Computer Graphics (CS 543)
Lecture 7 (Part 2): Per-Vertex lighting, Shading and Per-Fragment lighting

Prof Emmanuel Agu

Computer Science Dept.
Worcester Polytechnic Institute (WPI)
Computation of Vectors

- To calculate lighting at vertex P
  - Need \( \mathbf{l}, \mathbf{n}, \mathbf{r} \) and \( \mathbf{v} \) vectors at vertex P

- User specifies:
  - Light position
  - Viewer (camera) position
  - Vertex (mesh position)

- \( \mathbf{l} \): Light position – vertex position
- \( \mathbf{v} \): Viewer position – vertex position
- \( \mathbf{n} \): Newell method
- Normalize all vectors!
Specifying a Point Light Source

- For each light source component, set RGBA
- \( \text{alpha} = \text{transparency} \)

\[
\begin{align*}
\text{vec4 diffuse0} &= \text{vec4}(1.0, 0.0, 0.0, 1.0) ; \\
\text{vec4 ambient0} &= \text{vec4}(1.0, 0.0, 0.0, 1.0) ; \\
\text{vec4 specular0} &= \text{vec4}(1.0, 0.0, 0.0, 1.0) ; \\
\text{vec4 light0_pos} &= \text{vec4}(1.0, 2.0, 3.0, 1.0) ;
\end{align*}
\]

- Set position is in homogeneous coordinates

\[
\text{vec4 light0_pos} = \text{vec4}(1.0, 2.0, 3.0, 1.0) ;
\]
Recall: Mirror Direction Vector $r$

- Can compute $r$ from $l$ and $n$
- $l$, $n$ and $r$ are co-planar

$$r = 2 (l \cdot n) \ n - l$$
Finding Normal, $n$

- Normal calculation in application. E.g. Newell method
- Passed to vertex shader
Material Properties

- Normal, material, shading functions now deprecated
- \((\text{glNormal}, \text{glMaterial}, \text{glLight})\) deprecated
- Specify material properties of scene object ambient, diffuse, specular (RGBA)
- \(w\) component gives opacity (transparency)
- **Default?** all surfaces are opaque

```cpp
vec4 ambient = vec4(0.2, 0.2, 0.2, 1.0);
vec4 diffuse = vec4(1.0, 0.8, 0.0, 1.0);
vec4 specular = vec4(1.0, 1.0, 1.0, 1.0);
GLfloat shine = 100.0
```

Material Shininess
(alpha in specular)
Recall: CTM Matrix passed into Shader

- **Recall**: CTM matrix concatenated in application
  
  ```
  mat4 ctm = ctm * LookAt(vec4 eye, vec4 at, vec4 up);
  ```

- CTM matrix passed in contains object transform + Camera

- Connected to matrix **ModelView** in shader

OpenGL Application Builds CTM

CTM

vertex Shader

```plaintext
in vec4 vPosition;
Uniform mat4 ModelView ;

main( )
{
    // Transform vertex position into eye coordinates
    vec3 pos = (ModelView * vPosition).xyz;
    ............
}
```
Per-Vertex Lighting: Declare Variables

Note: Phong lighting calculated at EACH VERTEX!!

// vertex shader
in vec4 vPosition;
in vec3 vNormal;
out vec4 color; //vertex shade

// light and material properties
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;

Ambient, diffuse, specular (light * reflectivity) specified by user

\[ k_a I_a \quad k_d I_d \quad k_s I_s \]

exponent of specular term
Per-Vertex Lighting: Compute Vectors

- CTM transforms vertex position into eye coordinates
  - Eye coordinates? Object, light distances measured from eye

```glsl
void main()
{
    // Transform vertex position into eye coordinates
    vec3 pos = (ModelView * vPosition).xyz;

    vec3 L = normalize(LightPosition.xyz - pos);  // light Vector
    vec3 E = normalize(-pos);                     // view Vector
    vec3 H = normalize(L + E);                    // halfway Vector

    // Transform vertex normal into eye coordinates
    vec3 N = normalize(ModelView*vec4(vNormal, 0.0)).xyz;
}
```

GLSL normalize function
// Compute terms in the illumination equation
vec4 ambient = AmbientProduct;  // \( k_a I_a \)

float cos_theta = max( dot(L, N), 0.0 );
vec4  diffuse = cos_theta * DiffuseProduct;  // \( k_d I_d l \cdot n \)

float cos_phi = pow( max(dot(N, H), 0.0), Shininess );
vec4  specular = cos_phi * SpecularProduct;  // \( k_s I_s (n \cdot h) ^ \beta \)

if( dot(L, N) < 0.0 )  specular = vec4(0.0, 0.0, 0.0, 1.0);

gl_Position = Projection * ModelView * vPosition;

color = ambient + diffuse + specular;
color.a = 1.0;

I = \( k_a I_a \) + \( k_d I_d l \cdot n \) + \( k_s I_s (n \cdot h) ^ \beta \)
Per-Vertex Lighting Shaders IV

// in vertex shader, we declared color as out, set it

       .......
       color = ambient + diffuse + specular;
       color.a = 1.0;

}

// in fragment shader (in vec4 color;

void main()
{
    gl_FragColor = color;
}

Graphics Hardware

color set in vertex shader

color used in fragment shader
Spotlights

- Derive from point source
  - **Direction I** (of lobe center)
  - **Cutoff**: No light outside $\theta$
  - **Attenuation**: Proportional to $\cos^\alpha \phi$

See section 5.2.4, pg 264 of Angel textbook
Shading
Shading?

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

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

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

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

Lighting calculation at vertices (in vertex shader)
Flat Shading

- compute lighting once for each face, assign color to whole face
- Benefit: Fast!!
Flat shading

● 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 amplify discontinuity at the boundary

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
Gouraud Shading

- Lighting calculated for each polygon vertex
- Colors are **interpolated** for interior pixels
- Interpolation? Assume linear change across face
- 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

......

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

\[
\begin{align*}
C_a &= \text{lerp}(C_1, C_2) \\
C_b &= \text{lerp}(C_1, C_3) \\
\text{Lerp}(C_a, C_b)
\end{align*}
\]

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)
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 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`

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

  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 fixes this, look smooth
**Phong Shading**

- Phong shading computes lighting in fragment shader.

- Need vectors \( n, l, v, r \) for each pixel – not provided by user.

- Instead of interpolating vertex color:
  - Interpolate *vertex normal and vectors*.
  - Use pixel *vertex normal and vectors* to calculate Phong lighting at pixel (*per pixel lighting*).
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:**
  - Set up vectors \((l,n,v,h)\) in vertex shader
  - Move lighting calculation to fragment shaders

**a. Gouraud Shading**
- Set Vectors \((l,n,v,h)\)
- Calculate vertex colors

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

**Equations**

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

in vec4 vPosition;
in vec3 vNormal;

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

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

Declare variables n, v, l as out in vertex shader
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  \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 \((l \cdot n)\) 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>
References