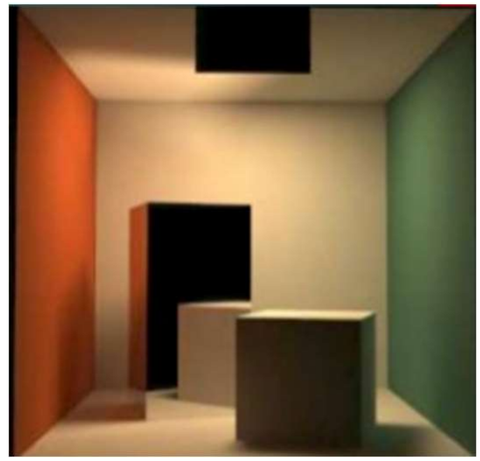
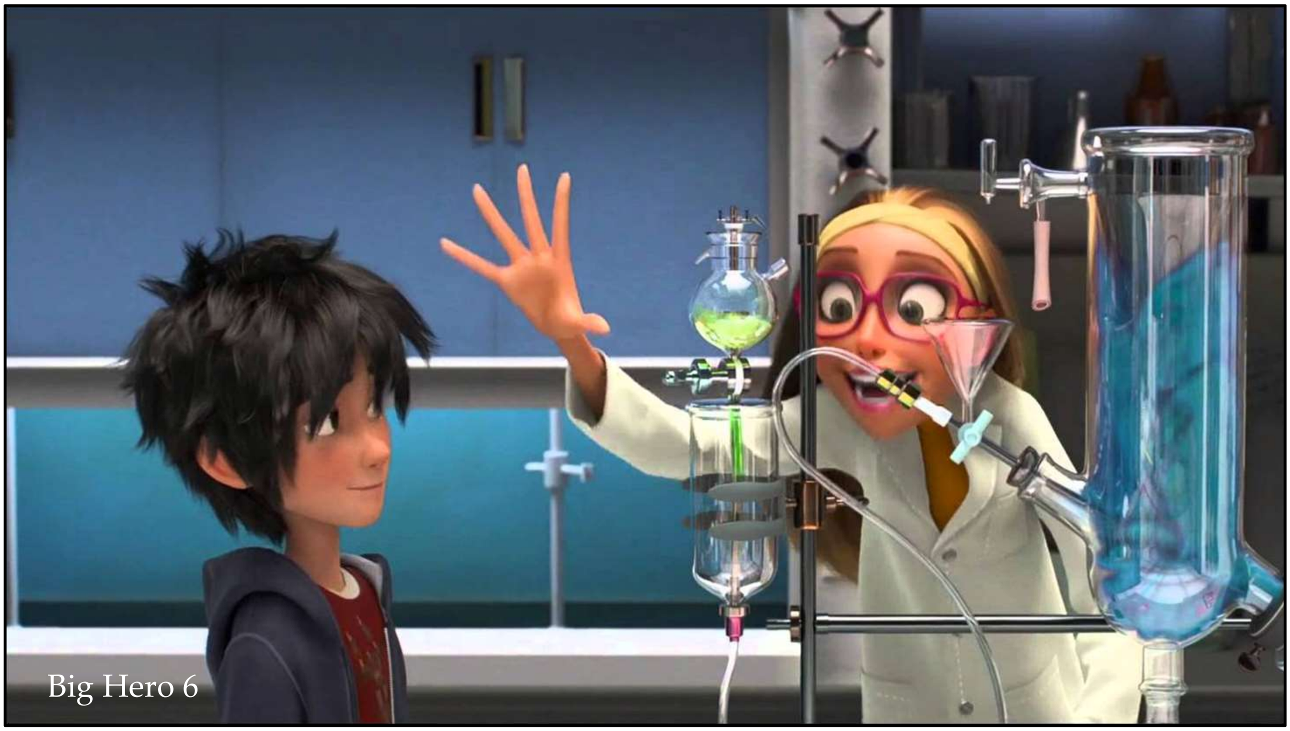
Path Tracing Review
Morgan McGuire
CS888 Fall '19
University of Waterloo

Which is Real?

