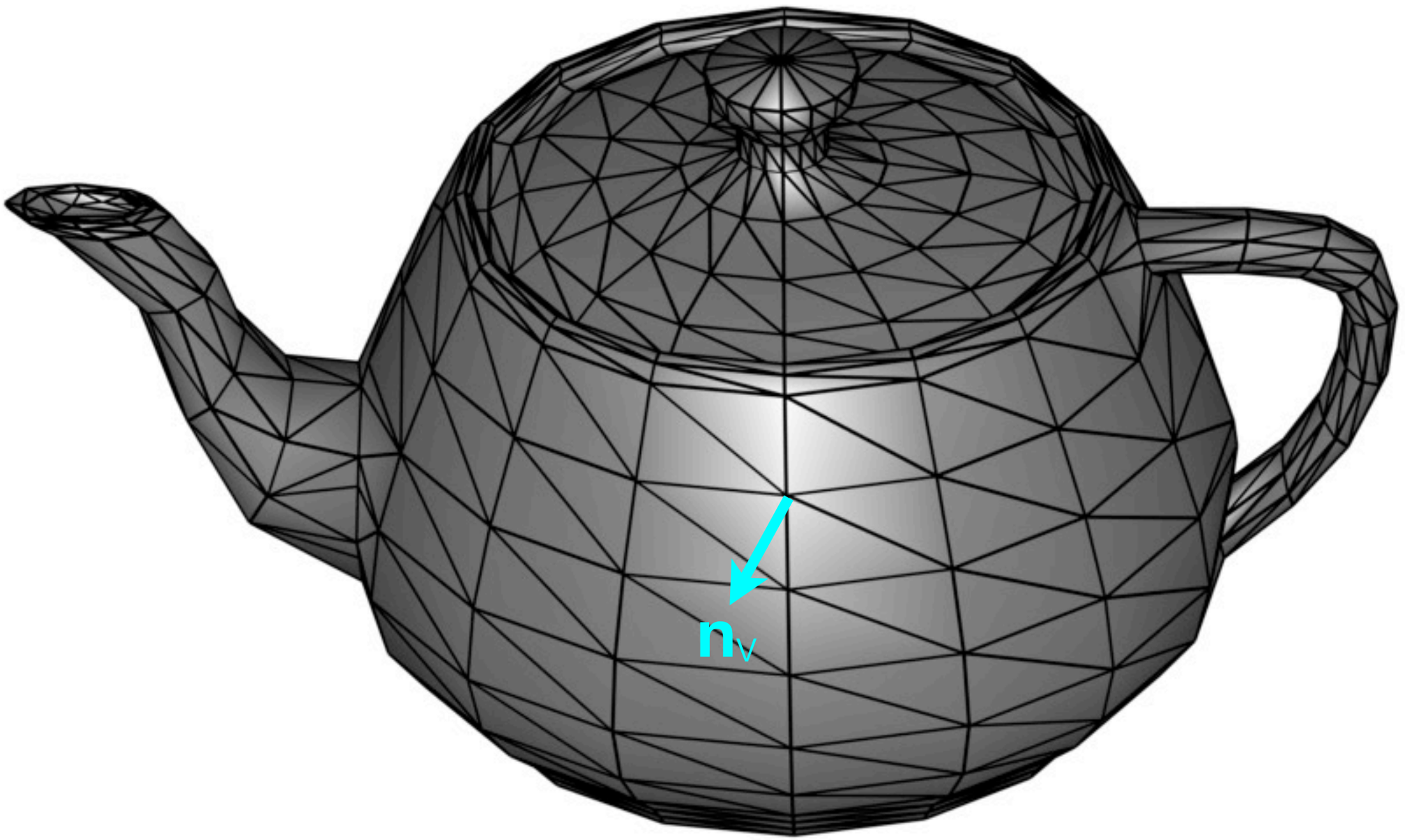
Interpolated Shading

Gives you a smoother result.
Reflection is specified **per vertex**.
Vertex colors are interpolated to face pixels.



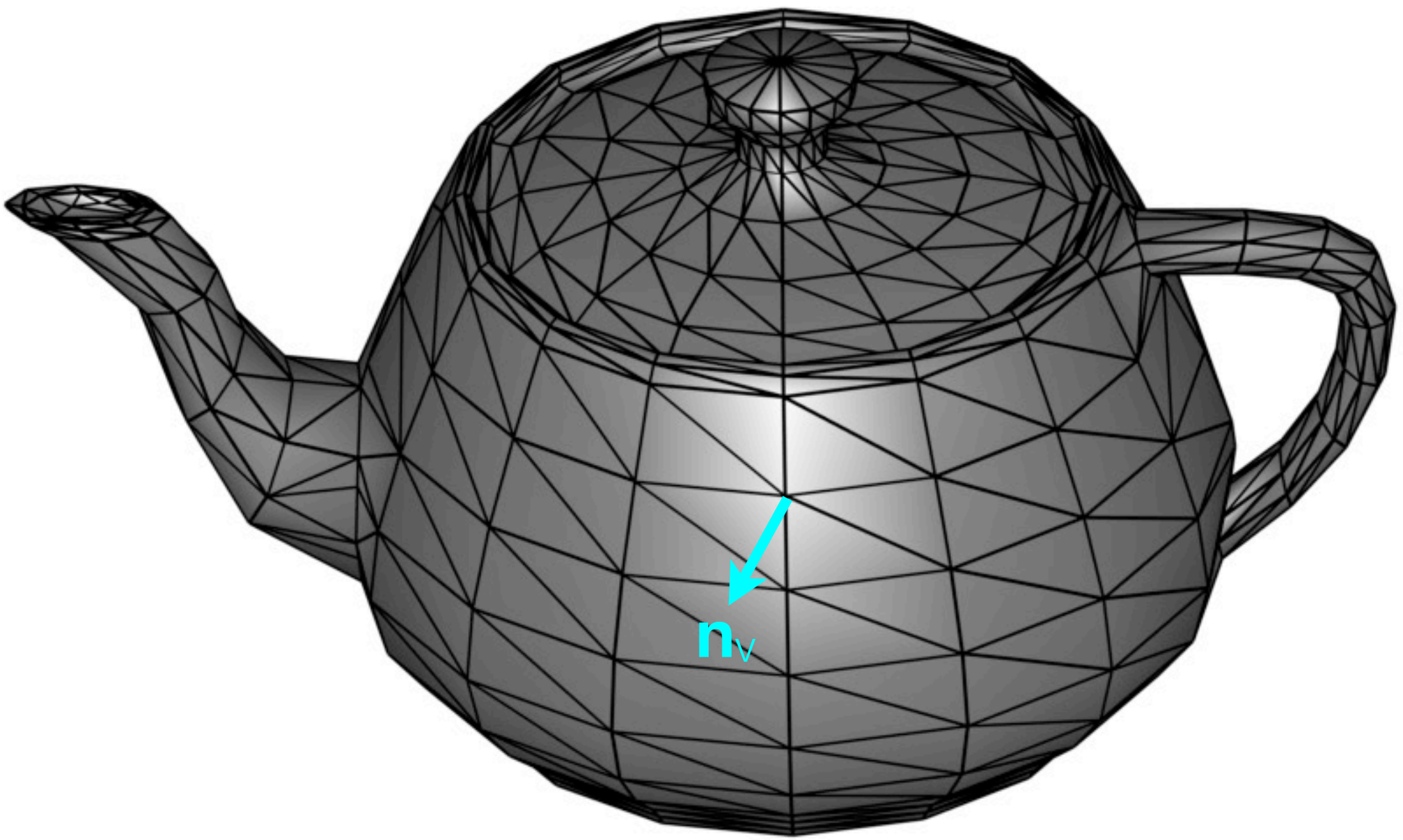
Interpolated Shading

We need **vertex normals** (normal at each vertex).



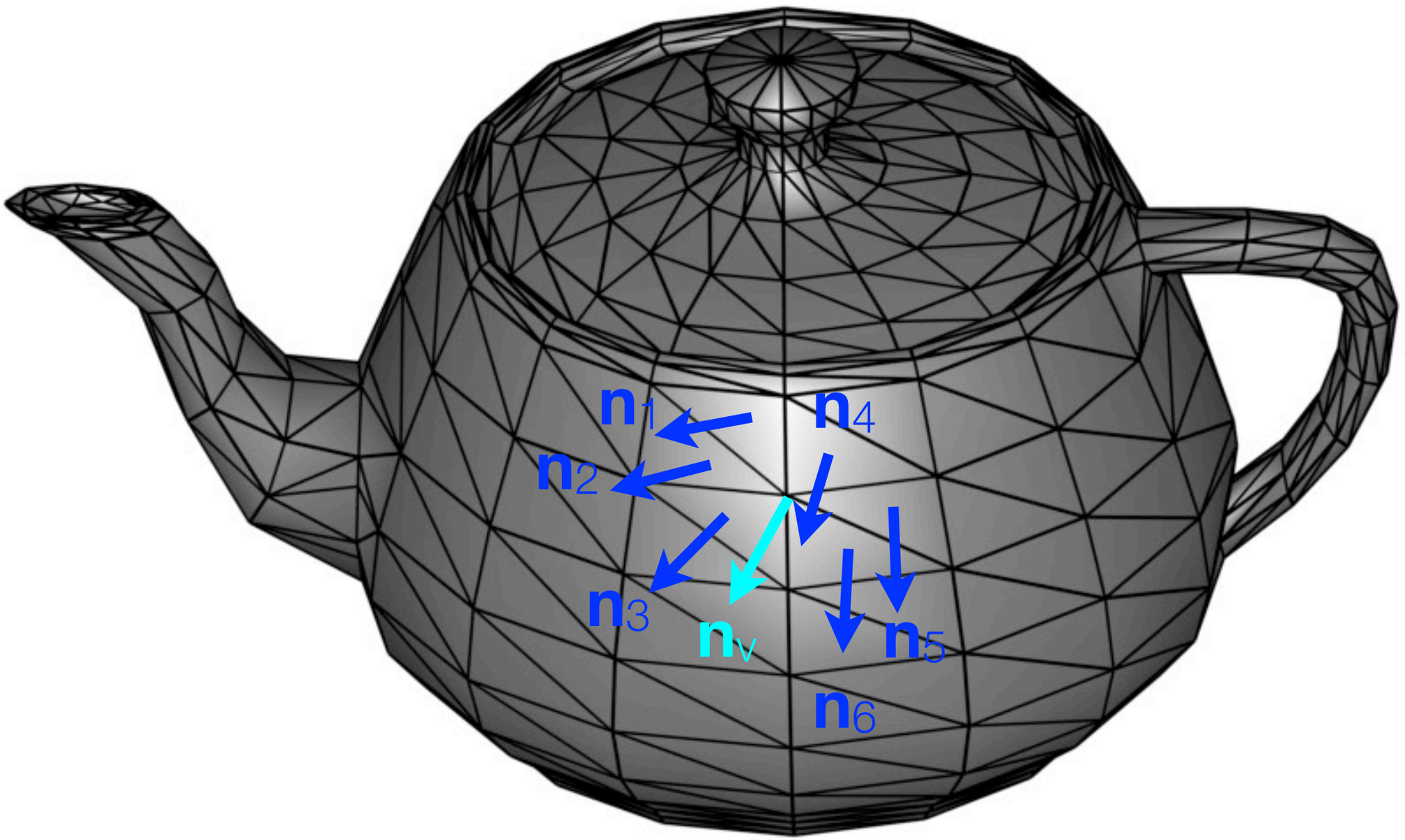
Interpolated Shading

How can you get the normal shown $\mathbf{n}_V$?



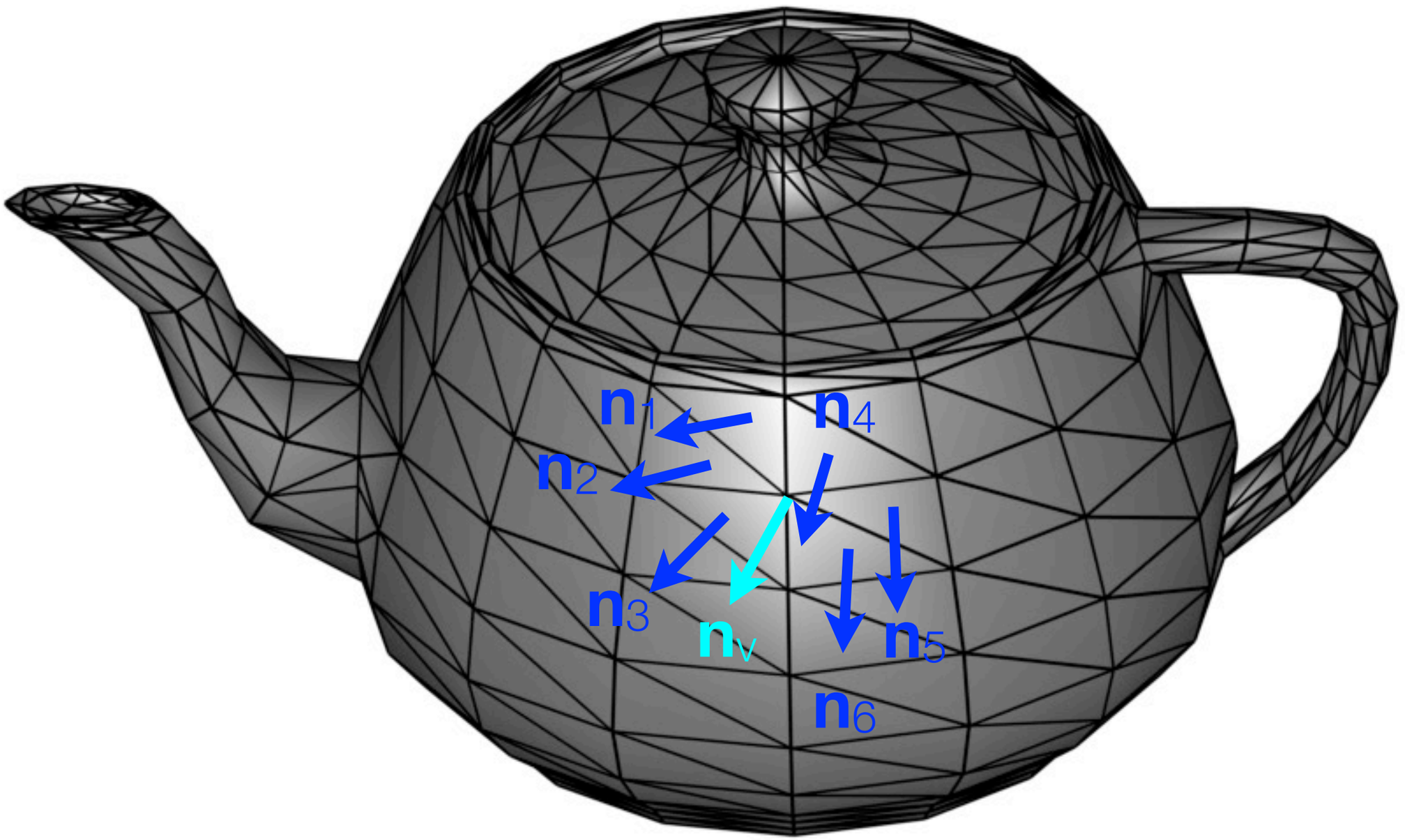
Interpolated Shading

Take average of all face normals that contain that vertex.



Interpolated Shading

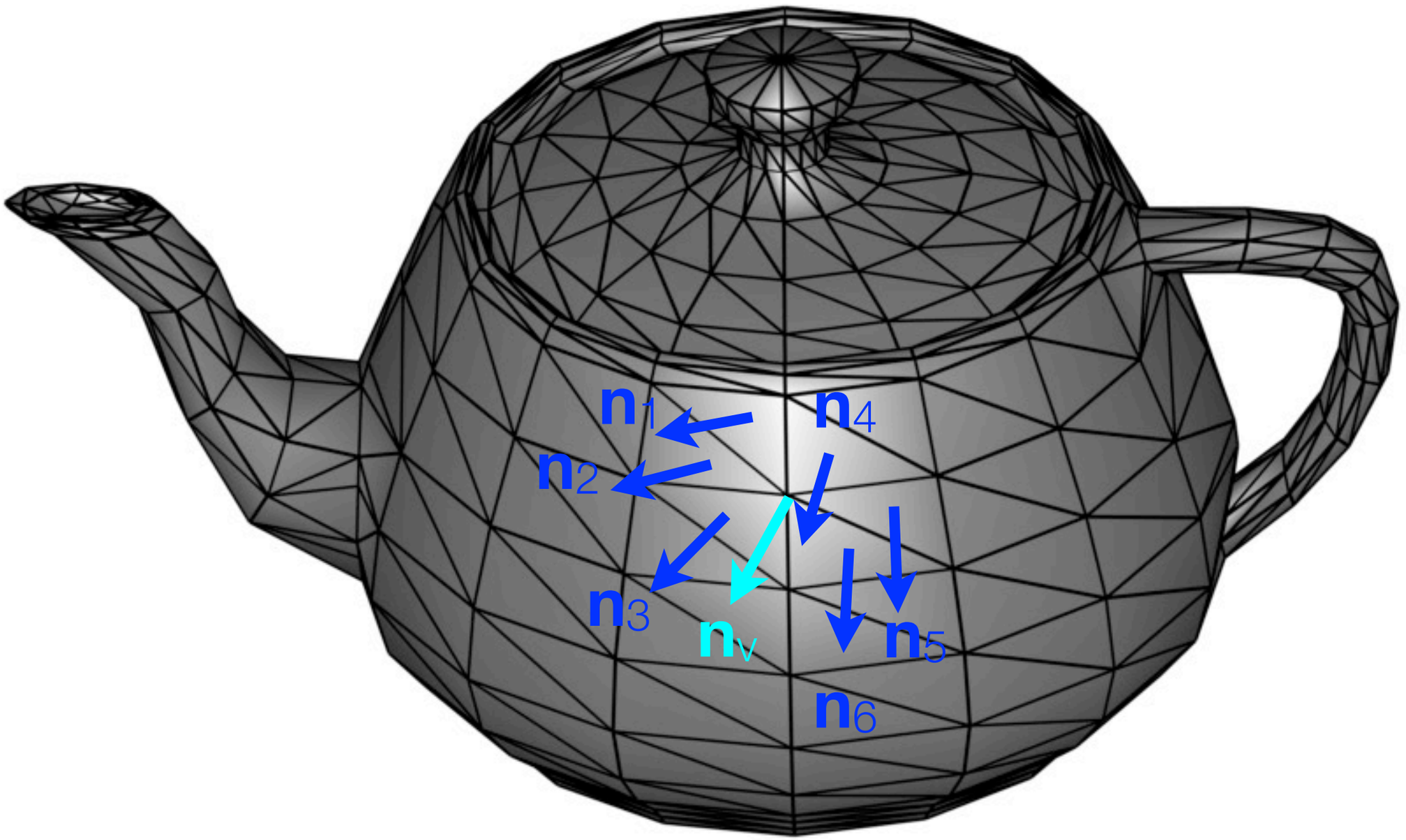
$$\mathbf{n}_v = \frac{\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4+\mathbf{n}_5+\mathbf{n}_6}{6}$$



Interpolated Shading

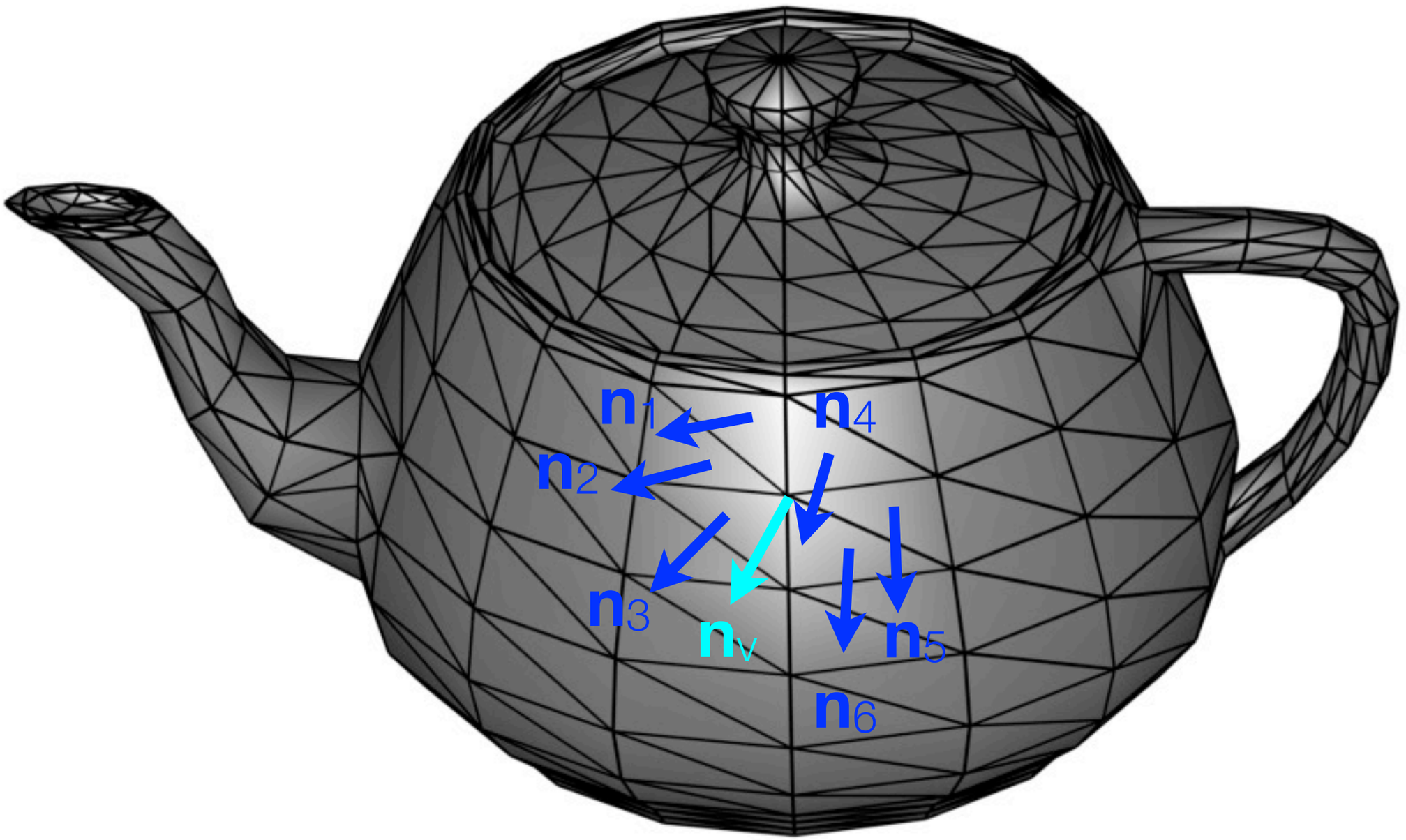
$$\mathbf{n}_v = \frac{\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4+\mathbf{n}_5+\mathbf{n}_6}{6}$$

Problem: $\mathbf{n}_v$ may not remain a unit vector.
So we need to normalize it.



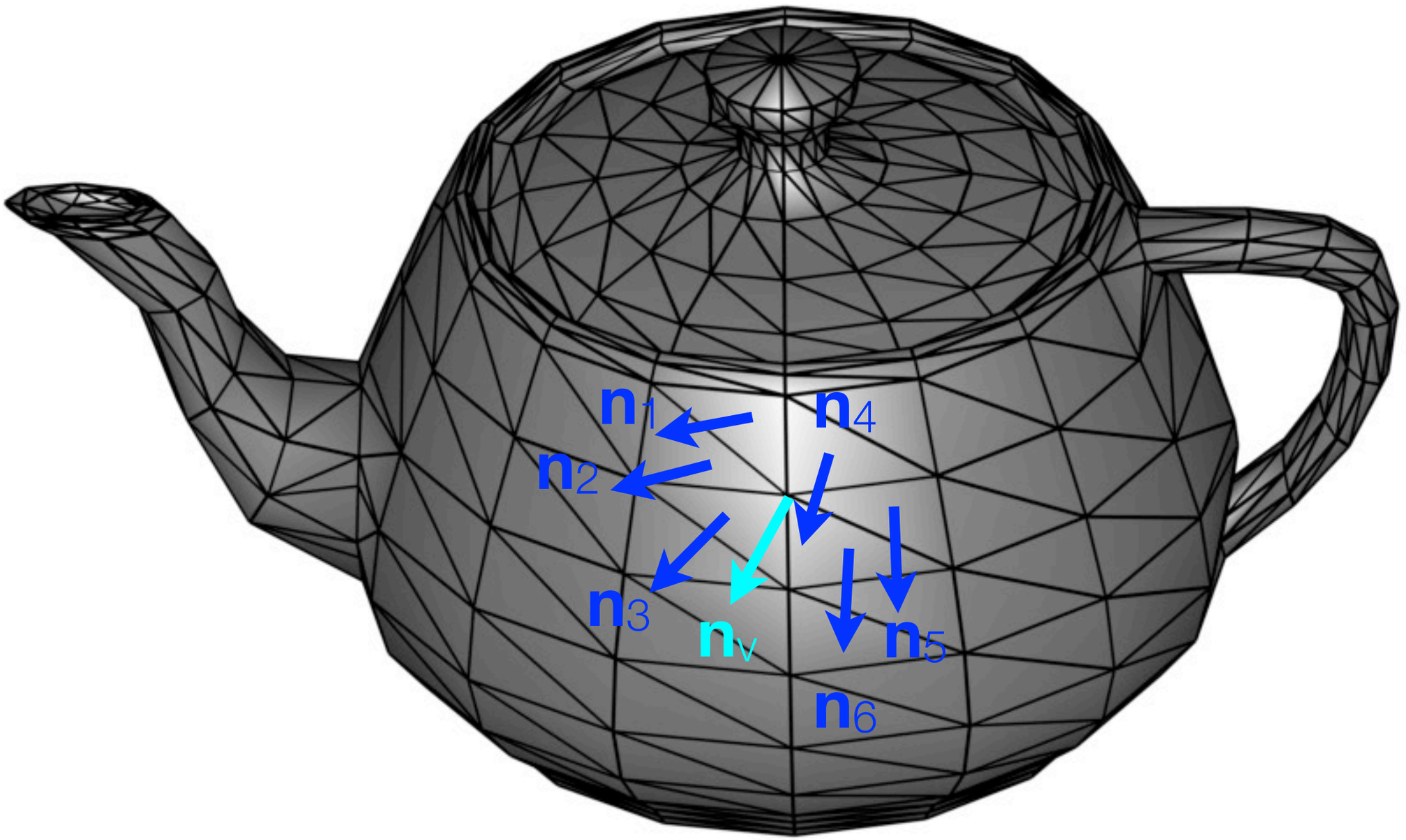
Interpolated Shading

$$\mathbf{n}_v = \frac{\frac{\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4+\mathbf{n}_5+\mathbf{n}_6}{6}}{\left\| \frac{\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4+\mathbf{n}_5+\mathbf{n}_6}{6} \right\|}$$



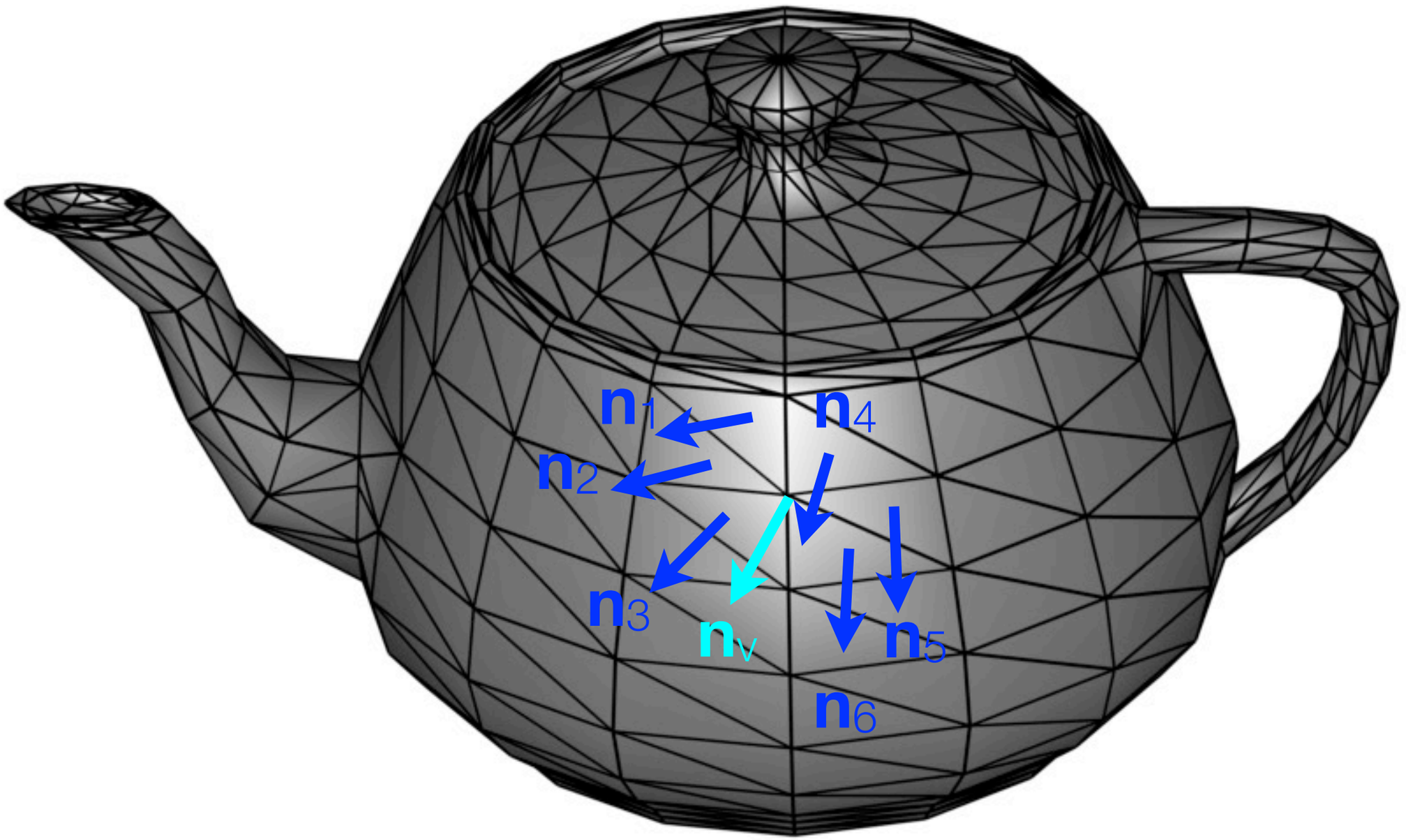
Interpolated Shading

$$\mathbf{n}_v = \frac{\frac{\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4+\mathbf{n}_5+\mathbf{n}_6}{\cancel{6}}}{\left\| \frac{\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4+\mathbf{n}_5+\mathbf{n}_6}{\cancel{6}} \right\|}$$



