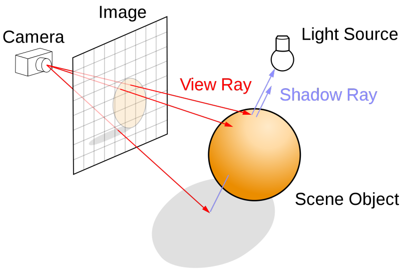
Introduction to Path Tracing

Marc Sunet

Table of contents

Plan

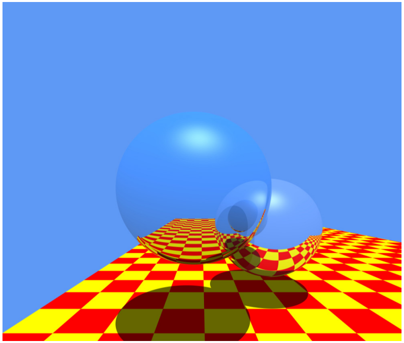
Ray Tracing



- Ray tracing is a rendering technique that generates images by tracing rays through each pixel and simulating the interaction of light with the objects in a scene.

Whitted Ray Tracing

- First recursive ray tracer by Turner Whitted (1979).
- Simulates (perfect specular) reflections, refractions and (hard) shadows.



Whitted Ray Tracing

```
render_image():
  for each pixel:
    ray = camera_ray(pixel)
    colour = trace(ray)

trace(ray):
  point, normal = intersect(ray, scene)
  colour = shade(point, normal)

shade(point, normal):
  colour = 0
  for each light source:
    trace shadow ray to light source
    if intersects(shadow ray, light source)
      colour = colour + direct illumination
  if specular:
    colour = colour + trace(reflected/refracted ray)
```

Whitted Ray Tracing

Limitations:

- No indirect illumination in diffuse surfaces.
- Hard shadows only.
- No glossy surfaces.
- No subsurface scattering.
- No participating media.
- No focusing effects due to camera lens.
- No motion blur.
- etc.

Whitted ray tracing is not a full global illumination algorithm.

Path Tracing

https://www.youtube.com/watch?v=abqAanC2NZs Brigade

Path Tracing

- Generalisation of the original ray tracing algorithm.
- Full global illumination light transport algorithm.
- Stochastically samples all light paths to simulate all light/scene interactions.
- Algorithm is *unbiased*[1] if implemented carefully.

Buzzword definition to impress friends & family:

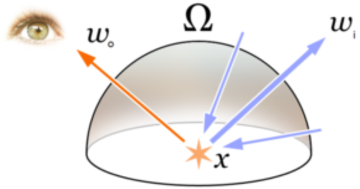
"Mathematically, path tracing is a continuous Markov chain random walk technique for solving the rendering equation. The solution technique can be seen as a Monte Carlo sampling of the Neumann series expansion of the rendering equation." — [Realistic Image Synthesis Using Photon Mapping]

[1] https://en.wikipedia.org/wiki/Bias_of_an_estimator

Plan

Light Reflection

Before delving into the path tracing algorithm, we need to understand what are we actually trying to compute.



Goal: generate an image of the scene being viewed.

- Viewer sees point x through a given pixel.
- Light reflected at x towards viewer depends on light incident at x from all directions in the normal-oriented hemisphere.
- Reflected light also depends on the properties of the surface.

Light Reflection

To compute light reflected towards viewer, we therefore need two ingredients:

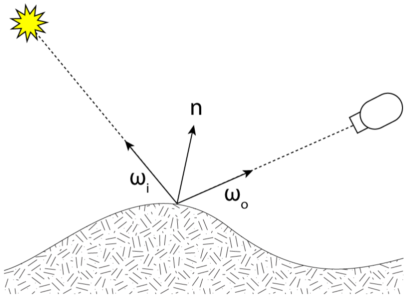
- Determine the light incident at x from all directions in the hemisphere $\rightarrow$ **ray tracing**.
- Determine the properties of the surface, which describe how incident light is reflected $\rightarrow$ **BRDF**[2].

[2] Although more general light reflection models exist, we will focus on the BRDF to keep it simple.

