Ray Tracing
Part I
What is ray tracing?

- Trace the path of a ray of light.
- Model its interactions with the scene.
- When a ray intersects an object, send off secondary rays (reflection, shadow, transmission) and determine how they interact with the scene.
Ray Tracing Capabilities

• Basic algorithm allows for:
  • Hidden surface removal (like z-buffering)
  • Multiple light sources
  • Reflections
  • Transparent refractions
  • Hard shadows

• Extensions can achieve:
  • Soft shadows
  • Motion blur
  • Blurred reflections (glossiness)
  • Depth of field (finite apertures)
  • Translucent refractions and more
Ray Tracing

• Produces realistic images

• Strengths:
  • Specular reflections
  • Transparency

• Weaknesses:
  • Color bleeding (diffuse reflections)
  • Time consuming

• References:
Ray Traced Images
“Backward” ray tracing:
- Traces the ray forward (in time) from the light source through potentially many scene interactions
- Problem: most rays will never even get close to the eye
- Very inefficient since it computes many rays that are never seen
Ray Tracing

• “Forward” ray tracing:
  • Traces the ray \textit{backward (in time)} from the eye, through a point on the screen
  • More efficient: computes only visible rays (since we start at eye)
  • Generally, ray tracing refers to \textit{forward} ray tracing.
Ray Tracing

• Ray tracing is an 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
Minimal Ray Tracer

