Advanced Rendering II, Clipping I

Week 8, Wed Mar 10

Project 3 out
• due Fri Mar 26, 5pm
• raytracer
• template code has significant functionality
• clearly marked places where you need to fill in required code

Spawn secondary rays
• one primary ray per pixel

Review: Specifying Normals
• OpenGL state machine
  • uses last normal specified
  • if no normals specified, assumes all identical
• per-vertex normals
  gNorms=0
  gNorms=1
• per-face normals
  gNorms=0
  gNorms=1
• normal interpreted as direction from vertex location
• can automatically normalize (computational cost) gNormalize(NORMAL)

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 first 6 colors
• 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: 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-Triangle Intersection
• method in book is elegant but a bit complex
  • easier approach: triangle is just a polygon
  • intersect ray with plane
    normal: n=(b-a)x(c-a)
    ray: x=e+td
    plane: (p-x)n=0
    if p is a or b or c
      check if ray inside triangle
    else
      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)
      (b-a)x(x-a)n>=0
      (c-b)x(x-b)n>=0
      (a-c)x(x-c)n>=0

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

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

Local Lighting
• 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
  \[
  n(x,y,z) = \frac{\partial F(x,y,z)}{\partial x} \hat{i} + \frac{\partial F(x,y,z)}{\partial y} \hat{j} + \frac{\partial F(x,y,z)}{\partial z} \hat{k}
  \]
  • example:
    \[
    \begin{align*}
    n(x,y,z) &= \frac{2x}{2x^2 + 2y^2 + 2z^2} \\
    &= \frac{2}{\sqrt{2x^2 + 2y^2 + 2z^2}} \\
    &= \frac{2}{\sqrt{2r^2}} \\
    &= \frac{2}{\sqrt{2}} \frac{1}{r}
    \end{align*}
    \]
  needs to be normalized!
Local Lighting
- local surface information
  • alternatively: can interpolate per-vertex information for triangles/meshes as in rendering pipeline
  • now easy to use Phong shading!
  • as discussed for rendering pipeline
- difference with rendering pipeline:
  • interpolation cannot be done incrementally
  • have to compute barycentric coordinates for every intersection point (e.g. plane equation for triangles)

Global Shadows
- approach
  • to test whether point is in shadow, send out shadow rays to all light sources
  • if ray hits another object, the point lies in shadow

Global Reflections/Refractions
- approach
  • send rays out in reflected and refracted direction to gather incoming light
  • that light is multiplied by local surface color and added to result of local shading

Radiosity
- illumination as radiative heat transfer
- conserve light energy in a volume
- model light transport as packet flow until convergence
- solution captures diffuse-diffuse bouncing of light
- view-independent technique
- calculate solution for entire scene offline
- browse from any viewpoint in realtime

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)

Subsurface Scattering: Marble

Subsurface Scattering: Milk vs. Paint

Subsurface Scattering: Skin

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

Total Internal Reflection
http://www.physicsclassroom.com/Class/refrn/U14L3b.html
Non-Photorealistic Rendering
• simulate look of hand-drawn sketches or paintings, using digital models

www.red3d.com/cwr/npr/

Cohen-Sutherland Line Clipping
• assign outcode to each vertex of line to test
  • line segment: \((p1, p2)\)
  • trivial cases
    • \(OC(p1) = 0 \& OC(p2) = 0\)
      • both points inside window, thus line segment completely visible
        (trivial accept)
    • \(OC(p1) \& OC(p2) \neq 0\)
      • there is (at least) one boundary for which both points are outside
        (same flag set in both outcodes)
        • thus line segment completely outside window (trivial reject)

Clipping
• analytically calculating the portions of primitives within the viewport

Clipping
• we've been assuming that all primitives (lines, triangles, polygons) lie entirely within the viewport
• in general, this assumption will not hold:

Clipping
• naïve approach to clipping lines:
  • for each line segment
  • for each edge of viewport
  • find intersection point
  • pick “nearest” point
  • if anything is left, draw it
  • what do we mean by “nearest”?
  • how can we optimize this?

Clipping
• 2D
  • determine portion of line inside an axis-aligned rectangle (screen or window)
• 3D
  • determine portion of line inside axis-aligned parallelepiped (viewing frustum in NDC)
  • simple extension to 2D algorithms

Clipping Lines To Viewport
• combining trivial accepts/rejects
  • trivially accept lines with both endpoints inside all edges of the viewport
  • trivially reject lines with both endpoints outside the same edge of the viewport
  • otherwise, reduce to trivial cases by splitting into two segments

Clipping
• 2D
  • bad idea to rasterize outside of framebuffer bounds
  • also, don’t waste time scan converting pixels outside window
  • could be billions of pixels for very close objects!

Why Clip?
• Q: how can we know a line is outside viewport?
  • A: if both endpoints on wrong side of same edge, can trivially reject line

Why Clip?
• 3D
  • Q: how can we quickly determine whether a line segment is entirely inside the viewport?
  • A: test both endpoints

Next Topic: Clipping
• we've been assuming that all primitives (lines, triangles, polygons) lie entirely within the viewport
• in general, this assumption will not hold:

Why Clip?
• Q: how can we quickly determine whether a line segment is entirely inside the viewport?
  • A: test both endpoints

Why Clip?
• Q: how can we know a line is outside viewport?
  • A: if both endpoints on wrong side of same edge, can trivially reject line

Clipping
• 3D
  • determine portion of line inside axis-aligned parallelepiped (viewing frustum in NDC)
  • simple extension to 2D algorithms

Clipping
• 2D
  • determine portion of line inside an axis-aligned rectangle (screen or window)
Cohen-Sutherland Line Clipping
- intersect line with edge

Sutherland-Hodgeman Clipping
- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped

Viewport Intersection Code
- \((x_1, y_1), (x_2, y_2)\) intersect horizon edge at \(y_{bottom}\)
  - \(x_{intersect} = x_1 + (y_{bottom} - y_1)/m\)
  - \(m = (y_2 - y_1)/(x_2 - x_1)\)

Polygon Clipping
- objective
  - 2D: clip polygon against rectangular window
  - or general convex polygons
  - extensions for non-convex or general polygons
  - 3D: clip polygon against parallelepiped

Why Is Clipping Hard?
- a really tough case:
**Sutherland-Hodgeman Clipping**

- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped

**Sutherland-Hodgeman Discussion**

- similar to Cohen/Sutherland line clipping
- inside/outside tests: outcodes
- intersection of line segment with edge: window-edge coordinates
- clipping against individual edges independent
  - great for hardware (pipelining)
- all vertices required in memory at same time
  - not so good, but unavoidable
- another reason for using triangles only in hardware rendering

**Clipping Against One Edge**

- **p[i] inside:** 2 cases

  - inside
  - outside

  - output: p[i]

- **p[i] outside:** 2 cases

  - inside
  - outside

  - output: p, p[i]

**Clipping Against One Edge Example**

```
clipPolygonToEdge( p[n], edge ) {
    for( i= 0 ; i< n ; i++ ) {
        if( p[i] inside edge ) {
            output p[i];
        }
        else { // p[i] is outside edge
            if( p[i-1] inside edge ) {
                p = intersect(p[i-1], p[i], edge );
                output p, p[i];
            }
            else { // p[i] is outside edge
                p = intersect(p[i-1], p[i], edge );
                output p;
            }
        }
    }
}
```