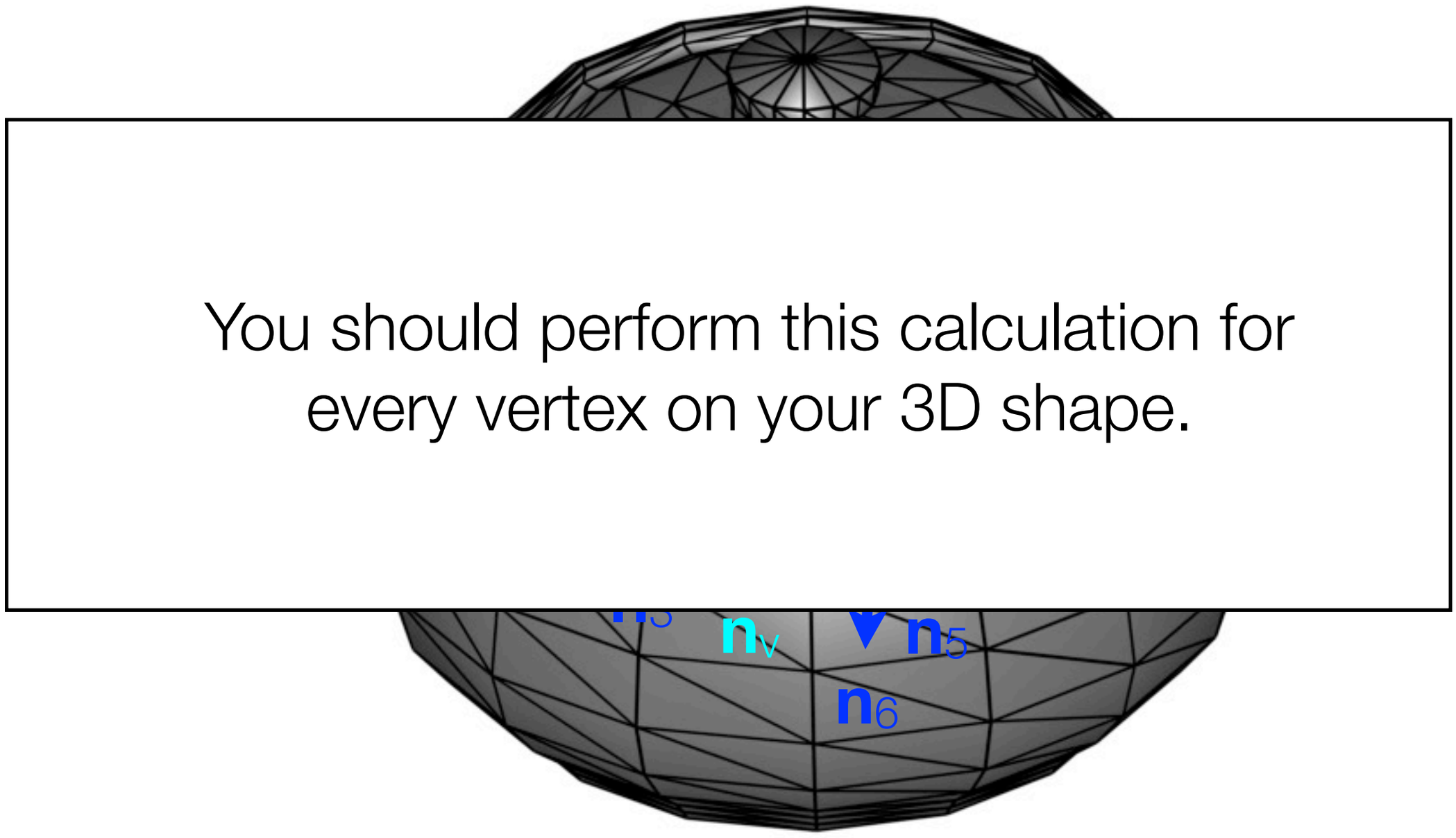
Interpolated Shading

$$\mathbf{n}_v = \frac{\frac{\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4+\mathbf{n}_5+\mathbf{n}_6}{6}}{\left\| \frac{\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4+\mathbf{n}_5+\mathbf{n}_6}{6} \right\|} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4 + \mathbf{n}_5 + \mathbf{n}_6}{\|\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4 + \mathbf{n}_5 + \mathbf{n}_6\|}$$



Interpolated Shading

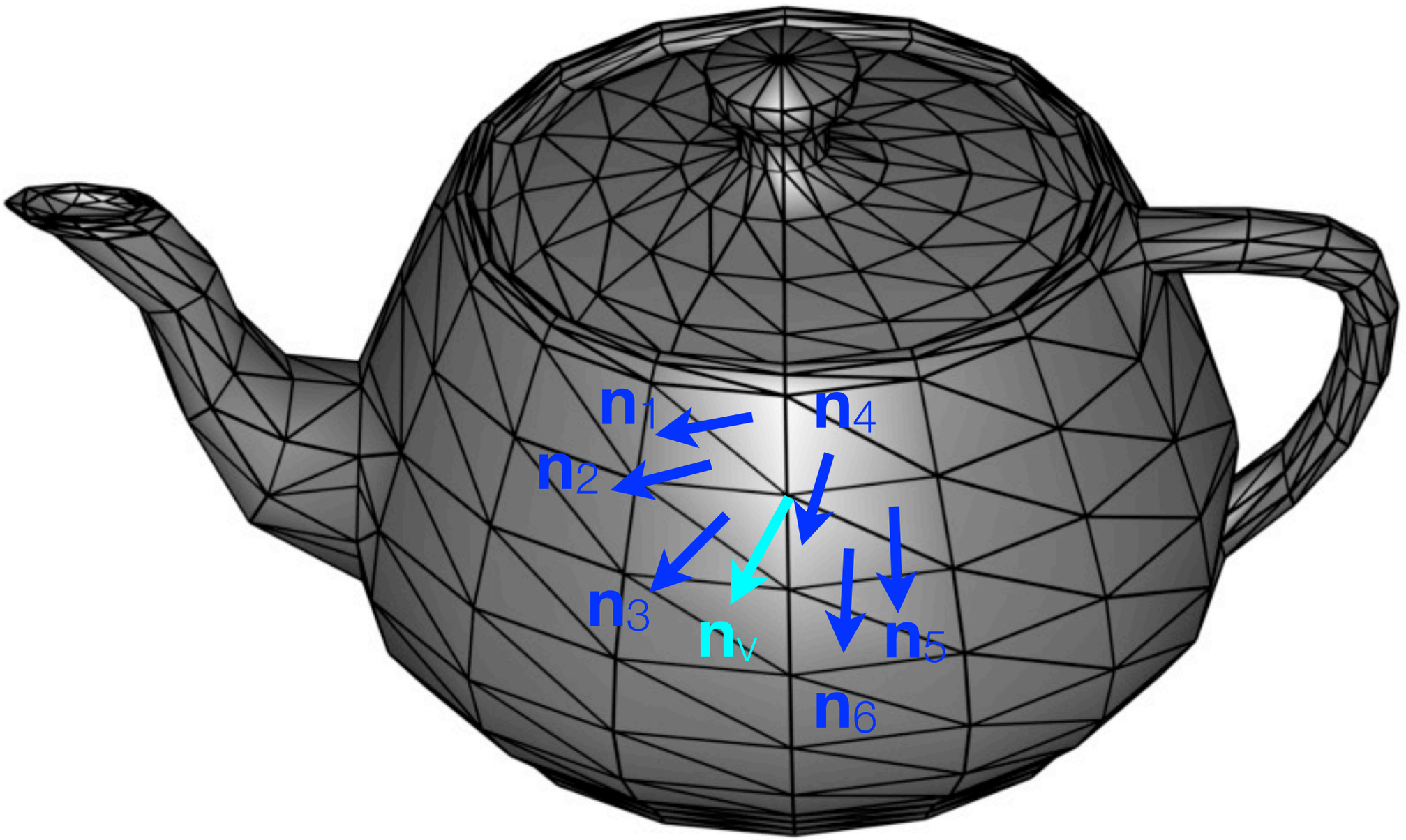
$$\mathbf{n}_v = \frac{\frac{\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4+\mathbf{n}_5+\mathbf{n}_6}{6}}{\left\| \frac{\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4+\mathbf{n}_5+\mathbf{n}_6}{6} \right\|} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4 + \mathbf{n}_5 + \mathbf{n}_6}{\|\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4 + \mathbf{n}_5 + \mathbf{n}_6\|}$$



You should perform this calculation for every vertex on your 3D shape.

Interpolated Shading

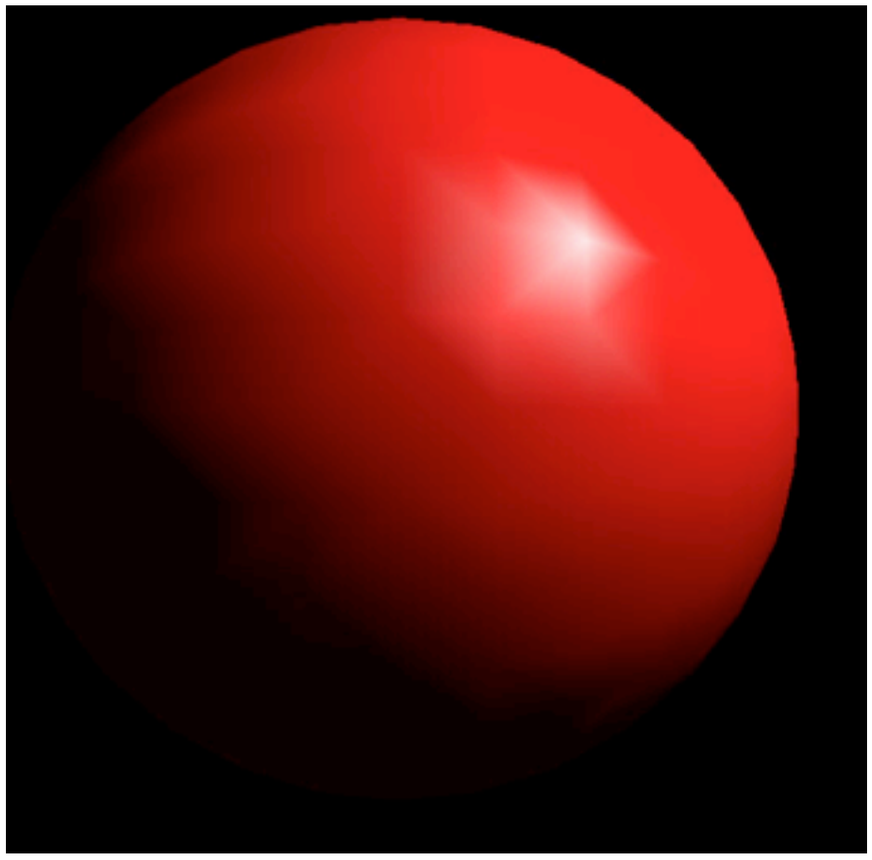
$$\mathbf{n}_v = \frac{\frac{\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4+\mathbf{n}_5+\mathbf{n}_6}{6}}{\left\| \frac{\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4+\mathbf{n}_5+\mathbf{n}_6}{6} \right\|} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4 + \mathbf{n}_5 + \mathbf{n}_6}{\|\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4 + \mathbf{n}_5 + \mathbf{n}_6\|}$$



Two Types of Interpolated Shading

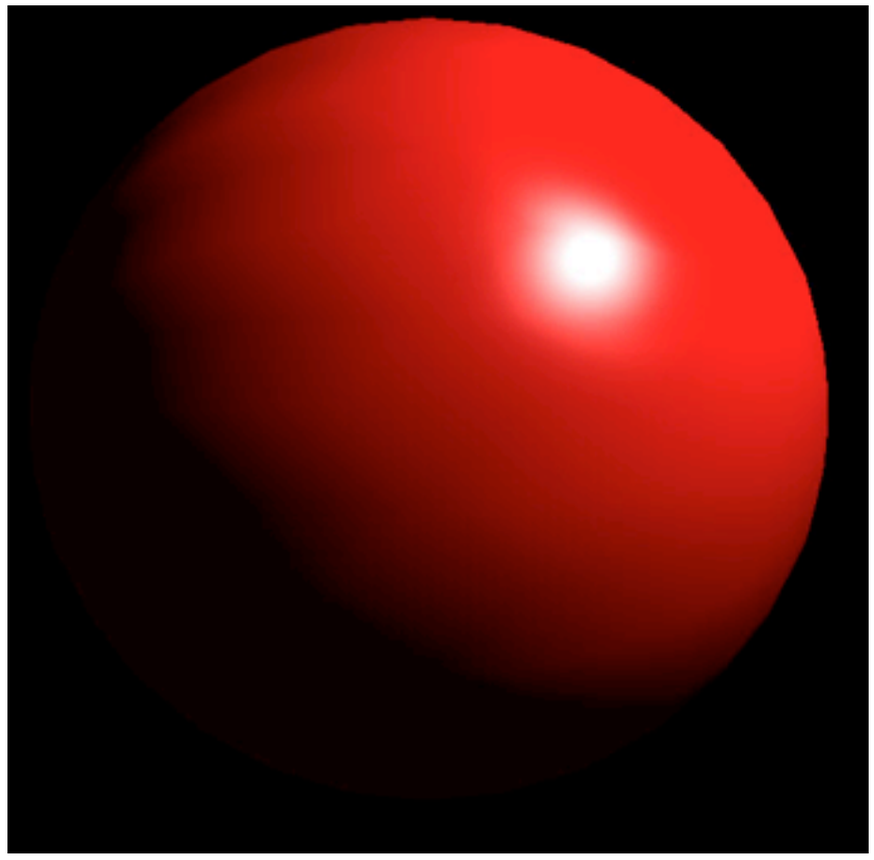
Gouraud Shading

Interpolates Colors



Phong Shading

Interpolates Normals



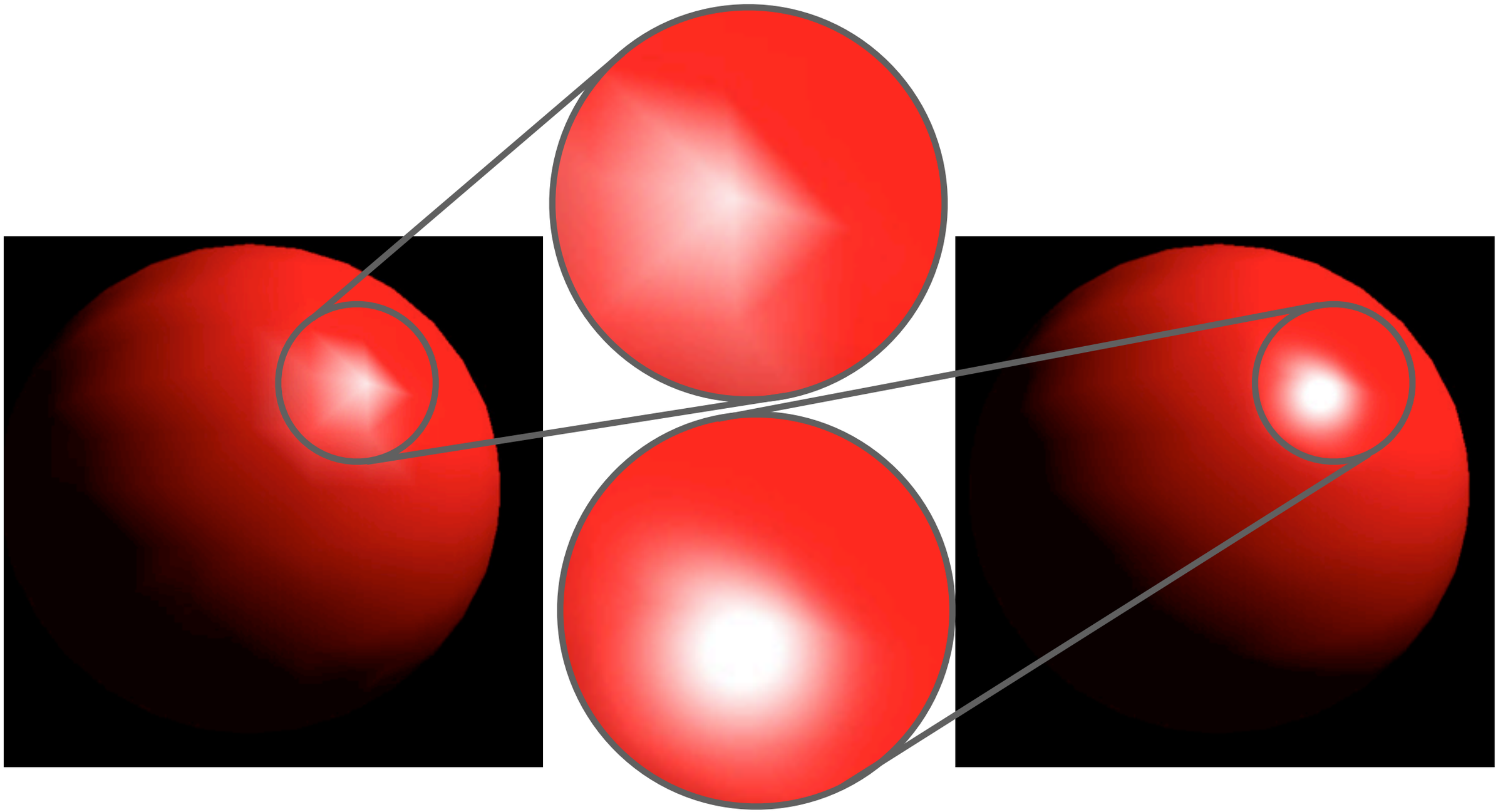
Two Types of Interpolated Shading

Gouraud Shading

Interpolates Colors

Phong Shading

Interpolates Normals



Two Types of Interpolated Shading

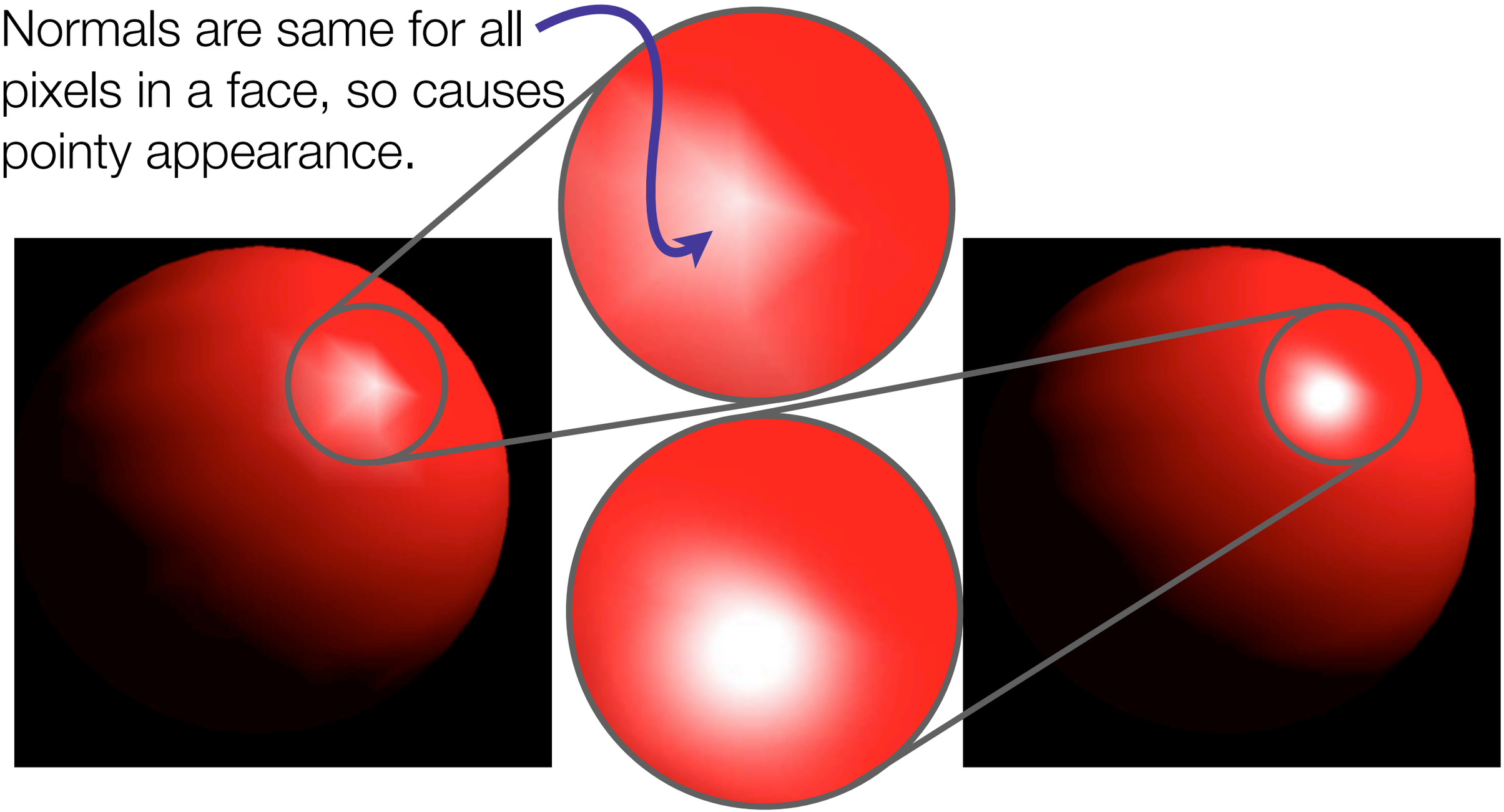
Gouraud Shading

Interpolates Colors

Phong Shading

Interpolates Normals

Normals are same for all pixels in a face, so causes pointy appearance.



Two Types of Interpolated Shading

Gouraud Shading

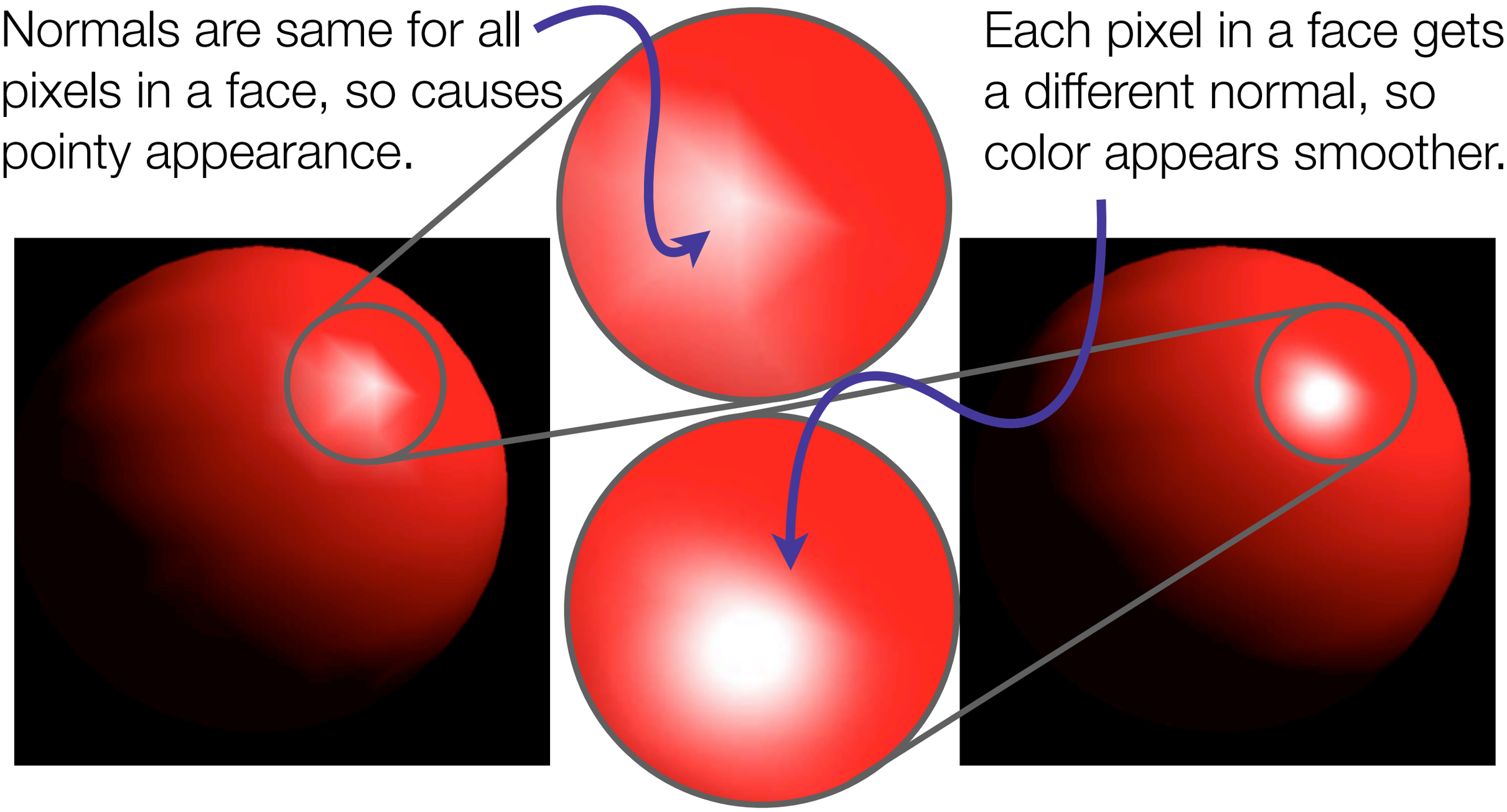
Interpolates Colors

Normals are same for all pixels in a face, so causes pointy appearance.

Phong Shading

Interpolates Normals

Each pixel in a face gets a different normal, so color appears smoother.