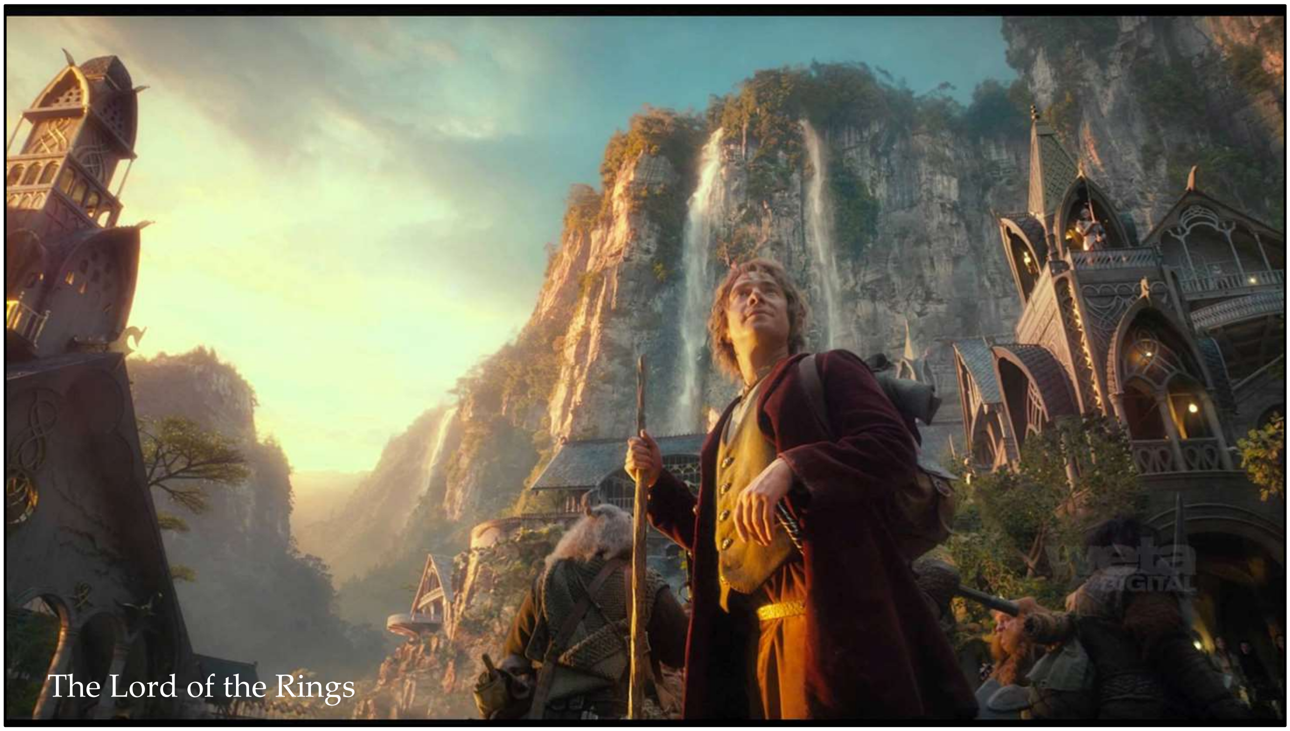


Left image is a photograph, right image is rendered by path tracing. This famous ground-truth test of a renderer is the origin of the “cornell box” 3D models—there’s a real box at Cornell.

Of course, modern renderers with good input can simulate *non-realistic scenes:*



In a way that still has realistic lighting, or fantastical geometry as if it was real…

