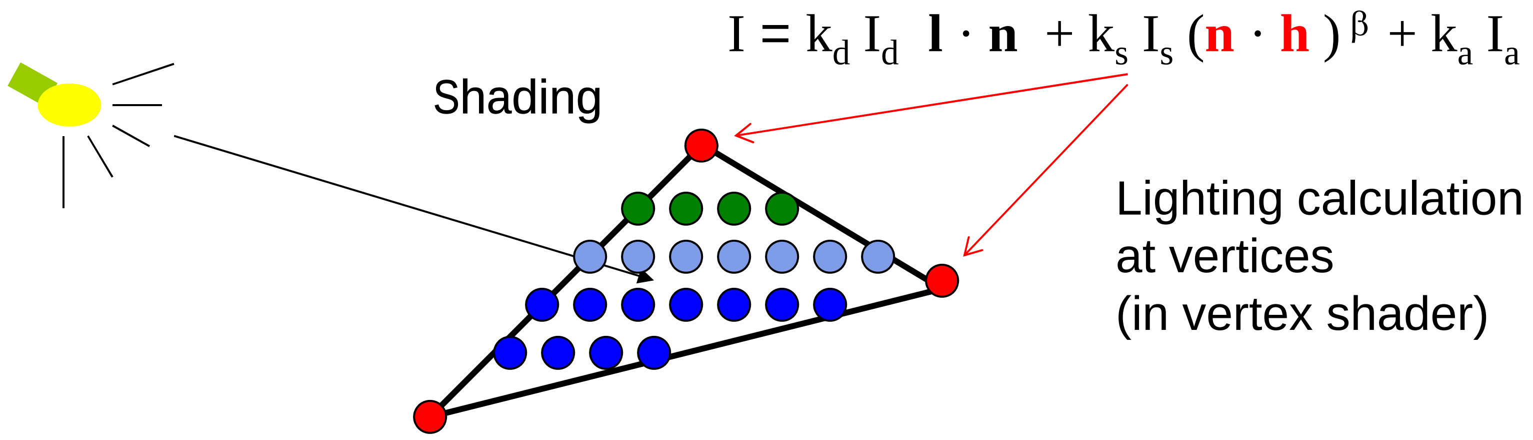
Shading?

- After triangle is rasterized/drawn
  - Per-vertex lighting calculation means we know color of pixels coinciding with vertices (red dots)
- Shading determines color of interior surface pixels



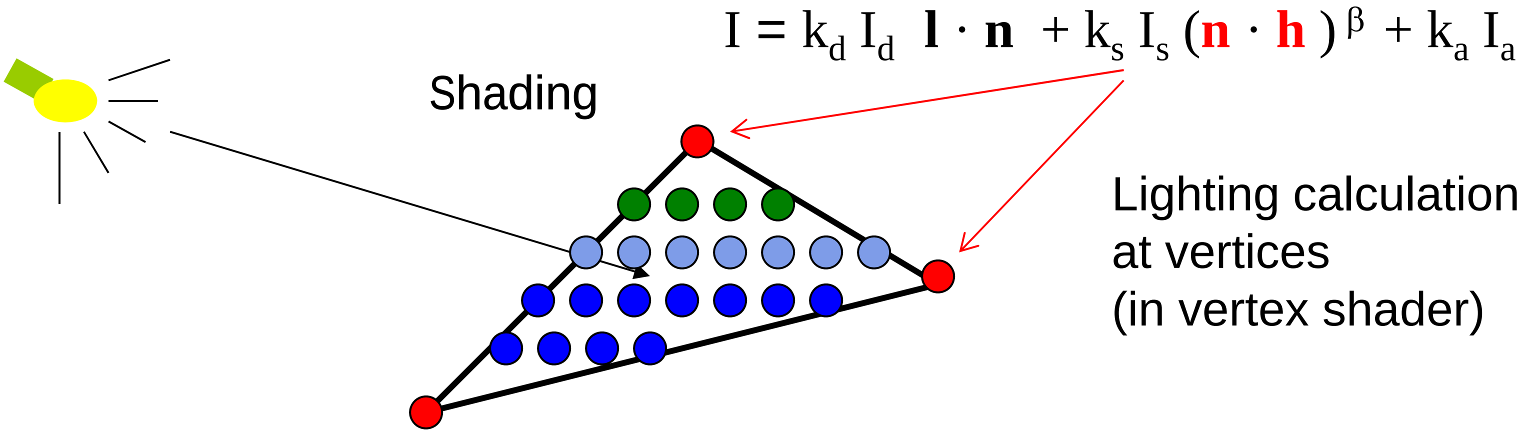
$$I = k_d I_d \; \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$

Shading

Lighting calculation
at vertices
(in vertex shader)

Shading?

- Two types of shading
  - Assume linear change => interpolate (Smooth shading)
  - No interpolation (Flat shading)



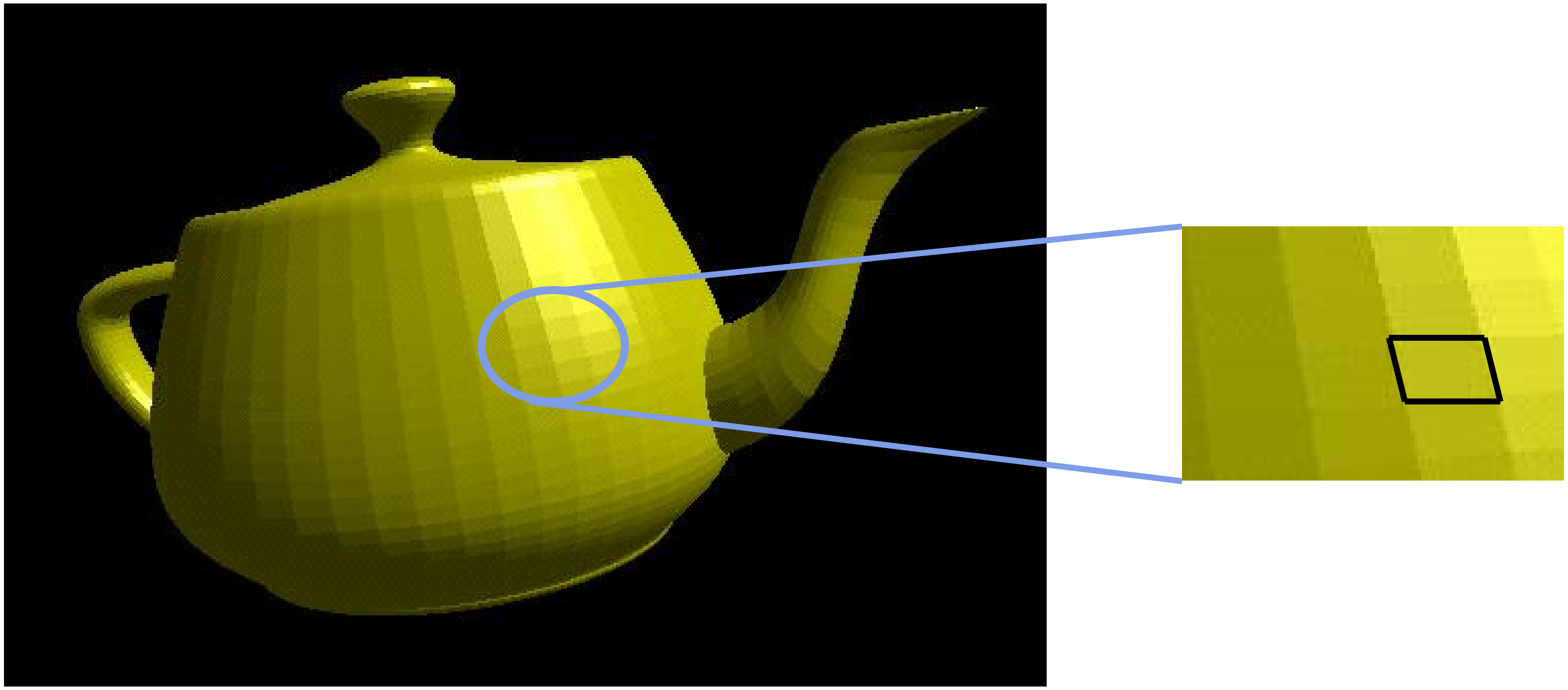
$$I = k_d I_d \; \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$

