# An Explanation of Algorithms for Fast Rendering

January 19, 2017

## 1 An Explanation of Algorithms for Fast Rendering

The following is an explanation of how to efficiently solve the problem of rendering: that is, given a description of a scene, and a location for a source and a detector, find the temporal response of the returning light as a function of time. Solving this problem is of interest to us because it is a subproblem in many plausible solutions to the problem of inferring a scene that is not within direct line of sight, based on the response of the light from that scene. Such solutions tend to take the form of a two-step process:

1. Generate a theory. This theory generally takes the form of some tentative arrangement for the scene.
2. If our current theory does not fit the response we see, generate a new theory.

Given the above template, it becomes apparent that knowing how to render scenes efficiently is equivalent to knowing how to test hypotheses efficiently. If the goal is to perform some kind of hypothesis search (as it is in many common solutions to this problem[1]) then testing hypotheses efficiently is critical to being able to make the search as exhaustive as possible, leading to the best possible solution.

### 1.1 Line- or Strip-based Computation

In a previous write-up[2], we described how it was possible to find a closed form for the intensity response over time of a point source reflecting off

[1]One prominent example of this approach is *Transient Imaging for Real-Time Tracking Behind a Corner*, by Hullin, Klein, and Laurenzis

of an infinite plane. Here, we 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."

## 1.2 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. The solution, being a polynomial (including radicals) of about 30 terms is unwieldy and is not reproduced here.

[2] Although reading this previous writeup is not necessary to understanding the calculations that follow, it is an example of applying the technique to solving a simpler problem. The reader may find it to be a helpful introduction, if the calculations here are confusing. The previous writeup can be found at `https://adamyedidia.files.wordpress.com/2014/11/spherepoint.pdf`

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.

$$I_p(t) = I_{p1}(t) \cdot I_{p2}(t)$$

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

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

## 1.3 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 $\frac{\max(w,h)}{\Delta x}$.

## 1.4 Constant-Time Approximation

We can do even better with an approximation. Such an approximation would rest on the assumption that the intensity response of the triangle would closely follow 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

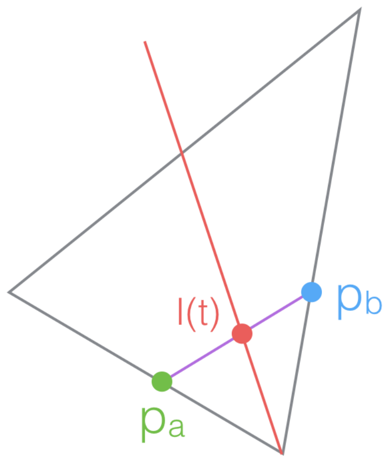

Figure 1: A visualization of the approximation described in this section. In red is the strip bisecting the triangle, whose intensity we are assuming is proportional to the intensity of the triangle. We multiply this by the line in purple that connects $p_a$ and $p_b$, the points of contact of the light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

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. 2 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 discretization.

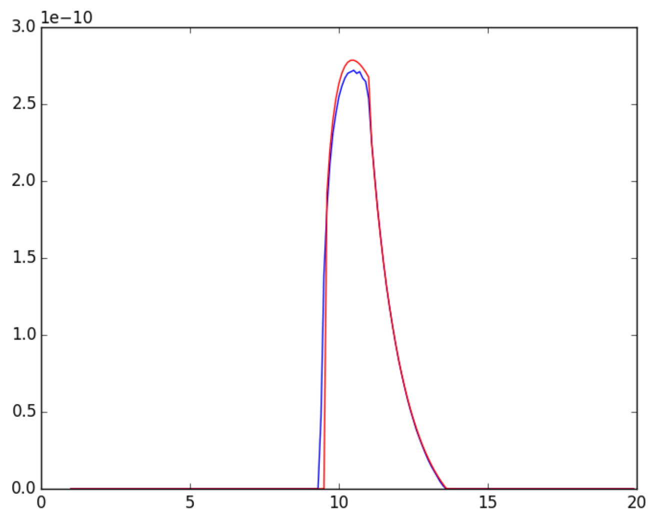

Figure 2: A comparison of the response generated by the approximation (red) to the “ground truth” calculated through discretization (blue). The approximation is imperfect, but still closely matches the truth. In this calculation, the source was at $(0, 0, 0)$, the detector was at $(0, 0, 1)$, and the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

