



Ray Tracing

- What is it?
- Why use it?
- Basics
- Advanced topics
- References



Ray-Tracing: Why Use It?

- Simulate rays of light
- Produces natural lighting effects
 - Reflection
 - Refraction
 - Soft Shadows
 - Depth of Field
 - Motion Blur
 - Caustics



Ray-Tracing: Why Use It?

- Hard to simulate effects with rasterization techniques (OpenGL)
- Rasterizers require many passes
- Ray-tracing easier to implement



Ray-Tracing: Who Uses It?

- Entertainment (Movies, Commercials)
- Games pre-production
- Simulation



Ray-Tracing: History

- Decartes, 1637 A.D. - analysis of rainbow
- Arthur Appel, 1968 - used for lighting 3D models
- Turner Whitted, 1980 - "An Improved Illumination Model for Shaded Display" really kicked everyone off.
- 1980-now - Lots of research



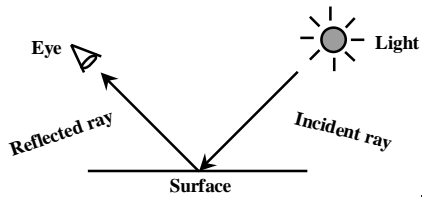
The Basics

- Generating Rays
- Intersecting Rays with the Scene
- Lighting
- Shadowing
- Reflections



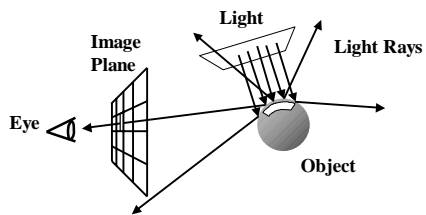
The Basic Idea

- Simulate light rays from light source to eye



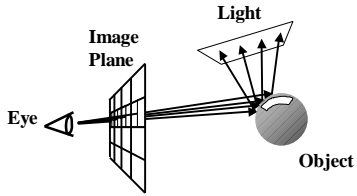
“Forward” Ray-Tracing

- Trace rays from light
- Lots of work for little return



“Backward” Ray-Tracing

- Trace rays from eye instead
- Do work where it matters



This is what most people mean by "ray tracing".

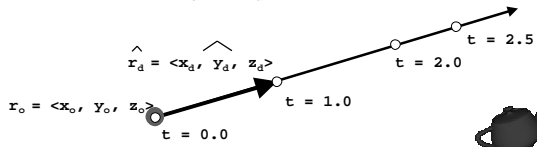


Ray Parametric form

- Ray expressed as function of a single parameter (“t”)

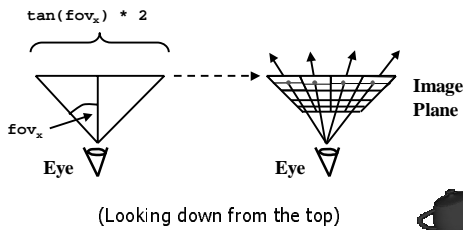
$$\langle x, y, z \rangle = \langle x_o, y_o, z_o \rangle + t * \langle x_d, y_d, z_d \rangle$$

$$\langle x, y, z \rangle = r_o + tr_d$$



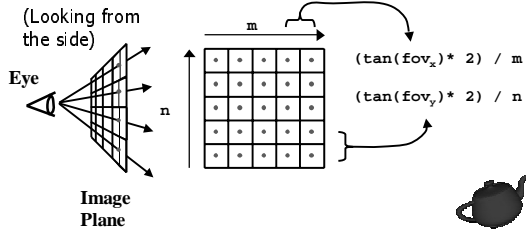
Generating Rays

- Trace a ray for each pixel in the image plane



Generating Rays

- Trace a ray for each pixel in the image plane



Generating Rays

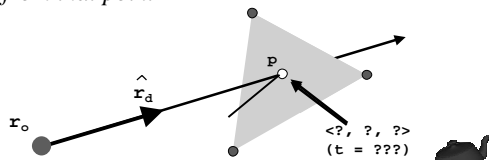
- Trace a ray for each pixel in the image plane

```
renderImage(){
  for each pixel i, j in the image
    ray.setStart(0, 0, 0); // r_o
    ray.setDir ((.5 + i) * tan(fov_x) * 2 / m,
               (.5 + j) * tan(fov_y) * 2 / n,
               1.0); // r_d
    ray.normalize();
    image[i][j] = rayTrace(ray);
}
```