Shading

Lighting calculation
at vertices
(in vertex shader)

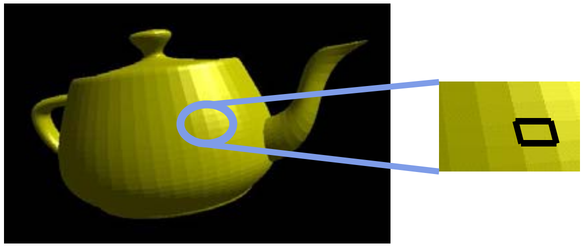
Flat Shading

- compute lighting once for each face, assign color to whole face



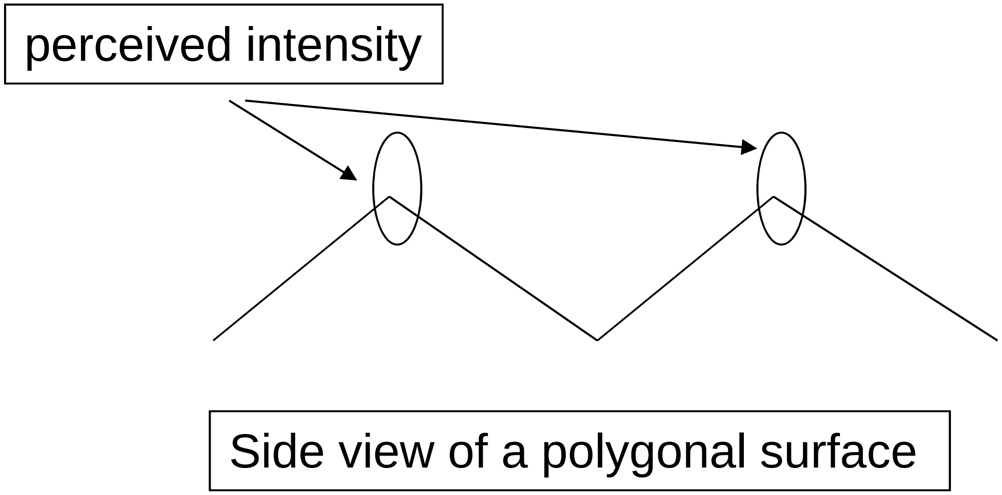
Flat shading

- Only use face normal for all vertices in face and material property to compute color for face
- Benefit: Fast!
- Used when:
  - Polygon is small enough
  - Light source is far away (why?)
  - Eye is very far away (why?)
- Previous OpenGL command: glShadeModel(GL_FLAT) **deprecated!**



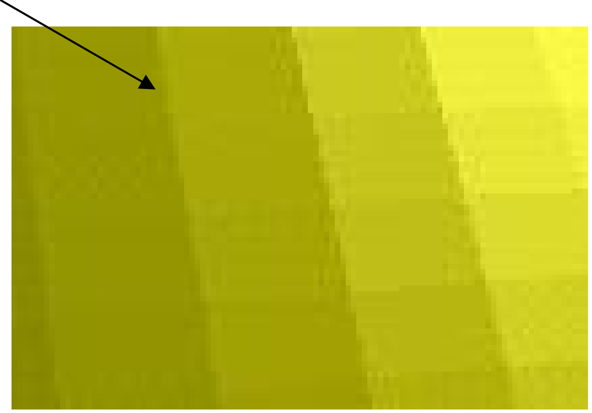
Mach Band Effect

- Flat shading suffers from “mach band effect”
- Mach band effect – human eyes accentuate the discontinuity at the boundary



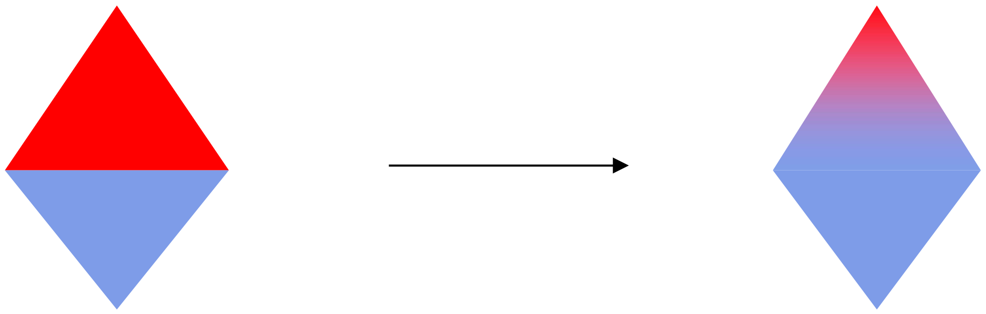
perceived intensity

Side view of a polygonal surface



Smooth shading

- Fix mach band effect – remove edge discontinuity
- Compute lighting for more points on each face
- 2 popular methods:
  - Gouraud shading
  - Phong shading