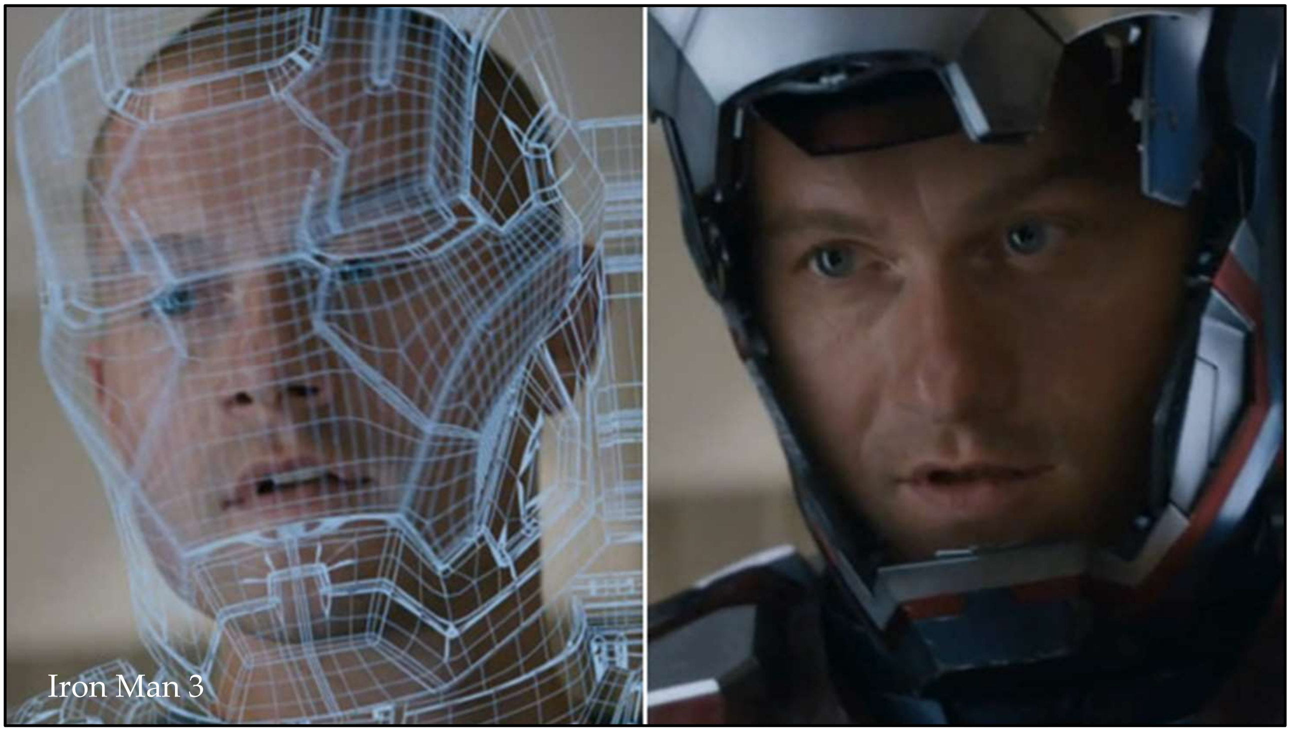

The Lord of the Rings

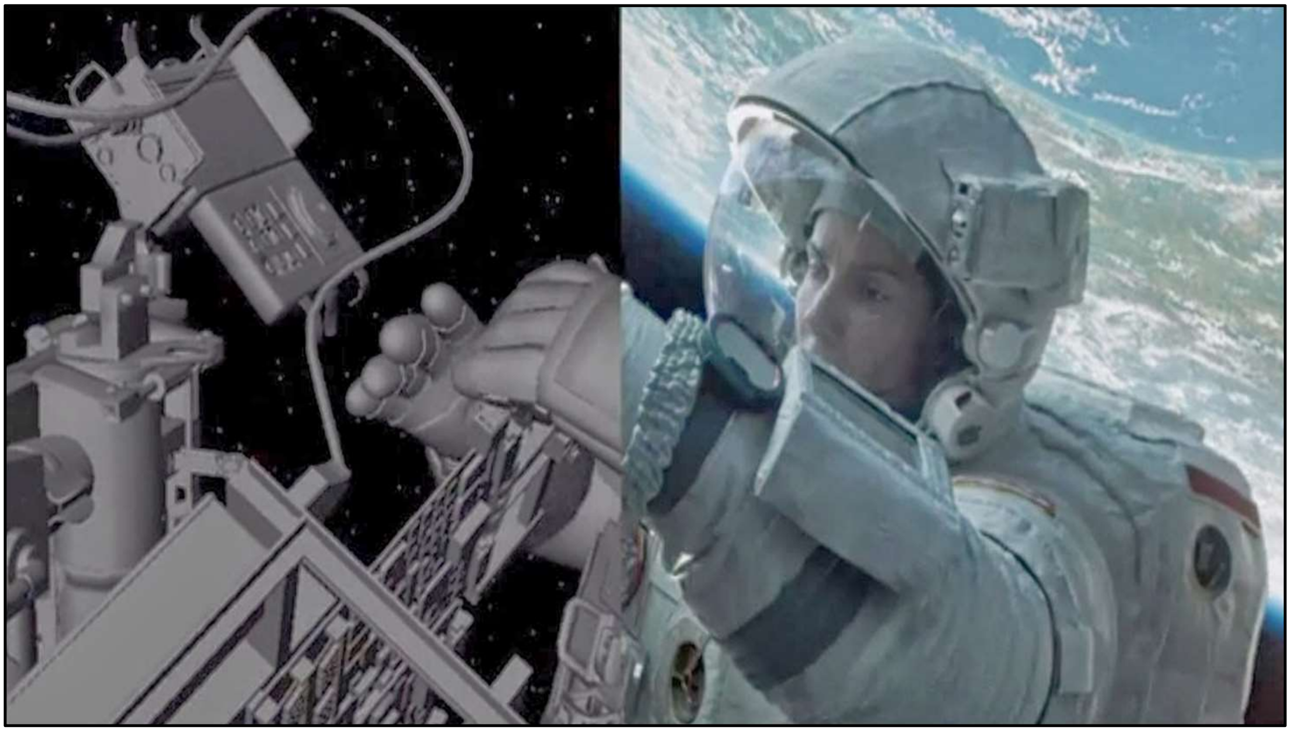
I can’t tell what is captured by a camera and what is rendered anymore in movies…

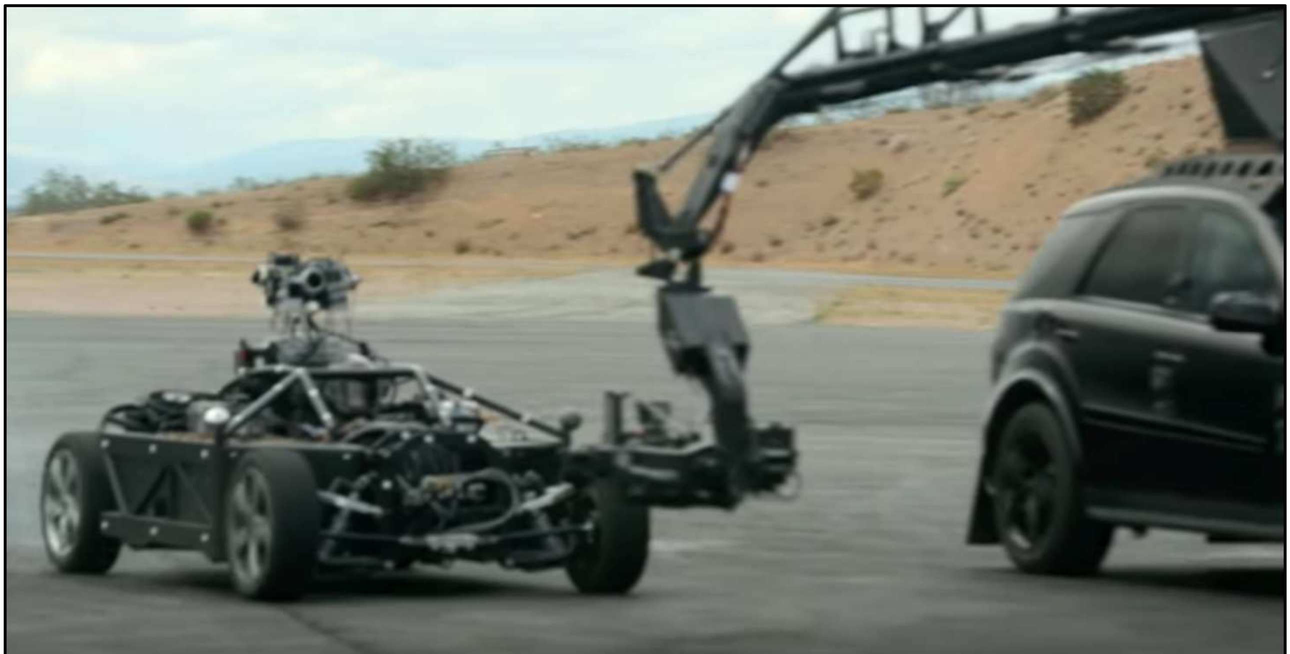
In this scene from Deadpool, you probably expect there to be some computer graphics…

But did you expect *the entire scene* to be CGI?
Action movies are increasingly like this…


DIGITAL DOMAIN


Iron Man 3





https://www.youtube.com/watch?v=OnBC5bwV5y0

And pretty much every car commercial and car scene in a film is now virtual, thanks to The Mill, one of my favorite VFX studies.

This is terrific stuff. Let's review the math that you studied previously in other graphics courses that lets us create algorithms for rendering images like these.

Measurements

Quantity	Symb.	SI Unit	1 ≈
Distance	x	meters (**m**)	Counter height
Area	A	Square meters (**m^2**)	Desk top
Angle	θ	radians (**rad**)	57 degrees Northermost latitude in ON
Solid Angle	Γ	steradians (**sr**)	Dodecahedron face The Americas
Power (energy/time)	Φ	Watts (**W**)	Cell phone flashlight
Radiance	L	**W/(m^2 sr)**	A bright indoor ray
"Biradiance"	β	**W/m^2**	Cell flashlight at arm's length

A tall person has a height of 2m and surface area of 2m^2.