Triangle Intersection

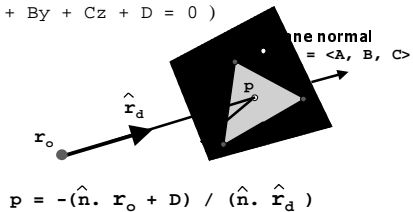
- Want to know: at what *point* (p) does ray intersect triangle?
- Compute lighting, reflected rays, shadowing from that point



Triangle Intersection

- Step 1 : Intersect with plane

$$(Ax + By + Cz + D = 0)$$

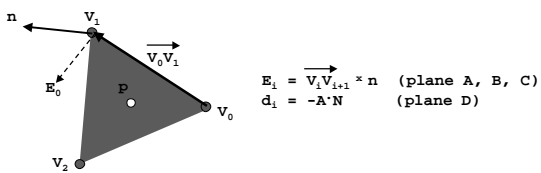


$$p = -(\hat{n} \cdot r_o + D) / (\hat{n} \cdot \hat{r}_d)$$



Triangle Intersection

- Step 2 : Check against triangle edges



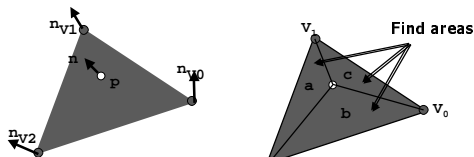
Plug p into $(p \cdot E_i + d_i)$ for each edge

if signs are all positive or negative, point is inside triangle!



Triangle Normals

- Could use plane normals (flat shading)
- Better to interpolate from vertices



$$n = \frac{a \hat{n}_{v_0} + b \hat{n}_{v_1} + c \hat{n}_{v_2}}{\text{area}(v_0, v_1, v_2)}$$



Finding Intersections

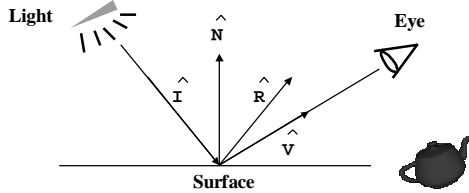
- Check all triangles, keep the closest intersection

```
hitObject(ray) {  
  for each triangle in scene  
    does ray intersect triangle?  
    if(intersected and was closer)  
      save that intersection  
  if(intersected)  
    return intersection point and normal  
}
```



Lighting

- We'll use triangles for lights
 - Build complex shapes from triangles
- Some lighting terms



Lighting

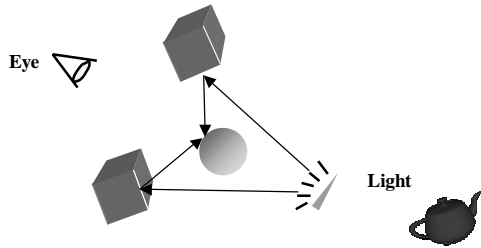
- Use modified Phong lighting
 - similar to OpenGL
 - simulates rough and shiny surfaces

```
for each light  
   $I_n = I_{ambient}K_{ambient} +$   
     $I_{diffuse}K_{diffuse} (L \cdot N) +$   
     $I_{specular}K_{specular} (R \cdot V)^n$ 
```



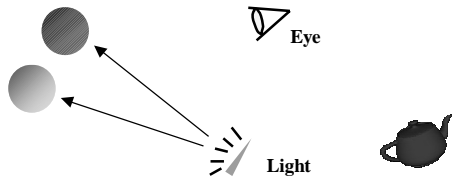
Ambient Light

- I_{ambient} Simulates the indirect lighting in a scene.



