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

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 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

Reading for Today and Tomorrow
- FCG Chap 10 Ray Tracing
  - only 10.1-10.7
- FCG Chap 25 Image-Based Rendering

News
- extra lab coverage: TAs available to answer questions
  - Wed 2-3, 5-6 (Matt)
  - Thu 11-2 (Matt)
  - Thu 3:30-5:30 (Gordon)
  - Fri 2-5 (Gordon)

Final Correction/Clarification: 3D Shear
- General shear: shear along x, y, z
- “n-shear” usually means shear along n in direction of some other axis
- correction: not shear along some axis in direction of n
- to avoid ambiguity, always say “shear along n in direction of n”

Correction/Review: Reflection Equations
- Blinn improvement
  \[ I_\text{specular} = k_i I_\text{light} (\mathbf{h} \cdot \mathbf{n})^{\alpha \text{spec}} \]
  \[ \mathbf{h} = (\mathbf{l} + \mathbf{v})/2 \]
  - full Phong lighting model
    - combine ambient, diffuse, specular components
    \[ I_\text{total} = I_\text{ambient} + \sum_{i=1}^{3} k_i (\mathbf{n} \cdot \mathbf{l}_i) + k_s (\mathbf{v} \cdot \mathbf{r}_i)^{\alpha \text{spec}} \]
  - don’t forget to normalize all vectors: n,i,v,h

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

Shading
- Gouraud
- Phong

Phong 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

News
- Project 2
  - rolling ball mode should rotate around center of world, not center of camera
  - corrected example binary will be posted soon

Correction
- Homework 2 Q9 was underconstrained
  - “Sketch what the resulting image would look like with an oblique angle of 70 degrees”
  - add: and a length of .7 for lines perpendicular to the image plane
  - question is now extra credit

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

Extra lab coverage: TAs available to answer questions
- Wed 2-3, 5-6 (Matt)
- Thu 11-2 (Matt)
- Thu 3:30-5:30 (Gordon)
- Fri 2-5 (Gordon)

Shutterbug: Flat Shading
Ray Tracing
• simple basic algorithm
• well-suited for software rendering
• flexible, easy to incorporate new effects
  • Turner Whitted, 1990

Reminder: Computing Normals
• per-vertex normals by interpolating per-facet normals
• OpenGL supports both
• computing normal for a polygon
• three points form two vectors
  • which side is up?
  • convention: points in counterclockwise order

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

Specifying Normals
• OpenGL state machine
  • uses last normal specified
  • if no normals specified, assumes all identical

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

Shutterbug: Gouraud Shading
• vertex lighting/shading

Shutterbug: Phong Shading
• mirror effects
  • perfect specular reflection

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

Recursion 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

Simple Ray Tracing
• 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 $c_1$, $c_2$

Advanced Rendering
• simple lighting/shading methods simulate local illumination models
  • no object-object interaction

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 })

Shading
• perfect specular reflection

Global Illumination Models
• simple lighting/shading methods simulate local illumination models
  • no object-object interaction

Basic Ray Tracing Algorithm
```
RayTrace(r, scene)
obj := FirstIntersection(r, scene)
if (no obj) return BackgroundColor;
else begin
  if (Reflect(obj)) then refect_color := RayTrace(RefractRay(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

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

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 $c_1$, $c_2$
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)
- ray tree: all rays directly or indirectly spawned

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: \( l, r, t, b \)
- ray in 3D space:
  \[
  R_{ij}(t) = C + t \cdot (P_{ij} - C) = C + t \cdot v_{ij}
  \]
  where \( i = 0 \ldots \infty \)

Ray Intersections: Spheres

- to determine intersection:
  - insert ray \( R_{ij}(t) \) into \( S(x,y,z) \):
    \[
    (c_x + t \cdot v_{ij})^2 + (c_y + t \cdot v_{ij})^2 + (c_z + t \cdot v_{ij})^2 = r^2
    \]
  - solve for \( t \) (find roots)
  - simple quadratic equation

Ray Intersections: Spheres

- implicit function
  \[
  S(x,y,z): x^2 + y^2 + z^2 = r^2
  \]
- ray equation
  \[
  R_{ij}(t) = C + t \cdot v_{ij} = \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} + t \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} c_x + t \cdot v_x \\ c_y + t \cdot v_y \\ c_z + t \cdot v_z \end{pmatrix}
  \]

Ray Intersections

- 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 Tracing

- task:
  - given an object \( o \), find ray parameter \( t \), such that \( R_{ij}(t) \) is a point on the object
  - such a value for \( t \) may not exist
  - intersection test depends on geometric primitive

Credits

- some of raytracing material from Wolfgang Heidrich
  - http://www.ugrad.cs.ubc.ca/~cs314/WHmay2006/

Ray Generation

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

Ray Tracing Terminology

- terminology:
  - primary ray: ray starting at camera
  - shadow ray
  - reflected/refracted ray
  - ray tree: all rays directly or indirectly spawned

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 Intersections

- ray: \( x(t) = p_x + v_x \cdot t \), \( y(t) = p_y + v_y \cdot t \), \( z(t) = p_z + v_z \cdot t \)
- unit sphere: \( x^2 + y^2 + z^2 = 1 \)
- quadratic equation in \( t \):
  \[
  x^2 + y^2 + z^2 = \frac{1}{r^2} \]

Ray Intersections: Spheres

- sphere at arbitrary positions
- same thing
- conic sections (hyperboloids, ellipsoids, paraboloids, cones, cylinders)
- same thing (all are quadratic functions)