CS 450: COMPUTER GRAPHICS

RAY TRACING

SPRING 2016

DR. MICHAEL J. REALE
Let’s say we have a scene of 3D objects
We want to render this to a 2D raster image

In general, there are two ways we can do this:

- **Object-order rendering**
  - For each OBJECT → find pixels that object affects
  - Often used for real-time rendering

- **Image-ordering rendering**
  - For each PIXEL → find all objects that affect this pixel
  - Often used for non-real-time rendering
RAY TRACING

• We’re going to talk about ray tracing
  • Form of image-order rendering
  • Advantages:
    • High-quality images
      • Easy to compute:
        • Shadows
        • Reflections
        • Refraction
  • Disadvantages:
    • Generally pretty slow
BASIC RAY-TRACING ALGORITHM
BASIC IDEA

- We have a virtual camera that starts at point E
- Imagine that there are a grid of pixels floating in front of the camera
- To get the color for each pixel:
  - Shoot a ray from the starting point E to the pixel position
  - See what object(s) we hit with the ray
  - Pick the nearest intersection point
  - Compute color at that point
    - Shading model \(\rightarrow\) looking at normal at that point, lighting direction, color, etc.
THREE BASIC STEPS

• This means we have three basic steps:
  • **Ray generation**
    • Compute viewing ray for each pixel
  • **Ray intersection**
    • Find closest intersection point among all objects in scene
    • This step → *why raytracers are generally slow*
  • **Shading**
    • Compute pixel color based on results of ray intersection
3D TO 2D

• Our first question is how our camera will **project** the 3D world onto a 2D image?
  • *Old problem*: artists had been working on this one for centuries

• **Simplest approach:**
  • Distance from camera doesn’t matter
  • Only care where it is in terms of left/right and up/down

• **More realistic approach:**
  • Take *perspective* into account → things farther away should look smaller

https://static.greatbigcanvas.com/categories/medieval-art-11307.jpg
PROJECTIONS

• Two most commonly used projection methods:
  • Orthographic (or parallel)
  • Perspective

• Note: when defining a projection, also define:
  • Near plane = closer than near plane \(\rightarrow\) don’t render
  • Far plane = beyond far plane \(\rightarrow\) don’t render

• Coupled with other aspects of camera, defines **view volume**
  • Objects in view volume will be rendered

• Projection direction = direction our camera is pointed
ORTHOGONAL PROJECTION

• Orthographic (or parallel)
  • Move object along projection direction until you hit image plane
    • Distance from image plane doesn’t change appearance
  • Parallel lines remain parallel
  • View volume = rectangular box
ORTHOGRAPHIC PROJECTION: PROS AND CONS

• Orthographic (or parallel)
  • Advantages:
    • Good for mechanical/architectural drawings
      • Size of objects remains the same, no matter how far away they are
      • Parallel lines preserved
  • Disadvantages:
    • Not realistic → effectively assumes multiple camera (eye) starting points
      • In reality, just one (or two, with the human vision system)
**Perspective Projection**

- **Perspective**
  - Single starting point (eye point) → all objects project towards eye point
    - Things look smaller when farther away
  - **View volume** = truncated pyramid with rectangular base → called **view frustum**
ASIDE: OBLIQUE PROJECTION

- If the image plane is NOT orthogonal to the project direction → oblique projection
  - *Below*: example of an oblique parallel projection
COMPUTING VIEWING RAYS
CREATING RAYS

• To define our rays, we’ll just use 3D parametric rays

• Assuming we have:
  • Starting point E (also called *eye point*)
  • Point S on the image plane

• Then our ray is defined by:

\[ P(t) = E + t(S - E) \]
BEHIND BLUE EYES

• We know that:
  \[ P(0) = E \]
  \[ P(1) = S \]

• We also know that, if:
  \[ 0 < t_1 < t_2 \]

  • Then \( P(t_1) \) is closer to eye than \( P(t_2) \)

• If \( t < 0 \) → “behind” the eye (shouldn’t be able to see item)
CAMERA FRAME

• We need to find the S points
  • E will be given to us, depending on the kind of projection
  • To do this, we will look at things from the perspective of the camera frame
    • **Camera frame** = orthonormal coordinate frame defined by the eye point E and three vectors:
      • U \( \rightarrow \) points to camera’s RIGHT
      • V \( \rightarrow \) points upward
      • W \( \rightarrow \) points BACKWARD

  • *Remember*: RIGHT-HAND RULE, so **camera is actually looking along the –W axis**!
GETTING THE IMAGE PLANE

• Based on the camera frame, we’ll construct our **image plane**

• We know:
  • Right (U) and Up (V) directions for plane
  • Assume we also know where the plane is centered

