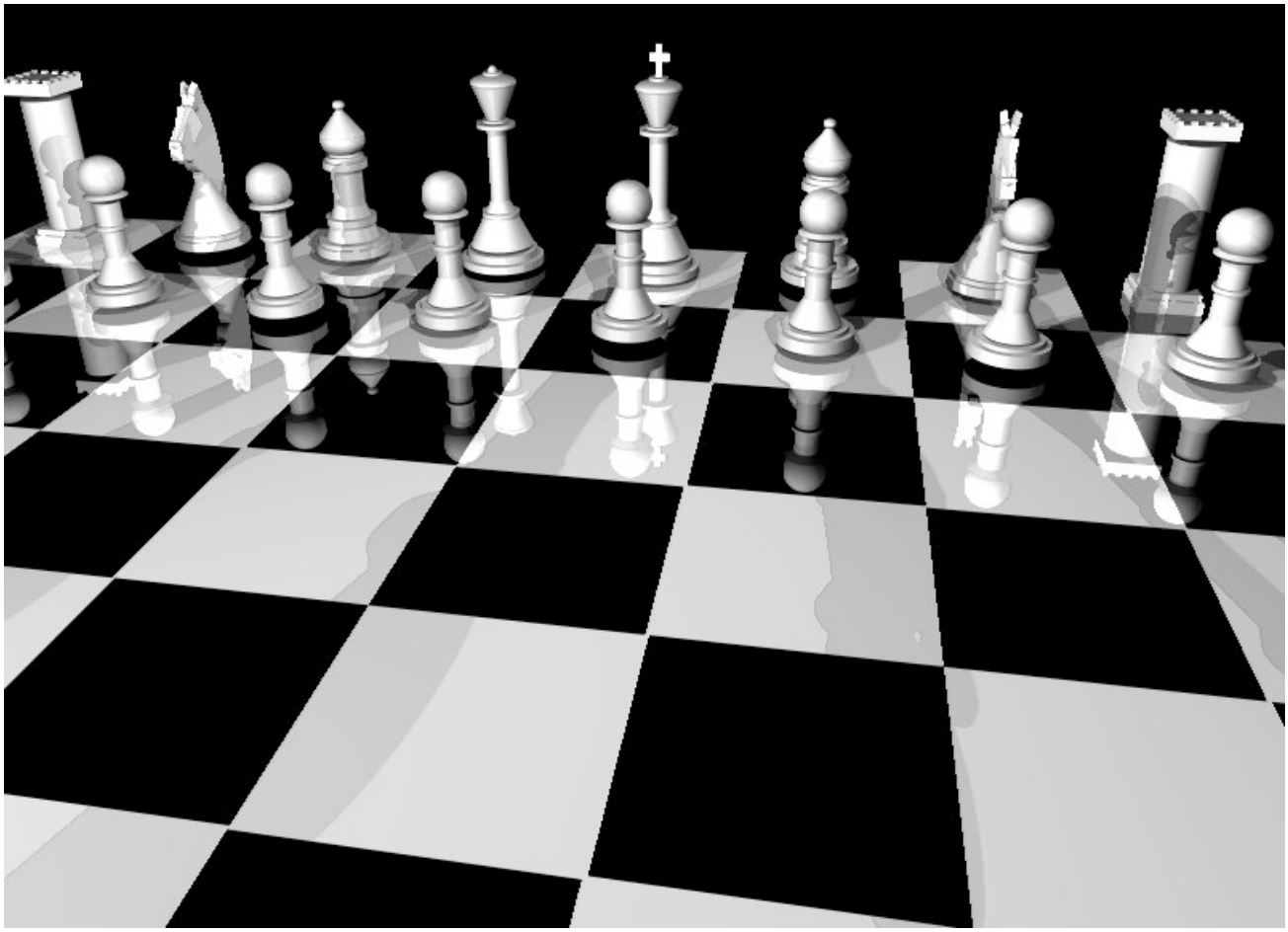
Raytracing



Hari Khalsa

Thanks: John Hughes, John Alex, David Mount

Final Project Proposals: due 11/14

Submit via wiki by 10am:

- Title
- Team
- One paragraph description of final project
- Timeline and deliverables
- Work distribution (if team)

Notes:

- No solar systems, ray-tracers, cloth simulations
- Adrian needs to OK your proposal
- Not meeting the deadline (or submitting a lousy proposal) will cost points

Example (D. Oliphant & C. Henne)

CS1566 Final Project Proposal

Nov 20th 2010

Networked Game Engine
Chris Henne and Dan Oliphant

It's going to be essentially a rudimentary game-engine. It will load an arbitrary number of wavefront .obj files and any associated textures into a scene. Then it will take care of collision detection and user input so that you can move, climb, and jump about a scene. To top it off, it will be multiplayer over a network connection, so you can explore these Blender-created worlds with a friend or two.

We reckon we'll have two weeks to complete this. The work distribution will be thus:

Dec 1st : Dan will make it load the .obj files and their associated textures. Chris will tackle the daunting task of creating both the server and the client sides of the network interface.

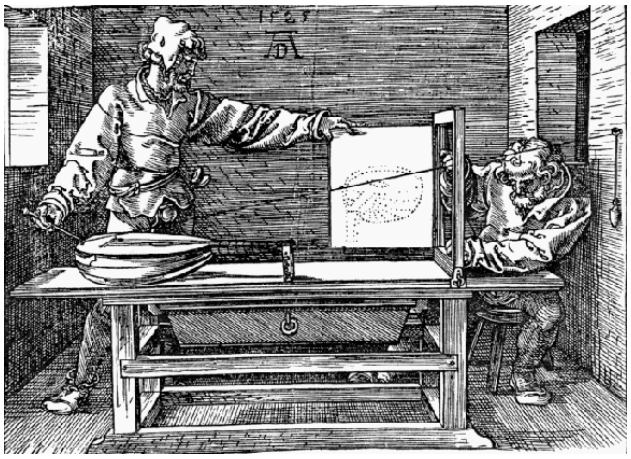
Dec 8th: Dan will make the collision detection happen using a ray-plane intersection technique. Chris will demo networked worlds.

Dec 15th:Dan will define the user controls to allow the user to walk, jump, climb, and explore these areas. Chris will take care of the synchronization and connection issues inherent in such a product. We will demo the project.

