CS 450/550 – Computer Graphics

• Ray Tracing
Ray Tracing

What is ray tracing?

- Follow (trace) the path of a ray of light and model how it interacts with the scene
- When a ray intersects an object, send off secondary rays (reflection, shadow, transmission) and determine how they interact with the scene
- Basic algorithm allows for:
  - Hidden surface removal
  - Multiple light sources
  - Hard shadows
- Extensions can achieve:
  - Soft shadows
  - Blurred reflections (glossiness)
  - Translucent refractions
  - Motion blur
  - Depth of field (finite apertures)
  - and more
Ray Tracing: Backward vs. Forward

- “Forward” ray tracing:
  - Traces the ray *forward* (in time) from the light source through potentially many scene interactions
  - Physically based
  - Global illumination model:
    - Color bleeding
    - Caustics
    - Etc.
  - Problem: most rays will never even get close to the eye
  - Very inefficient since it computes many rays that are never seen
“Backward” ray tracing:

- Traces the ray *backward* (in time) from the eye, through a point on the screen
- Not physically based
- Doesn’t properly model:
  - Color bleeding
  - Caustics
  - Other changes in light intensity and color due to refractions and non-specular reflections
- More efficient: computes only visible rays (since we start at eye)
- Generally, ray tracing refers to this ‘backward’ ray tracing
Ray Tracing

- Ray tracing is a image-precision algorithm: Visibility determined on a per-pixel basis
  - Trace one (or more) rays per pixel
  - Compute closest object (triangle, sphere, etc.) for each ray

- Produces realistic results
- Computationally expensive

1024×1024, 16 rays/pixel
~ 10 hours on a 99 MHz HP workstation
A basic (minimal) ray tracer is simple to implement:

- The code can even fit on a 3×5 card (code courtesy of Paul Heckbert with a small change to output as a PPM file):

```c
#define struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{vec cen,color;double rad,kd,ks,kt,kl,ir}*s,*best,spth[]={0.,6.,5.,1.,1.,1.,.9,.05,.2,.85,0.,1.7,-1.,8.,-.5,1.,5.,2.,1.,7.,3.,0.,.05,1.2,1.,8.,-.5,1.,8.,8.,1.,.3,.7,0.,0.,1.2,3.,-.6,15.,1.,8.1.,7.,0.,0.,0.,0.,.6,1.5,-3.,-3.,12.,8.1.,1.,5.,0.,0.,0.,.5,1.5,};yx;double u,b,tmin,sqrt(),tan();double vdot(A,B)vec A, B;{return A.x*B.x+A.y*B.y+A.z*B.z;};vec vcomb(a,A,B)double a;vec A,B;{B.x+=a*A.x;B.y+=a*A.y;B.z+=a*A.z;return B;};vec vunit(A)vec A;{return vcomb(1./sqrt(vdot(A,A)),A,black);}struct sphere*intersect(P,D)vec P,D;{best=0;for(s=spth; s-->spth);u=vdot(D,U=vcomb(-1.,P,s->cen)),u=b*b-vdot(U,U)+s->rad*s->rad,u=u>0?sqrt(u):1e31,u=b-u>1e-7?b-u:b+u,tmin=u>=1e-7&&u<tmin?best=s,u:tmin;return best;};vec trace(level,P,D)vec P,D;{double d,eta,e;vec N,color;struct sphere*s,*l;if(!level--)return black;if(s=intersect(P,D));else return amb;color=amb;eta=s->ir;d= -vdot(D,N=vunit(vcomb(-1.,P,vcomb(s->cen,D),s->cen))));if(d<0)N=vunit(vcomb(-1.,N,black),eta=1/eta,d= -d;l=spth+5;while(1--spth));if((e=1-l->kl*vdot(D,N=vunit(vcomb(-1.,P,vcomb(s->cen,D),s->cen))))>0&&intersect(P,U)==l)color=vcomb(e,N,vunit(color.x*U.x;color.y*U.y;color.z*U.z);e=1-eta*eta*(1-d*d);return vcomb(s->kt,e>0?trace(level,P,vcomb(eta,D,vcomb(eta*D-sqrt(e),N,black)))));black,vcomb(s->ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s->kd,color,vcomb(s->kl,U,black)))));main(){puts("P3n32 32\n255");while(yx<32*32)U.x=yx%32-32/2,U.z=32/(yx++)/32,U.y=32/2/tan(25/114.5915590261),U=vcomb(255.,trace(3,black,vunit(U)),black),printf("%.0f %.0f %.0f\n",U);}/*minray*/
```
Minimal Ray Tracer