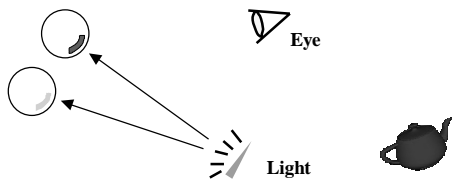
Diffuse Light

- I_{diffuse} simulates direct lighting on a rough surface
- Viewer independent
- Paper, rough wood, brick, etc...



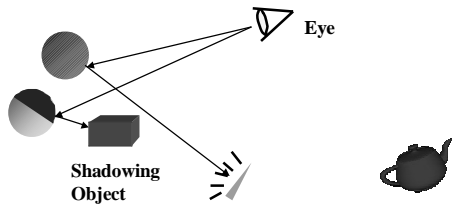
Specular Light

- I_{specular} simulates direct lighting on a smooth surface
- Viewer dependent
- Plastic, metal, polished wood, etc...



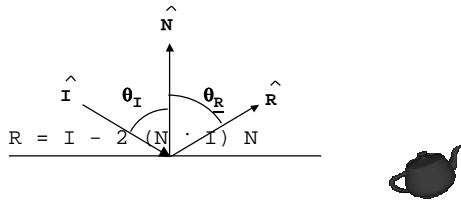
Shadow Test

- Check against other objects to see if point is shadowed



Reflection

- Angle of incidence = angle of reflection ($\theta_I = \theta_R$)
- \vec{I} , \vec{R} , \vec{N} lie in the same plane



Putting It All Together

- Recursive ray evaluation

```
rayTrace(ray) {  
  hitObject(ray, p, n, triangle);  
  color = object color;  
  if(object is light)  
    return(color);  
  else  
    return(lightning(p, n, color));  
}
```



Putting It All Together

- Calculating surface color

```
lighting(point) {  
  color = ambient color;  
  for each light  
    if(hitObject(shadow ray))  
      color += lightcolor *  
        dot(shadow ray, n);  
  color += rayTrace(reflection) *  
    pow(dot(reflection, ray), shininess);  
  return(color);  
}
```



Putting It All Together

- The main program

```
main() {  
  triangles = readTriangles();  
  image = renderImage(triangles);  
  writeImage(image);  
}
```



This is A Good Start

- Lighting, Shadows, Reflection are enough to make some compelling images
- Want better lighting and objects
- Need more speed



More Quality, More Speed

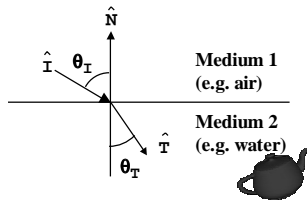
- Better Lighting + Forward Tracing
- Texture Mapping
- Modeling Techniques
- Motion Blur, Depth of Field, Blurry Reflection/Refraction
 - *Distributed Ray-Tracing*
- Improving Image Quality
- Acceleration Techniques



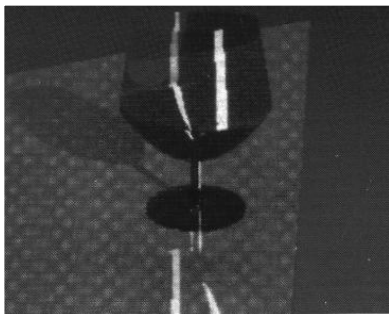
Refraction

- Keep track of medium (air, glass, etc)
- Need *index of refraction* (η)
- Need solid objects

$$\frac{\sin(\theta_I)}{\sin(\theta_T)} = \frac{\eta_1}{\eta_2}$$



Refraction



Improved Light Model

- Cook & Torrance
 - Metals have different color at angle
 - Oblique reflections leak around corners
 - Based on a microfacet model

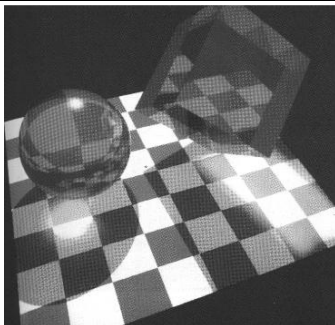


Using “Forward” Ray Tracing

- Backward tracing doesn't handle indirect lighting too well
- To get *caustics*, trace forward and store results in texture map.



