GGEMS

version 1.1

GGEMS Collaboration

mai 12, 2021

Contents

Welcome to GGEMS Documentation

GGEMS is an advanced Monte Carlo simulation platform using CPU and GPU architecture targeting medical applications (imaging and particle therapy). This code is based on the well-validated Geant4 physics model and capable to be executed in both CPU and GPU devices using the OpenCL library.

This documentation is divided into three parts.

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

Second, for a standard user, informations about all GGEMS potential are given.Examples and tools are also illustrated and explained. And all the command lines are listed using both C++ and python instructions.

And finally, in the last part of this documentation, a more detailed description concerning GGEMS core for advanced user. The purpose of this part is to give enough informations to an user to implement a custom part of code in GGEMS.

Introduction

GGEMS (GPU Geant4-based Monte Carlo Simulations) is an advanced Monte Carlo simulation platform using the OpenCL library managing CPU and GPU architecture. GGEMS is written in C++, and can be used using python commands. The reader is assumed to have some basic knowledge of object-oriented programming using C++.

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 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
- Navigation in simple box volume or voxelized volume
- Flat or curved detector for CBCT/CT application

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:

- Visualization
- Positron particle tracking
- Electron particle tracking
- Mesh volume
- Voxelized source
- PET imaging
- SPECT imaging
- Intra-operative radiotherapy (brachytherapy and intrabeam)

- AMD architecture validation
- MacOS system validation

Requirements

GGEMS is a multiplatform application using OpenCL

Operating Systems

Supported

- Linux (Any distribution, Debian, Ubuntu, …)
- Windows 10

Maybe Supported

GGEMS should work on a MacOS system, but this has not been tested

- MacOS X

Python Version

GGEMS supports the following version

- Python 3.6+

OpenCL Version

GGEMS validated using the following version

- OpenCL 1.2

Hardware

GGEMS can be used on lot of different hardwares such as CPU, GPU and graphic cards included in CPU (Intel HD Graphics)

Supported

- Intel (CPU + HD Graphics)
- NVIDIA

Maybe Supported

GGEMS should work on the following hardware, but not tested yet

- AMD

Building & Installing

> ***Note***
>
> GGEMS is written in C++ and using the OpenCL C++ API. However, the most useful GGEMS functions have been wrapped to be called in python version 3. Python is not mandatory, GGEMS can be used only in C++ too. Lot of C++ and python examples are given in this manual.

Prerequisites

GGEMS code is based on the OpenCL library. For each platform (NVIDIA, Intel, AMD) in your computer, you have to install a specific driver provided by the vendor.

NVIDIA

Linux & Windows

CUDA and NVIDIA driver have to be installed if you want to use GGEMS on a NVIDIA architecture. The easiest way to install OpenCL on NVIDIA platform is to download CUDA and NVIDIA driver in the same time from the following link: https://developer.nvidia.com/cuda-downloads.

GGEMS has been tested on the lastest CUDA and NVIDIA versions, and also some older versions.

Warning

CUDA is not used in GGEMS, but the OpenCL library file in included in CUDA folder.

Warning

It is recommanded to install CUDA and the NVIDIA driver directly from the NVIDIA website. Using packaging tool (as apt) is very convenient but can produce some troubles during GGEMS executation.

INTEL

Linux & Windows

Using GGEMS on Intel architecture require Intel driver. More information about Intel SDK for OpenCL applications can be find in the following link: https://software.intel.com/content/www/us/en/develop/tools/opencl-sdk.html. Both drivers for Linux and Windows can be downloaded here: https://software.intel.com/content/www/us/en/develop/articles/opencl-drivers.html

AMD

Linux & Windows

AMD platform has not be tested, but surely with few modifications GGEMS will run on a AMD platform. The correct driver for CPU and/or GPU should be available on the following link: https://www.amd.com/en/support. Don't hesitate to contact the GGEMS team if you need help for AMD implementation. For the next releases, AMD platform will be tested and validated.

