GGEMS

Release 1.3

GGEMS Developers

Oct 03, 2025

PREAMBLE

GGEMS (GPU Geant4-based Monte Carlo Simulations) is a Monte Carlo numerical simulation platform dedicated to medical applications such as CT/CBCT imaging and hadron therapy. The physical models from the Geant4 platform, which is a toolkit for simulating the interaction of particles through matter, are implemented in GGEMS using the OpenCL library. This approach enables parallel computing on heterogeneous architectures such as CPUs and GPUs. The core of the GGEMS library is fully written in C++, including the components that integrate OpenCL and OpenGL (for visualisation). However, the library can only be used through commands written in Python.

This documentation is mainly for users running CT/CBCT imaging simulations or photon dosimetry. It also provides guidance for those who want to understand how GGEMS works so they can integrate their own code.

This documentation is divided into three parts:

First, an introduction to GGEMS and the informations are given in order to install your GGEMS environment.

Second, informations about all GGEMS potentials are given. Examples and tools are also illustrated and explained. Command lines are listed using python instructions.

And finally, a more detailed description concerning the GGEMS implementation for advanced user. The goal is to provide sufficient information on the implementation of GGEMS so that any user who wishes can integrate their own code.

CHAPTER

ONE

INTRODUCTION

GGEMS is an advanced Monte Carlo simulation platform using the OpenCL library managing CPU and GPU architecture. GGEMS is fully developed in C++ and accessible via Python command line.

Well-validated Geant4 physic models are used in GGEMS and implemented using OpenCL.

The aim of GGEMS is to provide a fast simulation platform for imaging application (CT/CBCT for moment) and particle therapy. To favor speed of computation, GGEMS is not a very generic platform as Geant4 or GATE. For very realistic simulation with lot of information results, Geant4 and GATE are still recommended.

GGEMS features:

- Photon particle tracking
- Multithreaded CPU
- GPU
- Multi devices (GPUs+CPU) approach
- Single or double float precision for dosimetry application
- External X-ray source
- Voxelized source
- Navigation in simple box volume, voxelized volume or meshed volume
- Flat or curved detector for CBCT/CT application
- Visualisation using OpenGL

GGEMS medical applications:

- CT/CBCT imaging (standard, dual-energy)
- External radiotherapy (IMRT and VMAT)
- Portal imaging from LINAC system

In the next GGEMS releases, the aim is to implement the following applications and features:

- Positron particle tracking
- Electron particle tracking
- PET imaging
- SPECT imaging
- Intra-operative radiotherapy (brachytherapy and intrabeam)

CHAPTER

TWO

REQUIREMENTS

GGEMS is a multi architecture and multi platform application using OpenCL.

2.1 Operating Systems

- Linux (Ubuntu 24.04 LTS, other system should work)
- Windows 10 & 11

Warning

GGEMS has not been validated on macOS. Only minor modifications to the source code may be required.

2.2 C++ Compiler Version

- Linux: GNU v13.3 and Clang v18.1.3
- Windows:
  - Visual C++ v19.44.35217 for x64 from Visual Studio Community 2022
  - Visual C++ v19.50.35503 for x64 from Visual Studio Community 2026

2.3 Python Version

GGEMS supports the following versions

- Python 3.6+

2.4 OpenCL Version

GGEMS validated using the following version

- OpenCL 3.0

Important

GGEMS version 1.3 is only compatible with OpenCL 3.0. Ensure that this version is installed on your system. OpenCL 3.0 is compatible with NVIDIA and Intel.

2.5 Hardware

GGEMS can be used with lots of different hardwares such as CPU, GPU and Intel HD Graphics

- Intel (CPU + HD Graphics)
- NVIDIA

Warning

GGEMS has not been validated with AMD hardwares (CPU and GPU). Do not hesitate to contact the GGEMS developers if you have AMD hardware and encounter installation or operational issues with GGEMS.

CHAPTER

THREE

BUILDING & INSTALLING

3.1 Prerequisites

GGEMS is based on the OpenCL library. For each platform (NVIDIA, Intel and AMD), you must install the corresponding drivers.

Warning

GGEMS has not been validated with AMD. Useful AMD drivers could be found here.

3.1.1 NVIDIA

To use GGEMS on an NVIDIA platform, simply install CUDA (version 12.6 is recommended) along with the corresponding driver.

Important

The CUDA library is not used by GGEMS. Only the OpenCL library provided with CUDA is used. It's recommended to install CUDA from NVIDIA website. For linux user, installing cuda using the 'apt' program for instance is not recommended. If the NVIDIA driver does not match the correct CUDA version, GGEMS may not work.

3.1.2 INTEL

To use GGEMS on an Intel platform (CPU or GPU) you must install the driver. For this, it is recommended to install the driver provided by Intel oneAPI Base Toolkit. The library will be installed with the Intel compiler and the other libraries.

3.1.3 OpenGL visualization

Since GGEMS v1.2, the OpenGL library can be used to visualize a simulation in 3D space. OpenGL can be used on any OS. 3 libraries have to be installed to use OpenGL correctly:

- GLFW : an Open Source and multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop. It provides a simple API for creating windows, contexts and surfaces, receiving input and events.
- GLEW : a cross-platform open-source C/C++ extension loading library.
- GLM : a C++ header for mathematics library based on the OpenGL Shading Language (GLSL) specifications.

Important

For linux users, GLEW library must be installed from source (glew-XXX.zip). It's mandatory to link GGEMS and libGLEW.a static library.

Important

For linux users, GLFW and GLM libraries can be installed using the 'apt' program for example.

Warning

For Windows users the libraries should be downloaded from their respective website and installed if possible in the standard location C:\Program Files (x86)

3.2 GGEMS Installation

To install GGEMS, CMake and setuptools (a python project combined with CMake) are required.

In the following section, a recommended installation procedure is provided as an example. First clone the GGEMS project:

```
$ git clone https://github.com/GGEMS/ggems.git
```

Enter the directory ggems and launch the installation command (set opengl to OFF to deactivate it):

```
$ cd ggems
$ python setup.py build_ext --opengl=ON install --user
```

Note

On Windows OS, multi-processor compilation is possible using Ninja, and running the command:

```
$ python setup.py build_ext --generator=Ninja --opengl=ON install --user
```

GGEMS is now installed on your machine. To check the installation, you can try the examples or launch GGEMS in a Python console.

```
from ggems import *
opencl_manager = GGEMSOpenCLManager()
opencl_manager.print_infos()
exit()
```

CHAPTER

FOUR

DOCUMENTATION

This documentation section is dedicated to users. All parts of GGEMS macro are described in details.

4.1 Multi-Devices

GGEMS can be used on any platform compatible with OpenCL 3.0. Each GPU or CPU is recognised as a device. There are different ways to activate a device.

- Using the device index (assumed to be known by the user):

```
from ggems import *

opencl_manager = GGEMSOpenCLManager()
opencl_manager.set_device_index(0) # Activate device id 0
opencl_manager.set_device_index(2) # Activate device id 2, if it exists

exit()
```

- Explicitly indicate the device(s):

```
from ggems import *

opencl_manager = GGEMSOpenCLManager()
opencl_manager.set_device_to_activate('gpu', 'nvidia') # Activate all NVIDIA GPU only
opencl_manager.set_device_to_activate('gpu', 'intel') # Activate all Intel GPU only
opencl_manager.set_device_to_activate('cpu') # Activate cpu only
opencl_manager.set_device_to_activate('all') # Activate all found devices
opencl_manager.set_device_to_activate('0;2') # Activate devices 0 and 2

exit()
```

4.2 OpenGL Visualization

In GGEMS, user-defined volumes and sources can be visualised. OpenGL can be enabled through the GGEMS macro. Several colours are already predefined, such as:

- black
- blue
- lime
- cyan

- red
- magenta
- yellow
- white
- gray
- silver
- maroon
- olive
- green
- purple
- teal
- navy

Important

OpenGL settings must be initialized at the beginning of the simulation. All volumes and all sources must be declared only after this initialization.

```
from ggems import *

opengl_manager = GGEMSOpenGLManager()

# Multisample anti-aliasing, can be 1, 2, 4 or 8
opengl_manager.set_msaa(8)
# Background color for OpenGL window
opengl_manager.set_background_color('black')
# Draw axis X, Y and Z
opengl_manager.set_draw_axis(True)
# Output folder storing OpenGL images you want to save (*.png format)
opengl_manager.set_image_output('axis')
# Number of maximum particles drawn in the OpenGL viewport (max: 65536)
opengl_manager.set_displayed_particles(128)
# Change the color of photons, either using a color defined in GGEMS or using RGB indices
opengl_manager.set_particle_color('gamma', 152, 251, 152)
# opengl_manager.set_particle_color('gamma', color_name='red')
# Set the size of your world surrounding your simulation. It is important that this is␣
↪large enough because the field of view of the OpenGL viewport depends on this value.
opengl_manager.set_world_size(3.0, 3.0, 3.0, 'm')
# Size of OpenGL viewport
opengl_manager.set_window_dimensions(500, 500)
# Initialize OpenGL
opengl_manager.initialize()
# Show OpenGL window outside GGEMS simulation
opengl_manager.display()
```

If you have defined a navigator (phantom or detector) it is also possible to change the color of the material either by using a defined color or by using RGB indices. It is also possible to disable the color.

```
# Supposing you defined a phantom before composed by air and water
phantom.set_material_visible('Air', True) # or False if you do not want to draw the air␣
↪voxels
phantom.set_material_color('Water', color_name='blue')

# Supposing you define a CBCT before composed by GOS
cbct_detector.set_material_color('GOS', 255, 0, 0) # Custom color using RGB
#cbct_detector.set_material_color('GOS', color_name='red') # Using registered color
```

4.3 World

Outside the navigator, particles are not tracked. GGEMS includes a tool to record photons exiting the navigator. Particles are projected into the world using a DDA algorithm.

The world module is GGEMSWorld:

```
world = GGEMSWorld()
```

After creating a GGEMSWorld, the world dimensions and voxel size can be defined:

```
world.set_dimensions(200, 200, 200)
world.set_element_sizes(10.0, 10.0, 10.0, 'mm')
```

For world output, the user can save various information such as: photon energy and squared energy through voxel, photon momentum, and fluence (photon tracking):

```
world.set_output_basename('data/world')
world.energy_tracking(True)
world.energy_squared_tracking(True)
world.momentum(True)
world.photon_tracking(True)
```

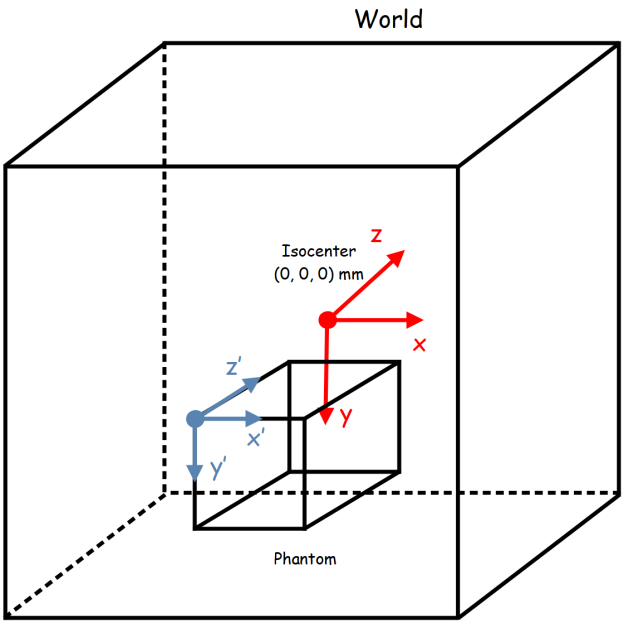
4.4 Navigators

In GGEMS, photons are tracked through a navigator. For each navigator, physical processes, a list of materials and a geometry are defined. The detector and phantom (patient or simulated object) are considered navigators.

Two types of navigators have been developed:

4.4.1 Voxelized Navigator

Voxelized navigator is useful for voxelized phantom or pixelized detector. The phantom must be described in a MHD file combined with a TXT file storing the material labels. The axis of the phantom corresponds to the axis of the world (global coordinates).