The BRDF

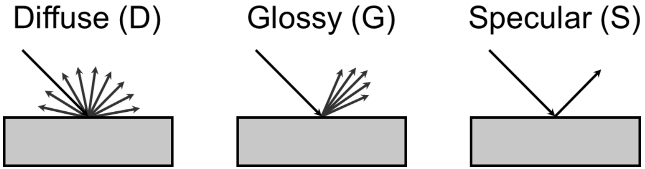
Q: How much light is reflected at point p in direction ω_o due to light coming in direction ω_i?

A: BRDF $f(p, \omega_o, \omega_i)$



The BRDF describes how light reflects off a surface.

Common BRDFs



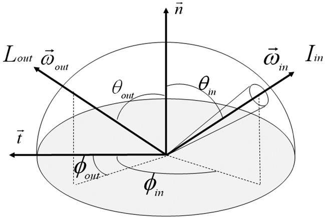
Specular reflects light in a single direction - the direction of mirror reflection.

Diffuse reflects light equally in all directions.

Glossy reflects light in a cone centered in the direction of mirror reflection.

The Rendering Equation

Determines outgoing radiance as a function of incoming radiance and surface BRDF.



$$L_o(p,\omega_o) = \int_\Omega f(p,\omega_o,\omega_i)\, L_i(p,\omega_i)\, \cos\theta_i \,\mathrm{d}\omega_i$$

$L_o(p,\omega_o)$ total outgoing radiance reflected at p in direction ω_o.

$L_i(p,\omega_i)$ radiance incident at p in direction ω_i.

$f(p,\omega_o,\omega_i)$ determines how much radiance is reflected at p in direction ω_o due to irradiance incident at p in direction ω_i.[3]

$\cos\theta_i$ Lambert's cosine law.

[3] For a full definition, see the BRDF section on Wikipedia.

Plan

Monte Carlo Integration

To compute the light that is reflected off a surface, we need to find a solution to the rendering equation.

An analytical solution does not generally exist, so we rely on methods that approximate the integral numerically.

Several methods exist to this end. However, we will focus solely on *Monte Carlo integration*.

Monte Carlo Integration

A technique for numerical integration that uses random numbers.

Pros

- Flexible and easy to implement: just need to evaluate the integrand at arbitrary points in order to evaluate the value of the integral.
- The convergence rate is independent of the number of dimensions in the integral. Monte Carlo integration does not suffer from the curse of dimensionality, where convergence rate grows exponentially with the number of dimensions.
- The number of samples is arbitrary; it does not depend on the dimensions of the integral.

Cons

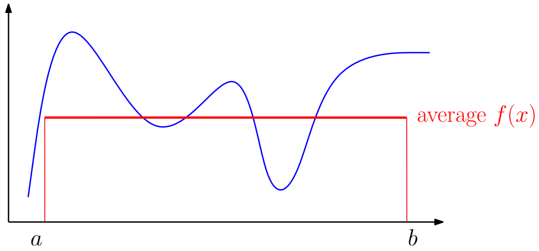
- Variance is proportional to $\frac{1}{\sqrt{N}}$, where N is the number of samples, i.e. if you want to reduce variance by half, you need to quadruple the number of samples.

Monte Carlo Integration

Given

$$I = \int_a^b f(x)\,\mathrm{d}x$$

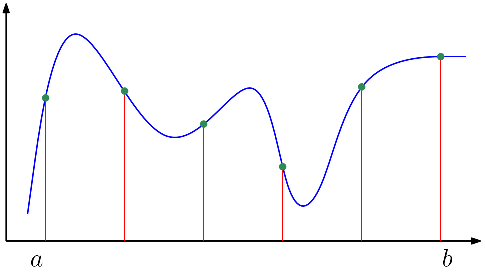
evaluate integral by taking the average value of $f(x)$ along the interval $[a, b]$ and multiply by the length of the interval $b - a$:



Monte Carlo Integration

To find the average value of $f(x)$, evaluate $f(x)$ at N different locations $X_1 \ldots X_N$, where $X_1 \ldots X_N$ are uniformly distributed random numbers in the interval $[a, b]$:

$$\bar{f}(x) = \frac{1}{N} \sum_{i=1}^{N} f(X_i)$$



Monte Carlo Estimator

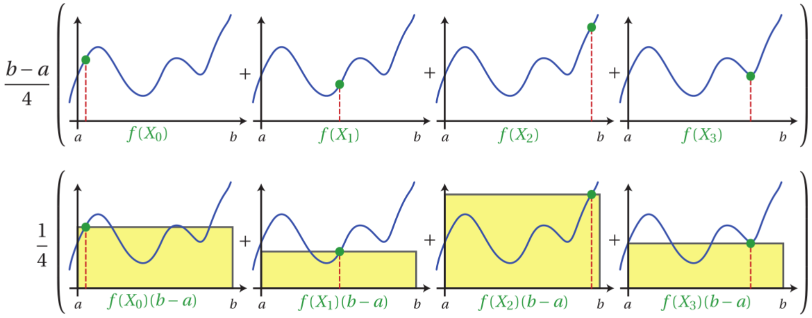
The average value of $f(x)$ multiplied by the length of the interval $b - a$ gives us the Monte Carlo estimator:

$$F_N = \frac{b-a}{N} \sum_{i=1}^{N} f(X_i)$$

In the limit, as we gather more and more samples, the Monte Carlo estimator takes on the value of the integral:

$$\lim_{N \to \infty} F_N = \int_a^b f(x) \, \mathrm{d}x$$

Monte Carlo Integration: Two Interpretations



Monte Carlo Estimator: Proof

The expected value of the Monte Carlo estimator is equal to the integral:

$$E[F_N] = \int_a^b f(x)\,\mathrm{d}x$$

Proof

Since $X_1 \dots X_N$ are uniformly distributed, their pdf $p(x)$ must be equal to $\frac{1}{b-a}$. Then...

Monte Carlo Estimator: Proof

$$
\begin{aligned}
E[F_N] &= E\left[\frac{b-a}{N}\sum_{i=1}^{N} f(X_i)\right] \\
&= \frac{b-a}{N}\sum_{i=1}^{N} E[f(X_i)] \\
&= \frac{b-a}{N}\sum_{i=1}^{N} \int_a^b f(x)\,p(x)\,\mathrm{d}x \\
&= \frac{1}{N}\sum_{i=1}^{N} \int_a^b f(x)\,\mathrm{d}x \\
&= \int_a^b f(x)\,\mathrm{d}x
\end{aligned}
$$

Monte Carlo Estimator: Generalised Version

The cool part about Monte Carlo integration is that the random variables X_i need not actually follow a uniform distribution. Monte Carlo integration works with any arbitrary pdf as long as the variables are independent and identically distributed (i.i.d).

Given $X_1 \dots X_N$ distributed according to pdf $p(x)$, the generalised Monte Carlo estimator is given by:

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}$$

Again, the expected value of the estimator is the value of the integral:

$$E[F_N] = \int_a^b f(x) \, \mathrm{d}x$$

Monte Carlo Estimator: Proof

Proof

$$
\begin{aligned}
E[F_N] &= E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right] \\
&= \frac{1}{N}\sum_{i=1}^{N} E\left[\frac{f(X_i)}{p(X_i)}\right] \\
&= \frac{1}{N}\sum_{i=1}^{N}\int_a^b \frac{f(x)}{p(x)}\, p(x)\,\mathrm{d}x \\
&= \frac{1}{N}\sum_{i=1}^{N}\int_a^b f(x)\,\mathrm{d}x \\
&= \int_a^b f(x)\,\mathrm{d}x
\end{aligned}
$$

Importance Sampling

Having the freedom to choose any arbitrary pdf $p(x)$ is very useful in path tracing.

Importance sampling allows us to better approximate the integral by choosing an appropriate pdf. The idea is to choose a pdf $p(x)$ that is similar to the function being integrated $f(x)$:

$$p(x) \sim f(x)$$

It can be shown that using importance sampling results in faster convergence than using an arbitrary pdf.

We will re-visit this concept later with specific examples.

Plan

Path Tracing 101

Rendering equation:

$$L_o(p,\omega_o) = \int_\Omega f(p,\omega_o,\omega_i)\, L_i(p,\omega_i)\, \cos\theta_i \,\mathrm{d}\omega_i$$

Monte Carlo estimator:

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}$$

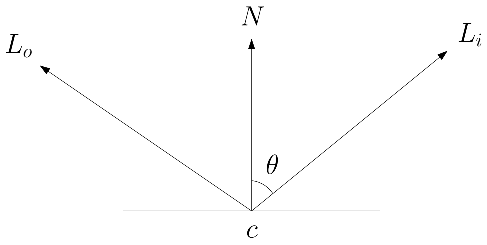
Monte Carlo path tracing:

$$\langle L_o(p,\omega_o)\rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{f(p,\omega_o,\omega_i)\, L_i(p,\omega_i)\, \cos\theta_i}{p(\omega_i)}$$

Path Tracing 101

- For every pixel, distribute N samples in that pixel.
- For every sample, generate a ray and trace a random path in the scene by randomly bouncing the ray around.
- For every bounce, compute the incoming radiance along the randomly reflected/refracted ray, weigh the result by the BRDF and divide by the pdf.
- Stop the recursion when you reach a maximum number of bounces or hit an emissive surface / light.
- Average the results of the N samples to produce the pixel's final colour.