Data Structures

```
class Vector3 {
public: float x, y, z;
};

typedef Vector3 Point3;
```

```
class Ray {
public:
   Point3 origin;
   Vector3 direction;
};
```

```
class Color3 {
public: float r, g, b;
};

typedef Color3 Radiance3;
typedef Color3 Power3;
```

```
class Light {
public:
   Power3 power;
   Point3 position;
};
```

```
class Image {
public:
    Array<Color3> pixel;
    int width, height;
};
```

```
class Surface {
public:
      Array<Point3> position;
      Array<int> index;
};
```

Stop for questions on what we covered last time.

Transformations

$$\mathbf{R}_{(x,y,z),\theta} = \mathbf{I} + \sin\theta \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix} + (1-\cos\theta) \begin{bmatrix} -y^2-z^2 & xy & zx \\ xy & -z^2-x^2 & yz \\ zx & yz & -x^2-y^2 \end{bmatrix}$$

$$\text{Roll: } \mathbf{R}_{\hat{z},\theta} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Yaw: } \mathbf{R}_{\hat{y},\theta} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$\mathbf{S}_{x,y,z} = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}_{(x,y,z)} = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

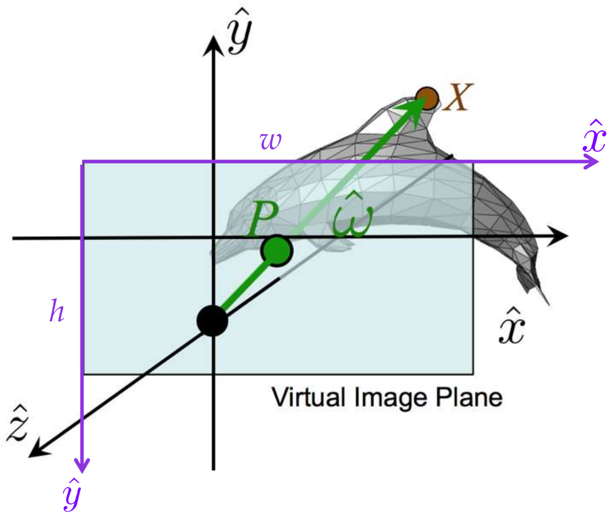
$$\text{Pitch: } \mathbf{R}_{\hat{x},\theta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

