Implementation of YMC3D: 3D Monte Carlo photon transport in tissue inclusions

Asgeir Bjorgan, Matija Milanic, Lise Lyngsnes Randeberg

May 11, 2015

1 Introduction

This document is meant to detail the implementation of YMC3D, a package for calculating photon transport in smaller 3D tissue inclusions by employing NVIDIA CUDA technology. The package is presented and tested against comparable Monte Carlo packages in Bjorgan et al. [2].

2 Usage

YMC3D is run using

./YMC3D [geometry file] [tissue file] [output base filename].

Output is currently limited to **[output]_drs.bin**, which contains the spatially resolved diffuse reflectance. There are functions available for writing absorption, fluence and beam information.

2.1 Output

Outputs are currently in binary file formats. Format specification in the given order:

- 1 ints (specifies number of dimensions D, either 2 or 3)
- D ints (specifies number of voxels along coordinate direction, N_x, N_y, N_z)
- D floats (specifies voxel sizes dx, dy, dz)
- $N_x \cdot N_y \cdot N_z$ floats (specifies output data array, interleaved as **z*N_x*N_y + y*N_x + x**)

Output can be read using the MATLAB/octave function **matlab/read/read_ymc3d_outputfile.m**.

2.2 Input

Inputs are tissue data (μ_a, μ_s, g, n) and tissue geometry.

Tissue optical property format specification:

- 1 int (specifies the number of different tissue types T)
- $4{\cdot}T$ floats (specifies tissue optical properties for each tissue type. Saved as [$\mu_{a,1}$ $\mu_{s,1}$ g_1 n_1, $\mu_{a,2}$ $\mu_{s,2}$ g_2 n_2, ...].

Geometry format specifications:

- 1 int (specifies number of dimensions. This will always be 3, but included for compatibility to output file format.)

- 3 ints (specifies number of voxels along each coordinate direction, N_x, N_y, N_z)
- 3 floats (specifies voxel sizes dx, dy, dx)
- $N_x \cdot N_y \cdot N_z$ ints (specifies tissue voxel geometry. Interleaved as **z*N_x*N_y + y*N_x + x**.)

Tissue types are automatically incremented by 1 internally in YMC3D, and 0 is assigned to air. If input tissue type is set to 255, this is also interpreted as air.

Note that the input/output file format might change in the future. Input/output was implemented in a similar way to an older 3D Monte Carlo implementation [3] due to ease of inter-compatibility and comparison. On one hand, binary formats eases input and output, but becomes completely indechifferable if input/output functions ever change. This is also not neccessarily platform independent.

Examples of generation of tissue properties and geometry specifications are located in **matlab/geometry/**.

3 Implementation

3.1 Files

List of source files:

- **mc3d_gpu.cu/.h**: Photon tracking on GPU, photon reinitialization on GPU, kernel control from CPU.
- **mc3d_io.cu/.h**: Input/output functions for reading of geometry/tissue properties and writing of results.
- **mc3d_main.cu**: main() function, starting point. Specifies parameters, starts simulation, writes results to file.
- **mc3d_photons.cu/.h**: Photon structure. Contains detector functions.
- **mc3d_types.h**: Geometry and optical property structures. Specifies general Monte Carlo parameters (threshold for W, survival chance for Russian roulette, ...)

3.2 Overview

The implementation is based on the basic MCML photon tracking loop presented in [4], but with voxels instead of layer boundaries. The basic formulas are used (i.e. scattering directions, transmission, reflection, Russian roulette), albeit modified for consideration of boundaries also in x and y directions. The random number generator used by Alerstam et al. [1] was found to be suitable.

The implementation is structured around GPU- and CPU-suitable work and concurrent transfer of data between GPU and host. An overview of the data flow between GPU and the host is shown in figure 1. Photons are initialized and tracked on the GPU and transferred to the CPU for reflectance recording.

Photons are reinitialized in a separate function from the photon tracker. This eases implementation of various photon sources without interfering too much with e.g. the available amount of registry, GPU coalescing or SIMD parallelization in the main photon tracking loop. Detector functions are implemented on the CPU, where atomic access to double precision arrays is possible without introducing thread divergence on the GPU.