Important

All previous drivers have to be installed before to install GGEMS. Install NVIDIA driver before Intel driver is recommanded, if using GGEMS on both architecture is required.

GGEMS Installation

CMAKE is required to install GGEMS. The minimal CMAKE version is 3.8.

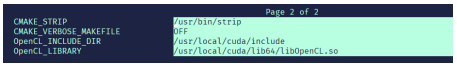
Linux

Intel and/or NVIDIA drivers are supposed to be installed. Download GGEMS from the website, or from a terminal:

```
$ wget https://ggems.fr/download/ggems_v1.0.zip
```

Unzip the downloaded file, create a folder named GGEMS_build (or another name), and launch the command 'ccmake'. Create an install folder is recommanded.

```
$ unzip ggems_v1.0.zip
$ mkdir GGEMS_build
$ mkdir GGEMS_install
$ cd GGEMS_build
$ ccmake ../GGEMS
```



Note

By default, the GNU compiler is used on Linux. CLANG can also be used. The compiler can be modify in the CMakeLists.txt file and empty the CMAKE cache.

In the previous images 'YOUR_PATH' is automatically found by CMAKE. CMAKE parameters shown previously are explained here. Last step, compile and install GGEMS.

```
$ make -jN
$ make install
```

Final step, configuration of your environment. There are many ways to do that. Since GGEMS is a library, you have to indicate its location in your environment file (.bashrc for instance).

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:YOUR_PATH/GGEMS_install/ggems/lib
```

To load permanently the GGEMS python module, we also recommand to add the following line in your environment file

```
export PYTHONPATH=$PYTHONPATH:YOUR_PATH/GGEMS_install/ggems/python_module
export PYTHONPATH=$PYTHONPATH:YOUR_PATH/GGEMS_install/ggems/lib
```

GGEMS is now installed on your system. To test the installation, try to launch GGEMS examples or load the GGEMS python module from python console.

```
from ggems import *
opencl_manager.print_infos()
opencl_manager.clean()
exit()
```

Windows

Note

The following installation process for Windows is done using the classical Windows command prompt. Visual Studio is the compiler by default. CLANG can be selected by modifying the CMakeLists.txt file.

> ***Important***
>
> Only Visual Studio (CL) and CLANG are validated on Windows. GNU GCC is not recommanded.

Visual Studio is assumed well configured. The command ‘cl.exe’ should be recognize in your command prompt. If not, there some useful commands to configure Visual Studio 2019 in a batch script file (named ‘set_compilers.bat’ for instance). If a previous version of Visual Studio code is installed on your computed, you might modify this script.

```
@echo OFF
if "%VCTOOLKIT_VARS_ARE_SET%" == "true" goto done

echo --- Setting Microsoft Visual C++ Toolkit 2019 environment variables... ---

call "C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Auxiliary\Build\vcvar

set PATH="%VCToolkitInstallDir%"\bin;%PATH%
set INCLUDE="%VCToolkitInstallDir%"\include;%INCLUDE%
set LIB="%VCToolkitInstallDir%"\lib;%LIB%

set VCTOOLKIT_VARS_ARE_SET=true
echo Done.
:done
```

Calling the previous script typing the following command:

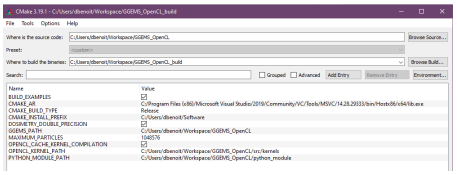
```
C:\Users\XXX> set_compilers.bat
```

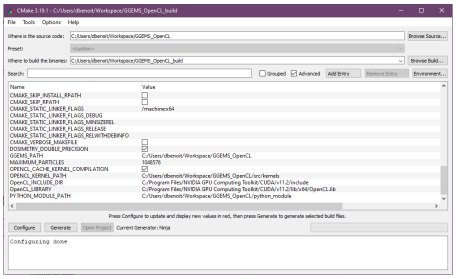
Download GGEMS from the website. Unzip GGEMS in your environment folder (for instance C:\Users\XXX). Then create both GGEMS_build and GGEMS_install folder.

