CHROMA RAYTRACING SCRIPT FOR VUV SETUP: USER MANUAL

Date Last Compiled: September 6, 2019

Contact: Ben Page
Email: benjaminpage.acer@gmail.com
Affiliation: Swansea University

Contents

1 Overview **1**

2 Installation **1**
2.1 Accessing the Singularity Container 1
2.2 Downloading the Scripts 2

3 Operation **2**
3.1 Simulating Photons 2
3.2 Understanding the Output 3

4 Analysis **3**
4.1 History Flags 3
4.2 Plotting 4

A Requirements **6**

1 Overview

This document serves as the user manual for the current version of the VUV setup raytracer. The following contains details on how to install and run the simulation - for information regarding the specifics of each script, please consult the *Technical Manual*.

2 Installation

The scripts use *Chroma*[1], a raytracing package developed for Python 2.7, and operates out of a *Singularity*[2] container, which itself includes all the necessary packages to run the simulation.

2.1 Accessing the Singularity Container

The singularity container comes as either CUDA 8 or CUDA 10 compatible (see Appendix A: Requirements) - the CUDA 8 container has ROOT installed, whereas the CUDA 10 container does not. The scripts are written in such a way that they bypass the need for ROOT - if you intend on editing the code to allow ROOT compatibility, you will need to either use the CUDA 8 install or (also if your video card does not support CUDA 8) you will need to contact the container host (Ako Jamil: ako.jamil@yale.edu) and kindly ask if he can provide[3] you with a container with ROOT installed.
To pull a copy of the container, use the command

```
singularity pull --name chroma.img <insert url>
```

where the available urls are:

(CUDA 10)	`shub://akojamil/nexo-chroma-container:noroot-cuda10`
(CUDA 8)	`shub://akojamil/nexo-chroma-container:wroot`

Once you have pulled the container, you can access it using the command

```
singularity shell --nv chroma.img
```

in the directory local to the container.

[1] https://chroma.bitbucket.io/
[2] https://singularity.lbl.gov/
[3] I would not recommend installing Chroma yourself, as Python 2.7 is soon to be deprecated and much of the required packages are not easily installable anymore - proceed with caution.

2.2 Downloading the Scripts

All the necessary scripts to run the program have been uploaded to the TRIUMF VUV Setup Google Drive, along with the documentation. If you do not have permission to access this drive, you can alternatively download the scripts from github: https://github.com/jadot-bp/triumf-vuv-sim (*Note: You will still need to pull a copy of the container.*)

3 Operation

3.1 Simulating Photons

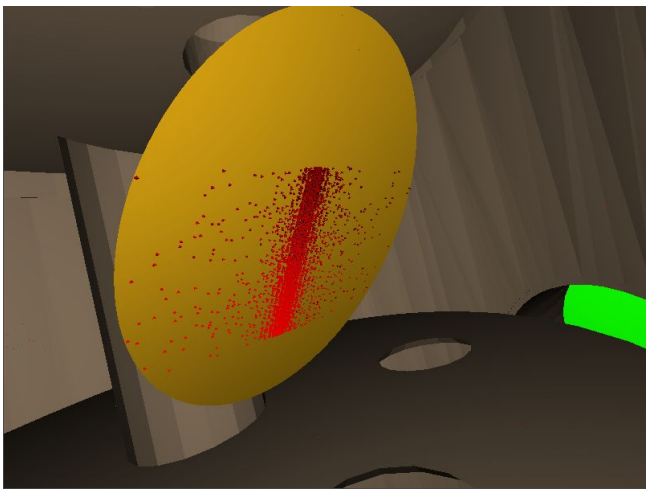

Figure 1: Screenshot showing terminated photons (red) on the mirror. These photons have been absorbed by the mirror.

The script `main.py` serves as the front end for the simulator. Run this script using Python 2.7 and it will prompt you to enter the necessary information to begin the simulation This includes the number of photons, the wavelength of light (in nm), the width of the post-monochromator slit (in µm) and the width of the beam (called `beamsize`, in µm). Alternatively, you can bypass the user input by passing a `.txt` file as a command line argument, whose first line contains a string of the following form:

```
<Number of photons>:<Wavelength>:<Slit width>:<Beam size>
```

For example, for a selection of 100,000 160nm photons with a slit width of 50µm and a beam width of 30µm, the first line of the `.txt` passed to the script should read:

`100000:160:50:30`

> *Note: Only the first line of the file is read by the script, so you can use the following lines to store comments about those values if necessary.*

The slit width can be no greater than 1000µm. The beam width here is also more a measure of the 'height' of the source (the slit width acts as the horizontal constraint). The housing for the slit has a circular bore of diameter 0.5in, so the maximal beam width is 12000µm.

