Shading in OpenGL

Prof. George Wolberg
Dept. of Computer Science
City College of New York
Objectives

• Introduce the OpenGL shading methods
  - per vertex vs per fragment shading
  - Where to carry out

• Discuss polygonal shading
  - Flat
  - Smooth
    • Gouraud shading
    • Phong shading
Shading Principles

• Shading simulates how objects reflect light
  - material composition of object
  - light’s color and position
  - global lighting parameters

• Usually implemented in
  - vertex shader for faster speed
    • Gouraud shading
  - fragment shader for nicer shading
    • Phong shading
OpenGL shading

• Need to specify:
  - Normals
  - Material properties
  - Lights
  - Get computed values in application or send attributes to shaders
Surface Normals

• Normals define how a surface reflects light
  - Application usually provides normals as a vertex attribute
  - Current normal is used to compute vertex’s color
  - Use unit normals for proper lighting
    • scaling affects a normal’s length
Normal for Triangle

plane \quad \mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0

\mathbf{n} = (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)

normalize \mathbf{n} \quad \leftarrow \quad \mathbf{n}/|\mathbf{n}|

Note that right-hand rule determines outward face
Specifying a Point Light Source

• For each light source, we can set an RGBA for the diffuse, specular, and ambient components, and XYZW for the position

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

• The position is given in homogeneous coordinates
  - If \( w = 1.0 \), we are specifying a finite location
  - If \( w = 0.0 \), we are specifying a parallel source with the given direction vector
Spotlights

- Derive from point source
  - Direction
  - Cutoff
  - Attenuation Proportional to $\cos^\alpha \phi$
Moving Light Sources

• Light sources are geometric objects whose positions or directions are affected by the model-view matrix.

• Depending on where we place the position (direction) setting function, we can
  - Move the light source(s) with the object(s)
  - Fix the object(s) and move the light source(s)
  - Fix the light source(s) and move the object(s)
  - Move the light source(s) and object(s) independently
Material Properties (1)

• Define the surface properties of a primitive

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diffuse</td>
<td>Base object color</td>
</tr>
<tr>
<td>Specular</td>
<td>Highlight color</td>
</tr>
<tr>
<td>Ambient</td>
<td>Low-light color</td>
</tr>
<tr>
<td>Emission</td>
<td>Glow color</td>
</tr>
<tr>
<td>Shininess</td>
<td>Surface smoothness</td>
</tr>
</tbody>
</table>

- you can have separate materials for front and back
Material Properties (2)

- Material properties should match the terms in the light model
- Specifies amount of reflected light
  - An object appears red because it reflects red component of light
- w component gives opacity

```cpp
vec4 ambient  = vec4(0.2, 0.2, 0.2, 1.0);
vec4 diffuse  = vec4(1.0, 1.0, 1.0, 1.0);
GLfloat shine = 100.0
```
Flat Shading

• Use triangle normal across all fragments of triangle
• One color per triangular facet
Smooth Shading

• Set a new normal at each vertex
• Easy for sphere model
  - If centered at origin $\mathbf{n} = \mathbf{p}$
• Note *silhouette edges*
Gouraud Shading

• Computes a color for each vertex using
  - Surface normals
  - Diffuse and specular reflections
  - Viewer’s position and viewing direction
  - Ambient light
  - Emission

• Vertex colors are interpolated across polygons by the rasterizer
  - *Phong shading* does the same computation per pixel, interpolating the normal across the polygon
    • more accurate results
Polygonal Shading

- In per vertex shading, shading calculations are done for each vertex
  - Vertex colors become vertex shades and can be sent to the vertex shader as a vertex attribute
  - Alternately, we can send the parameters to the vertex shader and have it compute the shade
- By default, vertex shades are interpolated across an object if passed to the fragment shader as a varying variable (smooth shading)
- We can also use uniform variables to shade with a single shade (flat shading)
Mesh Shading

• The previous example is not general because we knew the normal at each vertex analytically.
• For polygonal models, Gouraud proposed we use the average of the 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 and Phong Shading

- **Gouraud Shading**
  - Find average normal at each vertex (vertex normals)
  - Apply Phong illumination equation at each vertex
  - Interpolate vertex shades across each polygon

- **Phong shading**
  - Find vertex normals
  - Interpolate vertex normals across edges
  - Interpolate edge normals across polygon
  - Apply modified Phong model at each fragment
Comparison

• If the polygon mesh approximates surfaces with a high curvatures, Phong shading may look smooth while Gouraud shading may show edges.