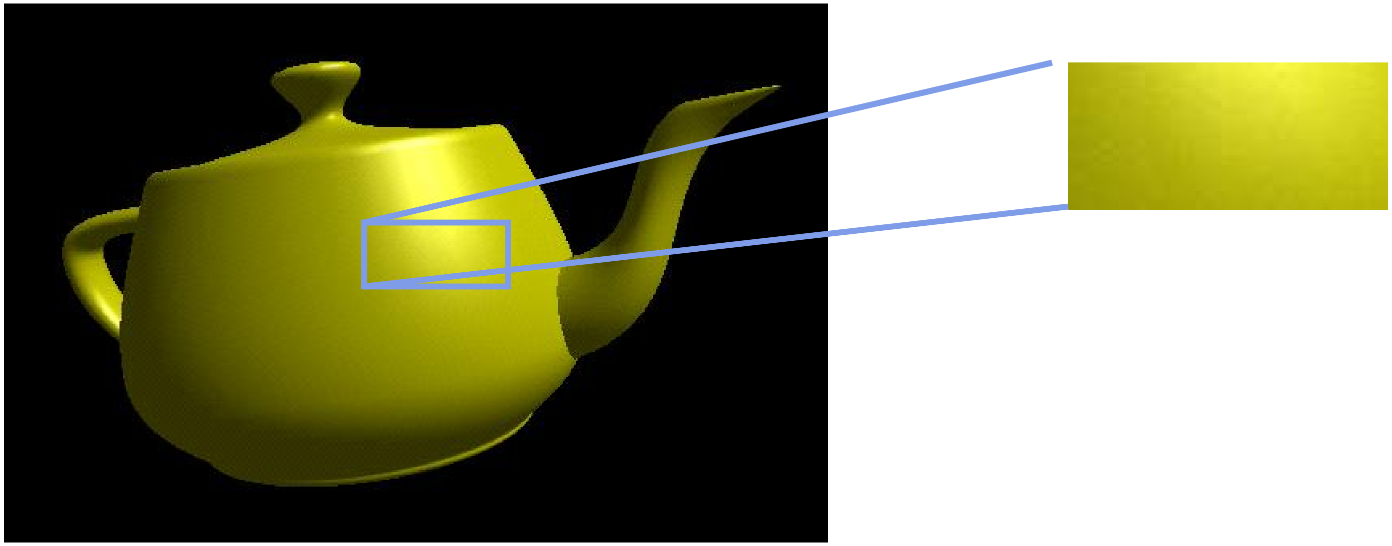


Flat shading

Smooth shading

Gouraud Shading

- Lighting calculated for each polygon vertex
- Colors are interpolated for interior pixels
- Interpolation? Assume linear change from one vertex color to another
- Gouraud shading (interpolation) is OpenGL default



Flat Shading Implementation

- Default is **smooth shading**
- Colors set in vertex shader interpolated
- **Flat shading?** Prevent color interpolation
- In vertex shader, add keyword **flat** to output **color**

```
flat out vec4 color;  //vertex shade
 ……
   color = ambient + diffuse + specular;
    color.a = 1.0;
```

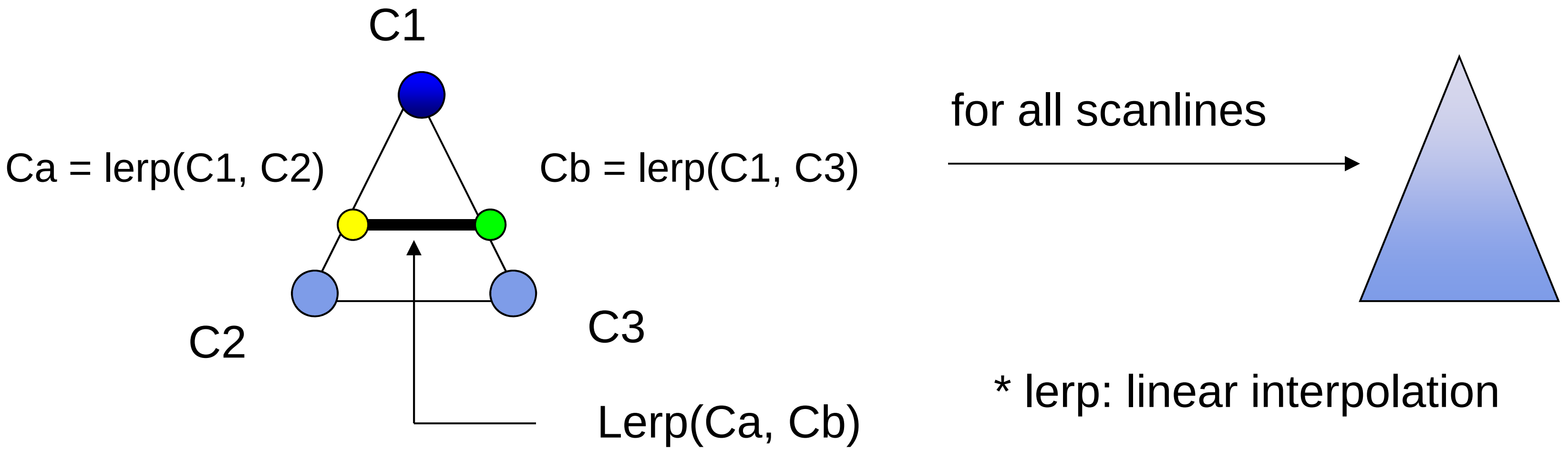
Flat Shading Implementation

- Also, in fragment shader, add keyword **flat** to color received from vertex shader

```
flat in vec4 color;

void main()
{
   gl_FragColor = color;
}
```

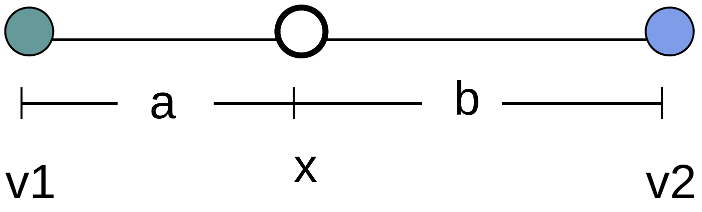
Gouraud Shading

- Compute vertex color in vertex shader
- Shade interior pixels: vertex color **interpolation**



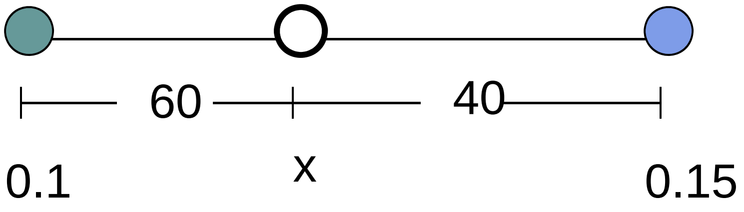
* lerp: linear interpolation

Linear interpolation Example



$$x = \frac{b}{(a+b)} * v1 + \frac{a}{(a+b)} * v2$$

- If a = 60, b = 40
- RGB color at v1 = (0.1, 0.4, 0.2)
- RGB color at v2 = (0.15, 0.3, 0.5)
- Red value of v1 = 0.1, red value of v2 = 0.15