Create a voxelized phantom can be done by the following commands:

```
phantom = GGEMSVoxelizedPhantom('phantom')
phantom.set_phantom('phantom.mhd', 'range_phantom.txt')
```

In the 'range_phantom.txt' are the material labels (a voxel with a value of 3 corresponds to the lung):

```
0 0 Air
1 1 Water
2 2 RibBone
3 3 Lung
```

The phantom can be positioned and rotated in space using the following commands:

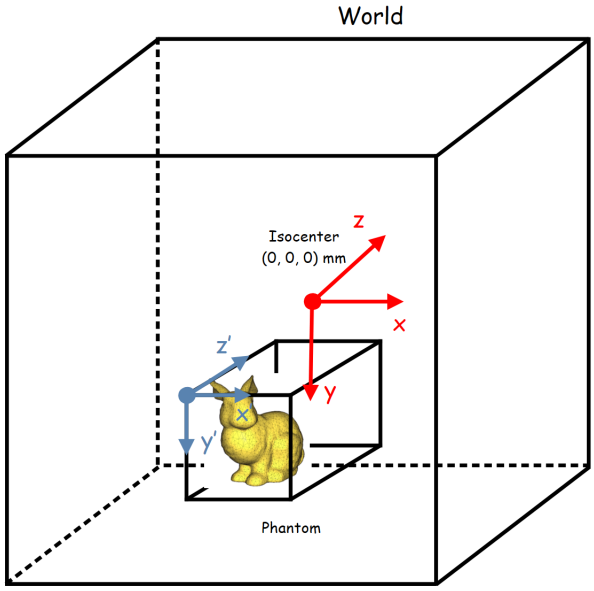
```
phantom.set_rotation(0.0, 0.0, 0.0, 'deg')
phantom.set_position(0.0, 0.0, 0.0, 'mm')
```

The voxelized phantom can be visualize by activating OpenGL. Disabling 'Air' is recommended:

```
phantom.set_visible(True)
phantom.set_material_visible('Air', False)
phantom.set_material_color('Water', color_name='blue')
```

4.4.2 Meshed Navigator

Meshed navigator is useful for simulating volumes with complicated geometry. The meshed phantom must be stored in a STL file (dedicated format describing surface using triangles). To reduce computation time, photon navigation through the triangles is performed via an octree. Then the triangles are stored in the octree subdivisions, called nodes. The octree division level can be set by the user. The axis of the phantom corresponds to the axis of the world (global coordinates). It is important to note that a meshed volume is associated with only one material.



Create a meshed phantom can be done by the following commands:

```
mesh_phantom = GGEMSMeshedPhantom('phantom_mesh')
mesh_phantom.set_phantom('data/Stanford_Bunny.stl')
```

The octree division level improves photon navigation. For instance, for level 1, the octree has no impact as all triangles are stored in the same node. At level 2, there are 8 nodes and navigation becomes more efficient because some triangles are excluded from the navigation. At level 3, there are 64 nodes, etc. The maximum level is 8, but this requires more memory. Level of 4 or 5 is recommended.

```
mesh_phantom.set_mesh_octree_depth(4)
```

The phantom can be positioned and rotated in space using the following commands:

```
mesh_phantom.set_rotation(90.0, 90.0, 0.0, 'deg')
mesh_phantom.set_position(0.0, 0.0, 0.0, 'mm')
```

A material is associted to a meshed phantom with this command:

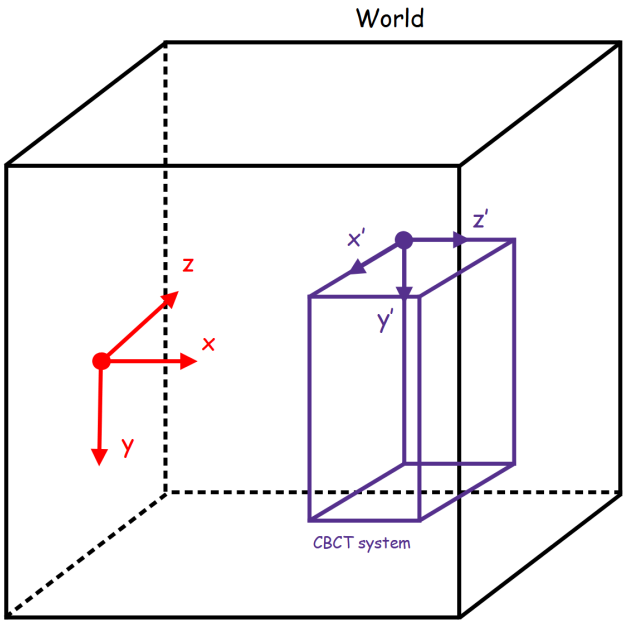
```
mesh_phantom.set_material('Water')
```

The voxelized phantom can be visualize by activating OpenGL. Disabling 'Air' is recommended:

```
mesh_phantom.set_visible(True)
mesh_phantom.set_material_color('Water', color_name='white') # Uncomment for automatic␣
↪color
```

4.5 CT/CBCT System

In GGEMS basic CT and CBCT systems are available. The detector is made up of pixels assembled in a module. The figure below shows the reference axis of the world (in red) as well as a CT/CBCT system with its axis (in purple).



A CT/CBCT system is created using the following line:

```
cbct_system = GGEMSCTSystem('detector') # detector is a custom name for your system
```

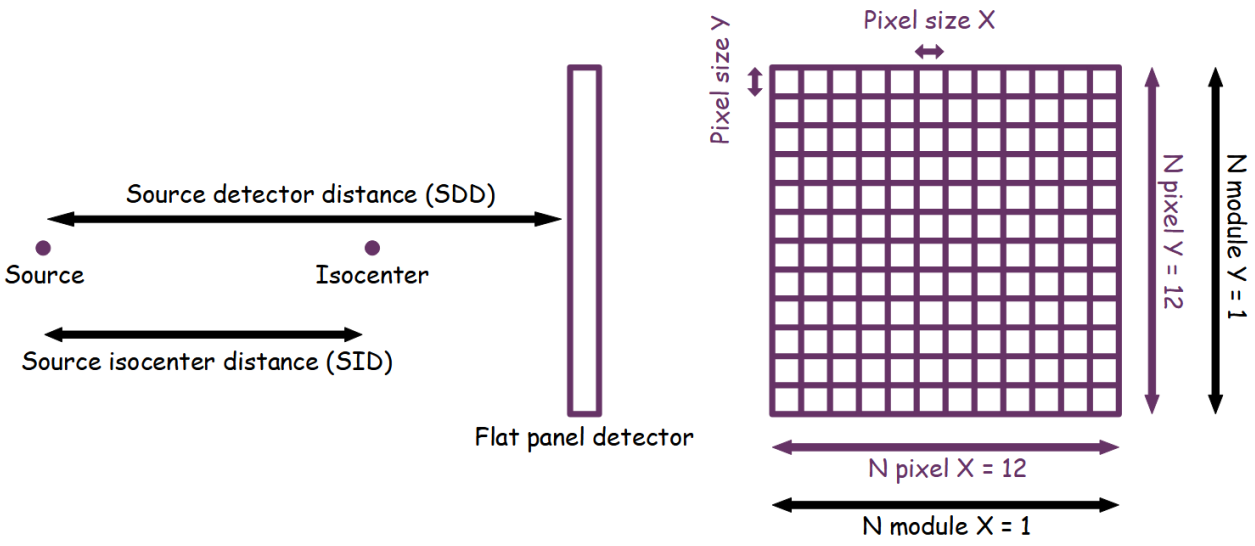
Types of CT/CBCT detector are:

- flat panel
- curved

4.5.1 Flat panel

This type of geometry is designed for CBCT systems.

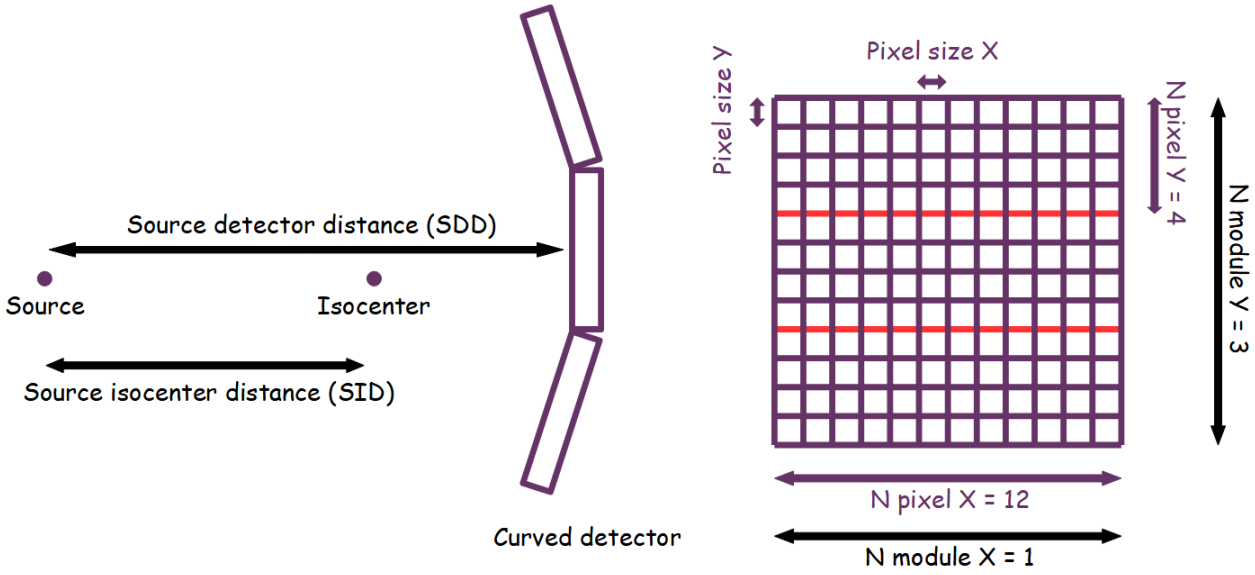
```
cbct_system.set_ct_type('flat')
```



4.5.2 Curved

This type of geometry is well suited for CT systems.

```
cbct_system.set_ct_type('curved')
```



Note

For curved geometry, the angle between modules is automatically calculated. There is no gap between the modules. The center of rotation of the system is the center of the world.

For each type of detector, the number of modules, the number of detection elements inside the module and their respective sizes are set as following:

```
cbct_system.set_number_of_modules(1, 3)
cbct_system.set_number_of_detection_elements(12, 4, 1)
cbct_system.set_size_of_detection_elements(1.0, 1.0, 1.0, 'mm')
```

A detector can be composed by only one type of material:

```
cbct_system.set_material('GOS')
```

An energy detection threshold can also be specified:

```
cbct_system.set_threshold(10.0, 'keV')
```

Source isocenter distance (SID) and source detector distance (SDD) is set with the following commands:

```
# Do not forget to add half size of detection element !!!
cbct_system.set_source_detector_distance(1500.5, 'mm')
cbct_system.set_source_isocenter_distance(900.0, 'mm')
```

Note

The position of the detector is calculated according to the values of the SID and SDD

A CT/CBCT system can be rotated around the world axis with the following command:

```
# 40 degree rotation around Z world axis
cbct_system.set_rotation(0.0, 0.0, 40.0, 'deg')
```

A CT/CBCT system can be translated along the world axis with the following command:

```
# 400 mm translation along Z world axis
cbct_system.set_global_system_position(0.0, 0.0, 400.0, 'mm');
```

The final projection including all photon interactions is saved in a file in MHD format, and scattered photons can be also save in another file.

```
cbct_system.save('projection')
cbct_system.store_scatter(True)
```

To enable detector visualization and change the default color:

```
cbct_detector.set_visible(True)
cbct_detector.set_material_color('GOS', 255, 0, 0) # Custom color using RGB
cbct_detector.set_material_color('GOS', color_name='red') # Or using registered color
```

4.6 Dosimetry

The dosimetry module can be activated to compute absorbed dose. Currently, only photons are simulated; electrons are ignored.

