Ray Tracing Notes

Basic Ray Tracer

for each pixel

compute a viewing ray

find the first object hit by the ray

set the pixel color based on the hit point, the location of the lights and **n**

computing the viewing ray

a ray is a line with one end point

the easiest representation is a parametric one

parametric equation for the line AB: A + (B-A)t

a ray is the same, but A is the end point

alternatively, A + **d**t, where **d** is the direction of the ray

given a pixel, what is the viewing ray?

corresponds to the type of projection

Ray equations

orthographic projection

all of the rays should pass through the projection plane in parallel

imagine a plane aligned with the lens of the camera

we pick a point on this plane that corresponds to the pixel we want and then shoot a ray in the -Z direction to the projection plane (which can be anywhere in the -Z direction since this is parallel projection)

we do have the problem of matching the pixels to points in our camera space

image has pixels and we want it to map to a region of our space defined by l,r,t, and b

the size of a pixel in our camera space is thus

$$\frac{r-l}{n_x} \times \frac{t-b}{n_y}$$

we want a point that is in the center of the pixel, so given a pixel at (i,j)

$$x = l + \frac{(r-l)(i+0.5)}{n_x}$$

$$y = t - \frac{(t-b)(j+0.5)}{n_y}$$

so, our ray is (x,y) + -**z**t

perspective projection

much of this works the same

what is the only difference?

all rays have the same origin

eye + (x**x** + y**y** - d**z**)t

everything is controlled by the size of our projection plane and its distance from us

Finding intersections

find the first object struck by the ray

intersection with a plane

there are a number of ways we can do this, but we will use a technique that relies on the **point normal form** of a plane (also known as the **implicit form**)

the point normal form is based on the observation that if we have two points (P and Q) on a plane and we know the normal, then $\mathbf{n} \cdot (Q - P) = 0$

this may not seem very useful, but given *one* point and a normal vector, this is now a useful test to see if a point is on the plane or not

our intersection point would be a point, so…

$\mathbf{n} \cdot (E + \mathbf{d}t - P) = 0$

$\mathbf{n} \cdot (\mathbf{d}t + (E - P)) = 0$

$\mathbf{n} \cdot \mathbf{d}t + \mathbf{n} \cdot (E - P) = 0$

$\mathbf{n} \cdot \mathbf{d}t = \mathbf{n} \cdot (P - E)$

$t = (\mathbf{n} \cdot (P - E))/\mathbf{n} \cdot \mathbf{d}$

of course, we have a problem if $\mathbf{n}\cdot\mathbf{d} = 0$, but what does that mean anyway?

if $\mathbf{n}\cdot\mathbf{d} = 0$, the ray is orthogonal to the normal (i.e., parallel to the plane), no strike

if $\mathbf{n}\cdot\mathbf{d} > 0$, the ray is hitting the top of the plane

if $\mathbf{n}\cdot\mathbf{d} < 0$, the ray is coming up through the plane in the direction of the normal

if it strikes, then we solve for t and use that to figure out where

intersection with a triangle

the most common approach is to use **barycentric coordinates** (or the parametric form)

basically a non-orthogonal coordinate system formed by the points of the triangle

the basic idea is that we can express any point on the plane defined by the three points of the triangle with respect to those three points

$P = A + (B-A)\beta + (C - A)\gamma$

$P = (1 - \beta - \gamma)A + B\beta + C\gamma$

$\alpha = (1 - \beta - \gamma)$

this gives us the parametric equation

$P(\alpha, \beta, \gamma) = A\alpha + B\beta + C\gamma$, with the constraint that $\alpha + \beta + \gamma = 1$

a point is inside the triangle if $0<\alpha<1$, $0<\beta<1$, $0<\gamma<1$

this parametric form is actually how we do the interpolation of color (and normals) across the surface of a triangle

Given triangle ABC, and ray E +**d**t

E + **d**t = A + (B - A)β + (C-A)ɣ

this gives us three equations and three unknowns

$$x_e + x_d t = x_a + (x_b - x_a)\beta + (x_c - x_a)\gamma$$
$$y_e + y_d t = y_a + (y_b - y_a)\beta + (y_c - y_a)\gamma$$
$$z_e + z_d t = z_a + (z_b - z_a)\beta + (z_c - z_a)\gamma$$

|

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}$$

to make the discussion easier, we will abstract to

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} j \\ k \\ l \end{bmatrix}$$

