Reading for Last and This Time
• FCG Chap 10 Ray Tracing
  • only 10.1-10.7
• FCG Chap 25 Image-Based Rendering

Review/Clarification: Specifying Normals
• OpenGL state machine
  • uses last normal specified
  • if no normals specified, assumes all identical
• per-vertex normals
  • normals interpreted as direction from vertex location
    • can automatically normalize (computational cost)
  • glNormalize();
• per-face normals
  • implicit function generation of rays
    • only 10.1-10.7
  • intersection of rays with geometric primitives

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

Ray Trees
• all rays directly or indirectly spawned off by a single primary ray

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

Review: Recursive Ray Tracing
• ray tracing can handle
  • reflection (chrome/mirror)
  • refraction (glass)
  • shadows
• one primary ray per pixel
• 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
• termination criteria
  • no intersection (ray exits scene)
  • max bounces (recursion depth)
  • attenuated below threshold

Review: Reflection and Refraction
• refraction: mirror effects
• perfect specular reflection
• refraction: at boundary
  • Snell's Law
    • light ray bends based on refractive indices \( c_1, c_2 \)
    \[ \theta_1 \sin \theta_1 = c_1 \sin \theta_2 \]

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
  • compute averaged vertex normals
  • interpolate normals across polygon and perform Phong lighting across polygon

Review: Ray Intersections: Spheres
• spheres at origin
• implicit function
  \[ S(x, y, z); x^2 + y^2 + z^2 = r^2 \]
• ray equation
  \[ R_{ij}(t) = C + t \cdot v_{ij} = \left( \begin{array}{c} C_x + t \cdot v_{ij} \cdot x \\ C_y + t \cdot v_{ij} \cdot y \\ C_z + t \cdot v_{ij} \cdot z \end{array} \right) \]
**Ray Intersections: Spheres**
- to determine intersection:
  - insert ray \( R(t) \) into \( S(x,y,z) \):
    \[
    (x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2 = r^2
    \]
- 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
  - suffice to test whether point is on the correct side of every boundary edge
- similar to computation of outsides in line clipping (upcoming)

**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-Intersection**
- method in book is elegant but a bit complex
- easier approach: triangle is just a polygon
  - intersect ray with plane
    \[
    \text{normal: } n = (b-a) \times (c-a) \\
    \text{ray: } x = e + td
    \]
  - plane: \( (p-x) \cdot n = 0 \Rightarrow x = \frac{p \cdot n}{n} \]
  - solve for \( t\)
  - if dot is positive
    \[
    \begin{vmatrix}
    b-a \times (x-a) \cdot n \\
    c-b \times (x-b) \cdot n \\
    a-c \times (x-c) \cdot n
    \end{vmatrix} \geq 0
    \]
- check if ray inside triangle
  - check if point counterclockwise from each edge (to its left)
  - check if cross product points in same direction as normal (i.e. if dot is positive)

**Optimized Ray-Tracing**
- basic algorithm simple but very expensive
- optimize by reducing:
  - number of rays traced
  - number of ray-object intersection calculations
- methods
  - bounding volumes: boxes, spheres
  - spatial subdivision
    - uniform
    - BSP trees
- (more on this later with collision)

**Example Images**
- local surface information (normal…)
  - for implicit surfaces \( F(x,y,z)=0 \): normal \( n(x,y,z) \)
    can be easily computed at every intersection point using the gradient:
    \[
    n(x,y,z) = \frac{\partial F}{\partial x} \times \frac{\partial F}{\partial y} \times \frac{\partial F}{\partial z}
    \]
- example:
  \[
  n(x,y,z) = \frac{2x}{2} \times \frac{2y}{2} \times \frac{2z}{2}
  \]
  needs to be normalized!

---

**Geometric Transformations**
- similar goal as in rendering pipeline:
  - modeling scenes more convenient using different coordinate systems for individual objects
- problem
  - not all object representations are easy to transform
  - problem is fixed in rendering pipeline by restriction to polygons, which are affine invariant
  - ray tracing has different solution
  - ray itself is always affine invariant
  - thus: transform ray into object coordinates!

**Geometric Transformations**
- ray transformation
  - for intersection test, it is only important that ray is in same coordinate system as object representation
  - transform all rays into object coordinates
    - transform camera point and ray direction by inverse of model/view matrix
  - shading has to be done in world coordinates (where light sources are given)
    - transform object space intersection point to world coordinates
    - thus have to keep both world and object-space ray

---

**Total Internal Reflection**
- as the angle of incidence increase from 0 to greater angles ...
  - the reflected ray becomes dimmer (there is less refraction)
  - the reflected ray becomes brighter (there is more reflection)
  - the angle of refraction approaches 90 degrees until finally a reflected ray no longer be seen.

http://www.physicsclassroom.com/class/refmU14L3b.html
**Image-Based Modelling and Rendering**

- store and access only pixels
  - no geometry, no light simulation, ...
- input: set of images
- output: Image from new viewpoint
  - surprisingly large set of possible new viewpoints
  - interpolation allows translation, not just rotation
  - lightfield, lumigraph: translate outside convex hull of object
  - QuickTimeVR: camera rotates, no translation
- can point camera in or out

**Image-Based Rendering**

- display time not tied to scene complexity
- expensive rendering or real photographs
- example: Matrix bullet-time scene
- array of many cameras allows virtual camera to "freeze time"
- convergence of graphics, vision, photography
- computational photography

**Subsurface Scattering: Translucency**

- light enters and leaves at different locations on the surface
- bounces around inside
- technical Academy Award, 2003
  - Jensen, Marschner, Hanrahan

**Subsurface Scattering: Marble**

**Subsurface Scattering: Milk vs. Paint**

**Non-Photorealistic Rendering**

- simulate look of hand-drawn sketches or paintings, using digital models

**Non-Photorealistic Shading**

- cool-to-warm shading
  \[ k_w = 1 + n \cdot l^2, c = k_w c_w + (1 - k_w) c_c \]
- standard cool-to-warm with edges/creases

**Better Global Illumination**

- ray-tracing: great specular, approx. diffuse
- view dependent
- radiosity: great diffuse, specular ignored
- view independent, mostly-enclosed volumes
- photon mapping: superset of raytracing and radiosity
- view dependent, handles both diffuse and specular well