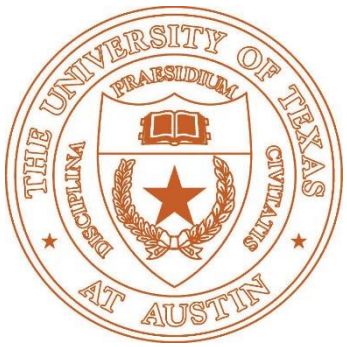
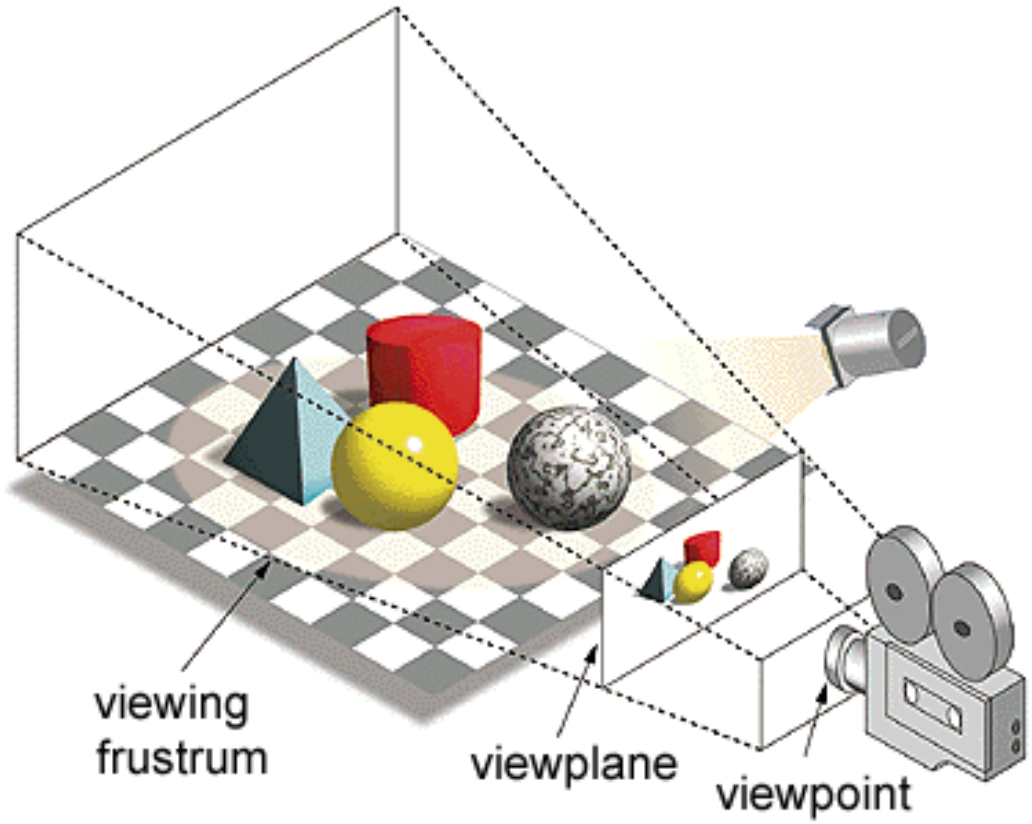
CS354 Computer Graphics
Ray Tracing

