# Algorithms for Fast Rendering

Adam Yedidia

January 13, 2017

## 1 Introduction

This essay is a follow-up to my previous writeup *Locating a Target in Three Dimensions*. It draws upon many of the same ideas, and concerns itself with the same structure of problem.

Just as in the previous problem, the question of interest is inferring what lies behind an obstacle, from the visible reflections on a Lambertian surface. A lot of what has been done for this problem, including my own work, treats the solution to the problem as a two-step recipe:

1. Construct a forward model. I call this "rendering": given a description of the scene, render a movie that shows the number of photons that arrive over time for each pixel.
2. Invert the forward model. I describe some ideas for this in the Search section of *Locating a Target*. There are a variety of ideas here, including convolutional neural nets.

This two-step process is a typical way to attack the problem. For example, in *Transient Imaging for Real-Time Tracking Around a Corner* by Laurenzis, Hullin, and Klein, this is how they describe their approach, and the primary contribution of their work is their fast rendering with GPU's. This write-up takes a similar approach. I'm not going to talk at all about inverting the forward model. Instead, I'm just going to talk about how to do a better job rendering, and leave the problem of inverting the forward model for later, since it's almost a completely separate consideration.

## 2 Line- or Strip-based Computation

In my previous write-up *The Point-Plane Problem*, I described how it was possible to find a closed form for the intensity response over time of a point source reflecting off of an infinite plane. Here, I will use a very similar approach to find a closed form for the intensity response over time for a line, approximated in three dimensions as a very thin strip. This turns about to be a much more practically useful formula, because of the fact that any two-dimensional manifold can be broken down into "strips."

### 2.1 Overview of Computation Process

In this problem, we imagine that our point source is at the location $\vec{p_s} = (0, 0, 0)$, and that our point detector is at the location $\vec{p_d} = (x_d, 0, 0)$. Our detector is assumed to be a small patch of side-length $\Delta x$, and normal vector $\vec{v_d}$.

Let's suppose that the strip lies along the line that connects the points $\vec{p_0} = (x_0, y_0, z_0)$ and $\vec{p_1} = (x_1, y_1, z_1)$, and has a normal vector of $\vec{v_l}$. We'll suppose that if this line extended infinitely in both directions, that $(x_0, y_0, z_0)$ would be the point of first light. Let's also suppose that a time $t$ has passed since the pulse of light was emitted, we know the following. We can describe where we are along the line using a parameter $k$:

$$x = x_0 + k(x_1 - x_0)$$

$$y = y_0 + k(y_1 - y_0)$$

$$z = z_0 + k(z_1 - z_0)$$

Note that $k = 0$ indicates that the light is currently hitting $(x_0, y_0, z_0)$, and $k = 1$ indicates that the light is currently hitting $(x_1, y_1, z_1)$.

$t = \sqrt{x^2 + y^2 + z^2} + \sqrt{(x - x_d)^2 + y^2 + z^2}$

These equations easily imply how to compute $t$ as a function of $k$. Unfortunately, the thing that we are going to want is the other way around—given a $t$, we want to compute an intensity $I(t)$, and our formula for $I(t)$ is going to depend on $k$. Therefore, we'd like a solution for $k$ in terms of $t$.

Fortunately, this system of equations can be solved by Mathematica. Although it is unwieldy and I do not reprint it here, the solution is a polynomial (including radicals) of about 30 terms.

The process for computing the intensity as a function of time is the same basic approach as in *The Point-Plane Problem.* We can split the intensity function into two terms:

$$I(t) = N_p(t) \cdot I_p(t)$$

where $N_p(t)$ describes the number of "paths" from the source to the strip—in other words, the number of individual points on the strip that the light hits. In order to do this, we'll want to say that a point on the strip is a little patch on the wall of size $\Delta x \times \Delta x$. $I_p(t)$ describes the intensity per path—for each point hit by a beam, how much intensity is contributed? This will give us the total intensity function exactly, assuming $\Delta x$ is small.

