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

Tamara Munzner

## Lighting/Shading II

## Week 7, Mon Mar 1

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

## News

- Homework 3 out today
- Homework 2, Project 2 due tomorrow
- TA office hours in lab
  - (Mon 2-3 lab, Shailen)
  - Tue 11-1, Shailen
  - (Tue 1-2 lab, Kai)
  - Tue 3:50-5, Kai
- (my office hours in X661 Mon 4-5)
  - intended for CS111, but will answer 314 questions if there are no 111 students waiting
- department news



Department of Computer Science Undergraduate Events

**Events this week**

**Resume & Cover Letter Drop-In Session**
Date: Wed., Mar 3
Time: 12 – 3 pm (20 mins. sessions)
Location: Rm 255, ICICS/CS

**Find a Job Fast! Info Session**
Date: Thurs., Mar 4
Time: 12:30 – 1:45 pm
Location: DMP 201
Registration: Email dianejoh@cs.ubc.ca

**Townhall Meting – 1st Year CS Students**
Date: Thurs., Mar 4
Time: 12:30 - 2 pm
Location: DMP 310
Lunch will be provided!

**Faculty Talk – Son Vuong**
Title: Mobile Learning via LIVES
Date: Thurs., Mar 4
Time: 12:30 – 1:45 pm
Location: DMP 201

**Events next week**

**Townhall Meeting – Combined Majors/Honours, BA, B.Comm in CS**
Date: Thurs., Mar 11
Time: 12:30 – 2 pm
Location: DMP 310
Lunch will be provided!

**CS Distinguished Lecture Series – Featuring David Parkes**
Title: Incentive Mechanism Engineering in the Internet Age
Date: Thurs., Mar 11
Time: 3:30 – 4:50 pm
Location: DMP 110

**CSSS Moive Night – “Zombieland” & “Iron Man”**
Date: Thurs., Mar 11
Time: 6 – 10 pm
Location: DMP 310
Free pop & popcorn!



## Review: Computing Barycentric Coordinates

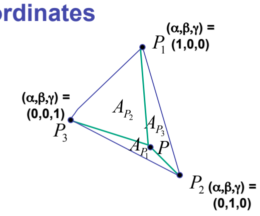
- 2D triangle area
  - half of parallelogram area
    - from cross product

$A = A_{P1} + A_{P2} + A_{P3}$

$\alpha = A_{P1}/A$

$\beta = A_{P2}/A$

$\gamma = A_{P3}/A$

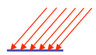



weighted combination of three points

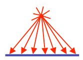


## Review: Light Sources

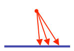
- directional/parallel lights
  - point at infinity: $(x,y,z,0)^T$



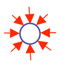
- point lights
  - finite position: $(x,y,z,1)^T$



- spotlights
  - position, direction, angle



- ambient lights





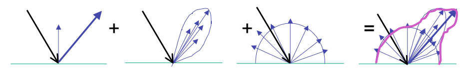
## Review: Light Source Placement

- geometry: positions and directions
- standard: world coordinate system
  - effect: lights fixed wrt world geometry
- alternative: camera coordinate system
  - effect: lights attached to camera (car headlights)



## Review: Reflectance

- *specular*: perfect mirror with no scattering
- *gloss*: mixed, partial specularity
- *diffuse*: all directions with equal energy

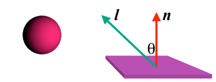



specular + glossy + diffuse = reflectance distribution



## Review: Reflection Equations

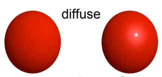
$$\mathbf{I}_{\mathbf{diffuse}} = \mathbf{k}_{\mathbf{d}}\, \mathbf{I}_{\mathbf{light}}\, (\mathbf{n} \bullet \mathbf{l})$$






## Specular Reflection

- shiny surfaces exhibit specular reflection
  - polished metal
  - glossy car finish




- specular highlight
  - bright spot from light shining on a specular surface
- view dependent
  - highlight position is function of the viewer's position



## Specular Highlights

Michiel van de Panne



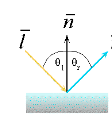
## Physics of Specular Reflection

- 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

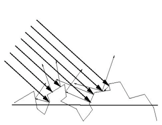



$$\theta_{(l)ight} = \theta_{(r)eflection}$$



## Non-Ideal Specular Reflectance

- Snell's law applies to perfect mirror-like surfaces, but aside from mirrors (and chrome) few surfaces exhibit perfect specularity
- 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…





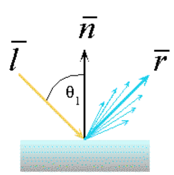
## Empirical Approximation

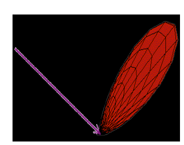
- we expect most reflected light to travel in direction predicted by Snell's Law
- but because of microscopic surface variations, some light may be reflected in a direction slightly off the ideal reflected ray
- as angle from ideal reflected ray increases, we expect less light to be reflected



## Empirical Approximation

- angular falloff



- how might we model this falloff?

