Johns Hopkins
Engineering for Professionals

605.767 Applied Computer Graphics

Brian Russin

Module 1D
Intersection Methods

Implicit Surfaces vs. Explicit Surfaces

- Implicit surface is of form $f(p) = \ldots = 0$
  - A point on the surface satisfies an equation of the form
    $f(p) = f(p_x, p_y, p_z) = 0$
  - Example is the equation of a sphere
    $$p_x^2 + p_y^2 + p_z^2 = r^2 \qquad f(p) = p_x^2 + p_y^2 + p_z^2 - r^2 = 0$$
  - Explicit surface is defined by a vector function f and some parameters
    - Rather than a point on the surface

$$p = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = f(\rho, \varphi) = \begin{bmatrix} f_x(\rho, \varphi) \\ f_y(\rho, \varphi) \\ f_z(\rho, \varphi) \end{bmatrix}$$

$$f(\rho, \varphi) = \begin{bmatrix} r\sin(\rho)\cos(\varphi) \\ r\sin(\rho)\sin(\varphi) \\ r\cos(\varphi) \end{bmatrix}$$

Explicit Representation of a Sphere

$$t(a, b) = (1 - a - b)v_0 + av_1 + bv_2$$

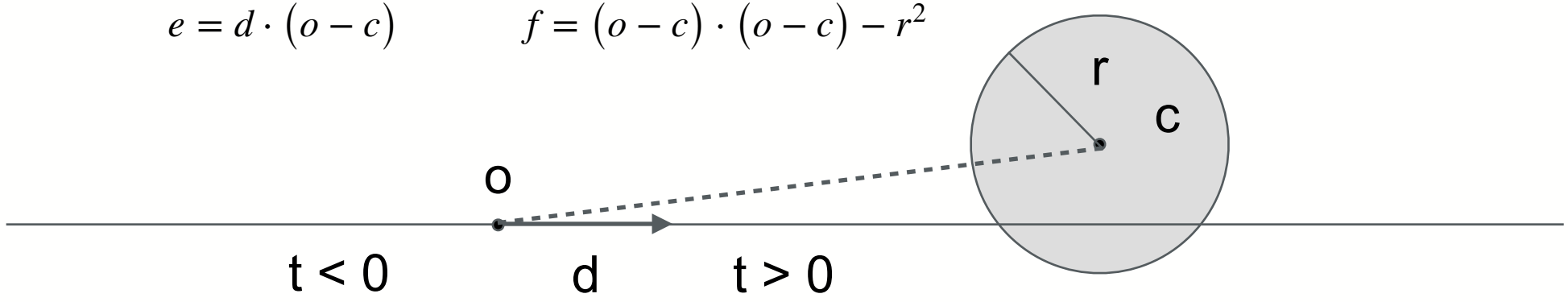
Explicit Representation of a Triangle

Ray / Sphere Intersection

- Intersection between a ray and a sphere is efficiently calculated
- Haines provides a mathematical derivation
  - Sphere at center c with radius r is defined by:
    - $(c_x - l)^2 + (c_y - m)^2 + (c_z - n)^2 = r^2$
  - Substituting for x,y,z using the parametric ray equations gives a quadratic equation in t
    - If d is unit length the equation is simplified:

$$t^2 + 2te + f = 0$$

$$e = d \cdot (o - c) \qquad f = (o - c) \cdot (o - c) - r^2$$



Ray / Sphere Intersect (cont.)

- Solved using the quadratic formula $t = -e \pm \sqrt{e^2 - f}$
  - Discriminant is $e^2 - f$
  - Discriminant < 0
    - No intersect (no real roots of the quadratic)
    - Can avoid sqrt and any further calculations
  - Discriminant = 0
    - Intersect at a tangent point (1 root)
    - t = -e
  - Discriminant > 0
    - 2 intersections
    - Roots of the quadratic give the front and back intersections
    - Substitute the smallest (positive) value of t back into the ray parametric equations to get the nearest intersect coordinate
  - t < 0 are behind the ray start point and are not relevant