Primary Rays

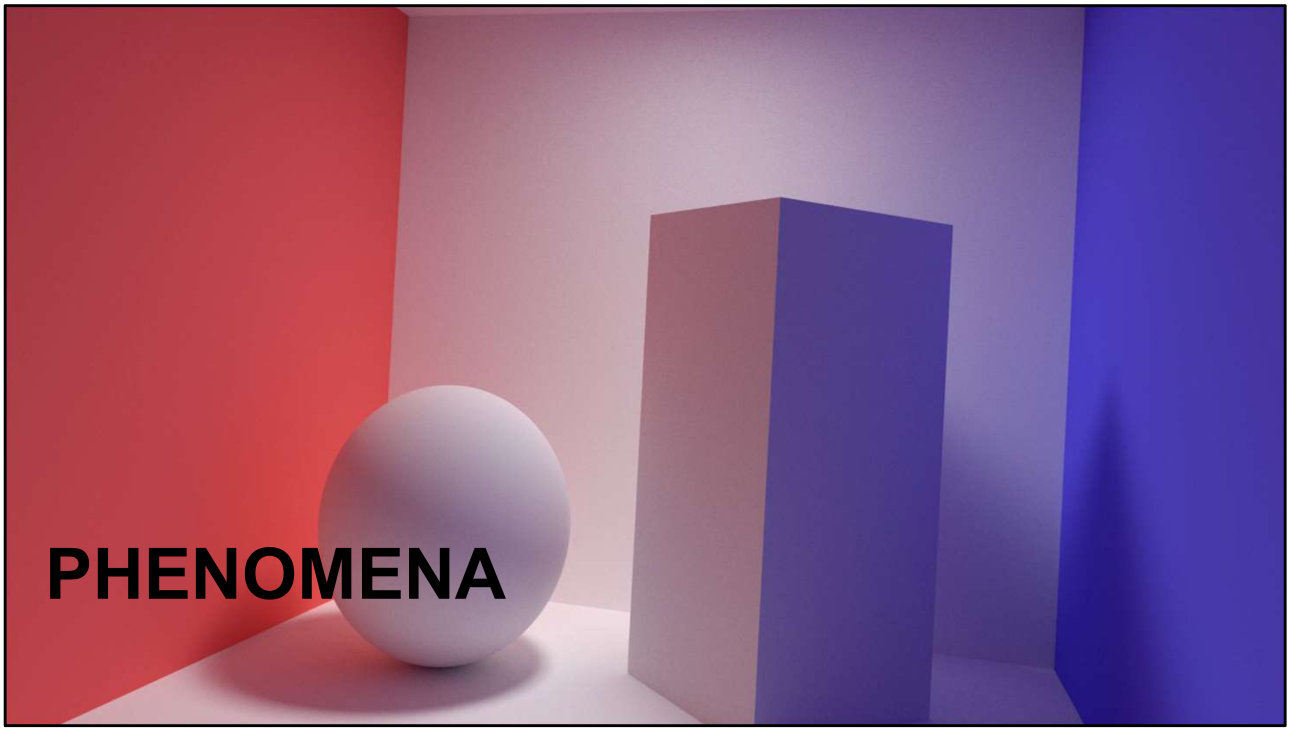
Screen space:
x, y, w, h

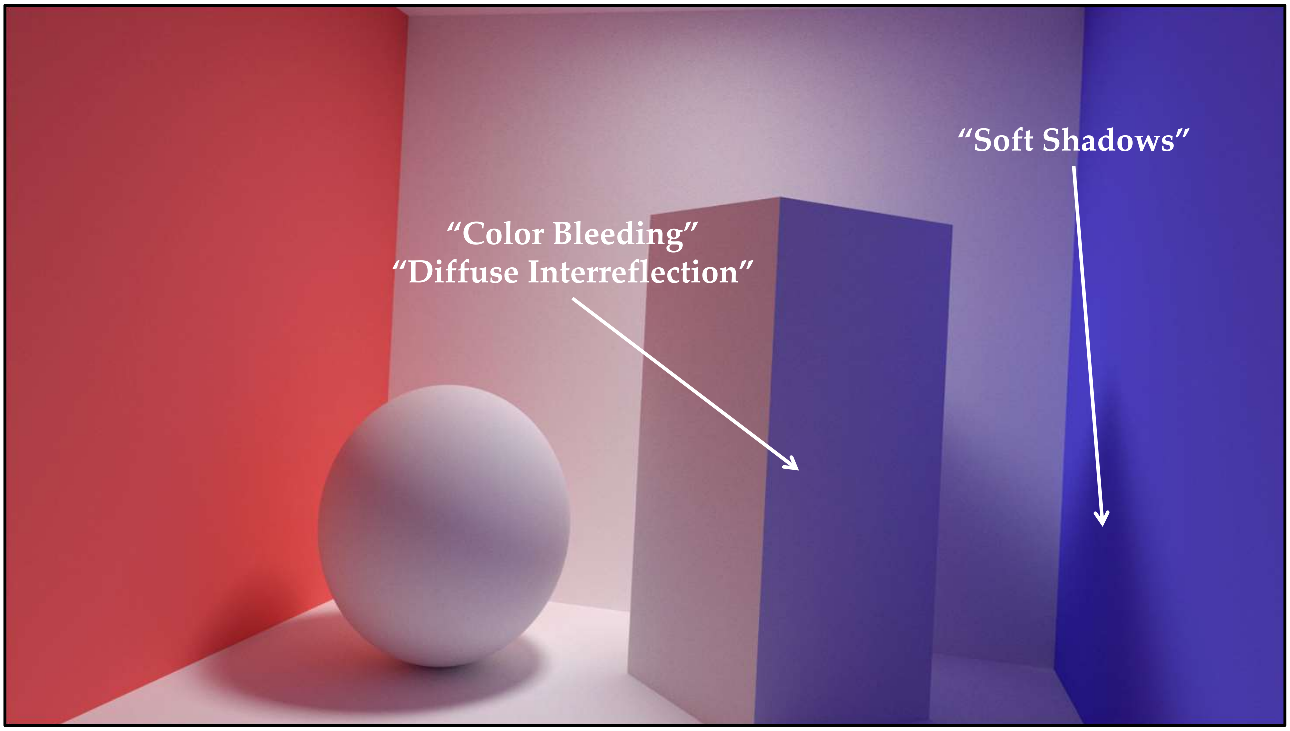
World space:
$z_P = -1$
$H = -2 \tan(\theta_y / 2)\, z_P$
$W = H * w / h$

$y_P = -(y / h - ½) * H$
$x_P = \quad (x / w - ½) * W$



(You don't have to do this in this week's project)

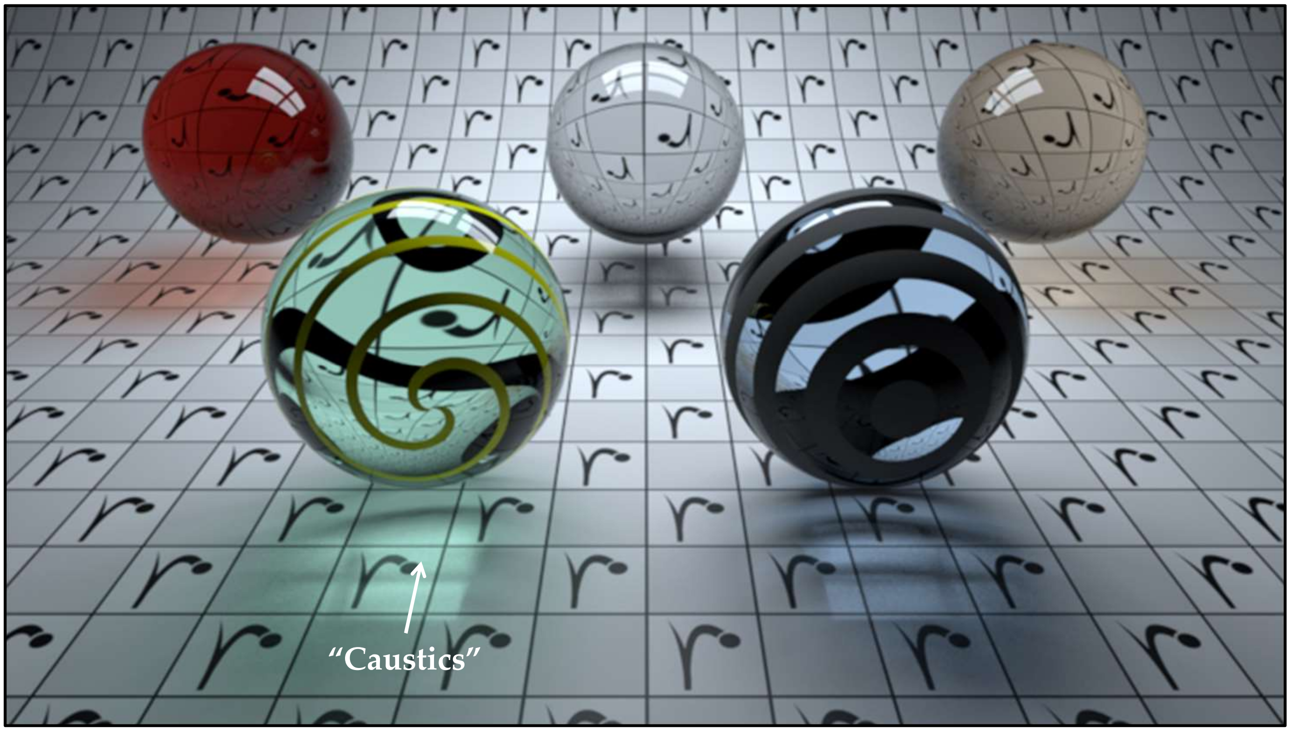

PHENOMENA


"Soft Shadows"
"Color Bleeding"
"Diffuse Interreflection"


