Vikram S.Nerurkar Lecturer: Steve Marschner

1 Introduction

Path tracing is a name for a class of global illumination algorithms that construct paths (sequence of surface points) that connect the source and camera.

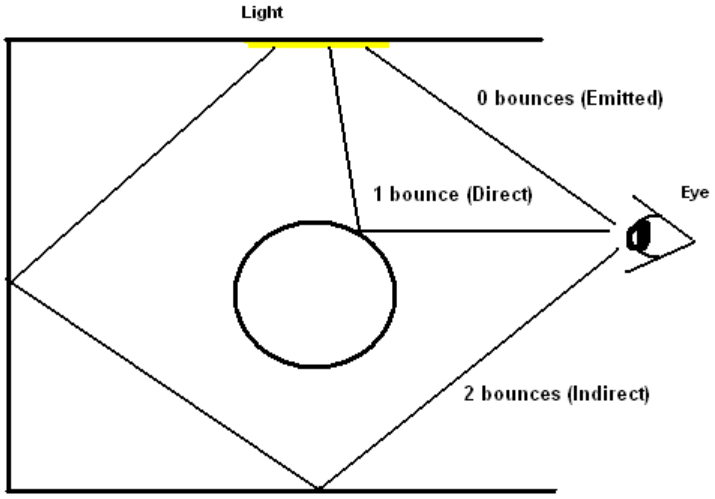

Figure 1: Basic Paths

1.1 Nomenclature for Paths

Its handy to be able to categorize different kinds of paths that behave differently. This is often done using Heckberts notation as follows:
A path is written as a string of characters -

L - for Light Source
E - for Eye
D - for Diffuse reflection
S - for specular reflection or transmission

1.2 Categorization of Paths

Several different kinds of behaviors exist for Reflectance Models. They are categorized as follows: For instance returning to the scene of a bit with a glass ball in it

As shown in the above figure, different paths can be classified as -

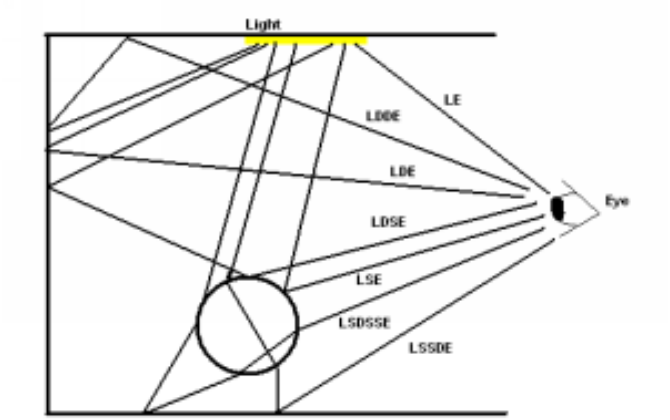

Figure 2: Standard Paths

LE - Emitted
LDE - Diffuse Shading(Direct Illumination)
LSE - Specular Highlight (Direct Illumination)
LD*E - Diffuse Inter-Reflection
LDS*E - Reflected Image of Diffuse
LS*DE - Directly Viewed Caustic
LS*DS*E - Swimming Pool Scene

Different types of paths are handled well by different algorithms, and some are left out by some algorithms. An unbiased global illumination renderer should account for all paths, but it is not so easy to account for all paths well.

Standard Paths are - LE, LDE, LSE (In General L (D|S)? E)

Simple Paths are - LD*S*E work OK

Caustic Paths - $LS^{+}DE$ work badly

Note that interposing a flat sheet of glass between a surface and a source, even though it hardly affects the answer turns lesser direct illumination paths into (hard) caustic paths and breaks most renderers.

2 Basic Path Tracing Algorithm - Kajiya Style

The Basic idea is to compute local illumination but use a recursive call to evaluate incident radiance (rather than including only direct.)
This is therefore a from-the-eye kind of method and the outer loop looks just like any other ray tracer.