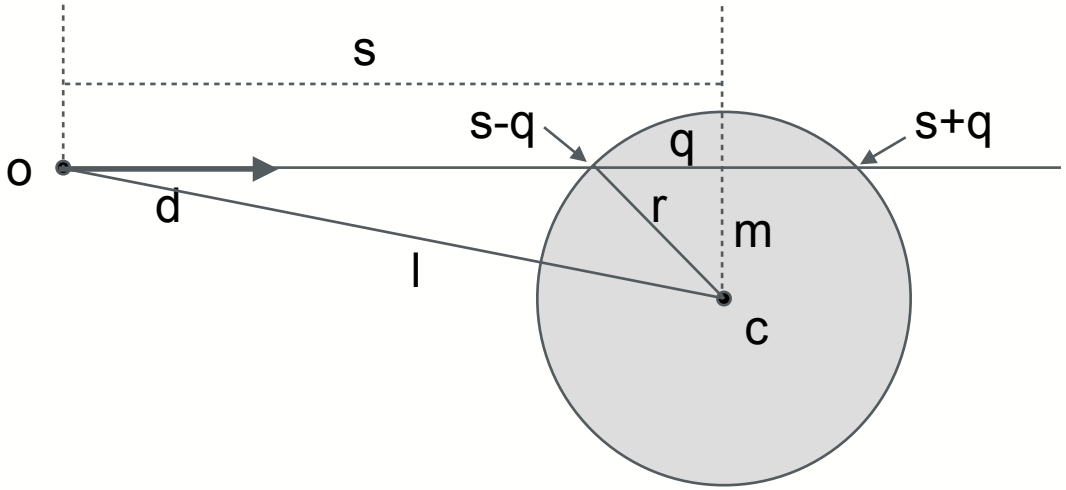
Ray / Sphere Intersect (cont.)

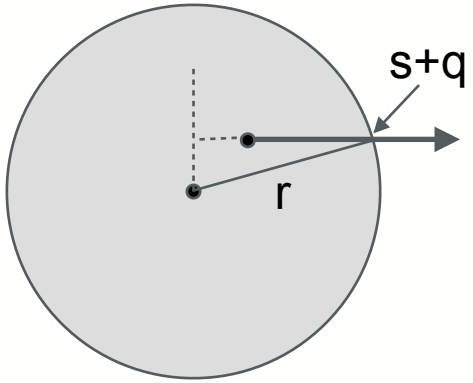
- Eberly presents an optimized method for determining if an intersection occurs

```
/**
 * Test if the ray intersects a sphere. Uses optimized method by David Ebert.
 */
bool DoesIntersect(const BoundingSphere& sphere) const
{
   // quadratic is t^2 + 2et + f = 0

   // Construct vector from sphere center to ray origin.
   Vector3 q = o - sphere.center;
   float f = q.Dot(q) - sphere.radius * sphere.radius;
   if (f <= 0.0f)
      return true;      // Ray origin is inside the sphere

   float e = d.Dot(q);
   if (e >= 0.0f)
      return false;     // Sphere center is behind ray origin

   // Quadratic has real roots if discriminant is non-negative
   return (e * e >= f);
}
```

Ray-Sphere Intersect Using Vectors

```
RaySphereIntersect(o,d,c,r)   // Ray origin and direction (unit length), sphere center and radius)
   l=c-o                      // Vector from ray origin to sphere center
   s=l Dot d                  // Projection of l onto ray
   l2 = l Dot l               // Squared length of l
   if (s < 0 && l2 > r2)      // Sphere center is behind ray origin AND ray origin is outside sphere
         return false
   m2 =  l2 – s*s
   if (m2 > r2)               // Distance of closest point along the ray (perpendicular)
         return false         // ray passes outside the sphere
   q = sqrt(r2 – m2)          // By Pythagorean theorem
   if (l2 > r2)               // Ray origin is outside sphere: nearest intersection is at value t=s-q
        t = s-q
   else                       // Ray origin is inside sphere
       t = s+q
```





Ray / Plane Intersection

- Plane equation $n_p \cdot x = d_p$
  - Same as ax + by + cz - d_p = 0
    - a,b,c are components of the normal
    - d_p is found by substituting a vertex of the polygon as x,y,z
- Intersect with parametrically defined ray
  - Substitute ray into plane equation as x

$$n_p \cdot (o + td) - d_p = 0 \qquad t = \frac{d_p - (n_p \cdot o)}{n_p \cdot d} = \frac{-(ao_x + bo_y + co_z - d_p)}{n_p \cdot d}$$

- No intersect if t < 0
  - Plane is “behind” the ray origin
- If denominator == 0 the ray is parallel to the plane
  - No intersect occurs
  - Usually best to us: abs(denominator) < ε, to account for precision errors