As a function of $k$, what is the number of paths? Well, the number of paths as a function of time should be the derivative of the cumulative number of paths since first light. The number of paths since first light, of course, is easily computed: get the $k(t)$ as described above, and then compute the current point on the line that the light is hitting:

$$x = x_0 + k(x_1 - x_0)$$
$$y = y_0 + k(y_1 - y_0)$$
$$z = z_0 + k(z_1 - z_0)$$

Then the distance along the line that has been travelled so far, divided by $\Delta x$, is the cumulative path count. So:

$$N_c(t) = \frac{\sqrt{(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2}}{\Delta x}$$

Now that we can compute the cumulative path count as a function of $t$, we can get the paths as a function of time by taking the numeric derivative:

$$N_p(t) = \frac{N_c(t + \Delta t) - N_c(t)}{\Delta t}$$

Finally, we can get the per-path intensity quite straightforwardly: because the set of points being hit by light at any given point in time is small in all three dimensions, the basic assumption of intensity constancy is guaranteed to hold.

$$I_p(t) = I_{p1}(t) \cdot I_{p2}(t)$$
$$I_{p1}(t) = \frac{(\Delta x)^2}{2\pi(x^2 + y^2 + z^2)} \left| \frac{(\vec{p_s} - \vec{p}) \cdot v_l}{||\vec{p_s} - \vec{p}||_2} \right|$$
$$I_{p2}(t) = \frac{(\Delta x)^2}{2\pi((x - x_d)^2 + y^2 + z^2)} \frac{(\vec{p_d} - \vec{p}) \cdot v_l}{||\vec{p_d} - \vec{p}||_2}$$

## 3 Triangles

## 4 Exact Strip-based Improvement

One of the most common file formats used to represent scenes is the STL file. Originally created to stand for "STereoLithography", it is more now more typically known to stand for "Standard Triangle Language" and "Standard Tesselation Language." This is because an STL file represents a 3D scene as a mesh of two-dimensional triangles, by specifying, for each triangle, the locations in three dimensions of each vertex in that triangle.

The typical method for estimating the intensity response of one such triangle is to assume that the triangle is small, implying that the response should be the delta function response that would result from a small patch at the center of the triangle. This is effective for small triangles, but for larger triangles, this approach won't work. The standard approach in this case is to subdivide the large triangle into many smaller triangles, each of which will have a delta-function response.

This isn't perfect either, though. Imagine a triangle whose width and height are $w$ and $h$, respectively. Then, the number of mini-triangles that you must subdivide the larger triangle into in order to get an accurate response is $O(\frac{wh}{(\Delta x)^2})$.

The key insight of the strip-based approach is that instead of subdividing a large triangle into smaller triangles, we can subdivide into small *strips.* If the triangle is taller than it is wide, we'll subdivide into vertical strips; if it's wider than it is tall, we'll subdivide into horizontal strips. The number of strips we need to subdivide the triangle into in order to get an accurate response is $O(\frac{\min(w,h)}{\Delta x})$. Because we can find the temporal response of a strip in constant time (using the closed form from the previous section), this is a speedup by a factor of $\frac{\max(w,h)}{\Delta x}$.

### 4.1 Constant-Time Approximation

We can do even better with an approximation. In *Locating a Target in Three Dimensions*, I describe an approximation for finding the intensity response of a plane, given arbitrary source and target points. The approximation rests on the assumption that all points along the contact ellipse have the same intensity density. This assumption is very accurate, as can be seen from the section from *Locating a Target* on computing $I_p(t)$.

Here, the way such an approximation would work is it would rest on the assumption that the intensity response of the triangle would closely follow

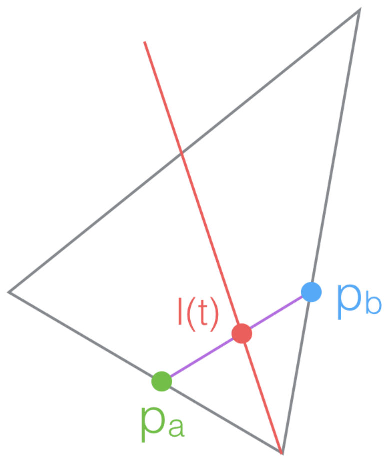

Figure 1: A visualization of the approximation described in this section. In red is the strip bisecting the triangle, whose intensity we are assuming is proportional to the intensity of the triangle. We multiply this by the line in purple that connects $p_a$ and $p_b$, the points of contact of the light on the each of the two edges of the triangle.

the intensity response of the strip bisecting the triangle, if you also multiplied by the width of the triangle at that point in time (which is easily computed). More concretely, let $I_s(t)$ be the intensity response of the strip as a function of time, which can be computed as explained in the previous section in constant time. Let $\vec{p_a}$ be the contact point of the light on one side of the triangle, and let $\vec{p_b}$ be the contact point of the light on the other side of the triangle. Then, $\hat{I}(t) = \frac{||\vec{p_a} - \vec{p_b}||_2}{\Delta x} I_s(t)$. Fig. 3 visualizes this approximation.

