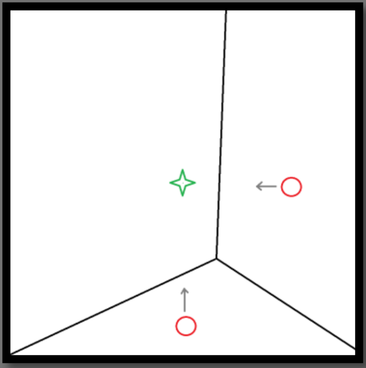
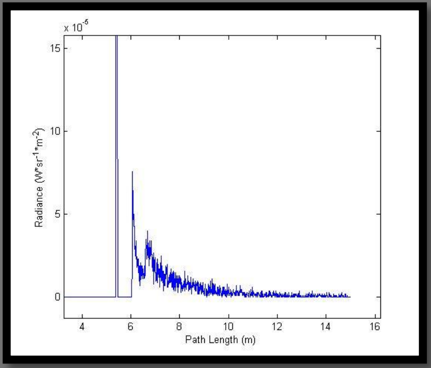
Time-of-Flight Tracer

Phil Pitts
Malcolm Slaney
Arrigo Benedetti
Phil Chou

3-Dimensional Scene



Time-of-Flight Histogram



Microsoft Research
11/8/2014

Table of Contents

1. Introduction

This document describes a ray tracing application that calculates backscatter information useful in understanding time-of-flight phenomena. This system is particularly useful in understanding the effects of multipath (or global illumination) in time-of-flight (ToF) camera depth calculations.

The Time-of-Flight Tracer (ToF Tracer) system is based on path tracing. In path tracing, rays are traced from their point of origin until they intersect with scene geometry. At these intersections, the ray directions are recalculated and the process repeats. The Time-of-Flight Tracer calculates the radiance and path length at each of these intersection points. In combination, these data points provide a **radiance** vs. **path length** histogram for the scene.

Users of the ToF Tracer specify a scene description and indicate rendering parameters. The ToF Tracer calculates the time (path length) between each illuminator and the camera. Because of multipath there will be more than one path between a source of illumination, the corner, and the camera. Thus the ToF Tracer returns its results as a histogram as a function of time (distance) for one or more pixels captured by the camera. The cover photos show the ToF histogram at a single pixel for a scene consisting of a single corner and a single point-source illuminator near the camera. Because the light reflects off multiple faces of the corner there is a range of reflections, after the first direct path.

In most cases it only makes sense to have a single illumination source and a single camera. This model reflects traditional time-of-flight camera hardware which consists of a single modulated or gated near infrared illumination source and a single camera. However, in some cases it may be desirable to model the effects of outside illumination (ex. sunlight) on a scene. In this case there are multiple illumination sources.

The full Time-of-Flight Tracer system consists of:

- Ray tracer
- Histogram plotting tool
- Automation script for batch data generation

This software is an extension of the open source Physically Based Ray Tracer (pbrt) program written by Matt Pharr and Greg Humphreys. This documentation makes several references to their book, *Physically Based Rendering*. This book is an excellent reference on physically based rendering, and was written by the authors of pbrt as an in-depth reference for their ray tracer.

This document describes how to install and build the ToF Tracer (3. Building), how to control the ray tracer (4. Usage), describes two example renderings (5. Examples), and finishes with some tips and tricks (6. Tips & Tricks).

2. Conventions & Acronyms

Term	**Definition**
pbrt	Physically Based Ray Tracer (name)
< >	Used to denote that this term should be replaced with its described value (i.e. <name> --> teapot)

3. Building

The full Time-of-Flight Tracer system is broken into several components:

- Ray tracing application (C++)
- Histogram plotting application (MATLAB)
- Batch data generation (Python)

This section of the report describes how to install and build each of these three components.

3.1 Ray Tracer (C++)

The Time-of-Flight Tracer ray tracing application is built using the Visual Studio 2013 solution file found under the $SRC_ROOT/src/vs2013 directory. The method for building the ray tracing application is the same as the method used to build pbrt. Additional building instructions can be found online at: http://www.pbrt.org/building.php. Compiling the ToF Tracer with other environments has not been tested.

3.2 Plotting Tool (MATLAB)

The histogram plotting MATLAB application can be compiled into a standalone executable using mcc (MATLAB Compiler). This is required for:

a) Running the full Python automation script
b) Generating histogram plots on workstations without MATLAB

The histogram plotting tool can be compiled by running the following command in the MATLAB command window:

```
>> cd path/to/Time-of-Flight_Tracer
>> cd matlab
>> mcc -mv tofplot_batch.m -a getparams.m
```

MATLAB should provide output similar to:

```
Compiler version: 4.18 (R2012b)
Processing D:\Programs\MATLAB\R2012b\toolbox\matlab\mcc.enc
Processing D:\Programs\MATLAB\R2012b\toolbox\shared\spcuilib\mcc.enc
Processing include files...
2 item(s) added.
Processing directories installed with MCR...
The file mccExcludedFiles.log contains a list of functions excluded from the CTF
archive.
0 item(s) added.
Generating MATLAB path for the compiled application...
Created 41 path items.
Begin validation of MEX files: Sun Oct 19 20:35:31 2014
End validation of MEX files: Sun Oct 19 20:35:31 2014
Parsing file "path/to/Time-of-Flight_Tracer\matlab\tofplot_batch.m"
       (Referenced from: "Compiler Command Line").
Parsing file "D:\Programs\MATLAB\R2012b\toolbox\compiler\deploy\deployprint.m"
       (Referenced from: "Compiler Command Line").
Parsing file "D:\Programs\MATLAB\R2012b\toolbox\compiler\deploy\printdlg.m"
       (Referenced from: "Compiler Command Line").
Deleting 0 temporary MEX authorization files.
Generating file "path/to/Time-of-Flight_Tracer\matlab\readme.txt".
```

3.3 Automated Data Generation Script (Python)

An (included) Python script automates the generation and subsequent plotting of time-of-flight backscatter histograms. On systems with Python installed, this script can be run directly assuming:

The MATLAB executable has been generated (see

- 3.2 Plotting Tool (MATLAB) above)
- All requirements listed in the script file have been met (see Template Input File Format)

In some cases there is a need to run the script on a system without Python (such as a high-performance computing cluster). In this case, it is useful to compile the Python script as a standalone executable. This may be accomplished using PyInstaller (http://www.pyinstaller.org).

