Raytracing

CS4611 - Foundations of Computer Graphics

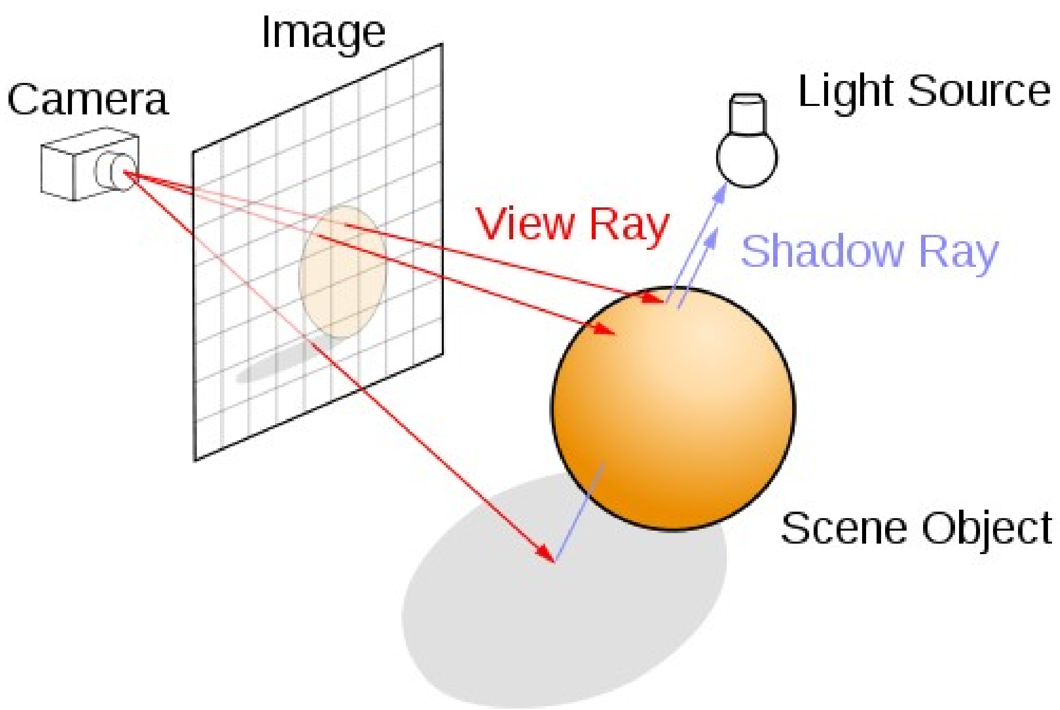
Scott Kuhl
kuhl@mtu.edu
Michigan Tech

Raytracing assignment

- You can work with a partner on this assignment
  - If you work alone, you are still responsible for completing everything!
  - Everybody will be required to complete a short “quiz” on Canvas indicating how you worked together
    - If there is an indication of uneven effort, grading will be handled on a case-by-case basis.
    - Contact me early if you are having issues

Overview

- Shoot a ray into scene for each pixel; see which geometry it intersects
  - Use the closest intersection
  - Check if you can see the light



Assignment expectations

- **View:** Perspective projection
- **Shapes:** Spheres and Triangles
- **Lighting:**
  - Diffuse shading
  - Shadows (one point light source)
  - Reflections

- I will provide specifications for a scene you will need to draw
  - You will need your own test scene too!

What is a ray?

- A ray is composed of a position and a vector
  - You may want to keep vectors normalized
- You will need to make a Ray class (C++) or Ray struct
- Your program will need to:
  - Create rays
  - Determine if a ray intersects any object in the scene

Generating Rays

- You should make a Perspective class (C++) or struct (C).
  - Camera position; distance to screen; screen width in the world; screen width in pixels
  - You should assume the screen is square

- You will need a function such as:
  - Ray GetRay(Perspective p, vec2 screenCoord)
    - Or: void GetRay(Perspective p, vec2 screenCoord, Ray *ray)
    - Or: Make Perspective.GetRay() or similar

getRay()

- How does getRay() work?
  - Vector = normalize((3D position of pixel) –
    (3D position of camera))
  - Position = (3D position of pixel)
- You will need to call getRay() for each pixel on the screen:
  -
```
for(int x=0; x<512; x++)
    for(int y; y=0; y<512; y++)
    {
        getRay(...);
        // calculate and set color of pixel
    }
```

Geometry in the scene

- The geometry in your scene will consist of two lists: triangles and spheres
  - If you are using C++, you could make a “Geometry” class that triangles and spheres inherit from
  - Triangles and spheres will both need an intersect() function which takes a ray and returns a RayHit object.
    - RayHit will consist of the “time” of the intersection; a normal; a flag indicating if object is reflective or diffuse; location of the hit; incoming ray; etc.