Ray / Tapered Cylinder Intersections

- Solutions are discussed in F. S. Hill and Ericson
- Tapered cylinder with 3 surfaces:
  - Cylinder side
    - $F(x,y,z)=x^2+y^2-(1+(s-1)z)^2$ for $0<z<1$
  - Bottom circle with radius = 1 at z = 0
  - Top circle with radius s at z=1
    - S = 0 creates a cone, s = 1 a cylinder
- Intersection with cylinder wall
  - Solution to quadratic equation by substituting ray equation into implicit form above

    $A = d_x^2 + d_y^2 - m^2 \qquad B = o_x d_x + o_y d_y - mn \qquad C = o_x^2 + o_y^2 - n^2$

    - Where $m = (s-1)d_z$ and $n = 1 + (s-1)o_z$
- Intersection with cylinder caps:
  - Interest ray with plane z = 0 and see if $x^2+y^2 < 1$ for the cap
  - Interest ray with plane z = 1 and see if $x^2+y^2 < s^2$ for the cap

Ray / Quadric Intersections

- General implicit equation of a quadric is:
  - $Ax^2 + Ey^2 + Hz^2 + 2Bxy + 2\,Fyz + 2\,Cxz + 2Dx + 2\,Gy + 2Iz + J = 0$
- Watt gives the following quadric equations:
  - Ellipsoid
    - $(x - l)^2 / a^2 + (y - m)^2 / b^2 + (z - n)^2 / c^2 - 1 = 0$
      - a, b, c are the semi-axes lengths
  - Paraboloid
    - $(x - l)^2 / a^2 + (y - m)^2 / b^2 - z + n = 0$
  - Hyperboloid
    - $(x - l)^2 / a^2 + (y - m)^2 / b^2 + (z - n)^2 / c^2 + n = 0$
- Mathematical solutions: similar methods as used for sphere
  - Solve by substitution

Ray / Box Intersection

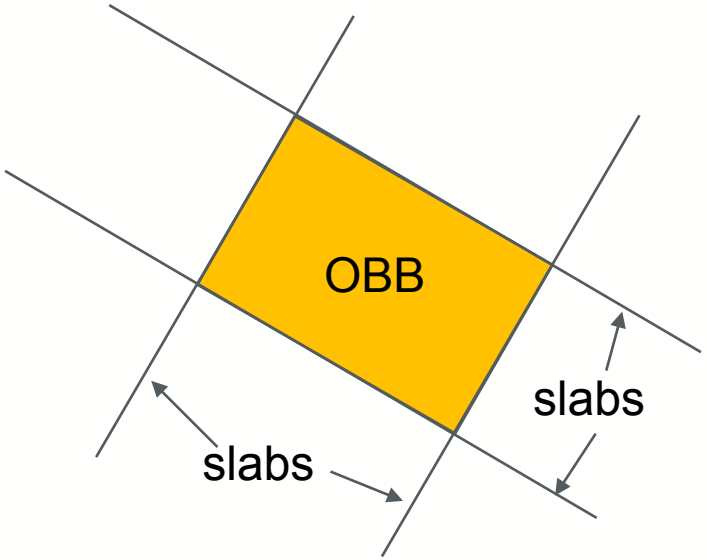
- Slabs method
  - Works for both AABB and OBB cases
  - Presented by Haines
    - Based on slabs method by Kay and Kajiya
    - Inspired by Cyrus-Beck line clipping algorithm
- Woo's method
  - Simplification of the slabs method
  - Only handles AABB
  - May be faster on some architectures
- Separating axis method – discussed in text
- Method by Schroeder
  - Treat the ray as a line segment and clip it against the bounding box
- Eismann method (2007)
  - Claimed to be faster than other methods
  - https://www.tandfonline.com/doi/abs/10.1080/2151237X.2007.10129248