Note: It is much simpler to install PyInstaller on top of a 32-bit Python installation regardless if the underlying operating system is 32-bit or 64-bit.

Instructions for installing and using PyInstaller can be found via the application website. Once PyInstaller is properly installed and configured, the batch data generation script can be compiled into an executable by executing the following commands:

```
$ cd path/to/Time-of-Flight_Tracer
$ cd python
$ pyinstaller -F autogenerate.py
```

A stand-alone executable should be generated at:

path/to/Time-of-Flight_Tracer/python/dist/autogenerate/autogenerate.exe

4. Usage

The following section documents utilizing the Time-of-Flight Tracer system. Portions of the system can be run manually for specialized jobs, or batch data can be produced from a template file by utilizing the Python automation script.

4.1 Ray Tracer (C++)

The ray tracing application represents the core of the system and is an extension of the open source Physically Based Ray Tracer (pbrt) written by Matt Pharr and Greg Humphreys. This section of the documentation describes usage of all extensions made to pbrt.

It is recommended that the user familiarize themselves with the input file format of pbrt. Components of pbrt that were not extended are not discussed in this documentation. However, these components are still important to understand. Documentation of the pbrt input file format can be found at: http://www.pbrt.org/fileformat.php

Input files specify the scene and control the pbrt and ToF Tracer programs. The ToF Tracer extends the control file to provide new capabilities unique for measuring time (or distance.) The remainder of this section will describe these new capabilities.

Renderers

Renderers are the center of the pbrt process flow. They are usually provided with a scene, sampler, camera, and several integrators. It is the job of the renderer to efficiently process the rendering workflow for a scene. This usually involves generating a set of tasks which are executed in parallel on different threads. These tasks usually request sampling parameters from the sampler component, cast rays into the scene, and request backscatter information from the integrator components. This backscatter information is then passed to the camera film where it is stored.

Renderers are the most loosely defined of the pbrt components. In truth, almost the entire ray tracing workflow could be encapsulated within a single renderer. However in order to take full advantage of the component-based architecture of pbrt, it is best to distribute the rendering workflow to sampler and integrator components as much as possible.

When analyzing time-of-flight systems, it is often useful to view both:

a) Radiance and path length at each ray intersection (see 1. Introduction)
b) The average radiance and path length at only the **first** ray intersection (i.e. ground truth)

The first of these data sets provides a backscatter histogram **with** multipath effects. The second of these data sets provides us with a backscatter histogram **without** multipath effects, or the ground truth. In most time-of-flight scenarios we do not know the ground truth, and are interested in it from data which includes multipath effects. It is therefore useful to have the ToF Tracer generate both data sets for later analysis.

The Time-of-Flight Tracer system supports two renderers which each generate one of the above data sets for a scene.

Name	**Implementation Class**
"sampler"	SamplerRenderer
"groundtruth"	GroundTruthRenderer

The SamplerRenderer provides a backscatter histogram **with** multipath effects. The GroundTruthRenderer provides a backscatter histogram **without** multipath effects.

These two renderers are very similar. In fact, the only difference is that the SamplerRenderer instructs camera films to discretize calculated radiances into multiple path length samples or bins. The GroundTruthRenderer instructs camera films to average calculated radiances into a single path length sample or bin.

In most cases, the GroundTruthRenderer should be used with the ToFDirectIntegrator (see Surface Integrators). The ToFDirectIntegrator uses direct lighting or lighting which is only calculated at the first ray-geometry intersection. This provides a good approximation of the ground truth for a scene. It is possible to use the GroundTruthRenderer with other integrators, but the results are not useful.

The SamplerRenderer should be used with all integrators except the ToFDirectIntegrator. The SamplerRenderer instructs camera films to save backscatter histograms into multiple discretized path length samples or bins rather than a single averaged path length sample or bin. These histograms are valuable for determining the effects of multipath on time-of-flight depth calculations.

For additional information regarding the SamplerRenderer see:
http://www.pbrt.org/fileformat.php#renderers

Films

Films are the medium which store the backscatter information calculated during the render. Two additional time-of-flight specific films have been added to the existing pbrt films. Note that there also exists an ImageFilm which is defined by the vanilla pbrt release. The ImageFilm is used for rendering regular graphical images of a scene. These graphical images can be useful quickly verifying camera and scene orientations. For more information see http://www.pbrt.org/fileformat.php#film.

The two additional time-of-flight specific films are defined as follows:

Name	**Implementation Class**
"histogram"	HistogramFilm
"signal"	SignalFilm

HistogramFilm

The HistogramFilm saves time-of-flight backscatter information into bins of user-defined width. The term 'bin' in this context refers to a range of path lengths for which an average radiance is computed.

