PATH TRACING: A NON-BIASED SOLUTION TO THE RENDERING EQUATION

ROBERT CARR AND BYRON HULCHER

Abstract. In this paper we detail the implementation of a path tracing renderer, providing a non-biased solution to the rendering equation. Path tracing is an algorithm, which in attempting to model the behavior of light as closely as possible, accurately captures effects such as: Caustics, Soft Shadows, and Ambient Occlusion. The algorithm may be extended with techniques from distribution ray tracing in order to simulate effects such as glossy reflections, depth of field, and motion blur. In this paper we detail a practical implementation of a C++ path tracer.

Part 1. Background

Introduction

Rendering techniques may be largely divided in to two categories: Real time non-photorealistic algorithms (such as scanline type approaches), and photorealistic methods attempting to accurately simulate the transport of light.

Photorealistic rendering methods are largely described by the ray tracing category of algorithms, introduced in Whitted, 1980[8]. In nature light is emitted from

Date: Mon, May 10. 2009.
Key words and phrases. Path Tracing, Rendering Equation, Ray Tracing.

Figure 0.1. Image produced by path tracer illustrating caustic effects.

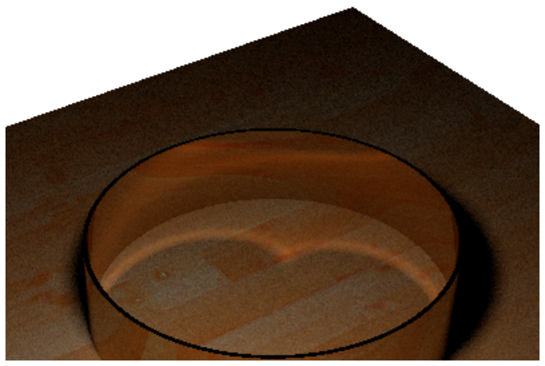
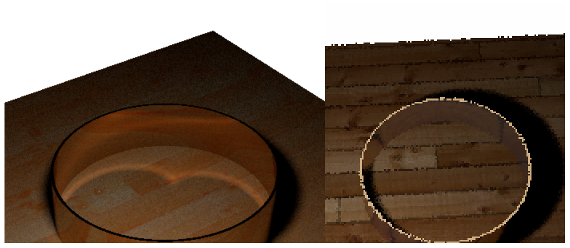


FIGURE 0.2. Comparison of path and ray traced images, note the caustic effects present in the path traced result.



a light source, and eventually through a series of surface interactions (absorption, reflection, refraction, or fluorescence) may reach the a viewer. Ray tracing approximates this interaction by generating rays from the eye, to each pixel, and finding which object we intersect. Then by using object properties, the shading at that point may be determined. The "genius" in the algorithm, is perhaps it's recursive nature, allowing intersections to spawn reflection, refraction, or shadow rays, in order to model some global effects. Ray tracing, while able to create very realistic images, fails to capture many non local effects, such as: Caustics, and Color Bleeding. Furthermore, while ray tracing may simulate effects such as soft shadows[1], we are perhaps now begining to deviate from our goal of photorealism.

The Rendering Equation

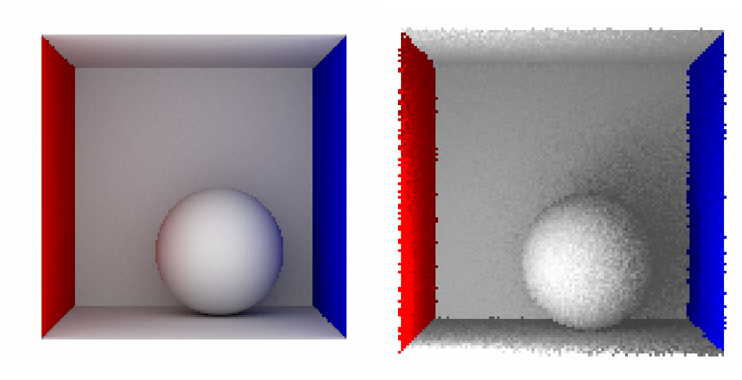
The rendering equation refers to an integral equation for computing the radiance at a point from a certain direction. The rendering equation was introduced to the field of computer graphics in Kajiya's 1986 SIGRAPH paper[4]. Given in the following form:

$$L_0(x,\omega,\lambda,t) = L_e(x,\omega,\lambda,t) + \int_\Omega f_r(x,\omega',\omega,\lambda,t)L_i(x,\omega',\lambda,t)(-\omega'\cdot\mathbf{n})d\omega'$$

While recent to computer graphics, the physical basis for the equation is relatively simple, and based on the law of conservation of energy. We say that the radiance at a point viewed from a particular direction, is the sum of the emitted light, and the reflected light, where the reflected light is the sum of all incoming light, shaded by the surface properties, and multiplied by the cosine of the incident angle.