2.1 PseudoCode for a Naive Path Tracer

The following gives the pseudo code for a simple path tracer.

```
pixelValue(camera,i, j)
   result = 0;
   for k = 1 to N
      x, w = camera.chooseRay(i, j); //uniform on a unit square or prop to filter
      result += rayRadianceEst(x, w);
   return result;

rayRadianceEst(x, w)
   y = traceRay(x, w);
   return emittedRadiance(y, -w) + reflectedRadianceEst(y, -w);

reflectedRadianceEst(x, w_r)
   w_i = uniformRandomPSA(normal(x));
   return π * brdf(x, w_i, w_r) * rayRadianceEst(x, w_i);
```

This program will (almost) render unbiased pictures. There is one problem that will keep it from producing any output: the recursion has no way to terminate.

2.2 Termination Techniques

A termination criterion has to be added to the naive path tracer to make it practically useful.
We could wait for paths to terminate by hitting the background on encountering non-reflective surfaces (which would be handled specially). Neither of these come with any guarantees though.
One of the following techniques can be used:

2.2.1 Simple Termination Scheme - Terminate Procedurally

For Less-Physics Based Recursive Ray-Tracers, there are simple termination schemes like:

- depth > maxDepth
- attenuation < minAttenuation

Both of these introduce bias, though, what if the next bounce was going to hit the sun? Thus, in Physics-Based Rendering Systems they should not be implemented.

2.2.2 Russian Roulette - Terminate Probabilistically

A clever way around this is to terminate probabilistically.ie- With some probability $0 < p < 1$, go ahead and evaluate the next bounce. With probability (1-p) stop and return zero. With no special treatment, the expected value of this estimator is p times the correct value. The widely used solution to this is to scale the result up by $1/p$. This is analogous to Monte Carlo Type Integration. This works for any value you want to sometimes treat as zero:

Let X be a Random Variable with $E\{X\} = \mu$

$$Y = \begin{cases} \text{X/p} & \text{with Probability } p, \\ 0 & \text{with Probability (1-}p\text{)} \end{cases}$$

$$E\{Y\} = p\ E\{X/p\} = E\{X\} = \mu$$

This works as long as p is independent of X, you can use whatever other information you want.

2.3 Simple Functional Path Tracer

Adding a termination criterion into the existing Path Tracer gives us a working version as follows:

```
pixelValue(camera,i, j)
   result = 0;
   for k = 1 to N
      x, w = camera.chooseRay(i, j); //uniform on a unit square or prop to filter
      result += rayRadianceEst(x, w);
   return result;

rayRadianceEst(x, w)
   y = traceRay(x, w);
   return emittedRadiance(y, -w) + reflectedRadianceEst(y, -w);

reflectedRadianceEst(x, w_r)
   if (random() < survivalProbability)
      w_i = uniformRandomPSA(normal(x));
      return π * brdf(x, w_i, w_r) * rayRadianceEst(x, w_i) / survivalProbability;
   else
      return 0
```

3 Better Path Tracer

This will produce pictures but not so efficiently. It really needs to take advantage of knowing about where the light sources are. We already saw how to compute direct illumination from sources. The most visual way to do this is to split the integral into direct and indirect, and evaluate the two separately:
We have:

$$L_r(w_r) = \int_{\mathcal{H}_i} f_r(w_i, w_r)\ L_i(w_i)\ d\mu(w_i)$$

We can write $L_i = L_i^0 + L_i^+$

$$L_r(w_r) = \int_{\mathcal{H}_i} f_r(w_i, w_r)\ (L_i^0(w_i) + L_i^+(w_i))\ d\mu(w_i)$$

$$L_r(w_r) = \int_{\mathcal{H}_i} f_r(w_i, w_r)\ L_i^0(w_i)\ d\mu(w_i) + \int_{\mathcal{H}_i} f_r(w_i, w_r)\ L_i^+(w_i)\ d\mu(w_i)$$

We already know how to compute the first integral. The second one is the same as the basic algorithm except we need to ignore emitted light. We can rewrite our radiance estimator as follows