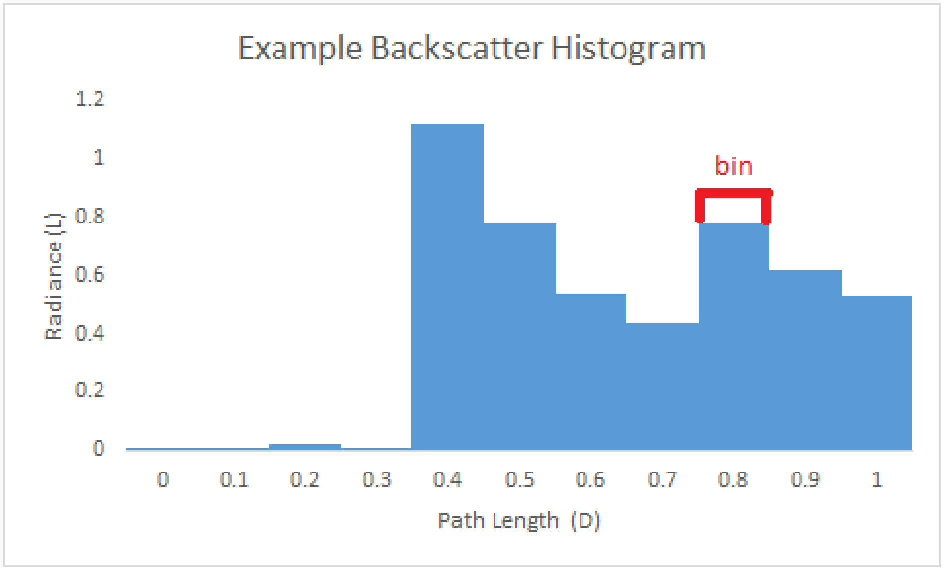


Figure 1: Example low-resolution histogram plot with marked bin

The following parameters are defined for the HistogramFilm:

Type	Name	Default	Units	Description
string	filename	output.txt	---	The output filename
integer	xresolution	640	pixels	The number of pixels in the x direction
integer	yresolution	480	pixels	The number of pixels in the y direction
float	minimumL	0	$W{\cdot}sr^{-1}{\cdot}m^{-2}$	The minimum bin radiance to output
float	range	1	meters (m)	The range (in meters) for which to store histograms
float	binsize	0.1	meters (m)	The width of each histogram bin

The first three parameters for the HistogramFilm act the same as those defined for the ImageFilm (see http://www.pbrt.org/fileformat.php#film).

The 'minimumL' parameter acts as a radiance threshold to reduce output file sizes. In most cases there is a small amount of noise located in uninteresting parts of the histogram. After rendering is completed, all histogram bins with a radiance below the 'minimumL' value are **not** written to the output file.

The 'range' parameter acts as a path length upper bound (in meters) for the histogram and helps to reduce output file sizes. In most cases histograms will consist of several peaks which slowly drop off as path length increases. The 'range' parameter trims histograms at a certain path length. If time-of-flight backscatter information is returned for a bin outside the user-defined range of the HistogramFilm, a warning is given.

The 'binsize' parameter controls the resolution of the time-of-flight backscatter histogram. It defines the width of each histogram bin in meters. Smaller 'binsize' values result in higher resolution histograms. However, higher resolution histograms also result in increased output file sizes. When the 'binsize' parameter is set to zero, raw samples are written to file without being averaged into a bin.

The HistogramFilm generates text files using a unique format. In this format, the '#' character separates individual pixel histograms. Following the '#' character are two integers which represent the coordinates of the current pixel. Following these values are pairs of floats representing a sample on the pixel's histogram. These pairs are composed of the distance followed by the radiance. Pixel coordinates are indexed with (0, 0) in the top left corner.

In summary, the HistogramFilm output file format is as follows:

```
# <x-coord> <y-coord> <distance1> <radiance1> <distance2> <radiance2>… # <x-coord>…
```

IMPORTANT:

When rendering using the HistogramFilm it is important to be aware of how much memory you are requesting the system to allocate. Currently, out-of-core processing is not supported for the HistogramFilm. This means that if you request the program to allocate more memory than is physically available on your system, the program will crash. This is especially important on 64-bit architectures as the operating system will attempt to allocate the additional memory using the system's swap space. This will result in a high cache miss rate and can effectively render the system inoperable. The reason for this behavior is due to an operating system design choice called overcommitment.

The requested memory (M_r) should always be less than the system's available memory (M_s). The requested memory is calculated as follows:

$$M_s > M_r = N_{pixels} * \frac{R}{S_{bin}} * 8 \quad (1)$$

In this equation, N_{pixels} represents the number of pixels for which histograms are being requested. R represents the range ('range' parameter) of the histogram, and S_{bin} represents the size of each histogram bin ('binsize' parameter). The constant term represents the memory size of each histogram sample (8 bytes). The resulting product (M_r) is in terms of bytes. The 'minimumL' parameter does not affect this calculation as it affects the final histogram output rather than histogram calculation during rendering.

SignalFilm

In certain cases it makes sense to perform operations on generated backscatter data during the rendering process. One example of this might be the convolution of the backscatter data with an integration kernel or other mathematical process. Processing the data in this way can reduce memory requirements and significantly increase the speed of post-processing tasks.

The SignalFilm does not serve a direct purpose in the Time-of-Flight Tracer, but was implemented as an example to show how operations can be performed on backscatter data before their final output.

The SignalFilm calculates the convolution of an integration kernel with the rendered time-of-flight backscatter histogram. This is useful for efficient processing of time-of-flight backscatter histograms into the phase offset and signal attenuation data needed for continuous wave intensity modulation (CWIM) depth camera calculations. An N x M matrix of N sampling phases by M sampling frequencies can be provided to the “signal” film. These sampling parameters simulate the sampling performed by physical time-of-flight cameras. A good reference for more information regarding time-of-flight depth calculation is “Technical Foundation and Calibration Methods for Time-of-Flight Cameras” [Lefloch et al. 2013].

The signal film supports the following parameters:

Type	Name	Default	Units	Description
string	filename	output.txt	---	The output filename
integer	xresolution	640	pixels	The number of pixels in the x direction
integer	yresolution	480	pixels	The number of pixels in the y direction
float[]	frequencies	null	hertz (Hz)	List of camera sampling frequencies
float[]	phases	null	radians	List of camera sampling phases

The first three parameters for the SignalFilm act the same as those defined for the ImageFilm (see http://www.pbrt.org/fileformat.php#film).