• **How large is the plane?**

• Usually define plane in terms of the following coordinates:
  • $r$ (right) $\rightarrow$ max coordinate ALONG U axis ($r > 0$)
  • $l$ (left) $\rightarrow$ min coordinate ALONG U axis ($l < 0$)
  • $t$ (top) $\rightarrow$ max coordinate ALONG V axis ($t > 0$)
  • $b$ (bottom) $\rightarrow$ min coordinate ALONG V axis ($b < 0$)
GETTING POINTS ON THE IMAGE PLANE

• We then need to define what points on the image plane correspond to our pixels
• If:
  • \( n_x \) = # of pixels in x
  • \( n_y \) = # of pixels of y
• To get the coordinates \((u,v)\) for a given point (assuming that pixels are ACTUALLY centered on the 2D coordinates):

\[
\begin{align*}
u &= l + (r - l)(i + 0.5) / n_x \\
v &= b + (t - b)(j + 0.5) / n_y
\end{align*}
\]
ORTHOGRAPHIC VIEWS

• For an orthographic view, all ray DIRECTIONS = -W
• Image plane centered at eye point E
• For each pixel:
  • Compute u and v (previous slide)

Ray direction  = \(-W\)
Ray starting point  = \(E + uU + vV\)

• For oblique views \(\rightarrow\) all ray directions = \(d\) (some other direction)
PERSPECTIVE VIEWS

• For **perspective** views, all ray **STARTING POINTS** = E
  • Image plane centered at (E - dW)
    • \( d = \) image plane distance
    • Sometimes loosely called **focal length** \( \rightarrow \) changing \( d \) does change **field of view**

• **For each pixel:**
  • Compute \( u \) and \( v \) (two slides back)

\[
\text{Ray direction} = uU + vV - dW
\]

\[
\text{Ray starting point} = E
\]

• For **oblique** views \( \Rightarrow \) use \( dD \) instead of \( -dW \)
RAY-OBJECT INTERSECTION
FINDING OBJECTS

• We now have our ray: $P(t) = E + tD$

• We need to find out what the ray intersects with in the 3D scene

• Want to pick the nearest intersection point with $t$ in the interval: $[t_0, t_1]$
  • Usually: $[0, +\infty]$  

• Although there are many possible objects to intersect with, we will discuss ray intersections with:
  • Spheres
  • Triangles
  • Polygons
RAY INTERSECTION WITH IMPLICIT SURFACE

• Here’s our ray again:

\[ P(t) = E + tD \]

• Given an implicit surface:

\[ f(P) = 0 \]

• Intersection points = points on ray that satisfy the implicit equation:

\[ f(P(t)) = 0 \quad \text{or} \quad f(E + tD) = 0 \]

• So we have to find values of \( t \) (if any) that satisfy the above equation.
RAY-SPHERE INTERSECTION
RAY-SPHERE INTERSECTION

- Here’s our implicit equation (in vector form) for a sphere:

\[ f(P) = (P - C) \cdot (P - C) - r^2 = 0 \]

- Let’s plug in our ray for \( P \):

\[ f(P) = (E + tD - C) \cdot (E + tD - C) - r^2 = 0 \]
RAY-SPHERE INTERSECTION

• Rearranging terms:

\[(E + tD - C) \cdot (E + tD - C) - r^2\]
\[= (E \cdot E) + (tD \cdot E) - (C \cdot E) + (E \cdot tD) + (tD \cdot tD) - (C \cdot tD) - (E \cdot C) - (tD \cdot C) + (C \cdot C) - r^2\]
\[= (D \cdot D)t^2 + 2t(D \cdot E) - 2(C \cdot E) - 2t(D \cdot C) + (E \cdot E) + (C \cdot C) - r^2\]
\[= (D \cdot D)t^2 + 2t(D \cdot E - D \cdot C) + (E \cdot E) - 2(C \cdot E) + (C \cdot C) - r^2\]
\[= (D \cdot D)t^2 + 2(D \cdot (E - C))t + (E - C) \cdot (E - C) - r^2\]
RAY-SPHERE INTERSECTION

• A closer look at our result: 

\[(D \cdot D)t^2 + 2(D \cdot (E - C))t + (E - C) \cdot (E - C) - r^2 = 0\]

• ...reveal that this is a quadratic equation: 

\[at^2 + bt + c = 0\]

• ...where:

\[
\begin{align*}
    a &= (D \cdot D) \\
    b &= 2(D \cdot (E - C)) \\
    c &= (E - C) \cdot (E - C) - r^2
\end{align*}
\]
RAY-SPHERE INTERSECTION

• This means we can use the quadratic formula!

\[
t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]

\[
a = (D \cdot D)
\]
\[
b = 2(D \cdot (E - C))
\]
\[
c = (E - C) \cdot (E - C) - r^2
\]
RAY-SPHERE INTERSECTION

• Cleaning things up a bit:

\[ t = \frac{- (D \cdot (E - C)) \pm \sqrt{(D \cdot (E - C))^2 - (D \cdot D)((E - C) \cdot (E - C) - r^2)}}{(D \cdot D)} \]