```
C:\Users\XXX> md GGEMS_build
C:\Users\XXX> md GGEMS_install
```

Go to the build folder and launch CMAKE.

```
C:\Users\XXX> cd GGEMS_build
C:\Users\XXX\GGEMS_build> cmake-gui
```





> ***Note***
>
> For multithreaded compilation it is recommanded to use Ninja generator and not nmake. Ninja can be install as a package during visual studio installation or directly from here https://ninja-build.org/

> ***Important***
>
> Order during installation of NVIDIA and Intel driver could be important. To check that, go to your environment variables in PATH variable and check which OpenCL library is call first. NVIDIA OpenCL library should be called first.

Final step, compilation and installation using nmake or ninja.

```
C:\Users\XXX\GGEMS_build> nmake install
```

or

```
C:\Users\XXX\GGEMS_build> ninja install
```

GGEMS have to be set in your environment variables by creating (or add an entry) a PYTHONPATH variable and add an entry to the PATH variable. The following batch script can do that for you in the command prompt.

```
@echo OFF
if "%GGEMS_VARS_ARE_SET%" == "true" goto done

echo --- Setting GGEMS... ---
set PYTHONPATH=%PYTHONPATH%;C:\Users\XXX\GGEMS_install\ggems\python_module
set PYTHONPATH=%PYTHONPATH%;C:\Users\XXX\GGEMS_install\ggems\lib
set PATH=%PATH%;C:\Users\XXX\GGEMS_install\ggems\lib

set GGEMS_VARS_ARE_SET=true
echo Done.
:done
```

GGEMS is now installed on your system. To test the installation, try to launch GGEMS examples or load the GGEMS python module from python console.

```
from ggems import *
opencl_manager.print_infos()
opencl_manager.clean()
exit()
```

CMAKE Parameters

BUILD_EXAMPLES

By default this option is set to ON. During the installation all C++ executables are copied to the respective example folder.

CMAKE_INSTALL_PREFIX

Path to your installation folder

DOSIMETRY_DOUBLE_PRECISION

By default this option is set to ON. For dosimetry the computation are in double float precision.

GGEMS_PATH

Path found automatically by CMAKE. It corresponds to GGEMS source folder.

MAXIMUM_PARTICLES

By default the batch of maximum particles simulated by GGEMS is 1048576. This number can be higher depending on your graphic cards.

OPENCL_CACHE_KERNEL_COMPILATION

By default this option is set to ON. It means the compiled OpenCL kernels are stored in the cache folder during the compilation process. It's recommanded to set this option to OFF, if you want modify code inside an OpenCL kernel and delete the OpenCL cache too.

OPENCL_KERNEL_PATH

Path to GGEMS OpenCL kernels. This path is automatically found by CMAKE.

PYTHON_MODULE_PATH

Path to GGEMS python module. This path is automatically found by CMAKE.

Getting Started

GGEMS can be called using a python console

```
$ python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win
Type "help", "copyright", "credits" or "license" for more information
>>> from ggems import *
>>> opencl_manager = GGEMSOpenCLManager()
>>> opencl_manager.print_infos()
>>> opencl_manager.set_device_index(0)
>>> opencl_manager.clean()
>>> exit()
```

With the previous command lines, the user has the possibility checking which device is recognized by GGEMS. The device 0 is selected.

> ***Important***
>
> If an OpenCL device is missing, please check your installation driver for the missing device.

The best way learning how GGEMS is working, is to try each example available in the example folders. Using GGEMS, for a personnal project, from scratch, using python or C++ is explained in the developer part.

Multi-Device

GGEMS can be used on multi-devices using OpenCL library. Two different ways are implemented to activate device:

- Device can be selected using device index

```
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
```

- Device can be selected using a string for a list of devices