Qixing Huang

Januray 24th 2017



Graphics Pipeline



Elements of rendering

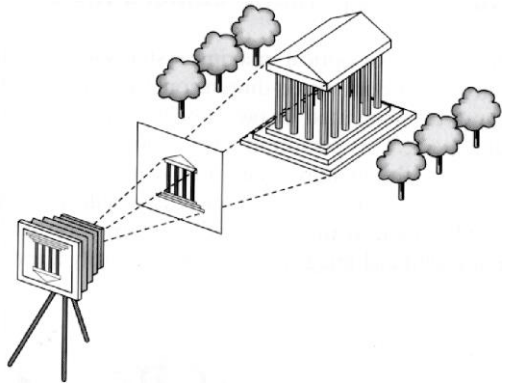
- Object
- Light
- Material
- Camera

Geometric optics

- Modern theories of light treat it as both a wave and a particle
- We will take a combined and somewhat simpler view of light – the view of geometric optics
- Here are the rules of geometric optics:
  - Light is a flow of photons with wavelengths. We'll call these flows "light rays"
  - Light rays travel in straight lines in free space
  - Light rays do not interfere with each other as they cross
  - Light rays obey the laws of reflection and refraction
  - Light rays travel from the light sources to the eye, but the physics is invariant under path reversal

Synthetic pinhole camera

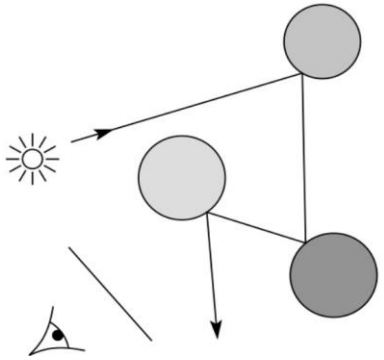
- The most common imaging model in graphics is the synthetic pinhole camera: light rays are collected through an infinitesimally small hole and recorded on an image plane



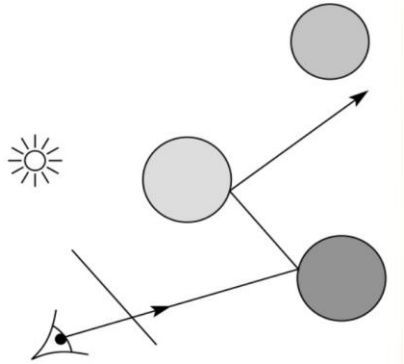
- For convenience, the image plane is usually placed in front of the camera, giving a non-inverted 2D projection (image)
- The image of an object point *P* is at the intersection of the viewing ray through *P* and the image plane

Eye vs. light ray tracing

- At the light: light ray tracing (a.k.a., forward ray tracing or photon tracing)



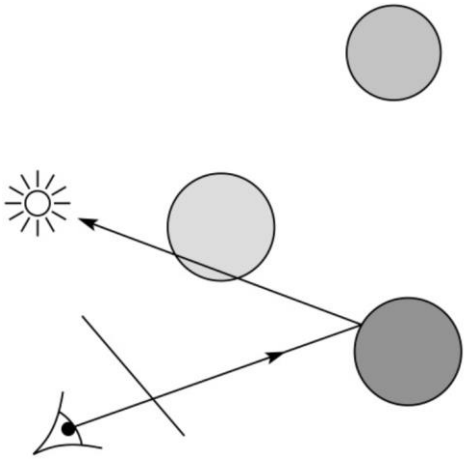
- At the eye: eye tracing (a.k.a., backward ray tracing)



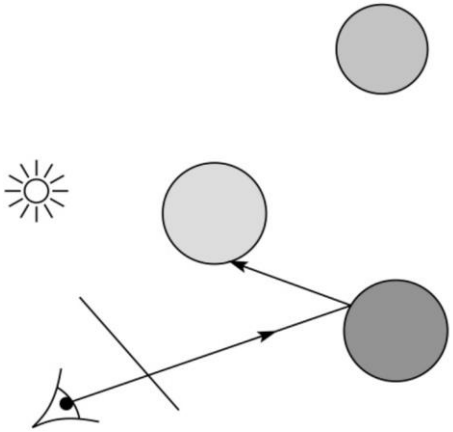
- We will generally follow rays from the eye into the scene

