University of British Columbia
CPSC 314 Computer Graphics
Jan-Apr 2010

Tamara Munzner

Lighting/Shading III

Week 7, Wed Mar 3

http://www.ugrad.cs.ubc.ca/~cs314/Vjan2010

News

- reminders
  - don't need to tell us you're taking grace days, they're assumed if you turn in late
  - separate for written homework and project
  - exception: HW2 not accepted after 11am Fri
    - solutions posted then so you can use them when studying for midterm

Midterm

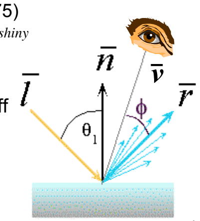
- Monday 3/8, 1-1:50
- topics
  - all material *through* Rasterization (Wed Feb 10 lecture)
- format
  - closed book
  - you may have simple (nongraphing) calculators
  - you may have notes on one side of 8.5"x11" sheet of paper
    - must be handwritten by you, cannot be xeroxed/printed
    - you'll keep these notes. for final, can use back side of page as well.
- logistics
  - must have UBC ID face up
  - backpacks/coats at front of room
  - phones off

Review: Phong Lighting

- most common lighting model in computer graphics
  - (Phong Bui-Tuong, 1975)

$$\mathbf{I}_{\mathbf{specular}} = \mathbf{k_s I_{light}} (\cos\phi)^{n_{shiny}}$$

- n_{shiny} : purely empirical constant, varies rate of falloff
- k_s: specular coefficient, highlight color
- no physical basis, works ok in practice

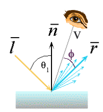


Calculating Phong Lighting

- compute **cosine** term of Phong lighting with vectors

$$\mathbf{I}_{\mathbf{specular}} = \mathbf{k_s I_{light}} (\mathbf{v} \bullet \mathbf{r})^{n_{shiny}}$$

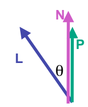
- v: unit vector towards viewer/eye
- r: ideal reflectance direction (unit vector)
- k_s: specular component
  - highlight color
- I_{light}: incoming light intensity



- how to efficiently calculate **r** ?

Calculating R Vector

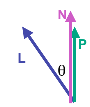
P = N cos θ = projection of **L** onto **N**



Calculating R Vector

P = N cos θ = projection of **L** onto **N**

P = N (N · L)

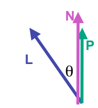


Calculating R Vector

P = N cos θ |**L**| |**N**| projection of **L** onto **N**

P = N cos θ **L, N** are unit length

P = N (N · L)



Calculating R Vector

P = N cos θ |**L**| |**N**| projection of **L** onto **N**

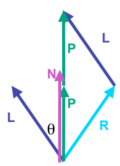
P = N cos θ **L, N** are unit length

P = N (N · L)

2 P = R + L

2 P – L = R

2 (N (N · L)) - L = R



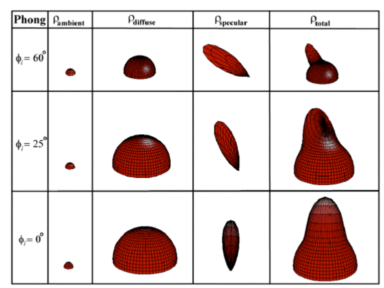
Phong Lighting Model

- combine ambient, diffuse, specular components

$$\mathbf{I}_{\mathbf{total}} = \mathbf{k_a I_{ambient}} + \sum_{i=1}^{\#\,lights} \mathbf{I_i} (\mathbf{k_d} (\mathbf{n} \bullet \mathbf{l_i}) + \mathbf{k_s} (\mathbf{v} \bullet \mathbf{r_i})^{n_{shiny}})$$

- commonly called *Phong lighting*
  - once per light
  - once per color component
- reminder: normalize your vectors when calculating!
  - normalize all vectors: n,l,r,v

Phong Lighting: Intensity Plots

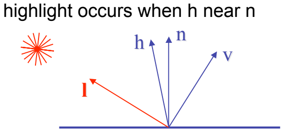


Blinn-Phong Model

- variation with better physical interpretation
  - Jim Blinn, 1977

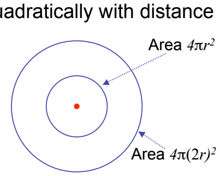
$$I_{out}(\mathbf{x}) = \mathbf{k_s} (\mathbf{h} \bullet \mathbf{n})^{n_{shiny}} \bullet I_{in}(\mathbf{x}); \text{with } \mathbf{h} = (\mathbf{l} + \mathbf{v})/2$$

- ***h***: halfway vector
  - h must also be explicitly normalized: h / |h|
  - highlight occurs when h near n



