University of British Columbia
CPSC 314 Computer Graphics
Jan-Apr 2008

Tamara Munzner

## Lighting/Shading IV
## Advanced Rendering I

### Week 8, Mon Mar 3

http://www.ugrad.cs.ubc.ca/~cs314/Vjan2008

## Midterm

- for all homeworks+exams
  - good to use fractions/trig functions as intermediate values to show work
  - but final answer should be decimal number
- allowed during midterm
  - calculator
  - one notes page, 8.5"x11", one side of page
    - your name at top, hand in with midterm, will be handed back
    - must be handwritten



## Midterm

- topics covered: through rasterization (H2)
  - rendering pipeline
  - transforms
  - viewing/projection
  - rasterization
- topics NOT covered
  - color, lighting/shading (from 2/15 onwards)
- H2 handed back, with solutions, on Wed



## FCG Reading For Midterm

- Ch 1
- Ch 2 Misc Math (except for 2.5.1, 2.5.3, 2.7.1, 2.7.3, 2.8, 2.9)
- Ch 5 Linear Algebra (only 5.1-5.2.2, 5.2.5)
- Ch 6 Transformation Matrices (except 6.1.6)
- Sect 13.3 Scene Graphs
- Ch 7 Viewing
- Ch 3 Raster Algorithms (except 3.2-3.4, 3.8)



## Red Book Reading For Midterm

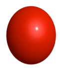
- Ch Introduction to OpenGL
- Ch State Management and Drawing Geometric Objects
- App Basics of GLUT (Aux in v 1.1)
- Ch Viewing
- App Homogeneous Coordinates and Transformation Matrices
- Ch Display Lists



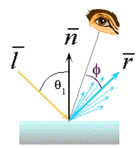
## Review: Reflection Equations

- Phong specular model

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



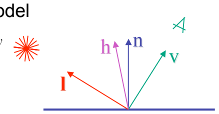
2 ( **N** (**N** · **L**)) – **L** = **R**




- or Blinn-Phong specular model

$$\mathbf{I}_{specular} = \mathbf{k}_s \mathbf{I}_{light} (\mathbf{h} \bullet \mathbf{n})^{n_{shiny}}$$

$$\mathbf{h} = (\mathbf{l} + \mathbf{v})/2$$






## Review: Reflection Equations

full Phong lighting model
- combine ambient, diffuse, specular components

$$\mathbf{I}_{total} = \mathbf{k}_a \mathbf{I}_{ambient} + \sum_{i=1}^{\#lights} \mathbf{I}_i (\mathbf{k}_d (\mathbf{n} \bullet \mathbf{l}_i) + \mathbf{k}_s (\mathbf{v} \bullet \mathbf{r}_i)^{n_{shiny}})$$

or $(\mathbf{h} \bullet \mathbf{n})$

- don't forget to normalize all vectors: n,l,r,v,h
  - n: normal to surface at point
  - l: vector between light and point on surface
  - r: mirror reflection (of light) vector
  - v: vector between viewpoint and point on surface
  - h: halfway vector (between light and viewpoint)



## Review: Lighting

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



## Review: Shading Models

- flat shading
  - compute Phong lighting once for entire polygon
- Gouraud shading
  - compute Phong lighting at the vertices and interpolate lighting values across polygon