Rendering with Raytracing (1/2)

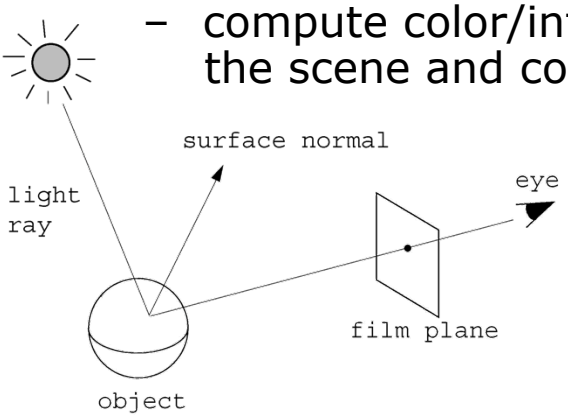
Simple idea: a ray-traced scene is a "virtual photo" comprised of many samples (pixels) on film plane;

Instead of one ray, imagine thousands of rays being shot from the eye through diff. points on film plane



- Durer: perspective projection

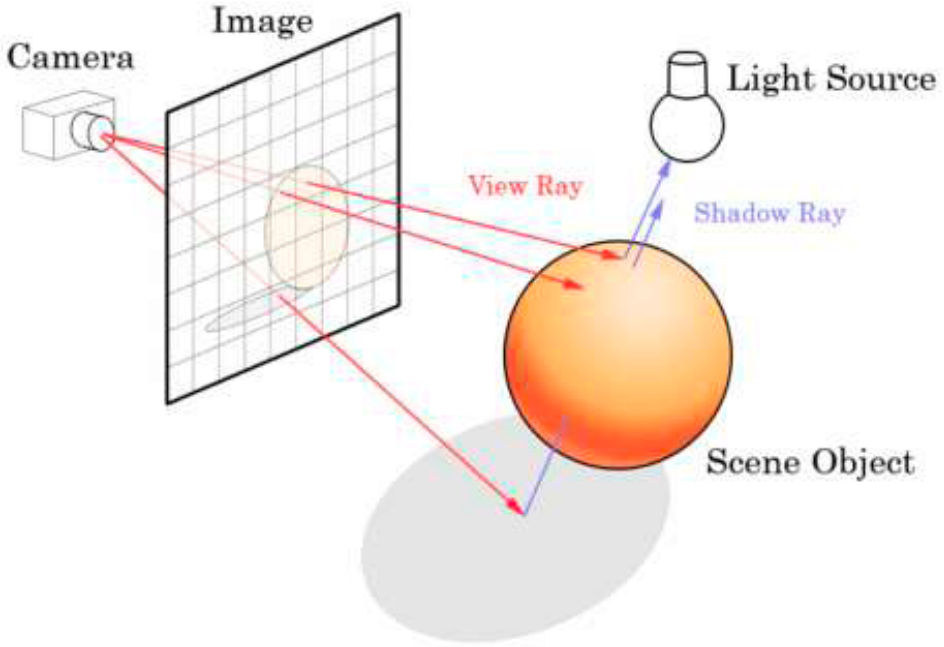
- Raytracing: shoot rays from eye through sample point (e.g., a pixel center) of a virtual photo (the image or film plane/screen)
  - compute color/intensity at the ray intersection with the scene and copy that color to the pixel



Rendering with Raytracing (2/2)

Subproblems to solve

1. Generate primary ('eye') ray
   - ray goes out from eye through a pixel center (or any other sample point on image plane)
2. Find closest object along ray path
   - find first intersection between ray and an object in scene
3. Light sample
   - use illumination model to determine direct contribution from light sources
   - specular (aka reflective objects) recursively generate secondary rays
     that also contribute to the color



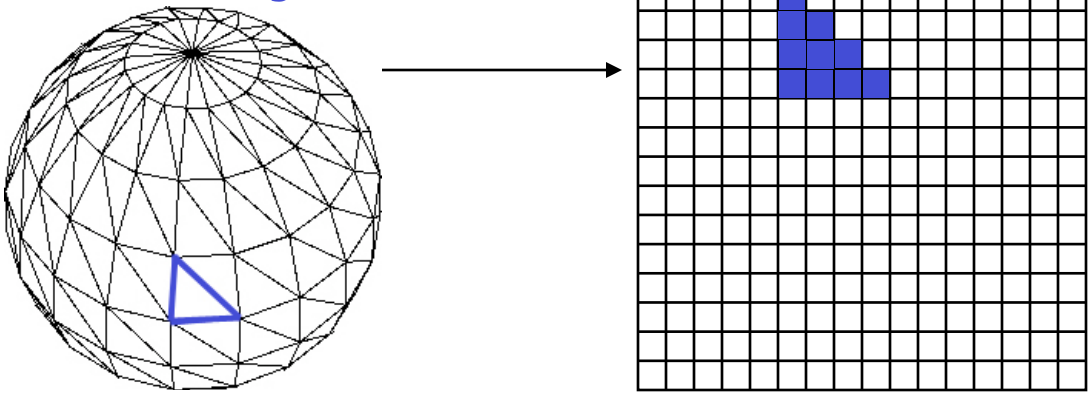
Raytracing

Raytracing versus scan conversion

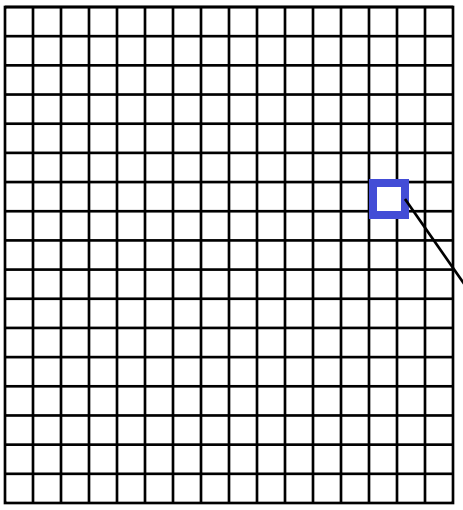
scan conversion

- After meshing and projection

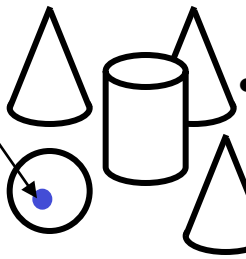
for each triangle in scene…



raytracing



for each sample
in pixel image…



- Could avoid meshing
- Avoid explicit projection of triangles by working directly with analytical surfaces

Raytracing

Raytracing versus scan conversion

- How is raytracing different from what you've been doing in your assignments? In Stitcher, Modeler you did:

```
for each object in scene
    for each triangle in the object
        pass vertex geometry, colors to
        OpenGL, which renders all interior
        points of triangle in
        framebuffer using the default shader
```

- This is fine if you're just using the simple hardware lighting model, and just using triangles

- In Intersect we started solving a different problem:

```
for each sample in our image
    - determine which object in scene
    is hit by ray through that sample;
    - paint that sample the color of
    the object at that point
    (accounting for lighting)
```