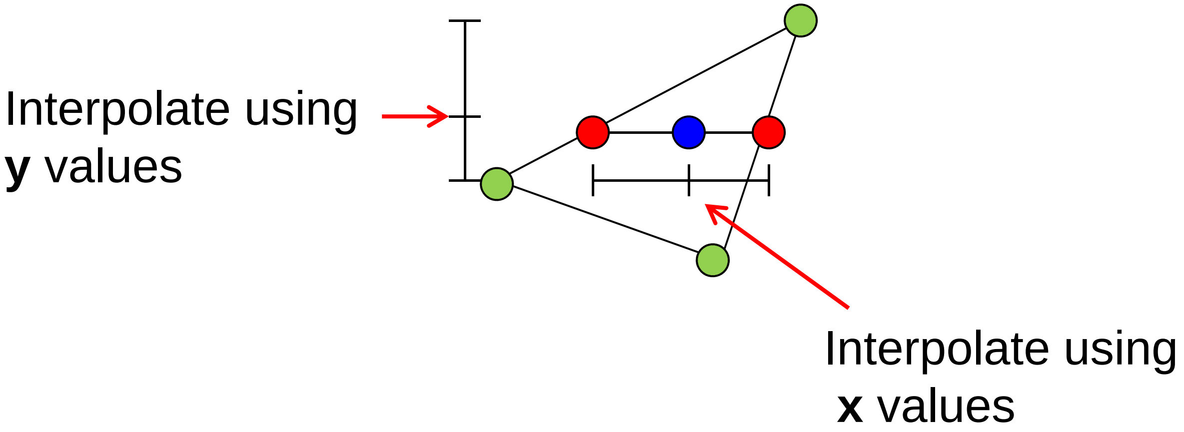


Red value of x = 40 /100 * 0.1 + 60/100 * 0.15
= 0.04 + 0.09 = 0.13

Similar calculations for Green and Blue values

Gouraud Shading

- Interpolate triangle color
  1. Interpolate **y distance** of end points (green dots) to get color of two end points in scanline (red dots)
  2. Interpolate **x distance** of two ends of scanline (red dots) to get color of pixel (blue dot)

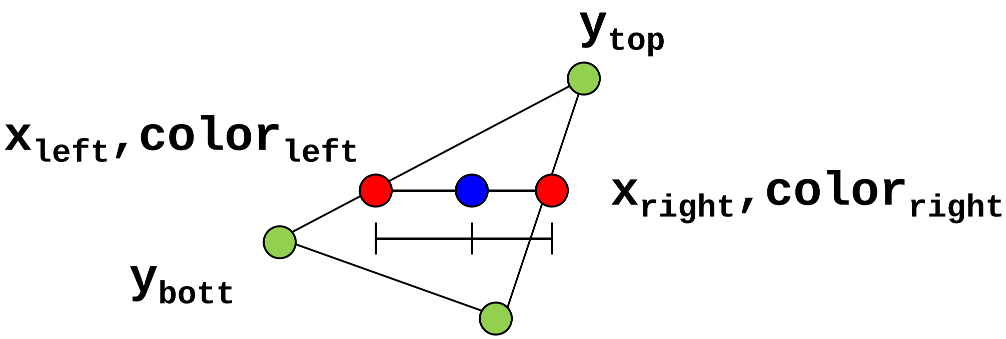


Interpolate using **y** values

Interpolate using **x** values

Gouraud Shading Function (Pg. 433 of Hill)

```
for(int y = $y_{bott}$; y < $y_{top}$; y++) // for each scan line
{
    find $x_{left}$  and $x_{right}$
    find $color_{left}$  and $color_{right}$
    $color_{inc}$ = ($color_{right}$ - $color_{left}$)/ ($x_{right}$ - $x_{left}$)
    for(int x = $x_{left}$, c = $color_{left}$; x < $x_{right}$; x++, c+ = $color_{inc}$)
    {
        put c into the pixel at (x, y)
    }
}
```

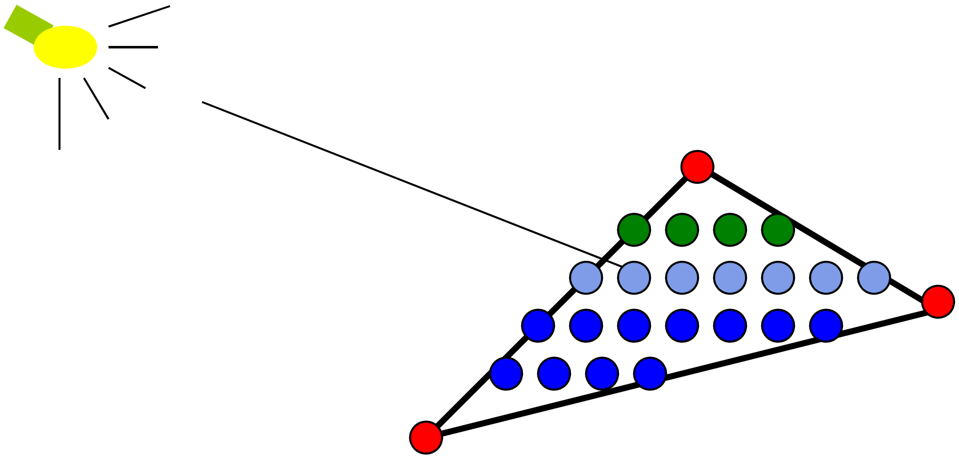


Gouraud Shading Implemenation

- Vertex lighting interpolated across entire face pixels if passed to fragment shader in following way
  1. **Vertex shader:** Calculate output color in vertex shader, Declare output vertex color as **out**

     $$I = k_d I_d \ \mathbf{l} \cdot \mathbf{n} \ + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$

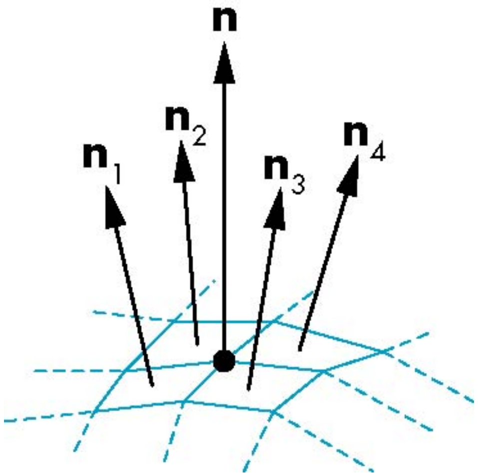
  2. **Fragment shader:** Declare color as **in,** use it, already interpolated!!