The last two parameters act as described above. Both parameters are float arrays of variable length. In pbrt, arrays are defined as a list of values separated by whitespace and enclosed in square brackets (‘[]’).

Example:

```
“float phases” [0 2.2128 4.8234]
```

The SignalFilm also generates text files using a unique format. The SignalFilm must generate values for each pixel. It is also true that each pixel has N x M values where N represents the number of sampling frequencies and M represents the number of sampling phases. Therefore the number of values written is known and there is no need for a delimiting character between pixels. The signal film instead writes output in raster order traversing along rows first starting from (0,0) and ending at ('xresolution', 'yresolution'). For each pixel, the signal film writes the N x M frequency by phase matrix. This matrix is written using the same raster order convention used for pixels.

For example, the output for a single pixel can be represented by:

```
(freq1, phase1) (freq1, phase2)… (freq2, phase1) (freq2, phase2)… (freqN, phaseM)
```

Surface Integrators

In pbrt, surface integrators determine backscatter information for surface reflections given a set of sampling parameters and camera rays. Examples of surface integrators implemented by pbrt include integrators for direct illumination, photon mapping, path tracing, and more. The Time-of-Flight Tracer system extends several of these surface integrators to provide time-of-flight specific backscatter information. These surface integrators are defined as follows:

Name	Implementation Class
“tofdirect”	ToFDirectIntegrator
“tofpath”	ToFPathIntegrator
“tofbipath”	ToFBipathIntegrator

These time-of-flight surface integrators simply extend the corresponding regular surface integrators. Each surface integrator samples the radiance and path length of their rays at each intersection with the scene geometry.

The ToFDirectIntegrator generates time-of-flight backscatter information using direct lighting. This means that only the first intersection of rays with the scene geometry is sampled. The ToFPathIntegrator provides time-of-flight backscatter information using path tracing to calculate the effects of multipath. The ToFBipathIntegrator uses bidirectional path tracing to calculate the effects of multipath. Both the ToFPathIntegrator and ToFBipathIntegrator sample each intersection of their rays with the scene geometry.

Note that a bidirectional surface integrator is not implemented in vanilla pbrt, but was implemented in the Time-of-Flight Tracer system to act as a base class for the ToFBipathIntegrator. An in-depth tutorial on bidirectional path tracing can be found in *Physically Based Rendering* by Matt Pharr and Greg Humphreys on page 770.

All new time-of-flight surface integrators support the following parameters:

Type	Name	Default	Units	Description
float	lambdamax	1000	nanometers (nm)	The maximum wavelength of light to consider
float	lambdamin	400	nanometers (nm)	The minimum wavelength of light to consider

The parameters 'lambdamax' and 'lambdamin' define the range of light wavelengths to consider. In most images this is simply the range of visible light. However, most time-of-flight cameras emit light at in the near infrared spectrum. Usage of these parameters allows the user to define a custom wavelength range of interest.

Finally the ToFPathIntegrator and ToFBipathIntegrator both also define the following parameter:

Type	Name	Default	Units	Description
integer	maxdepth	5	---	Maximum number of ray bounces

The 'maxdepth' parameter allows users to control the maximum number of intersections rays can make with the scene geometry before being terminated. The ray termination is controlled by the Russian roulette technique. This means that a ray may be randomly terminated even when it has intersected fewer times than the 'maxdepth”' parameter and can still intersect with geometry. This allows for an efficient and unbaised sampling approach. For more information on Russian roulette see *Physically Based Rendering* by Matt Pharr and Greg Humphreys page 680.

Samplers

Samplers provide parameters to integrator components during rendering. Samplers also control the number of rays fired into a scene during rendering time. While tracing more rays usually results in a less noisy image, the benefit usually drops exponentially as more rays are traced. There are various methods for sampling images. Efficient sampling can greatly reduce rendering times and lead to less noisy images.

The Time-of-Flight Tracer system mainly utilizes the samplers implemented in vanilla pbrt. Descriptions of these samplers can be found at http://www.pbrt.org/fileformat.php#samplers. One additional sampler has also been implemented:

Name	**Implementation Class**
“subrange”	SubrangeSampler

The SubrangeSampler allows the user to define a set of pixels to be sampled. It is useful for applications where a sparse sampling of the image should be completed rather than an exhaustive sampling. The SubrangeSampler takes as arguments the following parameters:

Type	**Name**	**Default**	**Units**	**Description**
integer	pixelsamples	4	---	Number of rays to trace per pixel
integer[]	xcoordinates	null	---	x-coordinates of pixels to sample
integer[]	ycoordinates	null	---	y-coordinates of pixels to sample

The SubrangeSampler uses Monte Carlo to sample the set of pixels defined by the 'xcoordinates' and 'ycoordinates' parameters. The lengths of these two arrays should be equal, and all pixels defined by the arrays must be within the bounds of the image resolution.

4.2 Batch Data Generation

In the Time-of-Flight Tracer system, batch data generation can be achieved using a Python script. This script takes as input a template file, a folder containing the scene geometry, a working directory, and a batch name. It automatically generates rendered scene images and histogram graphs. When each component is compiled into executables this script can be executed easily on a high-performance computing cluster. This allows for batch execution of different scenes as separate automated jobs.

Input

The batch data generation script takes two parameters as command line arguments:

Type	**Name**	**Description**
string	name	General name describing the batch of histogram plots being generated
string	root	Working directory for time-of-flight batch processes

The ‘name’ parameter acts as a unique identifier for the batch process being executed. If this parameter is non-unique, the previous batch data using the identifier will be overwritten. The ‘root’ parameter indicates the working directory for the batch process.