Dosimetry module is 'GGEMSDosimetryCalculator':

```
dosimetry = GGEMSDosimetryCalculator()
```

A navigator phantom must be attached to GGEMSDosimetryCalculator using:

```
dosimetry.attach_to_navigator('phantom')
```

Size of voxel (dosel) for dosimetry image could be set, if not, dosel size is the same than voxel phantom size:

```
dosimetry.set_dosel_size(0.5, 0.5, 0.5, 'mm')
```

Absorbed dose is computed in gray (Gy). By default, dose in computed using materials in phantom, otherwize the user can set water material everywhere in phantom.

```
dosimetry.water_reference(True)
```

A custom density threshold can be set. If density of phantom is below this threshold the dose value in 0 Gy.

```
dosimetry.minimum_density(0.1, 'g/cm3')
```

Since GGEMS v1.2, TLE method (Track Length Estimated) proposed in [Baldacci2014] can be activated to improve statistics.

```
dosimetry.set_tle(is_tle)
```

Many informations can be registered such as: uncertainty values of the dose, the deposited energy in dosel, the squared of deposited energy in dosel and the number of interactions (hits) in dosel:

```
dosimetry.set_output('dosimetry')
dosimetry.uncertainty(True)
dosimetry.edep(True)
dosimetry.hit(True)
dosimetry.edep_squared(True)
```

There is a special output named 'photon tracking'. This output registers the number of photons crossing a dosel. To use this option, the size of dosel has to be the same than the phantom voxel size, otherwize GGEMS will throw an error:

```
dosimetry.photon_tracking(True)
```


- F. Baldacci, A. Mittone, A. Bravin, P. Coan, F. Delaire, C. Ferrero, S. Gasilov, J. M. Létang, D. Sarrut, F. Smekens, et al., "A track length estimator method for dose calculations in low-energy x-ray irradiations: implementation, properties and performance", Zeitschrift Fur Medizinische Physik, 2014.

4.7 Physical Processes & Range cuts

4.7.1 Physical Processes

The photon processes impletemented are:

- Compton scattering
- Photoelectric effect
- Rayleigh scattering

Each of these processes are extracted from Geant4 version 10.6. For more informations about physics, please read the documentation on the Geant4 website.

Processes can be accessed with the following command:

```
processes_manager = GGEMSProcessesManager()
```

Important

Secondary particles (photon and electron) are not simulated yet. For Photoelectric effect, the photon is killed during the interaction and the energy is locally deposited, fluorescence photon is not emitted.

Compton Scattering

The Geant4 model extracted is the 'G4KleinNishinaCompton' standard model. Compton scattering is activated for all the navigators, or for a specific navigator.

```
# Activating Compton process for all navigators
processes_manager.add_process('Compton', 'gamma', 'all')

# Activating Compton process for a specific navigator name 'my_phantom'
processes_manager.add_process('Compton', 'gamma', 'my_phantom')
```

Photoelectric Effect

The Geant4 model extracted is the 'G4PhotoElectricEffect' standard model using Sandia tables. Photoelectric effect is activated for all the navigators, or for a specific navigator.

```
# Activating Photoelectric process for all navigators
processes_manager.add_process('Photoelectric', 'gamma', 'all')

# Activating Photoelectric process for a specific navigator name 'my_phantom'
processes_manager.add_process('Photoelectric', 'gamma', 'my_phantom')
```

Rayleigh Scattering

The Geant4 model extracted is the 'G4LivermoreRayleighModel' livermore model. Rayleigh scattering is activated for all the navigators, or for a specific navigator.

```
# Activating Rayleigh process for all navigators
processes_manager.add_process('Rayleigh', 'gamma', 'all')

# Activating Rayleigh process for a specific navigator name 'my_phantom'
processes_manager.add_process('Rayleigh', 'gamma', 'my_phantom')
```

Process Parameters Building

Cross sections are calculated during GGEMS initialization. The parameters for calculating the cross sections can be modified, but it is recommended to use the default ones. The parameters that can be modified are:

- Minimum energy of cross-section table
- Maximum energy of cross-section table
- Number of bins in cross-section table

Default parameters are defined as following:

```
processes_manager.set_cross_section_table_number_of_bins(220)
processes_manager.set_cross_section_table_energy_min(1.0, 'keV')
processes_manager.set_cross_section_table_energy_max(1.0, 'MeV')
```

Process Verbosity

Some informations about processes can be printed:

- Available processes
- Global informations about processes
- Cross-section values

The list of commands are:

```
processes_manager.print_available_processes()
processes_manager.print_infos()
processes_manager.print_tables(True)
```

4.7.2 Range Cuts

Cuts are defined for each particle in distance unit in all navigators or a specific navigator. During initialization cuts are converted in energy for each material in navigator. If the particle energy is below the cut, then this one is killed and the energy is locally deposited. By default cuts are 1 micron.

```
# For photon and all navigators
range_cuts_manager.set_cut('gamma', 0.1, 'mm', 'all')

# For photon and a specific navigator named 'my_phantom'
range_cuts_manager.set_cut('gamma', 0.1, 'mm', 'my_phantom')
```

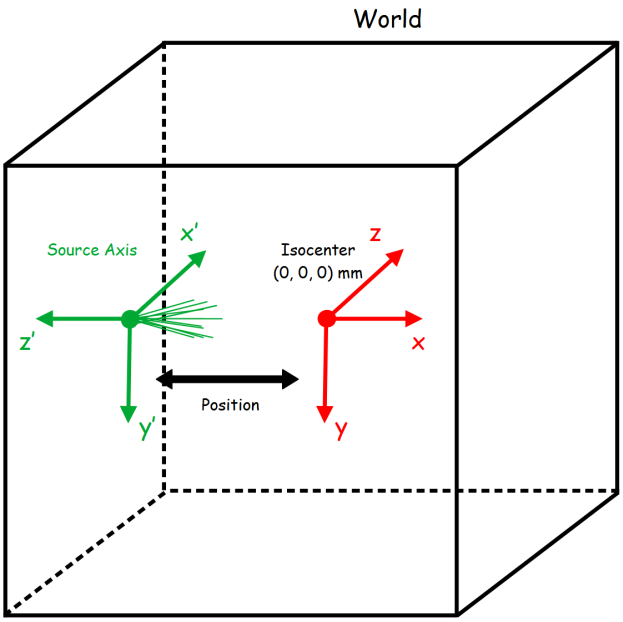
4.8 Sources

A Cone-beam X-Ray source has been developed in GGEMS for CT/CBCT application, and a voxelized source is also implemented. Voxelized source will be useful in next GGEMS releases for imagery application such as PET or SPECT.

4.8.1 CT/CBCT X-ray Source

X-Ray source is defined for a cone-beam geometry. The direction of the generated photons always point to the center of the world. The source has its own local axis.

The following commands define a basic X-ray source.




```
xray_source = GGEMSXRaySource('xray_source')
xray_source.set_source_particle_type('gamma')
xray_source.set_number_of_particles(1000000000)
# Position and rotation in global coordinate system
xray_source.set_position(-595.0, 0.0, 0.0, 'mm')
xray_source.set_rotation(0.0, 0.0, 0.0, 'deg')
xray_source.set_beam_aperture(12.5, 'deg')
# Focal spot size defined in local source axis
xray_source.set_focal_spot_size(0.0, 0.0, 0.0, 'mm')
```

Important

The focal spot size is defined in local source axis

4.8.2 Voxelized Source

Since GGEMS v1.3, it is possible to use a voxelised source. This will be useful when PET and SPECT imaging systems become available in GGEMS. The use of a voxelised source is very similar to that of a voxelised phantom. The user must provide a voxelised volume in MHD format, where an activity value (used as a weight) is assigned to each voxel. The source position can be defined by the user in the global coordinate system.

```
vox_source = GGEMSVoxelizedSource('vox_source')
vox_source.set_phantom_source('data/phantom_src.mhd')
vox_source.set_number_of_particles(100000)
vox_source.set_source_particle_type('gamma')
vox_source.set_position(0.0, 0.0, 0.0, 'mm')
```

4.8.3 Energy type

The energy source can be defined using a single energy value, a spectrum defined as a histogram in a TXT file or discrete energy peak

```
# Single energy value for a X-ray source at 25 keV
xray_source.set_monoenergy(25.0, 'keV')

# Polyenergetic source for a X-ray source
xray_source.set_polyenergy('data/spectrum_120kVp_2mmAl.dat')

# Voxelized source with 3 peak energies at 321, 112 and 208 keV with different weights
vox_source.set_energy_peak(321, 'keV', 0.0021)
vox_source.set_energy_peak(112, 'keV', 0.0617)
vox_source.set_energy_peak(208, 'keV', 0.1036)
```

4.9 GGEMS Commands

Useful commands to manage GGEMS and verbosity

```
# Create an GGEMS instance
ggems = GGEMS()
# Infos about OpenCL environment
ggems.opencl_verbose(True)
# Infos about whole material database
ggems.material_database_verbose(True)
# Infos about navigator (system/phantom)
ggems.navigator_verbose(True)
# Infos about source
ggems.source_verbose(True)
# Infos about memory usage
ggems.memory_verbose(True)
# Infos about activated processe(s)
ggems.process_verbose(True)
# Infos about range cuts
ggems.range_cuts_verbose(True)
# Print infos about first random state and initial seed
ggems.random_verbose(True)
# Infos about elapsed time in kernels
ggems.profiling_verbose(True)
# Infos about a specific photon index (here photon index 12)
ggems.tracking_verbose(True, 12)
```

Important commands to initialize and run a GGEMS application:

```
# Initialization with a specific seed (here 9383) or let GGEMS get a random seed
ggems.initialize(777)
# ggems.initialize() # random seed
# Running ggems
ggems.run()
```

4.10 Examples & Tools

Examples and tools are provided for GGEMS users. It could be useful to start a new project using one of these examples.

4.10.1 Examples 0: Cross-Section Computation

A cross-section computation for a specific material, energy and process

```
$ python cross_sections.py [-h] [-d DEVICE] [-m MATERIAL] -p [PROCESS]-e [ENERGY] [-v␣
↪VERBOSE]
-h/--help           Printing help into the screen
-d/--device         Setting OpenCL id
-m/--material       Setting one of material defined in GGEMS (Water, Air, ...)
-p/--process        Setting photon physical process (Compton, Rayleigh, Photoelectric)
-e/--energy         Setting photon energy in MeV
-v/--verbose        Setting level of verbosity
```

Verbosity level is defined in the range [0;3]. For a silent GGEMS execution, the level is set to 0, otherwise 3 for lot of informations.

```
GGEMSVerbosity(verbosity_level)

# Select a OpenCL device
opencl_manager.set_device_index(device_id)
```

Create a GGEMSMaterial instance then add a material. The initialization step is mandatory and compute all physical tables, and store them on an OpenCL device:

```
materials = GGEMSMaterials()
materials.add_material(material_name)
materials.initialize()
```

Create a GGEMSCrossSection instance and activate a process:

```
cross_sections = GGEMSCrossSections(materials)
cross_sections.add_process(process_name, 'gamma')
cross_sections.initialize()
```

For attenuation informations, create a GGEMSAttenuations:

```
attenuations = GGEMSAttenuations(materials, cross_sections)
attenuations.initialize();
```

