Rendering

Rendering is the process of creating an image of a geometric model. There are questions you need to ask:

- For what purpose am I doing this?
- How realistic do I need this image to be?
- How much compute time do I have to create this image?
- Do I need to take lighting into account?
- Does the illumination need to be Global or will Local do?
- Do I need to create shadows?
- Do I need to create reflections and refractions?
- How good do the reflections and refractions need to be?

The Rendering Equation

\[ B(P, d_0, \lambda) = E(P, d_0, \lambda) + \int \frac{B(P, d_i, \lambda)f(\lambda, d_i, d_0)(d_i \cdot n)}{d_i} d\Omega \]

This is the true rendering situation. Essentially, it is an energy balance:

- Light Shining from a point P =
- Light emitted by that point + (Reflectivity \times \sum \text{Light arriving from all other points})

But, this is time-consuming to solve "exactly". So, we need to know how much of an approximation we need.

Local vs. Global Illumination

- If the appearance of an object is only affected by its own characteristics and the characteristics of the light sources, then you have Local Illumination.
- If the appearance of an object is also affected by the appearances of other objects, then you have Global Illumination.

OpenGL rendering uses Local Illumination.

Global Illumination at Work

- The left wall is green.
- The right wall is red.
- The back wall is white.
- The ceiling is blue with a light source in the middle of it.
- The objects sitting on the floor are white.

"If the appearance of an object is also affected by the appearances of other objects, then you have Global Illumination."
Two Directions for the Rendering to Happen

1. Starts at the object, works towards the eye
2. Starts at the eye, works towards the object

Starts at the Object, Works Towards the Eye

- This is the kind of rendering you get on a graphics card (e.g., OpenGL).
- You have been doing this all along.
- Start with the geometry and project it onto the pixels.

Why do things in front look like they are really in front?

Solution #1: Sort your polygons in 3D by depth and draw them back-to-front. In this case 1-2-3-4-5-6 becomes 5-6-2-4-1-3. This is called the Painter's Algorithm. It sucked to have to do things this way.

Solution #2: Add an extension to the framebuffer to store the depth of each pixel. This is called a Depth-buffer or Z-buffer. Only allow pixel stores when the depth of the incoming pixel is closer to the viewer than the pixel that is already there.

Why do things in front look like they are really in front?

Why do things in front look like they are really in front?

With Depth Buffer

Without Depth Buffer
Ray-Tracing: Start at the Eye, Work Towards the Objects

Splat!

The pixel is painted the color of the nearest object that is hit.

In a Ray-Tracing, each Ray typically hits a lot of Things — You Need to Find All Hits, then Find the Nearest Hit and work from There

Ray-Tracing

It's also straightforward to see if this point lies in a shadow: Fire another ray towards each light source. If the ray hits anything, then the point does not receive that light.

Ray-Tracing

It's also straightforward to handle reflection Fire another ray that represents the bounce from the reflection. Paint the pixel the color that this ray sees.

Ray-Tracing

It's also straightforward to handle refraction

Fire another ray that represents the bend from the refraction. Paint the pixel the color that this ray sees.

The Physics of Refraction

Snell’s Law of Refraction:

\[
\frac{\sin \Theta_A}{\sin \Theta_B} = \frac{\eta_B}{\eta_A}
\]

<table>
<thead>
<tr>
<th>Material</th>
<th>Refractive Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vacuum</td>
<td>1.00000</td>
</tr>
<tr>
<td>Air</td>
<td>1.00029</td>
</tr>
<tr>
<td>Ice</td>
<td>1.309</td>
</tr>
<tr>
<td>Water</td>
<td>1.333</td>
</tr>
<tr>
<td>Plexiglass</td>
<td>1.49</td>
</tr>
<tr>
<td>Glass</td>
<td>1.56</td>
</tr>
<tr>
<td>Diamond</td>
<td>2.42</td>
</tr>
</tbody>
</table>
Determining Ray-Shape Intersections

1. Ray Equation
   \[ x = E_x + t(P_x - E_x) \]
   \[ y = E_y + t(P_y - E_y) \]
   \[ z = E_z + t(P_z - E_z) \]

2. Substitute \( t \) into Sphere Equation
   \[(x - P_x)^2 + (y - P_y)^2 + (z - P_z)^2 = R^2\\]

3. Collect terms
4. Solve using Quadratic Formula
\[ t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \]

Three cases of possible solutions:
1. Both \( t \)'s are complex: ray missed the sphere
2. Both \( t \)'s are real and identical: ray is tangent to the sphere
3. Both \( t \)'s are real and different: ray goes through the sphere

IronCAD Ray-Tracing Example

Blender Ray-Tracing Examples

More Ray-tracing Examples

IBM

More Ray-Tracing Examples

Bunkspeed
Subsurface Scattering

- Subsurface Scattering mathematically models light bouncing around within an object before coming back out.
- This is a good way to render skin, wax, milk, paraffin, etc.