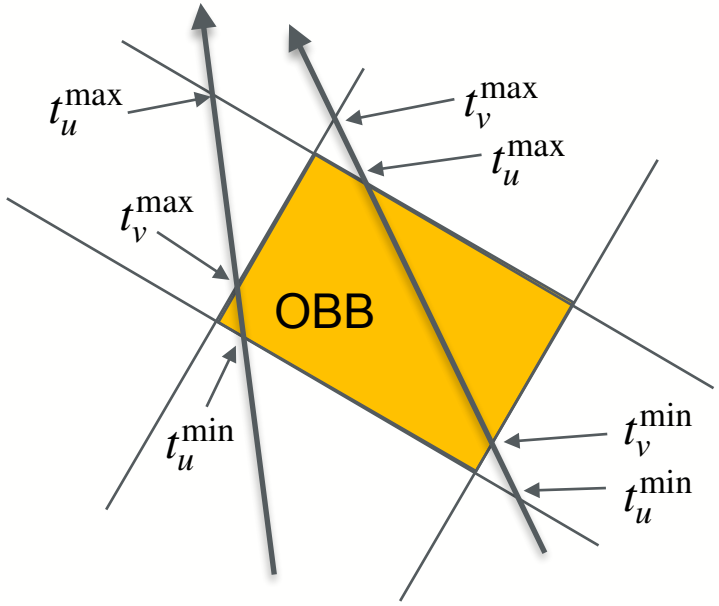
Ray / Box Intersection – Slabs Method

- Box is considered to be a set of 3 slabs
  - **A slab is two parallel planes grouped together for faster computation**
- For each slab there is a minimum and maximum t value
  - For intersection of the ray with the planes of the slab
  - Determine t^{min} and t^{max}

$$t^{\min} = \max\left(t_u^{\min}, t_v^{\min}, t_w^{\min}\right) \qquad t^{\max} = \min\left(t_u^{\max}, t_v^{\max}, t_w^{\max}\right)$$

- If $t^{min} <= t^{max}$ an intersect occurs, otherwise ray does not intersect the box



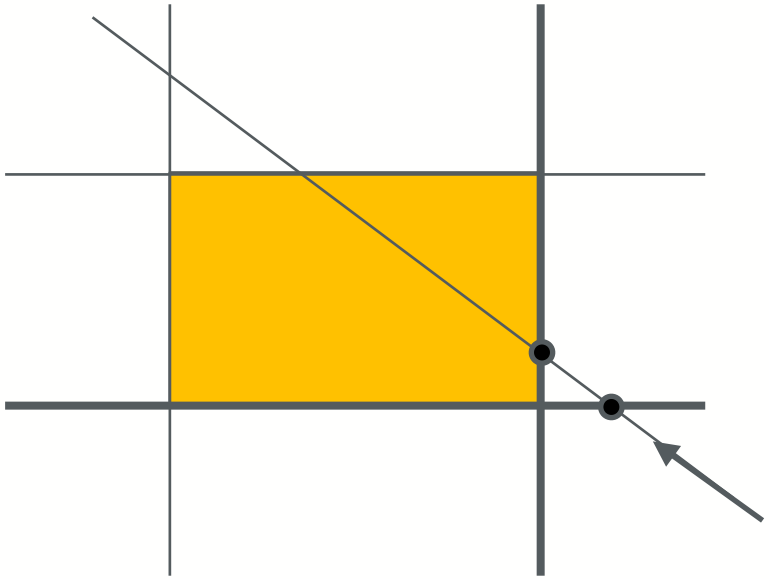


Ray / OBB Intersection

```
RayOBBIntersect(o,d,A)                      // Ray origin and direction (unit length), OBB
    tmin = - MAXVALUE
    tmax = MAXVALUE
    p=aCenter-o                             // Vector from ray origin to box center
    for each axis of OBB (u,v,w)
        e =ai Dot p                         // OBB axis dot product with vector p
        f = ai DOT d                        // OBB axis dot product with ray direction
        if (abs(f) > epsilon)               // Test that ray is not parallel to OBB axis
            t1=(e+hi)/f
            t2=(e-hi)/f
            if (t1>t2) swap(t1,t2)          // Ensure minimum of t1 and t2 is stored in t1
                                               // and max is stored in t2
            if (t1 > tmin) tmin = t1        // Test if this creates new tmin
            if (t2 < tmax) tmax = t2        // Test if this creates new tmax
            if (tmin > tmax) return REJECT  // Early exit – no intersect!
            if (tmax < 0) return REJECT     // Box is behind ray origin!
        else if (-e-hi >0 or –e+hi<0)       // Ray is parallel to plane and ray is outside the OBB
            return REJECT
    if (tmin >0) return INTERSECT, tmin     // Intersect occurs – choose closest positive value of t
    else return INTERSECT tmax
```