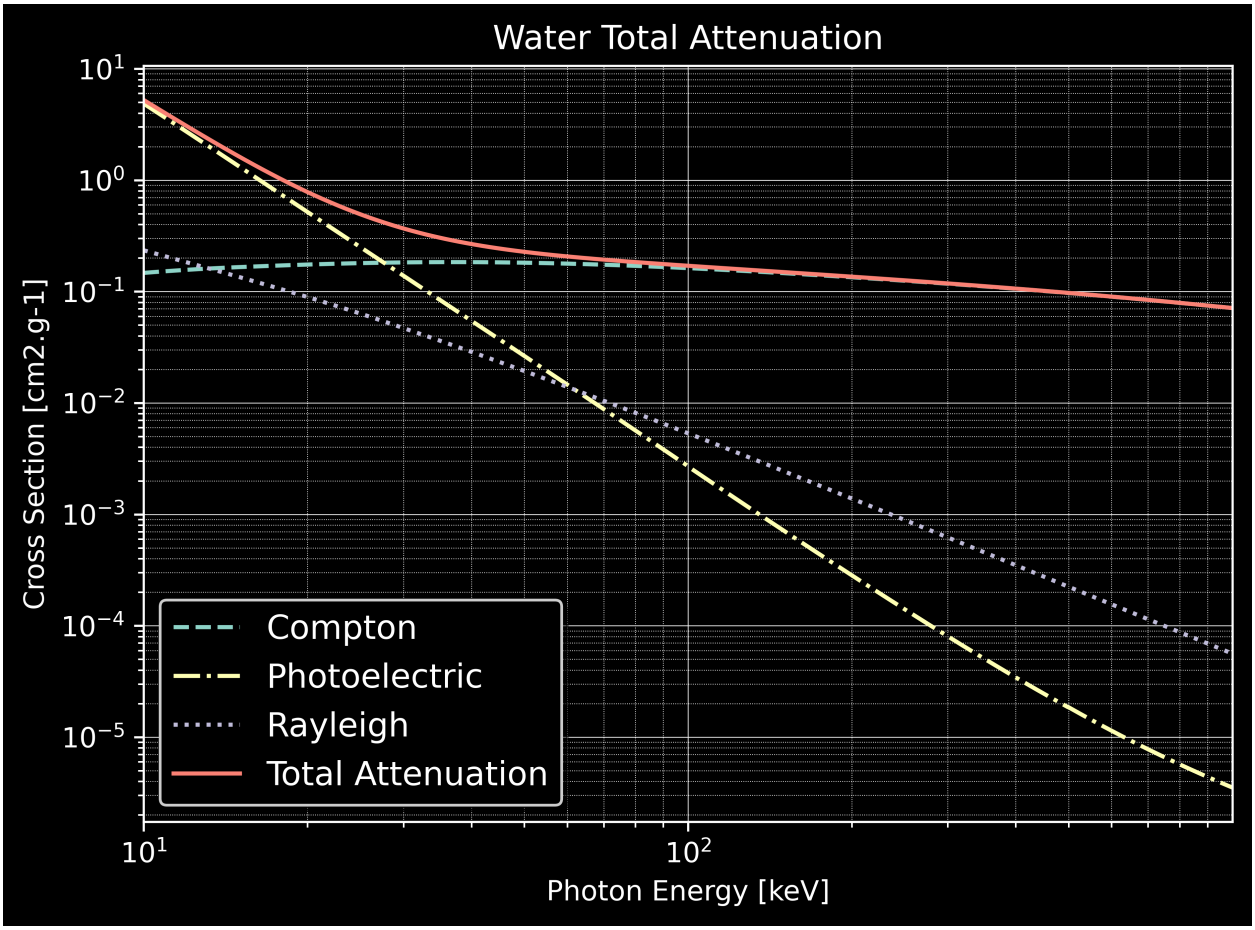
Computing cross section value (in cm2.g-1) for a specific energy (in MeV):

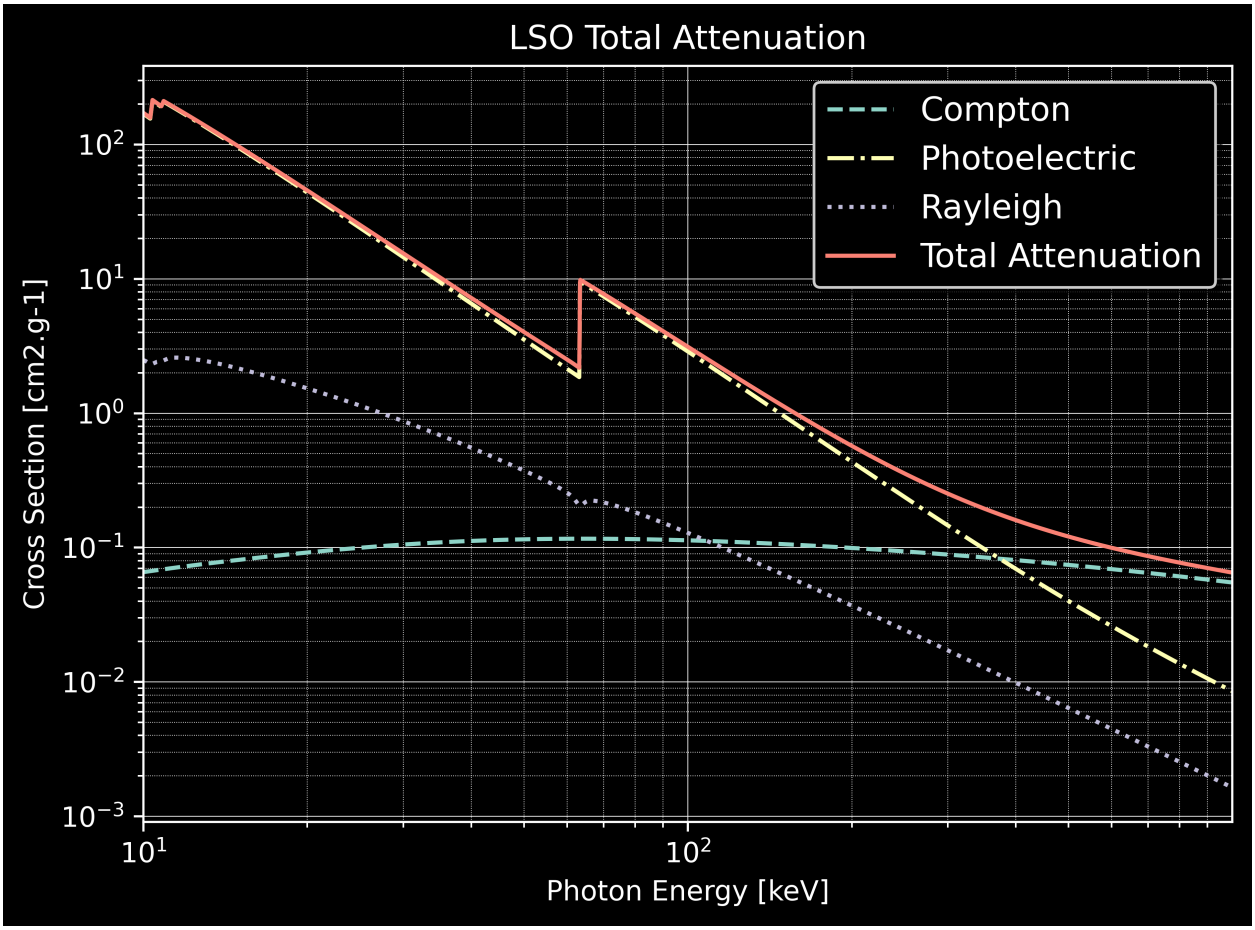
```
cross_sections.get_cs(process_name, material_name, energy_MeV, 'MeV')

# Get attenuation value in cm-1
attenuations.get_mu(material_name, energy_MeV, 'MeV')
# Get energy attenuation value
attenuations.get_mu_en(material_name, energy_MeV, 'MeV')
```

4.10.2 Examples 1: Total Attenuation

```
$ python total_attenuation.py [-h] [-d DEVICE] [-m MATERIAL] [-v VERBOSE]
-h/--help           Printing help into the screen
-d/--device         Setting OpenCL id
-m/--material       Setting one of material defined in GGEMS (Water, Air, ...)
-v/--verbose        Setting level of verbosity
```

Total attenuations for Water and LSO are shown below:





4.10.3 Examples 2: CT Scanner

CT scanner example : a water box is simulated associated to a CT curved detector. One projection is computed simulating 1e9 particles.

```
$ python ct_scanner.py [-h] [-d DEVICE] [-b BALANCE] [-n N_PARTICLES] [-s SEED] [-v␣
↪VERBOSE]
-h/--help           Printing help into the screen
-d/--device         OpenCL device (all, cpu, gpu, gpu_nvidia, gpu_intel, gpu_amd, "X;Y;Z
↪"...)
                    using all gpu: -d gpu
                    using device index 0 and 2: -d "0;2"
-b/--balance        Balance computation for device if many devices are selected "X;Y;Z"
                    60% computation on device 0 and 40% computatio on device 2: -b "0.6;
↪0.4"
-n/--nparticles     Number of particles (default: 1000000)
-s/--seed           Seed of pseudo generator number (default: 777)
-v/--verbose        Setting level of verbosity
```

Load a water box phantom:

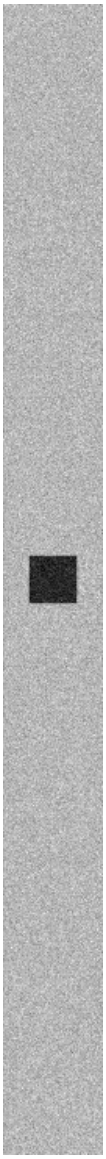
```
phantom = GGEMSVoxelizedPhantom('phantom')
phantom.set_phantom('data/phantom.mhd', 'data/range_phantom.txt')
phantom.set_rotation(0.0, 0.0, 0.0, 'deg')
phantom.set_position(0.0, 0.0, 0.0, 'mm')
```

Create a CT curved detector:

```
ct_detector = GGEMSCTSystem('Stellar')
ct_detector.set_ct_type('curved')
ct_detector.set_number_of_modules(1, 46)
ct_detector.set_number_of_detection_elements(64, 16, 1)
ct_detector.set_size_of_detection_elements(0.6, 0.6, 0.6, 'mm')
ct_detector.set_material('GOS')
ct_detector.set_source_detector_distance(1085.6, 'mm')
ct_detector.set_source_isocenter_distance(595.0, 'mm')
ct_detector.set_rotation(0.0, 0.0, 0.0, 'deg')
ct_detector.set_threshold(10.0, 'keV')
ct_detector.save('data/projection')
ct_detector.store_scatter(True)
```

Create a cone-beam X-ray source:

```
point_source = GGEMSXRaySource('point_source')
point_source.set_source_particle_type('gamma')
point_source.set_number_of_particles(1000000000)
point_source.set_position(-595.0, 0.0, 0.0, 'mm')
point_source.set_rotation(0.0, 0.0, 0.0, 'deg')
point_source.set_beam_aperture(12.5, 'deg')
point_source.set_focal_spot_size(0.0, 0.0, 0.0, 'mm')
point_source.set_polyenergy('data/spectrum_120kVp_2mmAl.dat')
```



Performance on Windows 11 system and Visual C++ 2022:

tabularytabulary

Device
GeForce GTX 1050 Ti
Quadro P400
RTX 4000
Xeon X-2245 8 cores / 16 threads
GeForce GTX 1050 Ti (80%) Quadro P400 (20%)

Performance on Ubuntu 20.04 and GNU GCC 9.3:

tabularytabulary

Device
GeForce GTX 1050 Ti
Quadro P400
Xeon X-2245 8 cores / 16 threads
GeForce GTX 1050 Ti (80%) Quadro P400 (20%)

4.10.4 Examples 3: Voxelized Phantom Generator

A voxelized phantom can be created using GGEMS. Only basic shapes are available such as tube, box and sphere. Output format is MHD, and a range material data is also generated.

```
$ python generate_volume.py [-h] [-d DEVICE] [-v VERBOSE]
-h/--help           Printing help into the screen
-d/--device         Setting OpenCL id
-v/--verbose        Setting level of verbosity
```

Create a global volume storing all voxelized objets:

```
volume_creator_manager = GGEMSVolumeCreatorManager()

volume_creator_manager.set_dimensions(450, 450, 450)
volume_creator_manager.set_element_sizes(0.5, 0.5, 0.5, "mm")
volume_creator_manager.set_output('data/volume')
volume_creator_manager.set_range_output('data/range_volume')
volume_creator_manager.set_material('Air')
volume_creator_manager.set_data_type('MET_INT')
volume_creator_manager.initialize()
```