Using “Forward” Ray Tracing



Texture Mapping

- Use texture map to add surface detail
 - Think of it like texturing in OpenGL
- Diffuse, Specular colors
- Shininess value
- Bump map
- Transparency value



Texture Mapping



Parametric Surfaces

- More expressive than triangle
- Intersection is probably slower
- u and v on surface can be used as texture s, t



Constructive Solid Geometry

- Union, Subtraction, Intersection of solid objects



- Have to keep track of intersections



Hierarchical Transformation

- Scene made of parts
- Each part made of smaller parts
- Each smaller part has transformation linking it to larger part
- Transformation can be changing over time - Animation

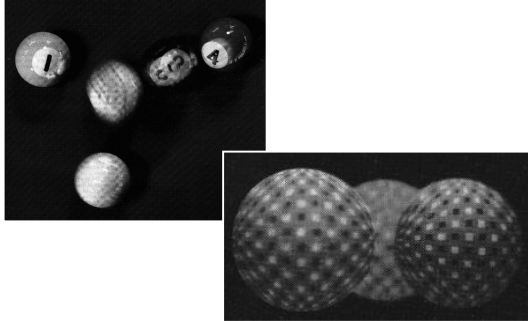


Distributed Ray Tracing

- Average multiple rays instead of just one ray
- Use for both shadows, reflections, transmission (refraction)
- Use for motion blur
- Use for depth of field



Distributed Ray Tracing



Distributed Ray Tracing

- One ray is not enough (jaggies)
- Can use multiple rays per pixel - *supersampling*
- Can use a few samples, continue if they're very different - *adaptive supersampling*
- Texture interpolation & filtering



Acceleration

- 1280x1024 image with 10 rays/pixel
- 1000 objects (triangle, CSG, NURBS)
- 3 levels recursion

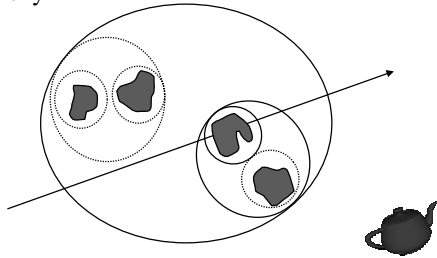
39321600000 intersection tests
100000 tests/second -> 109 days!

Must use an acceleration method!



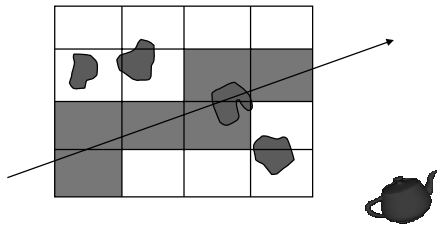
Bounding volumes

- Use simple shape for quick test, keep a hierarchy



Space Subdivision

- Break your space into pieces
- Search the structure linearly



Parallel Processing

- You can always throw more processors at it.



Really Advanced Stuff

- Error analysis
- Hybrid radiosity/ray-tracing
- Metropolis Light Transport
- Memory-Coherent Ray-tracing



References

- *Introduction to Ray-Tracing*, Glassner et al, 1989, 0-12-286160-4
- *Advanced Animation and Rendering Techniques*, Watt & Watt, 1992, 0-201-54412-1
- *Computer Graphics: Image Synthesis*, Joy et al, 1988, 0-8186-8854-4
- SIGGRAPH Proceedings (All)