```
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('gpu', 'amd') # Activate all AMD GPU 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
```

Navigators

The particles are only tracked in navigator volumes. There are two types of navigators in GGEMS:

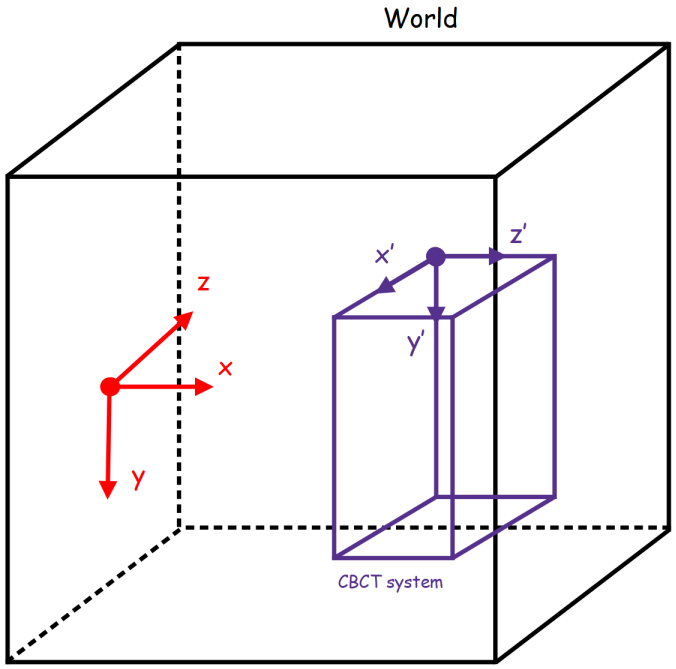
- System (detector)
- Phantom (object, patient)

For each navigator, three elements are associated:

- a solid: geometry of the navigator
- a list of materials
- physical processes

Systems

In GGEMS, only CT/CBCT system is available for moment. This detector is composed by pixels arranged in modules. The following figure shows the reference axis of CT/CBCT system.



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