Precursors to ray tracing

- Local illumination
  - Cast one eye ray
    - then shade according to light

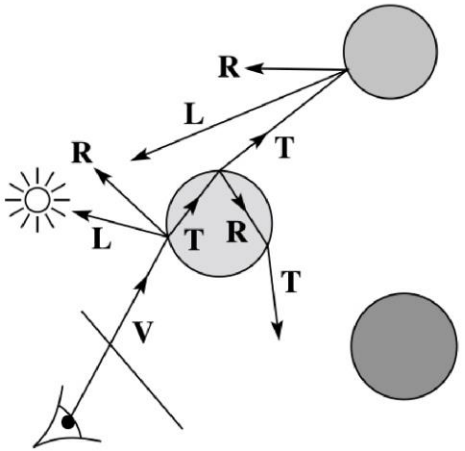


- Appel (1968)
  - Cast one eye ray + one ray to light



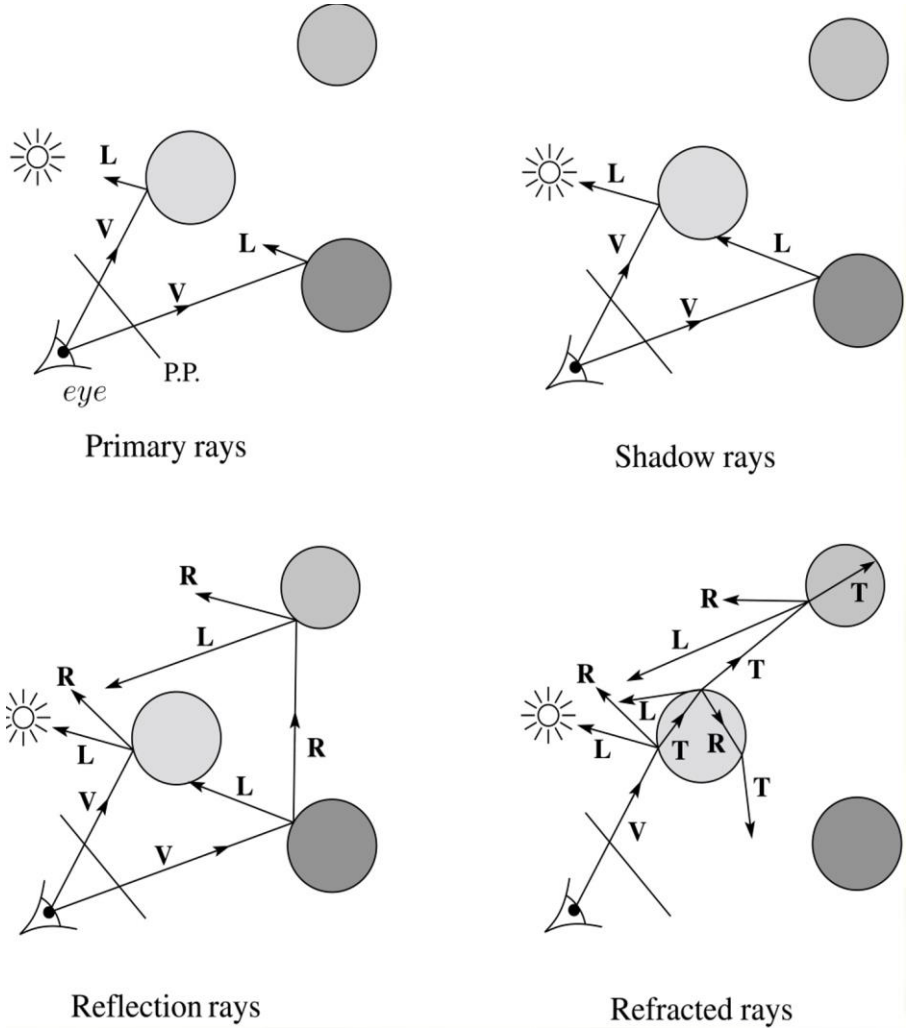
Whitted ray-tracing algorithm

- In 1980, Turner Whitted introduced ray tracing to the graphics community
  - Combines eye ray tracing + rays to light
  - Recursively traces rays