Photons are tracked for a specified number of steps before they are checked for possible reinitialization. The simulation time can primarily be tweaked by two parameters: The number of photon steps and the size of the photon array. Too many photon steps will lead to tracking of photons which have already been frozen. Too few will lead to too many useless reinitializations and memory copies back to the host. The number of photons controls the intermediary measure between too much data transfer between host and GPU and optimal GPU occupancy.

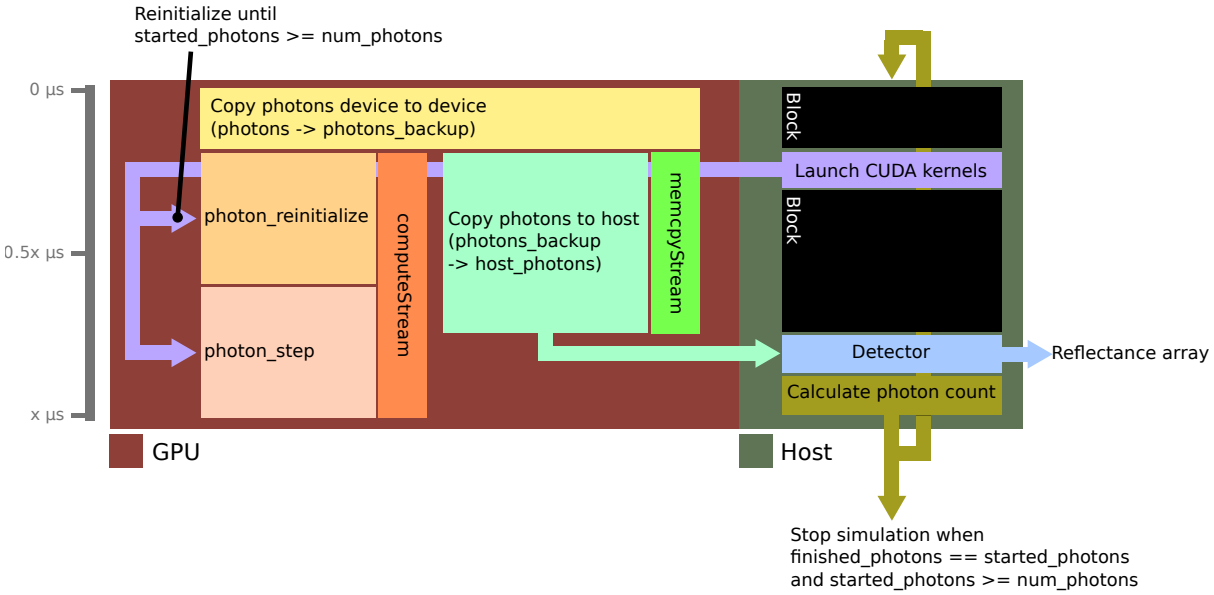


Figure 1: CUDA concurrency setup. The device properties are copied to a temporary device array (**photon_backup**). The photon properties are copied down to the host concurrently with reinitialization of dead or escaped photons and continued tracking of photons. Escaped host photons are then recorded in reflectance or transmission arrays.

3.3 Helper functions

List of helper device functions:

- **__device__ int getGridCoord()**: Get voxel grid coordinate from floating point coordinate, according to a fixed rule.
- **__device__ float intersection()**: Find next voxel intersection according to direction cosines and current coordinates, return distance. Move coordinates explicitly to the boundary.
- **__device__ int getTissueType()**: Get tissue type corresponding to voxel coordinates.
- **__device__ bool mirror()**: Modify coordinates according to mirror boundary conditions when outside the volume of interest.

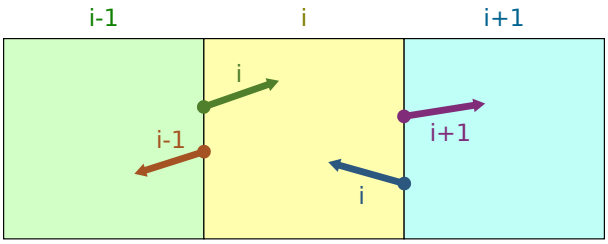


Figure 2: Calculation of grid coordinates. Photons with a positive direction cosine on a boundary are counted into the next cell.