True photorealism is approached when solving, or closely approximating the rendering equation. Such an approach will allow us to accurately capture global illumination effects based on the interaction of diffuse surfaces, an element missed in the classical ray tracing algorithm. The most popular approach (preferred for performance) is based on finite element methods, and is referred to as the radiosity algorithm[3]. This is often used to supplement an existing ray tracerIn this paper however, we intend to explore the Monte Carlo methods of solution, perhaps best characterized by the path tracing algorithm.

FIGURE 0.3. Ray and Path traced renderings of the Cornell box. Note the color bleeding effects visible in the path traced rendering. This effect requires modelling of the interaction of light from light sources between diffuse objects.



PATH TRACING

The path tracing algorithm is described in it's earliest form, in Kajiya's 1986 paper on The Rendering Equation[4], and provides a Monte Carlo method of solving the Rendering Equation. Path tracing operates in a similar manner to ray tracing, however in fact traces the entire path of light rays: From the camera to a light source. It is important to distinguish this from the ray tracing approach. In ray tracing, upon intersection with a diffuse surface, lights are directly sampled in order to determine shading. In path tracing, we spawn a random ray within the hemisphere of the object, and trace until we hit a light. In general it is quite unlikely that a path will intersect a light, and so we continue to refine our image by averaging multiple image samples. This is perhaps the curse of path tracing, in that most traced paths do not contribute to our rendering, and we must use a large number of samples to obtain a suitably noise free image.

Path tracing however, has many advantages to other rendering algorithms as well, this can be demonstrated by considering the kinds of paths that certain ray tracing techniques, using a convenient notation proposed by Peter Shirley (can't find original citation). We will describe components of paths as follows:

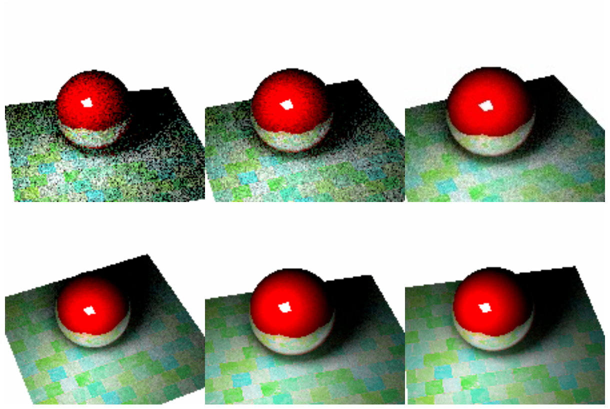
E: the eye.
L: the light.
D: diffuse interaction
G: glossy reflection
S: specular reflection or refraction

Then we can describe the paths generated by algorithms using a regular expression notation:

Ray Casting	Recursive Ray Tracing	Radiosity	Path Tracing
$E(D\|G)L$	$E[S^*](D\|G)L$	ED^*L	$E[(D\|G\|S)^+(D\|G)]L$

We can see then for example, that while recursive ray tracing will trace paths of any length, all must begin with a mirror reflection or refraction. Furthermore consider a path of the form $E(D|G)S^*L$, describing the paths which lead to caustic

FIGURE 0.4. Comparison of renderings at: 100, 250, 1000, 2500, 5000 and 10000 samples.



effects. Even radiosity fails to generate these effects[1]. Path tracing is shown to have an unbiased nature in Kajiya, that is to say as the number of samples approaches infinity, the solution converges correctly. In our experience, while an image may become quickly recognizable, it may take many thousand samples to achieve an acceptably low level of noise.

Part 2. Basic Implementation

Scene Definition and data structures

Before implementing a path tracer, we must first have a way of defining what we wish to render. Our implementation contains a `RayTracer` object, which contains a mesh composed of primitives. Primitives are assigned materials, which describes the diffuse, reflective, and emittive colors of the object, along with the index of refraction. Primitives implement an `Primitive::Intersect`method, which is used to determine intersection of a ray with primitive, and if applicable return hit information. For primitives such as spheres, this is a simple matter of substiting equations and is described in more detail in the recursive ray tracing paper[8].

