Shading so Far

• So far, we have discussed illuminating a single point

\[ I = k_a I_a + I_i \left( k_d (L \cdot N) + k_s (H \cdot N)^p \right) \]

• We have assumed that we know:
  – The point
  – The surface normal
  – The viewer location (or direction)
  – The light location (or direction)

• But commonly, normal vectors are only given at the vertices
• It is also expensive to compute lighting for every point
Shading Interpolation

• Several options:
  – Flat shading
  – Gouraud interpolation
  – Phong interpolation

• New hardware provides other options
Flat shading

- Compute shading at a representative point and apply to whole polygon
  - OpenGL uses one of the vertices
- Advantages:
  - Fast - one shading computation per polygon, fill entire polygon with same color
- Disadvantages:
  - Inaccurate
  - What are the artifacts?
Gouraud Shading

- Shade each *vertex* with its own location and normal
- *Linearly interpolate* the color across the face
- **Advantages:**
  - Fast - incremental calculations when rasterizing
  - Much smoother - use one normal per shared vertex to get continuity between faces
- **Disadvantages:**
  - What are the artifacts?
  - Is it accurate?
Phong Interpolation

• Interpolate normals across faces
• Shade each pixel
• Advantages:
  – High quality, narrow specularities
• Disadvantages:
  – Expensive
  – Still an approximation for most surfaces
• Not to be confused with Phong’s specularity model
Shading and OpenGL

- OpenGL defines two particular shading models
  - Controls how colors are assigned to pixels
  - `glShadeModel(GL_SMOOTH)` interpolates between the colors at the vertices (the default, Gouraud shading)
  - `glShadeModel(GL_FLAT)` uses a constant color across the polygon
The Current Generation

• Current hardware allows you to break from the standard illumination model

• *Programmable Vertex Shaders* allow you to write a small program that determines how the color of a vertex is computed
  – Your program has access to the surface normal and position, plus anything else you care to give it (like the light)
  – You can add, subtract, take dot products, and so on
The Full Story

• We have only touched on the complexities of illuminating surfaces
  – The common model is hopelessly inadequate for accurate lighting (but it’s fast and simple)

• Consider two sub-problems of illumination
  – Where does the light go? *Light transport*
  – What happens at surfaces? *Reflectance models*

• Other algorithms address the transport or the reflectance problem, or both
  – Much later in class, or a separate course
Light Sources

• Two aspects of light sources are important for a local shading model:
  – Where is the light coming from (the \( L \) vector)?
  – How much light is coming (the \( I \) values)?

• Various light source types give different answers to the above questions:
  – *Point light source*: Light from a specific point
  – *Directional*: Light from a specific direction
  – *Spotlight*: Light from a specific point with intensity that depends on the direction
  – *Area light*: Light from a continuum of points (later in the course)
Point and Directional Sources

- **Point light**: $L(x) = ||p_{light} - x||$
  - The $L$ vector depends on where the surface point is located
  - Must be normalized - slightly expensive
  - To specify an OpenGL light at 1,1,1:
    ```
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 1.0 };  
gllightfv(GL_LIGHT0, GL_POSITION, light_position);
    ```
- **Directional light**: $L(x) = L_{light}$
  - The $L$ vector does not change over points in the world
  - OpenGL light traveling in direction 1,1,1 ($L$ is in opposite direction):
    ```
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };  
gllightfv(GL_LIGHT0, GL_POSITION, light_position);
    ```
Spotlights

• Point source, but intensity depends on $L$:
  – Requires a position: the location of the source
    \[
    \text{glLightfv(GL\_LIGHT0, GL\_POSITION, light\_posn);}
    \]
  – Requires a direction: the center axis of the light
    \[
    \text{glLightfv(GL\_LIGHT0, GL\_SPOT\_DIRECTION, light\_dir);}
    \]
  – Requires a cut-off: how broad the beam is
    \[
    \text{glLightfv(GL\_LIGHT0, GL\_SPOT\_CUTOFF, 45.0);}
    \]
  – Requires and exponent: how the light tapers off at the edges of the cone
    • Intensity scaled by $(L \cdot D)^n$
      \[
      \text{glLightfv(GL\_LIGHT0, GL\_SPOT\_EXPONENT, 1.0);}
      \]