Grid coordinates are calculated from the photon coordinates using the **floor()** operation, thereby always using the lower grid line as the current grid coordinate along that axis. Coordinates corresponding exactly to the grid line are treated according to the direction cosine as shown in figure 2.

The intersection algorithm is based on the behavior of the grid coordinate system. A direction cosine less than zero will always mean that the next potential boundary to cross with a non-zero distance would be the grid line corresponding to the grid coordinate. A direction cosine larger than or equal to

zero will potentially cross the next grid line (see figure 2). The intersection with a boundary is found through a step-wise procedure: the photon is intersected with all potential grid lines. The smallest intersection parameter is then chosen as the distance to the boundary. The coordinates which have the primary intersection with a grid line are explicitly moved to the boundary in order to avoid numerical inaccuracies through multiplication operations. The intersection algorithm is shown in figure 3.

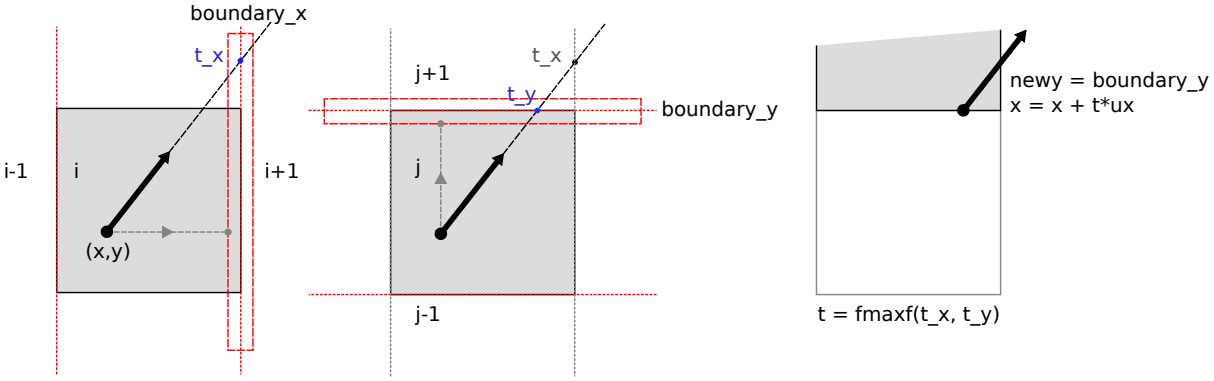


Figure 3: Algorithm for calculating the intersection between a photon and the grid.

The algorithm always chooses the smallest distance. It is therefore important to avoid numerical inaccuracies which may lead to the distance to a boundary being zero. Direction cosines less than a threshold are set exactly to zero in order to avoid e.g. [-0, -0, 1]. This could otherwise lead to ambiguous grid coordinates resulting in zero distance to the boundary. Direction cosines are renormalized to 1, though all operations on direction cosines are theoretically guaranteed to still sum to 1.

Tissue voxels are assigned a number corresponding to the type of tissue. The tissue geometry is contained in a 3D texture reference in order to improve on the memory access patterns (**getTissueType()**). Some coherency in photon locations can take advantage of the spatial caching provided by texture memory. Escape and extended layers boundary conditions are automatically implemented through use of texture memory. This avoids the need to introduce separate code for tracking photons outside the boundary. Mirror boundary conditions can be implemented explicitly on the coordinates at the end of each for loop iteration in **photon_step()** using **mirror()**. Boundary conditions are shown in figure 4.

Optical properties are looked up from a shared memory **float** array. Sparse randomized global access patterns are thus constrained to the tissue geometry lookup table.