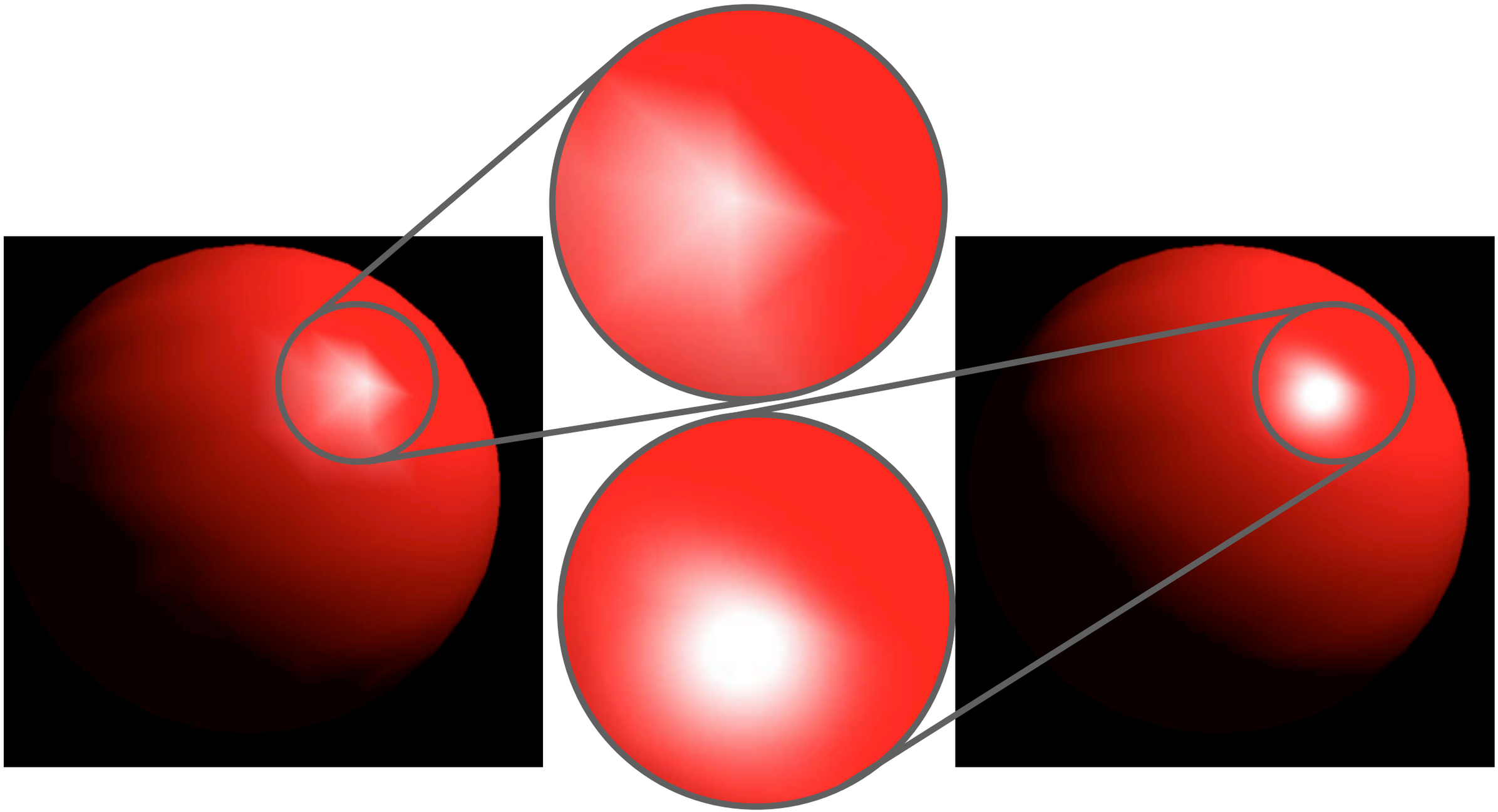
Two Types of Interpolated Shading

Gouraud Shading

Interpolates Colors

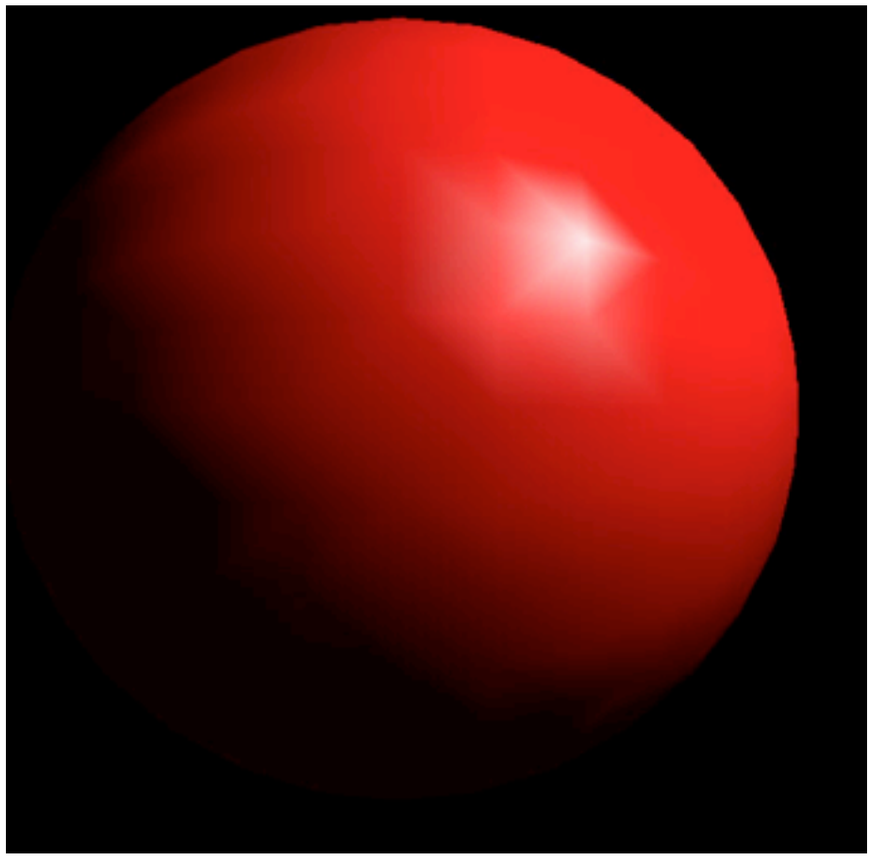
Phong Shading

Interpolates Normals

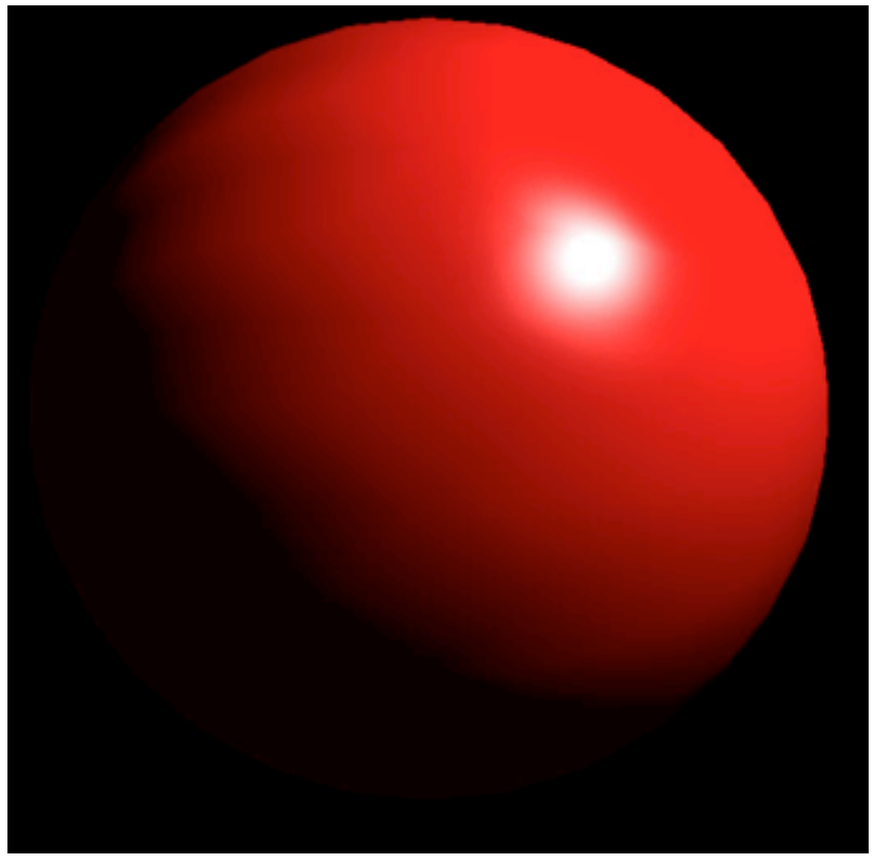


Two Types of Interpolated Shading

Gouraud Shading



Phong Shading

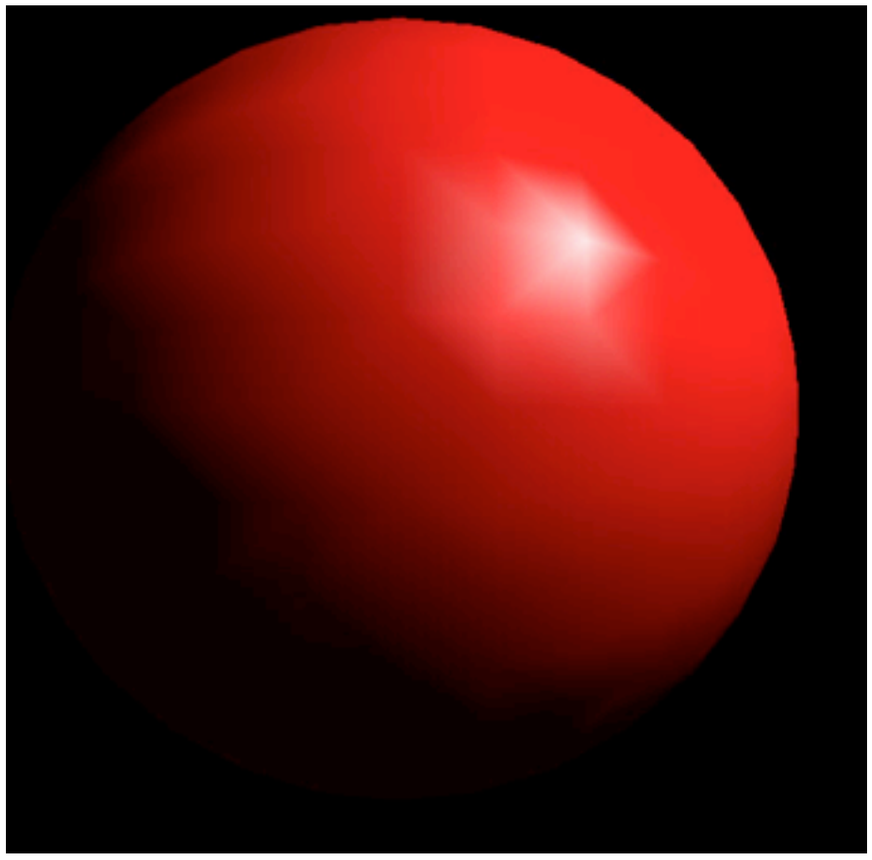


Two Types of Interpolated Shading

Gouraud Shading

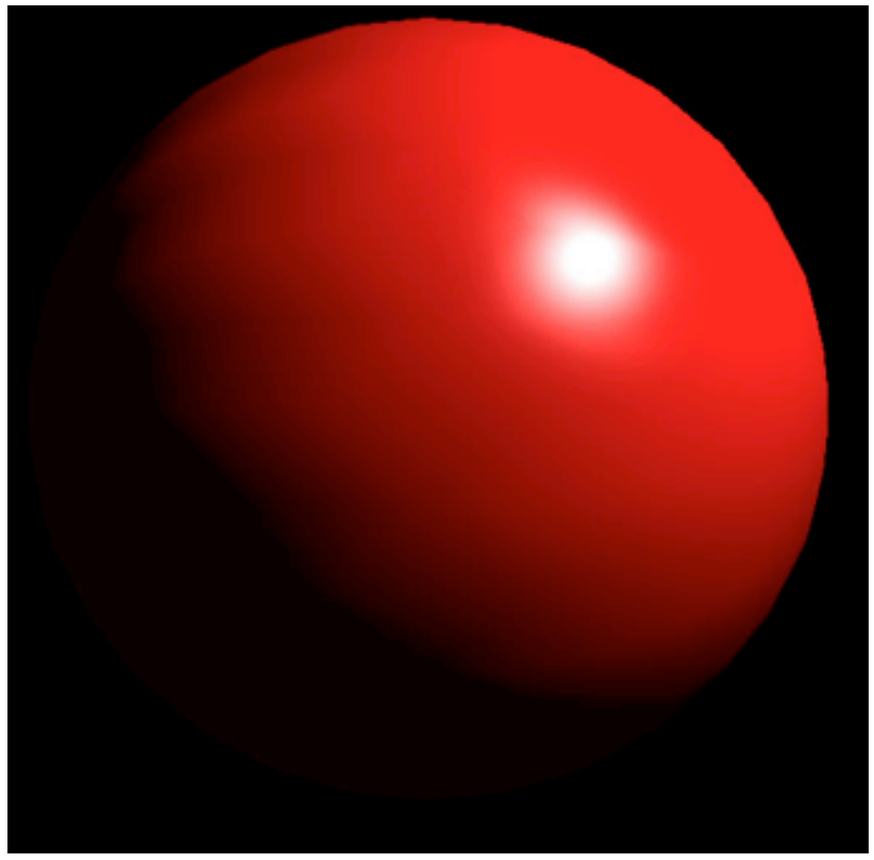
Compute Color
in VS.

Interpolate Color
from VS to FS.



VS = Vertex Shader
FS = Fragment Shader

Phong Shading



Two Types of Interpolated Shading

Gouraud Shading

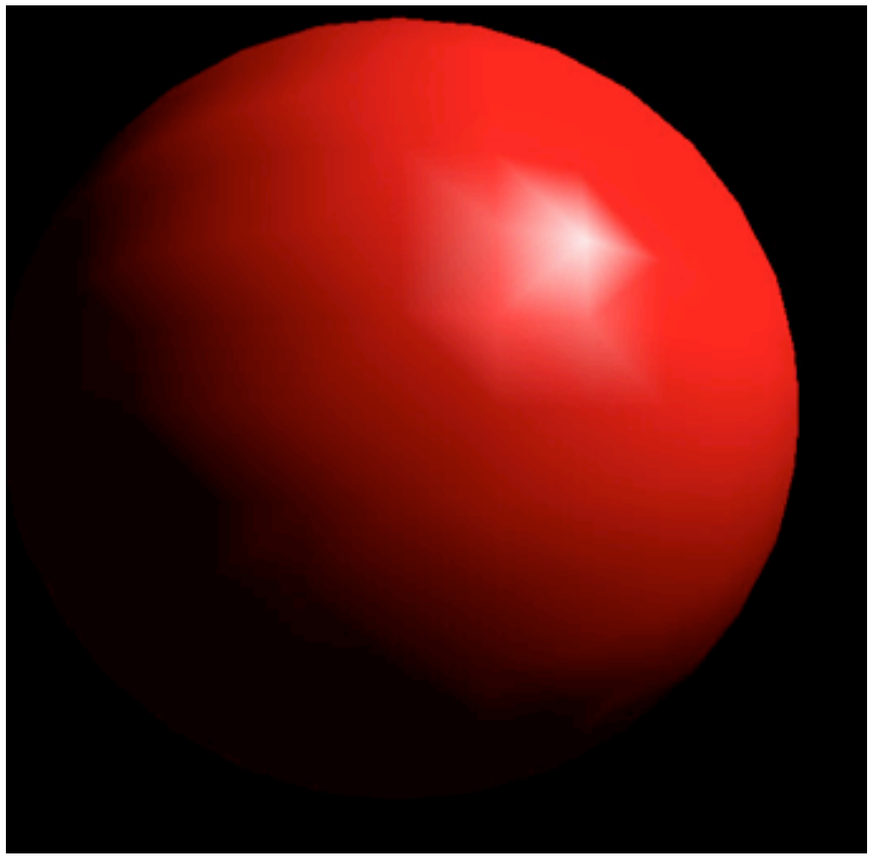
Compute Color
in VS.

Interpolate Color
from VS to FS.

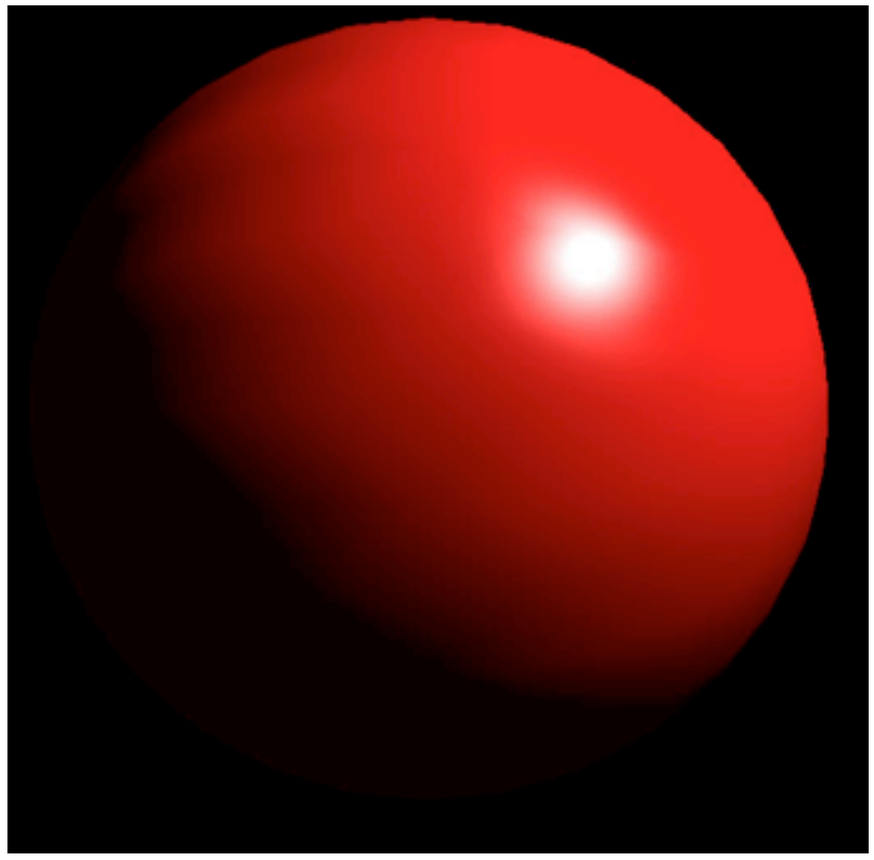
Phong Shading

Interpolate Normal
from VS to FS.

Compute Color
in FS.



VS = Vertex Shader
FS = Fragment Shader



Two Types of Interpolated Shading

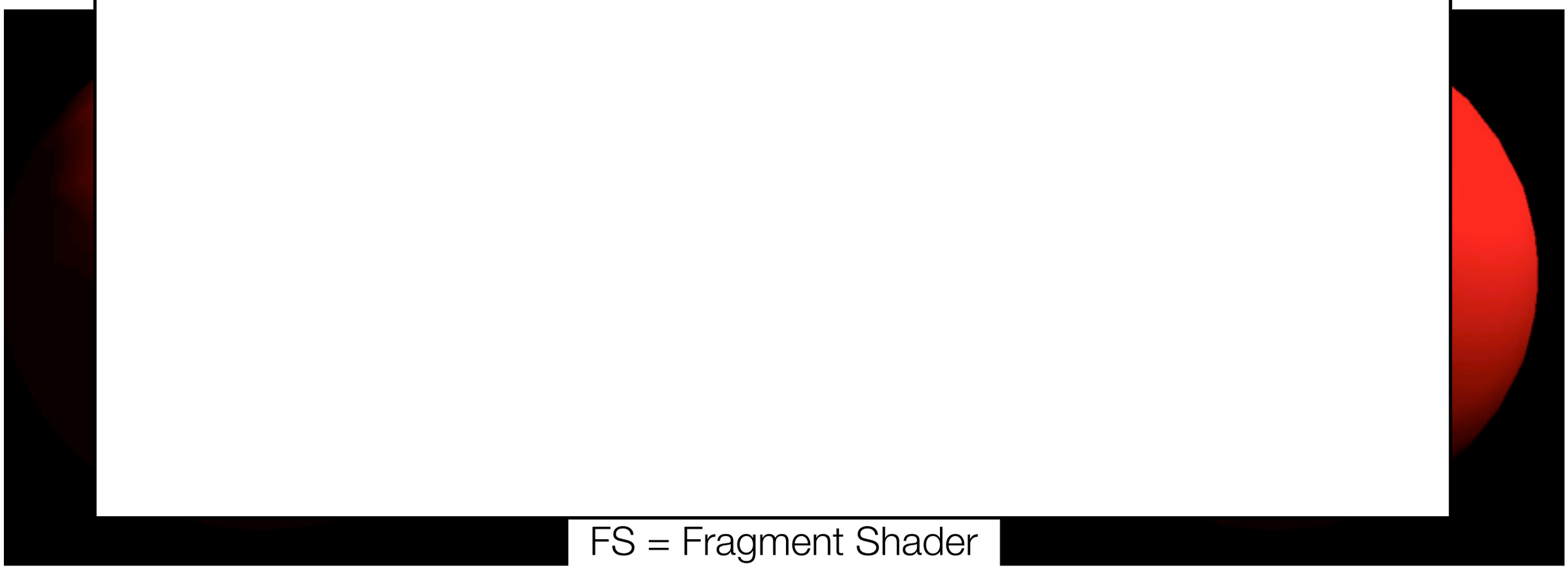
Gouraud Shading

Compute Color
in
In
fr

Phong Shading

Interpolate Normal

Phong shading appears more accurate.



FS = Fragment Shader

Two Types of Interpolated Shading

Gouraud Shading

Compute Color
in
In
fr

Phong Shading

Interpolate Normal

Phong shading appears more accurate.

Which is generally slower?

FS = Fragment Shader

Two Types of Interpolated Shading

Gouraud Shading

Compute Color
in
In
fr

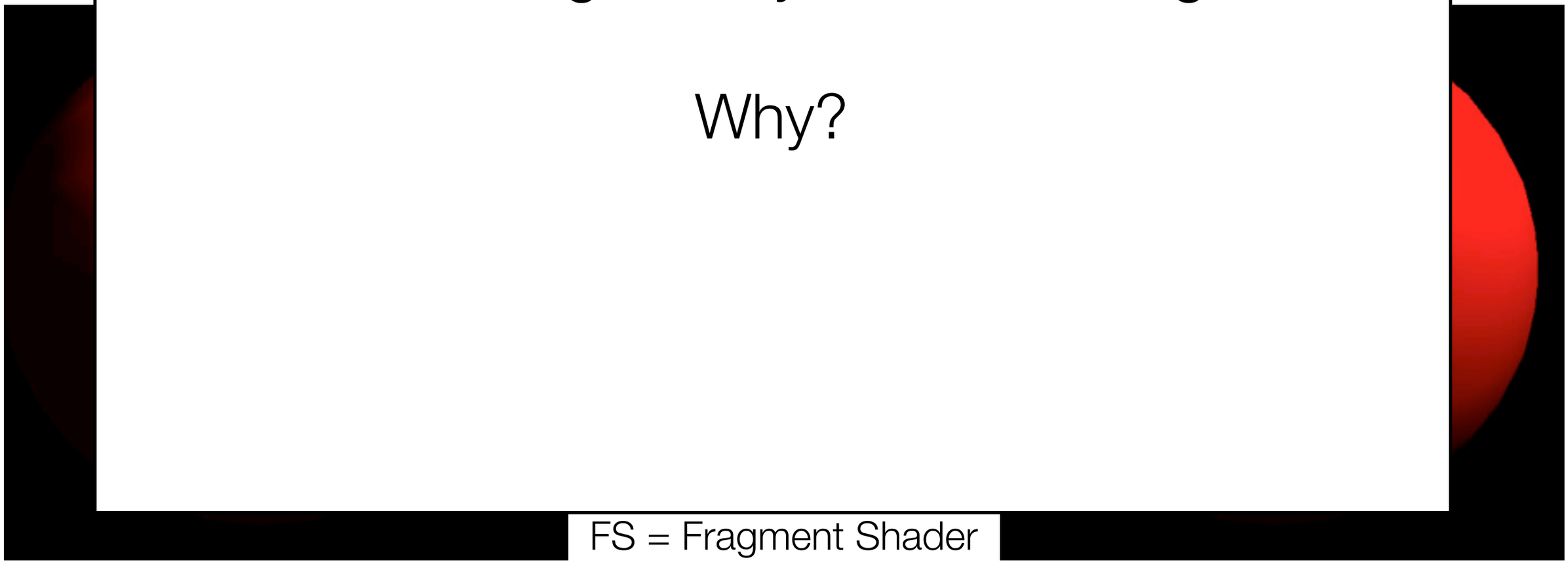
Phong Shading

Interpolate Normal

Phong shading appears more accurate.

Which is generally slower: Phong.

Why?



FS = Fragment Shader

Two Types of Interpolated Shading

Gouraud Shading

Compute Color
in
In
fr

Phong Shading

Interpolate Normal

Phong shading appears more accurate.

Which is generally slower: Phong.

Phong does lighting calculations per fragment.
Gouraud does lighting calculations per vertex.

For most shapes you will see,
there are way more fragments than vertices.

FS = Fragment Shader

Two Types of Interpolated Shading

Gouraud Shading

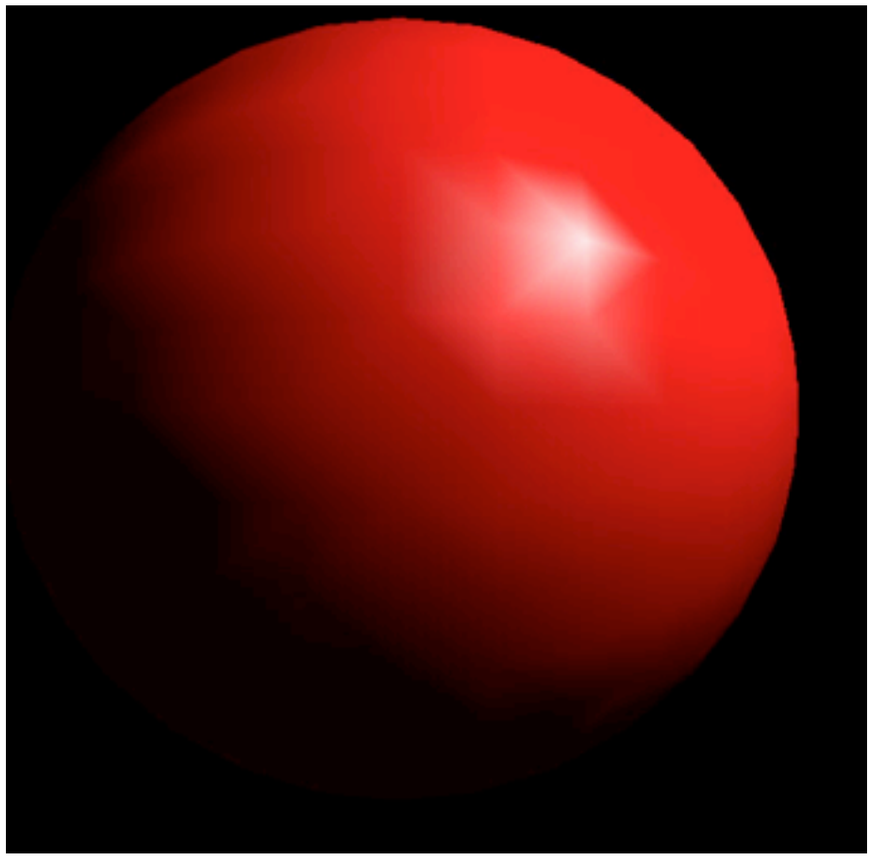
Compute Color
in VS.

Interpolate Color
from VS to FS.

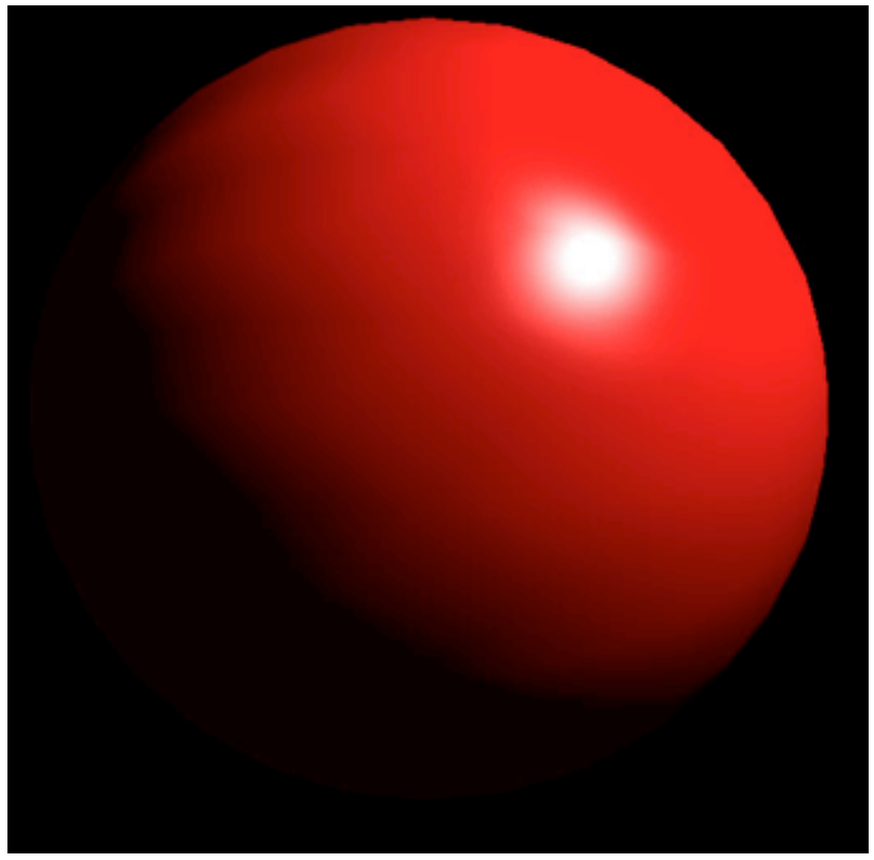
Phong Shading

Interpolate Normal
from VS to FS.

Compute Color
in FS.



VS = Vertex Shader
FS = Fragment Shader



WebGL examples

Lighting computations are some of the most mathematically intensive part of your shaders.

Let’s look at Gouraud Shading first. For these examples, we will use a point light source.

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// attenuated incident light components
vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
vec3 i, v;

// final reflected light
// (interpolated to fragment shader)
varying vec3 R;

mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

Global Variables for Vertex Shader

main() Function for Vertex Shader

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);

    vec3 Ra, Rd, Rs; // reflected light components

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
```

Global Variables for
Vertex Shader

```
void main() {
}
```

main() Function for
Vertex Shader

```
precision mediump float;
```

```
void main() {

}
```

```
precision mediump float;


mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

```
void main() {
    gl_PointSize = 1.0;









    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal
```

```
mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

```
void main() {
    gl_PointSize = 1.0;
```

```
    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;




// attenuated incident light components
vec3 Ia_pp0, Id_pp0, Is_pp0;



mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);









    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;
// coefficients for object
uniform vec3 ka, kd, ks;



// attenuated incident light components
vec3 Ia_pp0, Id_pp0, Is_pp0;
```

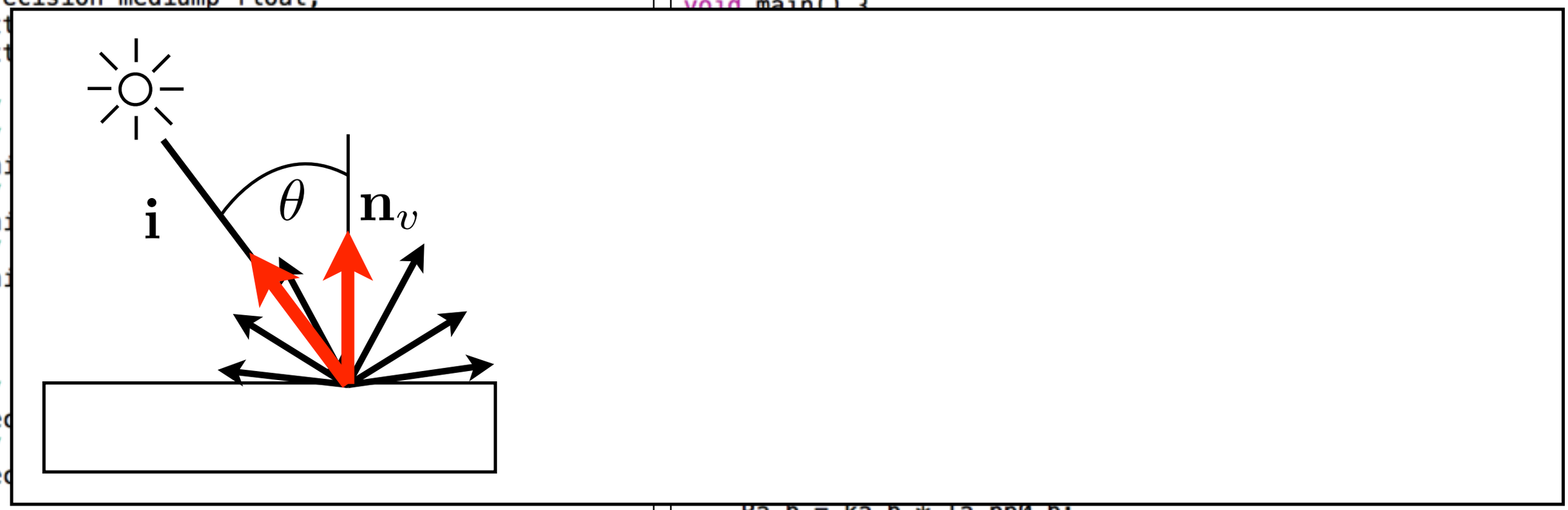
```
mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);

    vec3 Ra, Rd, Rs; // reflected light components

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;



    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;
// coefficients for object
uniform vec3 ka, kd, ks;



// attenuated incident light components
vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction
vec3 i   ;



mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);

    vec3 Ra, Rd, Rs; // reflected light components

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);









    gl_Position=projection * vertexPosition;
}
```

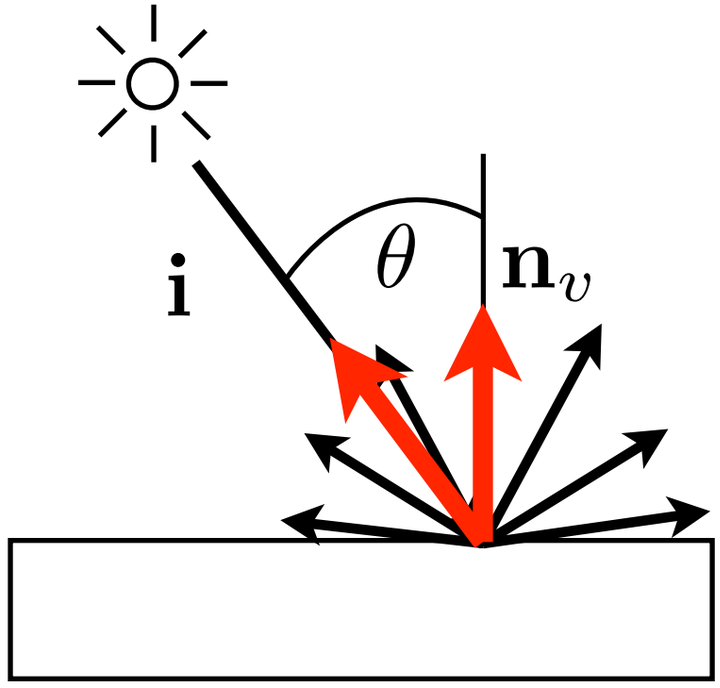
```
precision mediump float;
att
att

//
//
uni
//
uni
//
uni



//
vec
//
vec
```




```
mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

i is the unit vector
from the point
to the light source.

```
void main() {
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);



    gl_Position=projection * vertexPosition;
}
```

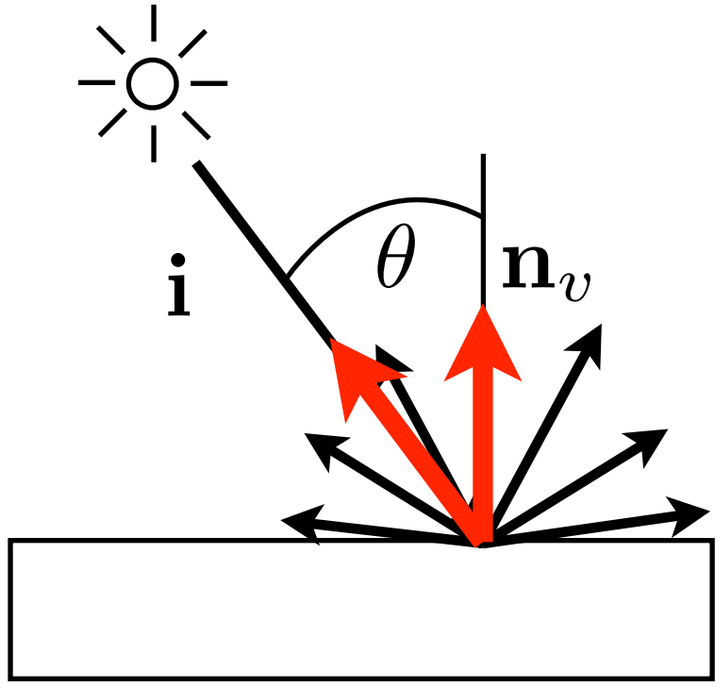


$$\cos\theta = \mathbf{i} \cdot \mathbf{n}_v$$

```
precision mediump float;
att
att

//
//
uni
//
uni
//
uni

//
vec
//
vec

mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

i is the unit vector
from the point
to the light source.