Ray intersection: Big picture

- To calculate the color of a pixel we need to:
  - Try to intersect our ray with *every* piece of geometry in the scene.
  - We need to keep track of the nearest “hit”
- Once we identify the nearest “hit”:
  - If reflective, reflect ray off of object and use the color of whatever we hit with the next intersection
  - If diffuse:
    - Check if we are in shadow (if yes, use ambient lighting)
    - Calculate diffuse lighting

Ray-sphere intersection

- Ray equation: $p(t) = \mathbf{e} + t*\mathbf{d}$
  - **e** = starting position of ray
  - **d** = vector representing ray
  - t = “time”
- Implicit sphere equation: $f(p(t))=0$
  - f is a function that returns 0 if the point is on the sphere
- Together: $f(\mathbf{e} + t*\mathbf{d})=0$

Sphere equations

- Center of sphere: **c**=(xc, yc, zc)
- Radius of sphere = R
- Implicit equation to check if **p**=(x,y,z) on a sphere?
  - Any ideas?

$$\sqrt{(x-xc)^2+(y-yc)^2+(z-zc)^2}=R$$

$$(x-xc)^2+(y-yc)^2+(z-zc)^2=R^2$$

$$(x-xc)^2+(y-yc)^2+(z-zc)^2-R^2=0$$

$$(p-c)\cdot(p-c)-R^2=0$$

Ray-sphere intersection

$$(p-c)\cdot(p-c)-R^2=0$$
$$(e+td-c)\cdot(e+td-c)-R^2=0$$

- Now, we need to solve for t!
- First, we can do some rearrangement:

$$(d\cdot d)t^2+2\mathrm{d}\cdot(e-c)t+(e-c)\cdot(e-c)-R^2=0$$

- How can we solve for t?

Quadratic equation!

- A nice, simple equation...

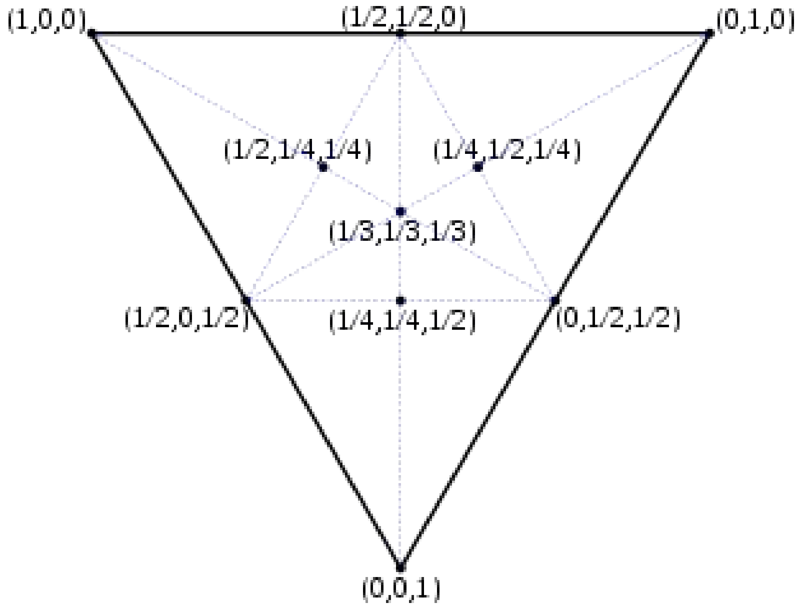
$$t=\frac{-d\cdot(e-c)\pm\sqrt{((d\cdot(e-c))^2-(d\cdot d)((e-c)\cdot(e-c)-R^2))}}{d\cdot d}$$

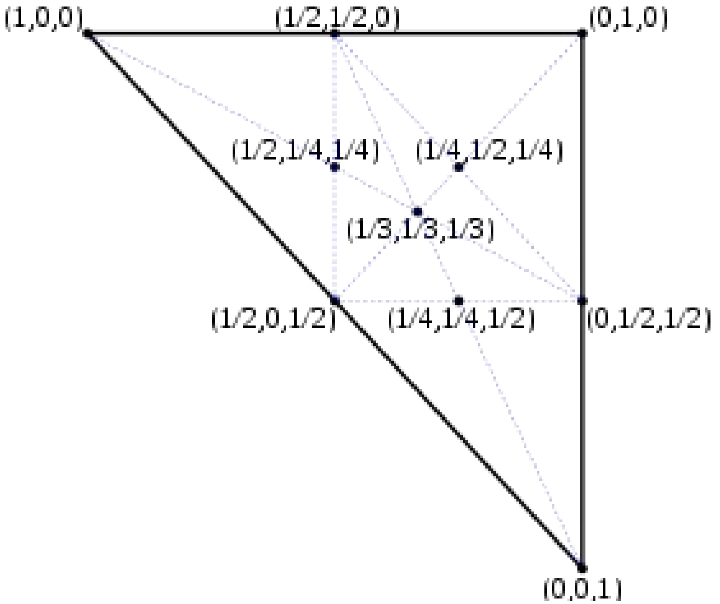
How many solutions will the equation return?

- If the discriminant (under the square root) is...
  - < 0 there is no solution
  - = 0 there is one solution
  - > 0 multiple solutions

- Is denominator ever zero?