Ray / AABB Intersect

- Can simplify the slabs method when intersecting an AABB
  - Calculation of t for each pair of planes is simplified
    - $e = p_i$
    - $f = d_i$
- Woo's Method
  - Calculate t values only for front-facing planes
    - Intersection distance between ray and plane
      - Omit back-facing planes from consideration
  - Largest of these distances corresponds to potential hit
    - If a potential hit is found the actual intersect point is found
      - Bounds testing to see if it is on corresponding face of the AABB



Candidate Planes are Front-Facing (bold)

Ray / Triangle Intersection

- Majority of methods are of the form:
  - Find the intersection between ray and triangle's plane
  - Project triangle vertices and intersect point to xy, xz, or yz plane
    - Maximize area of the projected triangle
  - Decide whether 2D point is within 2D triangle
  - Similar to ray/polygon intersection described later
- These methods rely on the normal to the triangle
  - Required for ray/plane intersection
  - Haines and Moller present a method that does not require pre-computed normals
    - Can save memory in a triangle mesh
      - e.g., if mesh only contains vertex normals and not face normals

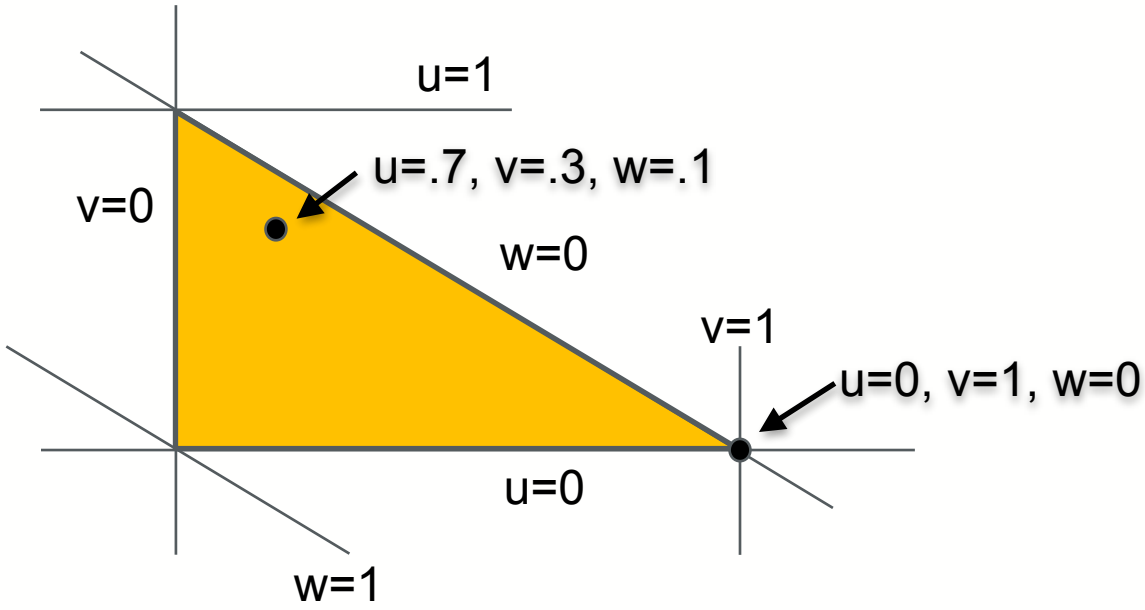
Triangle Represented in **Barycentric** Coordinates

- Point t on the triangle is given by:

$$t(u,v) = (1-u-v)v_0 + uv_1 + vv_2$$

$$u \geq 0, v \geq 0, (u+v) \leq 1$$

  - u,v are **barycentric coordinates**
  - u,v are weights that each vertex contributes to a location
    - w =(1-u-v)



Ray / Triangle Intersection in Barycentric Coordinates

- Intersection of ray r(t) and triangle t(u,v):

$$o + td = (1 - u - v)v_0 + uv_1 + vv_2$$

- Barycentric coordinates (u,v) and distance (t) from ray origin can be solved from the linear system of equations (rearranging terms above):

$$\begin{bmatrix} -d & v_1 - v_0 & v_2 - v_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = o - v_0$$

- Can be thought of as translating the triangle to the origin, transforming it to unit triangle with the ray direction aligned with x
  - See Figure 22.14 (16.10 3rd Edition)

Solving the Linear System

- Can obtain solution via Cramer's rule
  - $e_1=v_1-v_0$, $e_2=v_2-v_0$, $s=o-v_0$

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\det(-d, e_1, e_2)} \begin{bmatrix} \det(s, e_1, e_2) \\ \det(-d, s, e_2) \\ \det(-d, e_1, s) \end{bmatrix}$$