"Indirect Lighting"


Brigade 2
"Ambient Occlusion"
NO PARKING
KEEP CLEAR

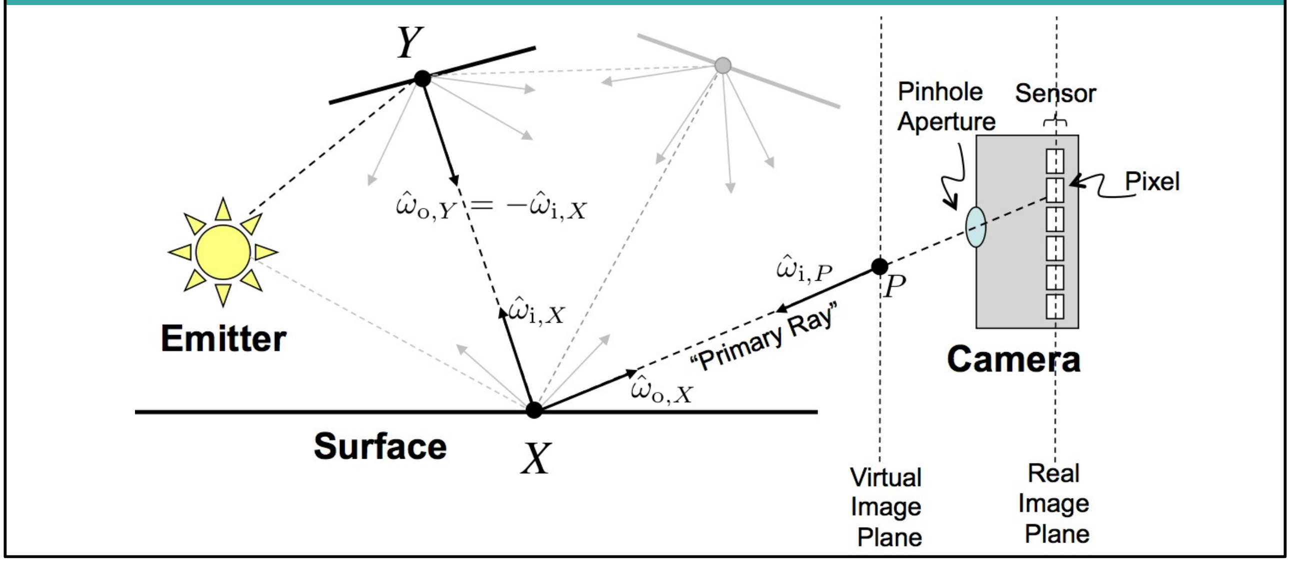

“Transmission”
“Glossy Reflection”


"Caustics"

Motion Blur

Defocus Blur ("Depth of Field")

The Rendering Equation



The Rendering Equation

Outgoing direction

Incoming direction

$$L_\mathrm{o}(\boxed{X}, \boxed{\hat{\omega}_\mathrm{o}}) = L_\mathrm{e}(X, \hat{\omega}_\mathrm{o}) + \int_{\boxed{\mathbf{S}^2}} L_\mathrm{i}(X, \hat{\omega}_\mathrm{i})~f_X(\hat{\omega}_\mathrm{i}, \hat{\omega}_\mathrm{o})~|\hat{\omega}_\mathrm{i} \cdot \hat{n}|~\boxed{d\hat{\omega}_\mathrm{i}}$$

A point in the scene

All incoming directions
(a sphere)

L_\mathrm{o}(X, \hat{\omega}_\mathrm{o}) = L_\mathrm{e}(X, \hat{\omega}_\mathrm{o}) + \int_{\mathbf{S}^2} L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})~f_X(\hat{\omega}_\mathrm{i}, \hat{\omega}_\mathrm{o})~|\hat{\omega}_\mathrm{i} \cdot \hat{n}|~d\hat{\omega}_\mathrm{i}

The Rendering Equation

$$\boxed{L_\mathrm{o}(X, \hat{\omega}_\mathrm{o})} = \boxed{L_\mathrm{e}(X, \hat{\omega}_\mathrm{o})} + \int_{\mathbf{S}^2} \boxed{L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})}~\boxed{f_X(\hat{\omega}_\mathrm{i}, \hat{\omega}_\mathrm{o})}~|\hat{\omega}_\mathrm{i} \cdot \hat{n}|~d\hat{\omega}_\mathrm{i}$$

Outgoing light Emitted light Incoming light Material

L_\mathrm{o}(X, \hat{\omega}_\mathrm{o}) = L_\mathrm{e}(X,
\hat{\omega}_\mathrm{o}) + \int_{\mathbf{S}^2}
L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})~f_X(\hat{\omega}_\mathrm{i},
\hat{\omega}_\mathrm{o})~|\hat{\omega}_\mathrm{i} \cdot
\hat{n}|~d\hat{\omega}_\mathrm{i}

Monte Carlo Integration

Integrand

$$\int_{A\subset\mathbf{M}} \underset{\mathbf{G}}{\boxed{g(m\in\mathbf{M})}}\ \underset{\subset\,\mathbf{M}}{\boxed{dm}} \approx \sum_{j=0}^{N-1} \frac{g(m_j)\in\mathbf{G}}{p(m_j)\in\mathbf{M}^{-1}}\ \frac{1}{N}$$

The integral we're estimating

Probability density function for choosing m

Number of samples

Note that the units check out…the left side has units from [G] * [M], and the right is [G] / [M]^-1

p() is a continuous probability DENSITY function. It has to be non-negative everywhere and integrate to 1 over M. It can have individual values of any magnitude, though, including infinity, as long as that is for a very small portion of the domain.