Barycentric coordinates





Equation for a triangle

- A weighted average of three vertices a, b, and c using Barycentric coordinates:
  - P = (1 – beta – gamma)a + (beta)b + (gamma)c
  - P = a + (beta)(b-a) + (gamma)(c-a)

- 3rd Barycentric coordinate is not needed since all three components must sum to 1
- We are on triangle if:
  - Beta > 0
  - Gamma > 0
  - Beta+Gamma < 1

Ray-triangle intersection

- Can define a series of parametric equations.
- In these equations, t is unknown
- We also don't know beta & gamma---changing them moves us on the triangle.

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

- Three equations, three unknowns...

Solving equations

- Details of solving these equations is outside of the scope of this class.
  - Involves creating a linear system and using Cramer's Rule

Ray-triangle intersection

$$A = x_a - x_b$$
$$B = y_a - y_b$$
$$C = z_a - z_b$$
$$D = x_a - x_c$$
$$E = y_a - y_c$$
$$F = z_a - z_c$$
$$G = x_d$$
$$H = y_d$$
$$I = z_d$$
$$J = x_a - x_e$$
$$K = y_a - y_e$$
$$L = z_a - z_e$$

$$M = A(EI - HF) + B(GF - DI) + C(DH - EG)$$

$$\beta = \frac{J(EI - HF) + K(GF - DI) + L(DH - EG)}{M}$$

$$\gamma = \frac{I(AK - JB) + H(JC - AL) + G(BL - KC)}{M}$$

$$t = \frac{-(F(AK - JB) + E(JC - AL) + D(BL - KC))}{M}$$

a,b,c subscripts refer to vertices of the triangle.
e subscript refers to the starting position of the ray.
d subscript refers to the components of the vector for the ray

Pseudocode

- Compute t
- If(t<0 or larger than closest hit so far)

```
    return no hit
calculate gamma
if( gamma < 0 or gamma > 1)
    return no hit
calculate beta
if( beta < 0 or beta > 1-gamma)
    return no hit
return hit
```

Green part can be skipped if you want to find the first intersection elsewhere in your code.

Diffuse shading

- Diffuse shading is calculated in raytracing the same way as we did it in OpenGL
  - Dot product of the normal and the direction to the light

Reflecting a ray

- Given d and n (see diagram), we can calculate the reflected ray r:
- Any ideas?

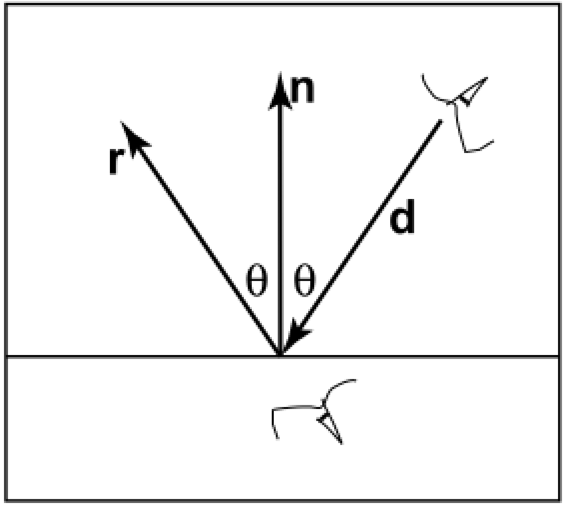


Figure 10.8: When looking into a perfect mirror, the viewer looking in direction **d** will see whatever the viewer "below" the surface would see in direction **r**.

$$r = d - 2(d \cdot n)\, n$$

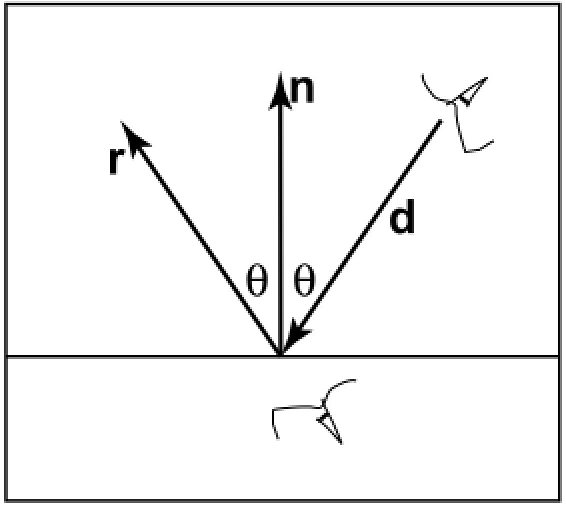


Figure 10.8: When looking into a perfect mirror, the viewer looking in direction **d** will see whatever the viewer "below" the surface would see in direction **r**.

Recursion limit

- You should prevent infinite recursion using two methods:
  - If you don't hit an object, pretend you hit a black object
  - If you recursively shoot more than 10 rays, return black

Debugging

- Debugging a raytracer is easier than OpenGL
  - To test intersection code, set up a simple scene with a single ray that you can predict exactly where the intersection would occur

Dividing work

- Decide if you are going to use C++ or C.
- Define the classes/structs that you are going to make
- Ways to potentially divide work:
  - Reflective light vs diffuse light
  - Spheres vs triangles
  - Generating rays from the camera
  - Creating a scene
    - I will also provide a simple test scene