  - Can be rewritten:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(d \times e_2) \cdot e_1} \begin{bmatrix} (s \times e_1) \cdot e_2 \\ (d \times e_2) \cdot s \\ (s \times e_1) \cdot d \end{bmatrix} = \frac{1}{p \cdot e_1} \begin{bmatrix} q \cdot e_2 \\ p \cdot s \\ q \cdot d \end{bmatrix}$$

    - where:

$$p = d \times e_2 \qquad q = s \times e_1$$

Ray / Triangle Intersection PseudoCode

- Pseudo code in Section 22.8.2 (16.8.2 3rd Edition)
  - Return true/false
    - Does an intersect occur
  - Return u,v,t
    - Can construct the intersect point from either (u, v) or t
      - u,v can be used for interpolation of color/texture/vertex normals
    - t is the distance along the ray at the intersect point
  - Code does not cull back-facing triangles
- This method is useful for picking operations
  - Determine if ray intersects triangle
- Methods for ray tracing require a normal as well as the intersect point
  - Could recover normal from object data or regenerate using cross product

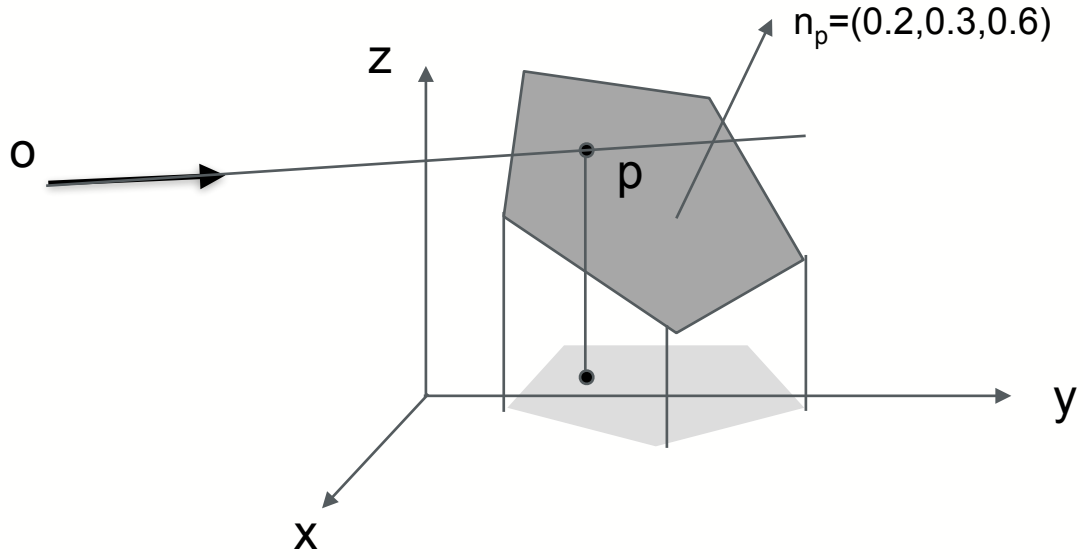
Ray / Polygon Intersection

- Most object models are made of triangles, but some use quadrilaterals or convex polygons
  - Useful to have a general ray-polygon intersection method
- Polygon is represented as an ordered vertex list and a normal n of the polygon's plane
  - Normal can be converted on the fly or may be stored with the polygon
- To find the ray / polygon intersection:
  - Find intersection between ray and the plane containing the polygon
  - Project polygon vertices and ray/plane intersect point to 2D
  - Check if intersect coordinate is inside the 2D polygon

Reducing the Problem to 2D

- Project the intersect point and the polygon orthographically onto one the coordinate axis planes
  - To be most accurate choose the coordinate with the largest coefficient in the polygon's plane equation / normal
    - This yields the largest area projection of the polygon
    - Projection is achieved by simply dropping the selected coordinate from the polygon and the intersect point
      - Skip the coordinate that corresponds to
    - Point-in-polygon test can then be done in 2D

$$\max\left(\left|n_{px}\right|, \left|n_{py}\right|, \left|n_{pz}\right|\right)$$