The simulation will then pass the values onto `simulator.py` and perform the raytracing. The terminal may throw instances of `AssertionError: fatal exception`, but these should not affect the simulation results. Using the DAQ17 PC with 8GB of system RAM and an RTX 2060 a simulation of 1 million photons with a pre-rendered geometry takes about 20 seconds.[4]

3.2 Understanding the Output

Once the simulation has completed, it will output the results to the `results/` directory and will have a filename of the form

`<Date>_<Time>_<Number of photons>:<Wavelength>:<Slit width>:<Beam size>.dat`

The file itself is a table of the form:

	Initial (in)			Final (in)				
ID	x	y	z	x	y	z	λ (nm)	Flag

Where *ID* represents the photon ID, *Initial* and *Final* represent the initial and final positions of the photons in inches[5] and the wavelength is self-explainable. The final column is the photon flags - these correspond to the 'history' of the photon, i.e. what processes the photon has undergone during the simulation.

4 Analysis

4.1 History Flags

The history of each photon is encoded into a binary number, composed of single bits called flags. The flags are as follows:

[4] If you are with ALPHA/PHAAR and wish to see if you can use this computer, contact Lars Martin (lmartin@triumf.ca) – the GPU itself is owned by Art Olin (olin@triumf.ca).

[5] This is because the STL files provided by Resonance had inches as the default unit. Changing the units to metric would require converting the fine-adjustment in the code by hand. The individual `.stl` files would also each need to be converted to metric.

```
NO_HIT           = 0x1 << 0
BULK_ABSORB      = 0x1 << 1
SURFACE_DETECT   = 0x1 << 2
SURFACE_ABSORB   = 0x1 << 3
RAYLEIGH_SCATTER = 0x1 << 4
REFLECT_DIFFUSE  = 0x1 << 5
REFLECT_SPECULAR = 0x1 << 6
SURFACE_REEMIT   = 0x1 << 7
SURFACE_TRANSMIT = 0x1 << 8
BULK_REEMIT      = 0x1 << 9
NAN_ABORT        = 0x1 << 31
```

The flags are added each time the photon undergoes a new process. For example, if a photon undergoes specular reflection and then is absorbed by a surface, the `REFLECT_SPECULAR` and `SURFACE_ABSORB` flags are added:

```
(0x1 << 6) + (0x1 << 3) = 1000000 + 0001000 = 1001000 = 72
```

Therefore, a photon with this history would have flag 72.

4.2 Plotting

The results of the simulation can be seen using the `analysis.py` script. This script requires a boolean command-line argument:

- `0` - Render photon hits for visualisation
- `1` - View plots of the photon hits at the SiPM location

> *Note: you can use the* `--help` *flag for more information.*

The analysis script will prompt the user to select a file to load from `results/`. The available files are displayed in an easy-to-read form. Once a file has been selected, it will load the data and count how many photons have been detected by both the PMT and the SiPM, and returns a relative detection rate (based on the initial number of photons). If the script is run in rendering mode, it will create a rendering of the setup geometry along with markers which indicate the photon start (blue) and end (red) positions.

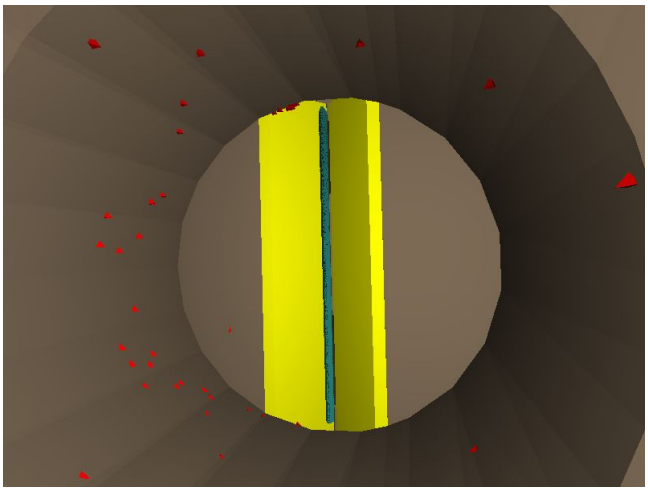

Figure 2: Screenshot showing the light source (blue) and slit housing (yellow). Some terminated photons (red) can also be seen.