• If we look at the discriminant \( G \):
  • \( G > 0 \) \( \rightarrow \) TWO possible intersection points
  • \( G = 0 \) \( \rightarrow \) ONE possible intersection point (grazing surface of sphere)
  • \( G < 0 \) \( \rightarrow \) NO intersection

• In implementation, one should check the discriminant first before computing other terms
  • In fact, if doing collision detection \( \rightarrow \) may only need to compute the discriminant!
RAY-SPHERE INTERSECTION

$D < 0$
No Intersection

$D = 0$
ONE intersection point

$D > 0$
TWO intersection points
RAY-SPHERE INTERSECTION: GEOMETRIC INTERPRETATION

• Let’s try to break this down geometrically...
• Vector going from center of circle TO eye point: \((E - C)\)
• Length of projection of \((E-C)\) onto \(D\):
  \[
  \frac{D \cdot (E - C)}{D \cdot D} = \frac{\|D\| \|E - C\| \cos \theta}{\|D\|^2}
  \]
  
• However, if we just wanted the adjacent side and assumed \(D\) was normalized:
  \[p = \|E - C\| \cos \theta\]

• THAT SAID, note that \((E-C)\) and \(D\) point in opposing directions, so the distance along \(D\) we ACTUALLY want is:
  \([-p]\)
That said, our original formula:

\[
t = \frac{-(D \cdot (E - C)) \pm \sqrt{(D \cdot (E - C))^2 - (D \cdot D)((E - C) \cdot (E - C) - r^2)}}{(D \cdot D)}
\]

...becomes (also converting certain dot products to the lengths-squared):

\[
t = \frac{-\|D\|p \pm \sqrt{\left(\|D\|p\right)^2 - \|D\|^2 \left(\|E - C\|^2 - r^2\right)}}{\|D\|^2}
\]
RAY-SPHERE INTERSECTION:
GEOMETRIC INTERPRETATION

\[ t = \frac{-\|D\| p \pm \sqrt{\left(\|D\| p\right)^2 - \|D\|^2 (\|E - C\|^2 - r^2)}}{\|D\|^2} \]

- We can actually move the length of D outside of the square root:

\[ t = \frac{-\|D\| p \pm \|D\| \sqrt{p^2 - (\|E - C\|^2 - r^2)}}{\|D\|^2} \]
RAY-SPHERE INTERSECTION: GEOMETRIC INTERPRETATION

\[
t = \frac{-\|D\|p \pm \|D\|\sqrt{p^2 - (\|E-C\|^2 - r^2)}}{\|D\|^2}
\]

• Let’s say that:

\[q = \|E - C\|\]

• Then our formula becomes:

\[
t = \frac{-\|D\|p \pm \|D\|\sqrt{p^2 - (q^2 - r^2)}}{\|D\|^2}
\]
RAY-SPHERE INTERSECTION: GEOMETRIC INTERPRETATION

• Let’s look at the discriminant more closely: \( p^2 - (q^2 - r^2) \)

• Rearranging it a bit: \( r^2 + (p^2 - q^2) \)

• Because of the Pythagorean theorem: \( q^2 = p^2 + s^2 \)

• So, if we’re trying to get \( s^2 \): \( s^2 = q^2 - p^2 \)

• SO, we are effectively getting the difference between \( r^2 \) and \( s^2 \):

\[
\begin{align*}
    r^2 + (p^2 - q^2) &= r^2 - (q^2 - p^2) \\
    &= r^2 - s^2
\end{align*}
\]
RAY-SPHERE INTERSECTION: GEOMETRIC INTERPRETATION

• SO, when we are computing the discriminant $G$: \[ G = r^2 - s^2 \]

$G > 0 \rightarrow r^2 > s^2 \rightarrow$ TWO intersecti on points

$G = 0 \rightarrow r^2 = s^2 \rightarrow$ ONE intersecti on point

$G < 0 \rightarrow r^2 < s^2 \rightarrow$ no intersecti on point
RAY-SPHERE INTERSECTION: GEOMETRIC INTERPRETATION

• Let’s assume we have TWO intersection points (G > 0)
• It turns out that, if we look at ANOTHER right triangle, then:

\[ s^2 + k^2 = r^2 \]

\[ k^2 = r^2 - s^2 \]

• Which means that the square root of our discriminant gives us:

\[ \pm \sqrt{G} = \pm \sqrt{r^2 - s^2} = \pm \sqrt{k^2} = \pm k \]
RAY-SPHERE INTERSECTION: GEOMETRIC INTERPRETATION

\[ t = \frac{-||D||p \pm ||D||\sqrt{p^2 - (q^2 - r^2)}}{||D||^2} \]

- SO, plugging this value into our original formula (and canceling out the extra \( ||D|| \)):
  \[ t = \frac{-p \pm k}{||D||} \]

- In other words, in terms of ACTUAL geometric distance, we need to go along \( D \) by:
  - \((-p + k)\)
  - \((-p - k)\)
  - ...to get our two intersection points.