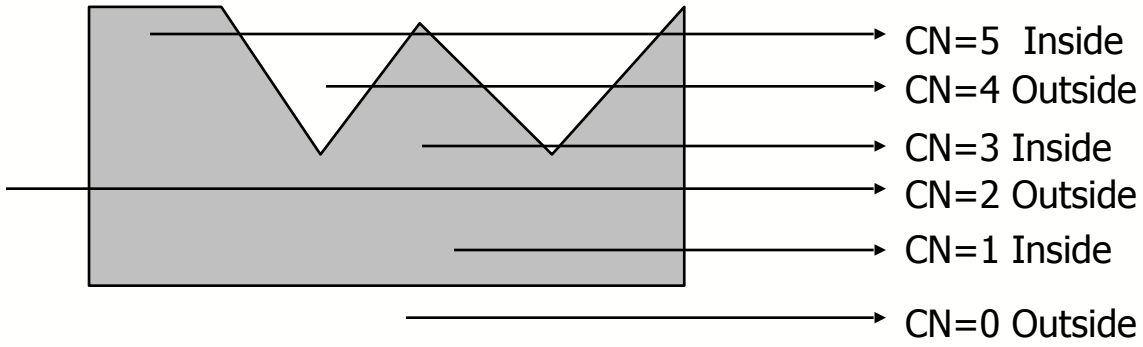


Since the z component of the normal is the largest, we “project” onto the xy plane

see Figure 22.15 (16.11 3rd Edition)

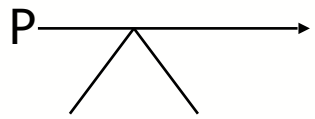
Crossing Number Method

- Based on the Jordan Curve Theorem
  - Count the number of times a ray P crosses the polygon edges
  - Each time it crosses the in-out parity changes
  - Go from inside to outside or from outside to inside
    - Ray eventually is outside the polygon
    - Last transition is inside to outside
  - Point inside has an odd number of crossings
    - e.g., in-out, out-in, in-out
  - Point outside has an even number of crossings
    - e.g., out-in, in-out, out-in, in-out

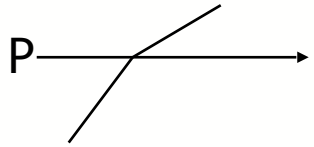


Crossing Number Pseudo Code

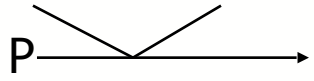
- Haines and Moller discuss an optimized point in polygon method
  - In the xy plane
    - Shooting a ray from test point p along the positive x axis
    - Can think of test point p as being at the origin
      - Test edges against the positive x axis
      - See Figure 22.17 (16.13 3rd Edition)
    - If y coords. of a polygon edge have same y sign they cannot cross x axis
    - If they differ it can cross and x intersect is checked
      - Optimization: test if both edge x values are left or right of test x
        - If so no need to calculate the exact intersect
  - Ray that intersects a vertex works properly
- By interpreting vertices with y>= ty as lying above the ray



Flips "inside" flag twice
Net result is inside-outside parity is unchanged



Flips "inside" flag once
Inside-outside parity changes



Does not flips "inside" flag

2D Point in Triangle Test

- Alternate ray/triangle intersection test
  - If face normals are available in object data and do not need u,v
- Find intersection between ray and plane containing the triangle
- Project triangle vertices and ray/plane intersect point to 2D
- Check if intersect coordinate is inside the 2D triangle
  - Iterate through triangle edges and test “side” of the intersect point
    - 2D cross product (IsLeft method)
  - If any time the sign of the return changes the point is outside