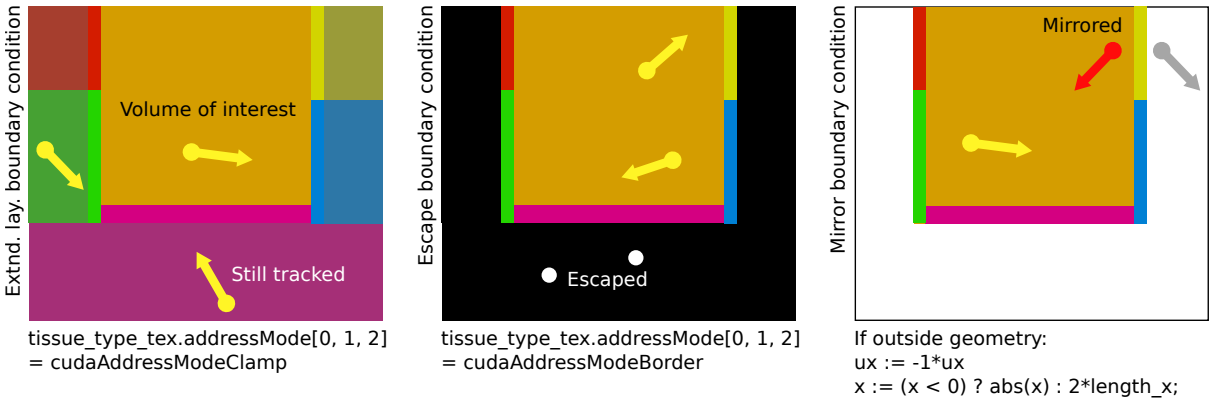


Figure 4: The different boundary conditions and implementations.

3.4 Photon tracking

List of global functions:

- **__global__ void photon_reinitialize()**: Reinitialize dead/escaped photons according to photon source.

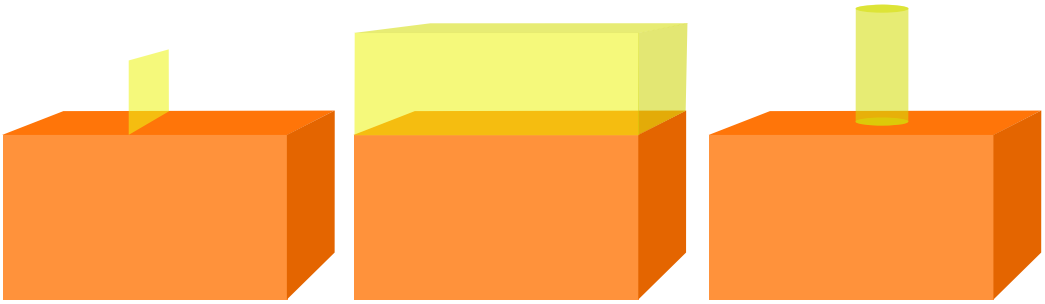
- **__global__ void photon_step()**: Track photons according to Monte Carlo rules for a specified number of steps.

List of host functions:

- **void detector()**: Record photon distribution according to detector rules.

The functions **photon_reinitialize()** and **photon_step()** provide the framework for the main photon tracking.

The basic interaction between the two was shown in figure 1. All photons are tracked for a specific number of steps before reinitialization to encourage some thread convergence.



```
x = x0;
y = rand()*length_y;
z = 0;
ux = 0;
uy = 0;
uz = 1;
```

```
x = rand()*length_x;
y = rand()*length_y;
z = 0;
ux = 0;
uy = 0;
uz = 1;
```

```
sincospif(&sintheta, &costheta, rand());
r = rand()*r0;
x = r*costheta + x0;
y = r*sintheta + y0;
z = 0;
ux = 0;
uy = 0;
uz = 1;
```

Figure 5: Implementation of various simple light sources.

The function **photon_reinitialize()** reinitializes dead or escaped photons once every while. This is illustrated in figure 6. Before reinitialization, photon properties are copied to a device array. Concurrently with reinitialization and photon tracking, the photon properties are copied to the host and recorded using a **detector()** function. Light sources are implemented directly in **photon_reinitialize()**. Some simpler examples are provided in figure 5. Separating reinitialization in a global function simplifies implementation of more advanced light sources as some recurrent thread divergence is avoided in the main tracking loop. The number of initialized and finished photons are tracked by an array indexed by the photons. Each thread increments its own count. These are copied to the host and summed to yield current photon counts. The host controls the launch of **photon_reinitialize()** according to whether the desired photon count has been reached, and stops the simulation when all photons are finished. Photons are implemented as arrays of fixed size. The arrays are never recreated, and dead or escaped photons scattered inbetween live photons are reinitialized in place throughout the array.