• This code implements:

- **Multiple spheres** (with different properties)
- **Multiple levels of recursion:**
  - Reflections
- **Transparency:**
  - Refraction
- **One point light source:**
  - Hard shadows
- **Hidden surface removal**
- **Phong illumination model**
- It even has a comment

```c
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{
  vec cen,color;double rad,kd,ks,kt,kl,ir}*s,*best,sph[]={{0.,6.,1.,1.,9.,
  .05,2.85,0.,1.7,-1.,8.,-.5,1.,.5,2.,1.,7.,3.,.05,1.2,1.,8.,-.5,1.,8.,8.,
  1.,-3.,7.,0.,.0.,1.2,3.,-6.,15.,1.,8.,1.,7.,0.,0.,0.,6.,5.,1.5,3.,-3.,12.,1.5,
  1.,5.,0.,0.,0.,0.,5.,1.,};yx;double u,b,tmin,sqrt(),tan();double vdot(A,B)vec A
,B;{return A.x*B.x+A.y*B.y+A.z*B.z;}

struct sphere*intersect(P,D)vec P,D;{best=0;tmin=1e30;s=sph+5;while(s-->sph)
  b=vdot(D,U=vcomb(-1.,P,s->cen)),u=b*b-vdot(U,U)+s->rad*s->rad,u=u>0?sqrt(u):1e31,
  u=b-u>1e-7?b-u:b+u,tmin=u>=1e-7&&u<tmin?best=s,u:tmin;return best;}

vec trace(level,P,D)vec P,D;{double d,eta,e;vec N,color;
  struct sphere*s,*l;if(!level--)return black;if(s=intersect(P,D));else return
  amb;color=amb;eta=s->ir;d= -vdot(D,N=vunit(vcomb(-1.,P=vcomb(tmin,D,P),s->cen
  ))));if(d<0)N=vcomb(-1.,N,black),eta=1/eta,d= -d;1=sph+5;while(1-->sph)
  if((e=1
  ->l*sqrt(D,vcomb(-1.,P,1-->cen))))>0&intersect(P,D)==1)color=vcomb(e
  ,1,color,color),U=s->color;vec x=E.x,color.y=E.y,color.z=E.z;e=1-eta*
  eta*(1-d*d);return vcomb(s->kt,e>0?trace(level,P,vcomb(e,D,vcomb(e*eta*sqrt
  (e),N,black))):black,vcomb(s->ks,trace(level,P,vcomb(2*D,N,D)),vcomb(s->kd,
  vcomb(s->kl,U,black))));}main(){puts("P3\n32 32
255\n");while(yx<32*32)
  U.x=yx%32-32/2,U.z=32/2-yx++/32,U.y=32/2/tan(25/114.5915590261),U=vcomb(255.,
  trace(3,black,vunit(U)),black),printf("%.0f %.0f %.0f\n",U);}/minray!*/
```
Ray Tracing: Types of Rays

• Primary ray \( (P) \):
  - Sent from the eye, through the image plane, and into the scene
  - May or may not intersect an object in the scene:
    - No intersection $\rightarrow$ set pixel color to background color
    - Intersects object $\rightarrow$ send out secondary rays and compute lighting model