Calculating Normals for Meshes

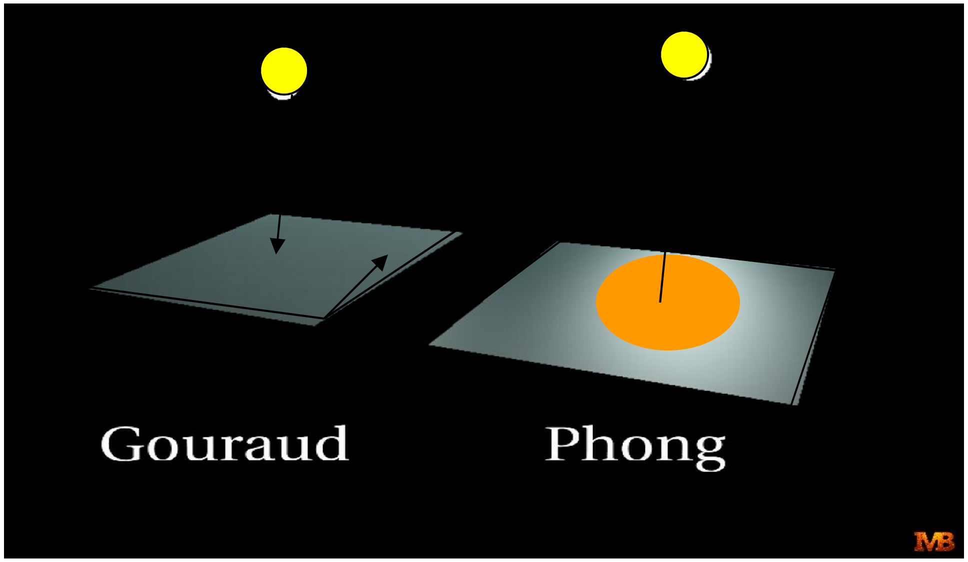
- For meshes, already know how to calculate face normals (e.g. Using Newell method)
- For polygonal models, Gouraud proposed using average of normals around a mesh vertex

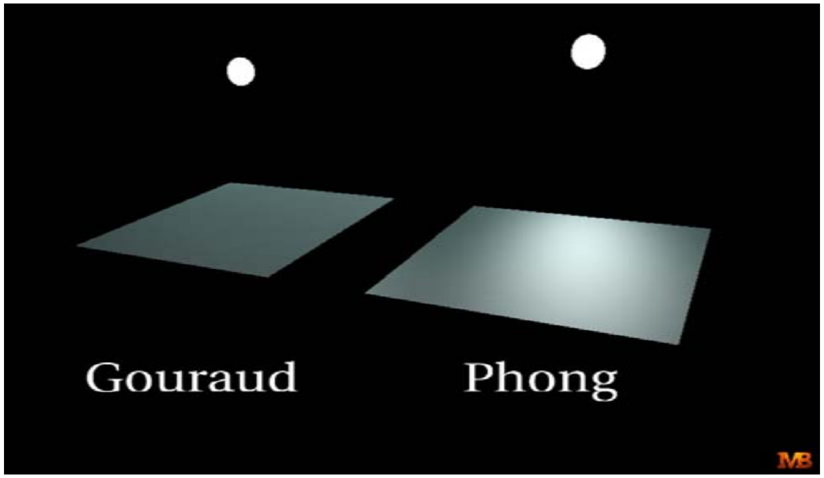
$$\mathbf{n} = (\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4)/\,|\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4|$$



Gouraud Shading Problem

- Assumes linear change across face
- If polygon mesh surfaces have high curvatures, Gouraud shading in polygon interior can be inaccurate
- Phong shading may look smooth





Phong Shading

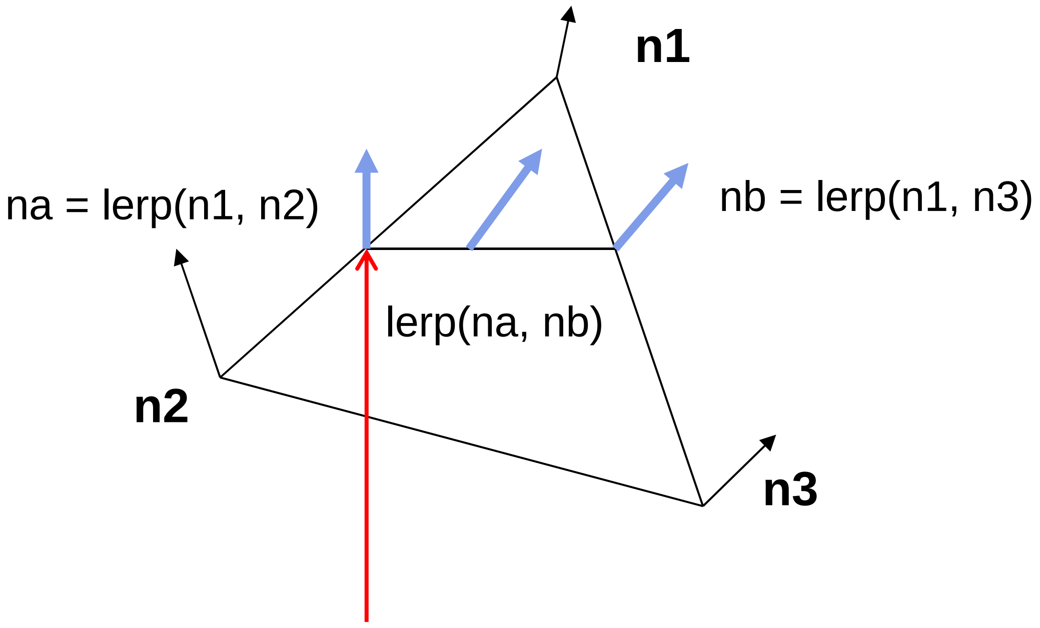
- Need vectors **n, l, v, r** for all pixels – not provided by user

- Instead of interpolating vertex color
  - Interpolate **vertex normal and vectors**
  - Use pixel **vertex normal and vectors** to calculate Phong shading at pixel (**per pixel lighting**)

- Phong shading computes lighting in fragment shader

Phong Shading (Per Fragment)

- Normal interpolation (also interpolate l,v)



At each pixel, need to interpolate
Normals (n) and vectors v and l

Gouraud Vs Phong Shading Comparison

- Phong shading more work than Gouraud shading
  - Move lighting calculation to fragment shaders
  - Just set up vectors (l,n,v,h) in vertex shader

a. Gouraud Shading

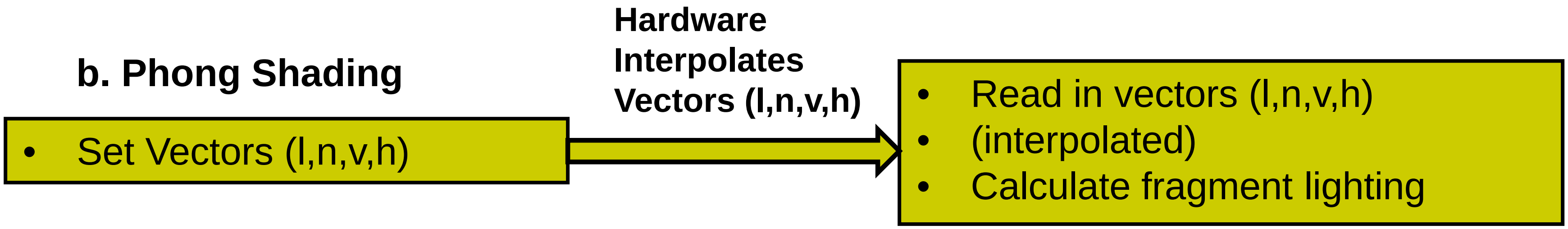
- Set Vectors (l,n,v,h)
- Calculate vertex colors