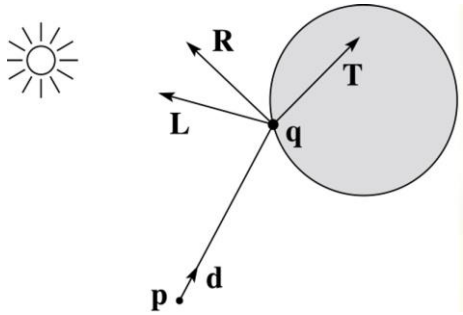


- Algorithm
  - For each pixel, trace a primary ray in direction V to the first visible surface
  - For each intersection, trace secondary rays:
    - Shadow rays in directions L to light sources
    - Reflected ray in direction R
    - Refracted ray or transmitted ray in direction T

Whitted algorithm



Shading



- A ray is defined by an origin P and a unit direction d and is parameterized by t:

$$P + t\mathbf{d}$$

- Let I(P, d) be the intensity seen along that ray. Then:

$$I(P,d) = I_{direct} + I_{reflected} + I_{transmitted}$$

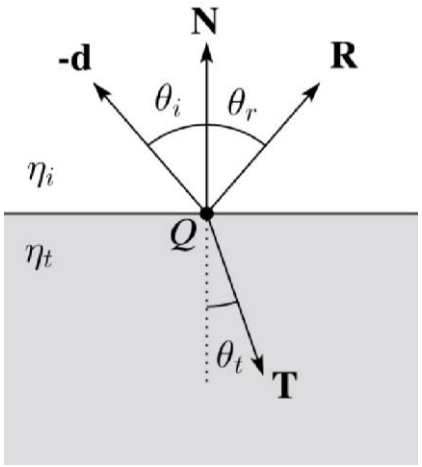
- Where
  - I_{direct} is computed from the Phong model

$$I_{reflected} = k_r I(Q,R)$$

$$I_{transmitted} = k_t I(Q,T)$$

- Typically, we set $k_r = k_s$ and $k_t = 1 - k_s$

Reflection and transmission



- Law of reflection:

$$\theta_i = \theta_r$$

- Snell's law of refraction

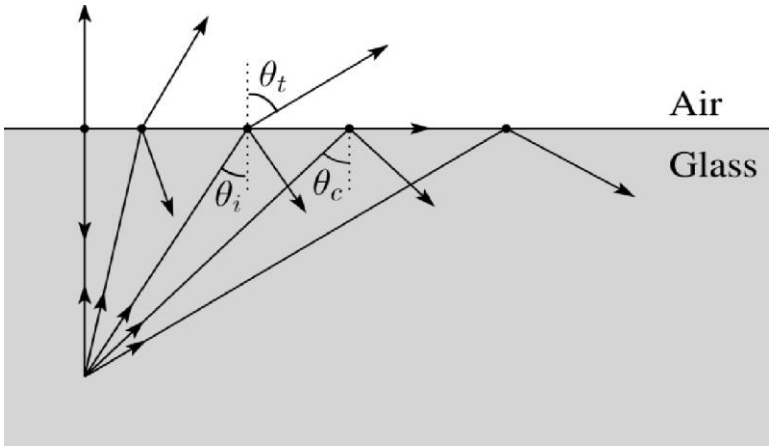
$$\eta_i \sin\theta_I = \eta_t \sin\theta_t$$

- Where η_i , η_t are **indices of refraction**

Total internal reflection

- The equation for the angle of refraction can be computed from Snell's law:

- What happens when $\eta_i > \eta_t$?
- When θ_t is exactly $90°$, we say that θ_I has achieved the "critical angle" θ_c
- For $\theta_I > \theta_c$, no rays are transmitted, and only reflection occurs, a phenomenon known as "total internal reflection" or TIR



Ray-tracing pseudocode

- We build a ray traced image by casting rays through each of the pixels

```
function traceImage (scene):
   for each pixel (i,j) in image
      S = pixelToWorld(i,j)
      P = COP
      d = (S - P)/|| S – P||
      I(i,j) = traceRay(scene, P, d)
   end for
end function
```