Summary

Simple, non-recursive raytracer

```
P = eyePt

for each sample of image

   Compute d

   for each object
     Intersect ray P+td with object

   Select object with smallest non-
   negative t-value (visible object)

   For this object, find object-space
   intersection point

   Compute normal at that point
   Transform normal to world space

   Use world-space normal for lighting
   computations
```

Shadows

- Each light in the scene contributes to the color of a surface element (sum over numLights):

$$objectIntensity_\lambda = ambient + \sum attenuation \cdot lightIntensity_\lambda \cdot [diffuse + specular]$$

- IFF the light reaches the object!
  - Construct ray from surface intersection to each light
  - If first object intersected in light, count light's full contributions
  - If first object intersected is not light, ignore light's contribution (i.e., count as shadow)

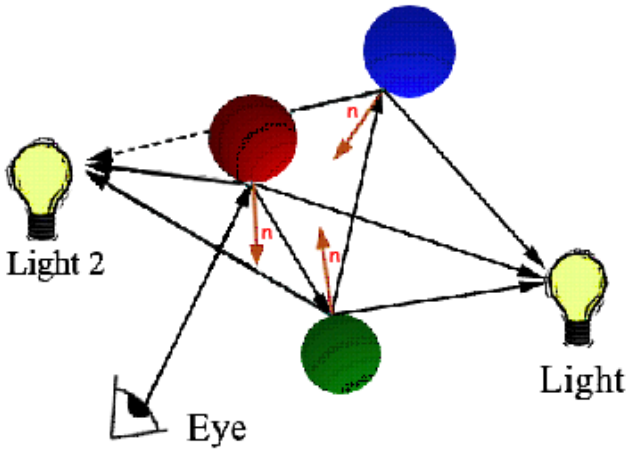
- This method causes hard shadows; soft shadows are harder to compute (must sample)

- What about transparent or specular (reflective) objects?
- Such lighting contributions mark the beginning of global illumination => need recursive ray tracing

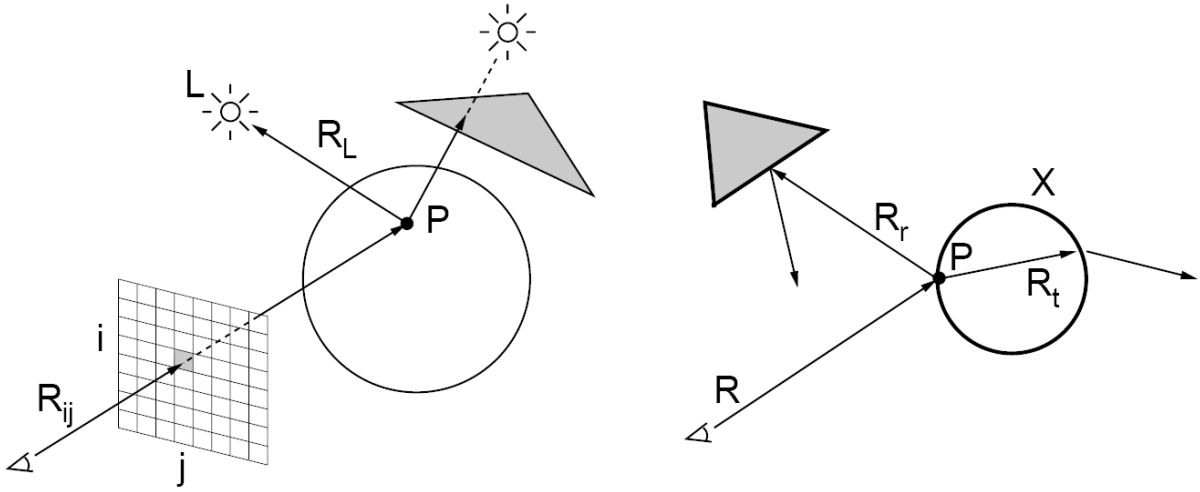
Recursive Raytracing (1/2)

Simulating global lighting effects (Whitted, 1979)

- By recursively casting new rays into scene, we can look for more information
- Start from point of intersection
- We'd like to send rays in all directions, but that's too hard/computationally taxing
- Send rays in directions likely to contribute most:
  - toward lights (blockers to lights create shadows for those lights)
  - specular bounce off other objects to capture specular inter-object reflections
  - use ambient hack to capture diffuse inter-object reflection
  - through object (transparency)



Recursive Raytracing (2/2)



- **raytrace():**
  foreach column i
    foreach row j
      - make ray Rij from eye to pixel i,j
      - pixcolor = **trace(Rij)**
      - set pixel i,j to pixcolor
- **trace(R):**
  - Find intersection point P, where R hits nearest object
  - If object is reflective: make reflection ray Rr; Colorl = **trace(Rr)**
  - If object is transparent: make refraction ray Rt; Colorr = **trace(Rt)**
  - Foreach light source: make ray RL from P to source
    - if object unobstructed from light source, use lighting model to find color Colorm
  - Return the combined color of: colorl, colorr, colorm

Recursive Raytracing

Recap

- Trace 'secondary' rays at intersections:
  - light: trace ray to each light source. If light source blocked by opaque object, it does not contribute to lighting
  - specular reflection: trace reflecting ray in mirror direction
  - refractive transmission/transparency: trace ray in refraction direction by Snell's law
  - recursively spawn new light, reflection, refraction rays at each intersection until contribution negligible / max recursion depth reached