Limiting Case

$$\int_A g(m)\ dm \approx \sum_{j=0}^{N-1} \frac{g(m_j)}{p(m_j)} \frac{1}{N}$$

Let $N = 1$

$$\approx \frac{g(m)}{p(m)}$$

\int_{\mathbf{S}^2}
L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})~f_X(\hat{\omega}_\mathrm{i},
\hat{\omega}_\mathrm{o})~|\hat{\omega}_\mathrm{i} \cdot
\hat{n}|~d\hat{\omega}_\mathrm{i}

The simplest version of the integrator takes a single sample and uses that as the value, weighting it by 1/p

Let's take this simple case and subsititute the rendering equation's integral into it…

Pure Path Tracing

$$\int_{\mathbf{S}^2} L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})~f_X(\hat{\omega}_\mathrm{i},\hat{\omega}_\mathrm{o})~|\hat{\omega}_\mathrm{i} \cdot \hat{n}|~d\hat{\omega}_\mathrm{i} \approx \frac{g(m)}{p(m)}$$

Let $m = \hat{\omega}_\mathrm{i}$

$g(m) = L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})~f_X(\hat{\omega}_\mathrm{i},\hat{\omega}_\mathrm{o})~|\hat{\omega}_\mathrm{i} \cdot \hat{n}|$

$A = \mathbf{S}^2$

$p(m) = \frac{1}{4\pi~\mathrm{sr}}$

$$\int_{\mathbf{S}^2} L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})~f_X(\hat{\omega}_\mathrm{i},\hat{\omega}_\mathrm{o})~|\hat{\omega}_\mathrm{i} \cdot \hat{n}|~d\hat{\omega}_\mathrm{i} \approx L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})~f_X(\hat{\omega}_\mathrm{i},\hat{\omega}_\mathrm{o})~|\hat{\omega}_\mathrm{i} \cdot \hat{n}|~4\pi~\mathrm{sr}$$

\int_{\mathbf{S}^2}
L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})~f_X(\hat{\omega}_\mathrm{i},
\hat{\omega}_\mathrm{o})~|\hat{\omega}_\mathrm{i} \cdot
\hat{n}|~d\hat{\omega}_\mathrm{i}

L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})~f_X(\hat{\omega}_\mathrm{i},
\hat{\omega}_\mathrm{o})~|\hat{\omega}_\mathrm{i} \cdot \hat{n}| ~
4\pi~\mathrm{sr}

L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})=L_\mathrm{o}(Y,-
\hat{\omega}_\mathrm{i})

L_\mathrm{o}(X,\hat{\omega}_\mathrm{i})=L_\mathrm{e}(X,\hat{\omega}_\mathrm
{i})+\int_{\mathbf{S}^2}\!\!\!\!\cdots d\hat{\omega}_\mathrm{i}

Let the pdf be uniform on the sphere. How do I choose an incoming vector uniformly at random on the sphere? Easy: choose a uniform point in the [-1, 1]^3 cube. Throw it away if it is outside of the sphere and try again.
Points in the unit radius ball map equally to the sphere, so if you just normalize the resulting point, it will be a uniformly distributed vector

The whole thing now boils down into this simple expression, which is just the

integrand applied to a random direction times 4 pi

Pure Path Tracing

Discover Y by ray casting

Choose $\hat{\omega}_\mathrm{i}$ uniformly on $\mathbf{S}^2$

$$L_\mathrm{i}(X,\hat{\omega}_\mathrm{i}) = L_\mathrm{o}(Y,-\hat{\omega}_\mathrm{i})$$

$$L_\mathrm{o}(X,\hat{\omega}_\mathrm{o}) = L_\mathrm{e}(X,\hat{\omega}_\mathrm{i})+L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})~f_X(\hat{\omega}_\mathrm{i},\hat{\omega}_\mathrm{o})~|\hat{\omega}_\mathrm{i}\cdot\hat{n}|~4\pi~\mathrm{sr}$$

Let's put it all together for a complete solution

\int_{\mathbf{S}^2}
L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})~f_X(\hat{\omega}_\mathrm{i},
\hat{\omega}_\mathrm{o})~|\hat{\omega}_\mathrm{i} \cdot
\hat{n}|~d\hat{\omega}_\mathrm{i}

\approx\frac{L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})~f_X(\hat{\omega}_\mathrm{
i}, \hat{\omega}_\mathrm{o})~|\hat{\omega}_\mathrm{i} \cdot \hat{n}| ~
4\pi~\mathrm{sr}

L_\mathrm{i}(X,\hat{\omega}_\mathrm{i})=L_\mathrm{o}(Y,-
\hat{\omega}_\mathrm{i})

L_\mathrm{o}(X,\hat{\omega}_\mathrm{i})=L_\mathrm{e}(X,\hat{\omega}_\mathrm
{i})+\int_{\mathbf{S}^2}\!\!\!\!\cdots d\hat{\omega}_\mathrm{i}

Let the pdf be uniform on the sphere. How do I choose an incoming vector uniformly at random on the sphere? Easy: choose a uniform point in the [-1, 1]^3 cube. Throw it away if it is outside of the sphere and try again.
Points in the unit radius ball map equally to the sphere, so if you just normalize the resulting point, it will be a uniformly distributed vector

And you just run a SECOND Monte Carlo integration by running this at each pixel many times and averaging the results.
That’s it. No shadow rays, no direct illumination, no biradiance.

Core Implementation

```
def L_in(X, w_in, pathDepth, triangleArray):
    if pathDepth > maxScatteringEvents: return ambientRadiance

    (Y, n, f, L_emitted) = findFirstIntersection(X, w_in, triangleArray)

    return L_out(Y, -w_in, n, f, L_emitted, pathDepth, triangleArray)


def L_out(X, w_out, n, f, L_emitted, pathDepth, triangleArray):
    if X == null: return backgroundRadiance(w_out)

    # Uniformly sample the sphere
    w_in = sphereRandom(); p = 1 / (4 * pi)

    # Bump the origin
    X += n * epsilon * sign(w_in.dot(n))

    # 1-sample Monte Carlo integration of the rendering equation
    return (L_emitted + L_in(X, w_in, pathDepth + 1, triangleArray) *
        f(w_in, w_out) * abs(w_in.dot(n)) / p)
```