- HOWEVER, because \( D \) may not be normalized, we have to divide by the length of \( D \).
RAY-TRIANGLE INTERSECTION
INTERSECTION WITH A TRIANGLE

• While there are many ways to do ray-triangle intersections, we will use:
  • Barycentric coordinates
  • A parametric plane containing the triangle
• Basically, all we need to store for this are the vertices of the triangle
RAY INTERSECTION WITH A PARAMETRIC SURFACE

• To intersect a ray with a **parametric surface** → set up the following equations:

\[
\begin{align*}
  x_E + tx_D &= f_x(u, v) \\
  y_E + ty_D &= f_y(u, v) \\
  z_E + tz_D &= f_z(u, v)
\end{align*}
\]

• *Three unknowns* \((t,u,v) → three equations*
  
  • Hopefully, we can solve this analytically, but in the worst case we can throw numerical methods at it...

• **Remember**: parametric form → plug in parameter(s) → return a POINT

  • Parametric line: \(P(t): R \mapsto \mathbb{R}^3\)
  
  • Parametric surface: \(f(u, v): \mathbb{R}^2 \mapsto \mathbb{R}^3\)
RAY INTERSECTION WITH A PARAMETRIC PLANE

- If our surface = parametric plane → we can use the vector form with barycentric coordinates!
  - \((\beta, \gamma)\) → parameters for surface
  - Returns a 3D point
  - Three points of triangle → define a plane
- So, our equation to solve becomes:

\[
E + tD = A + \beta(B - A) + \gamma(C - A)
\]

- Solve for \(t, \beta, \gamma\)
RAY INTERSECTION WITH A TRIANGLE

• INSIDE the triangle IF:
  • \( \beta > 0 \)
  • \( \gamma > 0 \)
  • \( (\beta + \gamma) < 1 \)

• OTHERWISE, misses triangle  (although it DOES hit the plane)

• If there IS no solution \( \rightarrow \) either:
  • Triangle is degenerate
  • OR
  • Ray is parallel to plane

\[
E + tD = A + \beta(B - A) + \gamma(C - A)
\]
SOLVING FOR PARAMETERS

\[ E + tD = A + \beta(B - A) + \gamma(C - A) \]

• First, we’ll rearrange things so that all our constants are on the right-hand side and everything else is on the left:
  • We’re going to turn this into a **standard linear system** so we can use matrices to solve the equations...hold tight...

\[
\begin{align*}
tD &= A - E + \beta(B - A) + \gamma(C - A) \\
&- \beta(B - A) - \gamma(C - A) + tD = A - E \\
\beta(A - B) + \gamma(A - C) + tD &= A - E
\end{align*}
\]
SOLVING FOR PARAMETERS

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

• Then, we’ll expand this out into the individual coordinates:
  
  • *Remember*: the \((x, y, z)\) coordinates are known values

\[
\begin{align*}
\beta(x_A - x_B) + \gamma(x_A - x_C) + tx_D &= x_A - x_E \\
\beta(y_A - y_B) + \gamma(y_A - y_C) + ty_D &= y_A - y_E \\
\beta(z_A - z_B) + \gamma(z_A - z_C) + tz_D &= z_A - z_E
\end{align*}
\]
SOLVING FOR PARAMETERS

Finally, we’ll turn this into a standard linear system of the form \( MV = N \)

\[
\begin{align*}
\beta (x_A - x_B) + \gamma (x_A - x_C) + tx_D &= x_A - x_E \\
\beta (y_A - y_B) + \gamma (y_A - y_C) + ty_D &= y_A - y_E \\
\beta (z_A - z_B) + \gamma (z_A - z_C) + tz_D &= z_A - z_E
\end{align*}
\]
QUICK REVIEW: MATRIX MULTIPLICATION

- **Matrix** = effectively, a 2D array of numbers
  - $M = 3 \times 3$ matrix

- **Vector** = basically a matrix with one of the dimensions equaling 1
  - $V$ and $N = 3 \times 1$ matrices = 3D column vectors

- When multiplying a matrix $M$ by a vector $V$ (IN THAT ORDER):
  - **Result** = vector $N$ with:
    - Same # of rows as $M$
    - Same # of columns as $V$
    - For each value $n_R$ in $N$:
      - Dot product of $(Rth$ row of $M)$ and $(1st$ column of $V)$

$$MV = N$$

\[
\begin{bmatrix}
m_{00} & m_{01} & m_{02} \\
m_{10} & m_{11} & m_{12} \\
m_{20} & m_{21} & m_{22} \\
\end{bmatrix} \begin{bmatrix}
v_0 \\
v_1 \\
v_2 \\
\end{bmatrix} = \begin{bmatrix}
n_0 \\
n_1 \\
n_2 \\
\end{bmatrix}
\]

\[
n_0 = m_{00}v_0 + m_{01}v_1 + m_{02}v_2
\]

\[
n_1 = m_{10}v_0 + m_{11}v_1 + m_{12}v_2
\]

\[
n_2 = m_{20}v_0 + m_{21}v_1 + m_{22}v_2
\]
BACK TO OUR LINEAR SYSTEM...