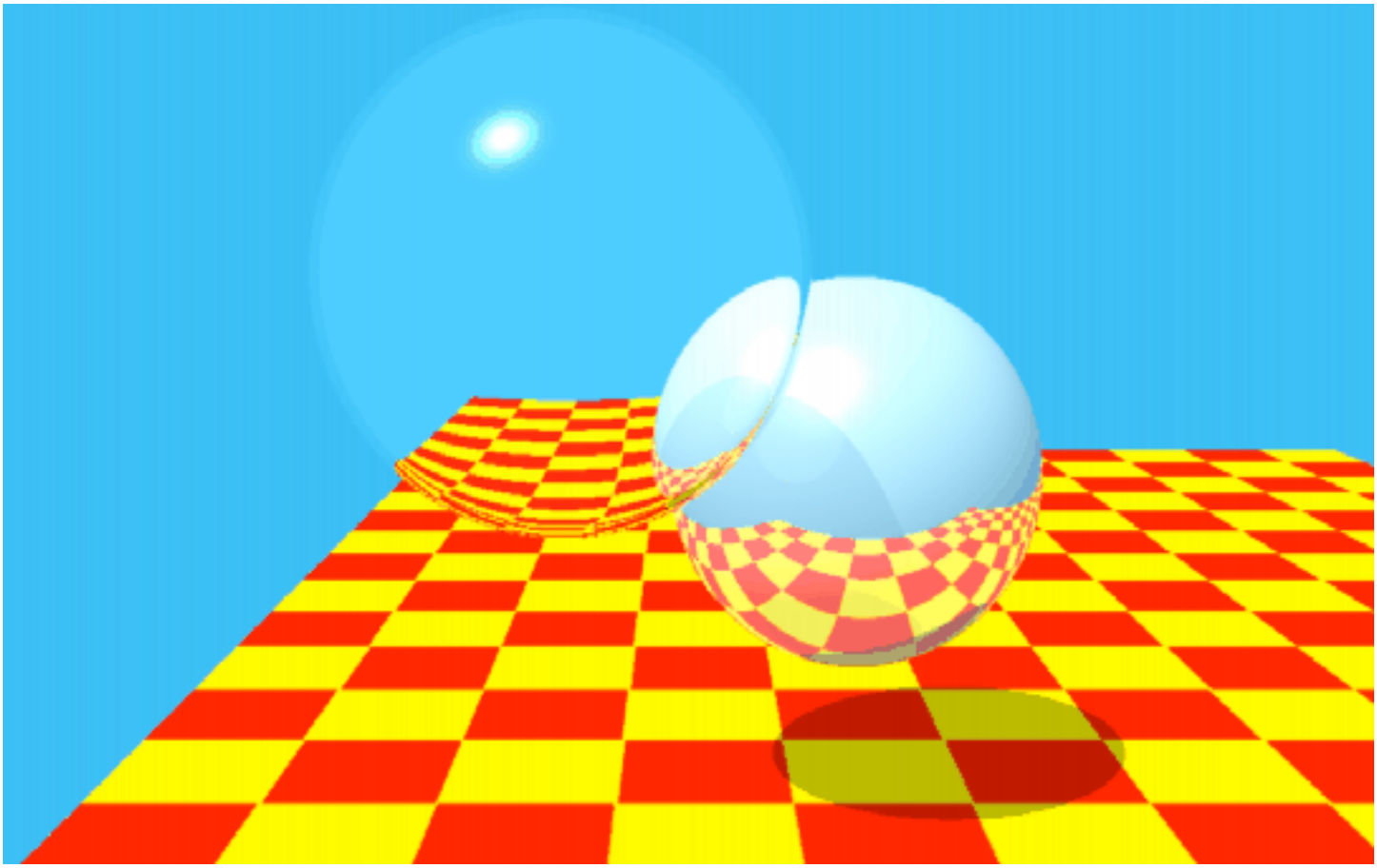
- Our new lighting equation…

$$I_\lambda = \underbrace{I_a \lambda_{ka} O_{d\lambda}}_{ambient} + \sum_m f_{att} I_{p\lambda} [\underbrace{k_d O_{d\lambda} \vec{N} \bullet \vec{L}}_{diffuse} + \underbrace{k_s O_{s\lambda} (\vec{R} \bullet \vec{V})^n}_{specular}] + \underbrace{\underbrace{k_s I_{s\lambda}}_{reflected} + \underbrace{k_t I_{t\lambda}}_{transmitted}}_{recursive}$$

  - note: intensity from recursed rays calculated with same eqn
  - light sources contribute specular and diffuse lighting

- Limitations
  - recursive inter-object reflection is strictly specular

Whitted 1980



- A ray traced image with recursive ray tracing, transparency and refractions

Transparent Surfaces (1/2)

Nonrefractive transparency

- For a partially transparent polygon

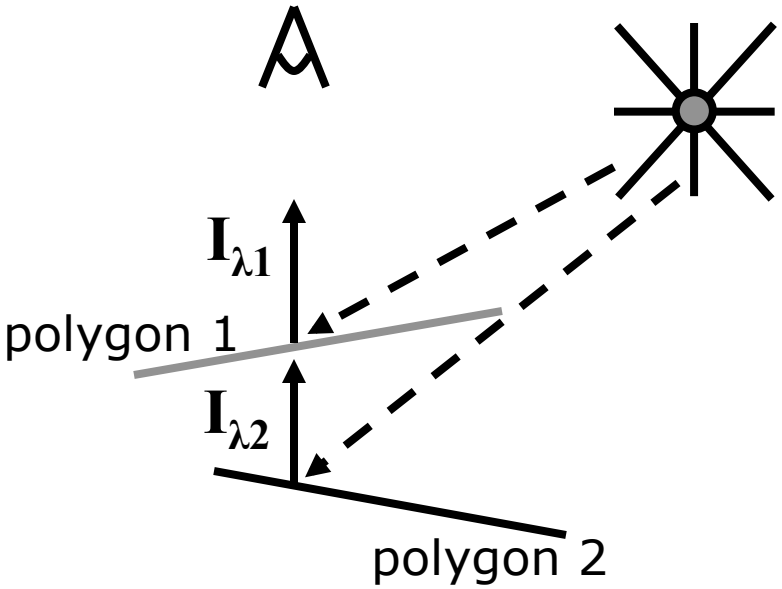
$$I_\lambda = (1 - k_{t1})I_{\lambda 1} + k_{t1}I_{\lambda 2}$$

k_{t1} = transmittance of polygon 1

$(0 = \text{opaqe}; 1 = \text{transparent})$

$I_{\lambda 1}$ = intensity calculated for polygon 1

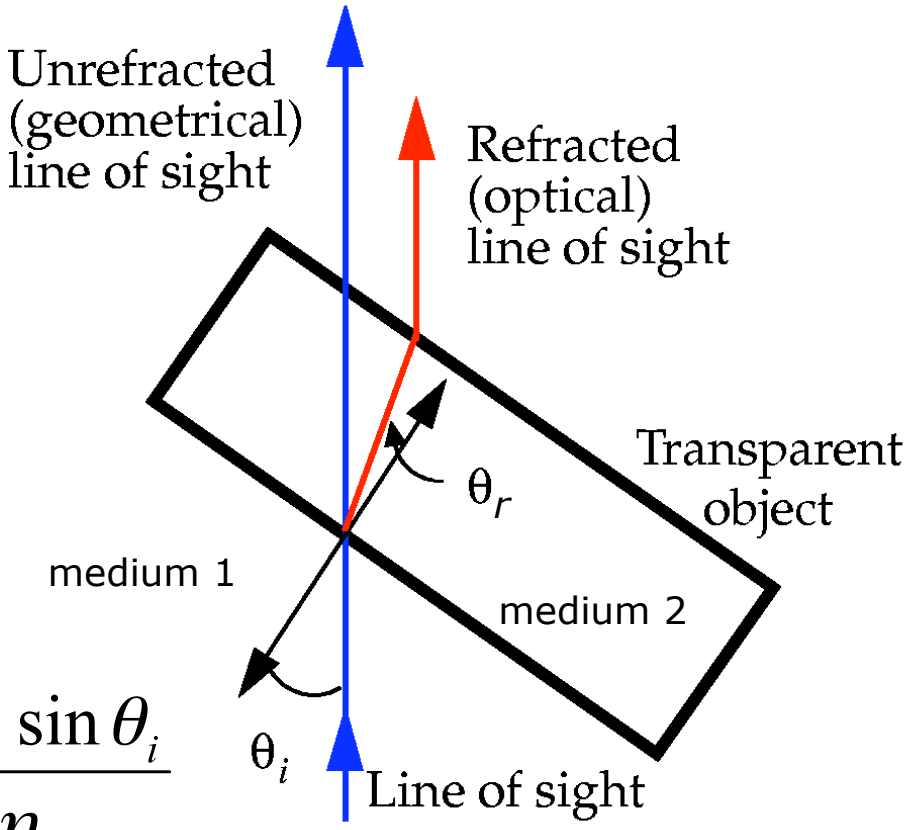
$I_{\lambda 2}$ = intensity calculated for polygon 2