• Secondary rays (sent from the point where the ray intersects an object):
  - Transmission \( (T) \): sent in the direction of refraction
  - Reflection \( (R) \): sent in the direction of reflection
  - Shadow \( (S) \): sent toward a light source (determines if point is in shadow or not)
Ray Tracing: Types of Rays

- **S**: Shadow rays
- **R**: Reflected rays
- **T**: Transmitted rays

**Opaque object**

**Transparent object**

**Eye**

**Light**

\[ \begin{align*}
S_1 & \rightarrow \text{Shadow rays} \\
R_1 & \rightarrow \text{Reflected rays} \\
T_1 & \rightarrow \text{Transmitted rays}
\end{align*} \]
Ray Tracing: **Ray Tree**

- Each intersection may spawn secondary rays:
  - Rays form a ray tree
  - Nodes → Intersection points
  - Edges → Reflected/transmitted ray

- Rays are recursively spawned until:
  - Ray does not intersect any object
  - Tree reaches a maximum depth
  - Light reaches some minimum value

- Shadow rays are sent from every intersection point (to determine if point is in shadow), but they do not recursively spawn secondary rays
Ray tree is evaluated from bottom up:

- Depth-first traversal or recursive algorithm
- Each node’s color is calculated as a function of its children’s colors
A ray can be represented explicitly (in parametric form) as a origin (point) and a direction (vector):

- **Origin**:  \( r_o = \begin{bmatrix} x_o \\ y_o \\ z_o \end{bmatrix} \)

- **Direction**:  \( r_d = \begin{bmatrix} x_d \\ y_d \\ z_d \end{bmatrix} \)

The ray consists of all points:

\[
 r(t) = r_o + r_d t
\]
The primary ray (or viewing ray) for a point $s$ on the view plane is computed as:

- **Origin:** $r_o = \text{eye}$
- **Direction:** $r_d = s - \text{eye}$

**Which coordinate space?**

- Want to define rays in terms world-space coordinates $(x, y, z)$
- Eye is already in specified in terms of $(x, y, z)$ position
- How do we find $s$ on the view plane?
• Given a pixel \((i, j)\) in the viewport, compute the point \(s\) on the view plane (in viewing-space coordinates):

  ▪ Reverse the windowing transform:

\[
\begin{align*}
    u_s &= w_w \left( \frac{i + 0.5}{n_x} \right) - \frac{w_w}{2} \\
    v_s &= h_w \left( \frac{j + 0.5}{n_y} \right) - \frac{h_w}{2} \\
    w_s &= n
\end{align*}
\]

where \(n_x\) and \(n_y\) are the viewport’s width and height in pixels and \(w_w\) and \(w_h\) are the viewplane’s width and height, respectively.
Ray Computation

• Need a ray from the eye to this point!
• Must make sure it’s in the right space
• Options
  - Compute ray in eye space, then transform to world
  - Transform world to eye space, then compute ray in eye space
Given the screen point in terms of viewing-space coordinates \((u, v, w)\), transform to world-space \((x, y, z)\):

\[ s = \text{eye} + u_s u + v_s v + w_s w \]

\[
\begin{bmatrix}
  x_s \\
y_s \\
z_s \\
1
\end{bmatrix} =
\begin{bmatrix}
  1 & 0 & 0 & \text{eye}_x \\
  0 & 1 & 0 & \text{eye}_y \\
  0 & 0 & 1 & \text{eye}_z \\
  0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  u_x & v_x & w_x & 0 \\
  u_y & v_y & w_y & 0 \\
  u_z & v_z & w_z & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
u_s \\
v_s \\
w_s \\
1
\end{bmatrix}
\]
Ray-Object Intersections

• Basic (non-recursive) ray tracing algorithm:
  1. Send a ray from the eye through the screen
  2. Determine which object that ray first intersects
  3. Compute pixel color