```
cbct_system = GGEMSCTSystem('detector')
```

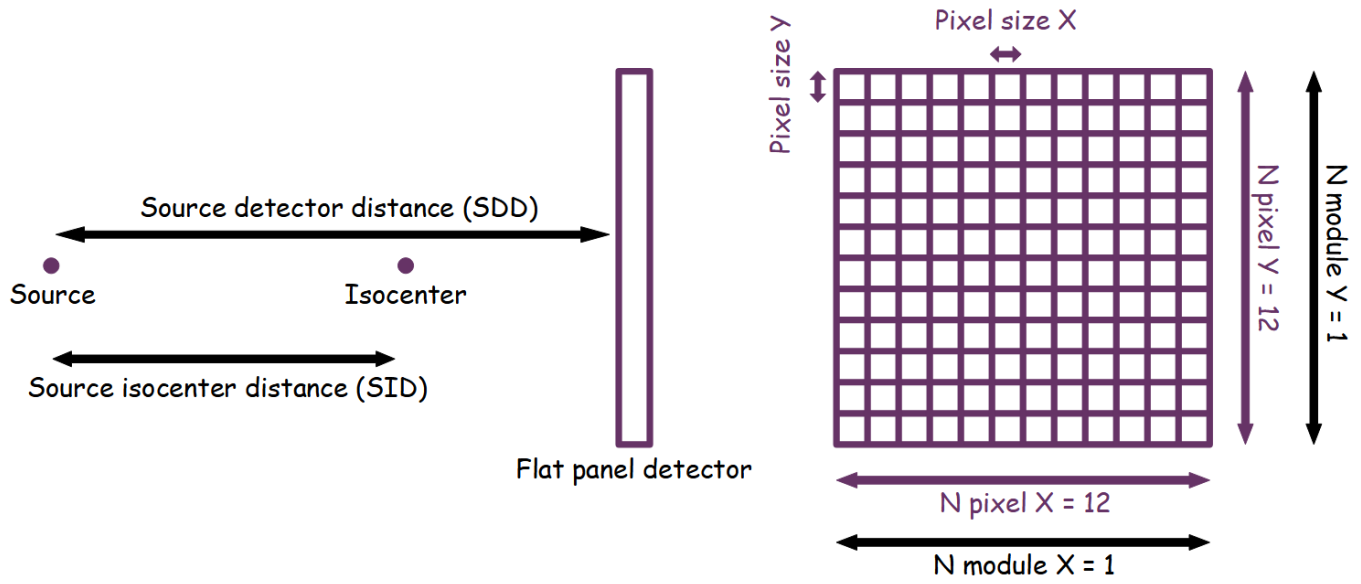
Types of CT/CBCT detector are:

- flat panel
- curved

Flat panel

This type of geometry is well adapted for CBCT configuration.

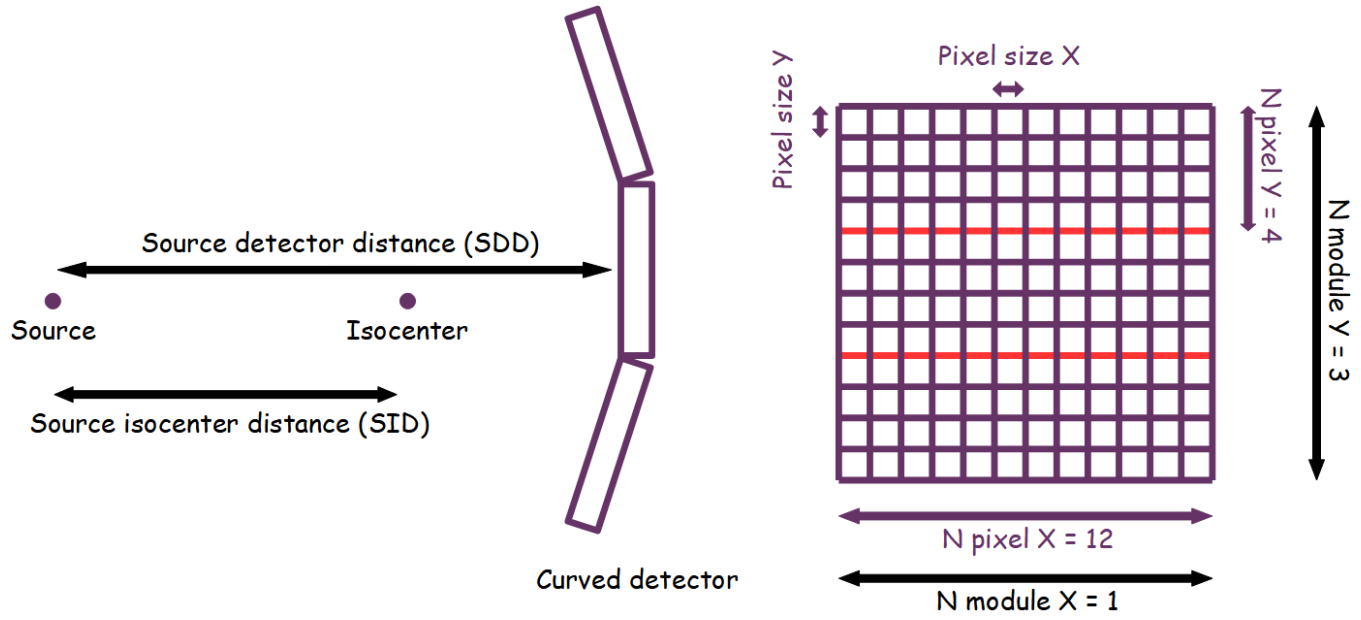
```
cbct_system.set_ct_type('flat')
```



Curved

This type of geometry is well adapted for CT configuration.

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



Note

For curved geometry, the angle between modules is automatically computed. The center of rotation is the source position, and there is no gap between modules.

For each type of detector, number of modules, number of detection elements within module and their 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')
```

Detector can be composed by one type of material:

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

And a threshold can be applyied specifically to the detector:

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

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

```
cbct_system.set_source_detector_distance(1085.6, 'mm')
cbct_system.set_source_isocenter_distance(595.0, 'mm')
```

The CT/CBCT system can be rotated around world axis as following:

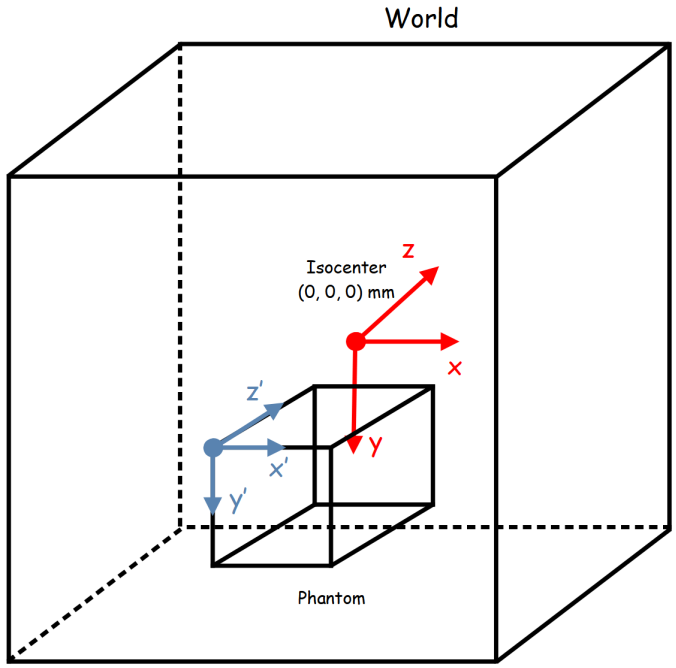
```
cbct_system.set_rotation(0.0, 0.0, 0.0, 'deg')
```

Final projection is saved in a MHD file and scatter registration can be activated:

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

Phantoms

In GGEMS, a voxelized phantom is the only available type of phantom. This phantom is defined in a mhd file and a range data file, storing all label data. The reference axis of the phantom are the same than the global world volume



First a phantom is created by choosing a name, and loading a mhd file and a range data file:

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

The user can set a position and a rotation applyied to the phantom using the following commands:

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

World

Outside navigator, particle are not tracked. However, a tool has been developped in GGEMSWorld class storing particle data (photon tracking, energy/energy squared voxel in world, and momentum) outside navigator. Particles are projected in GGEMSWorld using a DDA algorithm.

The world module is ‘GGEMSWorld’:

```
world = GGEMSWorld()
```

After creating the GGEMSWorld object, the dimension of the world and size of voxel can be set:

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

For world output, there are many informations the user can save such as: energy and energy squared of photon crossing 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)
```

Dosimetry

During GGEMS simulation, a photon dosimetry module can be activated to compute absorbed dose in a specific phantom.

> ***Note***
>
> Only photon are simulated in the current version of GGEMS. In next releases electron will be implemented.

The dosimetry module is 'GGEMSDosimetryCalculator':

```
dosimetry = GGEMSDosimetryCalculator()
```

After creating the GGEMSDosimetryCalculator object, a navigator is attached:

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

The size of voxel in dosimetry image (dosel) can be set. If not set the dosel size is the same than voxel phantom size:

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

The absorbed dose is computed in gray (Gy). By default the dose in computed using materials in phantom. Otherwize the user can set water material everywhere in phantom.

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

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

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

For dose output, there are many informations the user can save such as: uncertainty value of the dose, the deposited energy in dosel, the squared of deposited energy in dosel and the number of interaction (hit) in dosel:

```
dosimetry.set_output('data/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)
```

Physical Processes & Range cuts

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 information about physics, please read the documentation on the Geant4 website.

By using python, the variable ‘processes_manager’ can be called to manage processes.

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, and the fluorescence photon is not emitted.

Compton Scattering

The Geant4 model extracted is the ‘G4KleinNishinaCompton’ standard model. It is the fastest algorithm to simulate this process. Compton scattering is activated for all the navigators, or for a specific navigator.

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

In the previous line, Compton scattering is activated for all the navigators.

