Rendering
The Rendering Equation

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

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

Light Shining from a point \( P \) =
- Light emitted by that point +
- Reflectivity \( \times \Sigma \) (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**
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?
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.
• 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.**”

http://www.swardson.com/unm/tutorials/mentalRay3/
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.
How 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.
How 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 #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.
Incoming RGBZ

Z

Incoming RGBZ

Existing RGBZ

Compare

Yes

No

Z_{\text{incoming}} \text{ closer to the viewer than } Z_{\text{existing}}?

Allow RGBZ_{\text{incoming}} to overwrite RGBZ_{\text{existing}}

Do nothing
Starts at the Eye, Works Towards the Objects

The most common approach in this category is *ray-tracing*:

The pixel is painted the color of the nearest object that is hit.
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.
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.
Determining Ray-Shape Intersections

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

\[ (x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = R^2 \]

\[ At^2 + Bt + C = 0 \]

\[ t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \]

3 cases:
1. 
2. 
3.
IronCAD Ray-tracing Example
Blender Ray-tracing Example

Refraction

Reflection
More Ray-tracing Examples

Quake 4 Ray-Tracing Project
More Ray-tracing Examples

IBM's Cell Interactive Ray-tracer
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, etc.
Another From-the-Object Method -- 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 very good approximation to the Rendering Equation
The Radiosity Equation

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

- \( B_i \) 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 \( i \)
- \( \rho_i \) is surface element \( i \)'s reflectivity
- \( F_{j \rightarrow i} \) 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

\[ F_{j \rightarrow i} = \iint_{A_i A_j} \text{visibility}(d_i, d_j) \frac{\cos \Theta_i \cos \Theta_j}{\pi \text{Dist}(d_i, d_j)^2} dA_j dA_i \]
Does it seem to you that the light just keeps propagating and you never get an answer?

To many people, radiosity seems like this:

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

“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

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

x = 1
y = 8
The Radiosity Matrix Equation

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

For each surface element, and re-arrange to solve for the surface intensities, the \( B \)'s:

\[
\begin{bmatrix}
1 - \rho_1 F_{1\rightarrow 1} & -\rho_1 F_{1\rightarrow 2} & \cdots & -\rho_1 F_{1\rightarrow N} \\
-\rho_2 F_{2\rightarrow 1} & 1 - \rho_2 F_{2\rightarrow 2} & \cdots & -\rho_2 F_{2\rightarrow N} \\
\vdots & \vdots & \ddots & \vdots \\
-\rho_N F_{N\rightarrow 1} & -\rho_N F_{N\rightarrow 2} & \cdots & 1 - \rho_N F_{N\rightarrow N}
\end{bmatrix}
\begin{bmatrix}
B_1 \\
B_2 \\
\vdots \\
B_N
\end{bmatrix}
= 
\begin{bmatrix}
E_1 \\
E_2 \\
\vdots \\
E_N
\end{bmatrix}
\]

This is a lot of equations!
Radiosity Examples

Cornell University
Radiosity Examples

AR Toolkit
Path Tracing

Somewhat like radiosity where light can bounce around the scene, but has more sophisticated effects.
Path Tracing
Path Tracing
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

Recurse by applying this equation for all ray hits (yikes!)

\[
\text{LightGathered} = \frac{\sum_{0}^{N-1} \text{ResultOfRaysCastInRandomDirection}}{N}
\]
Path Tracing using the Blender Cycles Renderer

- Reflection
- Non-point Light Source
- Soft Shadows
- Refraction
- Attempt at Caustics