- So, for example, if I wanted the first equation back...

\[
\begin{bmatrix}
    x_A - x_B & x_A - x_C & x_D \\
    y_A - y_B & y_A - y_C & y_D \\
    z_A - z_B & z_A - z_C & z_D
\end{bmatrix}
\begin{bmatrix}
    \beta \\
    \gamma \\
    t
\end{bmatrix}
= 
\begin{bmatrix}
    x_A - x_E \\
    y_A - y_E \\
    z_A - z_E
\end{bmatrix}
\]

\[
\beta(x_A - x_B) + \gamma(x_A - x_C) + tx_D = x_A - x_E
\]
QUICK REVIEW: DETERMINANT OF A MATRIX

- The determinantal of a matrix is calculated for a 2x2 and 3x3 matrix in the following fashion:
  - NOTE: The determinant is related to the cross product, scalar triple product, and the area/volume of parallelograms/parallelepipeds formed by the vectors, but we’ll get to that later...

\[
M = \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} = 1*4 - 2*3
\]

\[
M = \begin{vmatrix} i & j & k \\ 1 & 2 & 3 \end{vmatrix} = i(2*6 - 3*5) + j(3*4 - 1*6) + k(1*5 - 2*4)
\]
CRAMER’S RULE

• For 3x3 systems, a fast way to solve them is with Cramer’s Rule:
  • Again, we’ll talk about how and why this works later...
• Given a system: \( MV = N \)
• For each coordinate of \( V \) (\( v_i \)):
  • Replace the \( i^{th} \) column of matrix \( M \) with \( N \)
  • Get the determinant
  • Divide by the determinant of the original matrix \( M \)
• WARNING: If \( |M| = 0 \) \( \rightarrow \) no unique solution!
  • May be no solution...may be INFINITE solutions...
APPLYING CRAMER’S RULE

So, in determinant form, to solve for our parameters:

\[
M = \begin{bmatrix}
  x_A - x_B & x_A - x_C & x_D \\
  y_A - y_B & y_A - y_C & y_D \\
  z_A - z_B & z_A - z_C & z_D
\end{bmatrix}
\]

\[
\beta = \frac{\begin{vmatrix}
  x_A - x_E & x_A - x_C & x_D \\
  y_A - y_E & y_A - y_C & y_D \\
  z_A - z_E & z_A - z_C & z_D
\end{vmatrix}}{|M|}
\]

\[
\gamma = \frac{\begin{vmatrix}
  x_A - x_B & x_A - x_C & x_D \\
  y_A - y_B & y_A - y_C & y_D \\
  z_A - z_B & z_A - z_C & z_D
\end{vmatrix}}{|M|}
\]

\[
t = \frac{\begin{vmatrix}
  x_A - x_B & x_A - x_C & x_A - x_E \\
  y_A - y_B & y_A - y_C & y_A - y_E \\
  z_A - z_B & z_A - z_C & z_A - z_E
\end{vmatrix}}{|M|}
\]
APPLYING CRAMER’S RULE

- If we use dummy variables:

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

- We can show exactly what we need to calculate (and we can reduce the number of operations by calculating things like \((ei - hf)\) once):

\[
\beta = \frac{j(ei - hf) + k(gf - di) + l(dh - eg)}{p}
\]

\[
\gamma = \frac{i(ak - jb) + h(jc - al) + g(bl - kc)}{p}
\]

\[
t = \frac{f(ak - jb) + e(jc - al) + d(bl - kc)}{p}
\]

\[
p = a(ei - hf) + b(gf - di) + c(dh - eg)
\]
CODING THE FUNCTION

• When coding the intersection function, we would want to terminate early if we can
  • More efficient
• Let’s assume t has to be in the interval \([t_0, t_1]\)
• SO, our algorithm might look like:
  • Compute t
  • If \((t < t_0) \text{ OR } (t > t_1)\) \(\rightarrow\) return false
  • Compute \(\gamma\)
  • if \((\gamma < 0) \text{ OR } (\gamma > 1)\) \(\rightarrow\) return false
  • Compute \(\beta\)
  • If \((\beta < 0) \text{ OR } (\beta > 1 - \gamma)\) \(\rightarrow\) return false
  • Return true
RAY-POLYGON INTERSECTION
WHAT IS A POLYGON, EXACTLY?

• The official definition varies, but at bare minimum:

• **Polygon** = a figure with 3 or more vertices connected in sequence by straight-line segments (edges or sides of the polygon)
  - *Most loose definition* → any closed-polyline boundary
  - *More finicky definitions* → contained in single plane, edges have no common points other than their endpoints, no three successive points collinear

• **Standard polygon or simple polygon** = closed-polyline *with no crossing edges*
DOES A POLYGON LIE IN A SINGLE PLANE?