The **photon_step()** function implements the tracking of live photons using the usual Monte Carlo steps [4]. It is illustrated in figure 7.

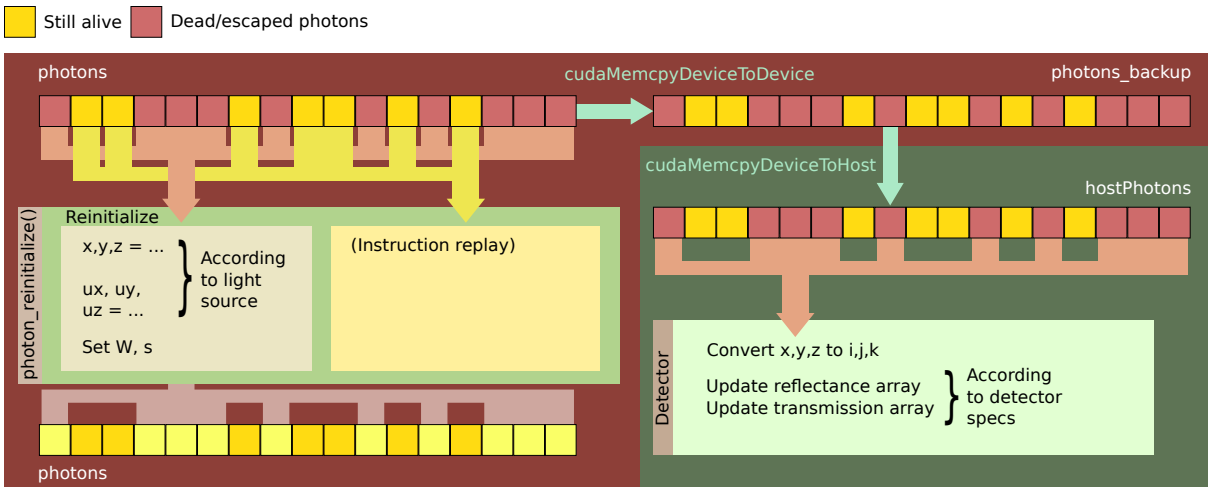


Figure 6: Overview over the **photon_reinitialize()** function and CUDA setup for avoiding thread stalling due to atomic operations on reflectance arrays. Photons are copied to a temporary device array before reinitialization, and copied to the host where the CPU records reflectances concurrently with GPU tracking of new photons.

4 Output properties

4.1 Reflectance

The number of photons can vary from the specified number of input photons, but it is always guaranteed that this canonical number of photons are all tracked to the finishing line. The total number of photons is output to a text file **[output]_numphotons.dat**.

Reflectance is calibrated by dividing all pixels by the total number of tracked photons. To get the reflectance as it would be obtained by a real detector, multiply by the number of pixels.

References

[1] E. Alerstam, W. C. Y. Lo, T. D. Han, J. Rose, S. Andersson-Engels, and L. Lilge. Next-generation acceleration and code optimization for light transport in turbid media using gpus. *Biomed. Opt. Express*, 1(2):658–675, 2010.

[2] A. Bjorgan, M. Milanic, and L. L. Randeberg. Ymc3d: Gpu-accelerated 3d monte carlo photon tracking in tissue inclusions. *In submission*.

[3] M. Milanic and B. Majaron. Three-dimensional monte carlo model of pulsed-laser treatment of cutaneous vascular lesions. *J. Biomed. Opt.*, 16(12):128002–1 – 128002–12, 2011.

[4] L. Wang, S. L. Jacques, and L. Zheng. Mcml monte carlo modeling of light transport in multi-layered tissues. *Comput. Meth. Prog. Bio.*, 47(2):131 – 146, 1995.

