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

\[ I = k_d I_d \textbf{l} \cdot \textbf{n} + k_s I_s (\textbf{n} \cdot \textbf{h})^\beta + k_a I_a \]
Shading?

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

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

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
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**

```glsl
flat out vec4 color;  //vertex shade
```

```glsl
do {...}

color = ambient + diffuse + specular;
color.a = 1.0;
```
Flat Shading Implementation

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

```glsl
flat in vec4 color;

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

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

\[ C_a = \text{lerp}(C_1, C_2) \]
\[ C_b = \text{lerp}(C_1, C_3) \]
\[ \text{Lerp}(C_a, C_b) \]

for all scanlines

* lerp: linear interpolation
Linear interpolation Example

- If $a = 60$, $b = 40$
- RGB color at $v_1 = (0.1, 0.4, 0.2)$
- RGB color at $v_2 = (0.15, 0.3, 0.5)$
- Red value of $v_1 = 0.1$, red value of $v_2 = 0.15$

\[
x = \frac{b}{(a+b)} * v_1 + \frac{a}{(a+b)} * v_2
\]

Red value of $x = \frac{40}{100} * 0.1 + \frac{60}{100} * 0.15 = 0.04 + 0.09 = 0.13$

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)
for(int y = ybott; y < ytop; 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 Implementation

- 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`
   \[ I = k_d I_d \cdot \mathbf{n} + k_s I_s (\mathbf{n} \cdot \mathbf{h})^\beta + k_a I_a \]

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

\[ \mathbf{n} = \frac{(\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4)}{|\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4|} \]
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 \textit{vertex normal and vectors}
  - Use pixel \textit{vertex normal and vectors} to calculate Phong shading at pixel \textit{(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

\[
I = k_d I_d \ l \cdot n + k_s I_s \ (n \cdot h)^\beta + k_a I_a
\]

**b. Phong Shading**
- Set Vectors \((l,n,v,h)\)

\[
I = k_d I_d \ l \cdot n + k_s I_s \ (n \cdot h)^\beta + k_a I_a
\]
// vertex shader
in vec4 vPosition;
in vec3 vNormal;

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

uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;
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;
}
// 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)
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;

    I = k_d I_d l \cdot n + k_s I_s (n \cdot h) \beta + k_a I_a

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;

I = k_d I_d l \cdot n + k_s I_s (n \cdot 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 $(l \cdot n)$ diffuse term (or $\cos \theta$) term

\[
I = k_d I_d \ l \cdot n + k_s I_s (n \cdot h)^\beta + k_a I_a
\]

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

<table>
<thead>
<tr>
<th>$l \cdot n$</th>
<th>Value used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between 0.75 and 1</td>
<td>0.75</td>
</tr>
<tr>
<td>Between 0.5 and 0.75</td>
<td>0.5</td>
</tr>
<tr>
<td>Between 0.25 and 0.5</td>
<td>0.25</td>
</tr>
<tr>
<td>Between 0.0 and 0.25</td>
<td>0.0</td>
</tr>
</tbody>
</table>
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 \((\cos^\alpha \phi)\),

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

- 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)
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