• Most (approx. 75%) of the time in step 2:
  ▪ Simple method: Compare every ray against every object and remember the closest object hit by each ray
  ▪ Very time consuming: Several optimizations possible
Many objects can be represented as implicit surfaces:

- Sphere (with center at $c$ and radius $R$): $f_{sphere}(p) = \|p - c\|^2 - R^2 = 0$

- Plane (with normal $n$ and distance to origin $D$): $f_{plane}(p) = p \cdot n + D = 0$

To determine where a ray intersects an object:

- Need to find the intersection point $p$ of the ray and the object
- The ray is represented explicitly in parametric form: $r(t) = r_o + r_d t$
- Plug the ray equation into the surface equation and solve for $t$: $f(r(t)) = 0$
- Substitute $t$ back into ray equation to find intersection point $p$: $p = r(t) = r_o + r_d t$
To find the intersection points of a ray with a sphere:

- Substitute the ray equation into the sphere equation:

\[ f_{sphere}(p) = f_{sphere}(r(t)) = \|r_o + r_d t - c\|^2 - R^2 = 0 \]

- Simplifying (in terms of \( t \)), we get:

\[ A t^2 + B t + C = 0 \]

where

\[ A = (x_d^2 + y_d^2 + z_d^2) = 1 \]

\[ B = 2 \left( x_d (x_o - x_c) + y_d (y_o - y_c) + z_d (z_o - z_c) \right) \]

\[ C = (x_o - x_c)^2 + (y_o - y_c)^2 + (z_o - z_c)^2 - R^2 \]

- Use quadratic equation to solve for \( t \):

\[ t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} = \frac{-B \pm \sqrt{B^2 - 4C}}{2} \]

If \( r_d \) is a unit vector (i.e., normalized)
Algorithm for ray-sphere intersection:

1. Calculate $B$ and $C$ of the quadratic
2. Calculate the discriminant: $D = B^2 - 4C$
3. If $D \leq 0$ return false (no intersection point)
4. Calculate smaller intersection parameter $t_0$: $t_0 = \frac{-B - \sqrt{D}}{2}$
5. If $t_0 \leq 0$ then calculate larger $t$-value $t_1$: $t_1 = \frac{-B + \sqrt{D}}{2}$
6. If $t_1 \leq 0$ return false (intersection point behind ray)
7. else set $t = t_1$
8. else set $t = t_0$
9. Return intersection point: $p = r(t) = r_o + r_d t$
Ray-Sphere Intersections: **Normal**

The normal $\mathbf{n}$ at an intersection point $\mathbf{p}$ on a sphere is:

$$\mathbf{n} = \frac{\mathbf{p} - \mathbf{c}}{R} = \frac{1}{R} \begin{bmatrix} x - x_c \\ y - y_c \\ z - z_c \end{bmatrix}$$
Ray-Plane Intersections

To find the intersection points of a ray with an infinite extent plane (i.e., it is not bounded by a triangle or polygon):

- Substitute the ray equation into the plane equation:

\[ f_{\text{plane}}(p) = f_{\text{plane}}(r(t)) = (r_o + r_d t) \cdot n + d = 0 \]

- Solving for \( t \) we get:

\[ t = \frac{-(r_o \cdot n + d)}{r_d \cdot n} \]

- If \( t \leq 0 \) then intersection point is behind the ray (return false)

- Compute intersection point: \( p = r_o + r_d t \)
Point in Triangle Test
Ray Tracing: Basic Algorithm

• The basic ray tracing algorithm is:

```plaintext
for each pixel do
    compute viewing ray \( r = r_o + t \cdot r_d \)
    for each object
        if (ray hits an object with \( 0 \leq t < \infty \))
            compute normal, \( n \)
            evaluate illumination model and set pixel color
        else
            set pixel color to background color
```

Intersect each ray with every object (spheres, triangles, etc.)
Ray Tracing: Ray Intersection