Ray-tracing pseudocode, cont'd

```
function traceRay(scene, P, d):
    (t, N, mtrl) ← scene.intersect (P, d)
    Q ← ray (P, d) evaluated at t
    I = shade(q, N, mtrl, scene)
    R = reflectDirection(N, -d)
    I ← I + mtrl.k_r * traceRay(scene, Q, R)
    if ray is entering object then
      n_i = index_of_air
      n_t = mtrl.index
    else
      n_i = mtrl.index
      n_t = index_of_air
    if (mtrl.k_t > 0 and notTIR (n_i, n_t, N, -d)) then
      T = refractDirection (n_i, n_t, N, -d)
      I ← I + mtrl.k_t * traceRay(scene, Q, T)
    end if
    return I
end function
```

Terminating recursion

- Q: How do you terminate out of recursive ray tracing?

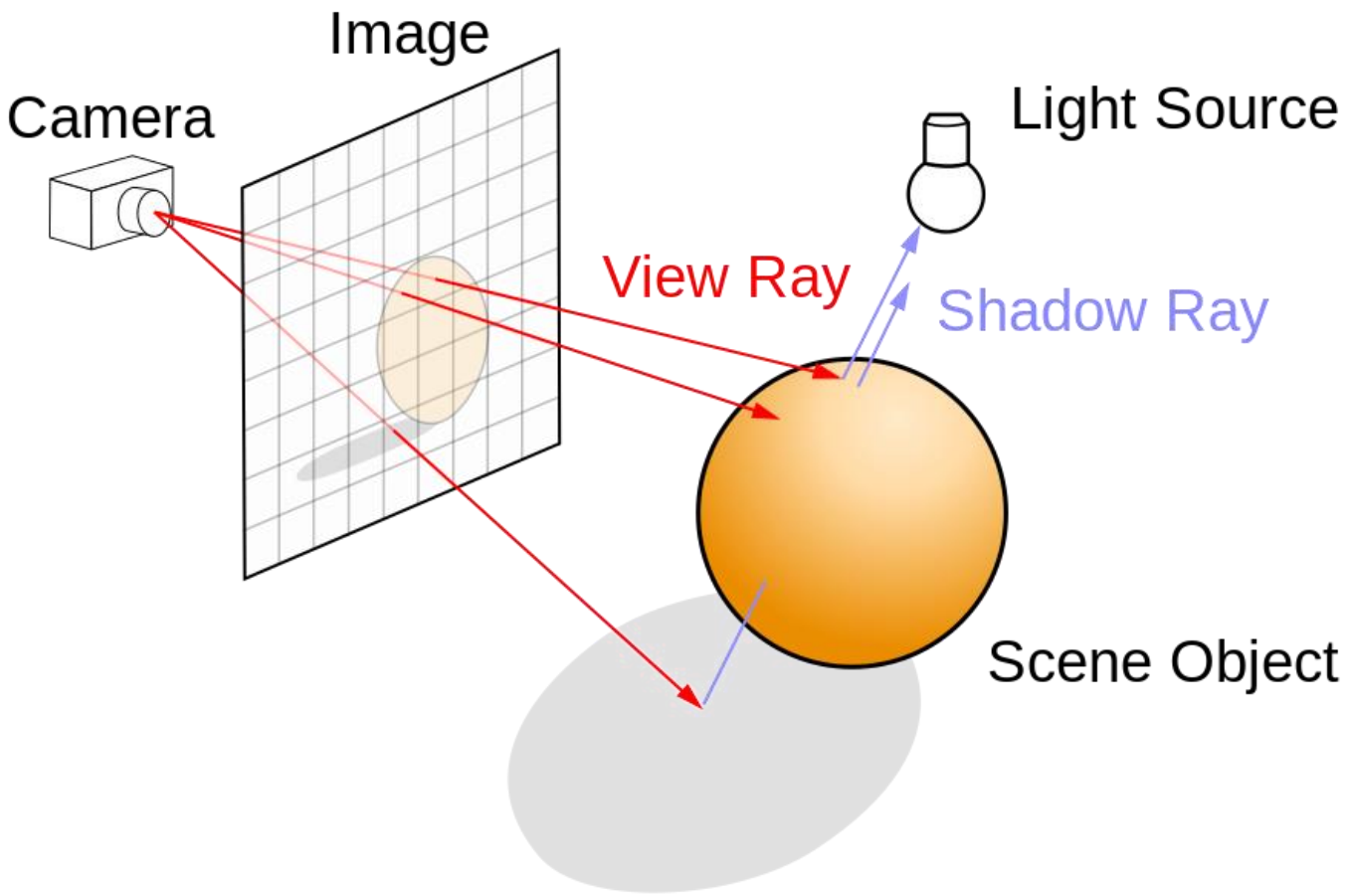
Shading pseudocode

- Next, we need to calculate the color returned by the *shade* function

```
function shade(mtrl, scene, Q, N, d):
   I ← mtrl.k_e + mtrl. k_a * scene->I_a
   for each light source [?] do:
      atten = [?] -> distanceAttenuation( Q ) *
      [?] -> shadowAttenuation( scene, Q )
      I ← I + atten*(diffuse term + spec term)
   end for
   return I
end function
```

This will be discussed in depth next Monday

Eye vs. light ray tracing



Shadow attenuation

- Computing a shadow can be as simple as checking to see if a ray makes it to the light source
- For a point light source:

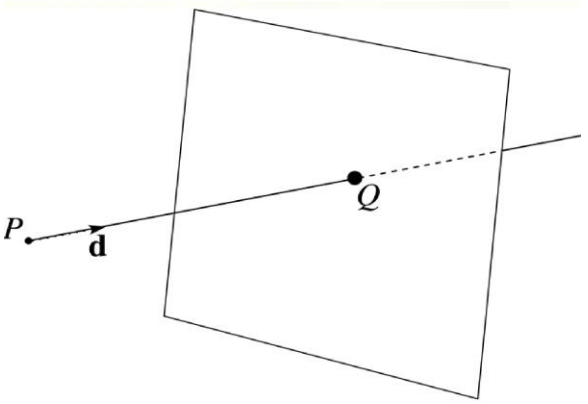
```
function PointLight::shadowAttenuation(scene, P)
    d = (?.position - P).normalize()
    (t, N, mtrl) ← scene.intersect(P, d)
    Q ← ray(t)
    if Q is before the light source then:
      atten = 0
    else
      atten = 1
    end if
    return atten
end function
```

- Q: What if there are transparent objects along a path to the light source?

Ray-plane intersection

- We can write down the plane equation

$$a \cdot x + b \cdot y + c \cdot z + d = 0$$
$$n = [a; b; c]$$
$$n^T \cdot [x; y; z] + d = 0$$