Transparent Surfaces (2/2)

Refractive transparency

- We model the way light bends at interfaces with Snell's Law

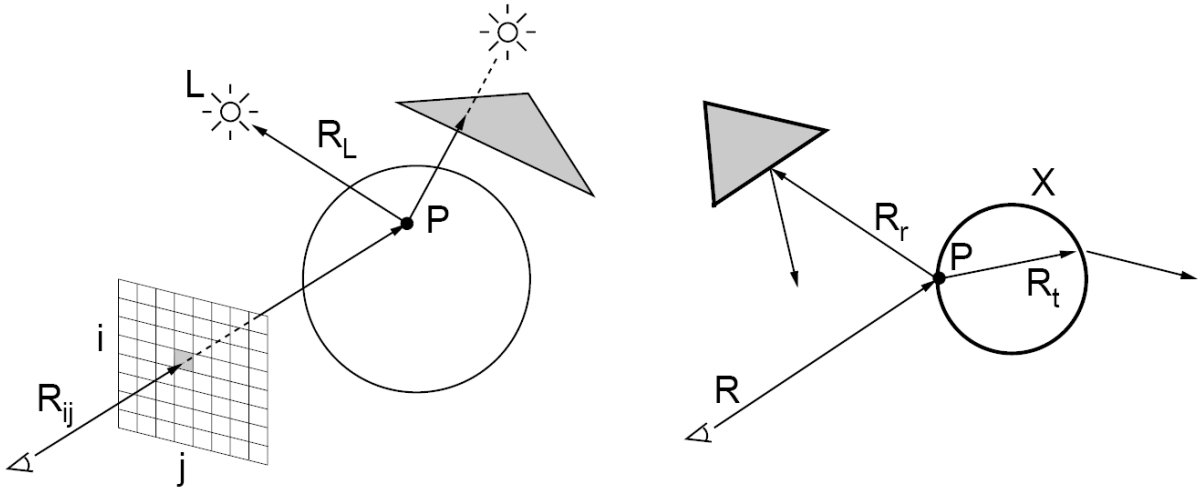


$$\sin\theta_r = \frac{\eta_{i\lambda}\sin\theta_i}{\eta_{r\lambda}}$$

$\eta_{i\lambda}$ = index of refraction of medium 1

$\eta_{r\lambda}$ = index of refraction of medium 2

Recursive Raytracing Recap



- raytrace():
  foreach column i
    foreach row j
      - make ray Rij from eye to pixel i,j
      - pixcolor = trace(Rij)
      - set pixel i,j to pixcolor
- **trace(R):**
  - find intersection point P, where R hits nearest object
  - if object is reflective: make reflection ray Rr; Colorl = **trace(Rr)**
  - if object is transparent: make refraction ray Rt; Colorr = **trace(Rt)**
  - foreach light source: make ray RL from P to source
    - if object unobstructed from light source, use lighting model to find color Colorm
  - return the combined color of: colorl, colorr, colorm

Drawing the Ray-Traced Image

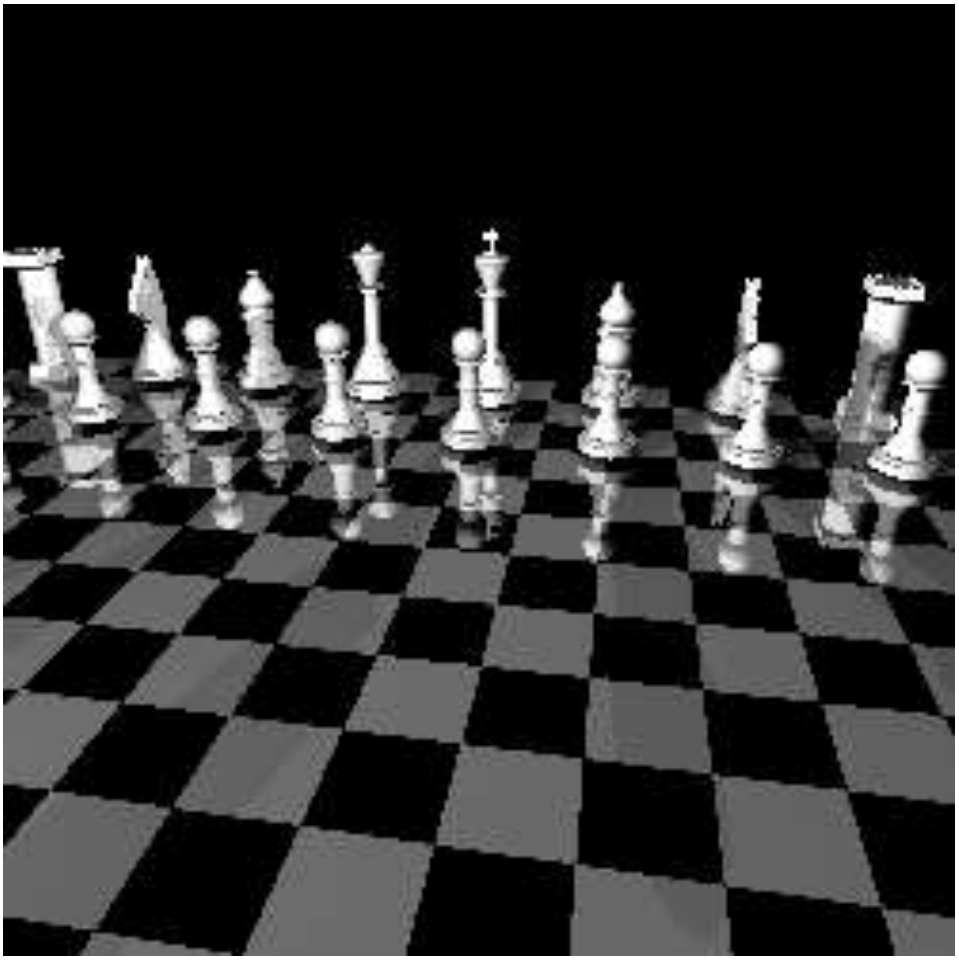
Steps:

1. Allocate space for pixel array and compute intensities via ray tracing, per pixel

2. Write array to Frame Buffer:
   - glRasterPos(x, y) – specify x, y position where we want to write some pixels
   - glDrawPixels(…, *pixel_array) – write a block of pixels to the frame buffer

Choosing Samples (1/2)

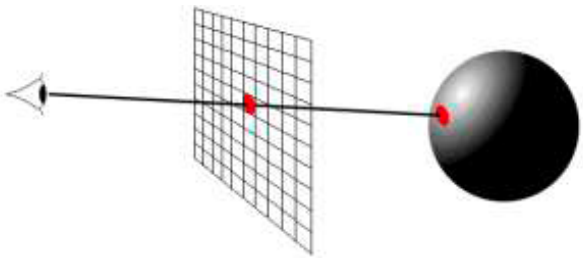
- In the examples and in your P06 assignment we sample once per pixel and get images similar to this:



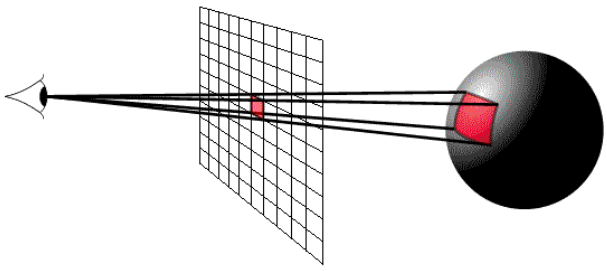
- We have a case of the jaggies.
- Can we do better?

Choosing Samples (2/2)

- In the simplest case, sample points are chosen at pixel centers
- For better results, *supersamples can be chosen (called supersampling)*
  - e.g., at corners and at center
- How do we convert samples to pixels? Filter to get weighted average of samples

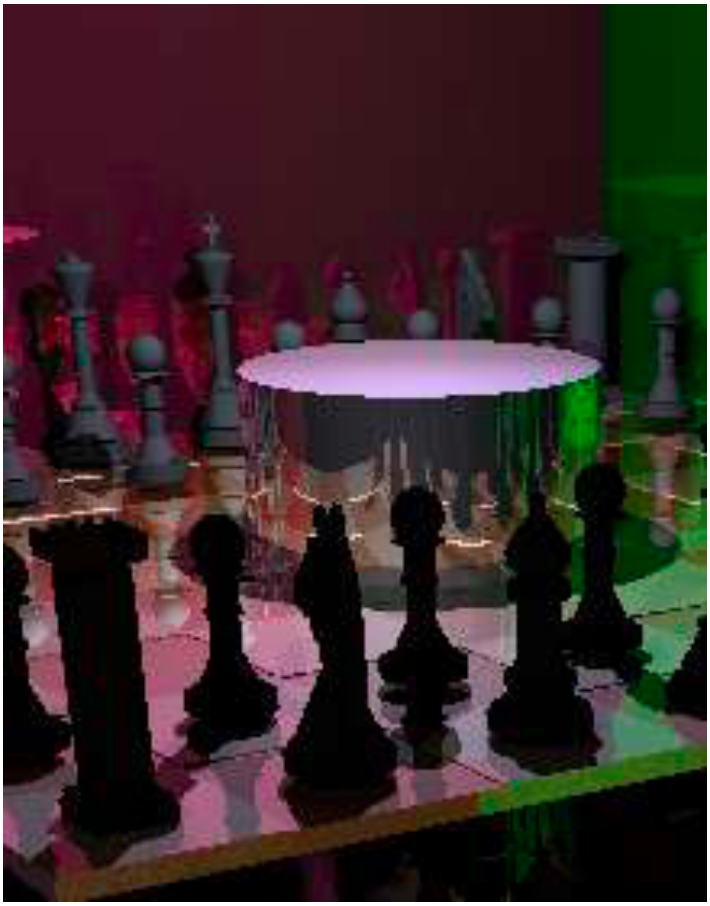


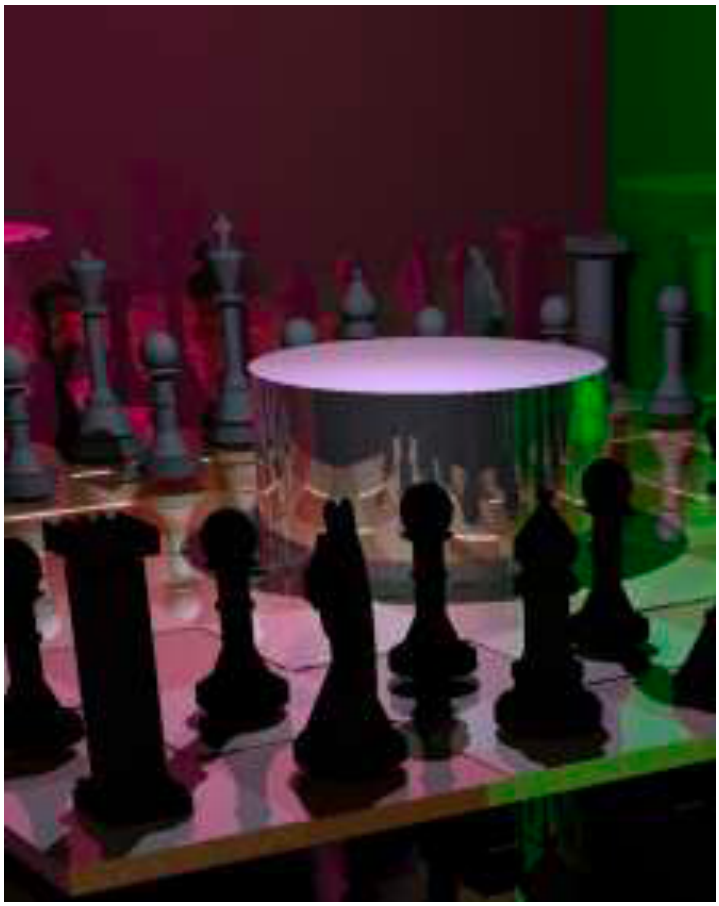
vs.



