What is rendering?

Generating an **image** from a 3D scene model

Ingredients …

- Representation of 3D geometry
- Specification for camera & lights
- Textures, material specifications, etc

Typically refers to *high-quality* image creation
Beyond “drawing”

Complex reflections
Soft shadows
Beyond “drawing”

Transparency
Refraction
Beyond “drawing”

Caustics
Dispersion
Beyond “drawing”

Depth of field
Defocus
Beyond “drawing”

Color bleeding
Beyond “drawing”

Subsurface scattering
Types of rendering

Model-centric (e.g. OpenGL/WebGL)

Centered on primitives (triangles)
Each object’s appearance is (largely) independent of other objects
Types of rendering

**Model-centric** (e.g. OpenGL/WebGL)

Advantages:

- Fast!! (can leverage *both* data-level and pipeline parallelism)
- Great standardization
- Can “fake” several effects (with hacks)
Types of rendering

Model-centric (e.g. OpenGL/WebGL)

Shortcomings:

Can’t natively do many advanced effects
Several effects even difficult to fake
Doesn’t come close to “photorealism”
Types of rendering

Light-centric

Simulate the behavior of light in a scene where light is emitted (and where to) ... how it disperses when it hits objects ... how it’s attenuated by various media ... how much of it reaches the camera ...
Types of rendering

Light-centric

Advantages:
Solid physics foundation (light transport)
Potential for photorealism
Broad gamut of demonstrated effects
(... and some parallelism)
Types of rendering

Light-centric

Shortcomings:
Not that fast …
Ray-parallelism (at best—no pipelining)
Often need lots of linear algebra
Photorealism limited by knowledge of material property details
Types of rendering

Light-centric

General idea:
There *is* a way to render anything
  (in theory ...)
... but it would be prohibitively expensive
... and we don’t fully understand materials

Remedy: Make educated compromises, to obtain acceptable results in realistic render times
Another dichotomy ...

<table>
<thead>
<tr>
<th>Direct illumination</th>
<th>Global illumination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace journey of light only to limited extent (typically minimal)</td>
<td>Seek <em>steady state</em> of light distribution within scene</td>
</tr>
<tr>
<td>Feed-forward algorithm</td>
<td>Iterative algorithms (better with more effort)</td>
</tr>
<tr>
<td>Minimal (or no) dependence between objects</td>
<td>Appearance of objects is tightly inter-dependent</td>
</tr>
<tr>
<td>Good potential for parallel performance</td>
<td>Need to try harder to expose performance potential</td>
</tr>
</tbody>
</table>
In the actual world ...

Light is emitted from sources
   ... bounces off objects (and disperses)
   ... reaches the eye or camera sensor

“Rays”?  
Light *is* quantized (photons)  
Photon bundles – rays – more practical
A ray’s journey

What happens with each “bounce”?
Light paths

Reality
L(D|S)*E

OpenGL/WebGL
L(D|S)E

Ray tracing
L(D|S)S*E

Radiosity
LD*E
Forward or backward?

Forward ray tracing is physically intuitive ...
... but many rays wasted
... need to spawn many rays to ensure

Backward ray tracing
Track rays from camera and out into the world (hopefully reaching a light source!)
(backward) Ray tracing

Camera

Image

Light Source

View Ray

Shadow Ray

Scene Object
Recursive ray tracing
Recursive ray tracing

Reflection rays

Reflected rays
Recursive ray tracing

Each ray allowed a maximum of N bounces
First N-1 bounces are specular (reflections)
Last bounce allowed to be specular and/or diffuse (apply chosen local lighting model)
At every bounce spawn: (a) reflection, (b) shadow, and (c) transmission rays
Recursive ray tracing

Reflection rays

Model specular reflection

Single ray, in the mirror-reflection direction

“Age” rays with each reflection

(stop spawning after N-1 bounces)
Recursive ray tracing

Shadow rays

Last bounce of each trace
  (but spawn at every bounce, just in case ray can reach light with fewer bounces)
If any object intersected, ray dies
  (light source shadowed)

If light reached, apply local (e.g. Phong) model
  (“eye” direction is the incoming ray direction)
Recursive ray tracing

Transmission rays

Spawn at each bounce, if a transparent object has been reached

Outgoing direction: Use Snell’s law
(need to know refractive indices)

Spawn another ray upon exiting object
Recursive ray tracing

Implementation considerations

How do we find ray-object intersections?
How are objects modeled?
Acceleration structures?
Coherence?