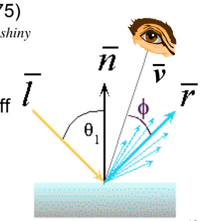


## Phong Lighting

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

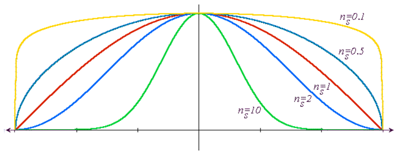
$$\mathbf{I}_{\mathbf{specular}} = \mathbf{k}_{\mathbf{s}} \mathbf{I}_{\mathbf{light}} (\cos\phi)^{n_{shiny}}$$

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



## Phong Lighting: The $n_{shiny}$ Term

- Phong reflectance term drops off with divergence of viewing angle from ideal reflected ray



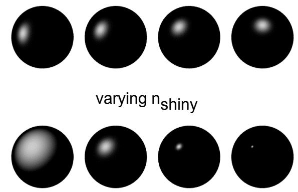

Viewing angle – reflected angle



## Phong Examples

varying l

varying $n_{shiny}$



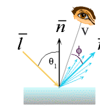


## Calculating Phong Lighting

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

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

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

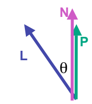



- how to efficiently calculate **r** ?



## Calculating R Vector

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

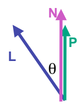





## Calculating R Vector

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

**P = N ( N · L )**



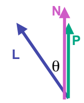



## Calculating R Vector

**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**

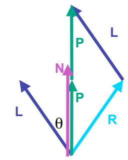
**P** = **N** cos θ **L, N** are unit length

**P = N ( N · L )**

**2 P = R + L**

**2 P – L = R**

**2 (N ( N · L )) - L = R**






## Phong Lighting Model

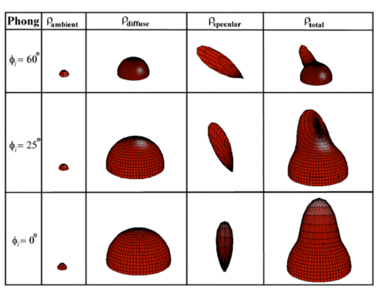
- combine ambient, diffuse, specular components

$$\mathbf{I}_{\mathbf{total}} = \mathbf{k}_\mathbf{a}\mathbf{I}_{\mathbf{ambient}} + \sum_{i=1}^{\#\,lights} \mathbf{I}_\mathbf{i}(\mathbf{k}_\mathbf{d}(\mathbf{n} \bullet \mathbf{l}_\mathbf{i}) + \mathbf{k}_\mathbf{s}(\mathbf{v} \bullet \mathbf{r}_\mathbf{i})^{n_{shiny}})$$

- 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



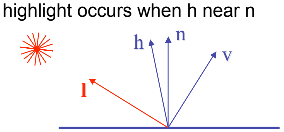



## Blinn-Phong Model

- variation with better physical interpretation
  - Jim Blinn, 1977

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

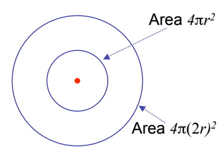
- ***h***: halfway vector
  - h must also be explicitly normalized: h / |h|
  - highlight occurs when h near n






## 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
    - $x$: 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

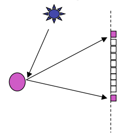
http://www.opengl.org/resources/features/KilgardTechniques/oglpitfall/



## 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



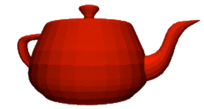
## Applying Illumination

- 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

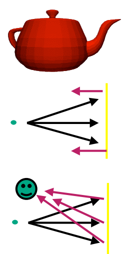


- obviously inaccurate for smooth surfaces



## Flat Shading Approximations

- if an object really is 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

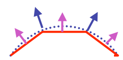




## Improving Flat Shading

- what if evaluate Phong lighting model at each pixel of the polygon?
  - better, but result still clearly faceted

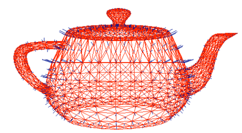
- for smoother-looking surfaces we introduce *vertex normals* at each vertex
  - usually different from facet normal
  - used *only* for shading
  - think of as a better approximation of the *real* surface that the polygons approximate





## 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

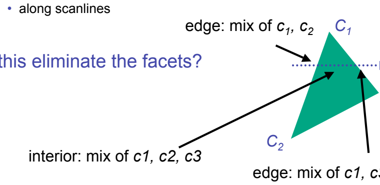




## Gouraud Shading

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

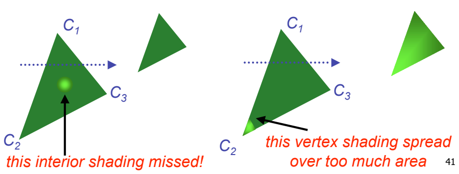
does this eliminate the facets?