• In computer graphics, polygon not always in same plane:
  • Round-off error
    • I.e., points originally in single plane, but, after transformation, the points may be slightly off
  • Fitting to surface makes non-planar polygons
    • E.g., if using quads, approximation may “bend” the quad in half

• Thus, we usually use triangles to avoid this problem
DEGENERATE POLYGONS

• **Degenerate polygons** = often used to describe polygon with:
  • **3 or more collinear vertices** → generates a line segment
    • E.g., in extreme case, triangle with no area

• **Repeated vertex positions** → generates shape with:
  • Extraneous lines
  • Overlapping edges
  • Edges with length 0
INTERSECTING WITH THE PLANE OF THE POLYGON

• If ALL of the vertices $P_1$ through $P_m$ of a polygon lie in the same plane $\rightarrow$ first get intersection with plane

• Point-normal form of plane (we’ll just use the first point):

\[(P - P_1) \cdot N = 0\]

• Plugging in our ray equation for $P$:

\[
\begin{align*}
(E + tD - P_1) \cdot N &= 0 \\
(E - P_1) \cdot N + tD \cdot N &= 0 \\
t(D \cdot N) &= -(E - P_1) \cdot N \\
t &= \frac{(P_1 - E) \cdot N}{D \cdot N}
\end{align*}
\]
ARE WE INSIDE THE POLYGON?

• Once we have our intersection point P, the question is: **are we inside the polygon?**

• **One trick:**
  - Project point P and polygon to the XY plane
  - Shoot a ray out from P
    • Usually easiest to use X axis
  - Check what we cross with this ray
    • Number of edges OR direction of edges

• **Two common approaches for the last step:**
  - Odd-Even Rule
  - Nonzero Winding-Number Rule
ODD-EVEN RULE

• Also called *odd-parity* or *even-odd rule*
• Draw ray starting from position P
• Count how many edges we cross
  • ODD $\rightarrow$ inside polygon
  • EVEN $\rightarrow$ outside polygon
ASIDE: OTHER USES OF ODD-EVEN RULE

- Can use to check other kinds of regions
  - E.g., area between two concentric circles
NONZERO WINDING-NUMBER RULE

- Count number of times boundary of object “winds” around a particular point in the clockwise direction

- Again, draw ray starting at position P
- Start “winding number” W at 0
- If intersection with segment that crosses the line going from:
  - Right-to-left (counterclockwise) \( \rightarrow \) add 1 to W
  - Left-to-right (clockwise) \( \rightarrow \) subtract 1 from W
  - *Remember: think right-hand-rule*
- Check value of W when done:
  - If \( W == 0 \) \( \rightarrow \) EXTERIOR
  - If \( W != 0 \) \( \rightarrow \) INTERIOR
NONZERO WINDING-NUMBER RULE

- To check the direction of the edge relative to the line from P:
  - Make vectors for each edge (in the correct winding order) → vector $E_i$
  - Make vector from P to distant point → vector $U$
  - Two options:
    - Use **cross product**:
      - If $(U \times E_i) = +Z$ axis → right-to-left → add 1 to winding number
      - If $(U \times E_i) = -Z$ axis → left-to-right → subtract 1 from winding number
    - Use **dot product**:
      - Get vector $V$ that is 1) perpendicular to $U$, and 2) goes right-to-left → $(u_y, u_x)$
      - If $(V \cdot E_i) > 0$ → right-to-left → add 1 to winding number
      - Otherwise → left-to-right → subtract 1 from winding number
ODD-EVEN VS. NONZERO WINDING-NUMBER

- For simple objects (polygons that don’t self-intersect, circles, etc.), both approaches give same result;
- However, more complex objects may give different results;
  - Usually, nonzero winding-number rule classifies some regions as interior that odd-even says are exterior.

Odd-Even Rule

Nonzero Winding-Number Rule
INSIDE-OUTSIDE PROBLEM

• *Caveat*: need to check we don’t cross endpoints (vertices)
  • Otherwise, ambiguous → we’ll talk in more detail how to deal with this when we get to filling polygons later...
ASIDE: CURVED PATHS?

