CS 543: Computer Graphics

Shading II

Robert W. Lindeman
Associate Professor
Interactive Media & Game Development
Department of Computer Science
Worcester Polytechnic Institute
gogo@wpi.edu

(with lots of help from Prof. Emmanuel Agu :-)

WPI
Recall: Setting Light Properties

- Define colors and position a light

```plaintext
var light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
var light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };;
var light_specular[] = { 1.0, 1.0, 1.0, 1.0 };;
var light_position[] = { 0.0, 0.0, 1.0, 1.0 };;
```

Colors

Position
Recall: Setting Material Properties

- Define ambient/diffuse/specular reflection and shininess

```c
Var mat_amb_diff[] = { 1.0, 0.5, 0.8, 1.0 };  // Ambient
Var mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };  // Specular
Var shininess[] = { 5.0 };  // Shininess, range: dull 0 – very shiny 128
```

Pass values to shader...see book code
Recall: Calculating Color at Vertices

- Illumination from a light
  \[
  \text{Illum} = \text{ambient} + \text{diffuse} + \text{specular}
  \]
  \[
  = K_a \times I + K_d \times I \times \cos(\theta) + K_s \times I \times \cos^f(\phi)
  \]

- If there are N lights
  \[
  \text{Total illumination for a point } P = \sum (\text{Illum})
  \]

- Sometimes lights or surfaces are colored

- Treat R, G, and B components separately
  - i.e., can specify different RGB values for either light or material

\[
\begin{align*}
\text{Illum}_r &= K_{ar} \times I_r + K_{dr} \times I_r \times \cos(\theta) + K_{sr} \times I_r \times \cos^f(\phi) \\
\text{Illum}_g &= K_{ag} \times I_g + K_{dg} \times I_g \times \cos(\theta) + K_{sg} \times I_g \times \cos^f(\phi) \\
\text{Illum}_b &= K_{ab} \times I_b + K_{db} \times I_b \times \cos(\theta) + K_{sb} \times I_b \times \cos^f(\phi)
\end{align*}
\]
Recall: Calculating Color at Vertices (cont.)

Illum = ambient + diffuse + specular

= $K_a \times I + K_d \times I \times \cos(\theta) + K_s \times I \times \cos^f(\phi)$

- $\cos(\theta)$ and $\cos^f(\phi)$ are calculated as dot products of Light vector $L$, Normal $N$, and Mirror-direction vector $R$

- To give

Illum = $K_a \times I + K_d \times I \times (N \cdot L) + K_s \times I \times (R \cdot V)$
Importance of Surface Normals

- Correct normals are essential for correct lighting
- Associate a normal with each vertex

```javascript
normalsArray.push( u, v, n );
pointsArray.push( x, y, z );
...
```

- All normals must be specified in unit length
  - Do in shader: `vec4 NN = vec4(vNormal, 0);`
Lighting Revisited

- Light calculation so far is at vertices
- Pixel may not fall right on vertex
- **Shading**
  - Calculates color of interior pixels
- Where are **lighting** & **shading** performed in the pipeline?
Example Shading Function

for( int y = \text{y}_{\text{bot}}; y \leq \text{y}_{\text{top}}; y++ ) {
    \text{find } \text{x}_{\text{left}} \text{ and } \text{x}_{\text{right}}
    for( int x = \text{x}_{\text{left}}; x < \text{x}_{\text{right}}; x++ ) {
        \text{find the color } \text{c for this pixel}
        \text{put } \text{c} \text{ into the pixel at } (x, y)
    }
}

- Scans pixels, row by row, calculating color for each pixel
Polygon Shading Models

- Flat shading
  - Compute lighting once and assign the color to the whole polygon (or mesh)
Flat Shading

- Only use one vertex normal and material property to compute the color for the polygon
- Benefit: fast to compute
- Used when
  - Polygon is small enough
  - Light source is far away (why?)
  - Eye is very far away (why?)
Mach-Band Effect

- Flat shading suffers from "mach banding"
  - Human eyes accentuate discontinuities at boundaries

Perceived intensity

Side view of a polygonal surface
Smooth Shading

- Fix the mach banding
  - Remove edge discontinuities
- Compute lighting for more points on each face

Flat shading  ➔ Smooth shading
Smooth Shading (cont.)

- Two popular methods
  - Gouraud shading (used by OpenGL)
  - Phong shading (better specular highlight, not in OpenGL)
Gouraud Shading

- Lighting is calculated for each of the polygon vertices
- Colors are interpolated for interior pixels
Gouraud Shading (cont.)

- Per-vertex lighting calculation
- Normal is needed for each vertex
- Per-vertex normal can be computed by averaging the adjacent face normals

\[
n = \frac{(n_1 + n_2 + n_3 + n_4)}{4.0}
\]
Gouraud Shading (cont.)

- Compute vertex illumination (color) before the projection transformation
- Shade interior pixels: color interpolation (normals are not needed for interior)

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

for all scanlines

* lerp: linear interpolation
Gouraud Shading (cont.)

- Linear interpolation

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

- Interpolate triangle color: use y distance to interpolate the two end points in the scanline, and use x distance to interpolate interior pixel colors.
Gouraud Shading Function

// for each scan line
for( int y = y_bot; y <= y_top; y++ )  {
    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 Problem

- Lighting in the polygon interior can be inaccurate
Phong Shading

- Instead of interpolation, we calculate lighting for each pixel inside the polygon (per-pixel lighting)

- Need normals for all the pixels
  - Not provided by user!

- Phong shading algorithm
  - Interpolate the normals across polygon
  - Compute lighting during rasterization
    - Need to map the normal back to world or eye space though
Phong Shading (cont.)

- Normal interpolation

- Slow
  - Not supported by OpenGL, but now graphics cards have pixel shaders that can be used to do this quickly

\[ n_a = \text{lerp}( n_1, n_2 ) \]
\[ n_b = \text{lerp}( n_1, n_3 ) \]
\[ \text{lerp}( n_a, n_b ) \]