The script executable (autogenerate.exe) should be launched as follows using either bidirectional path tracing or path tracing:

Bidirectional path tracing:

```
backscatter_simulator.exe –n <name> -r <root>
```

Path tracing:

```
backscatter_simulator.exe –n <name> -r <root> -p
```

The batch data generation script assumes the following executables exist prior to execution:

Name	**Description**
<root>/bin/pbrt.exe	C++ ray tracer application
<root>/bin/exrtotiff.exe	Conversion tool included in C++ ray tracer application
<root>/bin/tofplot.exe	Standalone MATLAB histogram plot generation script

The batch data generation script assumes a template input (.pbrt) file to be located at:

```
<root>/<name>.pbrt
```

Any additional geometry included by the template input file should be located in the following directory:

```
<root>/geometry/
```

Folder Hierarch

The Time-of-Flight Tracer system assumes a specific folder hierarchy. This folder hierarchy begins at the user specified ‘root’ directory and automatically generated batch data are generated into subfolders according to the user specified ‘name’ variable. For example:

- Assume ‘root’ = “C:/path/to/batch/directory”
- Assume ‘name’ = “example_name”
- Data for this job will be generated in: “C:/path/to/batch/directory/example_name/”

For each batch, a folder (<root>/<name>) is generated. For each processing subprocess, a corresponding subfolder hierarchy is created within the <root>/<name>/ directory. Within each of these subfolder hierarchies, folders are created to contain the scenes, ray tracer output, and post-processed data. The template input file is moved within the <root>/<name> directory and renamed to <name>_template.pbrt.

Post-processed output can be found in the following locations:

- <root>/<name>/images/images/
- <root>/<name>/histograms/images/figures/
- <root>/<name>/histograms/images/jpeg/

Template Input File Format

The template input file is the core of the batch data generation process. It defines a series of camera and rendering parameters which fully control the batch data generation process. It is based off the pbrt input file format (see http://www.pbrt.org/fileformat.php) but includes additional parameters, which are controlled by the batch data generation script.

Batch data generation script parameters must begin with a '#' character in order to differentiate themselves from pbrt parameters. The pbrt system uses '#' to being a comment. Therefore the pbrt system does not directly see the batch data generation parameters. The batch data generation system uses '//' to begin a comment.

The script parameters defined by the template input file are as follows (in order):

Type	**Units**	**Description**
integer	---	Number of outputs to generate per subprocess (n)
integer[2]	---	Starting and ending value to linearly vary along n times
float[2]	nanometers (nm)	Range of light wavelengths to capture
integer[]	---	x-coordinates to generate histograms for
integer[]	---	y-coordinates to generate histograms for

The Time-of-Flight Tracer batch data generation script supports a number of system controlled variables which can be used within the template file. These variables will be replaced by their correct values during script execution. All script variables are enclosed within <> to differentiate themselves from pbrt parameters.

The following batch data generation script variables are supported:

Name	**Description**
<lambdamax>	Maximum light wavelength captured by camera
<lambdamin>	Minimum light wavelength captured by camera
<xsub>	List of x-coordinates for pixels for which histograms will be generated
<ysub>	List of y-coordinates for pixels for which histograms will be generated
<file>	Filename (varies)
<var>	Variable which varies according to the user defined starting and ending points.

The final parameter, '<var>', is central to the usefulness of the script. This parameter is used to vary the term of interest within the described scene. It is controlled by the number of outputs and the line defined by the user defined starting and ending point parameters. For example, if a user wanted to generate 11 sets of images with increasing glossy reflectance they could would:

- Set the number of outputs (n) to 11
- Set the start/end points to 0 1 (this represents a gradient starting at 0 and continuing to 1)

Then for each of the 11 sets of images, the <var> variable would evaluate to 0.0, 0.1, 0.2, 0.3, 0.4… 1.0 as the index of the current set increased.

4.3 Post-Processing (MATLAB)

A post-processing tool to generate histogram plots has been built into the Time-of-Flight Tracer system. These graphs provide a visual representation of the time-of-flight backscatter histogram at a single pixel.

The graph shown in Figure 2 demonstrates a histogram generated for a pixel on the wall of a corner scene. The figure shows three peaks. The first peak is the largest and represents the direct backscatter contribution. The smaller second and third bounces represent the corresponding second and third bounces or intersections of the ray with the scene.