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



Light Source Falloff

- non-quadratic falloff
- many systems allow for other falloffs
- allows for faking effect of area light sources
- OpenGL / graphics hardware
  - I_o: intensity of light source
  - $\boldsymbol{x}$: object point
  - r: distance of light from x

$$I_{in}(\mathbf{x}) = \frac{1}{ar^2 + br + c} \cdot I_0$$

Lighting Review

- lighting models
- ambient
  - normals don't matter
- Lambert/diffuse
  - angle between surface normal and light
- Phong/specular
  - surface normal, light, and viewpoint

Lighting in OpenGL

- light source: amount of RGB light emitted
  - value represents percentage of full intensity e.g., (1.0,0.5,0.5)
  - every light source emits ambient, diffuse, and specular light
- materials: amount of RGB light reflected
  - value represents percentage reflected e.g., (0.0,1.0,0.5)
- interaction: multiply components
  - red light (1,0,0) x green surface (0,1,0) = black (0,0,0)

Lighting in OpenGL

```
glLightfv(GL_LIGHT0, GL_AMBIENT, amb_light_rgba );
glLightfv(GL_LIGHT0, GL_DIFFUSE, dif_light_rgba );
glLightfv(GL_LIGHT0, GL_SPECULAR, spec_light_rgba );
glLightfv(GL_LIGHT0, GL_POSITION, position);
glEnable(GL_LIGHT0);

glMaterialfv( GL_FRONT, GL_AMBIENT, ambient_rgba );
glMaterialfv( GL_FRONT, GL_DIFFUSE, diffuse_rgba );
glMaterialfv( GL_FRONT, GL_SPECULAR, specular_rgba );
glMaterialfv( GL_FRONT, GL_SHININESS, n );
```

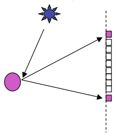
- warning: glMaterial is expensive and tricky
  - use cheap and simple glColor when possible
  - see OpenGL Pitfall #14 from Kilgard's list

http://www.opengl.org/resources/features/KilgardTechniques/oglpitfall/

Shading

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

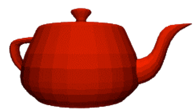
- we now have an illumination model for a point on a surface
- if surface defined as mesh of polygonal facets, *which points should we use?*
  - fairly expensive calculation
  - several possible answers, each with different implications for visual quality of result

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

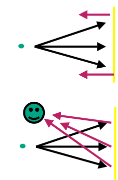
- simplest approach calculates illumination at a single point for each polygon



- obviously inaccurate for smooth surfaces

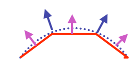
Flat Shading Approximations

- if an object really <u>is</u> faceted, is this accurate?
- no!
  - for point sources, the direction to light varies across the facet
  - for specular reflectance, direction to eye varies across the facet



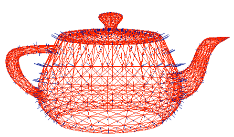
Improving Flat Shading

- 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



Vertex Normals

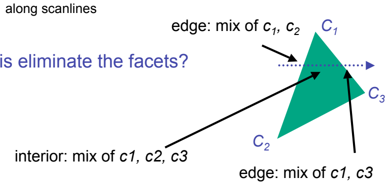
- vertex normals may be
  - provided with the model
  - computed from first principles
  - approximated by averaging the normals of the facets that share the vertex



Gouraud Shading

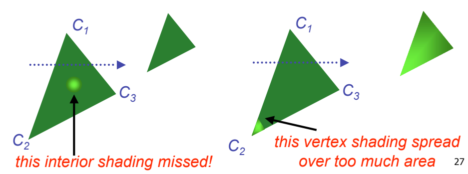
- most common approach, and what OpenGL does
  - perform Phong lighting at the vertices
  - linearly interpolate the resulting colors over faces
    - along edges
    - along scanlines

does this eliminate the facets?



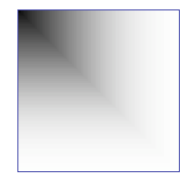
Gouraud Shading Artifacts

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



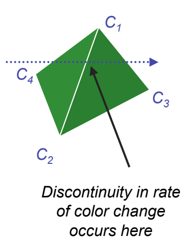
Gouraud Shading Artifacts

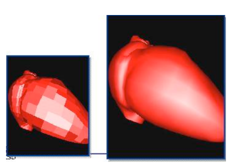
- Mach bands
  - eye enhances discontinuity in first derivative
  - very disturbing, especially for highlights



Gouraud Shading Artifacts

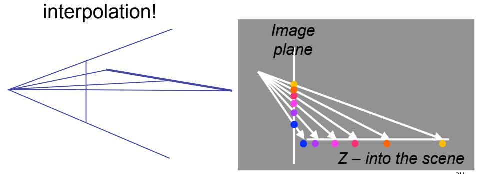
- Mach bands





Gouraud Shading Artifacts

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



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

Phong Shading

- linearly interpolating surface normal across the facet, applying Phong lighting model at every pixel
  - same input as Gouraud shading
  - pro: much smoother results
  - con: considerably more expensive