3.1 Pseudo Code for the Better Path Tracer

```
pixelValue(camera,i, j)
   result = 0;
   for k = 1 to N
      x, w = camera.chooseRay(i, j); //uniform on a unit square or prop to filter
      result += rayRadianceEst(x, w);
   return result;

rayRadianceEst(x, w)
   y = traceRay(x, w);
   return emittedRadiance(y, -w) + reflectedRayRadianceEst(y, -w);

reflectedRayRadianceEst(x, w)
   y = traceRay(x, w);
   return reflectedRadianceEst(y, −w);

reflectedRadianceEst(x, w_r)
   return directRadianceEst(x, w_r) + indirectRadianceEst(x,w_r);

directRadianceEst(x, w_r)
   //This is old hat

indirectRadianceEst(x,w_r)
   if (random() < survivalProbability)
      w_i = uniformRandomPSA(normal(x));
      return π * brdf(x, w_i, w_r) * rayRadianceEst(x, w_i) / survivalProbability;
   else
      return 0
```

This is the Kajiya's Path Tracing Algorithm which produces a vine-like path from the source to the eye.

4 Some Interesting Points

4.1 Neumann Expansion

Neumann's Expansion for operators with Spectral Radius < 1 is given as:
$(1 - \mathbb{M})^{-1} = 1 + \mathbb{M} + \mathbb{M}^2 + \mathbb{M}^3$
The series converges as $\mathbb{M}$ is lesser than unity.

4.2 Representation of Path Tracing Equation

The Path Tracing Equation as described in the earlier algorithm can be represented as $L_r = \mathbb{K}\mathbb{G}L_r + L_e$ which can be formally written as

$L_r = (1 - \mathbb{K}\mathbb{G})^{-1} L_e$
The RHS can be expanded algebraically (analogous to the Neumann Series) to get

$$L_r = \left[1 + (\mathbb{K}\mathbb{G}) + (\mathbb{K}\mathbb{G})^2 + (\mathbb{K}\mathbb{G})^3 + ...\right] L_e$$
$$L_r = L_e + \mathbb{K}\mathbb{G}\ L_e + (\mathbb{K}\mathbb{G})^2\ L_e + (\mathbb{K}\mathbb{G})^3\ L_e + ...$$

where

L_e gives the emittedRadiance
$(\mathbb{K}\mathbb{G}\ L_e)$ gives the Direct Illumination(Path of type LDE or LSE)
$\left[(\mathbb{K}\mathbb{G})^2\ L_e + (\mathbb{K}\mathbb{G})^3\ L_e\right]$ gives the Indirect Illumination(Path of type other than LDE and LSE)

Thus the answer is a sum of integrals, one for each path length.
Another way of writing this is

$$L_r = L_e + \mathbb{K}\mathbb{G}(L_e + \mathbb{K}\mathbb{G}(L_e + \mathbb{K}\mathbb{G}(....)))$$

This is very much like our recursive style of generating paths

4.3 Relationship with Multiple Importance Sampling

In the above algorithm, we have effectively divided the calculations into those for direct illumination and indirect illumination. Another way to deal with such a split is to use Multiple Importance Sampling.

reflectedRadianceEst(x, w_r)
$w_i^1\ p_1^1$ = randomDirToLight();
$w_i^2\ p_2^2$ = uniformRandomPSA(); //$p_2^2 = 1/\pi$
p_i^2 = probToLight(w_i^2)
$p_2^1 = 1/\pi$
return $\pi * brdf(w_i^1, w_r) * rayRadianceEst(x, w_i^1)/(p_1^1 + p_2^1)$
$+\pi * brdf(w_i^2, w_r) * rayRadianceEst(x, w_i^2)/(p_1^2 + p_2^2)$

The principal difference between the earlier approach and this one is that both rays now include both indirect and direct, so it is not a Path Tracer.

4.4 References

Interesting Further Readings could be
http://graphics.stanford.edu/courses/cs348b-01/course29.hanrahan.pdf