Computing a Ray Bounce



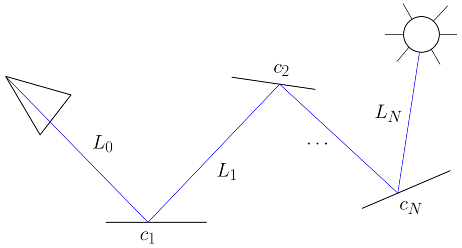
L_o is the outgoing radiance.

L_i is the incoming radiance along a randomly reflected ray.

Outgoing radiance for this sample is $c\,L_i$, where

$$c = \frac{f(p, \omega_o, \omega_i)\,\cos\theta}{p(\omega_i)}$$

Path Tracing



$$
\begin{aligned}
L_0 &= c_1 L_1 \\
&= c_1(c_2 L_2) \\
&= c_1(c_2(c_3 L_3)) \\
&= c_1(c_2(c_3(\dots(c_N L_N)\dots) \\
&= c_1 c_2 \dots c_N L_N \\
&= L_N \prod_{i=1}^{N} c_i
\end{aligned}
$$

Path Tracing

```
render_image():
  for each pixel:
    colour = 0
    for each sample:
      ray = camera_ray(pixel)
      colour += trace(ray)
    colour = colour / #samples
```

Path Tracing

```
trace(ray):
  coeff  = 1
  colour = 0
  for i in 1...max_depth:
    hit = intersect(ray, scene)
    if hit light:
      colour = coeff * emission
      return
    else:
      ray, brdf, pdf = random_sample(hit)
      coeff *= brdf * dot(normal, ray) / pdf
  if reachable(light):
    colour = coeff * emission
```

Path Tracing

```
ray, brdf, pdf = random_sample(hit)
```

The `random_sample` function must return:

- A randomly reflected ray.
- The value of the surface's BRDF for the in / out ray pair.
- The value of the pdf for the reflected ray.

This kind of random sampling will not work very efficiently in practise, however, so let us see why this is the case and improve the existing path tracer instead of going any deeper into the `random_sample` function...

Plan

(Multiple) Importance Sampling

For a basic path tracer to work properly, we should perform two optimisations:

- BRDF sampling
- Light sampling

These are two forms of importance sampling. For more details, see:

`http://www.slideshare.net/takahiroharada/introduction-to-bidirectional-path-tracing-bdpt-implementation-using-opencl-cedec-2015`

BRDF Sampling

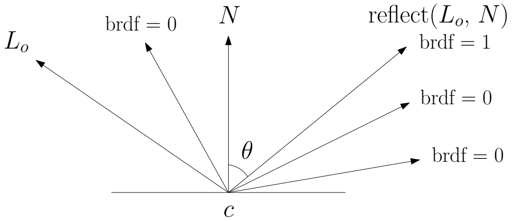
BRDF sampling is an application of importance sampling in Monte Carlo path tracing.

Recall from the Monte Carlo integration section that we are free to choose whatever pdf $p(x)$ we fancy to approximate an integral.

BRDF sampling chooses a pdf $p(x)$ that is similar to the surface's BRDF, resulting in faster convergence than just randomly shooting rays.

BRDF Sampling: Specular Reflection

Consider what happens when we shoot randomly reflected rays from a perfectly specular surface:



The specular BRDF is 1 in the direction of mirror reflection, and 0 everywhere else.

We cannot approximate the integral by shooting random rays; we need to sample the direction of mirror reflection explicitly.