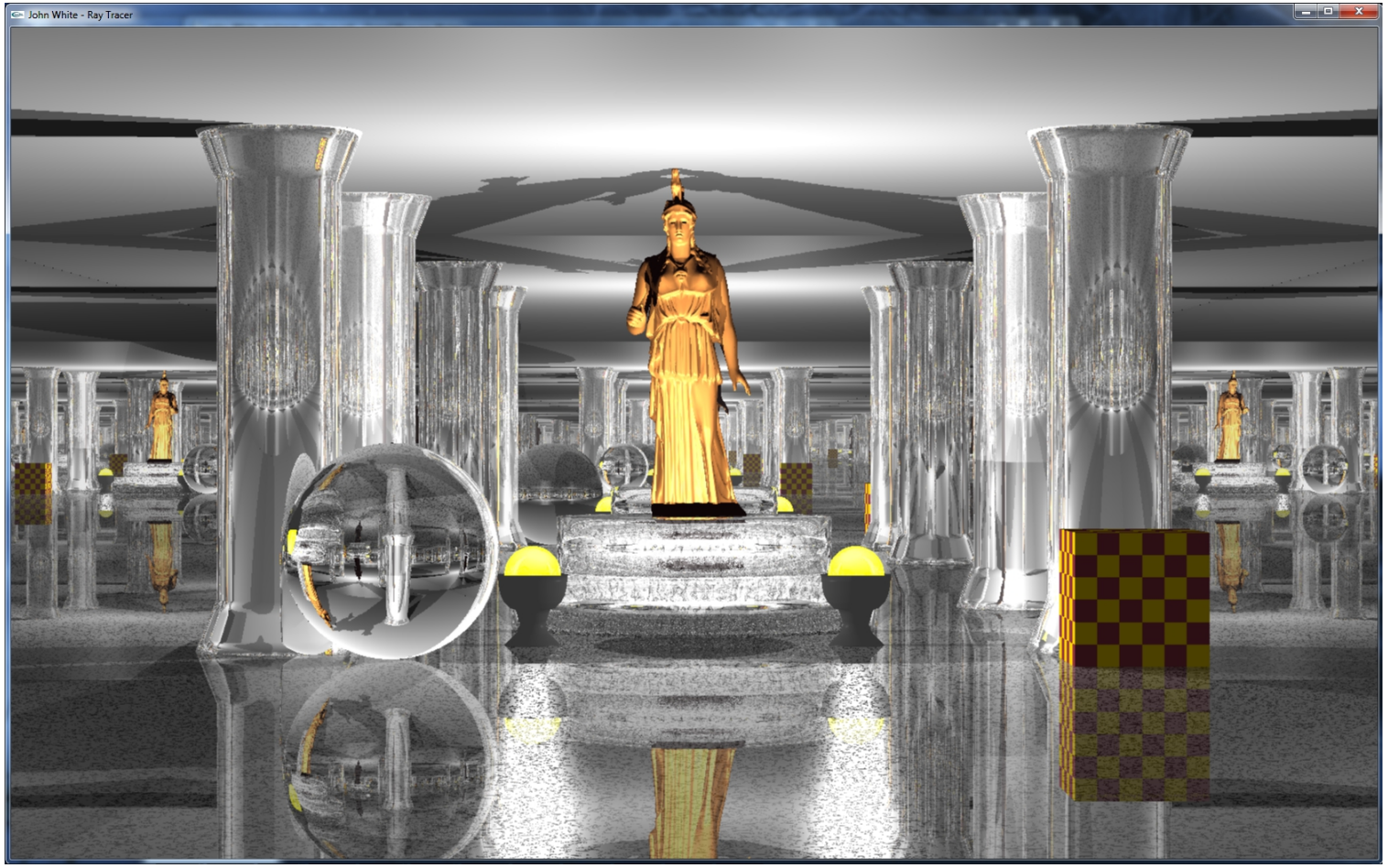
```
/**
 *  Tests if the point is left, on, or right of an infinite line
 *  specified by two other points (relative to the direction from p1 to p2)
 * @return   Returns >0 if the point is left of the line through P1 and P2
 *                          =0 if the point is on the line
 *                           <0 if the point is right of the line
 */
int Point2::IsLeft(Point2& p1, Point2&  p2)
{
    return ((p1.x - x) * (p2.y - y)  - (p2.x - x) * (p1.y - y));
}
```

Fast Inverse SQRT

- Quake used a fast inverse sqrt – 1999
  - Uses a unique first approximation for Newton's method
  - Probably developed by Silicon Graphics in 1990s
  - http://en.wikipedia.org/wiki/Fast_inverse_square_root
- More details in: http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf
  - Claimed to be roughly 4 times faster than (float)(1.0/sqrt(x))
  - Very small error
    - maximum relative error over all floating point numbers was 0.00175228
  - I found it not accurate enough for ray tracing!
- Useful to speed up normalization of vectors

```
float FastInvSqrt(float x)  {
   float xhalf = 0.5f * x;
   int i = *(int*)&x;              // get bits for floating value
   i = 0x5f3759df - (i>>1);        // give initial guess y0
   x = *(float*)&i;                // convert bits back to float
   return x*(1.5f - xhalf*x*x);    // newton step
   // x *= 1.5f - xhalf*x*x;       // repeating step increases accuracy
}
```

Importance of Precision



Black spots indicate areas where roundoff/precision impact computation of intersections.
John White – Ray Tracer Project. 2010