Non-Ideal Specular Reflectance
• Snell’s law applies to perfect mirror-like surfaces, but aside from mirrors (and chrome) few surfaces exhibit perfect specularly
• how can we capture the “softer” reflections of surface that are glossy, not mirror-like?
• one option: model the microgeometry of the surface and explicitly bounce rays off of it
  • or…

Optics of Reflection
• reflection follows Snell’s Law:
  • incoming ray and reflected ray lie in a plane with the surface normal
  • angle the reflected ray forms with surface normal equals angle formed by incoming ray and surface normal

Empirical Approximation
• angular falloff
  • how might we model this falloff?

Phong Lighting
• most common lighting model in computer graphics
  • (Phong Bui-Tuong, 1975)
  • simplest: purely empirical constant, varies rate of falloff
  • $k_s$: specular coefficient, highlight color
  • no physical basis, works ok in practice

Empirical Approximation
• angular falloff
  • $I_s = k_s I_l \cos(\theta)$

Physics of Specular Reflection
• at the microscopic level a specular reflecting surface is very smooth
• thus rays of light are likely to bounce off the microgeometry in a mirror-like fashion
• the smoother the surface, the closer it becomes to a perfect mirror

Optics of Reflection
• reflection follows Snell’s Law:
  • incoming ray and reflected ray lie in a plane with the surface normal
  • angle the reflected ray forms with surface normal equals angle formed by incoming ray and surface normal

Empirical Approximation
• angular falloff
  • $I_s = k_s I_l \cos(\theta)$

Physics of Specular Reflection
• at the microscopic level a specular reflecting surface is very smooth
• thus rays of light are likely to bounce off the microgeometry in a mirror-like fashion
• the smoother the surface, the closer it becomes to a perfect mirror

Optics of Reflection
• reflection follows Snell’s Law:
  • incoming ray and reflected ray lie in a plane with the surface normal
  • angle the reflected ray forms with surface normal equals angle formed by incoming ray and surface normal

Empirical Approximation
• angular falloff
  • $I_s = k_s I_l \cos(\theta)$
Phong Lighting: The n\textsubscript{shiny} Term
- Phong reflectance term drops off with divergence of viewing angle from ideal reflected ray

Viewing angle – reflected angle

Phong Examples
- varying I
- varying n\textsubscript{shiny}

Calculating Phong Lighting
- compute cosine term of Phong lighting with vectors
- \( I_{\text{specular}} = k \cdot I_{\text{light}} (v \cdot r)^n_{\text{shiny}} \)
- \( v \): unit vector towards viewer/eye
- \( r \): ideal reflectance direction (unit vector)
- \( k \): specular component
- \( n \): shininess
- \( I_{\text{light}} \): incoming light intensity

- how to efficiently calculate \( r \)?

Calculating R Vector
- \( P = N \cos \theta \cdot |L| \cdot |N| \) projection of \( L \) onto \( N \)
- \( P = N (N \cdot L) \)

Calculating R Vector
- \( P = N \cos \theta \cdot |L| \cdot |N| \) projection of \( L \) onto \( N \)
- \( P = N (N \cdot L) \)

Calculating R Vector
- \( P = N \cos \theta \cdot |L| \cdot |N| \) projection of \( L \) onto \( N \)
- \( P = N (N \cdot L) \)

Calculating R Vector
- \( P = N \cos \theta = \text{projection of } L \text{ onto } N \)

Phong Lighting Model
- \( \frac{k_s}{h \cdot n \cdot n} \) with \( h = (l + v)/2 \)
- \( h \): halfway vector
- \( n \): shininess
- \( k_s \): specular component

Blinn-Phong Model
- \( \frac{k_s}{h \cdot n \cdot n} \) with \( h = (l + v)/2 \)
- \( k_s \): specular component
- \( n \): shininess
- \( l \): incoming light intensity
- \( h \): halfway vector
- \( n \): shininess

Light Source Falloff
- quadratic falloff
- brightness of objects depends on power per unit area that hits the object
- the power per unit area for a point or spot light decreases quadratically with distance