```
void main() {
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    gl_Position=projection * vertexPosition;
}
```



$$R_{dr} = k_{dr} I_{dr} \max(\cos\theta, 0),\ 0 \le k_{dr} \le 1$$

$$R_{dg} = k_{dg} I_{dg} \max(\cos\theta, 0),\ 0 \le k_{dg} \le 1$$

$$R_{db} = k_{db} I_{db} \max(\cos\theta, 0),\ 0 \le k_{db} \le 1$$

$$\cos\theta = \mathbf{i} \cdot \mathbf{n}_v$$

```
precision mediump float;
att
att

//
//
uni
//
uni
//
uni

//
vec
//
vec

mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

i is the unit vector
from the point
to the light source.

```
void main() {
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    gl_Position=projection * vertexPosition;
}
```

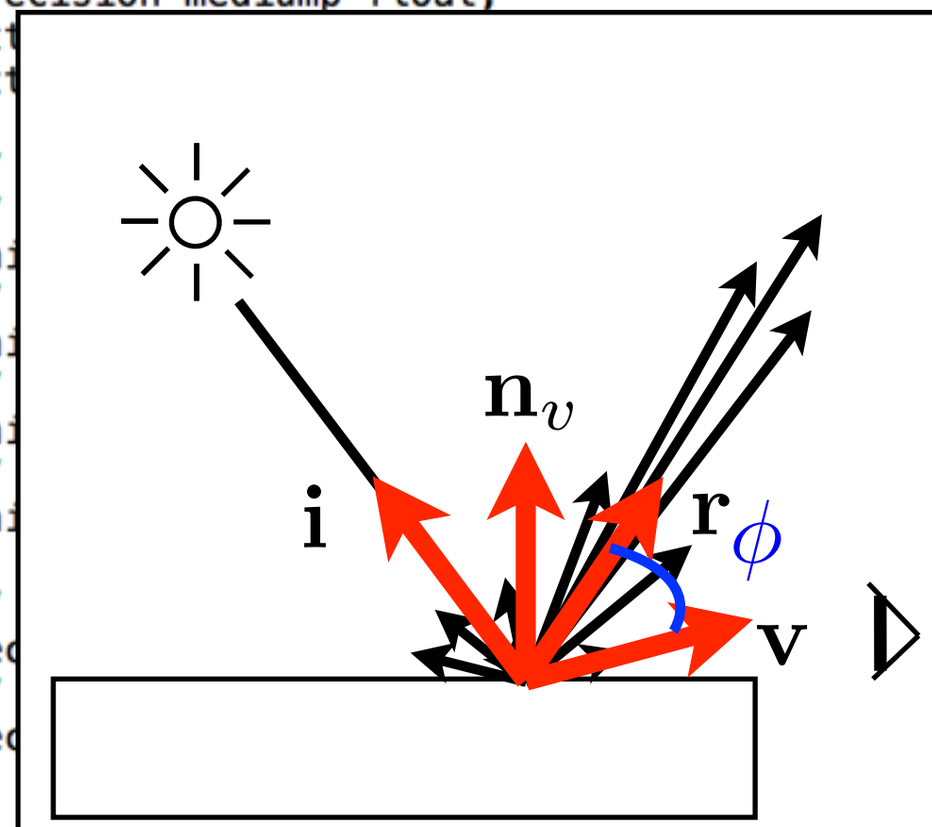
```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;
// coefficients for object
uniform vec3 ka, kd, ks;



// attenuated incident light components
vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction
vec3 i   ;



mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);

    vec3 Ra, Rd, Rs; // reflected light components

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);









    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// attenuated incident light components
vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
vec3 i, v;



mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);

    vec3 Ra, Rd, Rs; // reflected light components

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```

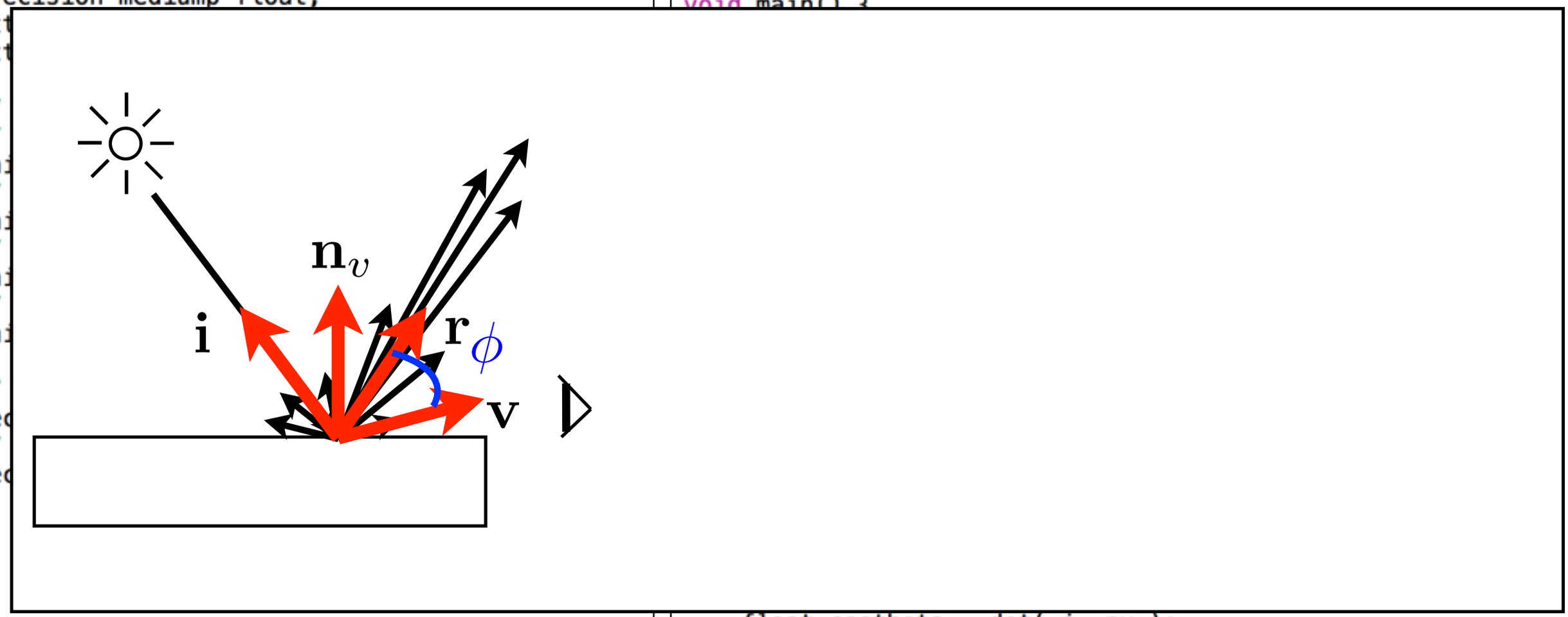
```
precision mediump float;
att
att
att

//
//
uni
//
uni
//
uni
//
uni

//
vec
//
vec
```




```
mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

```
void main() {
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```

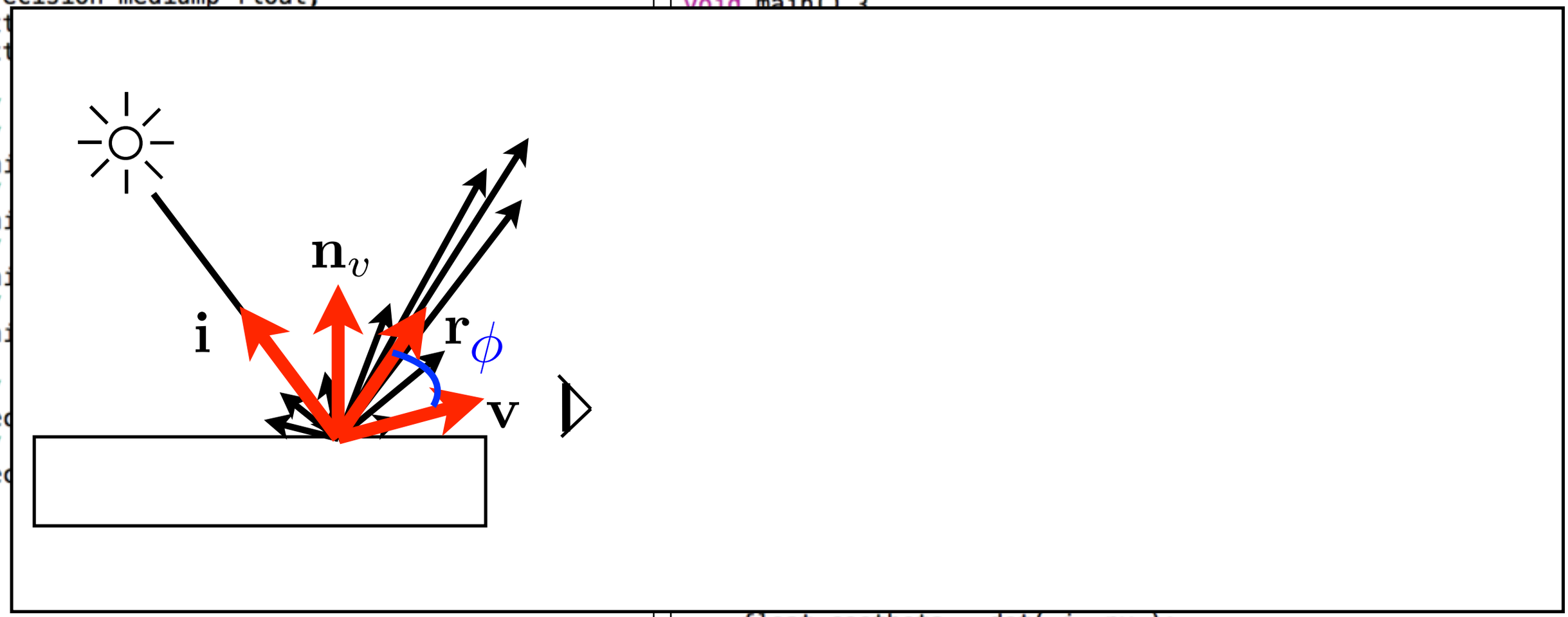
```
precision mediump float;
att
att
att

//
//
uni
//
uni
//
uni
//
uni
//
uni

//
vec
//
vec
```




```
mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

r is reflection vector.

```
void main() {
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```

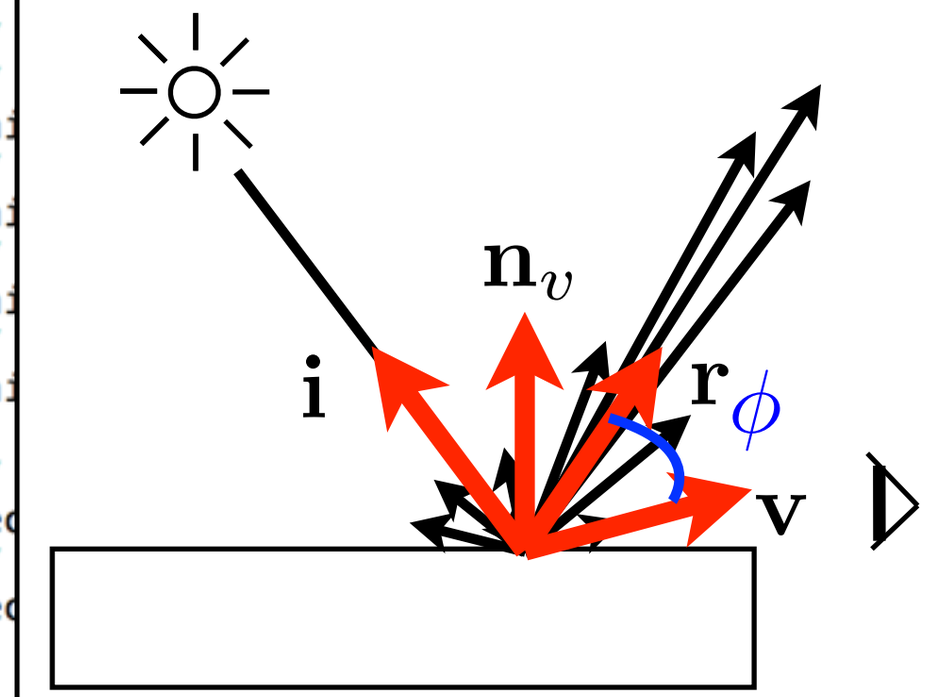



```
precision mediump float;
att
att
att

//
//
uni
//
uni
//
uni
//
uni

//
vec
//
vec

mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

r is reflection vector.

v is view vector. (vector from vertex to camera).

```
void main() {
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```

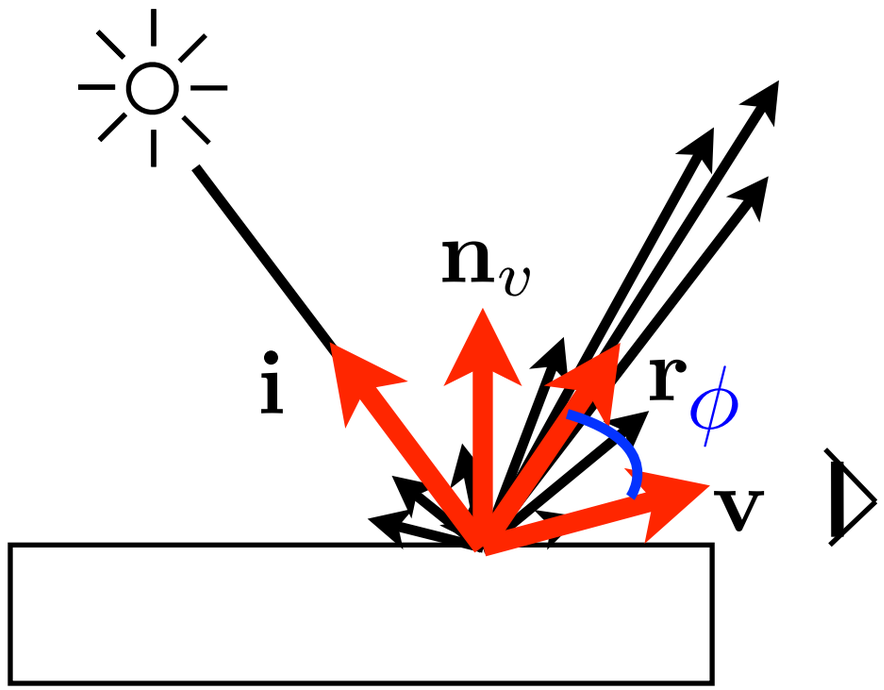


$$\cos\phi = \mathbf{r} \cdot \mathbf{v}$$

```
precision mediump float;
att
att
att

//
//
uni
//
uni
//
uni
//
uni

//
vec
//
vec

mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

cosphi is
dot product of **r** and **v**.

```
void main() {
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```



$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$

$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$

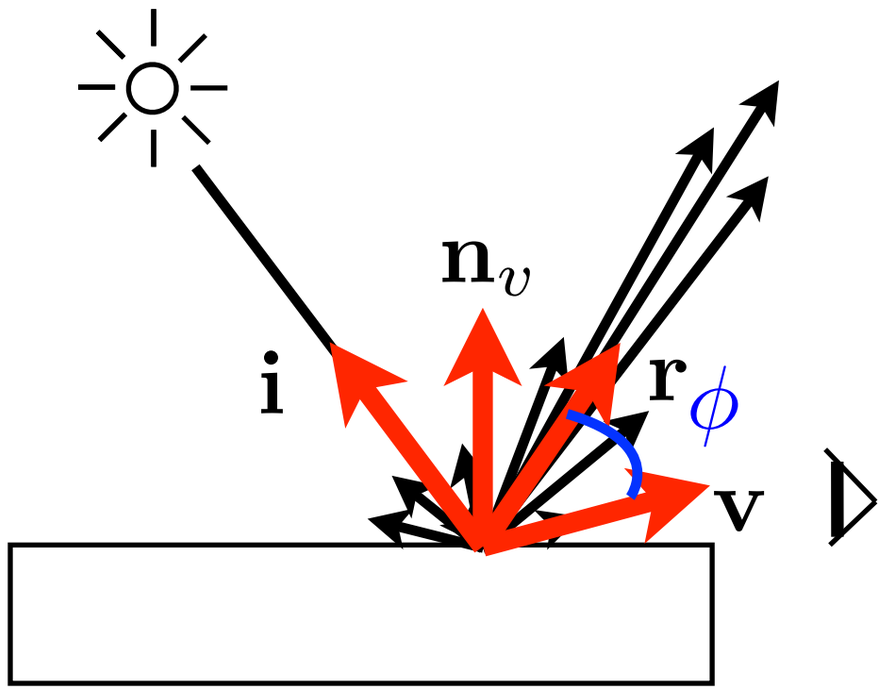
$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$

$$\cos\phi = \mathbf{r} \cdot \mathbf{v}$$

```
precision mediump float;
att
att

//
//
uni
//
uni
//
uni
//
uni

//
vec
//
vec

mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

shine is the clamped value of cosphi raised to the shininess alpha.

```
void main() {
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```



$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$

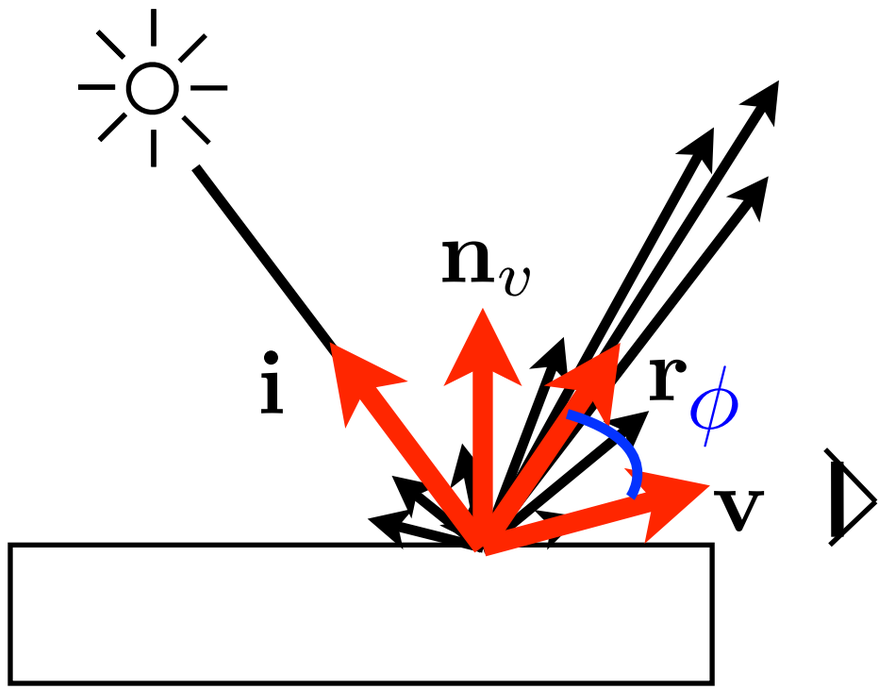
$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$

$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$

$$\cos\phi = \mathbf{r} \cdot \mathbf{v}$$

```
precision mediump float;
att
att

//
//
uni
//
uni
//
uni
//
uni

//
vec
//
vec

mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

```
void main() {
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```



$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$

$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$

$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$

$$\cos\phi = \mathbf{r} \cdot \mathbf{v}$$

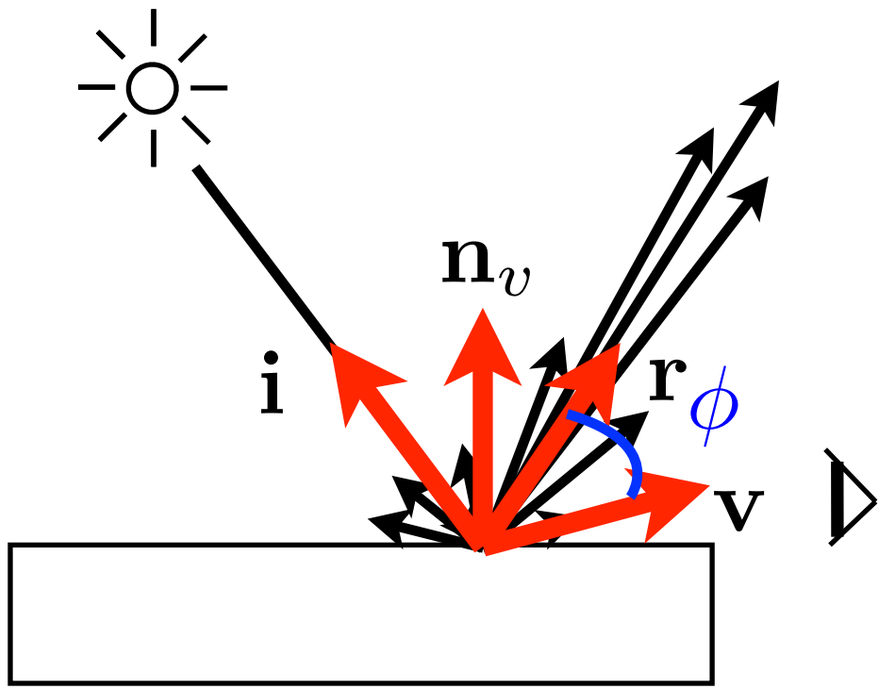
```
mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

costhetag0 is 0 if cos(theta) > 0, else it is 1.

If costhetag0 is 0, there is no specular reflection.

If it costhetag0 is 1, there is specular reflection.

```
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```



$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$

$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$

$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$

$$\cos\phi = \mathbf{r} \cdot \mathbf{v}$$

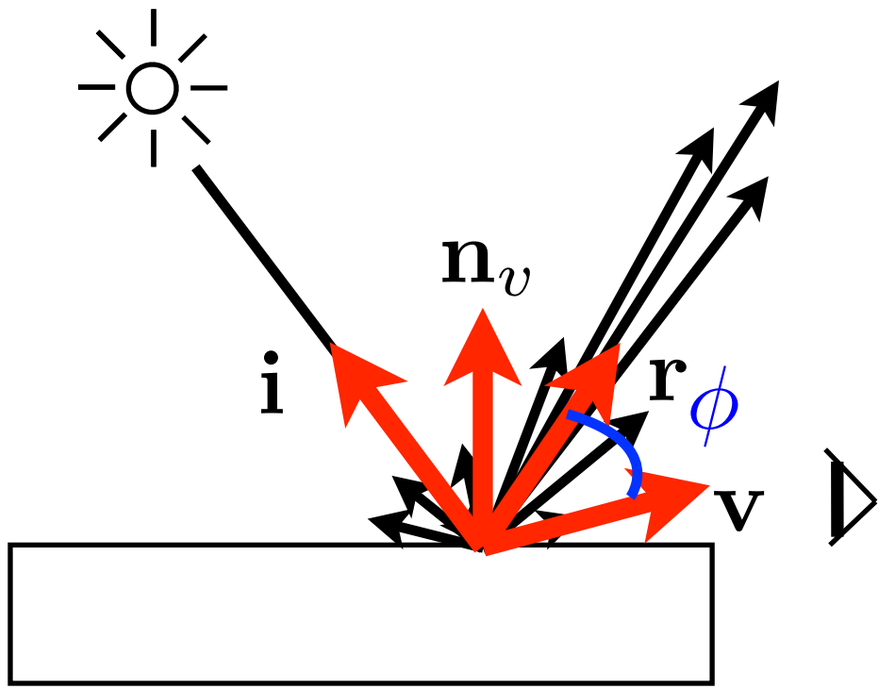
```
mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

costhetag0 is 0 if cos(theta) > 0, else it is 1.

If costhetag0 is 0, there is no specular reflection.