Hardware Interpolates Vertex color

- Read/set fragment color
- (Already interpolated)

$$I = k_d I_d \; \mathbf{l} \cdot \mathbf{n} \; + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$

b. Phong Shading



- Set Vectors (l,n,v,h)

Hardware Interpolates Vectors (l,n,v,h)

- Read in vectors (l,n,v,h)
- (interpolated)
- Calculate fragment lighting

$$I = k_d I_d \; \mathbf{l} \cdot \mathbf{n} \; + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$

Per-Fragment Lighting Shaders I

```
// vertex shader
in vec4 vPosition;
in vec3 vNormal;

// output values that will be interpolatated per-fragment
out vec3 fN;
out vec3 fE;
out vec3 fL;
```

Declare variables **n, v, l** as **out** in vertex shader

```
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;
```

Per-Fragment Lighting Shaders II

```
void main()
{
   fN = vNormal;
   fE = -vPosition.xyz;
   fL = LightPosition.xyz;

   if( LightPosition.w != 0.0 ) {
         fL = LightPosition.xyz - vPosition.xyz;
   }

   gl_Position = Projection*ModelView*vPosition;
}
```

← Set variables **n, v, l** in vertex shader

Per-Fragment Lighting Shaders III

```
// fragment shader

// per-fragment interpolated values from the vertex shader
in vec3 fN;
in vec3 fL;
in vec3 fE;

uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform float Shininess;
```

Declare vectors n, v, l as **in** in fragment shader
(**Hardware interpolates these vectors**)

Per=Fragment Lighting Shaders IV

```
void main()
{
    // Normalize the input lighting vectors

   vec3 N = normalize(fN);
    vec3 E = normalize(fE);
    vec3 L = normalize(fL);

    vec3 H = normalize( L + E );
    vec4 ambient = AmbientProduct;
```

Use interpolated variables n, v, l in fragment shader

$$I = k_d I_d \; \mathbf{l} \cdot \mathbf{n} \; + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$

Per-Fragment Lighting Shaders V

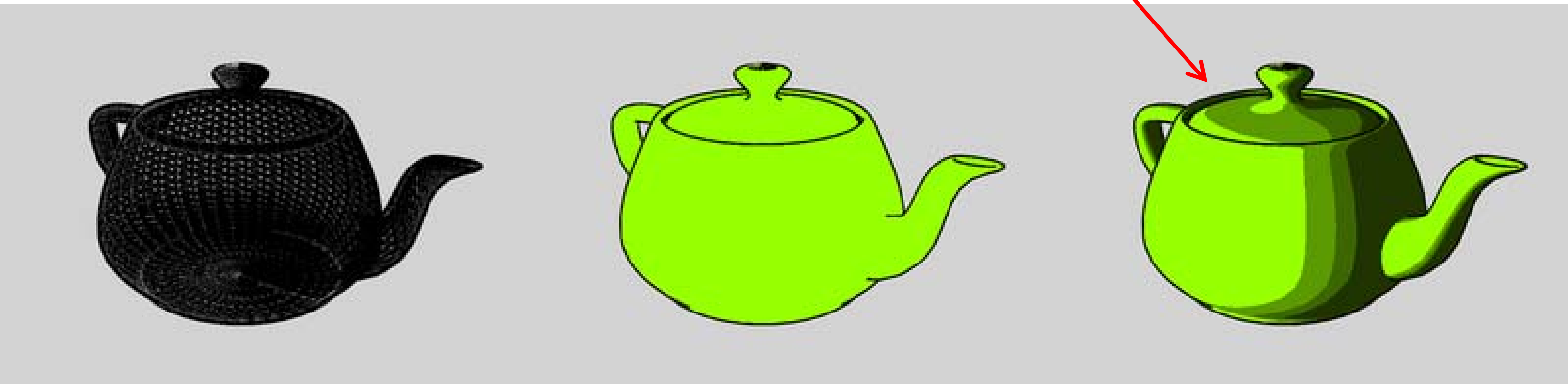
```
float Kd = max(dot(L, N), 0.0);
   vec4 diffuse = Kd*DiffuseProduct;

   float Ks = pow(max(dot(N, H), 0.0), Shininess);
   vec4 specular = Ks*SpecularProduct;

   // discard the specular highlight if the light's behind the vertex
   if( dot(L, N) < 0.0 )
         specular = vec4(0.0, 0.0, 0.0, 1.0);

   gl_FragColor = ambient + diffuse + specular;
   gl_FragColor.a = 1.0;
}
```

Use interpolated variables n, v, l in fragment shader

$$I = k_d I_d \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$

Toon (or Cel) Shading

- Non-Photorealistic (NPR) effect
- Shade in bands of color



Toon (or Cel) Shading

- How?
- Consider **(l · n)** diffuse term (or cos *θ*) term

$$I = k_d I_d \; \mathbf{l} \cdot \mathbf{n} \; + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$

- Clamp values to min value of ranges to get toon shading effect

$\mathbf{l} \cdot \mathbf{n}$	**Value used**
Between 0.75 and 1	0.75
Between 0.5 and 0.75	0.5
Between 0.25 and 0.5	0.25
Between 0.0 and 0.25	0.0

BRDF Evolution

- BRDFs have evolved historically
- 1970’s: Empirical models
  - Phong’s illumination model
- 1980s:
  - Physically based models
  - Microfacet models (e.g. Cook Torrance model)
- 1990’s
  - Physically-based appearance models of specific effects (materials, weathering, dust, etc)
- Early 2000’s
  - Measurement & acquisition of static materials/lights (wood, translucence, etc)
- Late 2000’s
  - Measurement & acquisition of time-varying BRDFs (ripening, etc)

Physically-Based Shading Models

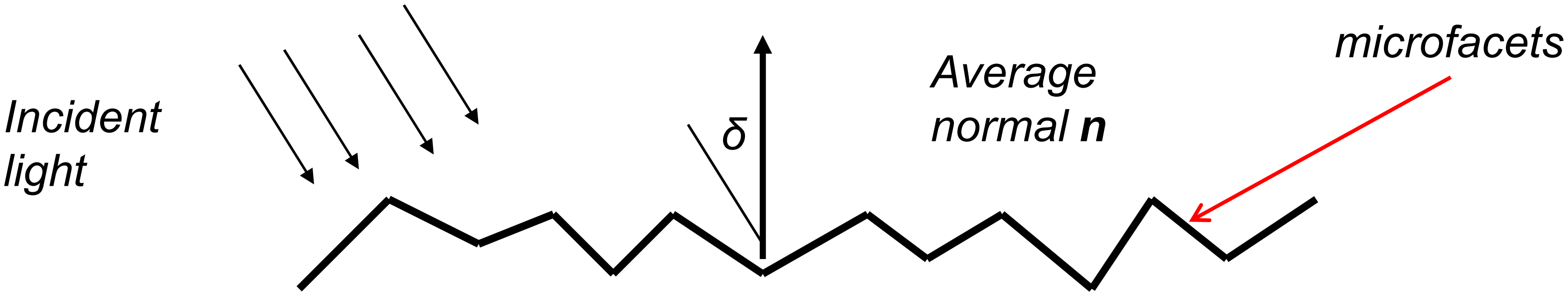
- Phong model produces pretty pictures
- **Cons:** empirical (fudged?) ($cos^{\alpha}\phi$), plastic look
- Shaders can implement better lighting/shading models
- Big trend towards Physically-based lighting models
- Physically-based?
  - Based on physics of how light interacts with actual surface
  - Apply Optics/Physics theories
- Classic: Cook-Torrance shading model (TOGS 1982)