Supersampling

W/out SS (left) and w/ SS (right):



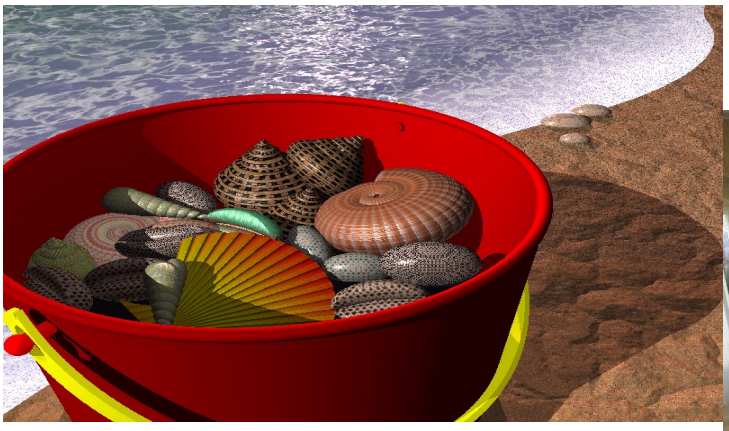


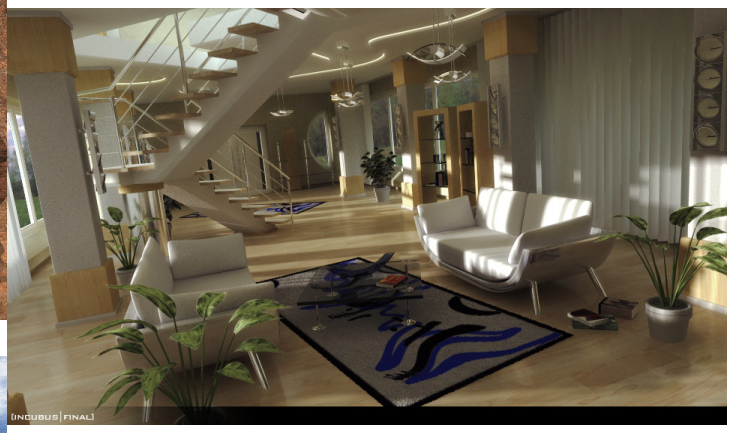
rendered in a matter of seconds with Travis Fischer Brown'09 ray tracer

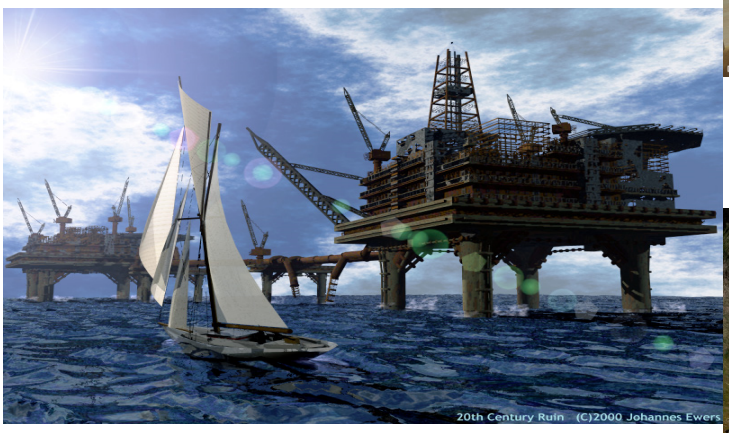
POV-Ray: Pretty Pictures

Free Advanced Raytracer

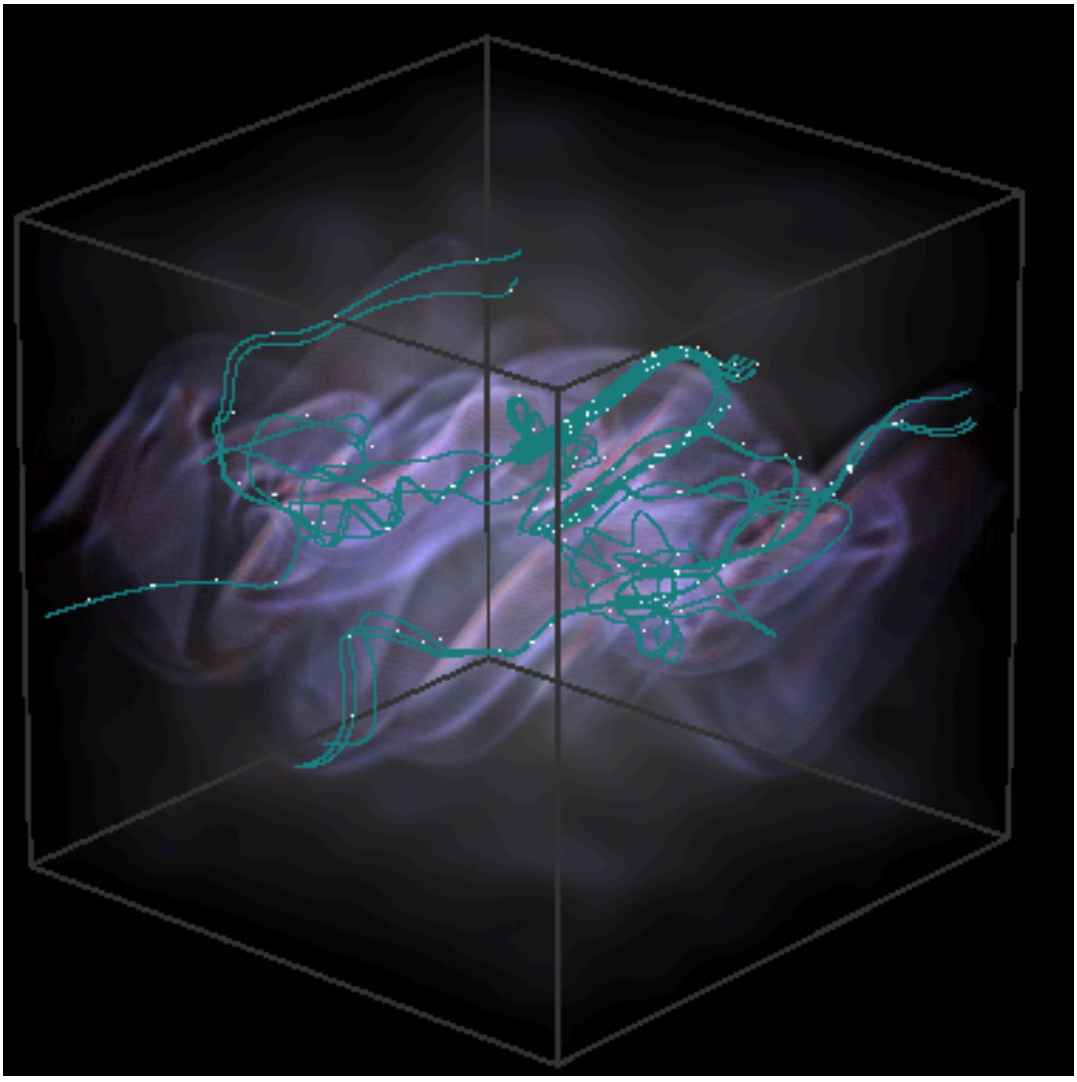
- Full-featured raytracer available online: povray.org
- Obligatory pretty pictures (see The Internet Raytracing Competition irtc.org):