Rendering Loop

The overall rendering loop of our tracer, is described in pseudo code as follows:

```
def TracePixel(i, j):
        r = generate_ray(i,j)
        return TracePath(r)

def render():
     samples[width*height] = {0,}
     for number_of_samples:
```

[1]Though this may be approximated using techniques such as Photon Mapping as in “Global Illumination Using Photon Maps”, Henrik Wann Jensen, 1996.

```
        for i = 0 to height:
                for j = 0 to width:
                        samples[width*i+j]+=
                    TracePath(i,j)/number_of_samples
    save_samples();
```

In addition, we keep track of a running average of time per sample in order to compute progress estimates.

Diffuse Interactions

To implement the logic of path tracing, we will start with diffuse interactions, the algorithm dictates that when impacting a diffuse surface, we emit a ray in a random direction constrained within the hemisphere of our surface normal.

This may be described as follows:

```
def CastRay(ray):
    # Intersect ray with scene primitives
    # and return hit information for
    # nearest intersection.

def TracePath(ray, bounce_count):
    (hit,material) = CastRay(ray)
        answer = Color(0,0,0)
        if hit == null:
                return answer
    if material.emittance.Length() > 0:
        return emittance;
        if material.diffuse.Length() > 0:
                dir = RandomDirectionInHemisphere(hit.normal)
                n = Ray(point+dir*.00001,dir)
                dp = dir.Dot3(hit.normal)

                answer += TracePath(n,bounce_count+1)*
                            matterial.diffuse*dp
    return answer
```

Some points of note:

- When constructing our new ray, the slight offset is used to avoid floating point error
- Incoming light is shaded according to cosine, as in the rendering equation.

Already we can begin to create some renderings:

Mirror Reflections

To calculate the reflected ray $\vec{R}$ from the intersection of a ray (with incident direction $\vec{V}$) with a surface (with normal at intersection $\vec{N}$), we use the following formula:

$$\vec{R} = \vec{V} - 2(\vec{V} \cdot \vec{N})\vec{N}$$

FIGURE 0.5. A rendering of the Cornell Box at 1000 samples, note the effects from diffuse surface interaction.

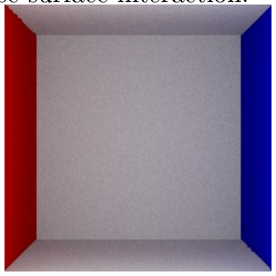
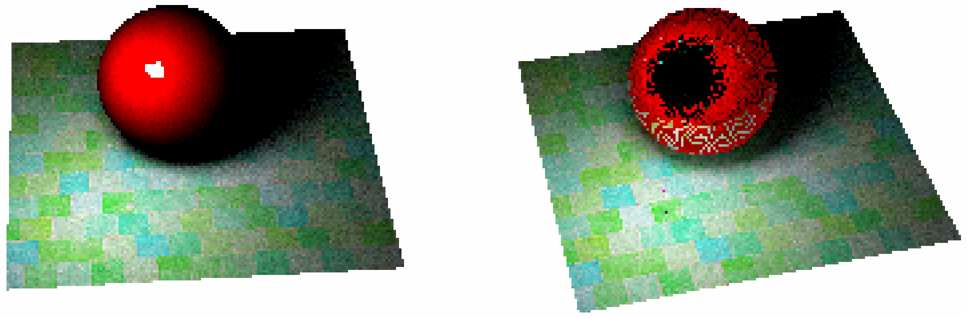


FIGURE 0.6. A Path Traced reflective sphere, and an example of floating point error caused by not using an epsilon.