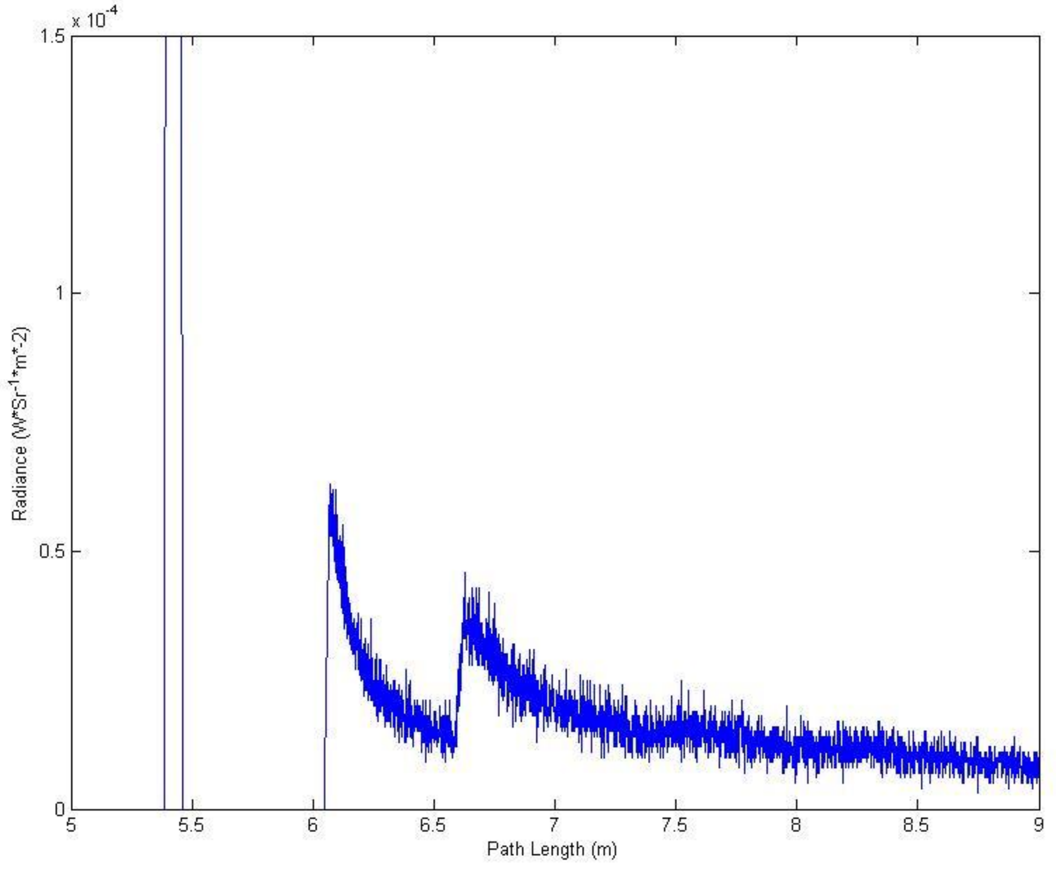


Figure 2: Example Histogram Plot

Individual histogram plots can be generated using the tofplot.m MATLAB script. The tofplot.m script takes the following parameters (in order):

Type	Name	Description
string	input	ray tracer output file to plot
integer	pixelX	x-coordinate of pixel to plot
integer	pixelY	y-coordinate of pixel to plot
string	output	Location to save plots

The output parameter describes the location to which both a JPEG and FIG file should be saved. This parameter should **not** have a file extension, but instead only a file name.

Example: path/to/output/histogram_file

Multiple histogram plots can be generated at one time using the tofplot_batch.m MATLAB script. The tofplot_batch.m script takes the following parameters (in order):

Type	Name	Description
string	name	General name describing the batch of histogram plots being generated

string	root	Working directory for time-of-flight batch processes

Both of these parameters are identical to the parameters passed to the batch data generation script (see Template Input File Format). The tofplot_batch.m script assumes the following directories exist:

- <root>/<name>/histograms/output/
- <root>/<name>/histograms/images/jpeg/
- <root>/<name>/histograms/images/figures/

These folders are automatically generated when using the batch data generation python script. The tofplot_batch.m script also takes arguments from a template file:

- <root>/<name>/histograms/<name>_template.pbrt

This template file acts as an input to the Time-of-Flight Tracer batch data generation script and is defined by the user. Details for defining the input template file can be found above (see Template Input File Format).

The histogram plot script will automatically generate graphs for every pixel requested in the input template file. For each defined pixel, the histogram plot will generate N graphs where N is the number of discrete steps defined in the input template file. Each graph requires a corresponding output file from the ray tracing application. These output files should be generated using the HistogramFilm. They are assumed to be named '<name>_<n>' where n ranges from 0 to N-1. These output files should be located in:

- <root>/<name>/histograms/output/

The script will generate histogram plots in both JPEG and MATLAB figure (.fig) formats. These sets of plots will be generated in the following directories (respectively):

- <root>/<name>/histograms/images/jpeg/
- <root>/<name>/histograms/images/figures/

5. Examples

The following describes the workflow for rendering an sample scene. This section describes two workflows. The first workflow illustrates the process of individually rendering and plotting a scene. The second workflow shows the process of rendering and plotting a batch of scenes using a template file.

These examples assume that each of the Time-of-Flight Tracer components has been built. Two example scene files (corner.pbrt, corner_batch.pbrt) are also included in the Time-of-Flight Tracer distribution. The following assumes these files exist:

- $WORKING/bin/exrtotiff.exe
- $WORKING/bin/pbrt.exe
- $WORKING/bin/tofplot_batch.exe
- $WORKING/bin/autogenerate.exe

- $WORKING/matlab/tofplot.m
- $WORKING/corner.pbrt
- $WORKING/corner_batch.pbrt

Where $WORKING is defined by some working directory.

5.1 Individual Data Generation

This example workflow demonstrates how to generate time-of-flight backscatter data using the ray tracer application and how to plot that data using the included MATLAB script.

On the command line, execute the following commands:

```
cd $WORKING
bin\pbrt.exe corner.pbrt
```

Output similar to the following should be displayed:

```
pbrt version 2.0.0 of Oct 19 2014 at 15:12:37 [Detected 8 core(s)]
Copyright (c)1998-2012 Matt Pharr and Greg Humphreys.
The source code to pbrt (but *not* the book contents) is covered by the BSD Lice
nse.
See the file LICENSE.txt for the conditions of the license.
corner.pbrt(1): Warning: This version of pbrt fixes a bug in the LookAt
    transformation. If your rendered images unexpectedly change, add a "Scale
    -1 1 1" to the start of your scene file.
Rendering: [++++++++++++++++++++++++++++++++++++++++++++]  (2.3s)
```

A file corner.txt should now exist at $WORKING/corner.txt. Next we will plot the generated data using the MATLAB script.

On the command line, execute the following commands:

```
cd matlab
matlab /r "tofplot('../corner.txt', 50, 50, '../corner')"
```

These commands launch MATLAB and generate a histogram plot from the ray tracer time-of-flight backscatter data. Additionally two files (corner.fig, corner.jpg) should be generated in the $WORKING directory.