- The test “if (ray hits object ...” can be implemented as:

```plaintext
hit = false
for each object obj do
    if (object is hit at ray parameter t and \( t_0 \leq t \leq t_1 \) then
        hit = true
        hitObj = obj
        \( t_1 = t \)
return hit

/* t1 initialized to a large number…closest hit point so far */
```
Shadows

Send a *shadow* ray from intersection point to the light:

- **Compute the following shadow ray properties:**
  - Shadow ray: \( s_d = (l - p) / \|l - p\| \)

- **Test if shadow ray intersects an object before reaching the light:**
  - Due to numerical error, test the shadow ray for \( t \in [\varepsilon, \infty] \)
Specular Reflection

- Reflection in same angle as light came in
- Light continues to bounce:

\[ r = d - 2n(d \cdot n) \]

Note the sign change

- Typically, some energy is lost on each bounce
Specular Reflection

- Implement specular reflection with a recursive call:
  \[
  \text{color} = \text{ambient} + \text{diffuse} + \text{specular} + c_s \text{reflectedColor}
  \]
  where \(c_s\) represents how “perfect” the mirror is and reflected color is the recursive call.

- Limit recursion: max depth or when the contribution of a ray is negligible.

- For efficiency, generate a reflection ray ONLY if this point is not in shadow:
  \[
  c = c + c_s \text{rayColor}(p + \text{spr}, \epsilon, \infty)
  \]
function rayColor(ray e + t*d, real t0, real t1)
if (scene->hit()) then  // Fill in hit object.
  p = e + rec.t*d       //compute the intersection point.
  col = ambient color;
  //Does shadow ray hit the scene anywhere?
  if (not scene->hit(p + s*l, ε, ∞)) then
    compute reflected vector, r
    col = phong model ()
    col = col + c_s*rayColor(p + sp * r, ε, ∞); // Compute reflected color.
  return col          // Either ambient, or full depending on if..
else return background color
Recursive Ray Tracing

• Basic ray tracing results in basic Phong illumination plus hidden surfaces

• Shadows require only one extra ray per light source
  ▪ Shadow rays do not reflect or refract
  ▪ No need to find the closest object, only need to hit once before reaching the light

• Reflection and refraction can spawn many new rays since light can keep bouncing!
Refraction (transparency)

- When an object is transparent, it transmits light.
- Light bends when moving from one medium to another according to Snell’s law:

\[ n_i \sin \theta = n_t \sin \phi \]

\[ \frac{\sin \theta}{\sin \phi} = \frac{n_t}{n_i} \]

Diagram:
- Light ray enters air from glass at angle \( \theta \).
- Refraction occurs at the boundary between air and glass.
- Angle of incidence \( \theta \) and angle of refraction \( \phi \) are related by Snell’s law.

Medium:
- Air
- Glass
### Refraction Indices

Index of refraction for various materials:

<table>
<thead>
<tr>
<th>Material</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vacuum</td>
<td>1.0</td>
</tr>
<tr>
<td>Air</td>
<td>1.0003</td>
</tr>
<tr>
<td>Water</td>
<td>1.33</td>
</tr>
<tr>
<td>Alcohol</td>
<td>1.36</td>
</tr>
<tr>
<td>Fused quartz</td>
<td>1.46</td>
</tr>
<tr>
<td>Crown glass</td>
<td>1.52</td>
</tr>
<tr>
<td>Flint glass</td>
<td>1.65</td>
</tr>
<tr>
<td>Sapphire</td>
<td>1.77</td>
</tr>
<tr>
<td>Heavy flint glass</td>
<td>1.89</td>
</tr>
<tr>
<td>Diamond</td>
<td>2.42</td>
</tr>
</tbody>
</table>
Refraction

Total internal reflection

- When going from a dense to less dense medium, the angle of refraction becomes more shallow.
- If the angle of incidence is shallow, it can get trapped in the dense material.
  - Optical cable
  - Diamonds
Demo

- Nvidia Kepler:
  http://www.youtube.com/watch?v=h5mRRElXy-w