If it costhetag0 is 1, there is specular reflection.

```
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```



$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$

$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$

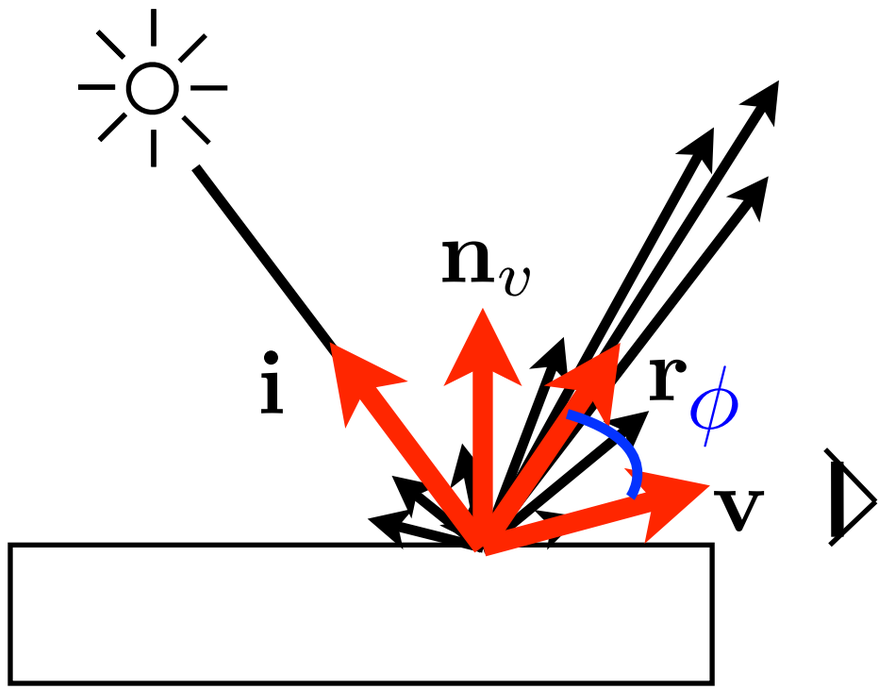
$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$

$$\cos\phi = \mathbf{r} \cdot \mathbf{v}$$

```
mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

sign(costheta) gives:
 1 if costheta > 0
 0 if costheta = 0
-1 if costheta < 0

```
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```



$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$

$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$

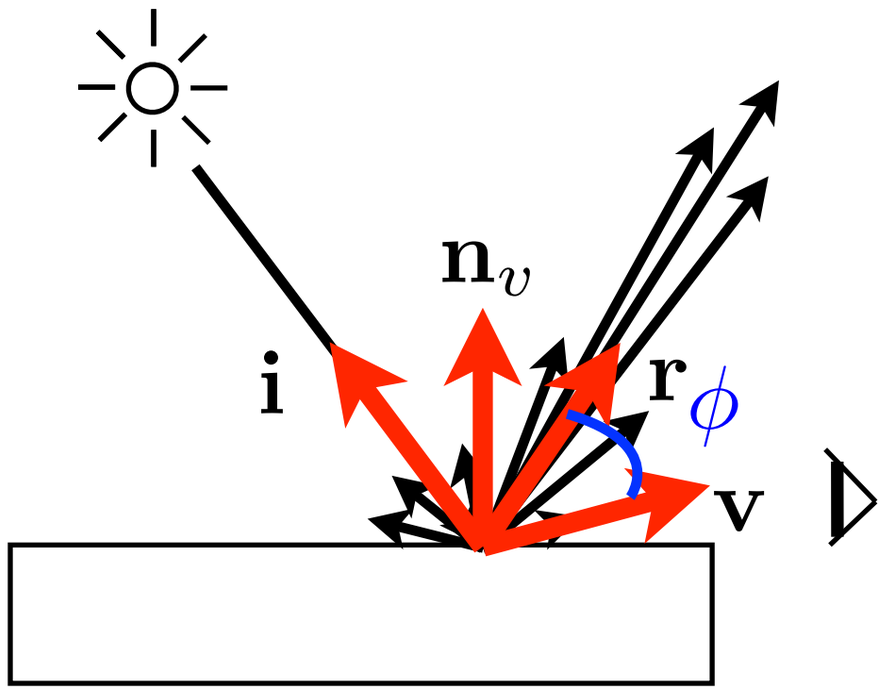
$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$

$$\cos\phi = \mathbf{r} \cdot \mathbf{v}$$

```
mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

sign(costheta)+1 gives:
 2 if costheta > 0
 1 if costheta = 0
 0 if costheta < 0

```
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```



$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$

$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$

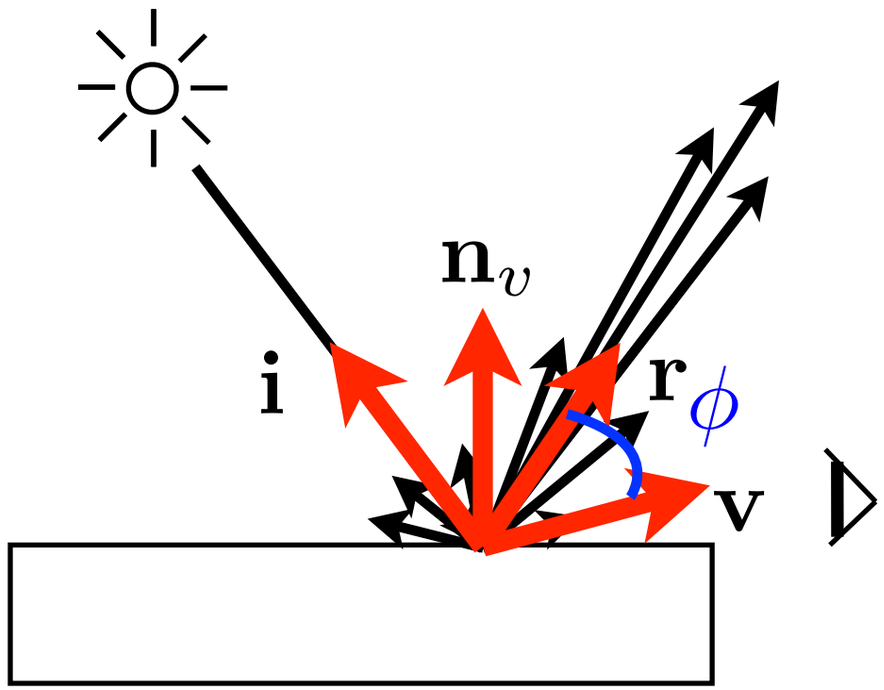
$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$

$$\cos\phi = \mathbf{r} \cdot \mathbf{v}$$

```
mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

0.5 * (sign(costheta)+1) gives:
1 if costheta > 0
0.5 if costheta = 0
0 if costheta < 0

```
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```



$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$

$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$

$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$

$$\cos\phi = \mathbf{r} \cdot \mathbf{v}$$

```
mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

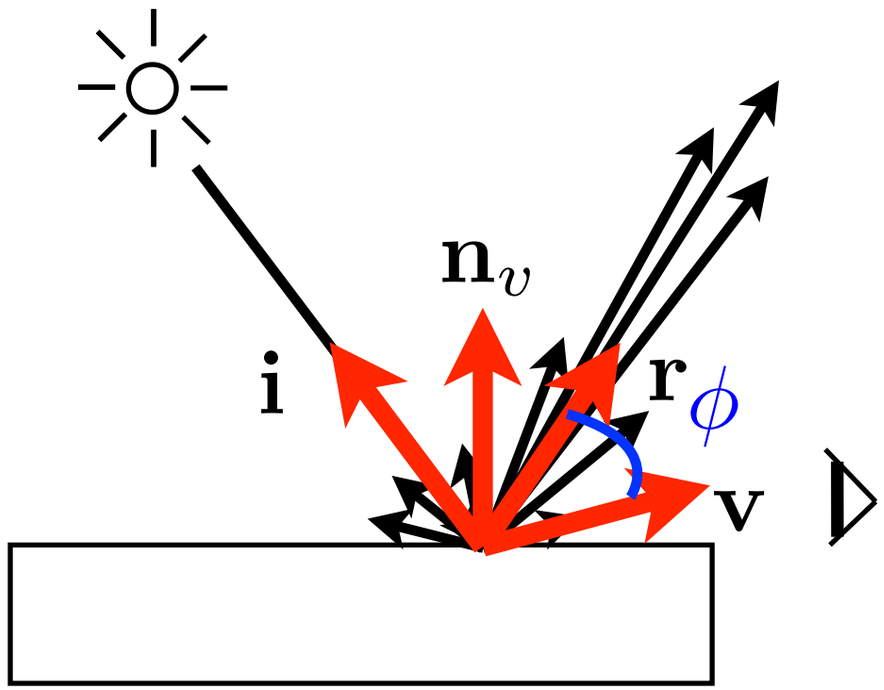
floor(0.5 * (sign(costheta)+1)) gives:

1 if costheta > 0

0 if costheta = 0

0 if costheta < 0

```
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```



$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$

$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$

$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$

$$\cos\phi = \mathbf{r} \cdot \mathbf{v}$$

```
mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

floor(0.5 * (sign(costheta)+1)) gives:
 1 if costheta > 0
 0 if costheta = 0
 0 if costheta < 0

costhetag0 ends up being:
 1 if costheta > 0
 0 if costheta <= 0

```
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;



    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;
// coefficients for object
uniform vec3 ka, kd, ks;



// attenuated incident light components
vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
vec3 i, v;

// final reflected light
// (interpolated to fragment shader)
varying vec3 R;

mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

Reflected light R is interpolated to fragment shader.

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);

    vec3 Ra, Rd, Rs; // reflected light components

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
varying vec3 R; // final reflected light (interpolated from vertex shader)

void main() {
    gl_FragColor=vec4( R.r, R.g, R.b, 1.0 );
}
```

Fragment shader is fairly simple.

Reflected light R is interpolated from vertex shader.

Gouraud Shading Result

p0 = (.0, .0, .6)

ka = (.5, .5, .5)
kd = (.5, .5, .5)
ks = (1.0, 1.0, 1.0)

la = (.1, .1, .1)
ld = (.8, .8, .8)
ls = (.8, .8, .8)

alpha = 10.0

Now let us look at Phong Shading

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// attenuated incident light components
vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
vec3 i, v;

// final reflected light
// (interpolated to fragment shader)
varying vec3 R;

mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

In Gouraud Shading, all light computations occur in vertex shader.

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);

    vec3 Ra, Rd, Rs; // reflected light components

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;



// attenuated incident light components
vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
vec3 i, v;



mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

In Phong Shading, interpolated unit vectors and light components are computed in vertex shader.

```
void main() {
    gl_PointSize = 1.0;









    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;




// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;



mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

In Phong Shading, interpolated unit vectors and light components are computed in vertex shader.

```
void main() {
    gl_PointSize = 1.0;









    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;




// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;



mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

In Phong Shading, interpolated unit vectors and light components are computed in vertex shader.

```
void main() {
    gl_PointSize = 1.0;









    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;




// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;



mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

In Phong Shading, interpolated unit vectors and light components are computed in vertex shader.

```
void main() {
    gl_PointSize = 1.0;









    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;




// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;



mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

In Phong Shading, interpolated unit vectors and light components are computed in vertex shader.

```
void main() {
    gl_PointSize = 1.0;
    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;



// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;



mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

In Phong Shading, the main() function of the shader does not calculate reflected colors.

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);

    vec3 Ra, Rd, Rs; // reflected light components

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;



// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;


mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);

    vec3 Ra, Rd, Rs; // reflected light components

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_Position=projection * vertexPosition;
}
```

In Phong Shading, the main() function of the shader does not calculate reflected colors.

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;




// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;



mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

You only calculate the light components, source vector and view vector for interpolation...

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);









    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );




    // Specular Reflection

    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);









    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;




// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;


mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

...and ensure
the fragment normal
gets interpolated.

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);

    // Fragment normal at vertex is just vertex normal
    n = nv;




    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );




    // Specular Reflection

    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);




    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;




// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;


mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

Vertex shader
is much smaller.

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);

    // Fragment normal at vertex is just vertex normal
    n = nv;




    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );




    // Specular Reflection

    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);




    gl_Position=projection * vertexPosition;
}
```

```
precision mediump float;

// uniforms:
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// Following variables are interpolated
// from vertex shader:
// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;

// Final reflected light calculated here
vec3 R;
```

```
void main() {

    // reflected light components
    vec3 Ra, Rd, Rs;

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    float costheta = dot( i, n );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0 );
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0 );
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0 );

    // Specular Reflection
    // unit vector for reflection:
    vec3 r = normalize(2.0 * costheta * n - i);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_FragColor=vec4( R.r, R.g, R.b, 1.0 );
}
```

Fragment shader is larger than for Gouraud shading.

```
precision mediump float;

// uniforms:
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// Following variables are interpolated
// from vertex shader:
// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;

// Final reflected light calculated here
vec3 R;
```

```
void main() {

    // reflected light components
    vec3 Ra, Rd, Rs;

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    float costheta = dot( i, n );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0 );
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0 );
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0 );

    // Specular Reflection
    // unit vector for reflection:
    vec3 r = normalize(2.0 * costheta * n - i);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_FragColor=vec4( R.r, R.g, R.b, 1.0 );
}
```

Fragment shader is larger than for Gouraud shading.

```
precision mediump float;

// uniforms:
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// Following variables are interpolated
// from vertex shader:
// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;

// Final reflected light calculated here
vec3 R;
```

```
void main() {

    // reflected light components
    vec3 Ra, Rd, Rs;

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    float costheta = dot( i, n );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0 );
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0 );
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0 );

    // Specular Reflection
    // unit vector for reflection:
    vec3 r = normalize(2.0 * costheta * n - i);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_FragColor=vec4( R.r, R.g, R.b, 1.0 );
}
```

Fragment shader is larger than for Gouraud shading.

```
precision mediump float;

// uniforms:
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// Following variables are interpolated
// from vertex shader:
// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;

// Final reflected light calculated here
vec3 R;
```

One issue with this implementation is that varying vectors **i**, **v**, and **n** are not necessarily unit vectors after interpolation.

```
void main() {

    // reflected light components
    vec3 Ra, Rd, Rs;

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    float costheta = dot( i, n );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0 );
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0 );
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0 );

    // Specular Reflection
    // unit vector for reflection:
    vec3 r = normalize(2.0 * costheta * n - i);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_FragColor=vec4( R.r, R.g, R.b, 1.0 );
}
```

```
precision mediump float;

// uniforms:
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// Following variables are interpolated
// from vertex shader:
// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;

// Final reflected light calculated here
vec3 R;
```

We need to renormalize **i**, **v**, and **n**.

```
void main() {

    // reflected light components
    vec3 Ra, Rd, Rs;

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    float costheta = dot( i, n );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0 );
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0 );
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0 );

    // Specular Reflection
    // unit vector for reflection:
    vec3 r = normalize(2.0 * costheta * n - i);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_FragColor=vec4( R.r, R.g, R.b, 1.0 );
}
```

```
precision mediump float;

// uniforms:
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// Following variables are interpolated
// from vertex shader:
// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;

// Final reflected light calculated here
vec3 R;
```

We need to renormalize **i**, **v**, and **n**.

```
void main() {
    // You need to renormalize i, v, and n
    // because interpolation
    // does not normalize them.
    vec3 i_renorm, r_renorm,
        v_renorm, n_renorm;
    i_renorm = normalize( i );
    v_renorm = normalize( v );
    n_renorm = normalize( n );

    // reflected light components
    vec3 Ra, Rd, Rs;

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    float costheta = dot( i, n );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0 );
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0 );
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0 );

    // Specular Reflection
    // unit vector for reflection:
    vec3 r = normalize(2.0 * costheta * n - i);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_FragColor=vec4( R.r, R.g, R.b, 1.0 );
}
```

```
precision mediump float;

// uniforms:
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// Following variables are interpolated
// from vertex shader:
// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;

// Final reflected light calculated here
vec3 R;
```

We need to renormalize **i**, **v**, and **n**.

We use re-normalized versions of the vectors in the fragment shader.

```
void main() {
    // You need to renormalize i, v, and n
    // because interpolation
    // does not normalize them.
    vec3 i_renorm, r_renorm,
        v_renorm, n_renorm;
    i_renorm = normalize( i );
    v_renorm = normalize( v );
    n_renorm = normalize( n );

    // reflected light components
    vec3 Ra, Rd, Rs;

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    float costheta = dot( i, n );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0 );
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0 );
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0 );

    // Specular Reflection
    // unit vector for reflection:
    vec3 r = normalize(2.0 * costheta * n - i);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_FragColor=vec4( R.r, R.g, R.b, 1.0 );
}
```

```
precision mediump float;

// uniforms:
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// Following variables are interpolated
// from vertex shader:
// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;

// Final reflected light calculated here
vec3 R;
```

We need to renormalize **i**, **v**, and **n**.

We use re-normalized versions of the vectors in the fragment shader.

```
void main() {
    // You need to renormalize i, v, and n
    // because interpolation
    // does not normalize them.
    vec3 i_renorm, r_renorm,
        v_renorm, n_renorm;
    i_renorm = normalize( i );
    v_renorm = normalize( v );
    n_renorm = normalize( n );

    // reflected light components
    vec3 Ra, Rd, Rs;

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    float costheta = dot( i, n );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0 );
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0 );
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0 );

    // Specular Reflection
    // unit vector for reflection:
    vec3 r = normalize(2.0 * costheta * n - i);
    float cosphi = dot( r, v );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_FragColor=vec4( R.r, R.g, R.b, 1.0 );
}
```

```
precision mediump float;

// uniforms:
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// Following variables are interpolated
// from vertex shader:
// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;

// Final reflected light calculated here
vec3 R;
```

We need to renormalize **i**, **v**, and **n**.

We use re-normalized versions of the vectors in the fragment shader.

```
void main() {
    // You need to renormalize i, v, and n
    // because interpolation
    // does not normalize them.
    vec3 i_renorm, r_renorm,
        v_renorm, n_renorm;
    i_renorm = normalize( i );
    v_renorm = normalize( v );
    n_renorm = normalize( n );

    // reflected light components
    vec3 Ra, Rd, Rs;

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection, use renormalized i
    float costheta = dot( i_renorm, n_renorm );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0 );
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0 );
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0 );

    // Specular Reflection, use renormalized v
    // unit vector for reflection:
    vec3 r = normalize(2.0 * costheta * n_renorm - i_renorm);
    float cosphi = dot( r, v_renorm );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_FragColor=vec4( R.r, R.g, R.b, 1.0 );
}
```

```
precision mediump float;

// uniforms:
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// Following variables are interpolated
// from vertex shader:
// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;

// Final reflected light calculated here
vec3 R;
```

We need to renormalize **i**, **v**, and **n**.

We use re-normalized versions of the vectors in the fragment shader.

```
void main() {
    // You need to renormalize i, v, and n
    // because interpolation
    // does not normalize them.
    vec3 i_renorm, r_renorm,
        v_renorm, n_renorm;
    i_renorm = normalize( i );
    v_renorm = normalize( v );
    n_renorm = normalize( n );

    // reflected light components
    vec3 Ra, Rd, Rs;

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection, use renormalized i
    float costheta = dot( i_renorm, n_renorm );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0 );
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0 );
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0 );

    // Specular Reflection, use renormalized v
    // unit vector for reflection:
    vec3 r = normalize(2.0 * costheta * n_renorm - i_renorm);
    float cosphi = dot( r, v_renorm );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_FragColor=vec4( R.r, R.g, R.b, 1.0 );
}
```

```
precision mediump float;

// uniforms:
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// Following variables are interpolated
// from vertex shader:
// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;

// Final reflected light calculated here
vec3 R;
```

```
void main() {
    // You need to renormalize i, v, and n
    // because interpolation
    // does not normalize them.
    vec3 i_renorm, r_renorm,
        v_renorm, n_renorm;
    i_renorm = normalize( i );
    v_renorm = normalize( v );
    n_renorm = normalize( n );

    // reflected light components
    vec3 Ra, Rd, Rs;

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection, use renormalized i
    float costheta = dot( i_renorm, n_renorm );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0 );
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0 );
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0 );