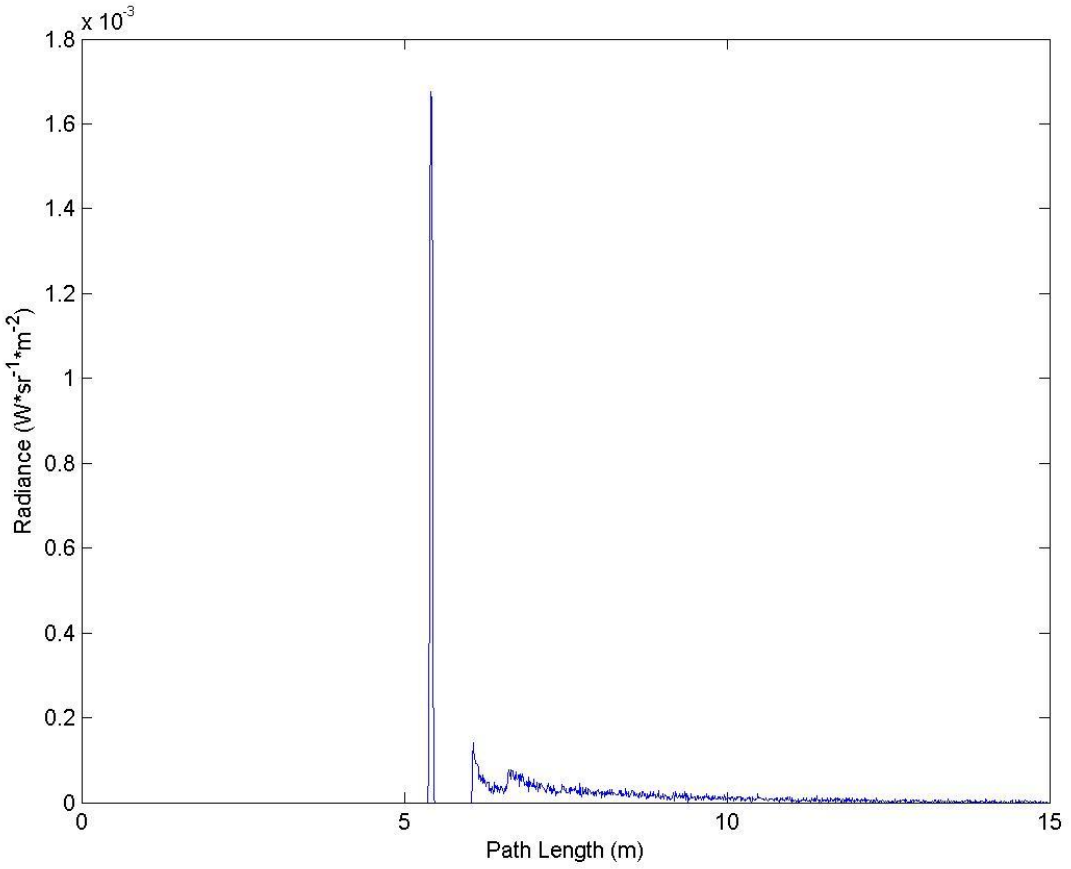


Figure 3: Generated Corner Example JPEG

5.2 Batch Data Generation

This example workflow will demonstrate how to generate batch time-of-flight backscatter data using the included python script. This workflow assumes that the python script has been compiled into an executable as discussed in the Building section (see 3.3 Automated Data Generation Script (Python)).

To execute the workflow using path tracing (vs. bidirectional path tracing), run the following from the command line:

```
cd $WORKING
bin\autogenerate.exe -r $WORKING -n corner_batch -p
```

After processing has completed a batch of data will be generated in the $WORKING/corner_batch folder. This data includes normal pictorial renders as well as histogram textual output and their corresponding plots.

Specifically the output includes:

Output Type	**Location**
EXR image files	$WORKING/corner_batch/images/output
TIFF image files	$WORKING/corner_batch/images/images

Histogram textual output	$WORKING/corner_batch/histograms/output
Histogram JPEG plots	$WORKING/corner_batch/histograms/jpeg
Histogram FIG plots	$WORKING/corner_batch/histograms/figures

6. Tips & Tricks

The following section describes some lessons learned that may prove useful to users of the Time-of-Flight Tracer system.

6.1 Path Tracing vs. Bidirectional Path Tracing

In most documentation you will read that bidirectional path tracing works better than path tracing. This is a true statement, most of the time. However, it is important to understand how both techniques function as there are certain cases where bidirectional path tracing produces much noisier images than path tracing.

The most significant difference between the two techniques is that path tracing follows a path from the camera to the light source while bidirectional path tracing also uses a separate path from the light source to the camera.

Let's pause to think of how each of these paths are generated.

Camera Path Generation

The path from the camera to the light source is generated by firing a ray from the camera into the scene. If this ray intersects with scene geometry we use a probability distribution function to determine the new direction of the ray after intersection. This ray continues to bounce around the scene until reaching the light source (unlikely) or being terminated.

Light Path Generation

The path from the light source to the camera is generated by firing a ray from the light source into the scene. If this ray intersects with scene geometry we again can determine a new direction and continue tracing until the ray intersects with the camera (unlikely) or is terminated.

Camera Ray Generation vs. Light Ray Generation

These two path generation methods almost sound identical, but it is key to realize that **light sources do not necessarily generate rays in the same way cameras do**. For example, think of a point light source. This light source illuminates in all directions. This means that rays pointing in all directions will be traced from the light source. On the other hand, a camera points in some direction. This means that rays pointing in single direction will be traced from the camera.

Consider a mostly empty scene. In most cases, the camera will point at the defined geometry (as blank space is rather boring). This means that rays traced from the camera have a good chance of intersecting with the scene geometry. However, rays traced from a point light source are generated in all directions. This means that **rays shot from the point light source have a very small chance of actually intersecting with the scene geometry**.

The resulting effect is that most paths traced from a point light source are empty. However, when a non-empty path is found from the point-light source (a ray that intersects the geometry is sampled), it makes a contribution to the final radiance of the pixel being rendered. Since in most cases there is not a non-empty path found for some rendered pixel this can lead to high variance when using paths from the light to the camera (i.e. bidirectional path tracing).