BRDF Sampling: Specular Reflection

To sample a specular BRDF, we define the following function:

```
brdf_sample_specular():
  ray  = reflect(Lo, N)
  brdf = ks / dot(ray,N)
  pdf  = 1
```

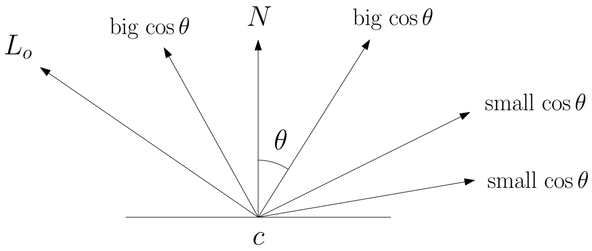
This function takes care of three things:

- Explicitly samples the direction of mirror reflection by always returning that direction.
- Returns the BRDF in that direction, which is *essentially** k_s, where k_s is the specular reflectance.
- Returns the pdf in that direction, which is simply 1.

* The $\frac{1}{\cos\theta}$ term in the BRDF (*1 / dot(ray, N)*) is there to cancel out the $\cos\theta$ term in the rendering equation.

BRDF Sampling: Diffuse Reflection

Similarly, consider shooting random rays from a diffuse surface:



Because of the $\cos\theta$ term in the rendering equation, rays at grazing angles have little contribution to the final result.

BRDF Sampling: Diffuse Reflection

For faster convergence on diffuse surfaces, we should focus more on rays near the normal than at grazing angles.

To this end, instead of sampling the hemisphere uniformly, we sample it according to a cosine-weighted distribution:

$$p(\omega) \propto \cos\theta$$

As usual, the pdf must add up to 1:

$$\int_\Omega k \cos\theta \, \mathrm{d}\omega = 1 \Leftrightarrow k = \frac{1}{\pi}$$

Therefore

$$p(\omega) = \frac{\cos\theta}{\pi}$$

BRDF Sampling: Diffuse Reflection

Again, we define a function to sample the BRDF:

```
brdf_sample_diffuse():
    ray  = cosine_weighted_hemisphere_sample(N)
    brdf = kd / pi
    pdf  = dot(ray,N) / pi
```

This function takes care of three things:

- Samples a random direction in the hemisphere according to a cosine-weighted distribution.
- Returns the BRDF in that direction, $\frac{k_d}{\pi}$, where k_d is the diffuse reflectance (i.e. surface colour or albedo).
- Returns the pdf in that direction, which is $\frac{\cos\theta}{\pi}$ as shown previously.

BRDF Sampling: Wrap-up

brdf_sample()	ray	BRDF	pdf
specular	reflect(L_o, N)	$\frac{k_s}{\cos\theta}$	1
diffuse	cosine_hemisphere()	$\frac{k_d}{\pi}$	$\frac{\cos\theta}{\pi}$

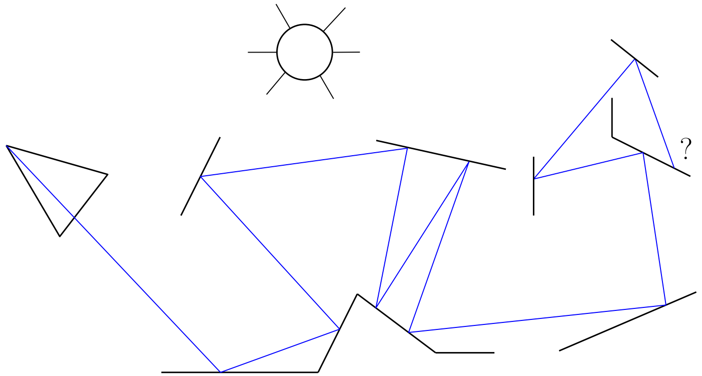
Note: for brevity, we have not shown how to derive the BRDF values above. See the `Additional Resources` section for more details.

BRDF Sampling

```
trace(ray):
  coeff  = 1
  colour = 0
  for i in 1...max_depth:
    hit = intersect(ray, scene)
    if hit light:
      colour = coeff * emission
      return
    else:
      ray, brdf, pdf = brdf_sample(hit)
      coeff *= brdf * dot(normal, ray) / pdf
  if reachable(light):
    colour = coeff * emission
```

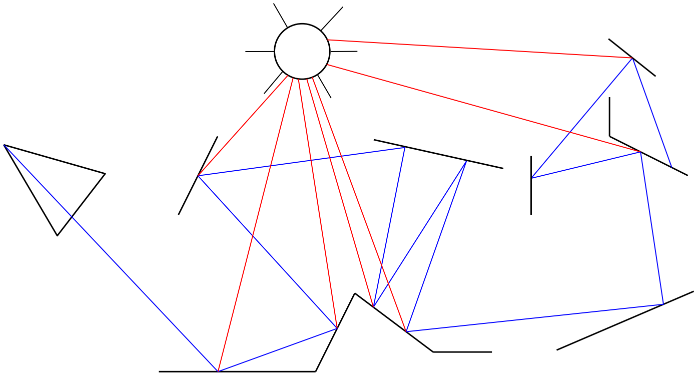
Light Sampling

It would be a shame to compute 20 light bounces only to realise that you have to return a radiance of zero simply because you cannot find a light...