    // Specular Reflection, use renormalized v
    // unit vector for reflection:
    vec3 r = normalize(2.0 * costheta * n_renorm - i_renorm);
    float cosphi = dot( r, v_renorm );
    float shine = pow( max( cosphi, 0.0 ), alpha );
    float costhetag0 = floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_FragColor=vec4( R.r, R.g, R.b, 1.0 );
}
```

Phong Shading Result

p0 = (.0, .0, .6)

ka = (.5, .5, .5)
kd = (.5, .5, .5)
ks = (1.0, 1.0, 1.0)

la = (.1, .1, .1)
ld = (.8, .8, .8)
ls = (.8, .8, .8)

alpha = 10.0

Gouraud Shading Result

p0 = (.0, .0, .6)

ka = (.5, .5, .5)
kd = (.5, .5, .5)
ks = (1.0, 1.0, 1.0)

la = (.1, .1, .1)
ld = (.8, .8, .8)
ls = (.8, .8, .8)

alpha = 10.0

Transforming normals under the modelview matrix

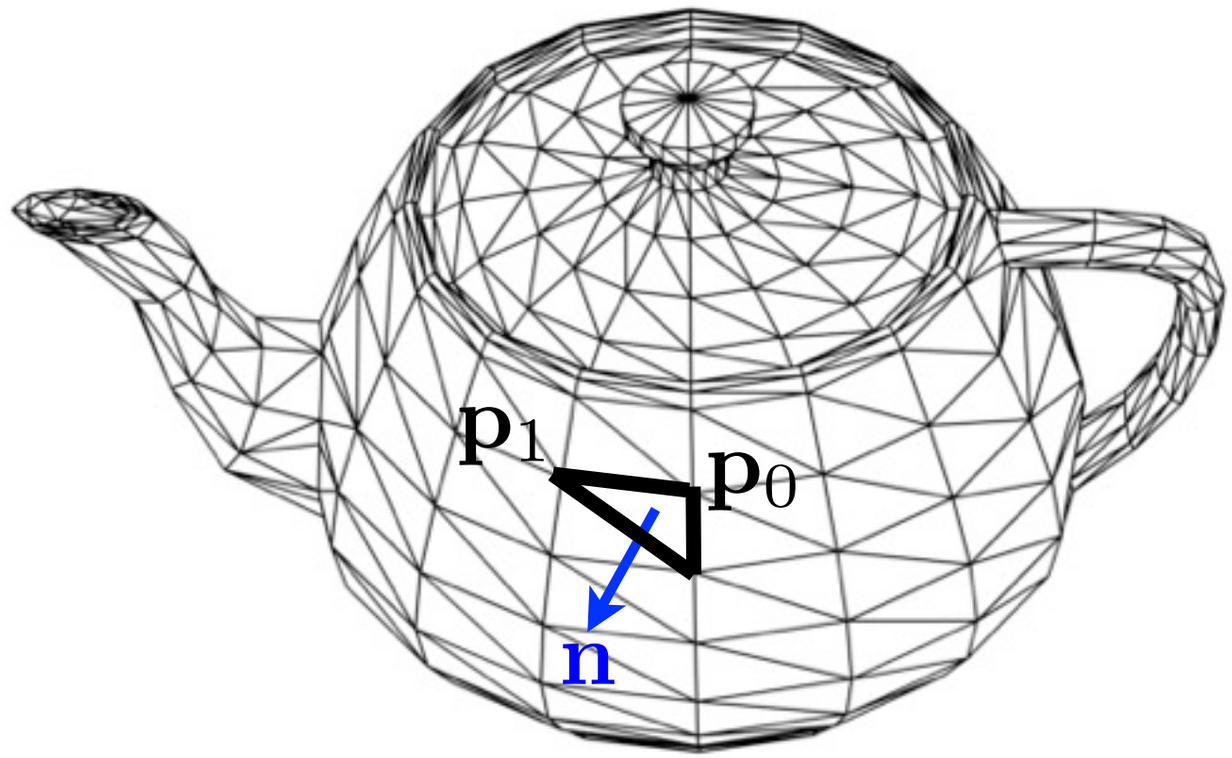
Phong Shading on Transformed Teapot

We need to transform the normals as well!

Phong Shading on Transformed Teapot

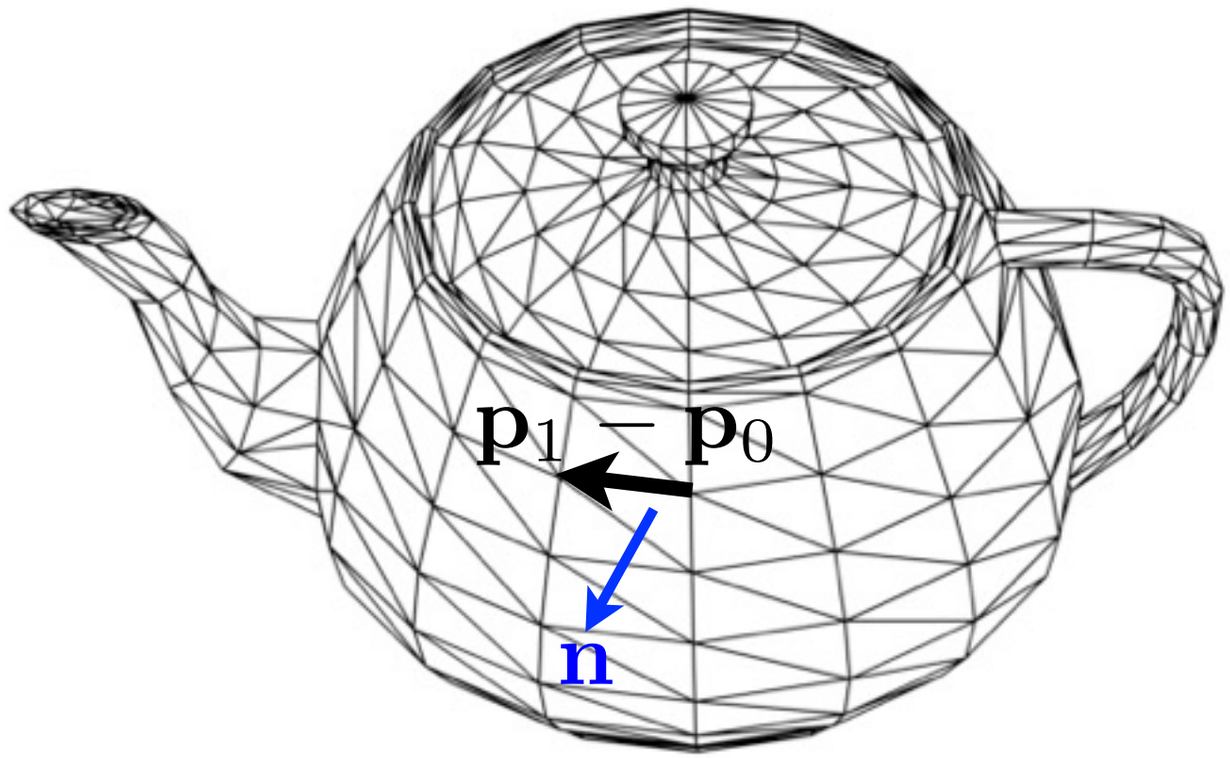
We need to transform the normals as well!

n is perpendicular to plane containing $\mathbf{p}_0$ and $\mathbf{p}_1$



n is perpendicular to plane containing $\mathbf{p}_0$ and $\mathbf{p}_1$

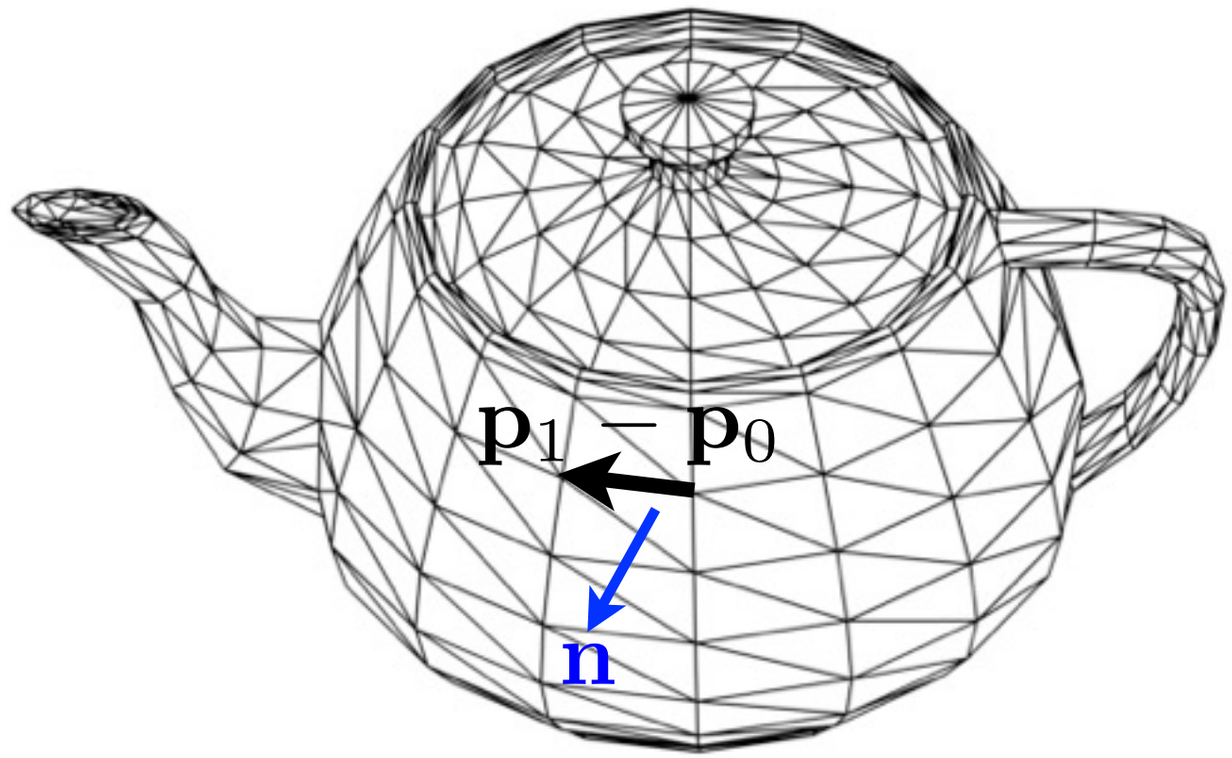
$$\mathbf{n} \cdot (\mathbf{p}_1 - \mathbf{p}_0) = 0$$



n is perpendicular to plane containing $\mathbf{p}_0$ and $\mathbf{p}_1$

$$\mathbf{n} \cdot (\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$\mathbf{n}^T(\mathbf{p}_1 - \mathbf{p}_0) = 0$$



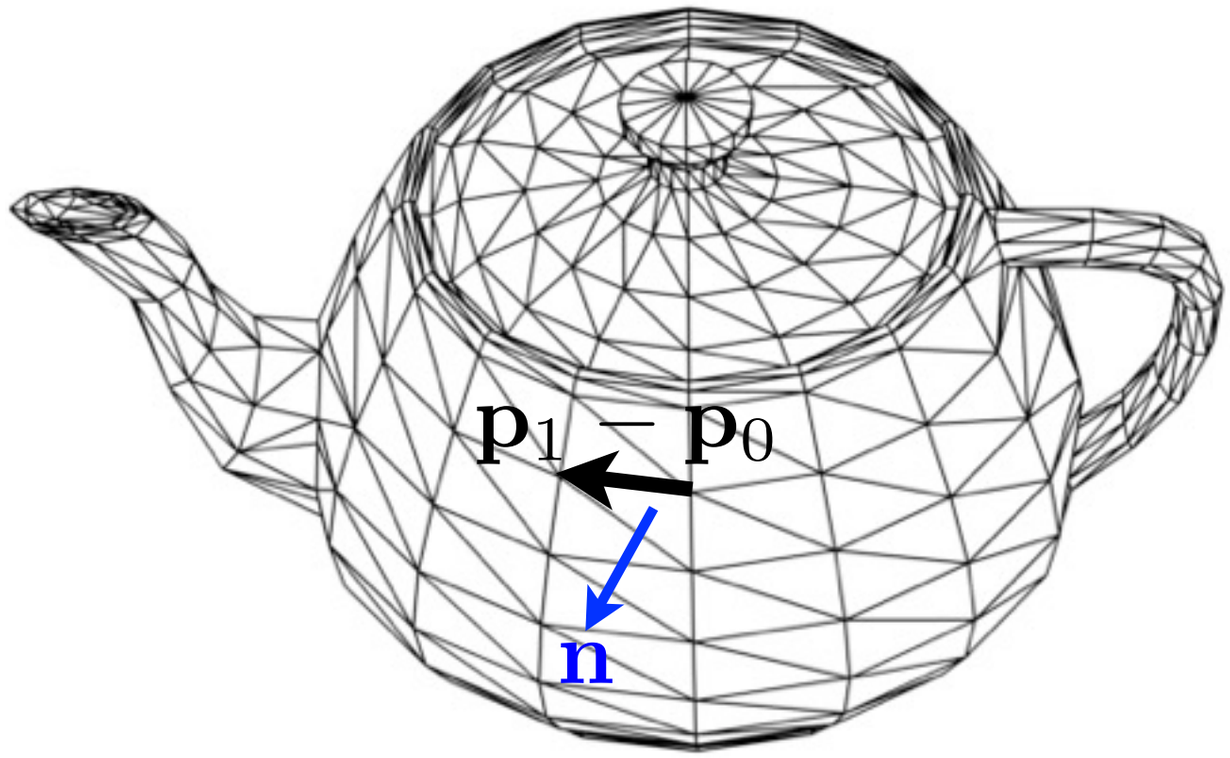
n is perpendicular to plane containing $\mathbf{p}_0$ and $\mathbf{p}_1$

M is a modelview matrix. $\mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$ (identity matrix).

$$\mathbf{n} \cdot (\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$\mathbf{n}^T(\mathbf{p}_1 - \mathbf{p}_0) = 0$$

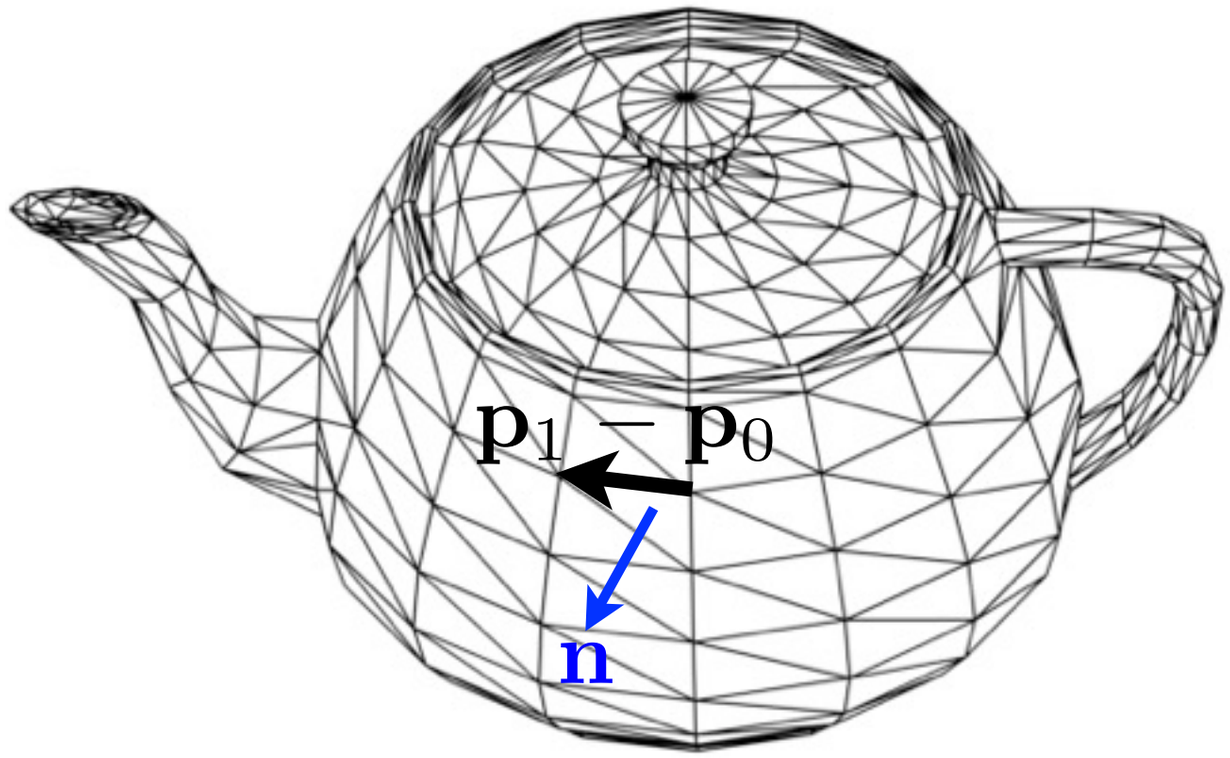
$$\mathbf{n}^T(\mathbf{M}^{-1}\mathbf{M})(\mathbf{p}_1 - \mathbf{p}_0) = 0$$



n is perpendicular to plane containing $\mathbf{p}_0$ and $\mathbf{p}_1$

Re-arranging parentheses.

$$\mathbf{n} \cdot (\mathbf{p}_1 - \mathbf{p}_0) = 0$$
$$\mathbf{n}^T(\mathbf{p}_1 - \mathbf{p}_0) = 0$$
$$\mathbf{n}^T(\mathbf{M}^{-1}\mathbf{M})(\mathbf{p}_1 - \mathbf{p}_0) = 0$$
$$(\mathbf{n}^T\mathbf{M}^{-1})(\mathbf{M}(\mathbf{p}_1 - \mathbf{p}_0)) = 0$$



n is perpendicular to plane containing $\mathbf{p}_0$ and $\mathbf{p}_1$

Distribute **M** to $\mathbf{p}_1$ and $\mathbf{p}_2$.

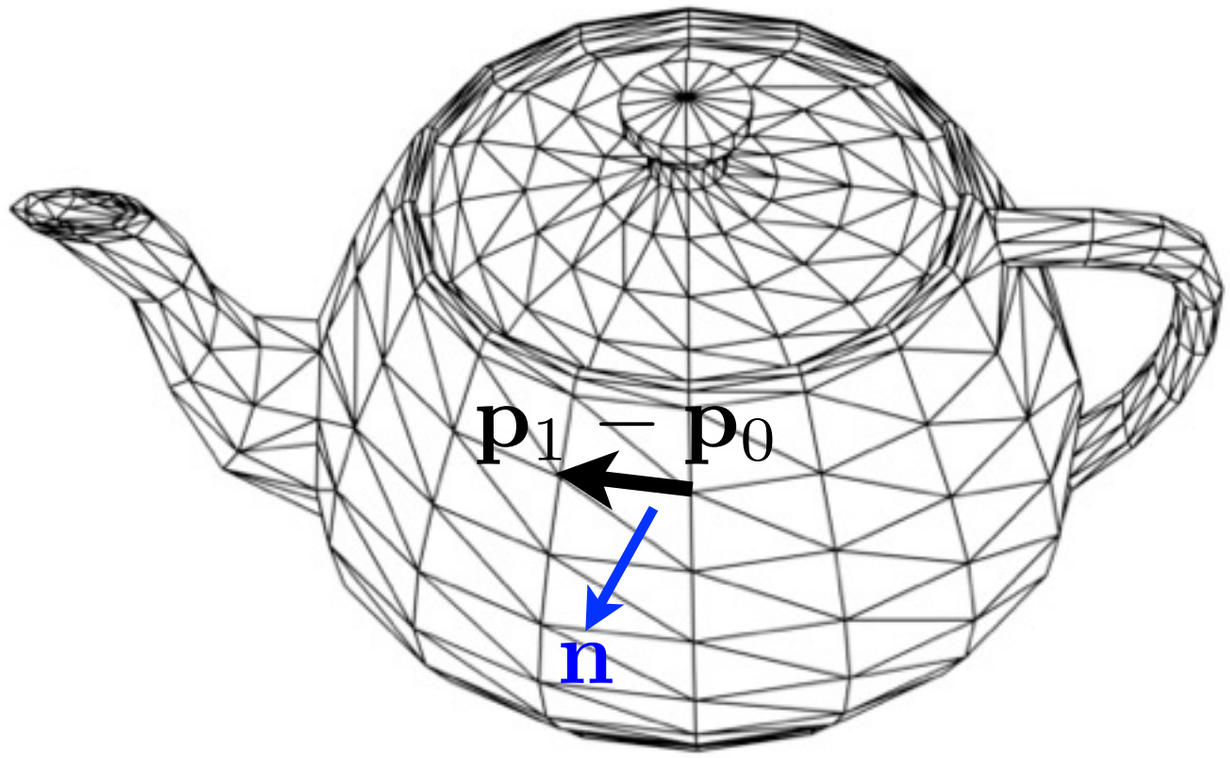
$$\mathbf{n} \cdot (\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$\mathbf{n}^T(\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$\mathbf{n}^T(\mathbf{M}^{-1}\mathbf{M})(\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$(\mathbf{n}^T\mathbf{M}^{-1})(\mathbf{M}(\mathbf{p}_1 - \mathbf{p}_0)) = 0$$

$$(\mathbf{n}^T\mathbf{M}^{-1})(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$



n is perpendicular to plane containing $\mathbf{p}_0$ and $\mathbf{p}_1$

Bear with me, while we do this double transpose.

$((\mathbf{M}^{-1})^T)^T = \mathbf{M}^{-1}$ $((\mathbf{A})^T)^T = \mathbf{A}$

$$\mathbf{n} \cdot (\mathbf{p}_1 - \mathbf{p}_0) = 0$$

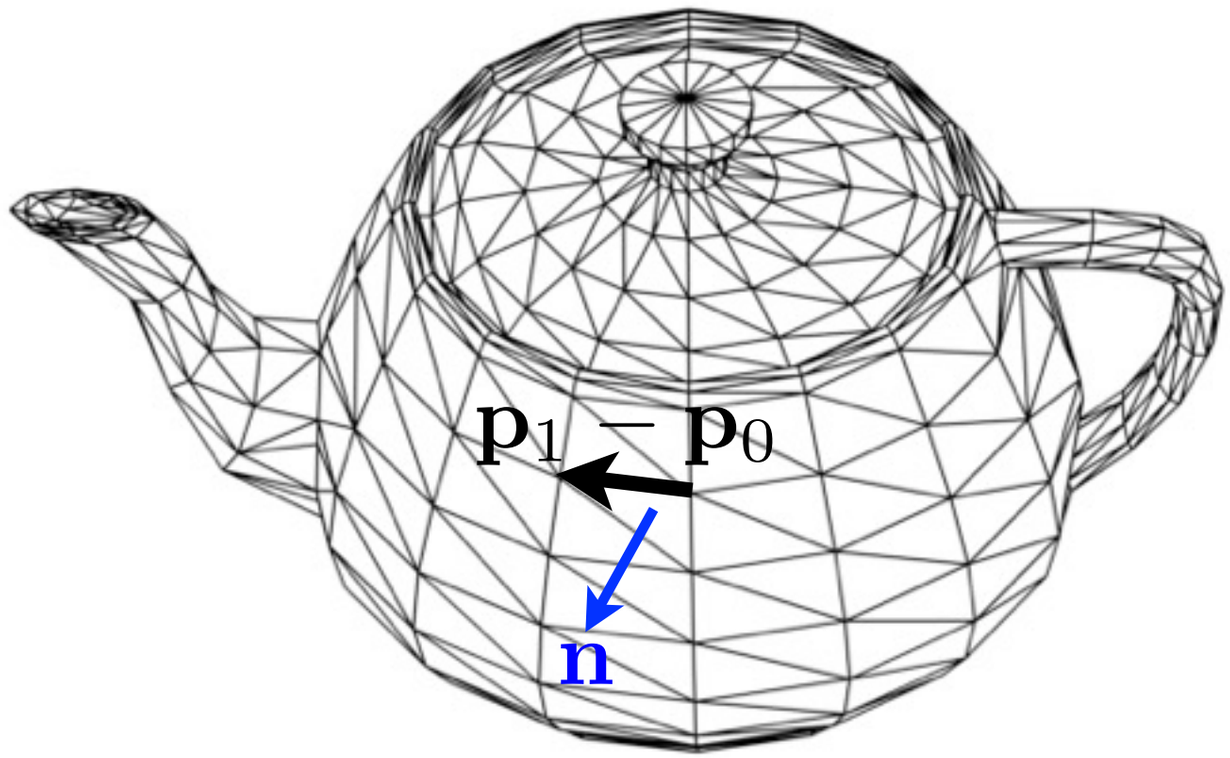
$$\mathbf{n}^T (\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$\mathbf{n}^T (\mathbf{M}^{-1}\mathbf{M})(\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$(\mathbf{n}^T \mathbf{M}^{-1})(\mathbf{M}(\mathbf{p}_1 - \mathbf{p}_0)) = 0$$

$$(\mathbf{n}^T \mathbf{M}^{-1})(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$

$$(\mathbf{n}^T ((\mathbf{M}^{-1})^T)^T)(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$



n is perpendicular to plane containing $\mathbf{p}_0$ and $\mathbf{p}_1$

Bring outer transpose around **n** and **M**.

$\mathbf{n}^T ((\mathbf{M}^{-1})^T)^T = ((\mathbf{M}^{-1})^T \mathbf{n})^T$ $\qquad$ $\mathbf{A}^T \mathbf{B}^T = (\mathbf{B}\,\mathbf{A})^T$

$$\mathbf{n} \cdot (\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$\mathbf{n}^T (\mathbf{p}_1 - \mathbf{p}_0) = 0$$

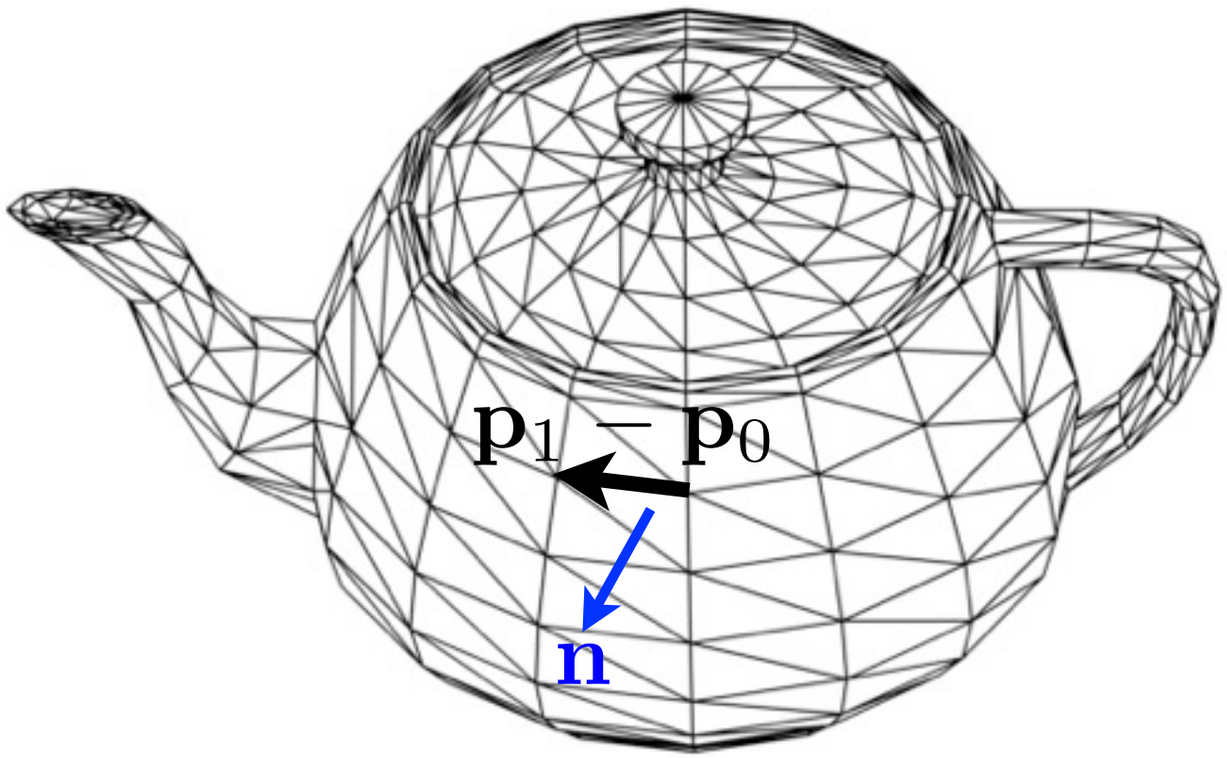
$$\mathbf{n}^T (\mathbf{M}^{-1}\mathbf{M})(\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$(\mathbf{n}^T \mathbf{M}^{-1})(\mathbf{M}(\mathbf{p}_1 - \mathbf{p}_0)) = 0$$

$$(\mathbf{n}^T \mathbf{M}^{-1})(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$

$$(\mathbf{n}^T ((\mathbf{M}^{-1})^T)^T)(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$

$$((\mathbf{M}^{-1})^T \mathbf{n})^T (\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$



n is perpendicular to plane containing $\mathbf{p}_0$ and $\mathbf{p}_1$

$$\mathbf{n} \cdot (\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$\mathbf{n}^T(\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$\mathbf{n}^T(\mathbf{M}^{-1}\mathbf{M})(\mathbf{p}_1 - \mathbf{p}_0) = 0$$

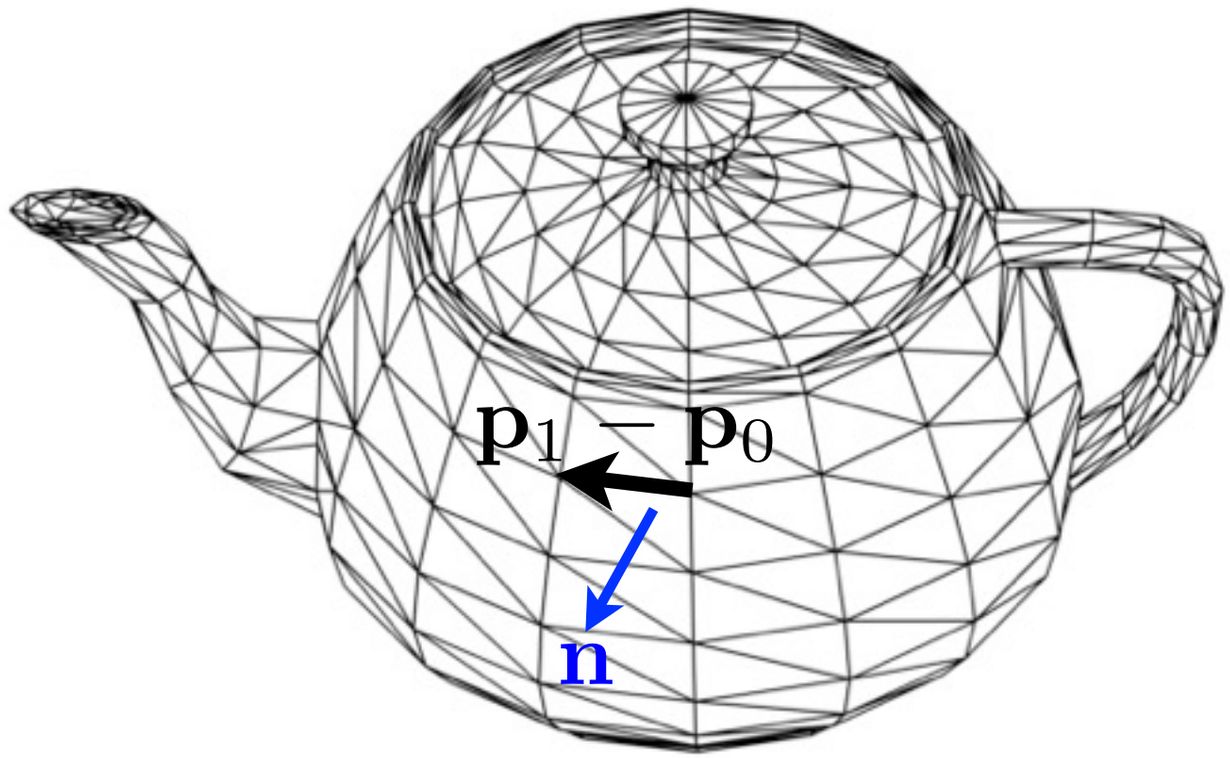
$$(\mathbf{n}^T\mathbf{M}^{-1})(\mathbf{M}(\mathbf{p}_1 - \mathbf{p}_0)) = 0$$

$$(\mathbf{n}^T\mathbf{M}^{-1})(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$

$$(\mathbf{n}^T((\mathbf{M}^{-1})^T)^T)(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$

$$((\mathbf{M}^{-1})^T\mathbf{n})^T(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$

$$(\mathbf{M}^{-T}\mathbf{n})^T(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$



n is perpendicular to plane containing $\mathbf{p}_0$ and $\mathbf{p}_1$

$$\mathbf{n} \cdot (\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$\mathbf{n}^T(\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$\mathbf{n}^T(\mathbf{M}^{-1}\mathbf{M})(\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$(\mathbf{n}^T\mathbf{M}^{-1})(\mathbf{M}(\mathbf{p}_1 - \mathbf{p}_0)) = 0$$

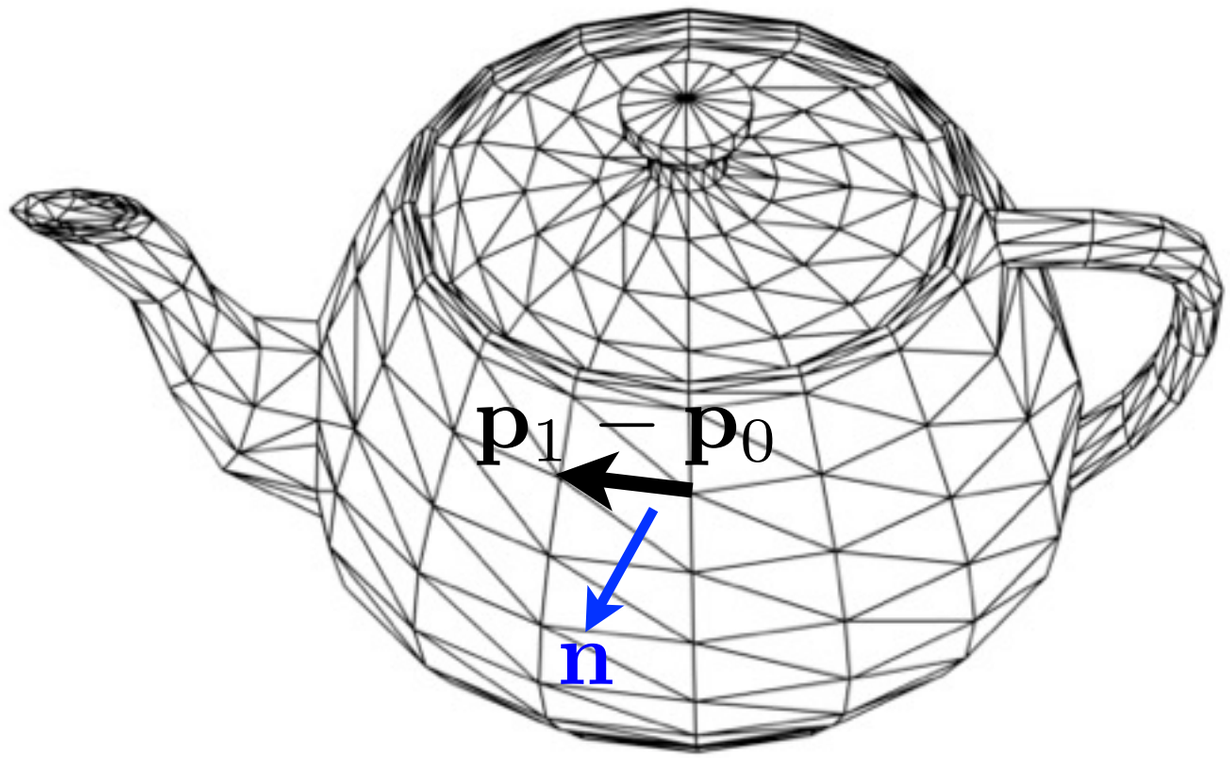
$$(\mathbf{n}^T\mathbf{M}^{-1})(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$

$$(\mathbf{n}^T((\mathbf{M}^{-1})^T)^T)(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$

$$((\mathbf{M}^{-1})^T\mathbf{n})^T(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$

$$(\underline{\mathbf{M}^{-T}\mathbf{n}})^T(\underline{\mathbf{M}\mathbf{p}_1} - \underline{\mathbf{M}\mathbf{p}_0}) = 0$$

$$\mathbf{n}_t \cdot (\mathbf{p}_{t1} - \mathbf{p}_{t0}) = 0$$



n is transformed by $\mathbf{M}^{-T}$