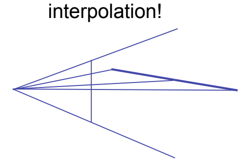


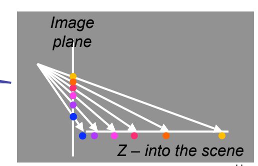
## Shading



## 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 illumination at a point on a surface

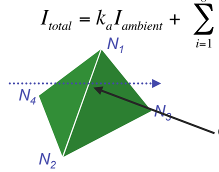


## Phong Shading

- 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



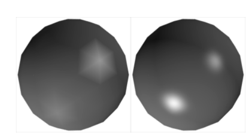
## Phong Shading Difficulties

- 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



## Shading Artifacts: Silhouettes

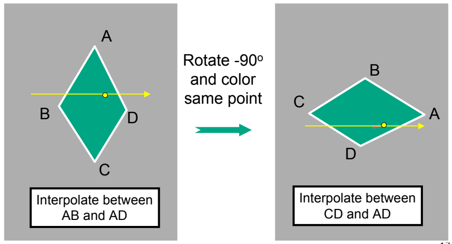
- polygonal silhouettes remain



*Gouraud* *Phong*

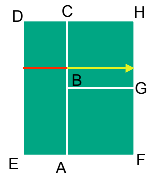
## Shading Artifacts: Orientation

- 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

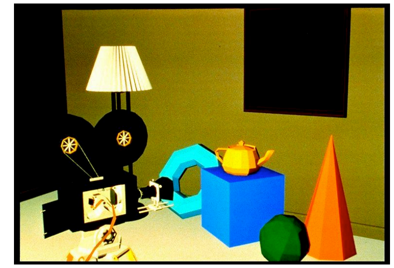


## Shading Models Summary

- 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

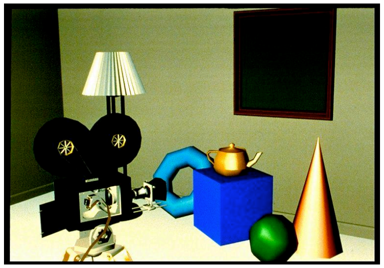


## Shutterbug: Flat Shading



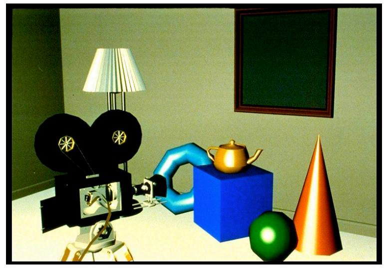


## Shutterbug: Gouraud Shading





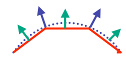
## Shutterbug: Phong Shading

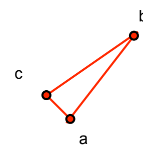




## Computing Normals

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

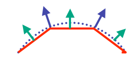


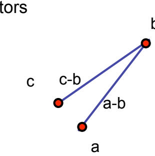





## Computing Normals

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

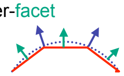


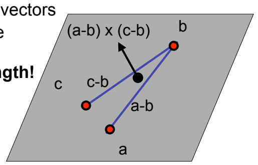





## Computing Normals

- 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);
```

- normal interpreted as direction from vertex location
- can automatically normalize (computational cost)