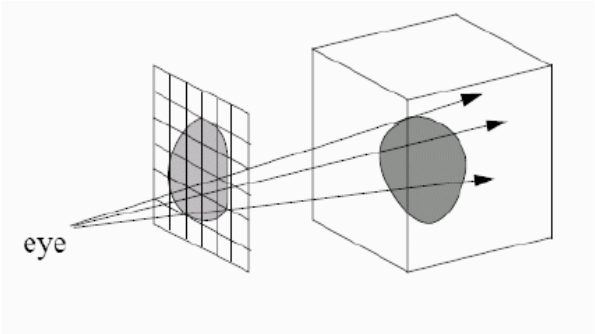
Variation: Volume Rendering

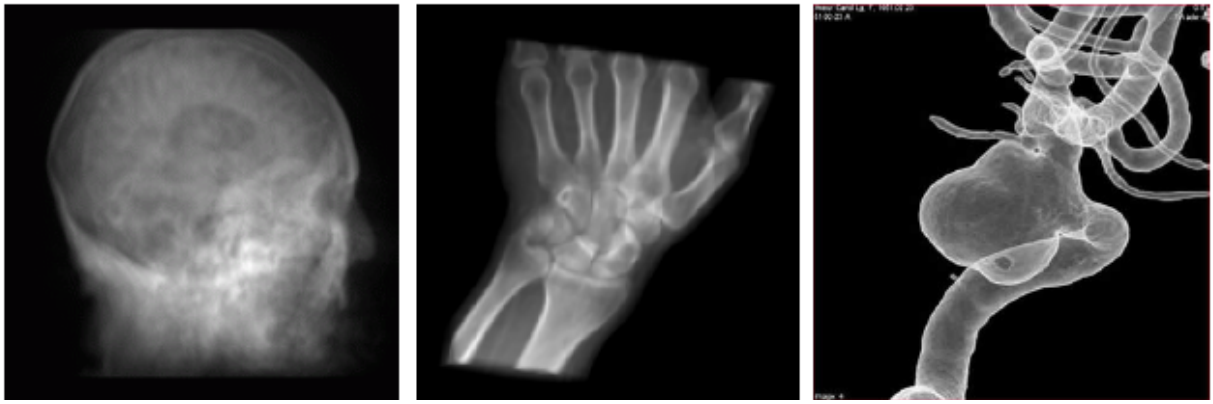


Ray Casting

Most intuitive rendering technique

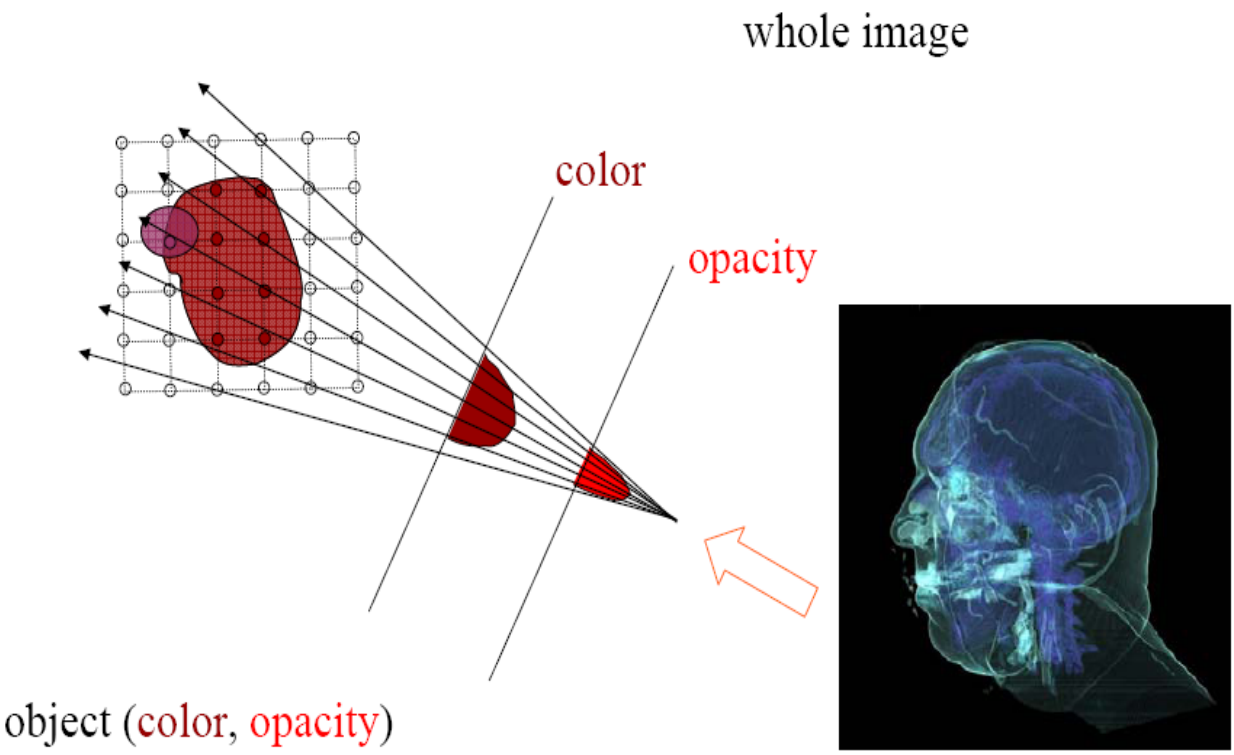
- shoot rays into the scene starting from the eye
- the "gold standard" of volume rendering
- use it to derive the fidelity of other paradigms





Rays simply sum everything up that falls into their path

Raycasting: Color and Opacity



Summary

- Ray-tracing: forget (almost) everything we've learned about the polygon-based rendering pipeline

- Work backwards (see Durer):
  - instead of generating vertices, then projecting to 2D, then converting polygons to pixels, and coloring the pixels…
  - start with the pixels, compute mathematically intersections with the objects (no need to tessellate into vertices!!), compute color/illumination at intersection, draw colored pixels

  aka non-recursive ray-tracing

- Recursive ray-tracing
  - trace rays beyond first intersection point
  - send rays preferentially towards light sources and towards other objects in the scene
  - add their contributions to the pixel color

- Transparency in ray-tracing
- Super-sampling to avoid jagged boundaries
- Variation: Volume rendering

- The simplest global illumination model there is
  - embarrassingly parallelizable