```
processes_manager.add_process('Compton', 'gamma', 'my_phantom')
```

In the previous line, Compton scattering is activated only for a navigator named ‘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.

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

In the previous line, Photoelectric effect is activated for all the navigators.

```
processes_manager.add_process('Photoelectric', 'gamma', 'my_phantom')
```

In the previous line, Photoelectric effect is activated only for a navigator named ‘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.

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

In the previous line, Rayleigh scattering is activated for all the navigators.

```
processes_manager.add_process('Rayleigh', 'gamma', 'my_phantom')
```

In the previous line, Rayleigh scattering is activated only for a navigator named ‘my_phantom’

Process Parameters Building

The cross-sections are computed during the GGEMS initialization step. The parameters used for the cross-sections building can be customized by the user, however it is recommanded to use the default parameters. The customizable parameters are:

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

The 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

Informations about processes can be printed by GGEMS:

- Available processes
- Global informations about processes
- Cross-section value in tables

The list of commands are:

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

Range Cuts

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

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

In the previous line, cuts are activated for photon for all navigators.

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

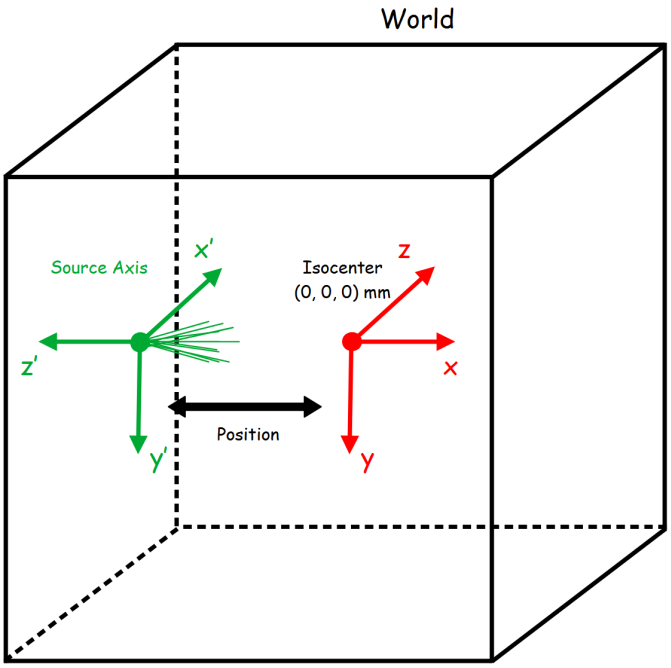
In the previous line, cuts are activated for photon for a navigator named ‘my_phantom’.

Sources

The source strategy in GGEMS is to develop a optimized source for each application. For moment, only CT/CBCT application is developed so the source type available is a cone-beam X-ray source.

X-ray Source

X-ray source is defined as a cone-beam geometry. The direction of the generated particles point always to the center of the world. This source has its own axis as defined in the image below:



Some commands are provided managing a X-ray source.

First, the user has to create source by choosing a name:

```
xray_source = GGEMSXRaySource('xray_source')
```

The particle type is only photon and can be selected with the following command:

```
xray_source.set_source_particle_type('gamma')
```

The number of generated particles during the run is defined by the user:

```
xray_source.set_number_of_particles(1000000000)
```

The position and rotation of the source are defined in the global world reference axis and the cone-beam source is defined with an aperture angle.

```
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')
```

A X-ray source is defined with a focal spot size. If defined at (0, 0, 0) mm, it is similar to a point source, otherwize it is a more realistic X-ray source with a small rectangular surface defined in source axis reference:

```
xray_source.set_focal_spot_size(0.0, 0.0, 0.0, 'mm')
```

Important

The focal spot size is defined in source axis reference and not in global world reference!!!

The energy source can be defined using a single energy value or a spectrum included in a text file.