Draw an object in previous global volume:

```
# Creating a box
box = GGEMSBox(24.0, 36.0, 56.0, 'mm')
box.set_position(-70.0, -30.0, 10.0, 'mm')
box.set_label_value(1)
box.set_material('Water')
box.initialize()
box.draw()
box.delete()

# Creating a tube
tube = GGEMSTube(13.0, 8.0, 50.0, 'mm')
tube.set_position(20.0, 10.0, -2.0, 'mm')
tube.set_label_value(2)
tube.set_material('Calcium')
tube.initialize()
tube.draw()
tube.delete()

# Creating a sphere
sphere = GGEMSSphere(14.0, 'mm')
sphere.set_position(30.0, -30.0, 8.0, 'mm')
sphere.set_label_value(3)
```

(continues on next page)

(continued from previous page)

```
sphere.set_material('Lung')
sphere.initialize()
sphere.draw()
sphere.delete()
```

4.10.5 Examples 4: Dosimetry

A cylinder is simulated computing absorbed dose. Different results such as dose and energy deposited are registered in MHD files. An external source, using GGEMS X-ray source is simulated generating 2e8 particles and TLE is activated to improve statistics.

```
$ python dosimetry_photon.py [-h] [-d DEVICE] [-b BALANCE] [-n N_PARTICLES] [-s SEED] [-
↪v VERBOSE] [-t]
-h/--help           Printing help into the screen
-d/--device         OpenCL device (all, cpu, gpu, gpu_nvidia, gpu_intel, gpu_amd, "X;Y;Z
↪"...)
                    using all gpu: -d gpu
                    using device index 0 and 2: -d "0;2"
-b/--balance        Balance computation for device if many devices are selected "X;Y;Z"
                    60% computation on device 0 and 40% computatio on device 2: -b "0.6;
↪0.4"
-n/--nparticles     Number of particles (default: 1000000)
-t/--tle            Activating TLE method
-s/--seed           Seed of pseudo generator number (default: 777)
-v/--verbose        Setting level of verbosity
```

Load voxelized phantom:

```
phantom = GGEMSVoxelizedPhantom('phantom')
phantom.set_phantom('data/phantom.mhd', 'data/range_phantom.txt')
phantom.set_rotation(0.0, 0.0, 0.0, 'deg')
phantom.set_position(0.0, 0.0, 0.0, 'mm')
```

Dosimetry associated to the previous phantom:

```
dosimetry = GGEMSDosimetryCalculator('phantom')
dosimetry.set_output('data/dosimetry')
dosimetry.set_dosel_size(0.5, 0.5, 0.5, 'mm')
dosimetry.water_reference(False)
dosimetry.minimum_density(0.1, 'g/cm3')
dosimetry.set_tle(is_tle)

dosimetry.uncertainty(True)
dosimetry.photon_tracking(True)
dosimetry.edep(True)
dosimetry.hit(True)
dosimetry.edep_squared(True)
```

External source using GGEMSXRaySource:

```
point_source = GGEMSXRaySource('point_source')
point_source.set_source_particle_type('gamma')
```

(continues on next page)

(continued from previous page)

```
point_source.set_number_of_particles(200000000)
point_source.set_position(-595.0, 0.0, 0.0, 'mm')
point_source.set_rotation(0.0, 0.0, 0.0, 'deg')
point_source.set_beam_aperture(5.0, 'deg')
point_source.set_focal_spot_size(0.0, 0.0, 0.0, 'mm')
point_source.set_polyenergy('data/spectrum_120kVp_2mmAl.dat')
```

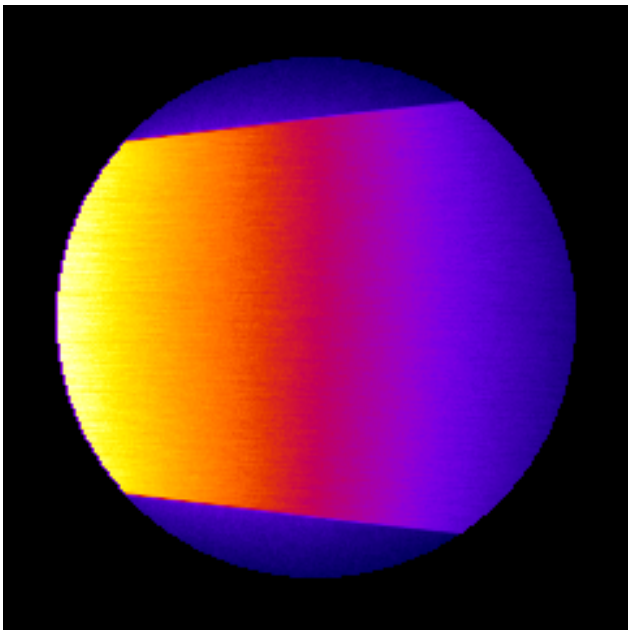


Fig. 1: Dose absorbed by cylinder phantom

Performance on Windows 11 system and Visual C++ 2022:

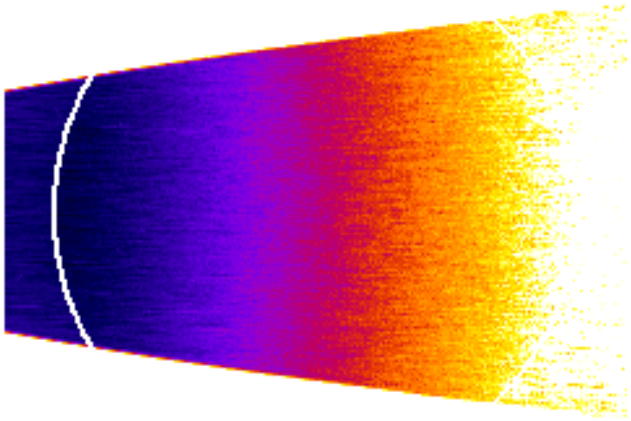


Fig. 2: Uncertainty dose computation

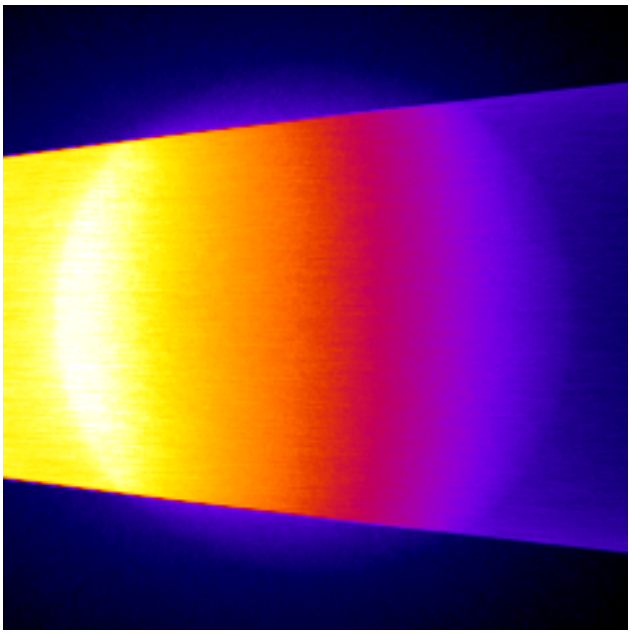


Fig. 3: Photon tracking in phantom

tabularytabulary

	Xeon X-
GeForce GTX 1050 Ti (58%)	Xeon X-2245 8

Performance on Ubuntu 20.04 and GNU GCC 9.3:

tabularytabulary

	Xeon X-
GeForce GTX 1050 Ti (55%)	Xeon X-2245 8

4.10.6 Examples 5: World Tracking

A cylinder is simulated computing absorbed dose, and a CBCT flat panel detector is also simulated. A GGEMSWorld volume is created in order to store the fluence outside cylinder phantom and detector. An external source, using GGEMS X-ray source is simulated generating 1e8 particles.

```
$ python world_tracking.py [-h] [-d DEVICE] [-b BALANCE] [-n N_PARTICLES] [-s SEED] [-v␣
↪VERBOSE]
-h/--help           Printing help into the screen
-d/--device         OpenCL device (all, cpu, gpu, gpu_nvidia, gpu_intel, gpu_amd, "X;Y;Z
↪"...)
                    using all gpu: -d gpu
                    using device index 0 and 2: -d "0;2"
-b/--balance        Balance computation for device if many devices are selected "X;Y;Z"
                    60% computation on device 0 and 40% computatio on device 2: -b "0.6;
↪0.4"
-n/--nparticles     Number of particles (default: 1000000)
-s/--seed           Seed of pseudo generator number (default: 777)
-v/--verbose        Setting level of verbosity
```

World definition:

```
world = GGEMSWorld()
world.set_dimensions(200, 200, 200)
world.set_element_sizes(10.0, 10.0, 10.0, 'mm')
world.set_output_basename('data/world')

world.energy_tracking(True)
world.energy_squared_tracking(True)
world.momentum(True)
world.photon_tracking(True)
```

Cylinder phantom is loaded and dosimetry module is associated to cylinder phantom, and all output are activated

```
phantom = GGEMSVoxelizedPhantom('phantom')
phantom.set_phantom('data/phantom.mhd', 'data/range_phantom.txt')
```

(continues on next page)

(continued from previous page)

```
phantom.set_rotation(0.0, 0.0, 0.0, 'deg')
phantom.set_position(0.0, 0.0, 0.0, 'mm')

dosimetry = GGEMSDosimetryCalculator()
dosimetry.attach_to_navigator('phantom')
dosimetry.set_output_basename('data/dosimetry')
dosimetry.set_dosel_size(1.0, 1.0, 1.0, 'mm')
dosimetry.water_reference(False)
dosimetry.minimum_density(0.1, 'g/cm3')

dosimetry.uncertainty(True)
dosimetry.photon_tracking(True)
dosimetry.edep(True)
dosimetry.hit(True)
dosimetry.edep_squared(True)
```

A CBCT flat panel detector definition:

```
cbct_detector = GGEMSCTSystem('custom')
cbct_detector.set_ct_type('flat')
cbct_detector.set_number_of_modules(1, 1)
cbct_detector.set_number_of_detection_elements(400, 400, 1)
cbct_detector.set_size_of_detection_elements(1.0, 1.0, 10.0, 'mm')
cbct_detector.set_material('Silicon')
cbct_detector.set_source_detector_distance(1500.0, 'mm')
cbct_detector.set_source_isocenter_distance(900.0, 'mm')
cbct_detector.set_rotation(0.0, 0.0, 0.0, 'deg')
cbct_detector.set_threshold(10.0, 'keV')
cbct_detector.save('data/projection.mhd')
```

External source using GGEMSXRaySource:

```
point_source = GGEMSXRaySource('point_source')
point_source.set_source_particle_type('gamma')
point_source.set_number_of_particles(100000000)
point_source.set_position(-900.0, 0.0, 0.0, 'mm')
point_source.set_rotation(0.0, 0.0, 0.0, 'deg')
point_source.set_beam_aperture(12.0, 'deg')
point_source.set_focal_spot_size(0.0, 0.0, 0.0, 'mm')
point_source.set_monoenergy(60.0, 'keV')
```

Performance on Windows 11 system and Visual C++ 2022:

tabularytabulary

Device
GeForce GTX 1050 Ti
Quadro P400
RTX 4000
Xeon X-2245 8 cores / 16 threads
GeForce GTX 1050 Ti (50%) Quadro P400 (50%)

Performance on Ubuntu 20.04 and GNU GCC 9.3:

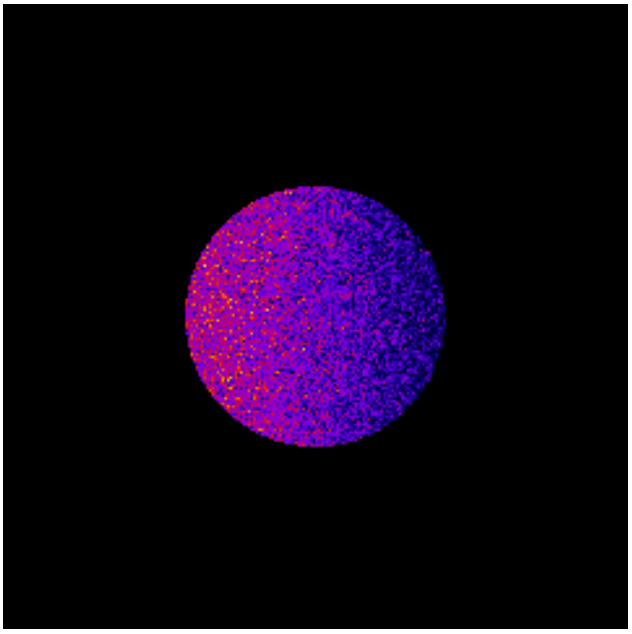


Fig. 4: Dose absorbed by cylinder phantom

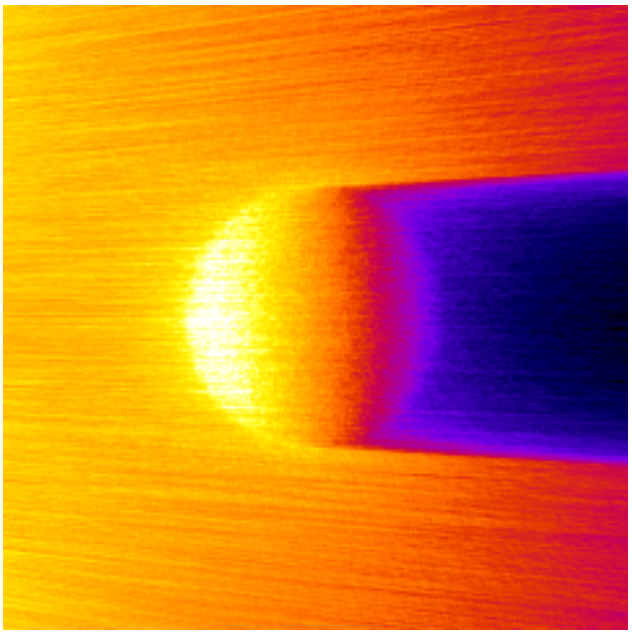


Fig. 5: Photon tracking in phantom

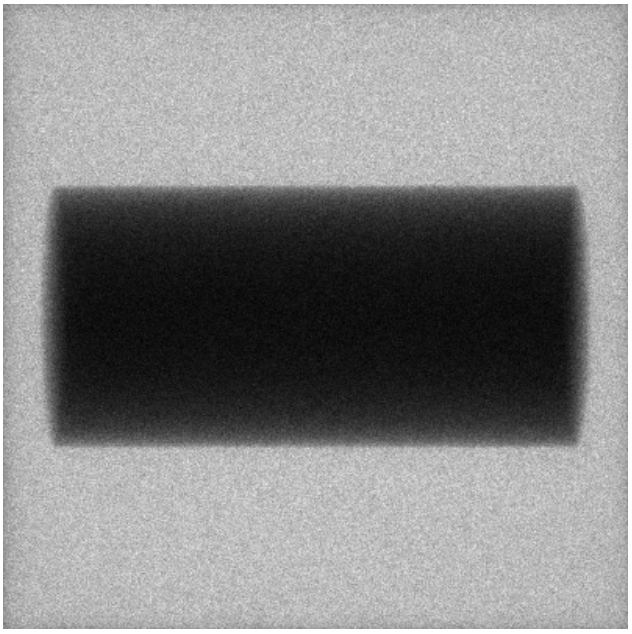


Fig. 6: Cylinder projection on flat panel detector

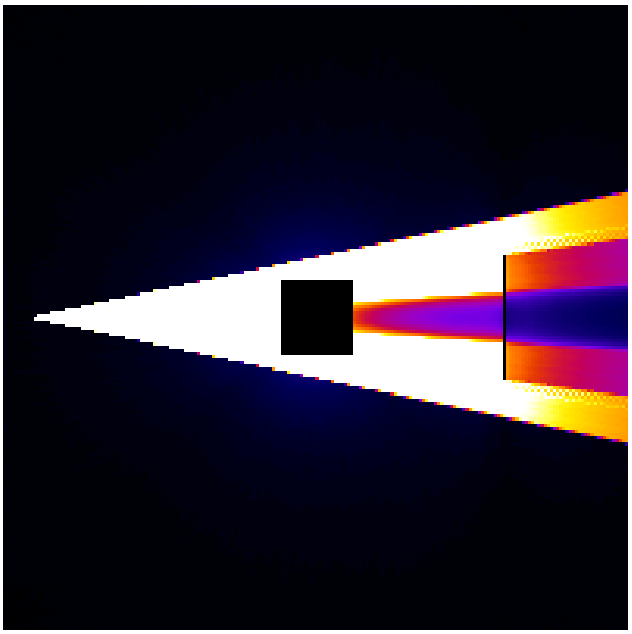


Fig. 7: World photon tracking

tabularytabulary

Device
GeForce GTX 1050 Ti
Quadro P400
Xeon X-2245 8 cores / 16 threads
GeForce GTX 1050 Ti (40%) Quadro P400 (60%)

4.10.7 Examples 6: OpenGL Visualization

OpenGL is available since GGEMS v1.2.

```
python visualization.py [-h] [-v VERBOSE] [-e] [-w WDIMS WDIMS] [-m MSAA] [-a] [-p␣
↪NPARTICLESGL] [-b] [-c WCOLOR] [-d DEVICE] [-n NPARTICLES] [-s SEED]
-h/--help             show this help message and exit
-v/--verbose          Set level of verbosity
-e/--nogl             Disable OpenGL
-w/--wdims            OpenGL window dimensions (default: [800, 800])
                      for a 400x400 window: -w 400 400
-m/--msaa             MSAA factor (1x, 2x, 4x or 8x) (default: 8)
-a/--axis             Drawing axis in OpenGL window
-p/--nparticlesgl     Number of displayed primary particles on OpenGL window (max: 65536)␣
↪(default: 256)
-b/--drawgeom         Draw geometry only on OpenGL window (default: False)
-c/--wcolor           Background color of OpenGL window (default: black)
                      using blue color for background: -c "blue"
-d/--device           OpenCL device running visualization (default: 0)
                      only 1 device available for this example
                      using device index 0: -d 0
-n/--nparticles       Number of particles (default: 1000000)
-s/--seed             Seed of pseudo generator number (default: 777)
```

Create instance of GGEMSOpenGLManager:

```
opengl_manager = GGEMSOpenGLManager()
```

Many parameters can be set for OpenGL:

```
opengl_manager = GGEMSOpenGLManager()
opengl_manager.set_window_dimensions(window_dims[0], window_dims[1])
opengl_manager.set_msaa(msaa)
opengl_manager.set_background_color(window_color)
opengl_manager.set_draw_axis(is_axis)
opengl_manager.set_world_size(3.0, 3.0, 3.0, 'm')
opengl_manager.set_image_output('data/axis')
opengl_manager.set_displayed_particles(number_of_displayed_particles)
opengl_manager.set_particle_color('gamma', 152, 251, 152)
# opengl_manager.set_particle_color('gamma', color_name='red') # Using registered color
opengl_manager.initialize()
```

Display only geometry and system:

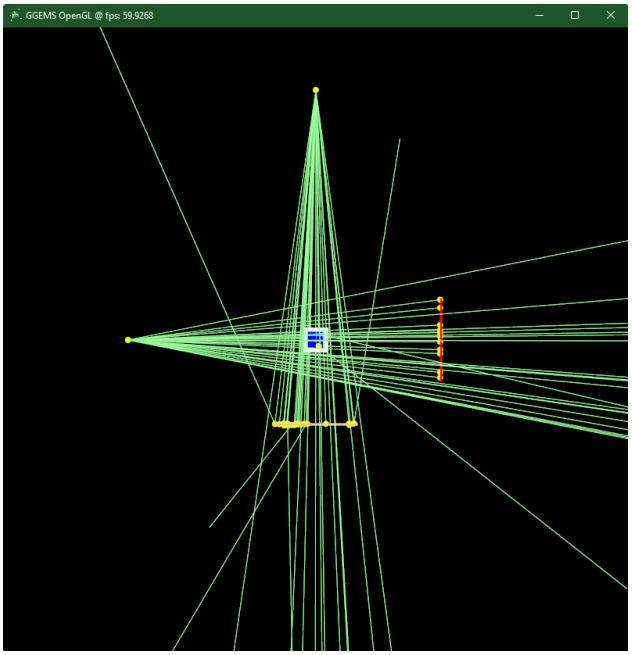
```
opengl_manager.display()
```

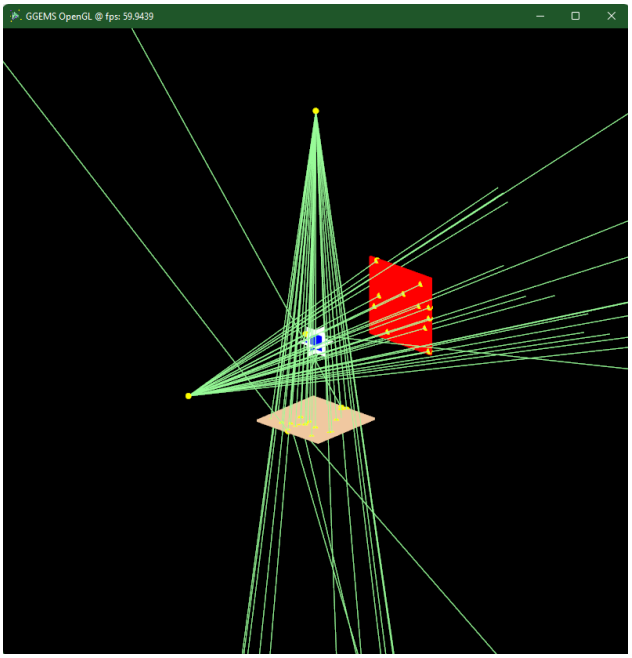
Using GGEMS for a complete visualization (geometry, system and particles):

```
ggems.run()
```

The OpenGL window is interactive, the scene can be moved using keyboard or mouse:

```
Keys:
    * [Esc/X]              Quit application
    * [C]                  Wireframe view
    * [V]                  Solid view
    * [R]                  Reset view
    * [K]                  Save current window to a PNG file
    * [+/-]                Zoom in/out
    * [Up/Down]            Move forward/back
    * [W/S]
    * [Left/Right]         Move left/right
    * [A/D]

Mouse:
    * [Scroll Up/Down]     Zoom in/out
    * [Left button]        Rotation
    * [Middle button]      Translation
```





4.10.8 Examples 7: Meshed Navigator

Photon tracking in a meshed volume has been available since GGEMS v1.3.

```
python mesh.py [-h] [-v VERBOSE] [-e] [-w WDIMS WDIMS] [-m MSAA] [-a] [-p NPARTICLESGL]␣
↪[-b] [-c WCOLOR] [-d DEVICE] [-n NPARTICLES] [-s SEED]
-h, --help            show this help message and exit
-v, --verbose VERBOSE
                      Set level of verbosity (default: 0)
-e, --nogl            Disable OpenGL (default: True)
-w, --wdims WDIMS WDIMS
                      OpenGL window dimensions (default: [800, 800])
-m, --msaa MSAA       MSAA factor (1x, 2x, 4x or 8x) (default: 8)
-a, --axis            Drawing axis in OpenGL window (default: False)
-p, --nparticlesgl NPARTICLESGL
                      Number of displayed primary particles on OpenGL window (max:␣
↪65536) (default: 256)
-b, --drawgeom        Draw geometry only on OpenGL window (default: False)
-c, --wcolor WCOLOR   Background color of OpenGL window (default: black)
-d, --device DEVICE   OpenCL device running visualization (default: 0)
-n, --nparticles NPARTICLES
                      Number of particles (default: 1000000)
-s, --seed SEED       Seed of pseudo generator number (default: 777)
```

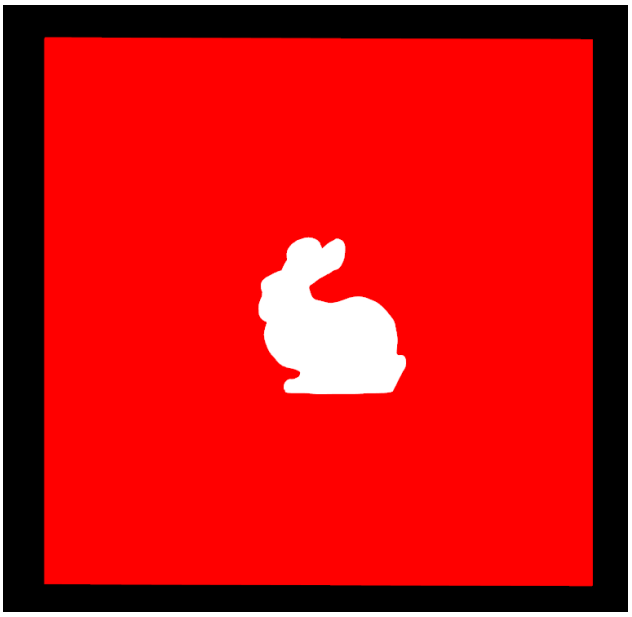
As an example, a famous meshed rabbit volume is used. The projection of the meshed volumed simulating 1e9 photons.