- Using parameterized line segment

$$n^T \cdot (p + td) = 0$$

- We can solve for the intersection parameter

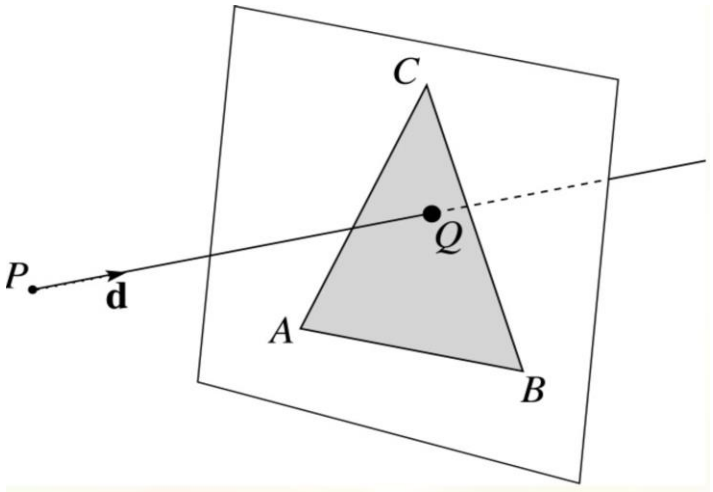
$$t = -\frac{n^T p}{n^T d}$$

Ray-triangle intersection

- To intersect with a triangle, we first obtain its plane equation

$$n = (A - C) \times (B - C)$$

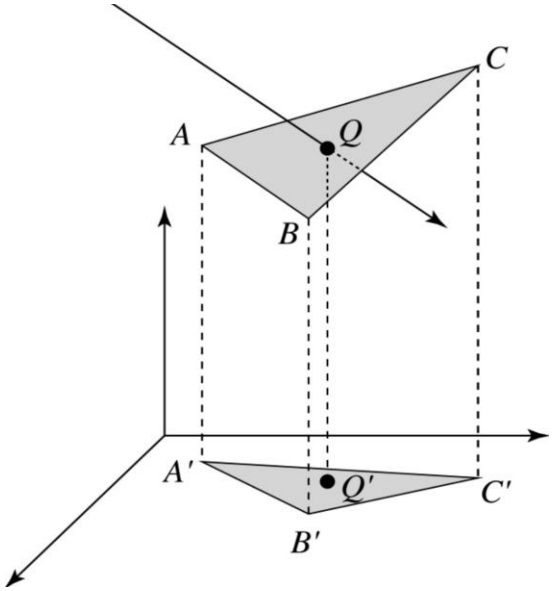
$$d = n^T A$$



- Then, we need to decide if the point is inside or outside of the triangle

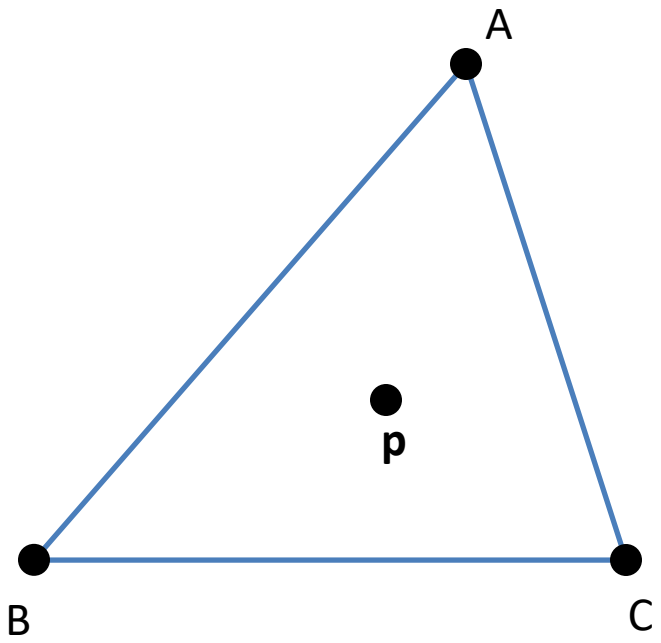
Ray-triangle intersection

- Project down a dimension and compute barycentric coordinates from 2D points



- Why is this solution possible? Which axis should you “project away”?

Barycentric coordinate from area ratios

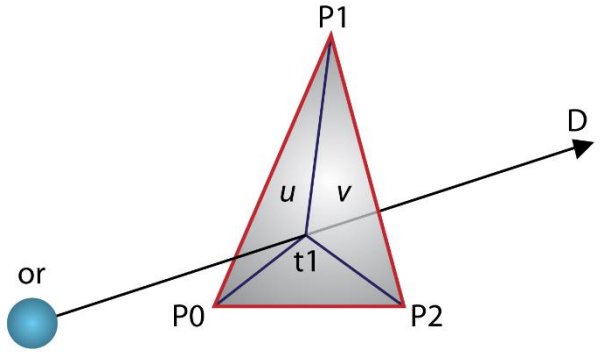


$$\alpha = \frac{\mathrm{SArea}(\mathbf{p}BC)}{\mathrm{SArea}(ABC)} \quad \beta = \frac{\mathrm{SArea}(A\mathbf{p}C)}{\mathrm{SArea}(ABC)} \quad \gamma = \frac{\mathrm{SArea}(AB\mathbf{p})}{\mathrm{SArea}(ABC)}$$