This is really the only light transport code that you need to turn any mesh with non-impulse materials…

Like this, into this…

And to kick it off, you just need a main loop that iterates over pixels…

Pixel Integrator

```
ambientRadiance = Radiance3(0.5, 0.5, 0.5)
maxScatteringEvents = 5
# Color gradient for the background
def backgroundRadiance(w):
   r = max(w.y, 0.6) * (1 - 0.5 * max(w.y, 0))
   return Radiance3(r, r, 1)

def traceImage(width, height, triangleArray, camera):
   radianceImage = Image()

   # For each pixel
   for y in range(0, height):
      for x in range(0, width):
         L = 0
         for r in range(0, raysPerPixel)
            # Generate a primary ray
            (X, w_in) = camera.worldRay(x + random(), y + random(), width, height)
            L += L_in(X, w_in, 0, triangleArray) / raysPerPixel

         radianceImage.set(x, y, L)
```

Note that this casts multiple rays at each pixel and averages their results. That’s a Monte Carlo Integrator applied to the pixel values instead of the rendering equation.

There are only two problems with this elegant solution:

Remaining Problems

1. Increase convergence rate
2. Handle impulses in L_i and f_X

1. It converges really slowly. You'd need millions of rays per pixel to get glossy surfaces to look good
2. It can't handle impulses in L_in (Point lights!) or in f (mirrors and refraction!)

We can fix both with an algorithmic optimization called importance sampling…


IMPORTANCE SAMPLING

Choose $p(m) \propto g(m)$

$$\int_A g(m)\ dm \approx \sum_{j=0}^{N-1} \frac{g(m_j)}{p(m_j)} \frac{1}{N}$$

- O(N) evaluations of g and p
- Small $g(m)$ values are inefficient: integral value per compute time is small
- Ideal: choose $p(m) = g(m) / c$

$$\approx \sum_{j=0}^{N-1} \frac{c}{N}$$

“Pure” path tracing doesn’t work for point lights. There’s zero probability that a ray hits a point light, and if it did, the radiance would be infinite. So, we have to put direct illumination, shadow rays, and biradiance back in to make point lights (vs. area lights) work. Sorry. We only use path tracing for indirect light. BUT: The direct illumimation code is a lot faster than hoping a random ray will hit the light source, and you already wrote and debugged it anyway.

Sampling Scattering

$$\approx \frac{L_{\mathrm{i}}(X, \hat{\omega}_{\mathrm{i}})\ f_X(\hat{\omega}_{\mathrm{i}}, \hat{\omega}_{\mathrm{o}})\ |\hat{\omega}_{\mathrm{i}} \cdot \hat{n}|}{p(\hat{\omega}_{\mathrm{i}})}$$

How do we choose p proportional to the integrand we don't yet know?

"BSDF importance sampling" assumes L_i is constant and chooses

$$p(\hat{\omega}_{\mathrm{i}}) \propto f_X(\hat{\omega}_{\mathrm{i}}, \hat{\omega}_{\mathrm{o}})\ |\hat{\omega}_{\mathrm{i}} \cdot \hat{n}|$$

Monte Carlo integration tells us that it would be optimal to make the denominator probability density function p() proportional to the integrand in the numerator. But the whole point was that we don't yet know the value of the numerator. What do we do?

One solution is that assume that light comes into this point X equally from all directions (it doesn't matter for correctness if this assumption is wrong, and it is often pretty reasonable). In that case, the hard value to compute: the incoming light, goes away and we are just sampling from a probability distribution that is proportional to the cosine-weighted material.

Sampling Scattering

$$\omega_i, \text{ weight} = \text{scatter}(X, n, \omega_o)$$

Where:

$$\text{weight} = \frac{f_X(\hat{\omega}_i, \hat{\omega}_o)\ |\hat{\omega}_i \cdot \hat{n}|}{p(\hat{\omega}_i)}$$

This also solves the problem of impulses (infinite values with zero range) in f() [and less importantly, in L]. We just build a function…or more likely, call into a library routine…that gives us the *ratio* of f*cos / p() after sampling (called a weight), where it will handle the infinities and make them “cancel” by taking a limit as the value approaches the impulse.

This also solves the problem of how we can choose a *single* ray when we have different values of f for difference frequencies (colors) of light: the weight is spectral, and varies depending on the ratio of f(lambda) to the monochrome p(w).

Direct Illumination

- Monte Carlo integration of direct illumination
- Compute biradiance β_j for a set of patches on lights
  - (assume no shadows)
- Choose light patch j with probability $p = \beta_j / \Sigma\beta_k$
- <u>Only</u> cast a shadow ray to patch j
- Compute direct illumination as $L = \dfrac{\beta_j |\hat{\omega}_\mathrm{i} \cdot \hat{n}|}{p}$

Tip: implement direct illumination from all lights at first with a loop. This is an optimization.

This is straight up importance sampling, except that we don't know the cumulative distribution function for the lights…so we estimate that stochastically as well.
This is called importance *re*sampling and was introduced to graphics by Talbot et al. 2005 https://dl.acm.org/citation.cfm?id=2383674, but the core idea is from stats and goes back to Rubin 1987.

<u>Three</u> Separate Monte Carlo Integrators in Basic Path Tracing

1. **Pixel Area**:
   - large N
   - p = uniform on square

2. **Direct Illumination**:
   - $N = 1$
   - p = relative biradiance

3. **Indirect Illumination**:
   - $N = 1$
   - $p \propto f_X(\hat{\omega}_\mathrm{i}, \hat{\omega}_\mathrm{o})\ |\hat{\omega}_\mathrm{i} \cdot \hat{n}|$

Summary

- The **Rendering Equation** describes steady-state recursive light flow
- **Monte Carlo integration**: add random samples weighted by the probability of choosing them
- Pure **path tracing** is amazingly simple: MC integrate the Rendering Equation
- **Importance sampling** makes path tracing more efficient and elegantly handles impulses [e.g., perfect mirrors]

For full details, see the Graphics Codex 2.17 chapters on Materials, Numerical Calculus, and Path Tracing