Cook-Torrance Shading Model

- Same ambient and diffuse terms as Phong
- New, better specular component than ($cos^{\alpha}\phi$),

$$\cos^{\alpha}\phi \rightarrow \frac{F(\phi,\eta)DG}{(\mathbf{n}\cdot\mathbf{v})}$$

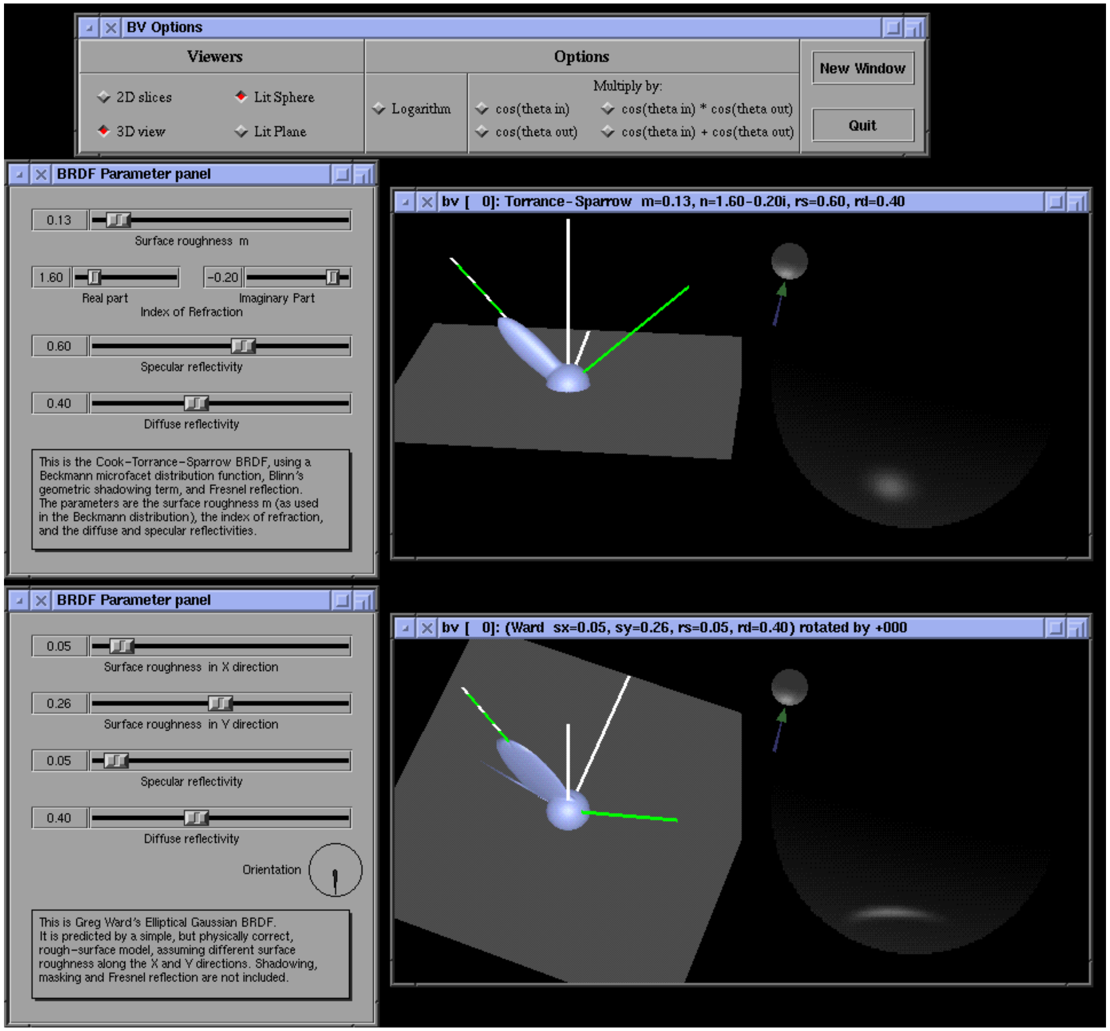
- **Idea:** surfaces has small V-shaped **microfacets (grooves)**



- Many grooves at each surface point
- **Distribution term D:** Grooves facing a direction contribute
- E.g. half of grooves face 30 degrees, etc

BV BRDF Viewer

BRDF viewer (View distribution of light bounce)



BV Options

Viewers: ○ 2D slices ● Lit Sphere ● 3D view ○ Lit Plane

Options: ○ Logarithm; Multiply by: ○ cos(theta in) ○ cos(theta out) ○ cos(theta in) * cos(theta out) ○ cos(theta in) + cos(theta out)

BRDF Parameter panel

0.13 Surface roughness m

1.60 Real part, −0.20 Imaginary Part — Index of Refraction

0.60 Specular reflectivity

0.40 Diffuse reflectivity

This is the Cook−Torrance−Sparrow BRDF, using a Beckmann microfacet distribution function, Blinn's geometric shadowing term, and Fresnel reflection. The parameters are the surface roughness m (as used in the Beckmann distribution), the index of refraction, and the diffuse and specular reflectivities.