Lighting in OpenGL
- light source: amount of RGB light emitted
- value represents percentage of full intensity e.g., (1.0, 0.5, 0.0)
- every light source emits ambient, diffuse, and specular light
- materials: amount of RGB light reflected
- value represents percentage reflected e.g., (0.0, 1.0, 0.0)
- interaction: multiply components
- red light (1.0, 0.0) x green surface (0.1, 0.0) = black (0.0, 0.0)

Shading
- warning: glMaterial is expensive and tricky
- use cheap and simple glColor when possible
- see OpenGL Pitfall #14 from Kilgard’s list

http://www.opengl.org/resources/features/KilgardTechniques/oglpitfall/
Lighting vs. Shading
- **lighting**
  - process of computing the luminous intensity (i.e., outgoing light) at a particular 3-D point, usually on a surface
- **shading**
  - the process of assigning colors to pixels
  - (why the distinction?)

Applying Illumination
- we now have an illumination model for a point on a surface
- if surface defined as mesh of polygonal facets, which points should we use?
  - fairly expensive calculation
  - several possible answers, each with different implications for visual quality of result

Applying Illumination
- polygonal/triangular models
  - each facet has a constant surface normal
  - if light is directional, diffuse reflectance is constant across the facet
  - why?

Flat Shading
- simplest approach calculates illumination at a single point for each polygon
  - obviously inaccurate for smooth surfaces

Gouraud Shading Artifacts
- perspective transformations
  - affine combinations only invariant under affine, not under perspective transformations
  - thus, perspective projection alters the linear interpolation!

Improving Flat Shading
- vertex normals may be
  - provided with the model
  - computed from first principles
  - approximated by averaging the normals of the facets that share the vertex

Improving Flat Shading
- what if evaluate Phong lighting model at each pixel of the polygon?
  - better, but result still clearly faceted
  - for smoother-looking surfaces we introduce vertex normals at each vertex
  - usually different from facet normal
  - used only for shading
  - think of as a better approximation of the real surface that the polygons approximate

Phong Shading
- linearly interpolate the vertex normals
  - compute lighting equations at each pixel
  - can use specular component
  - compute lighting: empirical model to calculate illumination at a point on a surface
    - $I_{\text{final}} = k_1 I_{\text{ambient}} + \sum_{i=1}^{N_{\text{lights}}} I_i (\hat{k}_i \cdot (\hat{n} \cdot \hat{r}_i))^{\text{power}}$
    - remember: normals used in diffuse and specular terms

Phong Shading
- discontinuity in normal's rate of change harder to detect

Gouraud Shading Artifacts
- often appears dull, chalky
  - lacks accurate specular component
  - if included, will be averaged over entire polygon

Gouraud Shading Artifacts
- Mach bands
  - eye enhances discontinuity in first derivative
  - very disturbing, especially for highlights

Gouraud Shading Artifacts
- perspective transformation problem
  - colors slightly "swim" on the surface as objects move relative to the camera
  - usually ignored since often only small difference
  - usually smaller than changes from lighting variations
  - to do it right
    - either shading in object space
    - or correction for perspective foreshortening
    - expensive – thus hardly ever done for colors

Gouraud Shading Artifacts
- discontinuity in rate of color change occurs here

Gouraud Shading Artifacts
- edge: mix of $c_1, c_2, c_3$

Gouraud Shading Artifacts
- image plane

Gouraud Shading Artifacts
- 2 – into the scene

Illuminating vs. Shading:
- lighting
  - process of computing the luminous intensity (i.e., outgoing light) at a particular 3-D point, usually on a surface

Shading:
- the process of assigning colors to pixels

Phong Shading Difficulties
- computationally expensive
  - per-pixel vector normalization and lighting computation!
  - floating point operations required
  - lighting after perspective projection
    - messes up the angles between vectors
    - have to keep eye-space vectors around
  - no direct support in pipeline hardware
    - but can be simulated with texture mapping
  - stay tuned for modern hardware: shaders
Shading Artifacts: Silhouettes
- polygonal silhouettes remain

Shading Artifacts: Orientation
- interpolation dependent on polygon orientation
- view dependence!