• Phong shading requires much more work than Gouraud shading:
  - Until recently not available in real time systems
  - Now can be done using fragment shaders

• Both need data structures to represent meshes so we can obtain vertex normals.
Normalization

• Cosine terms in lighting calculations can be computed using dot product
• Unit length vectors simplify calculation
• Usually we want to set the magnitudes to have unit length but
  - Length can be affected by transformations
  - Note that scaling does not preserve length
• GLSL has a normalization function
Gouraud Shading: Vertex Shader(1)

// 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;
void main()
{
    // transform vertex position into eye coordinates
    vec3 pos = (ModelView * vPosition).xyz;
    vec3 L = normalize( LightPosition.xyz - pos );
    vec3 E = normalize( -pos );
    vec3 H = normalize( L + E );

    // transform vertex normal into eye coordinates
    vec3 N = normalize( ModelView*vec4(vNormal, 0.0) ).xyz;
}
Gouraud Shading: Vertex Shader (3)

// compute terms in the illumination equation
// ambient lighting term
vec4 ambient = AmbientProduct;

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

// specular lighting term
float Ks = pow( max(dot(N, H), 0.0), Shininess );
vec4 specular = Ks * SpecularProduct;
if( dot(L, N) < 0.0 ) specular = vec4(0.0, 0.0, 0.0, 1.0);

// add lighting terms to form color
color = ambient + diffuse + specular;
color.a = 1.0;

gl_Position = Projection * ModelView * vPosition;
Gouraud Shading: Fragment Shader

// Fragment Shader
in vec4 color;

void main()
{
    gl_FragColor = color;
}

Phong Shading: Vertex Shader (1)

// 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;
}
Phong Shading:
Fragment Shader (1)

// 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;

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;

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
// is 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;
Fragment Shaders

Prof. George Wolberg
Dept. of Computer Science
City College of New York
Fragment Shader

- A shader that’s executed for each “potential” pixel
  - fragments still need to pass several tests before making it to the framebuffer

- There are lots of effects we can do in fragment shaders
  - Per-fragment lighting
  - Texture and bump Mapping
  - Environment (Reflection) Maps
Shader Examples

• Vertex Shaders
  - Moving vertices: height fields
  - Per vertex lighting: height fields
  - Per vertex lighting: cartoon shading

• Fragment Shaders
  - Per vertex vs. per fragment lighting: cartoon shader
  - Samplers: reflection Map
  - Bump mapping
A height field is a function $y = f(x, z)$
- $y$ represents height of point for a location in the $x$-$z$ plane
Heights fields are usually rendered as a rectangular mesh of triangles or rectangles sampled from a grid
- samples $y_{ij} = f(x_i, z_j)$
Displaying a Height Field

• First, generate a mesh data and use it to initialize data for a VBO

```cpp
float dx = 1.0/N, dz = 1.0/N;
for( int i = 0; i < N; i++ ) {
    float x = i*dx;
    for( int j = 0; j < N; j++ ) {
        float z = j*dz;
        float y = f( x, z );
        vertex[Index++] = vec3( x, y, z );
        vertex[Index++] = vec3( x, y, z + dz );
        vertex[Index++] = vec3( x + dx, y, z + dz );
        vertex[Index++] = vec3( x + dx, y, z );
    }
}
```

• Finally, display each quad using

```cpp
for( int i = 0; i < NumVertices; i += 4 )
    glDrawArrays( GL_LINE_LOOP, 4*i, 4 );
```
Time Varying Vertex Shader

```
in vec4 vPosition;
in vec4 vColor;

uniform float time; // in milliseconds
uniform mat4 ModelViewProjectionMatrix;

void main()
{
    vec4 v = vPosition;
    vec4 u = sin( time + 5*v );
    v.y = 0.1 * u.x * u.z;

    gl_Position = ModelViewProjectionMatrix * v;
}
```
Mesh Display
Adding Lighting

• Solid Mesh: create two triangles for each quad
• Display with
  
  ```glDrawArrays(GL_TRIANGLES, 0, NumVertices);```

• For better looking results, add lighting
• We’ll do per-vertex lighting
  - leverage the vertex shader since we’ll also use it to vary the mesh in a time-varying way
uniform float time, shininess;
uniform vec4 vPosition, lightPosition, diffuseLight, specularLight;
uniform mat4 ModelViewMatrix, ModelViewProjectionMatrix, NormalMatrix;

void main()
{
  vec4 v = vPosition;
  vec4 u = sin( time + 5*v );
  v.y = 0.1 * u.x * u.z;

  gl_Position = ModelViewProjectionMatrix * v;

  vec4 diffuse, specular;
  vec4 eyePosition = ModelViewMatrix * vPosition;
  vec4 eyeLightPos = lightPosition;
Mesh Shader (2)

```cpp
vec3 N = normalize(NormalMatrix * Normal);
vec3 L = normalize(vec3(eyeLightPos - eyePosition));
vec3 E = -normalize(eyePosition.xyz);
vec3 H = normalize(L + E);

float Kd = max(dot(L, N), 0.0);
float Ks = pow(max(dot(N, H), 0.0), shininess);
diffuse = Kd*diffuseLight;
specular = Ks*specularLight;
color = diffuse + specular;
```
Shaded Mesh