## Gouraud Shading Artifacts

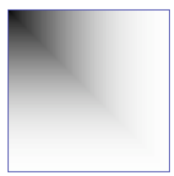
- often appears dull, chalky
- lacks accurate specular component
  - if included, will be averaged over entire polygon






## Gouraud Shading Artifacts

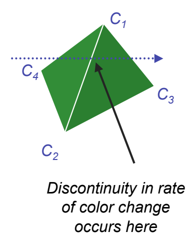
- Mach bands
  - eye enhances discontinuity in first derivative
  - very disturbing, especially for highlights

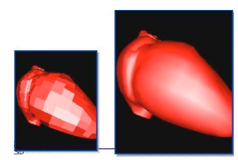




## Gouraud Shading Artifacts

- Mach bands

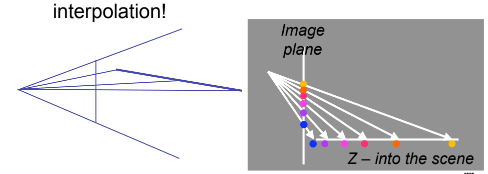




## Gouraud Shading Artifacts

- 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

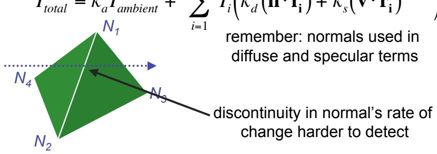


## Phong Shading

- 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



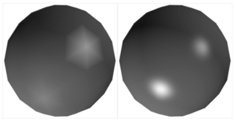



## Phong Shading Difficulties

- 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

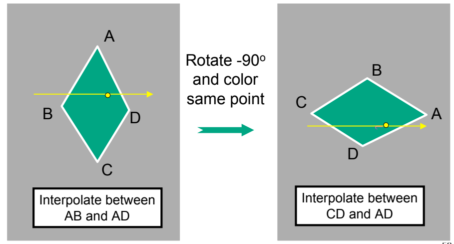
- polygonal silhouettes remain





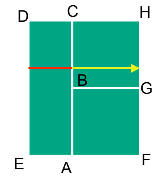

## Shading Artifacts: Orientation

- 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

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

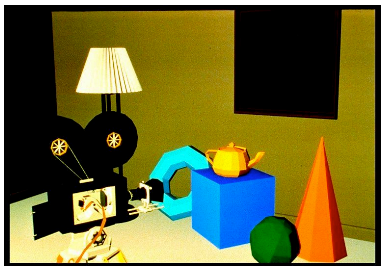


## Shading Models Summary

- 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

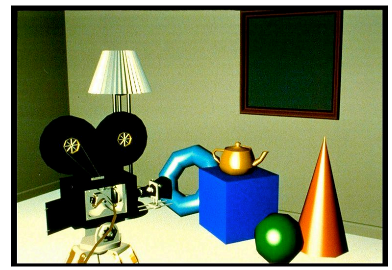




## Shutterbug: Gouraud Shading



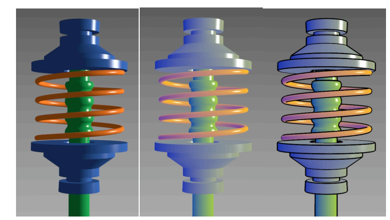
## Shutterbug: Phong Shading





## Non-Photorealistic Shading

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

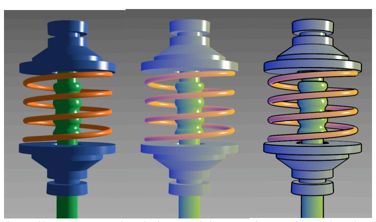


http://www.cs.utah.edu/~gooch/SIG98/paper/drawing.html



## 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 $(\mathbf{n_0}\cdot\mathbf{n_1}) \le threshold$

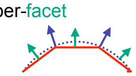


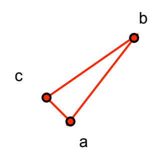
http://www.cs.utah.edu/~gooch/SIG98/paper/drawing.html



## Computing Normals

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

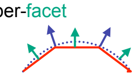


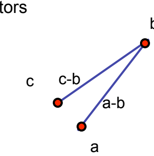





## Computing Normals

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

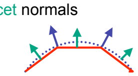


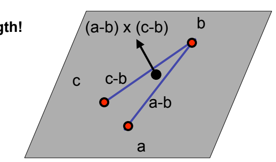





## Computing Normals

- 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);
```