• A basic (minimal) ray tracer is simple to implement:
  • The code can even fit on a 3×5 card (code courtesy of Paul Heckbert):

```c
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{
  vec cen,color;double rad,ks,kt,kl,ir}*s,*best,sph[]={0,.6,.5,1.,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.,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;tmin=1e30;s=
  sph+5;while(s-->sph)b=vdot(D,U=vcomb(-1.,P,vcomb(s,cen))),u=b*b-vdot(U,U)+s-
  rad*u>1e7?b-u:b+u,tmin=u>=1e7&&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-
  kl*vdot(N,U=vunit(vcomb(-1.,P,1->cen))))>0&intersect(P,U)==1)color=vcomb(e
  ,1->color,color);U=s-->color;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("P3
n32 32
n255");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!*/
```
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

```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.,.5,1.,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,.1,.5,.3,-3,.12,.8,.1,.1,.5,.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;tmin=1e30;s=sph+5;while(s--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-b>1e-7?b-b: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;
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-kl*vdot(N,U=vunit(vcomb(-1.,P,1->cen))))>0&&intersect(P,U=l))color=vcomb(e,l->color,color,x=x=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(ea,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=32/2-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 rays:
  - Sent from the eye, through the image plane, and into the scene
  - May or may not intersect an object in the scene.
    - No intersection: set pixel to background color
    - Intersects object: send out secondary rays and compute lighting model
Ray Tracing: Types of Rays

- **Secondary Rays:**
  - Sent from the point at which the ray intersects an object

- **Multiple types:**

  **Transmission (T):** sent in the direction of refraction

  **Reflection (R):** sent in the direction of reflection, and used in the Phong illumination model

  **Shadow (S):** sent toward a light source to determine if point is in shadow or not.
Ray Tracing: Types of Rays

- **Eye**
- **Light**
- **Opaque object**
- **Transparent object**

\[ P = \text{Primary rays} \]
\[ R = \text{Reflected rays} \]
\[ T = \text{Transmitted rays} \]
\[ S = \text{Shadow rays} \]
Ray Tracing: Ray Tree

- Each intersection may spawn secondary rays:
  - Rays form a ray tree.
  - Nodes are the intersection points.
  - Edges are the secondary rays.
- 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 spawn additional rays.
Ray Tracing: Ray Tree Example

Ray tree is evaluated from bottom up:

• Depth-first traversal
• The node color is computed based on its children’s colors.
Basic Ray Tracing Algorithm

• Generate one ray per pixel.
• For each ray:
  • Find the first object the ray intersects.
  • Compute color for the intersection point using an illumination model.
  • If the surface is reflective, trace a reflection ray.
  • If the surface is transparent, trace a transmission ray.
  • Trace shadow ray.
  • Combine results of the intensity computation, reflection, transmission, and shadow information.
  • If the ray misses all objects, set to the background color.
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 determine the closest object hit by each ray.
• Very time consuming:
  • Several optimizations possible.
Ray Representation

• A ray can be represented explicitly (in parametric form) as an origin (point) and a direction (vector):

  • Origin: \[ \mathbf{r}_o = \begin{bmatrix} x_o \\ y_o \\ z_o \end{bmatrix} \]

  • Direction: \[ \mathbf{r}_d = \begin{bmatrix} x_d \\ y_d \\ z_d \end{bmatrix} \]

• The ray consists of all points:
  \[ \mathbf{r}(t) = \mathbf{r}_o + \mathbf{r}_d t \]
Viewing Ray

• The primary ray (or viewing ray) for a point $s$ on the view plane (i.e., screen) 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
  • Screen point $s$ is easiest to define in terms of where it is on the window in viewing-space coordinates $(u, v, w)$
Viewing Ray: Screen Point

• Given:
  • Our scene in world-coordinates
  • A camera position in world-coordinates \((x, y, z)\)
  • A pixel \((i, j)\) in the viewport

• We need to:
  • Compute the point on the view plane window that corresponds to the \((i, j)\) point in the viewport
  • Transform that point into world-coordinates
View-reference coordinates
View-reference Window

View reference coordinates

Window

LookAt point

(u_{\text{max}}, v_{\text{max}})

LookFrom point

(u_{\text{min}}, v_{\text{min}})
Viewport

\[ (i_{\text{min}}, j_{\text{min}}) \]

\[ (i, j) \]

\[ (i_{\text{max}}, j_{\text{max}}) \]
Computing Window Point

- Step 1: Reverse the Window-to-Viewport transformation
Viewport-Window transform

- **Window-viewport:**

\[
i = (u - u_{\text{min}}) \left( \frac{i_{\text{max}} - i_{\text{min}}}{u_{\text{max}} - u_{\text{min}}} \right) + i_{\text{min}}
\]

\[
 j = (v - v_{\text{min}}) \left( \frac{j_{\text{max}} - j_{\text{min}}}{v_{\text{max}} - v_{\text{min}}} \right) + j_{\text{min}}
\]

- **Inverse transform (viewport-window)**

\[
 u = (i - i_{\text{min}}) \left( \frac{u_{\text{max}} - u_{\text{min}}}{i_{\text{max}} - i_{\text{min}}} \right) + u_{\text{min}}
\]

\[
 v = (j - j_{\text{min}}) \left( \frac{v_{\text{max}} - v_{\text{min}}}{j_{\text{max}} - j_{\text{min}}} \right) + v_{\text{min}}
\]

\[
w = 0
\]
View-reference to World transform

• Given the screen point in terms of viewing-space coordinates \((u, v, w)\), transform to world-space \((x, y, z)\):
  • The viewing transform takes a point from world space to view space:

\[
M_v = M_{CoB} T = \begin{bmatrix}
    u_x & u_y & u_z & 0 \\
    v_x & v_y & v_z & 0 \\
    w_x & w_y & w_z & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
    1 & 0 & 0 & -\text{LookAt}_x \\
    0 & 1 & 0 & -\text{LookAt}_y \\
    0 & 0 & 1 & -\text{LookAt}_z \\
    0 & 0 & 0 & 1
\end{bmatrix}
\]
View-reference to World transform

- Given the screen point in terms of viewing-space coordinates \((u, v, w)\), transform to world-space \((x, y, z)\):
  - We want to reverse this:

\[
\mathbf{s}_{\text{World}} = \mathbf{M}_v^{-1} \begin{bmatrix} u_s \\ v_s \\ w_s \\ 1 \end{bmatrix} = \mathbf{T}^{-1} \mathbf{M}_{\text{CoB}}^{-1} \begin{bmatrix} u_s \\ v_s \\ w_s \\ 1 \end{bmatrix}
\]

\[
\begin{bmatrix} u_x & v_x & w_x & \text{LookAt}_x \\ u_y & v_y & w_y & \text{LookAt}_y \\ u_z & v_z & w_z & \text{LookAt}_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_s \\ v_s \\ w_s \\ 1 \end{bmatrix}
\]

or \(\mathbf{s}_{\text{World}} = \mathbf{LookAt} + u_s \mathbf{u} + v_s \mathbf{v} + w_s \mathbf{w}\)