$$\mathbf{n} \cdot (\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$\mathbf{n}^T(\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$\mathbf{n}^T(\mathbf{M}^{-1}\mathbf{M})(\mathbf{p}_1 - \mathbf{p}_0) = 0$$

$$(\mathbf{n}^T\mathbf{M}^{-1})(\mathbf{M}(\mathbf{p}_1 - \mathbf{p}_0)) = 0$$

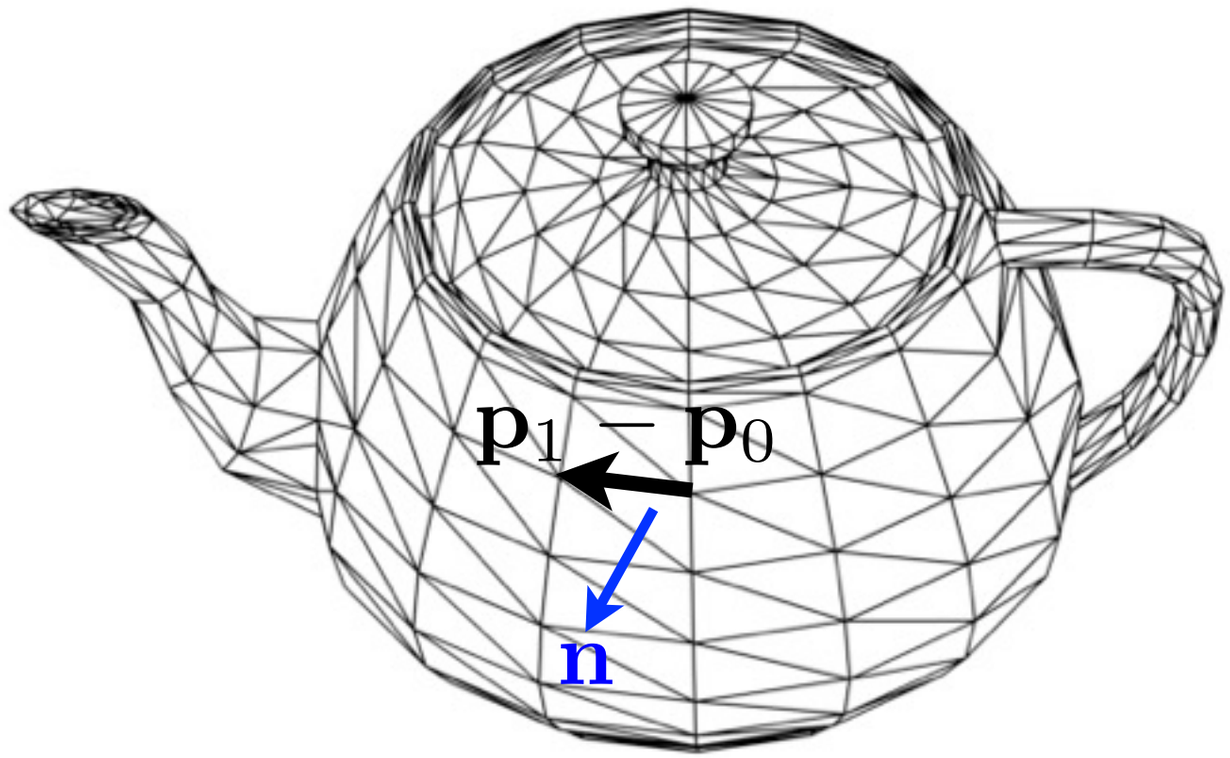
$$(\mathbf{n}^T\mathbf{M}^{-1})(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$

$$(\mathbf{n}^T((\mathbf{M}^{-1})^T)^T)(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$

$$((\mathbf{M}^{-1})^T\mathbf{n})^T(\mathbf{M}\mathbf{p}_1 - \mathbf{M}\mathbf{p}_0) = 0$$

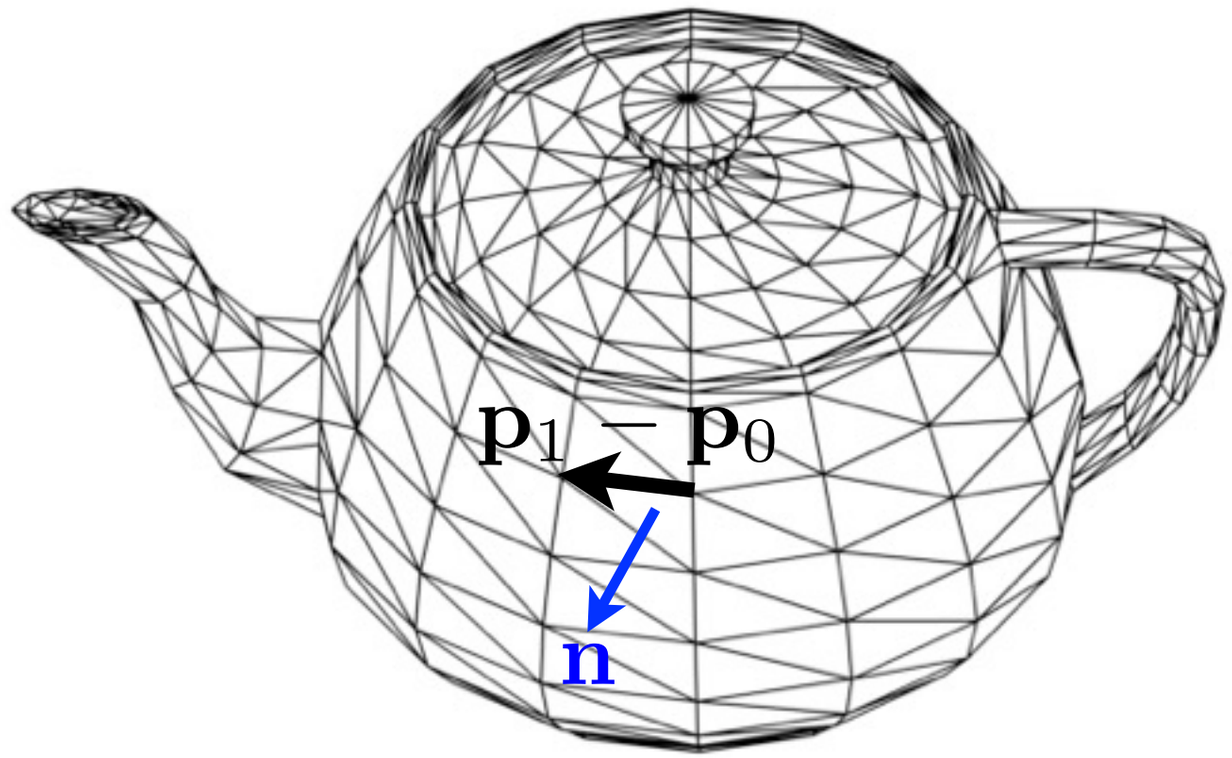
$$(\underline{\mathbf{M}^{-T}\mathbf{n}})^T(\underline{\mathbf{M}\mathbf{p}_1} - \underline{\mathbf{M}\mathbf{p}_0}) = 0$$

$$\mathbf{n}_t \cdot (\mathbf{p}_{t1} - \mathbf{p}_{t0}) = 0$$



n is transformed by $\mathbf{M}^{-\mathsf{T}}$

$$\mathbf{n}_t = \mathbf{M}^{-T}\mathbf{n}$$

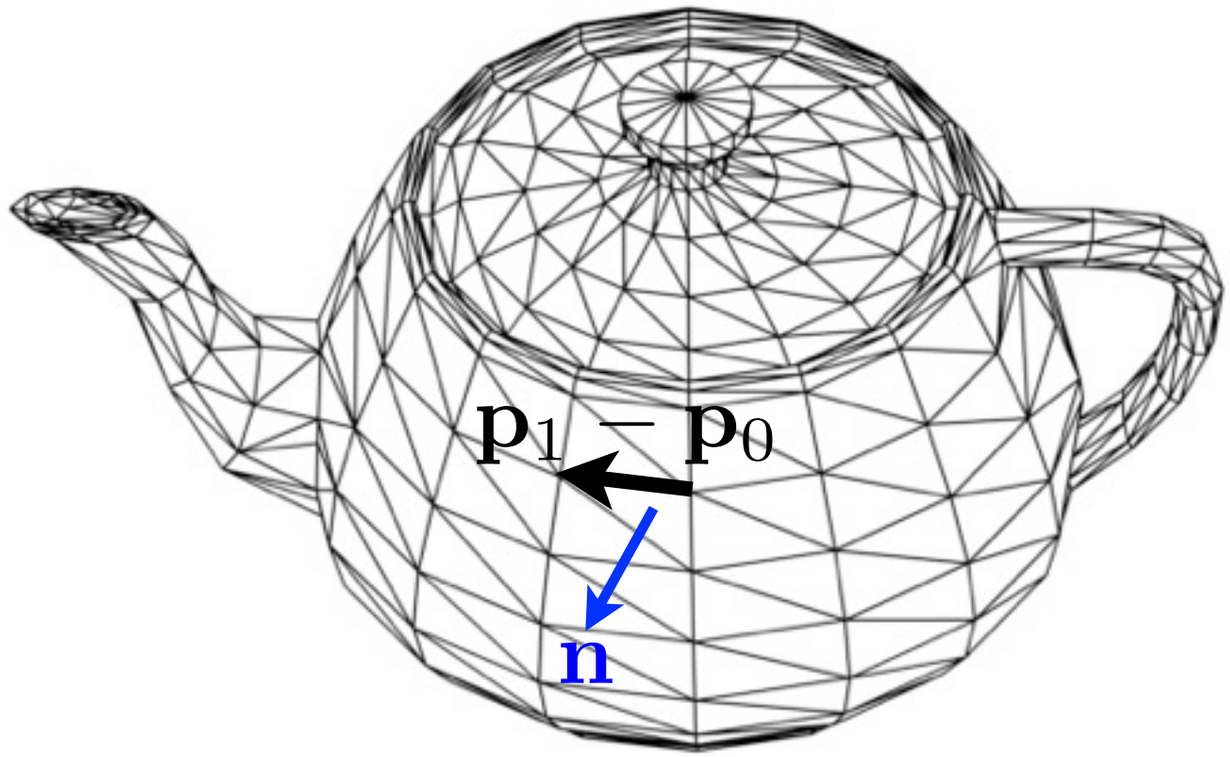


n is transformed by $\mathbf{M}^{-\mathrm{T}}$

$$\mathbf{n}_t = \mathbf{M}^{-T}\mathbf{n}$$

What if **M** is a product of many transformation matrices?

$$\mathbf{M} = \mathbf{M}_3\mathbf{M}_2\mathbf{M}_1$$



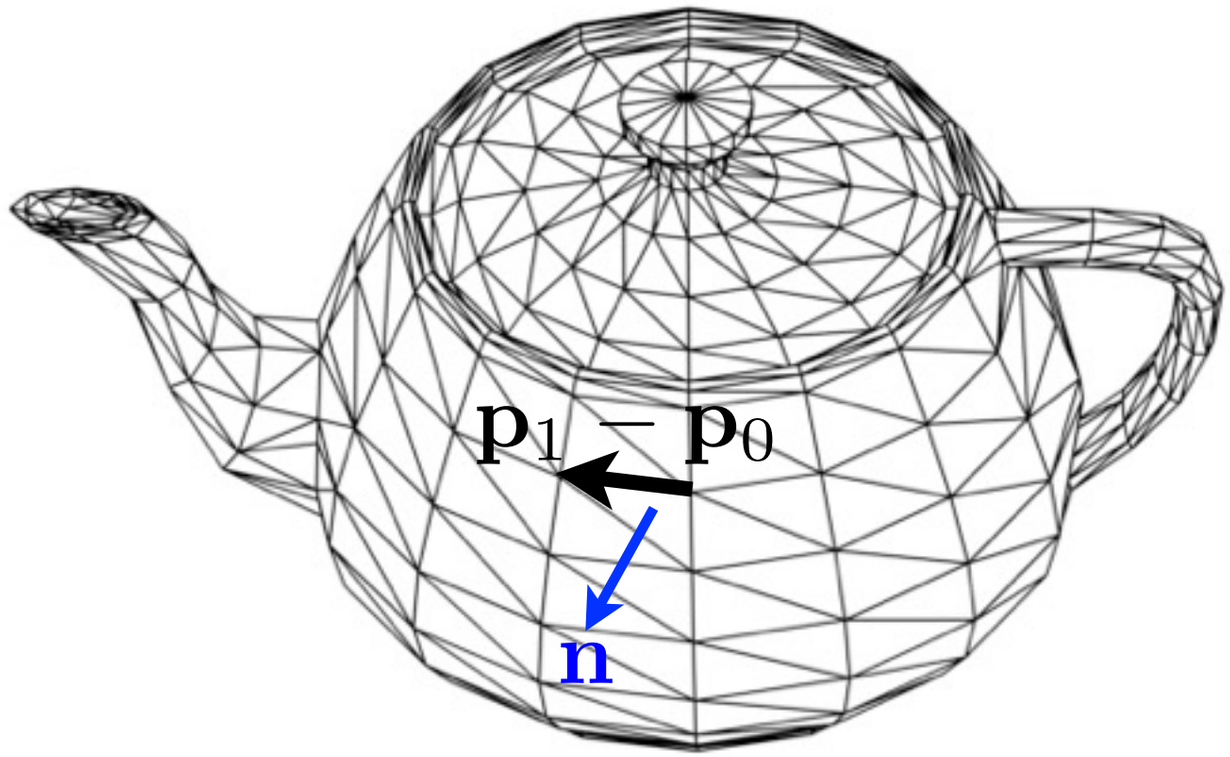
n is transformed by $\mathbf{M}^{-\mathrm{T}}$

$$\mathbf{n}_t = \mathbf{M}^{-T}\mathbf{n}$$

What if **M** is a product of many transformation matrices?

$$\mathbf{M} = \mathbf{M}_3\mathbf{M}_2\mathbf{M}_1$$

$$\mathbf{M}^{-T} = (\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1)^{-T}$$

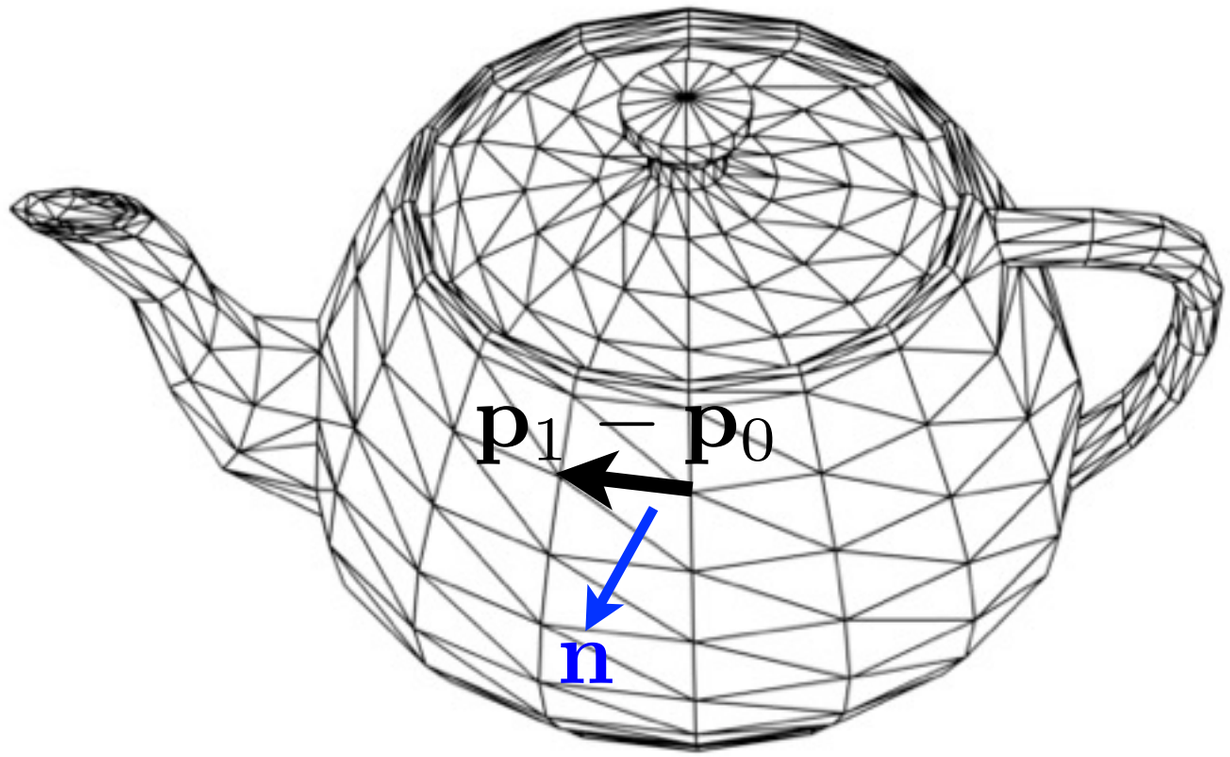


n is transformed by $\mathbf{M}^{-\mathrm{T}}$

$$\mathbf{n}_t = \mathbf{M}^{-T}\mathbf{n}$$

What if **M** is a product of many transformation matrices?

$$\mathbf{M} = \mathbf{M}_3\mathbf{M}_2\mathbf{M}_1$$
$$\mathbf{M}^{-T} = (\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1)^{-T}$$
$$\mathbf{M}^{-T} = ((\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1)^{-1})^{T}$$



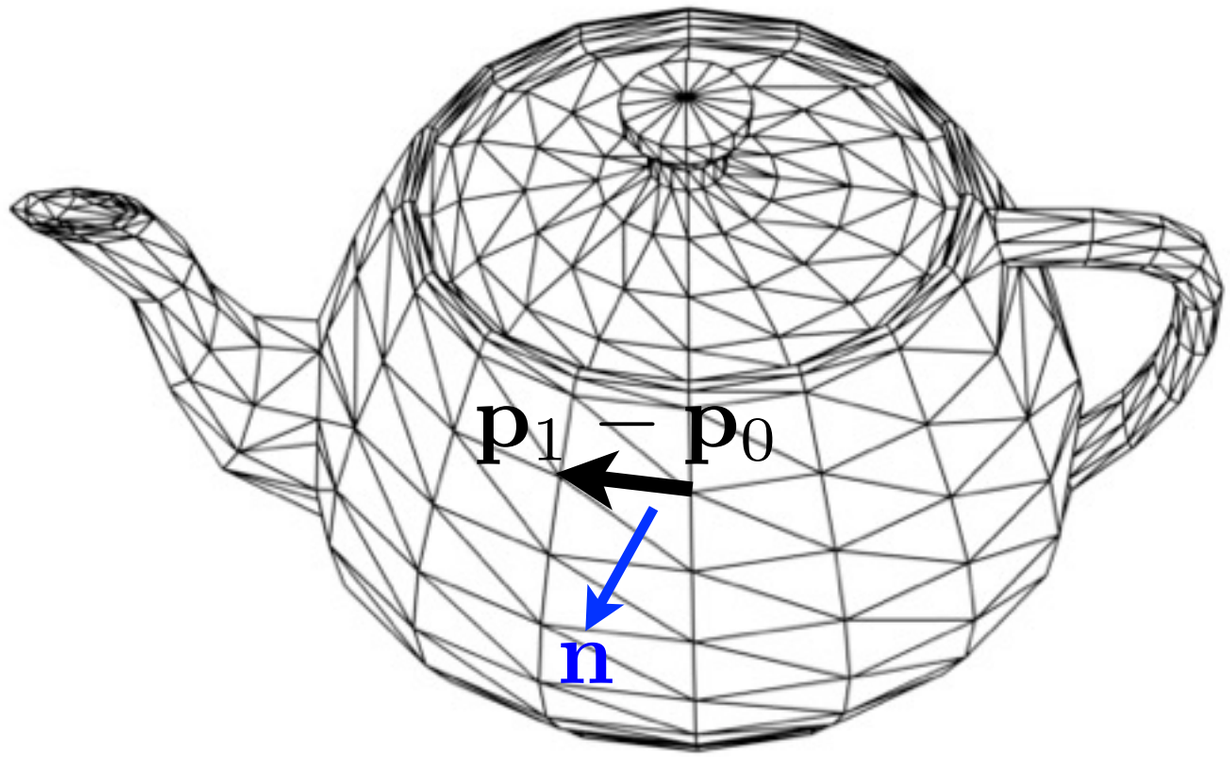
n is transformed by $\mathbf{M}^{-\mathrm{T}}$

$$\mathbf{n}_t = \mathbf{M}^{-T}\mathbf{n}$$

Spread inverse over all matrices. Matrix order is reversed.

$(\mathbf{A}\ \mathbf{B}\ \mathbf{C})^{-1} = \mathbf{C}^{-1}\ \mathbf{B}^{-1}\ \mathbf{A}^{-1}$

$$\mathbf{M} = \mathbf{M}_3\mathbf{M}_2\mathbf{M}_1$$
$$\mathbf{M}^{-T} = (\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1)^{-T}$$
$$\mathbf{M}^{-T} = ((\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1)^{-1})^{T}$$
$$\mathbf{M}^{-T} = ((\mathbf{M}_1)^{-1}(\mathbf{M}_2)^{-1}(\mathbf{M}_3)^{-1})^{T}$$



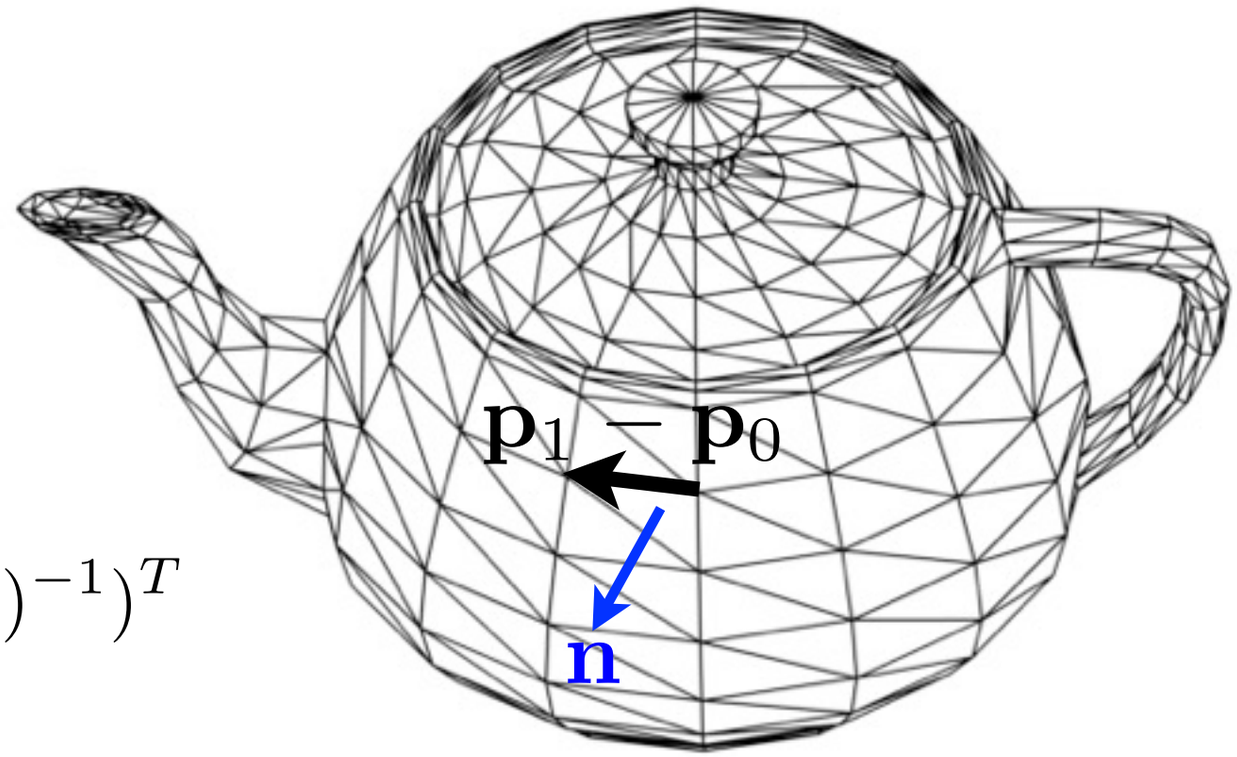
n is transformed by $\mathbf{M}^{-\mathsf{T}}$

$$\mathbf{n}_t = \mathbf{M}^{-T}\mathbf{n}$$

Spread transpose over all matrices. Matrix order is reversed (again).

(**P Q R**)$^{\mathsf{T}}$ = **R**$^{\mathsf{T}}$ **Q**$^{\mathsf{T}}$ **P**$^{\mathsf{T}}$

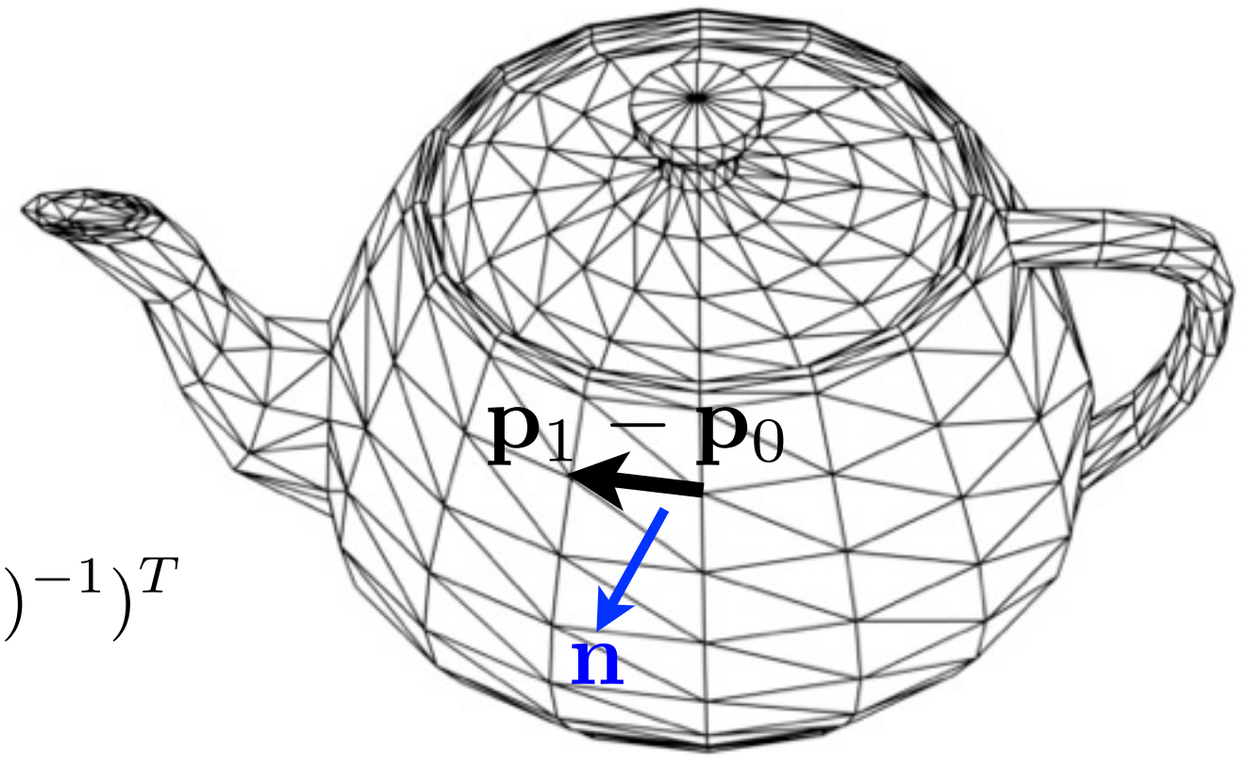
$$\begin{aligned}
\mathbf{M} &= \mathbf{M}_3\mathbf{M}_2\mathbf{M}_1 \\
\mathbf{M}^{-T} &= (\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1)^{-T} \\
\mathbf{M}^{-T} &= ((\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1)^{-1})^T \\
\mathbf{M}^{-T} &= ((\mathbf{M}_1)^{-1}(\mathbf{M}_2)^{-1}(\mathbf{M}_3)^{-1})^T \\
\mathbf{M}^{-T} &= ((\mathbf{M}_3)^{-1})^T((\mathbf{M}_2)^{-1})^T((\mathbf{M}_3)^{-1})^T
\end{aligned}$$



n is transformed by $\mathbf{M}^{-\mathsf{T}}$

$$\mathbf{n}_t = \mathbf{M}^{-T}\mathbf{n}$$

$$\begin{aligned}
\mathbf{M} &= \mathbf{M}_3\mathbf{M}_2\mathbf{M}_1 \\
\mathbf{M}^{-T} &= (\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1)^{-T} \\
\mathbf{M}^{-T} &= ((\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1)^{-1})^T \\
\mathbf{M}^{-T} &= ((\mathbf{M}_1)^{-1}(\mathbf{M}_2)^{-1}(\mathbf{M}_3)^{-1})^T \\
\mathbf{M}^{-T} &= ((\mathbf{M}_3)^{-1})^T((\mathbf{M}_2)^{-1})^T((\mathbf{M}_3)^{-1})^T \\
\mathbf{M}^{-T} &= (\mathbf{M}_3)^{-T}(\mathbf{M}_2^{-T})(\mathbf{M}_1)^{-T}
\end{aligned}$$

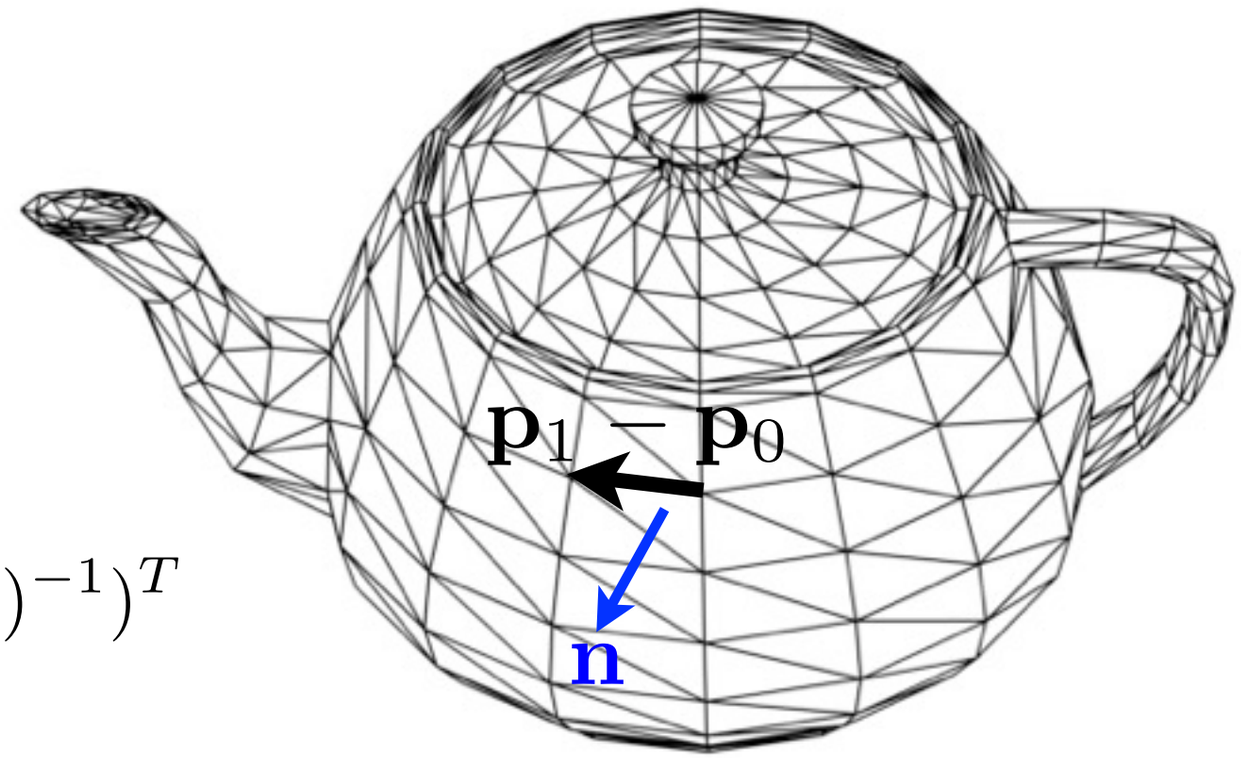


n is transformed by $\mathbf{M}^{-\mathrm{T}}$

$$\mathbf{n}_t = \mathbf{M}^{-T}\mathbf{n}$$

Make sure you correctly set up $\mathbf{M}^{-\mathrm{T}}$ in your vertex shader.

$$\mathbf{M} = \mathbf{M}_3\mathbf{M}_2\mathbf{M}_1$$
$$\mathbf{M}^{-T} = (\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1)^{-T}$$
$$\mathbf{M}^{-T} = ((\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1)^{-1})^{T}$$
$$\mathbf{M}^{-T} = ((\mathbf{M}_1)^{-1}(\mathbf{M}_2)^{-1}(\mathbf{M}_3)^{-1})^{T}$$
$$\mathbf{M}^{-T} = ((\mathbf{M}_3)^{-1})^{T}((\mathbf{M}_2)^{-1})^{T}((\mathbf{M}_3)^{-1})^{T}$$
$$\mathbf{M}^{-T} = (\mathbf{M}_3)^{-T}(\mathbf{M}_2^{-T})(\mathbf{M}_1)^{-T}$$



n is transformed by $\mathbf{M}^{-T}$

If **M** is a created using the ‘look at’ method, i.e., if **M** is

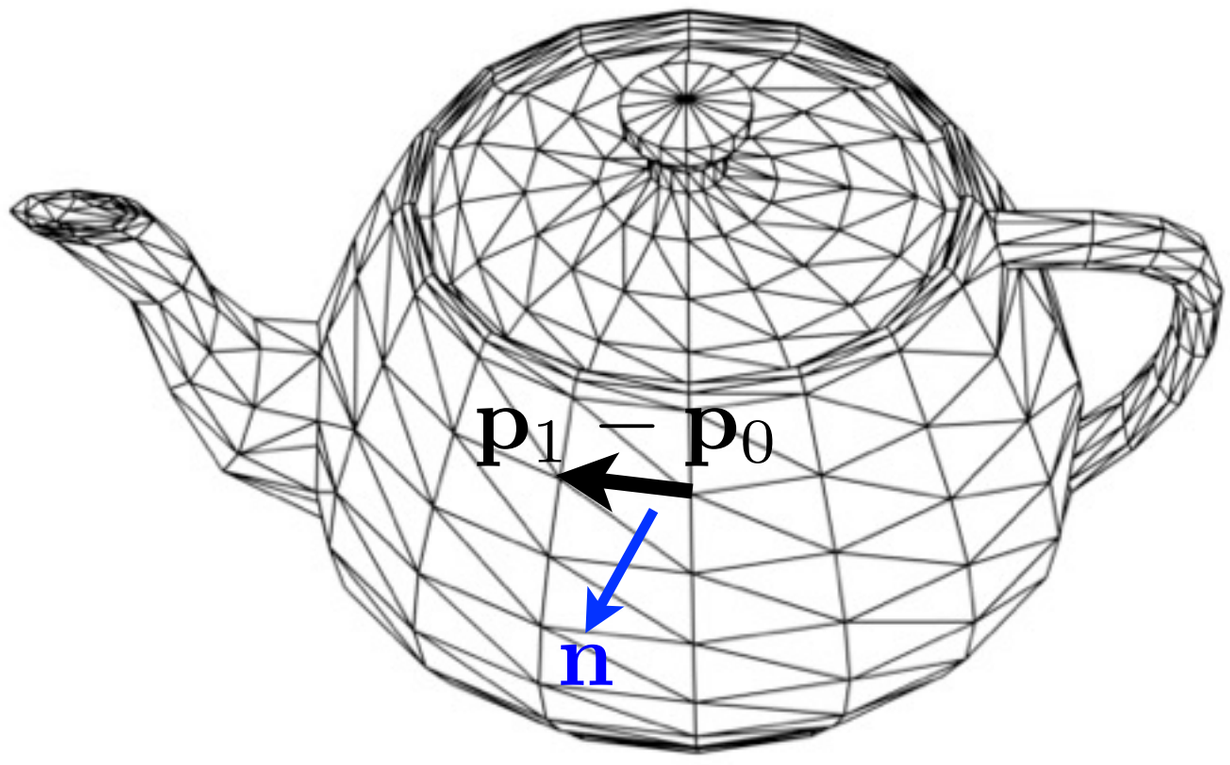
$$\mathbf{M} = \begin{bmatrix} u_x & u_y & u_z & -e_x u_x - e_y u_y - e_z u_z \\ v_x & v_y & v_z & -e_x v_x - e_y v_y - e_z v_z \\ n_x & n_y & n_z & -e_x n_x - e_y n_y - e_z n_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

then $\mathbf{M}^{-1}$ is

$$\mathbf{M}^{-1} = \begin{bmatrix} u_x & v_x & n_x & e_x \\ u_y & v_y & n_y & e_y \\ u_z & v_z & n_z & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and $\mathbf{M}^{-T}$ is

$$\mathbf{M}^{-T} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ e_x & e_y & e_z & 1 \end{bmatrix}$$



n is transformed by $\mathbf{M}^{-T}$

If **M** is a created using the 'look at' method, i.e., if **M** is

$$\mathbf{M} = \begin{bmatrix} u_x & u_y & u_z & -e_x u_x - e_y u_y - e_z u_z \\ v_x & v_y & v_z & -e_x v_x - e_y v_y - e_z v_z \\ n_x & n_y & n_z & -e_x n_x - e_y n_y - e_z n_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

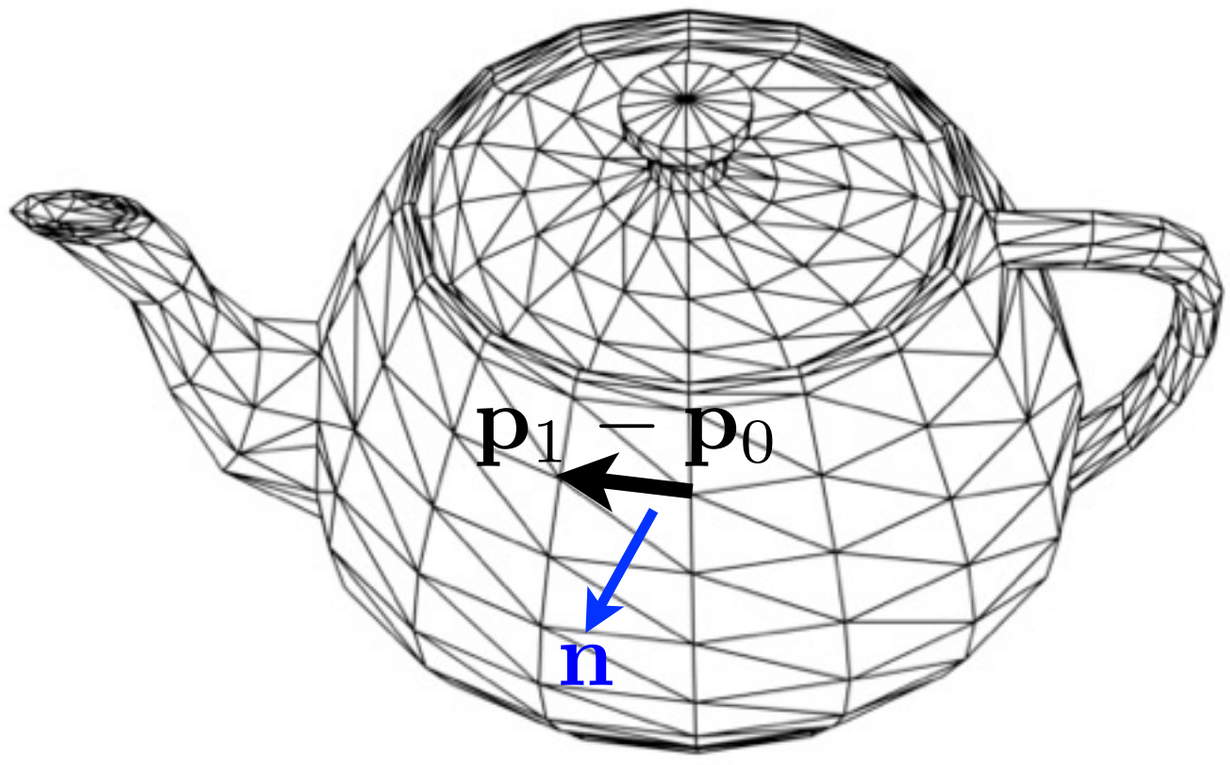
then $\mathbf{M}^{-1}$ is

$$\mathbf{M}^{-1} = \begin{bmatrix} u_x & v_x & n_x & e_x \\ u_y & v_y & n_y & e_y \\ u_z & v_z & n_z & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and $\mathbf{M}^{-T}$ is

$$\mathbf{M}^{-T} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ e_x & e_y & e_z & 1 \end{bmatrix}$$

You can find this in the slides on Viewing.



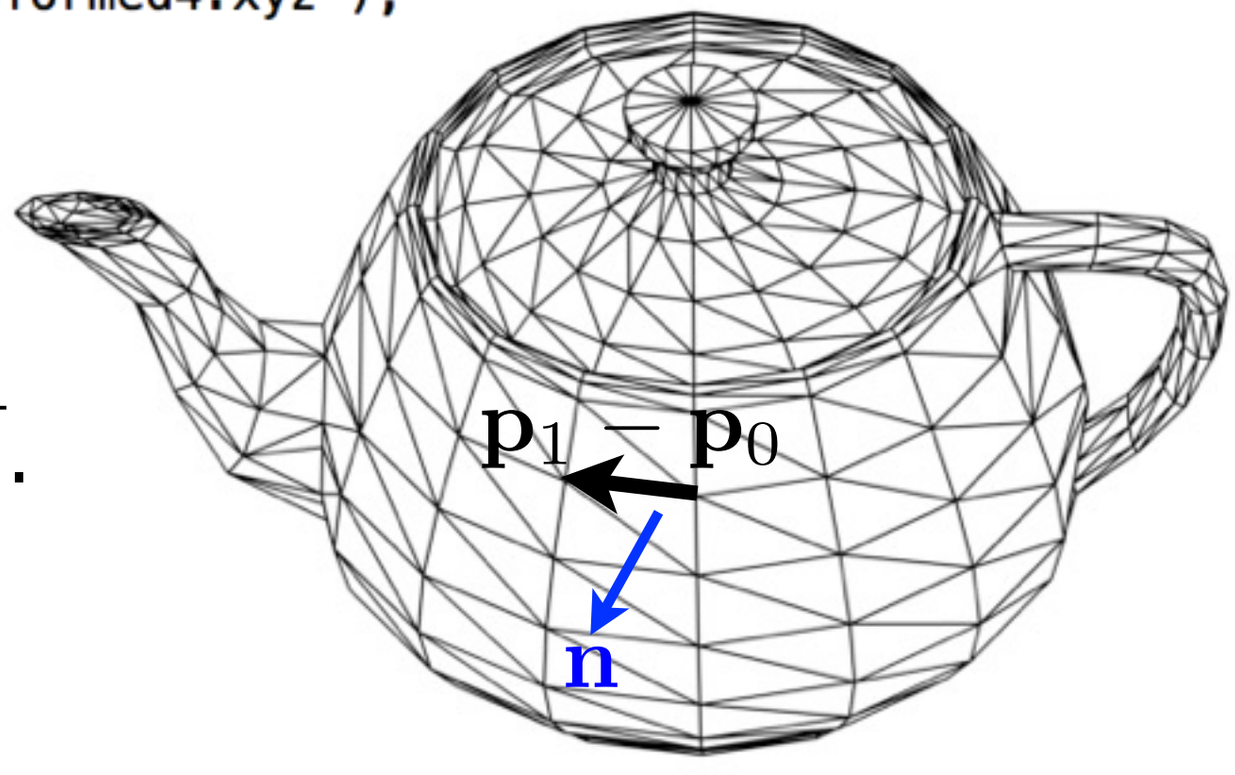
n is transformed by $\mathbf{M}^{-T}$

$$\mathbf{n}_t = \mathbf{M}^{-T}\mathbf{n}$$

You can use the following snippet of code in your vertex shader to transform your normals using $\mathbf{M}^{-T}$.