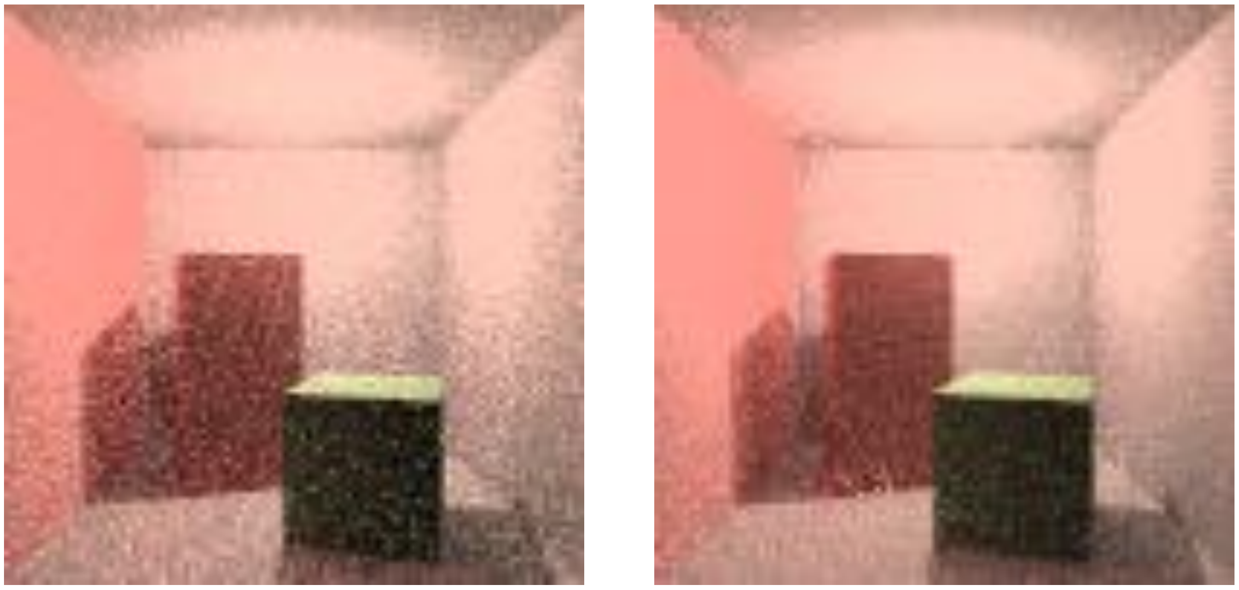


Figure 4: Cornell box rendered with path tracing (left) and bidirectional path tracing (right)

Consider the images in Figure 4 (above). Both images were rendered in similar time. However, the left image was generated using path tracing and contains a higher level of noise than the right image which was generated using bidirectional path tracing.

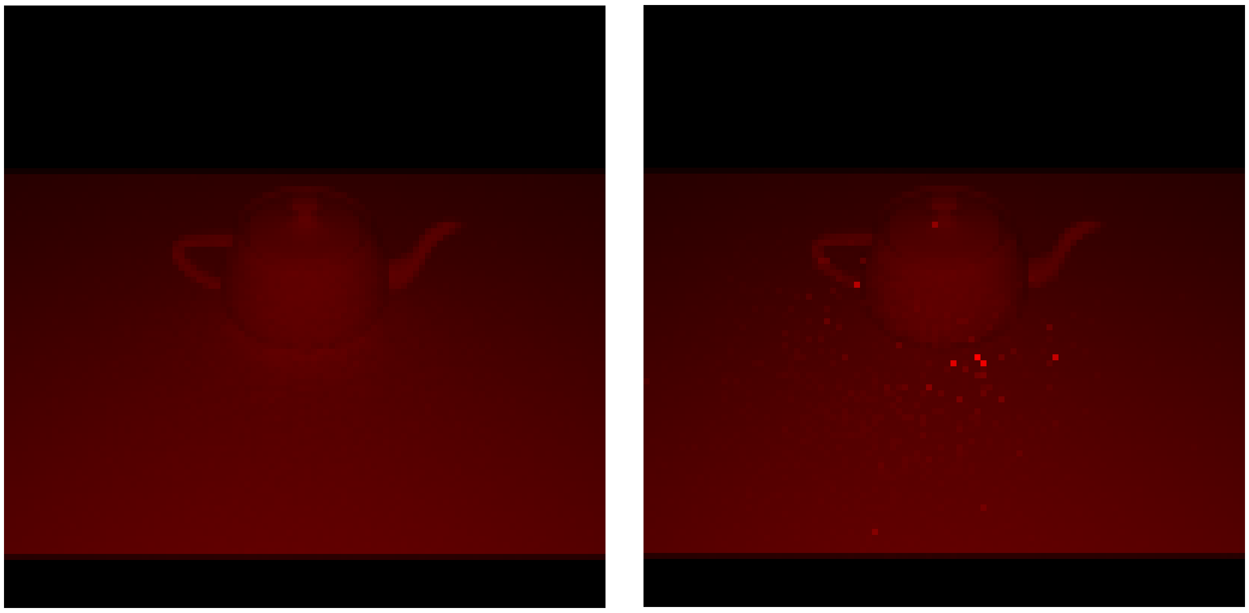


Figure 5: Semi-empty scene rendered with path tracing (left) and bidirectional path tracing (right)

Next consider the semi-empty scene rendered in Figure 5 (above). In this scene bidirectional path tracing (right) contains a much higher level of noise than the image rendered using path tracing (left).

These two sets of images demonstrate that bidirectional path tracing is more effective in fully enclosed scenes (such as the Cornell box), but is less effective in semi-empty scenes (such as the teapot scene).