Ray Tracing

Background

- Turner Whitted, about 1980
- Global Illumination, combining…
  - Direct lighting
  - Indirect lighting (reflected diffuse light)
  - Shadows, reflections, refractions, etc…

Types of Rays

Four types of rays used in a ray tracer:

1. **Pixel ray**: The ray that goes from the viewer through the pixel of the rendering surface.
2. **Reflection ray**: A reflected ray that goes from the intersection point on a surface in the reflected direction from that surface.
3. **Refraction ray**: A refracted ray that goes from the intersection point on a surface in the refracted direction from that surface.
4. **Shadow ray**: A ray that goes from the intersection point on a surface to a light; used to determine if the intersection point should be illuminated.

Definition of a Ray

Implicit form: $p(t) = o + td\,;\, o = origin\,,\, d = direction\,,\, t = time$

Given two points, a & b, a ray starting at a and going through b is

$$p(t) = a + t(b - a)\,;\, where\ t \in [0, \infty]$$

Generating Pixel Rays

Given a rendering surface with X horizontal pixels and Y vertical pixels and a window coordinate system of $[0, 1]^2$, a pixel ray is…

$$p_x = \frac{x + 0.5}{X},\ p_y = \frac{y + 0.5}{Y}$$ where x & y are in the range of [0, 1];

The + 0.5 is to have the ray go through the center of the pixel.

Implicit Surface Intersections

Implicit form of equation:

$f(x,y,z)=0; f(p)=0 \; where \; p=(x,y,z)$ defines a set of points on the surface

Given a ray $p(t)=o+td$ & surface $f(p)$ intersection occurs at:

$f(p(t))=0; f(o+td)=0$

Plane

- A point on a plane: a
- Normal to the surface: n
- Implicit equation is: $(p-a)\cdot n=0$; p is any point on the plane
- Plug in the ray and we get: $(o+td-a)\cdot n=0$; leaving t as the only unknown
- Solving for t we get: $t=\frac{(a-o)\cdot n}{d\cdot n}$
  - Case 1: denominator is 0: ray is parallel to the plane
  - Case 2: numerator and denominator are both 0 : ray is in the plane
  - Case 3: t is the intersection; plug into equation of ray to get the intersection point

Sphere

- Center of (c_x, c_y, c_z) and radius r
- Implicit equation is: $(x-c_x)^2+(y-c_y)^2+(z-c_z)^2-r^2=0$; (x, y, z) is a point on the surface
- In vector form: $(p-c)\cdot(p-c)-r^2=0$; p is a point on the surface
- Plug in the ray and we get: $(o+td-c)\cdot(o+td-c)-r^2=0$
- Rearrange to get: $(d\cdot d)t^2+2d\cdot(o-c)t+(o-c)\cdot(o-c)-r^2=0$
  - This is a quadratic form: $At^2+Bt+C=0$
  - Therefore can use the quadratic equation to solve for t: $t=\frac{-B\pm\sqrt{B^2-4AC}}{2A}$
  - $A=d\cdot d$
  - $B=2(d\cdot(o-c))$
  - $C=(o-c)\cdot(o-c)-r^2$
  - discriminant is B^2-4AC
- Substituting above: $t=\frac{-d\cdot(o-c)\pm\sqrt{(d\cdot(o-c))^2-(d\cdot d)((o-c)\cdot(o-c)-r^2)}}{(d\cdot d)}$
- Intersections
  - Case 1 : if discriminant is negative, square root is imaginary, therefore no intersection
  - Case 2 : if discriminant is 0, exactly 1 intersection (tangent to sphere)
  - Case 3 : if discriminant is positive, 2 intersections, entry & exit
- Normal at point p: $n=(p-c)$

(real-time) Ray Tracing Program

This section provides details on writing a ray tracing “program” using GLSL.

Basic Components

- Light structure
- Ray structure
- Plane structure
- Sphere structure
- Intersection structure: Contains all the information needed by the ray tracing algorithm to continue following an intersection.
    - boolean ‘hit’
    - t
    - surface normal
    - reflected ray
    - surface material properties
- Scene Details
    - Position of the eye/viewer
    - Scene geometry (planes & spheres in our case)

Ray Tracing Algorithm

- for each pixel
  - compute ray from eye through the pixel
    - "cast a ray"
      - for each object
        - hit test
        - if hit, compute color
          - if hit & diffuse return computed color
            - "cast a shadow ray" to each light, accumulate all intensities
          - if hit & reflective, use reflected ray and "cast a ray"
            - Advance the ray origin in the ray direction by a small epsilon to avoid self-intersections.
      - Select color from intersection that was closest to the eye
      - If nothing was hit, do something (sky color)
    - end "cast a ray"
- end for each pixel

GLSL Considerations

OpenGL is designed to rasterize polygons and we need to adapt this rasterization approach to enable us to do generalized ray tracing. The following notes help us do that.

- Simulate recursion through use of a stack (code provided to you)
- Use two triangles that cover the rendering surface and have OpenGL rasterize them, we'll use the position of each fragment as the "pixel" position to generate the pixel ray.
  - No need for projection, anything, define the triangles in Normalized Device Coordinates (NDC), $[-1, 1]^2$
  - Vertex shader simply passes the NDC position to the fragment shader
  - (in JavaScript) compute the half-pixel width and pass that into the shader so it can correctly compute the center of each pixel.