```
vec4 nv4 = vec4( nv.x, nv.y, nv.z, 1.0 );
vec4 nvtransformed4 = modelviewInverseTranspose * nv4;
vec3 nvtransformed = normalize( nvtransformed4.xyz );
```

Note that your normal needs to be 4-dimensional to correctly multiply with $\mathbf{M}^{-T}$.



Important: The code snippets for Gouraud and Phong Shading shown earlier do not apply $\mathbf{M}^{-T}$ to the normals, as they use **M** of identity!

In your Lab 4 code, you will have a modelview matrix **M** since you will use the look-at method. So you must apply $\mathbf{M}^{-T}$ to the normals.

```
var vertexPosition = gl.getAttribLocation(myShaderProgram,"vertexPosition");
gl.vertexAttribPointer( vertexPosition, 4, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vertexPosition );

// Insert your code here
```

```
    drawObject();

};
```

```
var vertexPosition = gl.getAttribLocation(myShaderProgram,"vertexPosition");
gl.vertexAttribPointer( vertexPosition, 4, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vertexPosition );

// Insert your code here

// set up uniform for model view matrix using look at method, by using
// eye point, at point, and up vector, and computing u, v, and n for camera.
```

```
    drawObject();

};
```

```
var vertexPosition = gl.getAttribLocation(myShaderProgram,"vertexPosition");
gl.vertexAttribPointer( vertexPosition, 4, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vertexPosition );

// Insert your code here

// set up uniform for model view matrix using look at method, by using
// eye point, at point, and up vector, and computing u, v, and n for camera.

 // call function to compute face normals
```

```
    drawObject();

};

// write a function to compute face normals from vertices and faces
// function should iterate over every face,
// get the three vertices p0, p1 and p2 each face points to (every triplet in indexList),
// compute the vectors v1 and v2,
// get the cross product between v1 and v2,
// and normalize the cross product to get a unit face normal.
// Use the functions ‘cross’ and ‘normalize’ from MV.js.
```

```
var vertexPosition = gl.getAttribLocation(myShaderProgram,"vertexPosition");
gl.vertexAttribPointer( vertexPosition, 4, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vertexPosition );

// Insert your code here

// set up uniform for model view matrix using look at method, by using
// eye point, at point, and up vector, and computing u, v, and n for camera.

 // call function to compute face normals

// call function to compute vertex normals
```

```
    drawObject();

};

// write a function to compute face normals from vertices and faces
// function should iterate over every face,
// get the three vertices p0, p1 and p2 each face points to (every triplet in indexList),
// compute the vectors v1 and v2,
// get the cross product between v1 and v2,
// and normalize the cross product to get a unit face normal.
// Use the functions ‘cross’ and ‘normalize’ from MV.js.

// write a function to compute vertex normals from face normals
// function should iterate over every vertex,
// for each vertex, iterate over every face, and see if the vertex belongs to that face,
// if it does, update a running sum for the vertex normal using that face normal,
// at the end, normalize the vertex normal.
```

```
var vertexPosition = gl.getAttribLocation(myShaderProgram,"vertexPosition");
gl.vertexAttribPointer( vertexPosition, 4, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vertexPosition );

// Insert your code here

// set up uniform for model view matrix using look at method, by using
// eye point, at point, and up vector, and computing u, v, and n for camera.

 // call function to compute face normals

// call function to compute vertex normals

// set up uniforms for diffuse, ambient, and specular components of light

// set up uniforms for diffuse, ambient, and specular material coefficients & shininess
    drawObject();

};

// write a function to compute face normals from vertices and faces
// function should iterate over every face,
// get the three vertices p0, p1 and p2 each face points to (every triplet in indexList),
// compute the vectors v1 and v2,
// get the cross product between v1 and v2,
// and normalize the cross product to get a unit face normal.
// Use the functions ‘cross’ and ‘normalize’ from MV.js.

// write a function to compute vertex normals from face normals
// function should iterate over every vertex,
// for each vertex, iterate over every face, and see if the vertex belongs to that face,
// if it does, update a running sum for the vertex normal using that face normal,
// at the end, normalize the vertex normal.
```

Gouraud shading

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;
// coefficients for object
uniform vec3 ka, kd, ks;
// shininess for specular light
uniform float alpha;

// attenuated incident light components
vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
vec3 i, v;

// final reflected light
// (interpolated to fragment shader)
varying vec3 R;

mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);

    vec3 Ra, Rd, Rs; // reflected light components

    // Ambient Reflection
    Ra.r = ka.r * Ia_pp0.r;
    Ra.g = ka.g * Ia_pp0.g;
    Ra.b = ka.b * Ia_pp0.b;

    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );
    float costheta = dot( i, nv );
    Rd.r = kd.r * Id_pp0.r * max( costheta, 0.0);
    Rd.g = kd.g * Id_pp0.g * max( costheta, 0.0);
    Rd.b = kd.b * Id_pp0.b * max( costheta, 0.0);

    // Specular Reflection
    vec3 r = normalize(2.0 * costheta * nv - i);
    normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);
    ( r, v );
    max( cosphi, 0.0 ), alpha );
    floor( 0.5 * (sign(costheta)+1.0) );
    Rs.r = ks.r * Is_pp0.r * shine * costhetag0;
    Rs.g = ks.g * Is_pp0.g * shine * costhetag0;
    Rs.b = ks.b * Is_pp0.b * shine * costhetag0;

    // Final reflection: sum of ambient, diffuse, and specular
    R = clamp( Ra + Rd + Rs, 0.0, 1.0);

    gl_Position=projection * vertexPosition;
}
```

```
vec4 nv4 = vec4( nv.x, nv.y, nv.z, 1.0 );
vec4 nvtransformed4 = modelviewInverseTranspose * nv4;
vec3 nvtransformed = normalize( nvtransformed4.xyz );
```

Replace **nv** with **nvtransformed**

Phong shading

```
precision mediump float;
attribute vec4 vertexPosition;
attribute vec3 nv; // vertex normal

// uniforms:
// point light source location
uniform vec3 p0;
// incident light components
uniform vec3 Ia, Id, Is;



// attenuated incident light components
varying vec3 Ia_pp0, Id_pp0, Is_pp0;
// unit vectors for source direction and view
varying vec3 i, v;
// fragment normal
varying vec3 n;



mat4 projection = mat4( 1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, -1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0);
```

```
void main() {
    gl_PointSize = 1.0;

    // Compute point light source attenuation
    float distance = length( vertexPosition.xyz - p0 );
    // (.xyz gets first 3 components,
    // length() function gives vector length or norm)

    // Ambient, diffuse, and specular light attenuated:
    Ia_pp0 = Ia / (distance * distance);
    Id_pp0 = Id / (distance * distance);
    Is_pp0 = Is / (distance * distance);

    // Fragment normal at vertex is just vertex normal
    n = nv;



    // Diffuse Reflection
    i = normalize( p0 - vertexPosition.xyz );



    // Specular Reflection

    v = normalize(vec3(0.0,0.0,0.0) - vertexPosition.xyz);

    gl_Position=projection * vertexPosition;
}
```

```
vec4 nv4 = vec4( nv.x, nv.y, nv.z, 1.0 );
vec4 nvtransformed4 = modelviewInverseTranspose * nv4;
vec3 nvtransformed = normalize( nvtransformed4.xyz );
```

Replace **nv** with **nvtransformed**

Improvements to Lighting Calculations

Improve Photorealism of Point Light and Spotlight

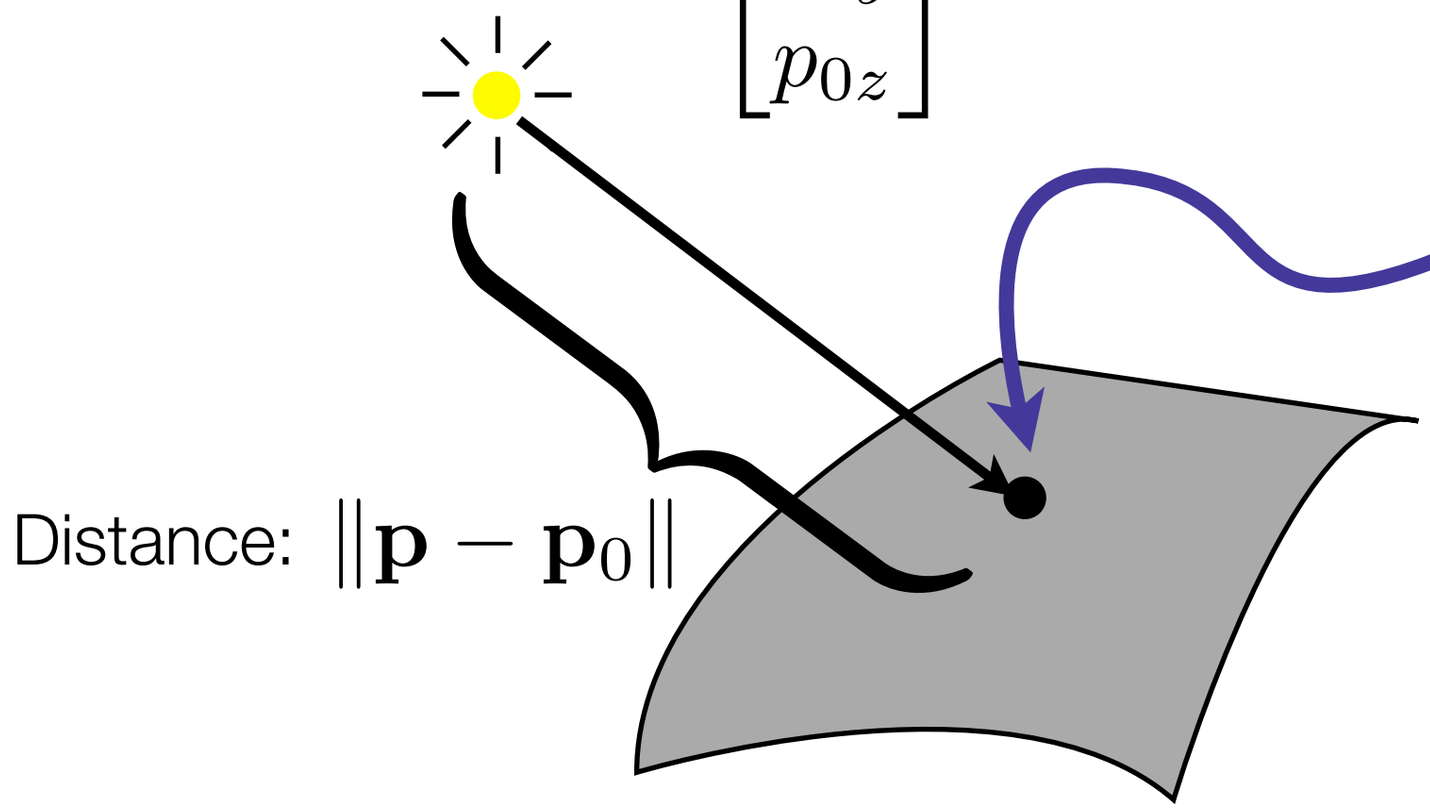
Improve Photorealism of Point Light and Spotlight

Source location

$$\mathbf{p}_0 = \begin{bmatrix} p_{0x} \\ p_{0y} \\ p_{0z} \end{bmatrix}$$

Location of point on surface

$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$



Distance: $\|\mathbf{p} - \mathbf{p}_0\|$

Brightness at source location $\mathbf{p}_0$:

$$\mathbf{I}(\mathbf{p}_0) = \begin{bmatrix} I_r(\mathbf{p}_0) \\ I_g(\mathbf{p}_0) \\ I_b(\mathbf{p}_0) \end{bmatrix}$$

Brightness at point $\mathbf{p}$:

$$\mathbf{I}(\mathbf{p}, \mathbf{p}_0) = \frac{1}{\|\mathbf{p} - \mathbf{p}_0\|^2} \mathbf{I}(\mathbf{p}_0)$$

Dependence on square of distance

Improve Photorealism of Point Light and Spotlight

Point Light:

$$\mathbf{I}(\mathbf{p}, \mathbf{p}_0) = \frac{1}{\|\mathbf{p} - \mathbf{p}_0\|^2} \mathbf{I}(\mathbf{p}_0)$$

Improve Photorealism of Point Light and Spotlight

Point Light:

$$\mathbf{I}(\mathbf{p}, \mathbf{p}_0) = \frac{1}{d^2}\mathbf{I}(\mathbf{p}_0)$$

$$d = \|\mathbf{p} - \mathbf{p}_0\|$$

Improve Photorealism of Point Light and Spotlight

Point Light:

$$\mathbf{I}(\mathbf{p}, \mathbf{p}_0) = \frac{1}{d^2}\mathbf{I}(\mathbf{p}_0)$$

$$d = \|\mathbf{p} - \mathbf{p}_0\|$$

This model often provides harsh lighting.

Improve Photorealism of Point Light and Spotlight

Point Light:

$$\mathbf{I}(\mathbf{p}, \mathbf{p}_0) = \frac{1}{a + bd + cd^2}\mathbf{I}(\mathbf{p}_0)$$

$$d = \|\mathbf{p} - \mathbf{p}_0\|$$

To soften the lighting, folks use $a + bd + cd^2$ instead of d^2. Here, a, b, and c are constants specified by you.

Improve Photorealism of Point Light and Spotlight

Spotlight:

$$\mathbf{I}(\mathbf{p}, \mathbf{p}_0) = \frac{1}{a + bd + cd^2} \mathbf{I}(\mathbf{p}_0) \cos^e \theta$$

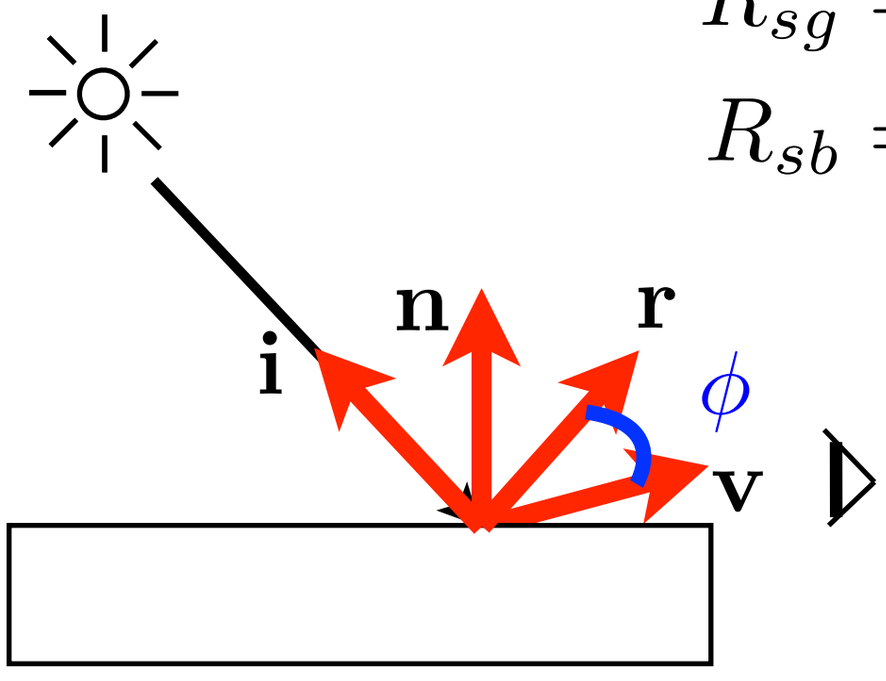
$$d = \|\mathbf{p} - \mathbf{p}_0\|$$

To soften the lighting, folks use $a + bd + cd^2$ instead of d^2. Here, a, b, and c are constants specified by you.

Improve Speed of Specular Lighting Computations

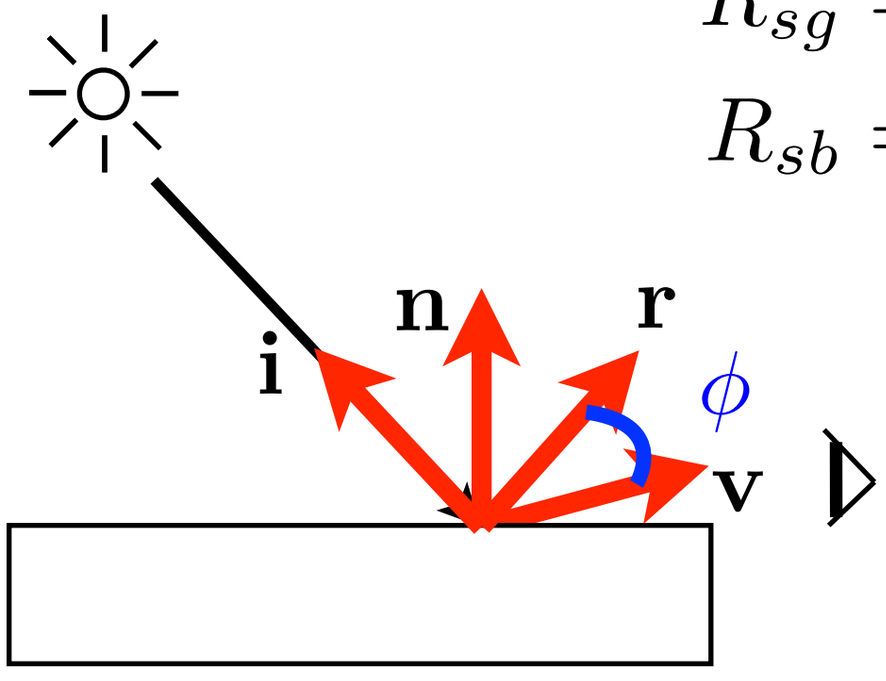
Improve Speed of Specular Lighting Computations

$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$
$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$
$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$
$$\cos\phi = \mathbf{r} \cdot \mathbf{v}$$



Improve Speed of Specular Lighting Computations

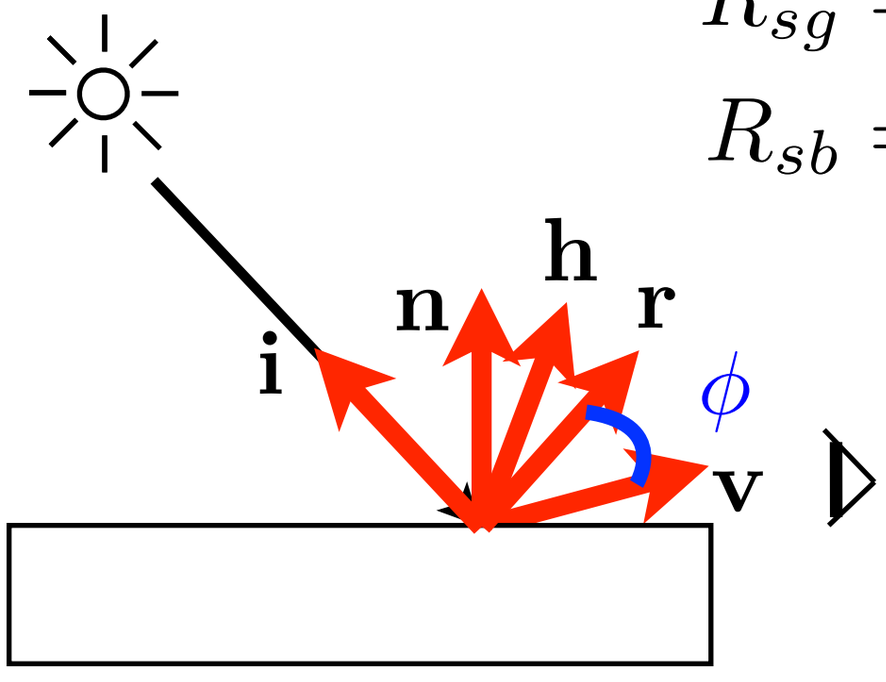
$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$
$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$
$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$
$$\cos\phi = \mathbf{r} \cdot \mathbf{v}$$



Computing **r** can be math-intensive:
remember we do **r** = 2 (**i** . **n**) **n** - **i** in the fragment shader.

Improve Speed of Specular Lighting Computations

$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$
$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$
$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$
$$\cos\phi = \mathbf{r} \cdot \mathbf{v}$$



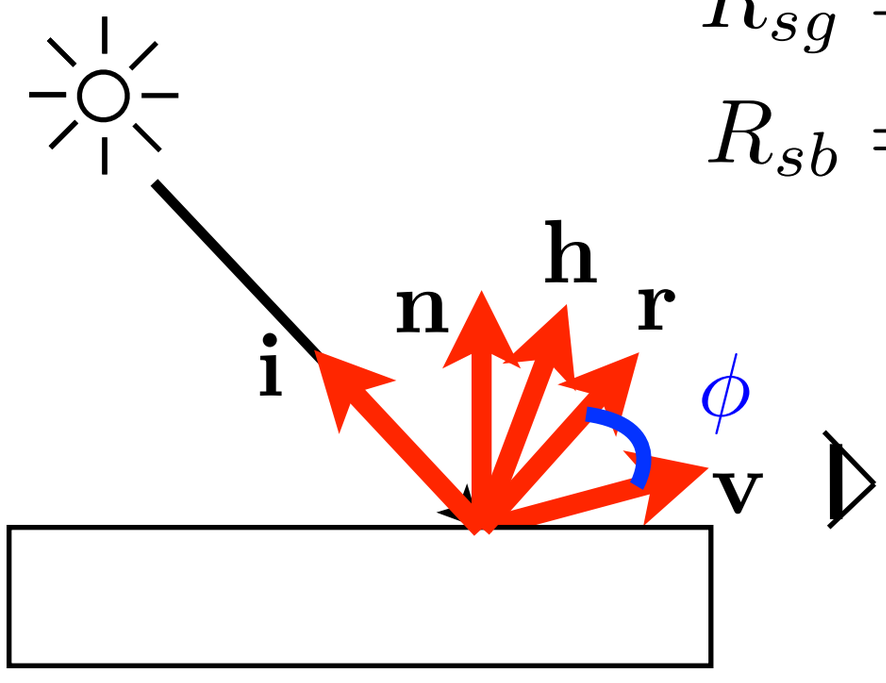
$$\mathbf{h} = \frac{\mathbf{i} + \mathbf{v}}{\|\mathbf{i} + \mathbf{v}\|}$$

We use a different vector **h** called halfway vector.

h is halfway between **i** and **v**.

Improve Speed of Specular Lighting Computations

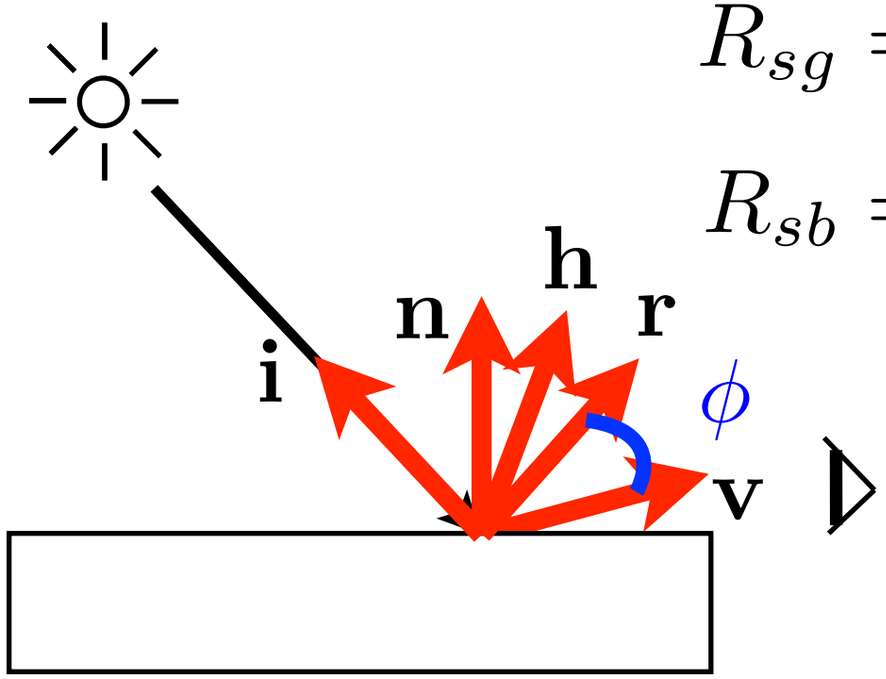
$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sr} \leq 1$$
$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sg} \leq 1$$
$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{r} \cdot \mathbf{v}, 0)\right)^{\alpha}, \ 0 \leq k_{sb} \leq 1$$



$$\mathbf{h} = \frac{\mathbf{i} + \mathbf{v}}{\|\mathbf{i} + \mathbf{v}\|}$$

Similar specularity is obtained using (**n** . **h**)$^{\alpha'}$ instead of (**r** . **v**)$^{\alpha}$.

Improve Speed of Specular Lighting Computations



$$R_{sr} = k_{sr} I_{sr} \left(\max(\mathbf{n} \cdot \mathbf{h}, 0)\right)^{\alpha'}, \ 0 \leq k_{sr} \leq 1$$

$$R_{sg} = k_{sg} I_{sg} \left(\max(\mathbf{n} \cdot \mathbf{h}, 0)\right)^{\alpha'}, \ 0 \leq k_{sg} \leq 1$$

$$R_{sb} = k_{sb} I_{sb} \left(\max(\mathbf{n} \cdot \mathbf{h}, 0)\right)^{\alpha'}, \ 0 \leq k_{sb} \leq 1$$

$$\mathbf{h} = \frac{\mathbf{i} + \mathbf{v}}{\|\mathbf{i} + \mathbf{v}\|}$$

Similar specularity is obtained using (**n** . **h**)$^{\alpha'}$ instead of (**r** . **v**)$^{\alpha}$.

You can compute **h** in the vertex shader and interpolate to fragment shader. Only **n** . **h** is computed in fragment shader.