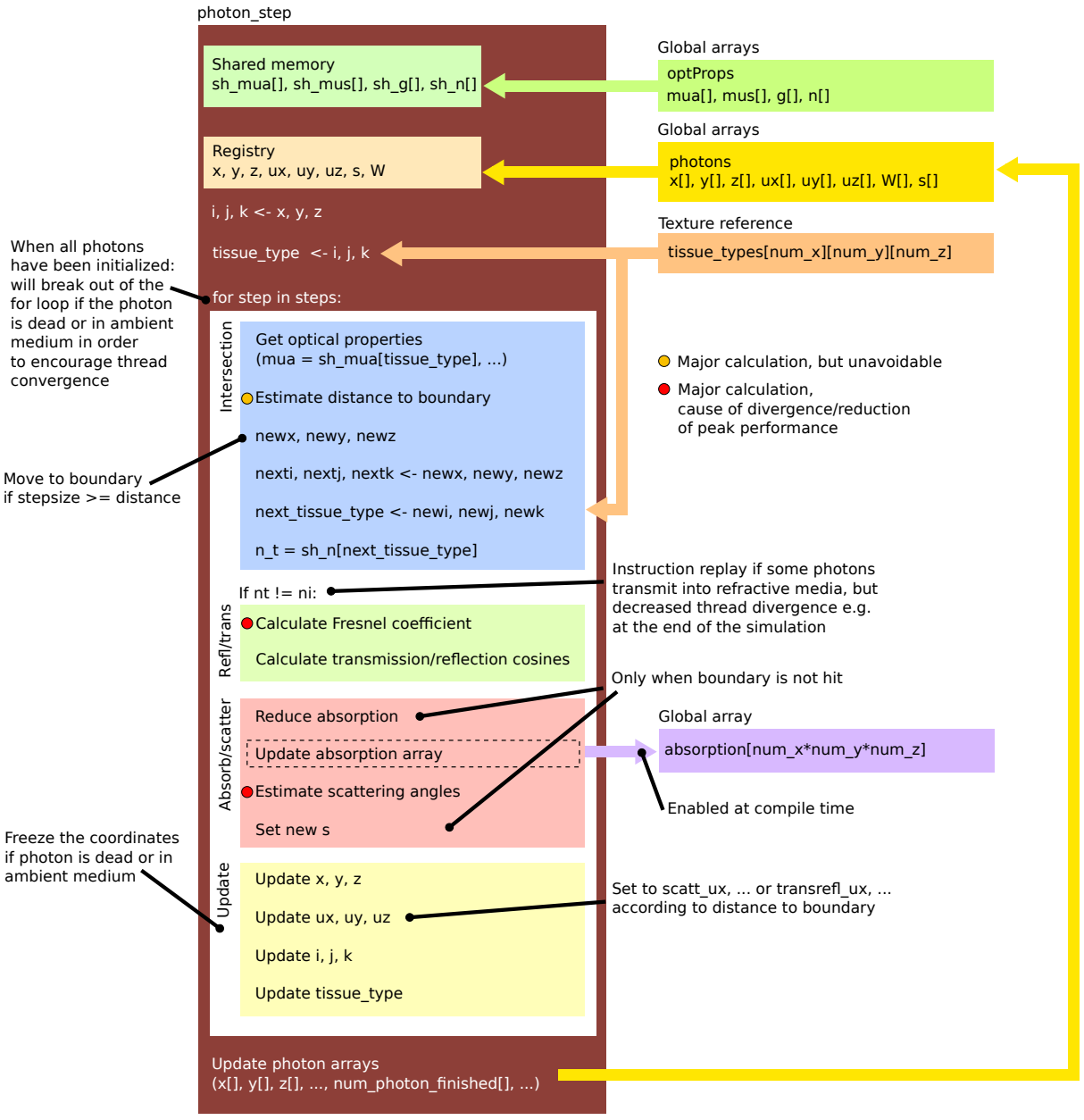


Figure 7: Overview over the photon_step function for tracking photons. All operations are run on all photons as instruction replay is for the most part unavoidable, and this reduces some thread divergence. Different operations are done primarily by setting the primary variables (**ux**, **uy**, ...) to the appropriate calculated variable (**scatt_ux**, **transrefl_ux**, ...) through ternary conditionals.