In the framework of our code this looks something like the following addition to our `TracePath` function.

```
if material.reflection.Length() > 0:
    dir = ray.direction - (2*ray.direction.dot(hit.normal))*hit.normal
    n = Ray(point+dir*.00001, dir)
    answer += TracePath(n,bounce_count+1)*material.reflection
```

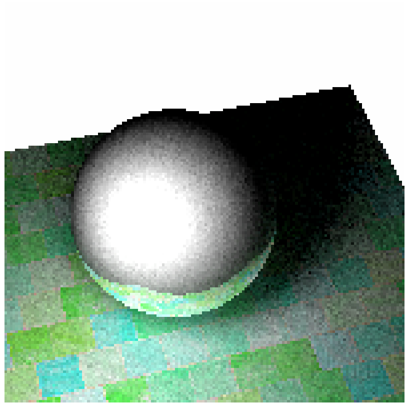
We can now produce a rendering demonstrating our results:

Refraction

Refraction refers to the effect of a change in direction in wave due to a change in speed, perhaps most commonly observed in light passing from one medium to a second. The bending of the rays depends on a property of the two mediums referred to as the refractive index, defined for a medium m as:

$$\eta = c/c_m$$

FIGURE 0.7. A refractive sphere



with c denoting the speed of light in a vacuum, and c_m denoting the speed of light in m. For example, air has a refractive index of approximately 1.0003, while the human cornea has an index of 1.38[6]. For a detailed discussion of the mathematics, we found the paper "Reflections and Refractions in Ray Tracing"[2] to provide a satisfactory description. To implement this in our TracePath, we must first extend the function with an additional argument: `last_index`, or the index of the material which a spawned ray will travel through, we give this a default value of 1.0003 for air.

To compute the correct normal, it is important to differentiate from intersections entering an object from another material, and intersections originating from inside the same object, we do this by having our intersection routine return an integer value: 0 for miss, 1 for hit, and -1 for hit inside the primitive.

Implementation in our TracePath function is as follows:

```
if (hit.material.refraction != 0):
    index = hit.material.refraction
    n = last_index / index
    N = hit.normal * intersect;
    cosI = -N.Dot3(ray.direction)
    cosT2 = 1.0-n*n*(1.0-cosI*cosI)
    if cosT2 > 0.0:
        dir = (n*ray.direction)+(n*cosI-sqrt(cosT2))*N
             n = Ray(point+dir*.00001, dir)
             answer += TracePath(n,bounce_count+1,index)
```

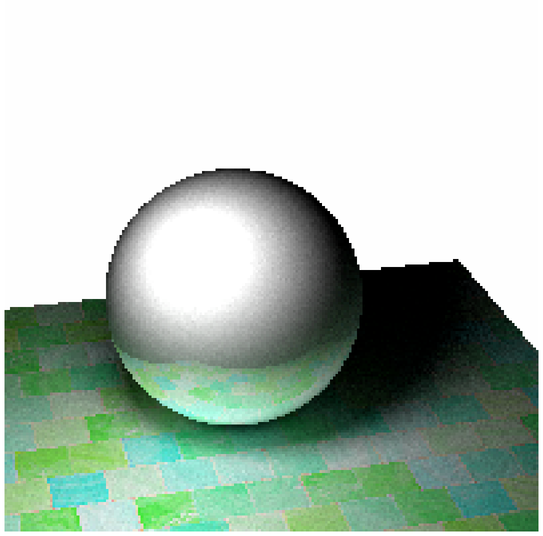
An example image demonstrating refraction:

Part 3. Implementation Improvements

RUSSIAN ROULETTE

In order to terminate paths which are likely to never hit lights, we implement a Russian Roulette, which introduces a random chance of killing paths with length

FIGURE 0.8. A refractive sphere with Beer's Law



greater than a variable number of bounces, this looks something like the following at the beginning of our `TracePath`

```
if ((bounce_count > max_bounces)
    && (RandomDouble() < max(diffuse.r,
                        diffuse.g, diffuse.b):
   return Color(0,0,0)
diffuse = diffuse * 1/p
```

BEERS LAW

In fact, our prior implementation of refraction was somewhat incomplete. To accurately shade our materials, we must respect Beer's Law, which states that the effect of a colored medium is stronger over longer distances. This involves changing our refraction calculation as:

```
rcol,distance = TracePath(n,bounce_count+1,index)
absorb = diffuse*0.15*-distance
transparency = Color(exp(absorb.x),exp(absorb.y),exp(absorb.z)
```

Implementing this dramatically improves results as shown in Figure 0.8.

AMBIENT LIGHTING

During testing, we often found it was frustrating to slowly wait for images to converge, and so we implemented an ambient lighting term, as follows in TracePath:

```
if intersect == false:
   if bounce_count >= 1:
      return ambient_light
   else:
      return Color(0,0,0)
```

FIGURE 0.9. Reflective ring at 50 samples with and without ambient lighting term.