• For curved paths, need to computer intersection points with underlying mathematical curve
  • For nonzero winding-number → also have to get tangent vectors
ASIDE: VARIATIONS OF NONZERO WINDING-NUMBER RULE

- Can be used to define Boolean operations:
  - Positive $W$ only, both counterclockwise $\rightarrow$ union of $A$ and $B$
  - $W > 1$, both counterclockwise $\rightarrow$ intersection of $A$ and $B$
  - Positive $W$ only, $B$ clockwise $\rightarrow$ $A - B$
PROJECTING THE POLYGON PROBLEMS

• What if the projection of the polygon into XY is a LINE?
  • E.g., the polygon is in the XZ plane?
  • Solution: choose best among XY, YZ, or ZX planes

```
if (abs(z_n) > abs(x_n)) & (abs(z_n) > abs(y_n)) → use XY plane
elseif (abs(y_n) > abs(x_n)) → use ZX plane
else → use YZ plane
```
SHADING
SHADING MODEL

• Let’s assume we’ve:
  • Intersected our view ray with every object in the scene
  • Found the nearest intersection point (and associated object)
• The pixel value is computed by evaluating a shading model

• Shading model (or lighting model)
  • Used to calculate the color of an illuminated position on the surface of an object

• In the slides that follow, we will concentrate on shading models that use point lights
POINT LIGHTS

• Point light
  • Located at a single point in space
  • Emit light in all directions
DIFFUSE REFLECTION

• **Diffuse reflection** = when white light hits an object, what we see as the “color” of an object
  - Example: apple → absorbs all frequencies except red → has red diffuse color
  - *Underlying physics*: surfaces with microfacets (bumpy, grainy, matte) → reflects light in lots of different directions

• **Ideal diffuse reflectors** or **Lambertian reflectors**
  - Incident light scattered with equal intensity in ALL directions, INDEPENDENT of viewing angle
  - Depends on angle between NORMAL and DIRECTION-to-LIGHT → **angle of incidence**
LAMBERTIAN SHADING

- Lambertian shading model
  - Simplest of all shading models
  - Amount of light energy that hits a surface $\rightarrow$ proportional to angle of surface to light
  - View-independent $\rightarrow$ the viewer's position doesn't matter
  - If:
    - $N =$ NORMALIZED normal
    - $L =$ NORMALIZED vector from point on surface to light source
  - Then dot product is proportional to angle between $L$ and $N$:

$$N \cdot L = \cos \theta$$
LAMBERTIAN SHADING

• If the light is BEHIND the surface → angle between L and N > 90 degrees → dot product will be NEGATIVE
• So, we only want the max of 0 and the dot product: \( \max (0, N \cdot L) \)
• That said, our pixel color will be:

\[
\text{Pixel color} = k_d I \max (0, N \cdot L)
\]

• ...where:
  • \( N \) = NORMALIZED normal
  • \( L \) = NORMALIZED vector from point on surface to light source
    • Subtract intersection point \( P \) from light position → then normalize vector
  • \( I \) = intensity of light
  • \( k_d \) = diffuse coefficient / diffuse color of surface
LAMBERTIAN SHADING

Pixel color = \( k_d I \max (0, N \cdot L) \)

• Keep in mind that we basically have THREE equations (one for red, one for green, and one for blue)
  • E.g., for the red pixel color \( \rightarrow \) have a red light intensity and a red diffuse coefficient
  • N and L are the same in all cases
PROBLEM WITH LAMBERTIAN SHADING

- Lambertian model $\rightarrow$ view independent
  - Provides \textit{diffuse component} of light
- HOWEVER, in real life $\rightarrow$ have shiny spots/specular reflections that DO depend on view position
  - Without this, models look kind of matte and chalky
- So, we need some way to model these \textit{specular highlights}...
BLINN-PHONG SHADING

• Blinn-Phong shading model
  • Given:
    • $V = \text{view direction} \rightarrow \text{NORMALIZED vector from intersection point to eye point}$
    • Produce brightest reflection of light when $V$ and $L$ are symmetric about normal $N$
      • Specular reflection = when all (or almost all) of the incident light is reflected back
        • A shiny or reflective spot
BLINN-PHONG SHADING

• First, compute **half-vector H**
  • Bisector of angle between V and L
  • Both V and L are the same length \(\rightarrow\) can just add together to get average direction vector, then normalize!

• If perfect reflection \(\rightarrow\) H will be perfectly in line with normal N

• So, we want to look at:

\[
\max(0, N \cdot H)
\]
BLINN-PHONG SHADING: ISSUES WITH THE HALF-VECTOR

- *Problem*: if V, L, and N are not coplanar → slightly off
- *Another problem*: need to check if V and L are on same side of N (\( L \cdot V > L \cdot N \)) → if so, don’t use specular effect at all
BLINN-PHONG SHADING: SHININESS

- Different surfaces reflect light over finite range of viewing positions
  - Shiny surfaces $\rightarrow$ narrow range
  - Duller surfaces $\rightarrow$ wider range

- So, we take the dot product to a power to make it decrease faster $\rightarrow$ use Phong exponent or shininess $s$:

$$
\max(0, N \cdot H)^s
$$
BLINN-PHONG SHADING: SHININESS

• Typical values of s:
  • 10 → “eggshell”
  • 100 → mildly shiny
  • 1000 → really glossy
  • 10,000 → nearly mirror-like
So, our final Blinn-Phong shading model will be:

$$\text{Pixel color} = k_d I \max (0, N \cdot L) + k_s I \max (0, N \cdot H)^S$$

...where:

- $k_s$ = specular coefficient / specular color of surface
  - Usually set this to gray or white
NORMALIZE YOUR VECTORS!!!