```
glEnable(GL_NORMALIZE);
```



## Advanced Rendering



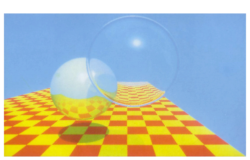
## Global Illumination Models

- simple lighting/shading methods simulate local illumination models
  - no object-object interaction
- global illumination models
  - more realism, more computation
  - leaving the pipeline for these two lectures!
- approaches
  - ray tracing
  - radiosity
  - photon mapping
  - subsurface scattering



## Ray Tracing

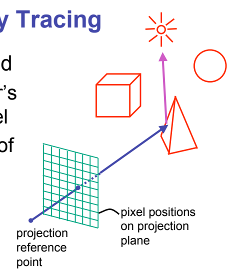
- simple basic algorithm
- well-suited for software rendering
- flexible, easy to incorporate new effects
  - Turner Whitted, 1990





## Simple Ray Tracing

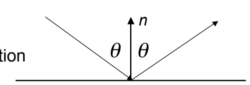
- view dependent method
  - cast a ray from viewer's eye through each pixel
  - compute intersection of ray with first object in scene
  - cast ray from intersection point on object to light sources

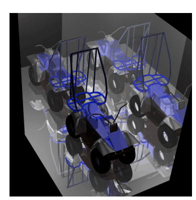


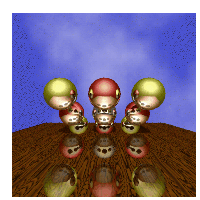



## Reflection

- mirror effects
  - perfect specular reflection



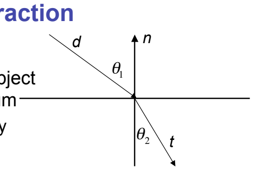



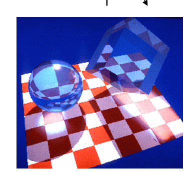




## Refraction

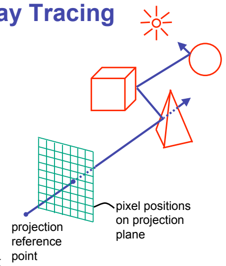
- happens at interface between transparent object and surrounding medium
  - e.g. glass/air boundary
- Snell's Law
  - $c_1 \sin\theta_1 = c_2 \sin\theta_2$
  - light ray bends based on refractive indices $c_1$, $c_2$



## Recursive Ray Tracing

- ray tracing can handle
  - reflection (chrome/mirror)
  - refraction (glass)
  - shadows
- spawn secondary rays
  - reflection, refraction
    - if another object is hit, recurse to find its color
  - shadow
    - cast ray from intersection point to light source, check if intersects another object



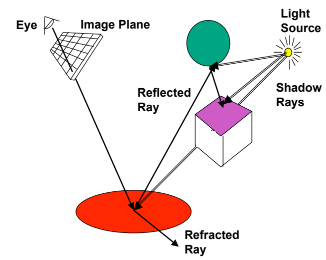



## Basic Algorithm

for every pixel $p_i$ {
  generate ray r from camera position through pixel $p_i$
  for every object o in scene {
    if ( r intersects o )
      compute lighting at intersection point, using local normal and material properties; store result in $p_i$
    else
      $p_i$= background color
  }
}



## Ray Tracing Algorithm






## Basic Ray Tracing Algorithm

**RayTrace**(r,scene)
obj := **FirstIntersection**(r,scene)
if (no obj) return BackgroundColor;
else begin
  if ( **Reflect**(obj) ) then
    reflect_color := **RayTrace**(**ReflectRay**(r,obj));
  else
    reflect_color := Black;
  if ( **Transparent**(obj) ) then
    refract_color := **RayTrace**(**RefractRay**(r,obj));
  else
    refract_color := Black;
  return **Shade**(reflect_color,refract_color,obj);
end;



## Algorithm Termination Criteria

- termination criteria
  - no intersection
  - reach maximal depth
    - number of bounces
  - contribution of secondary ray attenuated below threshold
    - each reflection/refraction attenuates ray



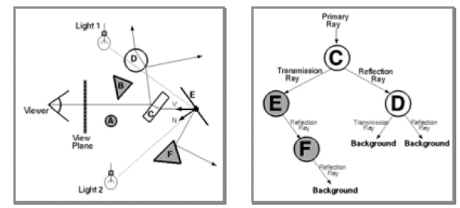
## Ray-Tracing Terminology

- terminology:
  - primary ray: ray starting at camera
  - shadow ray
  - reflected/refracted ray
  - ray tree: all rays directly or indirectly spawned off by a single primary ray
- note:
  - need to limit maximum depth of ray tree to ensure termination of ray-tracing process!



## Ray Trees

- all rays directly or indirectly spawned off by a single primary ray




Ray traced through scene | Ray tree

www.cs.virginia.edu/~gfx/Courses/2003/Intro.fall.03/slides/lighting_web/lighting.pdf



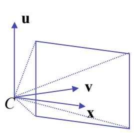
## Ray Tracing

- issues:
  - generation of rays
  - intersection of rays with geometric primitives
  - geometric transformations
  - lighting and shading
  - efficient data structures so we don't have to test intersection with *every* object



## Ray Generation

- camera coordinate system
  - origin: C (camera position)
  - viewing direction: $\mathbf{v}$
  - up vector: $\mathbf{u}$
  - x direction: $\mathbf{x} = \mathbf{v} \times \mathbf{u}$
- note:
  - corresponds to viewing transformation in rendering pipeline
  - like gluLookAt

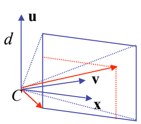





## Ray Generation

- other parameters:
  - distance of camera from image plane: $d$
  - image resolution (in pixels): $w, h$
  - left, right, top, bottom boundaries in image plane: $l, r, t, b$
- then:
  - lower left corner of image: $O = C + d\cdot\mathbf{v} + l\cdot\mathbf{x} + b\cdot\mathbf{u}$
  - pixel at position $i, j$ ($i=0..w\text{-}1, j=0..h\text{-}1$):

$$P_{i,j} = O + i\cdot\frac{r-l}{w-1}\cdot\mathbf{x} - j\cdot\frac{t-b}{h-1}\cdot\mathbf{u}$$
$$= O + i\cdot\Delta x\cdot\mathbf{x} - j\cdot\Delta y\cdot\mathbf{y}$$






## Ray Generation

- ray in 3D space:

$$R_{i,j}(t) = C + t\cdot(P_{i,j} - C) = C + t\cdot\mathbf{v}_{i,j}$$

where $t = 0\ldots\infty$



## Ray Tracing

- issues:
  - generation of rays
  - intersection of rays with geometric primitives
  - geometric transformations
  - lighting and shading
  - efficient data structures so we don't have to test intersection with *every* object



## Ray - Object Intersections

- inner loop of ray-tracing
  - must be extremely efficient
- task: given an object o, find ray parameter $t$, such that $\mathbf{R}_{i,j}(t)$ is a point on the object
  - such a value for $t$ may not exist
- solve a set of equations
- intersection test depends on geometric primitive
  - ray-sphere
  - ray-triangle
  - ray-polygon



## Ray Intersections: Spheres

- spheres at origin
  - implicit function

$$S(x,y,z): x^2 + y^2 + z^2 = r^2$$

  - ray equation

$$R_{i,j}(t) = C + t\cdot\mathbf{v}_{i,j} = \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} + t\cdot\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} c_x + t\cdot v_x \\ c_y + t\cdot v_y \\ c_z + t\cdot v_z \end{pmatrix}$$



## Ray Intersections: Spheres

- to determine intersection:
  - insert ray $\mathbf{R}_{i,j}(t)$ into $S(x,y,z)$:

$$(c_x + t\cdot v_x)^2 + (c_y + t\cdot v_y)^2 + (c_z + t\cdot v_z)^2 = r^2$$

  - solve for $t$ (find roots)
    - simple quadratic equation



## Ray Intersections: Other Primitives

- implicit functions
  - spheres at arbitrary positions
    - same thing
  - conic sections (hyperboloids, ellipsoids, paraboloids, cones, cylinders)
    - same thing (all are quadratic functions!)
- polygons
  - first intersect ray with plane
    - linear implicit function
  - then test whether point is inside or outside of polygon (2D test)
  - for convex polygons
    - suffices to test whether point in on the correct side of every boundary edge
    - similar to computation of outcodes in line clipping (upcoming)