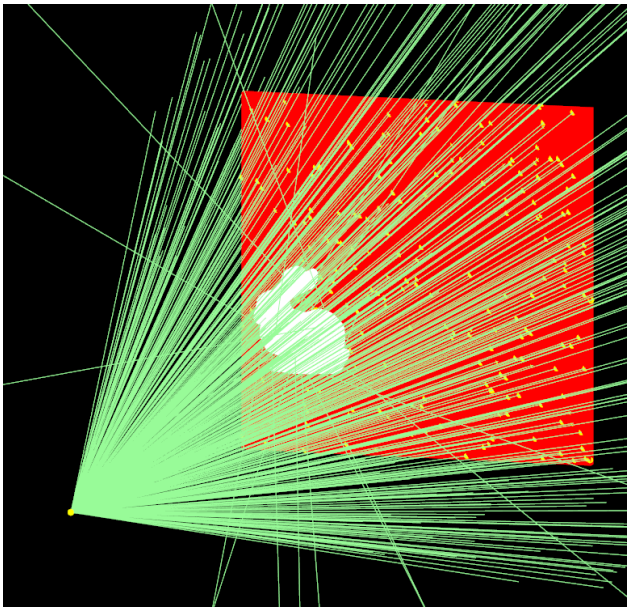
The meshed volume is defined in GGEMS with theses commands

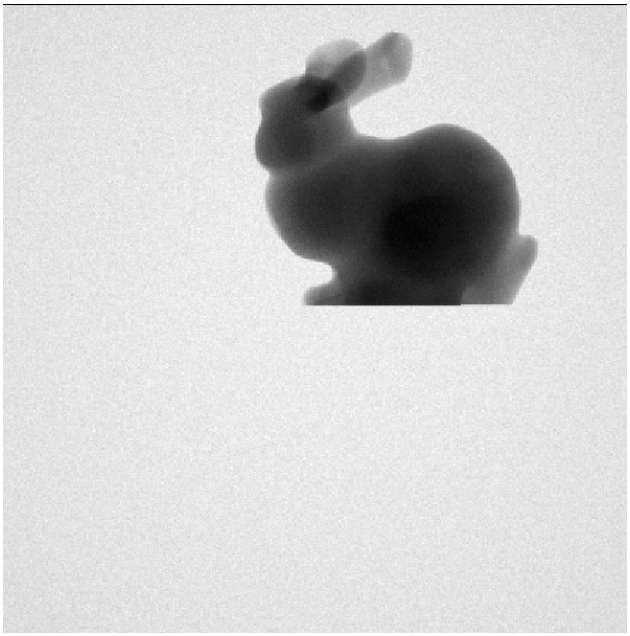
```
# Loading phantom in GGEMS
mesh_phantom = GGEMSMeshedPhantom('phantom_mesh')
mesh_phantom.set_phantom('data/Stanford_Bunny.stl')
mesh_phantom.set_rotation(90.0, 90.0, 0.0, 'deg')
mesh_phantom.set_position(0.0, 0.0, 0.0, 'mm')
mesh_phantom.set_mesh_octree_depth(4)
mesh_phantom.set_visible(True)
mesh_phantom.set_material('Water')
```

A flat panel detector defined with 400x400 pixels is used

```
cbct_detector = GGEMSCTSystem('custom')
cbct_detector.set_ct_type('flat')
cbct_detector.set_number_of_modules(1, 1)
cbct_detector.set_number_of_detection_elements(400, 400, 1)
cbct_detector.set_size_of_detection_elements(1.0, 1.0, 10.0, 'mm')
cbct_detector.set_material('GOS')
cbct_detector.set_source_detector_distance(1500.5, 'mm') # Center of inside detector,␣
↪adding half of detector (= SDD surface + 10.0/2 mm half of depth)
cbct_detector.set_source_isocenter_distance(900.0, 'mm')
cbct_detector.set_threshold(10.0, 'keV')
cbct_detector.save('data/projection')
```







Performance on Windows 11 system and Visual C++ 2022:

tabularytabulary

	Device	Computation Time [
	Quadro P400	170
	RTX 4000	2
	Xeon X-2245 8 cores / 16 threads	190

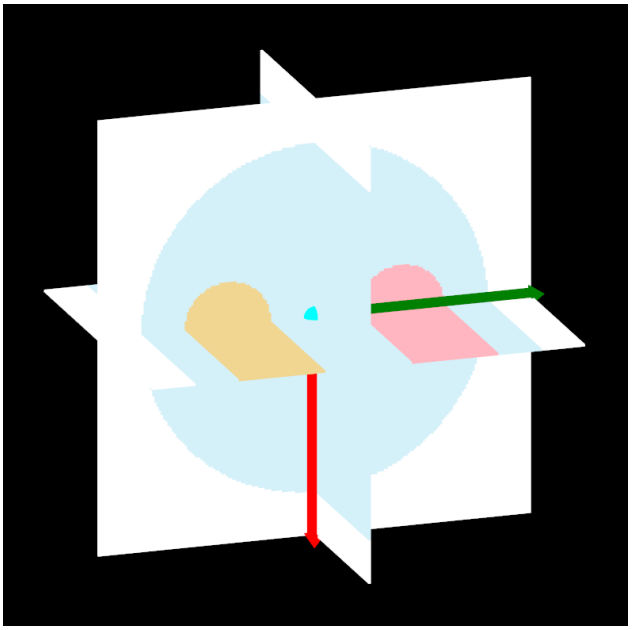
4.10.9 Examples 8: Voxelized Source

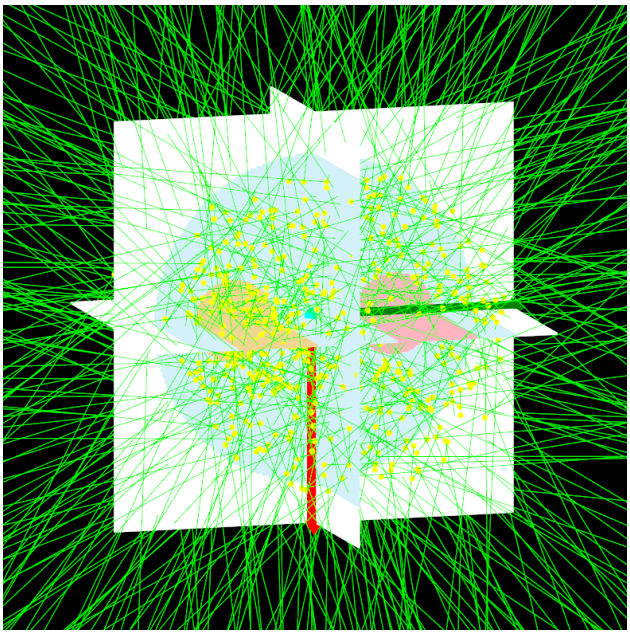
The voxelized source is available since GGEMS v1.3.

```
python voxelized_source.py [-h] [-d DEVICE] [-b BALANCE] [-s SEED] [-v VERBOSE] [-e] [-p␣
↪NPARTICLESGL] [-c] [-n NPARTICLES]
-h, --help            show this help message and exit
-d, --device DEVICE   OpenCL device (all, cpu, gpu, gpu_nvidia, gpu_intel, gpu_amd, X;Y;
↪Z...) (default: 0)
-b, --balance BALANCE
                      X;Y;Z... Balance computation for device if many devices are␣
↪selected. -b "0.5;0.5" means 50 % of computation on device 0, and 50 % of computation␣
↪on device 1 (default: None)
-s, --seed SEED       Seed of pseudo generator number (default: 777)
-v, --verbose VERBOSE
                      Set level of verbosity (default: 0)
-e, --nogl            Disable OpenGL (default: True)
-p, --nparticlesgl NPARTICLESGL
                      Number of displayed primary particles on OpenGL window (max:␣
↪65536) (default: 256)
-c, --drawgeom        Draw geometry only on OpenGL window (default: False)
-n, --nparticles NPARTICLES
                      Number of particles (default: 1000000)
```

A 177Lu voxelized source example is provided

```
vox_source = GGEMSVoxelizedSource('vox_source')
vox_source.set_phantom_source('data/phantom_src.mhd')
vox_source.set_number_of_particles(number_of_particles)
vox_source.set_source_particle_type('gamma')
vox_source.set_position(0.0, 0.0, 0.0, 'mm')

#177Lu source
vox_source.set_energy_peak(321.3, 'keV', 0.0021)
vox_source.set_energy_peak(249.7, 'keV', 0.0020)
vox_source.set_energy_peak(112.9, 'keV', 0.0617)
vox_source.set_energy_peak( 71.6, 'keV', 0.0017)
vox_source.set_energy_peak(208.4, 'keV', 0.1036)
vox_source.set_energy_peak(136.7, 'keV', 0.0005)
```