Shading Artifacts: Shared Vertices
- vertex B shared by two rectangles on the right, but not by the one on the left

Shading Models Summary
- flat shading
  - compute Phong lighting once for entire polygon
- Gouraud shading
  - compute Phong lighting at the vertices and interpolate lighting values across polygon
- Phong shading
  - compute averaged vertex normals
  - interpolate normals across polygon and perform Phong lighting across polygon

Non-Photorealistic Shading
- cool-to-warm shading
  \[ k_w = \frac{l + n}{2}, \quad k_w = k_w + (1 - k_w) c \]

Specifying Normals
- OpenGL state machine
  - uses last normal specified
  - if no normals specified, assumes all identical
- per-vertex normals
  - \( \text{glnormal} \)\( n_x, n_y, n_z \)
  - \( \text{glnormal}N_2, N_3 \)
  - \( \text{glnormal}F_1, F_2, F_3 \)
- per-face normals
  - \( \text{glnormal}F_1, F_2, F_3 \)
  - \( \text{glnormal}F_2, F_3, F_4 \)

Computing Normals
- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
  - computing normal for a polygon

Computing Normals
- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
  - computing normal for a polygon
  - three points form two vectors

Non-Photorealistic Shading
- draw silhouettes: if \( e^T a \cdot e \neq 0 \), \( e \) = edge-eye vector
- draw creases: if \( |a - b| \) < threshold

Specifying Normals
- OpenGL state machine
  - uses last normal specified
  - if no normals specified, assumes all identical
- per-vertex normals
  - \( \text{glnormal} n_x, n_y, n_z \)
  - \( \text{glnormal}N_2, N_3 \)
  - \( \text{glnormal}F_1, F_2, F_3 \)
- per-face normals
  - \( \text{glnormal}F_1, F_2, F_3 \)
  - \( \text{glnormal}F_2, F_3, F_4 \)

Computing Normals
- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
  - computing normal for a polygon
  - three points form two vectors

Computing Normals
- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
  - computing normal for a polygon
  - three points form two vectors
  - \( k_w = \frac{l + n}{2} \), \( k_w = k_w + (1 - k_w) c \)

Computing Normals
- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
  - computing normal for a polygon
  - three points form two vectors
  - cross: normal of plane gives direction
  - normalize to unit length!
  - which side is up?
    - convention: points in counterclockwise order

Computing Normals
- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
  - computing normal for a polygon
  - three points form two vectors
  - cross: normal of plane gives direction
  - normalize to unit length!
  - which side is up?
    - convention: points in counterclockwise order

Computing Normals
- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
  - computing normal for a polygon
  - three points form two vectors
  - cross: normal of plane gives direction
  - normalize to unit length!
  - which side is up?
    - convention: points in counterclockwise order

Computing Normals
- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
  - computing normal for a polygon
  - three points form two vectors
  - cross: normal of plane gives direction
  - normalize to unit length!
  - which side is up?
    - convention: points in counterclockwise order

Shutterbug: Flat Shading
- draw silhouettes: if \( e^T a \cdot e \neq 0 \), \( e \) = edge-eye vector
- draw creases: if \( |a - b| \) < threshold

Shutterbug: Gouraud Shading
- draw silhouettes: if \( e^T a \cdot e \neq 0 \), \( e \) = edge-eye vector
- draw creases: if \( |a - b| \) < threshold

Shutterbug: Phong Shading
- draw silhouettes: if \( e^T a \cdot e \neq 0 \), \( e \) = edge-eye vector
- draw creases: if \( |a - b| \) < threshold

Non-Photorealistic Shading
- cool-to-warm shading
  \[ k_w = \frac{l + n}{2}, \quad k_w = k_w + (1 - k_w) c \]

Specifying Normals
- OpenGL state machine
  - uses last normal specified
  - if no normals specified, assumes all identical
- per-vertex normals
  - \( \text{glnormal} n_x, n_y, n_z \)
  - \( \text{glnormal}N_2, N_3 \)
  - \( \text{glnormal}F_1, F_2, F_3 \)
- per-face normals
  - \( \text{glnormal}F_1, F_2, F_3 \)
  - \( \text{glnormal}F_2, F_3, F_4 \)