bv [0]: Torrance−Sparrow m=0.13, n=1.60−0.20i, rs=0.60, rd=0.40

BRDF Parameter panel

0.05 Surface roughness in X direction

0.26 Surface roughness in Y direction

0.05 Specular reflectivity

0.40 Diffuse reflectivity

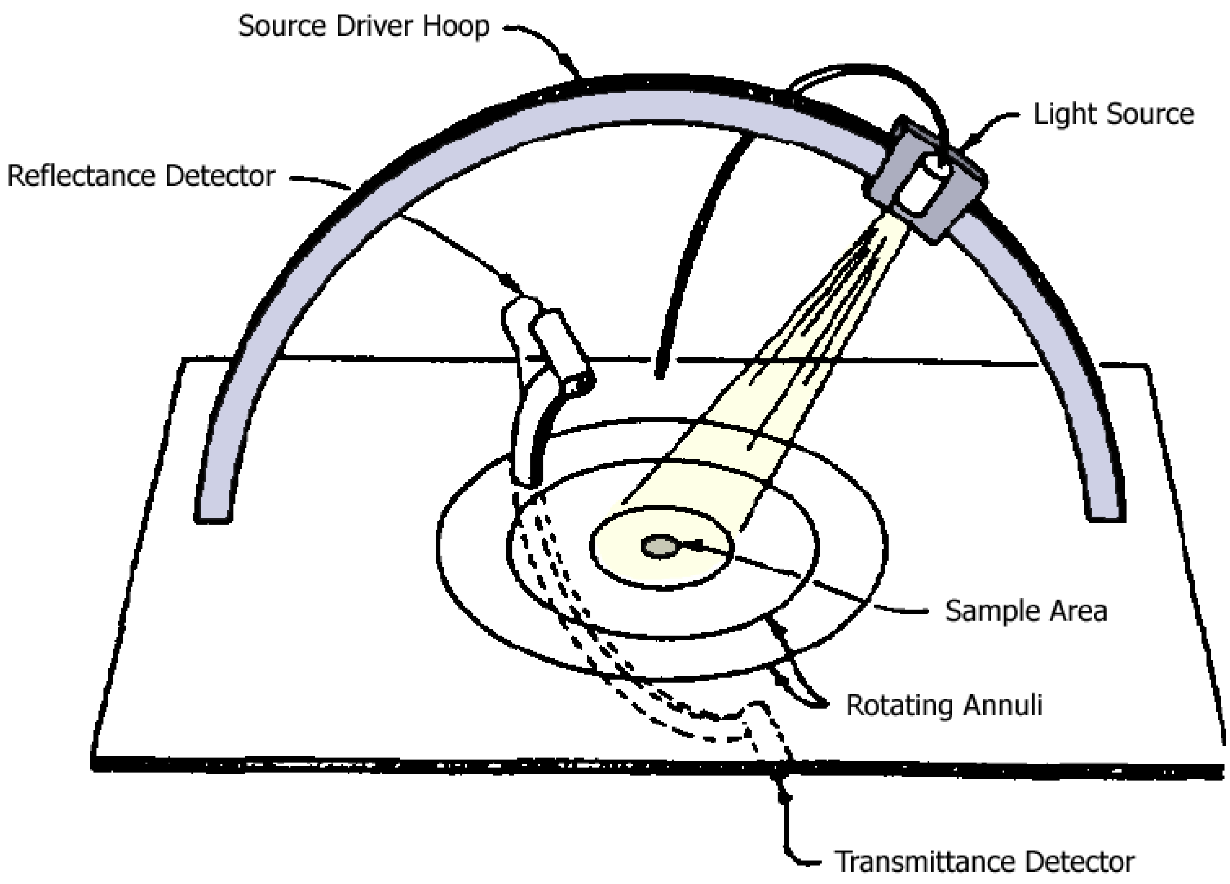
This is Greg Ward's Elliptical Gaussian BRDF. It is predicted by a simple, but physically correct, rough−surface model, assuming different surface roughness along the X and Y directions. Shadowing, masking and Fresnel reflection are not included.

bv [0]: (Ward sx=0.05, sy=0.26, rs=0.05, rd=0.40) rotated by +000

BRDF Evolution

- BRDFs have evolved historically
- 1970's: Empirical models
  - Phong's illumination model
- 1980s:
  - Physically based models
  - Microfacet models (e.g. Cook Torrance model)
- 1990's
  - Physically-based appearance models of specific effects (materials, weathering, dust, etc)
- Early 2000's
  - Measurement & acquisition of static materials/lights (wood, translucence, etc)
- Late 2000's
  - Measurement & acquisition of time-varying BRDFs (ripening, etc)

Measuring BRDFs



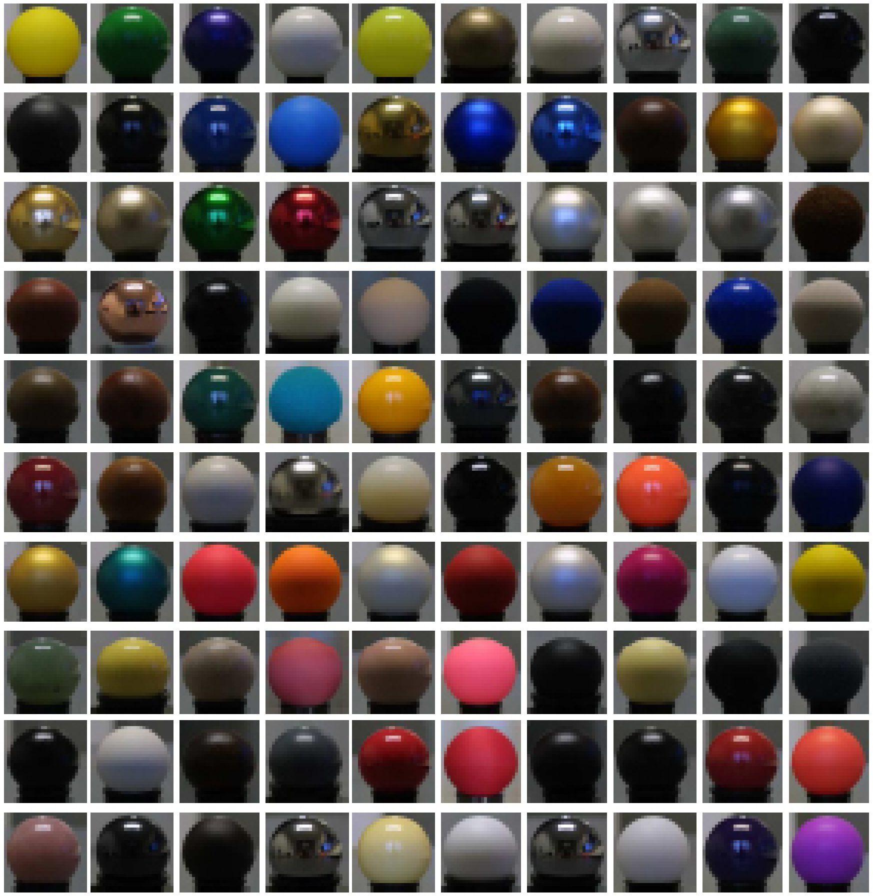
Murray-Coleman and Smith Gonioreflectometer. (Copied and Modified from [Ward92]).

Measured BRDF Samples

- Mitsubishi Electric Research Lab (MERL)

http://www.merl.com/brdf/

- Wojciech Matusik
- MIT PhD Thesis
- 100 Samples



BRDF Evolution

- BRDFs have evolved historically
- 1970's: Empirical models
  - Phong's illumination model
- 1980s:
  - Physically based models
  - Microfacet models (e.g. Cook Torrance model)
- 1990's
  - Physically-based appearance models of specific effects (materials, weathering, dust, etc)
- Early 2000's
  - Measurement & acquisition of static materials/lights (wood, translucence, etc)
- Late 2000's
  - Measurement & acquisition of time-varying BRDFs (ripening, etc)

Time-varying BRDF

- BRDF: How different materials reflect light
- Time varying?: how reflectance changes over time
-

References

- Interactive Computer Graphics (6th edition), Angel and Shreiner
- Computer Graphics using OpenGL (3rd edition), Hill and Kelley