CHAPTER

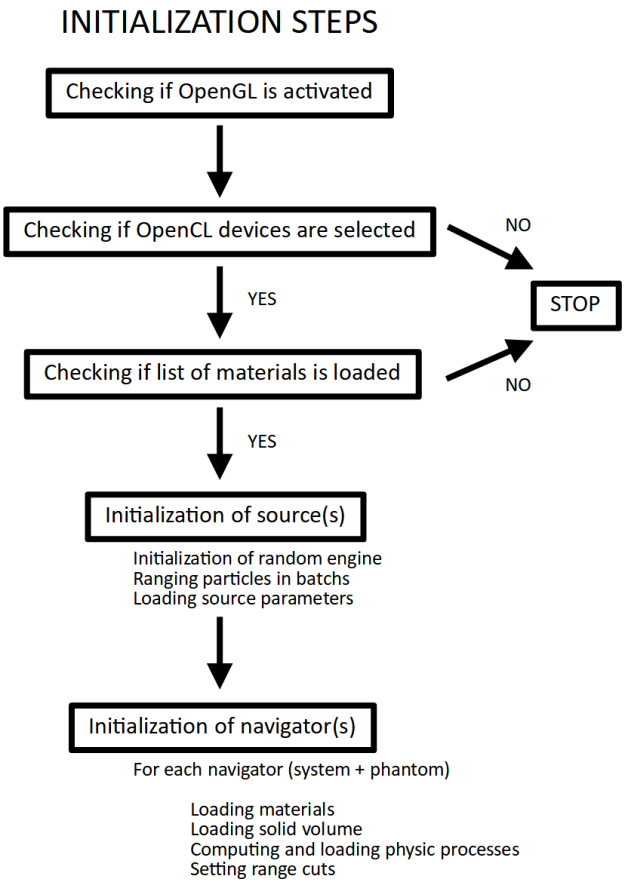
FIVE

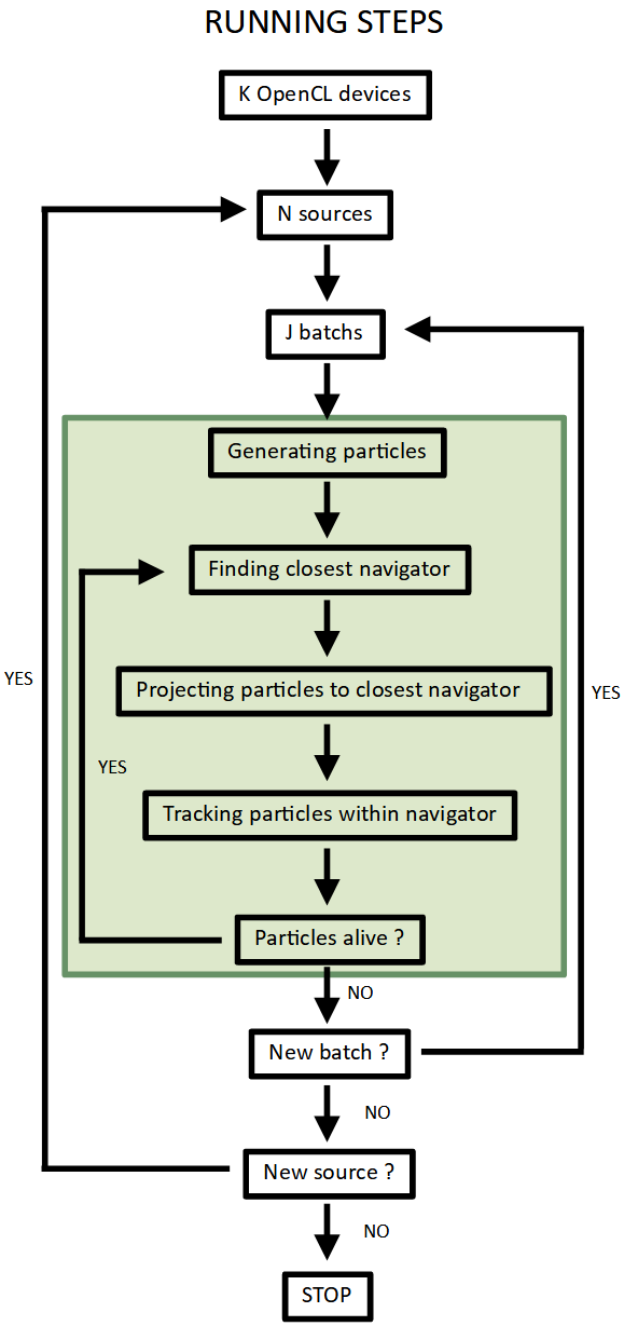
GGEMS DESIGN

The GGEMS has two important steps:

- Initialization
- Running

The following schemes summarize these steps.




RUNNING STEPS
K OpenCL devices
N sources
J batchs
Generating particles
Finding closest navigator
Projecting particles to closest navigator
Tracking particles within navigator
Particles alive ?
YES
YES
YES
NO
New batch ?
NO
New source ?
NO
STOP

CHAPTER

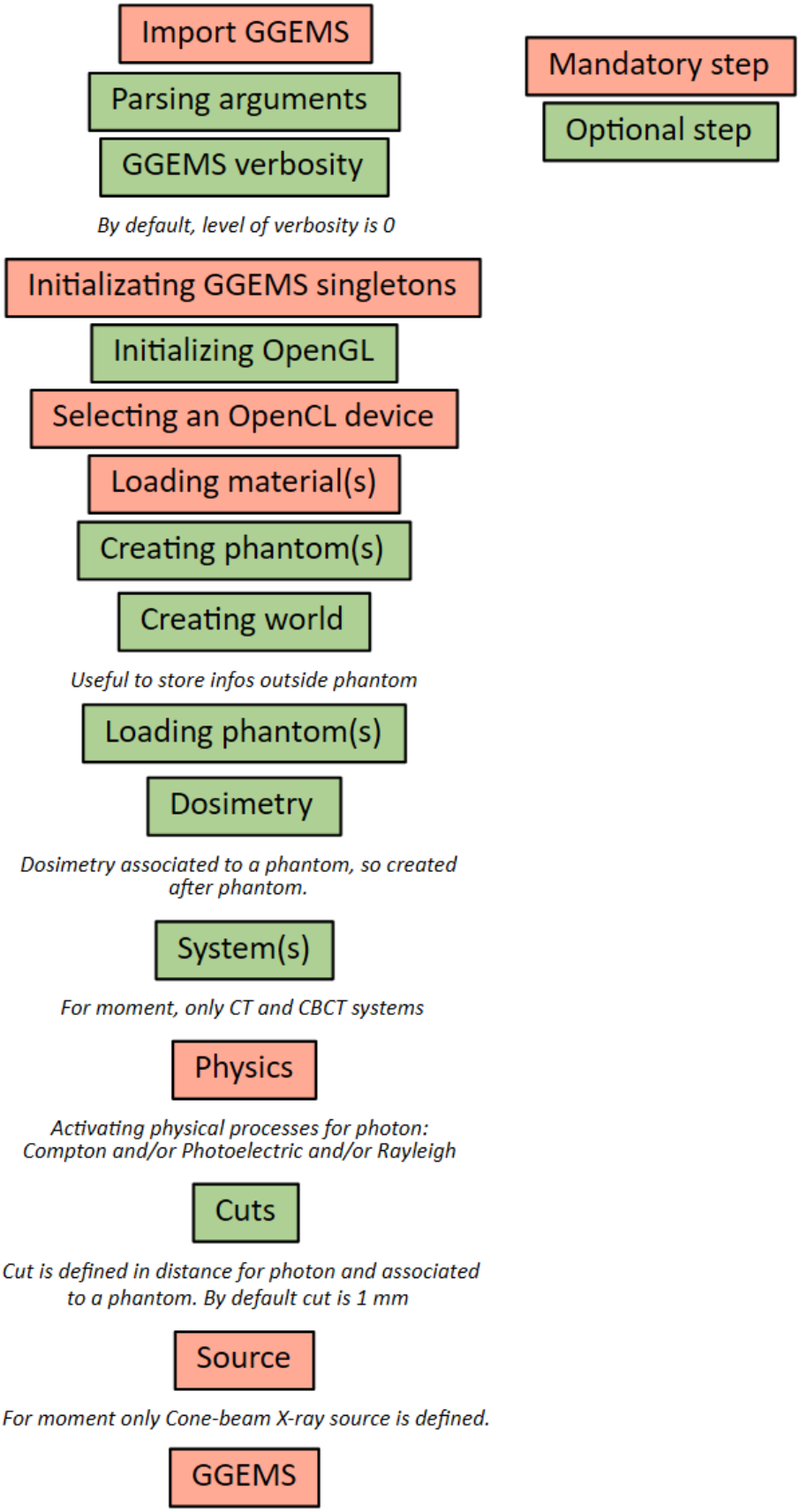
SIX

MAKE A GGEMS PROJECT

GGEMS is a C++ library with a Python interface.

6.1 Template

A GGEMS macro must be written following this model:


Import GGEMS
Parsing arguments
GGEMS verbosity
By default, level of verbosity is 0
Initializating GGEMS singletons
Initializing OpenGL
Selecting an OpenCL device
Loading material(s)
Creating phantom(s)
Creating world
Useful to store infos outside phantom
Loading phantom(s)
Dosimetry
Dosimetry associated to a phantom, so created after phantom.
System(s)
For moment, only CT and CBCT systems
Physics
Activating physical processes for photon: Compton and/or Photoelectric and/or Rayleigh
Cuts
Cut is defined in distance for photon and associated to a phantom. By default cut is 1 mm
Source
For moment only Cone-beam X-ray source is defined.
GGEMS
Mandatory step
Optional step

6.2 GGEMS macro using Python

Using GGEMS with python is simple. Here is a list of useful commands.

```
from ggems import *
```

Verbosity level is defined in the range [0;3]. For a silent GGEMS execution, the level is set to 0, otherwise 3 for lot of informations.

```
GGEMSVerbosity(0)
```

C++ singletons can be accessed via the following lines:

```
opencl_manager = GGEMSOpenCLManager()
opengl_manager = GGEMSOpenGLManager()
materials_database_manager = GGEMSMaterialsDatabaseManager()
processes_manager = GGEMSProcessesManager()
range_cuts_manager = GGEMSRangeCutsManager()
volume_creator_manager = GGEMSVolumeCreatorManager()
```

An OpenCL device could be selected.

```
opencl_manager.set_device_index(0)
```

A material database must be loaded in GGEMS. A material file is provided in GGEMS in 'data' folder. This file can be copy and paste in your project, and a new material can be added inside it.

```
materials_database_manager.set_materials('materials.txt')
```

Photon physical processes are activated using the process name, the particle name and the associated phantom name (or 'all' for all defined phantoms).

```
processes_manager.add_process('Compton', 'gamma', 'all')
processes_manager.add_process('Photoelectric', 'gamma', 'all')
processes_manager.add_process('Rayleigh', 'gamma', 'all')
```

Physical tables can be customized by changing the number of bins and the energy range. The following values are the default values.

```
processes_manager.set_cross_section_table_number_of_bins(220)
processes_manager.set_cross_section_table_energy_min(1.0, 'keV')
processes_manager.set_cross_section_table_energy_max(10.0, 'MeV')
```

Range cuts are defined in distance, particle type must be specified and cuts are associated to a phantom (or 'all' for all defined phantoms). The distance is converted in energy during the initialization step. During the simulation, if energy particle is below to a cut, the particle is killed and the energy is locally deposited.

```
range_cuts_manager.set_cut('gamma', 0.1, 'mm', 'all')
```

All verboses can be set to 'True' or 'False'. In 'tracking_verbose', the second parameters is the index of particle to track. The method 'initialize' intializes all objects in GGEMS, and the simulation starts with the method 'run'.

```
ggems = GGEMS()
ggems.opencl_verbose(True)
ggems.material_database_verbose(True)
```

(continues on next page)

(continued from previous page)

```
ggems.navigator_verbose(True)
ggems.source_verbose(True)
ggems.memory_verbose(True)
ggems.process_verbose(True)
ggems.range_cuts_verbose(True)
ggems.random_verbose(True)
ggems.profiling_verbose(True)
ggems.tracking_verbose(True, 0)

ggems.initialize(seed) #using a custom seed
# ggems.initialize()
ggems.run()
```

And finally exit GGEMS properly:

```
ggems.delete()
exit()
```

CHAPTER

SEVEN

RELEASE NOTES

7.1 Supported and Tested Platforms

GGEMS v1.3 has been validated only on 64 bits architecture.

Platforms:

Linux: Ubuntu 24.04 LTS

Windows: Windows 11

Compilers:

Linux: GNU v13.3, Clang v18.1.3

Windows: Visual C++ v19.44 and v19.50

OpenCL devices:

Intel

Xeon E5-2680, Xeon W-2245

HD Graphics 530

NVIDIA

Quadro P400, RTX 4000

OpenGL using:

GLFW v3.4

GLEW v2.1.0 or v2.2.0

GLM v1.0.2

7.2 What You Can Do in GGEMS

Applications:

CT and CBCT

Photon dosimetry

Sources:

Cone-beam source (energy can be defined as a spectrum, monoenergetic or discrete peak)

Voxelized source

Navigators:

Voxelized

Meshed

Physical processes:
Compton scattering

Rayleigh scattering

Photoelectric effect

Particles:
Photon

Output:
Raw file

MHD file

7.3 Compilation Warnings

There may be a few compilation warnings on some platforms, particularly on MacOS, where GGEMS has not been validated and tested.

7.4 GGEMS Software License

A Software License applies to the GGEMS code. Users must accept this license in order to use it. The details and the list of copyright holders is available at https://ggems.fr/about and also in the text file LICENSE distributed with the source code.

CHAPTER

EIGHT

CHANGE LOG

8.1 CMake

- Deleting CMake for C++ examples

8.2 GGEMS

- New kind of energy source, now a source can be defined using discrete peak energy
- Validation of GGEMS using OpenCL from CUDA 12.5 or 12.6
- Installation using setuptools
- New GGEMSMeshPhantom class for photon navigation through meshed navigator
- New GGEMSMeshedSolid and GGEMSMeshedSolidData class to store and handle geometric infos about meshed volume
- New GGEMSVoxelizedSource class

8.3 Fixed Bugs

- Bug in Rayleigh scattering process

8.4 Features

- Meshed navigator
- Voxelized source

8.5 Examples

- New example 7_Mesh declaring a meshed navigator
- New example 8_Voxelized_Source declaring a voxelized source associated to a voxelized phantom