This approximation is sound, but weakened relative to the one described in *Locating a Target* by the fact that in fact the pattern of light hitting the triangle is not in fact a straight line, but curves slightly: it is an arc of an ellipse. Sadly, because computing the perimeter of an ellipse is hard, it is difficult to do better. This weakness becomes more pronounced for triangles that are closer to the source and detector.

Because of this, the approximation performs slighty less well than the approximation used in *Locating a Target*. Fig. **??** shows how well this approximation compares to a "ground truth" generated through very fine dis-

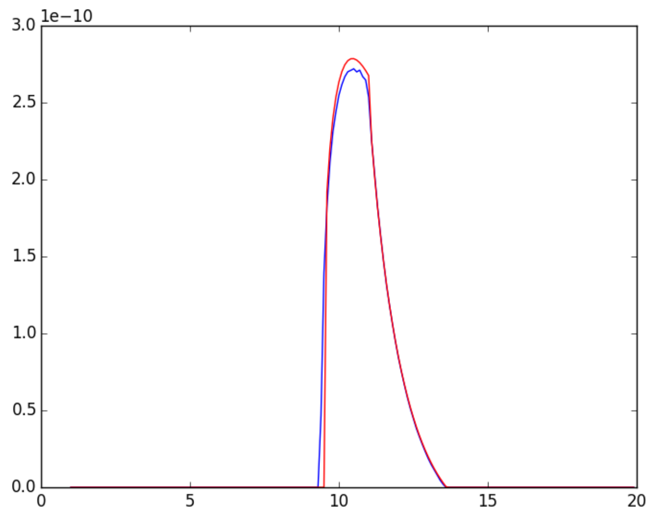

Figure 2: A comparison of the response generated by the approximation (red) to the "ground truth" calculated through discretization (blue). The approximation is imperfect, but still closely matches the truth. In this calculation, the source was at $(0, 0, 0)$, the detector was at $(0, 0, 1)$, and the triangle was defined by its three vertices at $(2, 1, 1)$, $(2, -1, 3)$, $(2, 1, 5)$.

cretization.

## 4.2 Discussion

How much of an improvement do these methods constitute over the standard method of assuming triangles are small, and subdividing them if they aren't? Well, naturally, they are a major improvement if the triangle-mesh representing the scene is made up of only a few large triangles. This is true in practice mainly for large, flat surfaces: a rectangular wall can be effectively represented with just two triangles, and even a structure like a flat circular table could be represented much more concisely with fewer large triangles than with many tiny triangles.

But what about curved surfaces, such as a sphere? Or worse yet, a human outline? In order to accurately represent the curves on the surfaces, you'll need to break it down into many small triangles *anyway.* So my methods for exactly computing the response of each triangle will yield only a tiny improvement—since every triangle's response was more or less a delta function regardless.

Here's the thing, though: I claim that this limitation is *not my fault*, but is a problem inherent in our choice of representation. When we demanded that we represent scenes as a mesh of triangles, there was no way we'd ever be able to concisely represent a sphere, evne though a sphere is an object that could in theory be described by very few degrees of freedom. If I knew that it was a sphere that I needed to render, I *could* in fact render it more efficiently using methods like these, by splitting the sphere up into circular strips. Each strip would be a one-dimensional manifold, and as such it would fulfill the requirement of intensity-per-path constancy. The exact same technique could be applied, just with a different set of equations to solve in order to find $k(t)$.

This is no real consolation in cases where we are handed STL files and asked to render them. What difference does it make if the true scene is a sphere, approximated by a mesh of tiny triangles? The problem of recovering the sphere from the less-concise mesh is terribly ill-posed. But if we go back to the original structure of our solution—that we wanted to have an efficient renderer, and then somehow invert the forward model to recover a scene from the movie we see experimentally—well, in many cases we can move beyond triangles and represent the sphere concisely. Consider the standard idea: generate a training set by creating many synthetic scenes, rendering them, and adding them to the training set by adding the (synthetic scene $\rightarrow$ movie) pairs to the training set. Then, use neural nets to generate a scene from a movie.

Well, if we know that our scenes should include examples with spheres in them, then we should make sure that we represent our spheres intelligently. That is, we want to represent them not as fine meshes of triangles, because then we won't be able to take any better advantage of their structure, but we want to represent them as spheres. This will give us concrete gains in terms of how quickly we can render them, as explained in this writeup.