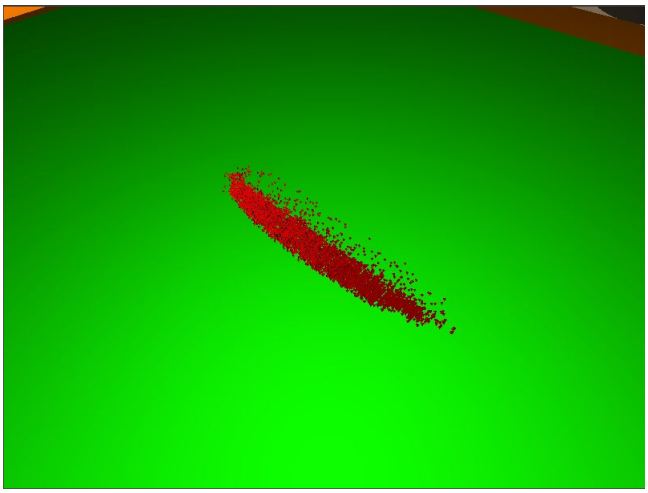

Figure 3: Screenshot showing the beam shape at the SiPM after a photon run, as displayed by `analysis.py` in 'visualization mode'.

Alternatively, if the script is run in 'plotting mode' then it will generate a centered scatter plot of the photon hits at the SiPM, along with a heat map representation (2D histogram) of these hits.

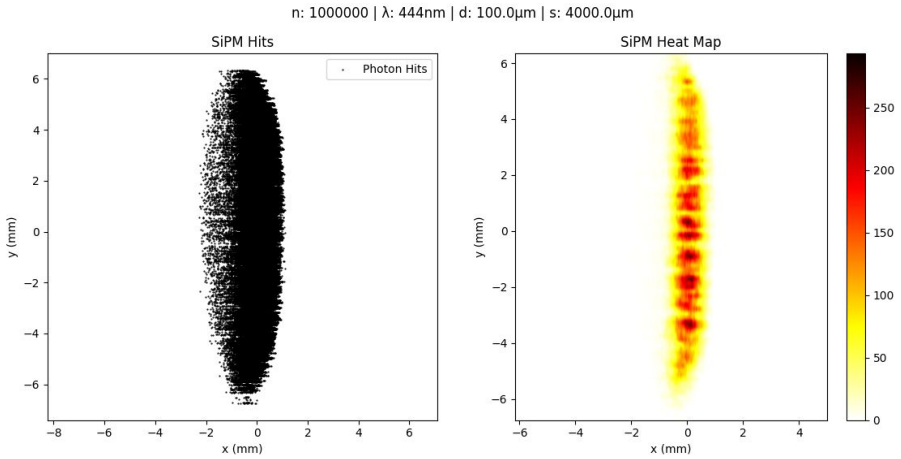

Figure 4: Screenshot of the scatterplot and heat map (with Gaussian interpolation) for a photon run. The color bar displays the relative intensity (in counts) at the SiPM.

A Requirements

The hardware requirements as specified by Chroma is listed below:

> At a minimum, Chroma requires:
>
> - An x86 or x86-64 CPU.
> - A NVIDIA GPU that supports CUDA compute capability 2.0 or later.
>
> We highly recommend that you run Chroma with:
>
> - An x86-64 CPU with at least four cores.
> - 8 GB or more of system RAM.
> - An NVIDIA GPU that supports CUDA compute capability 2.0 or later, and has at least 1 GB of device memory.
>
> Memory requirements on the CPU and GPU scale with the complexity of your model. A detector represented with 60.1 million triangles (corresponding to 20,000 detailed photomultipler tubes) requires 2.2 GB of CUDA device memory, and more than 6 GB of host memory during detector construction.

Chroma uses NVIDIA's CUDA to accelerate computations using the GPU. Since the CUDA installation in the container cannot be easily manipulated, you can check whether or not your GPU is compatible with the CUDA version in the container by comparing the compute-compatibility support of the CUDA SDK with the compute-compatibility of your GPU[6]. For example, the CUDA 8 SDK only supports compute compatibility 2.0-6.x, so Turing architecture GPUs would not be compatible with the container.

[6] You can check this at https://en.wikipedia.org/wiki/CUDA#GPUs_supported.