Using **Cramer's rule**

Ax = b, x is the column vector of unknown variables

$$x_i = \frac{\det(A_i)}{\det(A)} \quad i = 1, \dots, n$$

Where is the matrix formed by replacing the *i*-th column of A by the column vector b

$$M = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

$$\beta = \frac{\begin{vmatrix} j & d & g \\ k & e & h \\ l & f & i \end{vmatrix}}{|M|} \qquad \gamma = \frac{\begin{vmatrix} a & j & g \\ b & k & h \\ c & l & i \end{vmatrix}}{|M|} \qquad t = \frac{\begin{vmatrix} a & d & j \\ b & e & k \\ c & f & l \end{vmatrix}}{|M|}$$

$$t = -\frac{f(ak - jb) + e(jc - al) + d(bl - kc)}{a(ei - hf) + b(gf - di) + c(dh - eg)}$$

the others have similar solutions

so, to figure out if the ray has struck the triangle within some interval $[t_0, t_1]$

compute t

if $t < t_0$ or $t > t_1$

return -1

compute γ

if $\gamma < 0$ or $\gamma > 1$

return -1

compute β

if $\beta < 0$ or $\beta > 1$

return -1

else

return t

the assumption being that we would know that -1 meant we didn't hit it

intersection with a sphere

we can break out the implicit form again

the implicit representation of a sphere with center C and radius r is

$(P - C) \cdot (P - C) - r^2 = 0$

if P is on the surface, then P-C is a vector with length r, so the dot product should be r^2

$(E + \mathbf{d}t - C) \cdot (E + \mathbf{d}t - C) - r^2 = 0$

$(\mathbf{d} \cdot \mathbf{d})t^2 + 2\mathbf{d} \cdot (E-C)t + (E-C) \cdot (E-C) - r^2 = 0$

$$t = \frac{-\vec{d} \cdot (E - C) \pm \sqrt{(\vec{d} \cdot (E - C))^2 - (\vec{d} \cdot \vec{d})((E - C) \cdot (E - C) - r^2)}}{(\vec{d} \cdot \vec{d})}$$

the discriminant (the part in the square root) tells us something about the solutions

if it is negative, the result is imaginary (no intersection)

if it is positive, there are two solutions, one where the ray enters, and one where it exits

if it is zero, it just skims the surface, connecting once

we will frequently use the sphere as a bounding surface, in which case we only ever really need to check the discriminant to see if we hit or not

intersecting with a collection of objects

this primarily involves iterating over all of the objects and seeing if we hit them, and if we did if it is closer than any other hits we have made

$t_0 = 0$, $t_1 = \infty$
hit = false
for each object O
  t = find intersection point with O
  if $t_0 <= t <= t_1$
    hit = true
    hitObj = O
    $t_1 = t$
return t_1

Shading

setting the pixel color

we now have a point on a surface (provided the ray hit something, in which case we just use the background color)

so, we are back to familiar territory, we can just do Blinn-Phong shading and there we go…

what will it look like?

ah, well, basically a lot of work to get us back where we were before

we do get hidden faces culling and clipping for “free”, however…

adding shadows

once we have the principle of ray casting, it is pretty easy to add shadows

we create a new ray P + (L-P)t, where L is the position of the light and P is the point

we compute the intersection of this ray with all of our objects and get the t value in return

if t is <1 (and greater than some small offset to get us away from the surface we are checking), something is between the point and the light — it is in shadow

so just paint it with ambient light

adding reflections

another twist we can add is perfect specular reflections (i.e., mirror reflections)

we compute the perfect reflection of the ray and cast that

$\mathbf{r} = \mathbf{d} - 2(\mathbf{d}\cdot\mathbf{n})\mathbf{n}$

Almost the same as the reflection we looked at before

if it goes off to infinity, we return the background color

if it hits a light source, we return the color of the light

if it hits an object

we need to compute the surface color at that point of the object

so cast shadow ray to see if it is shadow

if not, do our normal lighting model to figure out its color

of course if *it* is reflective, then we need to recurse…

this process could recurse forever if it was in an enclosed space, so we will need to set a recursion depth limit on it

the color we compute at the reflected surface is attenuated by whatever the specular color is for the material

transparency and refraction

our surface may also be transparent and allow light to shine through

so, we shoot another ray in the direction of transmission as well (which could include refraction through the surface)

this will go gather color from objects that it strikes in the same way (we basically use one function and just adjust the rays that we send through it…)

diffuse reflections

in theory, we should be able to get rid of ambient lighting terms if we are using real physics of lights, but we don't in ray tracing

why not?

perfect specular highlights make our lives easy

it is a single bounce off in another direction along the perfect reflection vector

the point of diffuse lighting is that it bounces in all directions

this would be impractical to model

for a diffuse surface we would be sending off hundreds of rays trying to pick up color from the surroundings

as a result, ray tracing is best done with transparent and reflective objects

we can create matte objects, but they won't have the effect of bounce light on them

Putting it all together

```
Color(ray r, float t0, float t1, int depth)
      if depth == 0
            return background color
      obj = findHit(r, t0, t1)
      if obj
            p = r.p + r.v*t
            c = obj.ka*Ia
            block = findHit(p + sl, epsilon, infinity)
            if not block
                  c += blinn-phong lighting
            reflect = r.v - 2(r.v•obj.n)obj.n
            c += obj.km * color(p+reflect*s, epsilon, infinity, depth-1)
            if obj.translucent
                  refract = compute refraction ray
                  c += obj.kr * color(p+refract*s, epsilon, infinity, 
depth-1)

            return c
      else
            return background color
```