Photorealism -- Taking into Account Global Illumination

- Light can arrive at surfaces indirectly
- This light called global illumination
- To now we’ve approximated it with a constant, diffuse ambient term
  - This is wrong
- Need to take into account all the multiply reflected light in the scene
- Two different approaches:
  - Ray Tracing -- specularly reflected light
  - Radiosity -- diffusely reflected light

Photorealism: Ray Tracing

- See CS-460/560 Notes at:
  - http://www.cs.binghamton.edu/~reckert/460/raytrace.htm
- Persistence of Vision Ray Tracer (free): http://povray.org/

Ray Tracing

- What is seen by viewer depends on:
  - rays of light that arrive at his/her eye
- So to get “correct” results:
  - Follow all rays of light from all light sources
  - Each time one hits an object, compute the reflected color/intensity
  - Store results for those that go through projection plane pixels into observer’s eye
  - Paint each pixel in the resulting color
Forward Ray Tracing
- Infinite number of rays from each source
- At each intersection with an object – could have an infinite number of reflected rays
- Completely intractable
- Would take geological times to compute

Backward Ray Tracing
- Look only at rays observer sees
- Follow rays backwards from eye point through pixels on screen
- Check if they intersect objects
  - If so, can intersection point see a light source?
    - If so, compute intensity of reflected light
    - If not, point is in the shadow
  - If object has reflectivity/transparency
    - Follow reflected/transmission rays

Simple Ray Tracing
- Set up scene (position objects and light sources)
- For each pixel on screen
  - Form eye ray – vector from viewpoint through pixel
  - For each object in scene
    - If object is intersected by eye ray & is closest, store intersection point & object ID
    - Apply illumination model to closest intersection point
    - Plot pixel in resulting color
- (Another way of doing hidden surface Removal)
- Difficulty is getting intersection points for complex objects

Recursive Ray Tracing
- Good for scenes with lots of specular reflection and transparency
- Also gives shadows automatically
- At each intersection point send out:
  - Shadow feeler rays toward light sources
  - Reflection rays in ideal reflection direction
  - Transmission rays in refraction direction
- Treat last two as eye rays
- Recursive algorithm
Recursive Ray Tracing Algorithm

\[
\text{depth} = 0
\]

for each pixel \((x,y)\)

Calculate direction from eyepoint to pixel

\[
\text{TraceRay \ (eyepoint, direction, depth, color)}
\]

FrameBuf \([x,y]\) = \(\text{color}\)

---

Combining Color from Reflection Ray

- Add attenuated reflected color intensity to local color intensity:
  \[
  \text{color} = \text{local\_color} + k \times \text{refl\_color}
  \]
  - here \(\text{refl\_color}\) is \((r,g,b)\) - color returned by reflection ray
  - \(\text{local\_color}\) is \((r,g,b)\) - color computed by illumination model at intersection point
  - \(k\) is an attenuation factor (<1)

---

Combining Color from Transmission Ray

- Observer sees a mixture of light reflected off surface and light transmitted through surface
- So combine colors (interpolate)

\[
I(r,g,b) = k' \times I_{\text{local}}(r,g,b) + (1 - k') \times I_{\text{transmitted}}(r,g,b)
\]

- \(k'\) is opacity factor coefficient
- \(k'=0\) => perfectly transparent, \(k'=1\) ==> perfectly opaque

---

Ray Tracing Intersection Calculations

Example Ray Intersection Calculation:
An Eye Ray with a Sphere

\[
\begin{align*}
  x &= x_0 + (x_1-x_0) \times t = x_0 + ?x^*t \\
  y &= y_0 + (y_1-y_0) \times t = y_0 + ?y^*t \\
  z &= z_0 + (z_1-z_0) \times t = z_0 + ?z^*t
\end{align*}
\]

---

Parametric Equations for Eye Ray:
**Equation of a sphere of radius r, centered at (a,b,c)**

\[(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2\]

Substitute ray parametric equations:

\[(x0+x\cdot t-a)^2 + (y0+y\cdot t-b)^2 + (z0+z\cdot t-c)^2 = r^2\]

Rearrange terms:

\[((x0-a)+x\cdot t)^2 + ((y0-b)+y\cdot t)^2 + ((z0-c)+z\cdot t)^2 = r^2\]

This is a quadratic in parameter t

Solution(s): value(s) of t where ray intersects sphere

Three cases

- No real roots ==> no intersection point
- 1 real root ==> ray grazes sphere
- 2 real roots ==> ray passes thru sphere

• Select smaller t (closer to source)

**Sphere Normal Computation**

To apply illumination model at intersection point P, need surface normal

Can show: \[N = \left(\frac{x-a}{r}, \frac{y-b}{r}, \frac{z-c}{r}\right)\]

**Disadvantages of Ray Tracing**

- Extremely compute intensive
  - But there are several acceleration techniques
  - Bad for scenes with lots of diffuse reflection
  - But can be combined with other algorithms that handle diffuse reflection well
- Prone to aliasing
  - One sample per pixel
    - can give ugly artifacts
  - But there are anti-aliasing techniques

**Persistence of Vision Ray Tracer**

- POVRay free software
- Lots of capabilities
- Great for playing around with ray tracing
  - [http://povray.org/](http://povray.org/)
An Algorithm Animation of a Ray Tracer

Ray Tracing Algorithm Animator in VC++ (with David Goldman)
See:
http://www.cs.binghamton.edu/~reckert/3daape_paper.htm
Ray Tracing Algorithm Animation Java Applet and Paper (with Brian Maltzan)
See:
http://www.cs.binghamton.edu/~reckert/brian/index.html

General Texture Mapping

Pattern Mapping Technique
- Modulate surface color calculated by reflection model according to a pattern defined by a texture function
  - ("wrap" pattern around surface)
  - 2-D Texture function: T(u,v)
    - Could be a digitized image
    - Or a procedurally defined pattern

Ex: Inverse Mapping a Polygon

Choose axis S (unit vector) along a polygon edge
- will align with u-axis in texture space
Choose a polygon vertex Po(xo,yo,zo)
- will correspond to origin in texture space
Choose a scaling factor k
- k = max dimension of polygon
- 0-k in object space → 0-1 in texture space
Want to map point P(x,y,z) on polygon to (u,v)
- Construct V = P - Po
- V.S = k*U, projection of P onto S
- So u = V.S/k
- Choose orthogonal axis T in polygon plane (T=NxS)
- v=V.T/k

Pattern/Texture Mapping

Adding details or features to surfaces
- (variations in color or texture)

Inverse Pixel Mapping (Screen Scanning)

For each pixel on screen (xs, ys)
Compute pt(x,y,z) on closest surface projecting to pixel (e.g., ray tracing)
Determine color (e.g., apply illumination/reflection model)
Compute (u,v) corresponding to (x,y,z) (inverse mapping)
Modulate color of (xs,ys) according to value of T(u,v) at (u,v)