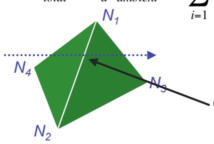
- **not** the same as Phong lighting
  - common confusion
  - Phong lighting: empirical model to calculate illum a point on a surface

Phong Shading

- linearly interpolate the vertex normals
  - compute lighting equations at each pixel
  - can use specular component

$$I_{total} = k_a I_{ambient} + \sum_{i=1}^{\#\,lights} I_i \left(k_d (\mathbf{n} \cdot \mathbf{l_i}) + k_s (\mathbf{v} \cdot \mathbf{r_i})^{n_{shiny}} \right)$$

remember: normals used in diffuse and specular terms



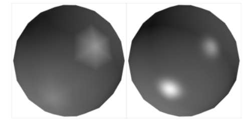
discontinuity in normal's rate of change harder to detect

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

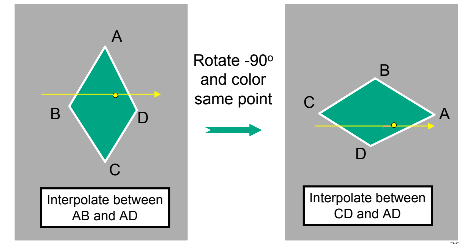
- polygonal silhouettes remain



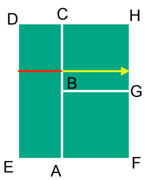
Gouraud *Phong*

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

first portion of the scanline is interpolated between DE and AC

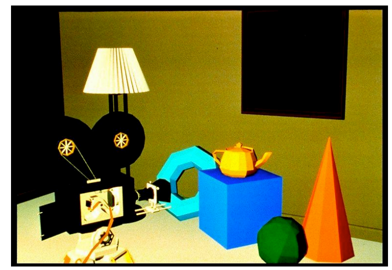
second portion of the scanline is interpolated between BC and GH

a large discontinuity could arise

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

Shutterbug: Flat Shading

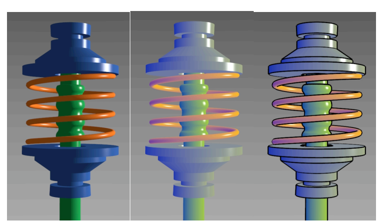


Shutterbug: Gouraud Shading

Shutterbug: Phong Shading

Non-Photorealistic Shading

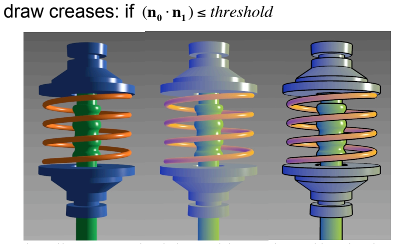
- cool-to-warm shading $k_w = \frac{1+\mathbf{n}\cdot\mathbf{l}}{2}, c = k_w c_w + (1-k_w) c_c$



http://www.cs.utah.edu/~gooch/SIG98/paper/drawing.html

Non-Photorealistic Shading

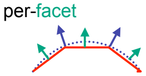
- draw silhouettes: if $(\mathbf{e}\cdot\mathbf{n_0})(\mathbf{e}\cdot\mathbf{n_1}) \le 0$, **e**=edge-eye vector
- draw creases: if $(\mathbf{n_0}\cdot\mathbf{n_1}) \le threshold$

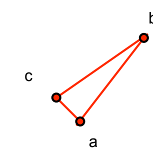


http://www.cs.utah.edu/~gooch/SIG98/paper/drawing.html

Computing Normals

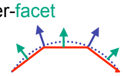
- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
- computing normal for a polygon

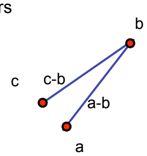




Computing Normals

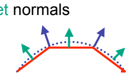
- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
- computing normal for a polygon
  - three points form two vectors

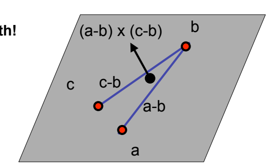




Computing Normals

- 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





Specifying Normals

- OpenGL state machine
  - uses last normal specified
  - if no normals specified, assumes all identical
- per-vertex normals

```
glNormal3f(1,1,1);
glVertex3f(3,4,5);
glNormal3f(1,1,0);
glVertex3f(10,5,2);
```

- per-face normals

```
glNormal3f(1,1,1);
glVertex3f(3,4,5);
glVertex3f(10,5,2);
```