Light Sampling

...so just sample the lights explicitly at every bounce (if a path exists):



Light Sampling

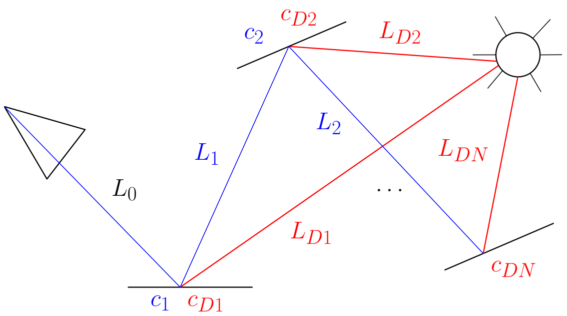
We define another set of functions that we will call `brdf_eval()` to evaluate the BRDF given an arbitrary pair of in / out ray directions.

- The out direction is the direction of outgoing radiance.
- The in direction is the direction towards the light source.

```
brdf_eval_specular():
  brdf = 0 // specular brdf is 0 for all directions
  pdf  = 0 // except direction of mirror reflection

brdf_eval_diffuse():
  brdf = k / pi             // diffuse brdf is constant
  pdf  = dot(ray,N) / pi    // for all directions
```

Path Tracing with Light Sampling



$$\begin{aligned}
L_0 &= c_{D1}L_{D1} + c_1 L_1 \\
&= c_{D1}L_{D1} + c_1(c_{D2}L_{D2} + c_2 L_2) \\
&= c_{D1}L_{D1} + c_1(c_{D2}L_{D2} + c_2(\dots(c_{DN-1}L_{DN-1} + c_{N-1}(L_{N-1} + c_{DN}L_{DN}))\dots) \\
&= L_{N-1}\prod_{i=1}^{N-1} c_i + \sum_{i=1}^{N} c_{Di}L_{Di}\prod_{j=0}^{i-1} c_j, c_0 = 1
\end{aligned}$$

Light Sampling

```
trace(ray):
  coeff  = 1
  colour = 0
  for i in 1...max_depth:
    hit = intersect(ray, scene)
    if hit light:
      colour = coeff * emission
      return
    if reachable(light):
      brdf, pdf = brdf_eval(hit, light_ray)
      c = coeff * brdf * dot(normal, ray) / pdf
      colour += c * emission
    ray, brdf, pdf = brdf_sample(hit)
    coeff *= brdf * dot(normal, ray) / pdf
```

Russian Roulette

Currently, our path termination strategy is not very intelligent: if a path has little contribution to the result, it makes little sense to compute additional bounces.

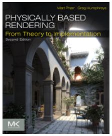
In addition, we have introduced bias in our path tracer by cutting the ray length to a given depth (i.e. we are not sampling all possible light paths).

Russian Roulette is an importance sampling technique that allows us to focus on paths that have an important contribution to the result by randomly terminating rays.

In addition, Russian Roulette gives us, on average, the same value as if we were tracing paths of infinite length, yielding an unbiased result.

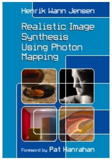
Plan

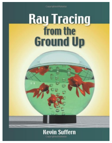
Additional Resources


Matt Pharr · Greg Humphreys
PHYSICALLY BASED RENDERING
From Theory to Implementation
Second Edition
MK

Physically Based Rendering: From Theory To Implementation
http://www.pbrt.org/

Realistic Image Synthesis Using Photon Mapping
http://graphics.ucsd.edu/~henrik/papers/book/





Ray Tracing from the Ground Up
http://www.raytracegroundup.com/

Additional Resources

Mathematical Foundations of Monte Carlo Methods
http://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-mathematical-foundations

Monte Carlo Methods in Practice
http://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice