Shading, Advanced Rendering

Week 7, Wed Feb 28

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)
News

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

• 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
Final Correction/Clarification: 3D Shear

• general shear \( \text{shear}(h_{xy}, h_{xz}, h_{yx}, h_{yz}, h_{zx}, h_{zy}) = \begin{bmatrix} 1 & h_{yx} & h_{zx} & 0 \\ h_{xy} & 1 & h_{zy} & 0 \\ h_{xz} & h_{yz} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \)

• "x-shear" usually means shear along x in direction of some other axis
  • correction: not shear along some axis in direction of x
  • to avoid ambiguity, always say "shear along <axis> in direction of <axis>"

\[
\begin{align*}
\text{shearAlongXinDirectionOfY}(h) &= \begin{bmatrix} 1 & h & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\text{shearAlongXinDirectionOfZ}(h) &= \begin{bmatrix} 1 & 0 & h & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\text{shearAlongYinDirectionOfX}(h) &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ h & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\text{shearAlongYinDirectionOfZ}(h) &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & h & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\text{shearAlongZinDirectionOfX}(h) &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ h & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\text{shearAlongZinDirectionOfY}(h) &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & h & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{align*}
\]
Correction/Review: Reflection Equations

- Blinn improvement

\[ I_{\text{specular}} = k_s I_{\text{light}} (h \cdot n)^{n_{\text{shiny}}} \]
\[ h = (l + v)/2 \]

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

\[ I_{\text{total}} = k_a I_{\text{ambient}} + \sum_{i=1}^{\text{# lights}} I_i (k_d (n \cdot l_i) + k_s (v \cdot r_i)^{n_{\text{shiny}}}) \]

- don’t forget to normalize all vectors: n,l,r,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
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
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

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

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
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
Reminder: Computing Normals

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

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

![Diagram showing vectors and normal calculation]

\[ \vec{c} - \vec{b}, \quad \vec{a} - \vec{b} \]
Reminder: 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);
  ```
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
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
    }
}
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 Algorithm

Eye ➔ Image Plane ➔ Reflected Ray ➔ Refracted Ray ➔ Shadow Rays ➔ Light Source
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 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
• solve a set of equations
  • ray-sphere
  • ray-triangle
  • ray-polygon
Ray - Sphere Intersection

• ray: \( x(t) = p_x + v_x t, \quad y(t) = p_y + v_y t, \quad z(t) = p_z + v_z t \)

• unit sphere: \( x^2 + y^2 + z^2 = 1 \)

• quadratic equation in \( t \):

\[
0 = (p_x + v_x t)^2 + (p_y + v_y t)^2 + (p_z + v_z t)^2 - 1
\]
\[
= t^2 (v_x^2 + v_y^2 + v_z^2) + 2t(p_x v_x + p_y v_y + p_z v_z)
\]
\[
+ (p_x^2 + p_y^2 + p_z^2) - 1
\]
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: \( O = C + d \cdot v + l \cdot x + b \cdot u \)
  - pixel at position \( i, j \) \((i=0..w-1, j=0..h-1)\):
    \[
    P_{i,j} = O + i \cdot \frac{r - l}{w - 1} \cdot x - j \cdot \frac{t - b}{h - 1} \cdot u \\
    = O + i \cdot \Delta x \cdot x - j \cdot \Delta y \cdot y
    \]
Ray Generation

• ray in 3D space:

\[ R_{i,j}(t) = C + t \cdot (P_{i,j} - C) = C + t \cdot v_{i,j} \]

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 Intersections

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

- spheres at origin
  - implicit function
    \[ S(x, y, z): x^2 + y^2 + z^2 = r^2 \]
- ray equation
  \[
  \mathbf{R}_{i,j}(t) = C + t \cdot \mathbf{v}_{i,j} = \\
  \begin{pmatrix}
  c_x \\
  c_y \\
  c_z
  \end{pmatrix} + t \cdot \\
  \begin{pmatrix}
  v_x \\
  v_y \\
  v_z
  \end{pmatrix} = \\
  \begin{pmatrix}
  c_x + t \cdot v_x \\
  c_y + t \cdot v_y \\
  c_z + t \cdot v_z
  \end{pmatrix}
  \]
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)
Credits

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