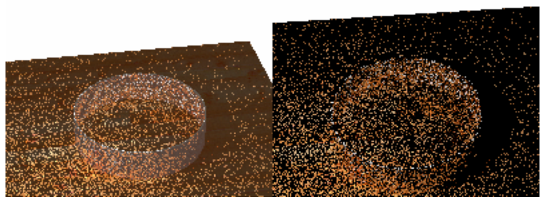
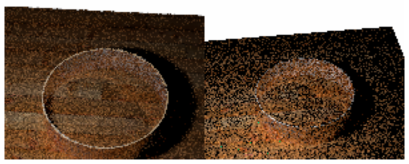


FIGURE 0.10. Rendering of reflective ring at 200 samples with and without hybrid rendering.



While drastically reducing the quality of images at higher noise ratios, it can be used to obtain quickly converging results while testing, as in Figure 0.9.

Hybrid Tracing

For another attempt at improving convergence, we implemented a hybrid tracing method, which generates a final image by averaging a recursively ray traced image, and a path traced image. This allows us to quickly create ray traced images preserving some global effects:

Distribution Ray Tracing Effects

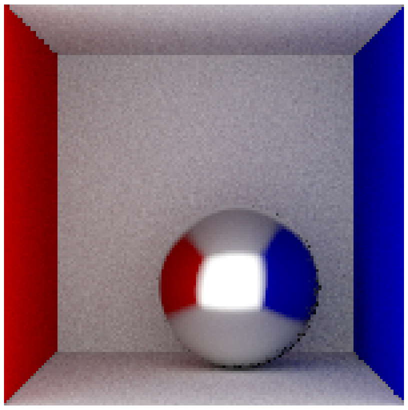
The 1984 paper "Distributed Ray Tracing" describes a refinement of the ray tracing algorithm allowing it so simulate soft phenomena[1]. Essentially by oversampling various ray parameters, various effects are described.

Glossy Reflections. In the real world, objects rarely reflect in perfect mirror fashion, consider for example the surface of a matte-poster, likely to give a very blurry reflection. This is the material property referred to as: Gloss. To simulate gloss as described by Cook, we scatter a number of reflection samples in a cone around the mirror direction to compute our reflected color. Each material is given a gloss property between 0 and 1, indicating the degree of scattering (with 0 being mirror reflection, and 1 being in fact a diffuse surface (with enough samples...).

Additional Distribution Ray Tracing Effects. Many other effects can be implemented with a similar technique:

- Depth of field, by sampling the ray starting positions over a lens.
- Antialiasing, by multisampling within pixels.

FIGURE 0.11. Cornell box, containing a sphere with a gloss coeffecient of .3



- Motion blur, by multisampling over time.

However, we ran out of time to complete these additional effects for our project.

Parallel Rendering

With recent versions of the g++ compiler, path tracing samples will automatically be computed in paralllel with our implementation, this is achieved via the use of the OpenMP framework, with the following pragma annotating our outer rendering loop:

```
#pragma omp parallel for schedule(static)
```

Near linear speedud with number of processors is observed.

Part 4. Implementation Notes

Future Possibilities

Despite the core simplicity of the path tracing algorithm, there is a lot left that we could implement. Several algorithms based on path tracing exist, designed to improve the convergence time including bidirectional path-tracing[5], an algorithm which uses a combination of eye->light and light->eye rays to accelerate convergence, and Metropolis light transport[7], an algorithm which perturbs previously computed paths to improve convergence.

A clear gap in our path tracer is proper support of distribution effects, such as depth of field and antialiasing. With the long render times however, we just ran out of time to properly implement and test these features. I believe that multisample antialiasing would drastically improve the resulting image quality.

Several "conveniences" could be added to the codebase, such as an iterative preview of the image up to our current sample.

References

[1] Porter Cook and Carpenter. Distributed ray tracing. 1984.
[2] Bram de Greve. Reflections and refractions in ray tracing. 2004.
[3] Goral et al. Modeling the interaction of light between diffuse surfaces. 1984.

[4] Kajiya. The rendering equation. 1986.
[5] Eric P. Lafortune. Bi-directional path tracing. 1993.
[6] robinwood.com. Refractive index list of substances.
[7] Eric Veah and Leonidas J. Guibas. Metropolis light transport. 1997.
[8] Turned Whitted. An improved illumination model for shaded display. 1980.

E-mail address: `racarr@gnome.org, byronhulcher@gmail.com`