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

\[ B(P, d_0, \lambda) = E(P, d_0, \lambda) + \int B(P, d_0, \lambda) f(\lambda, d_0, d_0) (d_1 \cdot \hat{n}) d\Omega \]

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

- Light shining from a point \( P \) = Light emitted by that point + Reflectivity * \( \Sigma \) (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**.
"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."

OpenGL rendering uses Local Illumination

"If the appearance of an object is also affected by the appearances of other objects, then you have Global Illumination."

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.

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?

Your application might draw this cube’s polygons in 1-2-3-4-5-6 order, but 1, 3, and 4 still need to look like they were drawn last:

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.
Ray-Tracing: Start at the Eye, Work Towards the Objects

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.
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.

Snell's Law of Refraction:

\[ \frac{\sin \theta_A}{\sin \theta_B} = \eta_A \]

<table>
<thead>
<tr>
<th>Material</th>
<th>Index of Refraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vacuum</td>
<td>1.00000</td>
</tr>
<tr>
<td>Air</td>
<td>1.0029</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>1.60</td>
</tr>
<tr>
<td>Diamond</td>
<td>2.42</td>
</tr>
</tbody>
</table>

Determining Ray-Shape Intersections

1. Ray Equation
   - \( c = E_x + t(P_x - E_x) \)
   - \( y = E_y + t(P_y - E_y) \)
   - \( z = E_z + t(P_z - E_z) \)
2. Substitute \( t > 1 \)
3. Collect terms
4. Solve using Quadratic Formula

IronCAD Ray-Tracing Example
Blender Ray-Tracing Examples

- Refraction
- Reflection

More Ray-tracing Examples

- Quake 4 Ray-Tracing Project

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

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 shining from the objects exceeds the amount of light that the light source is supplying!

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

Radiosity

Based on the idea that all surfaces gather light intensity from all other surfaces

The fundamental radiosity equation is an energy balance that says:

“The light energy leaving surface $i$ equals the amount of light energy generated by surface $i$ plus surface $i$'s reflectivity times the amount of light energy arriving from all other surfaces”

$$ B_i A_i = E_i A_i + \rho_i \sum_j {B_j A_j F_{j\rightarrow i}} $$

This is a good approximation to the Rendering Equation
The Radiosity Shape Factor

\[ F_{j\rightarrow i} = \int \int_{A_iA_j} \frac{\cos\Theta_i \cos\Theta_j}{\pi \text{Dist}(di, dj)} dA_jdA_i \]

\( \text{Dist} = \text{dA_i} \cdot \text{dA_j} \)

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.

\[-3x + y = 5 \]
\[-x - y = -7 \]

Use \( x \) to get \( y \)

\[ y = 3x + 5 \]

Use \( y \) to get \( x \)

\[ x = y - 7 \]

The Radiosity Matrix Equation

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

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

\[
\begin{bmatrix}
1 - \rho_1F_{1\rightarrow 1} & -\rho_1F_{1\rightarrow 2} & \cdots & -\rho_1F_{1\rightarrow N} \\
-\rho_2F_{2\rightarrow 1} & 1 - \rho_2F_{2\rightarrow 2} & \cdots & -\rho_2F_{2\rightarrow N} \\
\cdots & \cdots & \cdots & \cdots \\
-\rho_NF_{N\rightarrow 1} & -\rho_NF_{N\rightarrow 2} & \cdots & 1 - \rho_NF_{N\rightarrow N}
\end{bmatrix}
\begin{bmatrix}
B_1 \\
B_2 \\
\vdots \\
B_N
\end{bmatrix} =
\begin{bmatrix}
E_1 \\
E_2 \\
\vdots \\
E_N
\end{bmatrix}
\]

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!

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

Recursively apply this equation for all ray hits (yikes!)
Physically-Based Rendering using the Blender Cycles Renderer

Reflection
Area Light Source
Soft Shadows
Refraction
Caustics

Another Physically-Based Rendering Example

Reflection
Refraction
Soft Shadows

An Neat Global Illumination-ish Trick: Screen Space Ambient Occlusion (SSAO)

Image by Josiah Blaisdell

Kitware
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.

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

Make that part of the scene darker.