• IMPORTANT: DON’T FORGET TO NORMALIZE N, V, L, and H!!!
  • Otherwise, we have lengths that we don’t want creeping into the dot products...
AMBIENT SHADING

• Using either of the two previous models, if the surface is facing away from the light → completely black
  • *In real-life* → indirect reflections of other surfaces will cause SOME light to fall on those surfaces
• *HORRIBLE HACK*: Add a little bit of light to ALL surfaces → ambient shading
AMBIENT SHADING

• **Ambient shading**
  • Light all surfaces with “ambient” light that comes equally from everywhere
  • Combined with Blinn-Phong, we get:

\[
\text{Pixel color} = k_a I_a + k_d I \max(0, N \cdot L) + k_s I \max(0, N \cdot H)^S
\]

• ...where
  • \(I_a\) = ambient light intensity
  • \(k_a\) = surface’s ambient coefficient
    • Can be kept separate to tune ambient light per surface OR can set the same as diffuse light coefficient
FINAL MODEL: DIFFUSE, SPECULAR, AND AMBIENT

\[ \text{Pixel color} = k_a I_a + k_d I_{\text{max}} (0, N \cdot L) + k_s I_{\text{max}} (0, N \cdot H)^s \]
• Light has the property of **superposition**
  • I.e., effect caused by more than one light source = sum of effects from individual light sources
• So, to deal with multiple light sources, just sum all values up for diffuse and specular components (ambient light is only added once, however):

\[
\text{Pixel color} = k_d I_a + \sum_{i=1}^{n} I_i \left[ k_d \max(0, N \cdot L_i) + k_s \max(0, N \cdot H_i)^s \right]
\]
PROGRAMMING A RAY TRACER
BASIC ALGORITHM

• For the ray-tracer, the basic algorithm is as follows:
  • For each pixel
    • Compute viewing ray
    • If (ray hits an object with $t \geq 0$)
      • Compute $N$
      • Evaluate shading model and set pixel to that color
    • Else
      • Set pixel color to background color
As we’ve seen, there are a lot of different kinds of objects you could intersect with:

- Spheres, triangles, planes, polygons, etc.

**Good OOP-based approach:**

- Make generic class Surface
- Give Surface two functions:
  - "hit()" → returns whether a given ray hit the object
    - Also hands back some kind of "hit-record" (intersection point, normal, etc.)
  - "bounding-box()" → returns max extends of surface
    - Useful for optimization
- Have all other objects inherit from Surface and implement those two functions
- Then, you can just loop through one list of Surface instances
- Instead of separately going through your spheres, then your triangles, then...
• A Material class is a good idea as well
  • Have list of Materials that all objects share
  • Store a pointer in Surface class to appropriate Material
SHADOWS AND REFLECTIONS
ADVANTAGES OF RAY TRACERS

• One really neat thing about ray tracers → can add two additional effects very easily:
  • Shadows
  • Ideal Specular Reflections

• These are generally more difficult with object-order rendering
SHADOWS

• Once you get your intersection point:
  • Shoot ray (shadow ray) from intersection point to light
  • Loop through all objects to see if you hit something
  • If yes \(\rightarrow\) in shadow
  • Otherwise \(\rightarrow\) NOT in shadow

• Basically, you are repeating what you already have to do to fill in each pixel’s color 😊
SHADOWS

• **Two things to note:**
  1) You should add *ambient light* to the output pixel color, irrespective of whether the point is in shadow
  2) When shooting shadow ray ➔ usually start with $t > e$ (where $e$ is some tiny value)
    • Start at 0 ➔ might intersect with THE SAME SURFACE you’re already on! (round-off error)
IDEAL SPECULAR REFLECTIONS

• In the same way, **ideal specular reflections** (or **mirror reflections**) are straightforward to add:
  • Compute reflection vector \( R \):
    • Project \( D \) onto \( N \) \(\rightarrow\) subtract twice that vector to get \( R \)
    \[
    R = D - 2(D \cdot N)N
    \]
  • Check to see what object you hit with \( R \) and compute shading for that object \(\rightarrow\) assume it’s a function called “raycolor”
  • Use raycolor as part of output color:
    \[
    \text{color } c = c + k_m \text{raycolor}(P + sR)
    \]
    • ...where \( k_m \) = “mirror reflection” coefficient, since some surfaces will reflect some colors better than others
      • E.g., gold reflect yellow best
• *Three things to watch out for:*
  
  1) Same issue with shadows (don’t start at t = 0)
  2) Need to include some kind of stopping criteria → otherwise, keep bouncing around forever
  - E.g., limit number of times you can recursively call `raycolor()`
  3) For efficiency → don’t call `raycolor()` if $k_m = 0$
  - Not going to reflect anything anyway