### 1.5 Error Discussion

How much of an improvement do these methods represent over the standard method of assuming triangles are small, and subdividing them if they aren't? And how acceptable is the error introduced by the constant-time approximation? This depends heavily on how noisy the model we're working with is. There are a few sources of noise in the system—the first is the quantum nature of light, which leads to the fact that the intensity curve of returning light over time in fact represents the rate function of returning photons over time. When that rate is small, the error introduced by Poisson noise will be much greater than that introduced by the approximation.

Empirically, the approximation to the triangle light response introduces an error of no more than 5% in space or time over the course of the entire approximated curve. The variance of a Poisson distribution is equal the square root of its rate; a rate function of 400 photons/bin will lead to a Poisson error of about 5% across the whole curve—that is, equal to or greater than the error introduced by the constant-time approximation. If we want a smooth curve, we'll want around separate 10-100 time bins (consistent with a SPAD camera at the lower end of that range, and a streak camera at the higher end). That means about 40,000 returning photons.

If there are many more photons than that, the error introduced by the approximation will start to dominate, and it will become wiser to use the exact linear-time formula for computing the temporal response of each triangle. If there are many *fewer* photons than that, this whole elaborate algorithm will become superfluous—the original delta-function approximation to a triangle's response, perhaps with Gaussian smoothing, will be indistinguishable from the computations returned by this algorithm. Of course, returning photon count depends on a variety of factors, the most important of which are the surface reflectivities and the power of laser used.

| Returning Photon Count | Likely Triangle Response |
|---|---|
| 1-1,000 | Delta Function |
| 1,000-40,000 | Constant-Time Approximation |
| 40,000+ | Linear-Time Exact Computation |

The topic of neural nets and other trainable models merits special discussion. These models involve training a machine-learning model on a training set of synthetic scenes and their corresponding responses, building the training set using this augmented renderer. Once the training set is built, it can be used to train a convolutional neural net (or other such model) to take as input a temporal response and output a description of a scene that could have generated that response. This kind of approach represents an important special case of the search-based approach described initially.

This special case has the interesting property that, even if the final neural net is going to make predictions in a photon-sparse regime, it's possible that it will perform best if it is trained in a photon-rich (or perfect) regime. This question deserves further investigation—as does, in fact, the entire question of how the search should best be performed.

## 1.6 Modeling Discussion

Another issue for this approach is that it represents 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 as triangles, it will be necessary to break them down into many small triangles anyway. So these 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.

There's a subtle point here, though. When we assumed that we would represent curved surfaces using only triangles, we conceded high complexity to begin with. No algorithm can represent the response of an extremely complex mesh of triangles efficiently. The problem here is not the choice of algorithm, but is 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, even though a sphere is an object that could in theory be described by very few degrees of freedom. If we knew that it was a sphere that we needed to render, we *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 becomes important when we ourselves are generating the synthetic scenes. If we know that that the scenes of ultimate interest are going to be composed of objects from a known class, with relatively few degrees of freedom (such as spheres), we can generate our training set efficiently using this rendering technique, applied to a concise representation scheme for those objects.

In this sense, the true problem that we face at this point is a question of the best way to represent scenes. Maybe the real innovation that is possible here is a better way to represent scenes including curved objects. This could have important implications not only for this problem, but also for any problem that takes as input or output a scene representation. This includes the important and well-studied problem of, given an image of a scene, building an accurate 3D model of the scene.