Möller–Trumbore intersection algorithm

- Parametric Plane

$$P = A + u \cdot (B - A) + v \cdot (C - A)$$
$$= (1 - u - v) \cdot A + u \cdot B + v \cdot C$$



- Parametric line segment

$$P = P_0 + t \cdot D$$

- Linear system

$$P_0 + t \cdot D = A + u \cdot (B - A) + v \cdot (C - A)$$

$$u \cdot (A - B) + v \cdot (A - C) + t \cdot D = A - P_0$$

Möller–Trumbore intersection algorithm

- Linear system

$$P_0 + t \cdot D = A + u \cdot (B - A) + v \cdot (C - A)$$

$$u \cdot (A - B) + v \cdot (A - C) + t \cdot D = A - P_0$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

$$x = \frac{\det \begin{pmatrix} b_1 & b_2 & b_3 \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}}{\det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}}, \quad y = \frac{\det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ b_1 & b_2 & b_3 \\ a_{31} & a_{32} & a_{33} \end{pmatrix}}{\det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}}, \quad z = \frac{\det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ b_1 & b_2 & b_3 \end{pmatrix}}{\det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}}$$

https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm

```
bool RayIntersectsTriangle(Vector3D rayOrigin,
                           Vector3D rayVector,
                           Triangle* inTriangle,
                           Vector3D& outIntersectionPoint)
{
    const float EPSILON = 0.0000001;
    Vector3D vertex0 = inTriangle->vertex0;
    Vector3D vertex1 = inTriangle->vertex1;
    Vector3D vertex2 = inTriangle->vertex2;
    Vector3D edge1, edge2, h, s, q;
    float a,f,u,v;
    edge1 = vertex1 - vertex0;
    edge2 = vertex2 - vertex0;
    h = rayVector.crossProduct(edge2);
    a = edge1.dotProduct(h);
    if (a > -EPSILON && a < EPSILON)
        return false;
    f = 1/a;
    s = rayOrigin - vertex0;
    u = f * (s.dotProduct(h));
    if (u < 0.0 || u > 1.0)
        return false;
    q = s.crossProduct(edge1);
    v = f * rayVector.dotProduct(q);
    if (v < 0.0 || u + v > 1.0)
        return false;
    // At this stage we can compute t to find out where the intersection point is on the line.
    float t = f * edge2.dotProduct(q);
    if (t > EPSILON) // ray intersection
    {
        outIntersectionPoint = rayOrigin + rayVector * t;
        return true;
    }
    else // This means that there is a line intersection but not a ray intersection.
        return false;
}
```

Interpolating vertex properties

- The barycentric coordinates can also be used to interpolate vertex properties
  - material/texture coordinates/Type equation here.normals
- For example

$$k_d(Q) = \alpha k_d(A) + \beta k_d(B) + \gamma k_d(C), \quad \alpha + \beta + \gamma = 1$$

- Interpolating normal directions, known as Phong interpolation, gives triangle meshes a smooth shading appearance. (Note: don't forget to normalize interpolated normal directions)

Epsilons

- Due to finite precision arithmetic, we do not always get the exact intersection at a surface
- Q: What kinds of problems might this cause?

- Q: How to address this?

Intersecting with transformed geometry

- In general, objects will be placed using transformations. What if the object being intersected were transformed by a matrix M?

- Apply *the inverse of M* to the ray first and intersect in object (local) coordinates

- Q: The intersected normal is in object (local) coordinates. How do we transform it to world coordinates?

Next Lecture

- More about ray tracing, math, and transforms

- Special thanks for Don Fussell for many of the slides

Reading

- Whitted. An improved illumination model for shaded display. Communications of the ACM 23(6), 343-349, 1980

- https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Questions?