Subsurface scattering

Original rendering

The Three Elements of OpenGL Lighting

The biggest problem with the Ambient-Diffuse-Specular way of computing lighting is that we are trying to match an appearance, not necessarily follow the laws of physics.

For example, using A-D-S, you can easily create a scene where the amount of light arriving from the objects exceeds the amount of light that the light source is supplying!

This brings us to Physically-Based Rendering (PBR).

The Three Elements of OpenGL Lighting

Ambient

Diffuse

Specular

The Radiosity Equation

\[ B_iA_i = E_iA_i + \rho \sum_j B_jA_jF_{j-i} \]

- \( B_i \) is the light energy intensity shining from surface element \( i \)
- \( A_i \) is the area of surface element \( i \)
- \( E_i \) is the internally-generated light energy intensity for surface element \( i \)
- \( \rho \) is surface element \( i \)'s reflectivity
- \( F_{j-i} \) is referred to as the Shape Factor, and describes what percent of the energy leaving surface element \( j \) arrives at surface element \( i \)

The Radiosity Shape Factor

\[ F_{j-i} = \int_{A_j} \int_{A_i} \frac{\cos \theta_i \cos \theta_j}{\pi \text{Dist}(d_i, d_j)} dA_j dA_i \]

Does it seem to you that the light just keeps propagating and you never get an answer?

To many people, radiosity seems like this:

"x produces y, then y produces x, then x produces y, then..."

Not really – it is simply \( N \) equations, \( N \) unknowns – you solve for the unique solution.
The Radiosity Matrix Equation

\[ B_i A_i = E_i A_i + \rho_i \sum_{j} B_j A_j F_i \rightarrow B_i A_i - \rho_i \sum B_j A_j F_{ij} = E_i A_i \]

Expand for each surface element, and re-arrange to solve for the surface intensities, the \( B_i \):s:

\[
\begin{bmatrix}
1 - \rho_1 F_{i+1} & -\rho_1 F_{i+2} & \cdots & -\rho_1 F_{i+N} \\
-\rho_2 F_{i+1} & 1 - \rho_2 F_{i+2} & \cdots & -\rho_2 F_{i+N} \\
\cdots & \cdots & \cdots & \cdots \\
-\rho_N F_{i+1} & -\rho_N F_{i+2} & \cdots & 1 - \rho_N F_{i+N}
\end{bmatrix}
\begin{bmatrix}
B_i \\
B_{i+1} \\
\vdots \\
B_{i+N}
\end{bmatrix}
= 
\begin{bmatrix}
E_i \\
E_{i+1} \\
\vdots \\
E_{i+N}
\end{bmatrix}
\]

This is a lot of equations!

When light hits a surface, it bounces in particular ways depending on the angle and the material.

This distribution of bounced light rays is called the Bidirectional Reflectance Distribution Function, or BRDF.

For a given material, the BRDF behavior of a light ray is a function of 4 variables: the 2 spherical coordinates of the incoming ray and the 2 spherical coordinates of the outgoing ray.

The outgoing light energy in the outgoing BRDF’s is always less than or equal to the amount of light that shines in.

Usually it is easier to trace from the eye.

Somewhat like ray-tracing, somewhat like radiosity where light can bounce around the scene but this has more sophisticated effects.
Clearly this is capable of spawning an infinite number of rays. How do we handle this?

Monte Carlo simulation to the rescue!

\[ \text{LightGathered} = \sum_{i=1}^{N} \text{ResultOfRayCastInRandomDirection} \]

Each time a ray hits a surface, use the equation at that point. Continue until:

1. Nothing is hit
2. A light is hit
3. Some maximum number of bounces are found

Recurse by applying this equation for all ray hits (yikes!)
An Neat Global Illumination-ish Trick: Screen Space Ambient Occlusion (SSAO)

“Render” these normals into a software framebuffer.

Now, look for places in the framebuffer where there is a discontinuity in the normal.

Make that part of the scene darker.

This part of the scene should be darker because it is harder for ambient light to get down between objects.