```
xray_source.set_polyenergy('data/spectrum_120kVp_2mmAl.dat')
# OR
xray_source.set_monoenergy(25.0, 'keV')
```

GGEMS Commands

The main steps in GGEMS are initialize and run methods.

```
ggems = GGEMS()
ggems.initialize()
ggems.run()
```

Different useful information output are available for the user during GGEMS executions.

To print all informations about OpenCL device:

```
ggems.opencl_verbose(True)
```

To print all informations about material database:

```
ggems.material_database_verbose(True)
```

To print all information about navigator (system + phantom):

```
ggems.navigator_verbose(True)
```

To print all informations about source:

```
ggems.source_verbose(True)
```

To print all informations about allocated memory:

```
ggems.memory_verbose(True)
```

To print all informations about activated processes:

```
ggems.process_verbose(True)
```

To print all informations about range cuts:

```
ggems.range_cuts_verbose(True)
```

To print seed and state of the random:

```
ggems.random_verbose(True)
```

To print profiler data (elapsed time in OpenCL kernels):

```
ggems.profiling_verbose(True)
```

To print tracking informations about a specific particle index:

```
ggems.tracking_verbose(True, 12)
```

Cleaning GGEMS object

```
ggems.delete()
```

Examples & Tools

A list of examples and tools are provided for GGEMS users. Only python instructions are given. For C++, a CMakeLists.txt file is mandatory for compilation.

> ***Note***
>
> Examples are compiled and installed when the compilation option 'BUILD_EXAMPLES' is set to ON. C++ executables are installed in example folders.

Examples 0: Cross-Section Computation

The purpose of this example is to provide a tool computing cross-section for a specific material and a specific photon physical process. The energy (in MeV) and the OpenCL device are set by the user.

```
$ python cross_sections.py [-h] [-d DEVICE] [-m MATERIAL] -p [PROCESS]-e [ENERGY] [-v VERBOS
-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
```

The macro is in the file 'cross_section.py'.

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)
opencl_manager.set_device_index(device_id)
```

GGEMSMaterial object is created, and each new material can be added. 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()
```

Before using a physical process, GGEMSCrossSection object is created. Then each process can be added individually. And finally cross sections are computing by giving the list of materials.

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

Getting the cross section value (in cm2.g-1) for a specific energy (in MeV) is done by the following command:

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

Examples 1: Total Attenuation

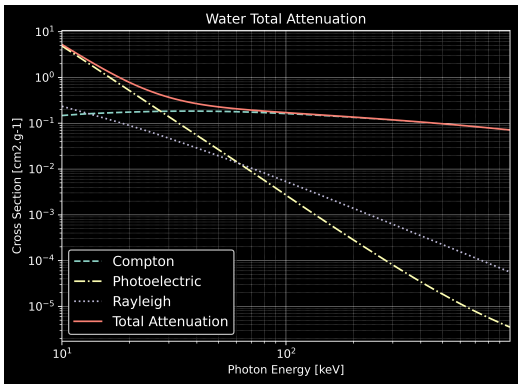
Warning

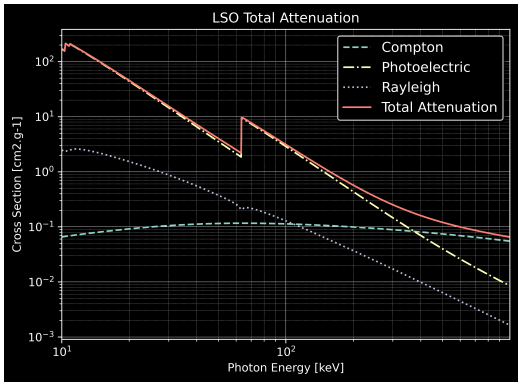
This example is only available using python and the matplotlib library is mandatory.

This example is a tool for plotting the total attenuation of a material for energy between 0.01 MeV and 1 MeV. The commands are similar to example 0, and all physical processes are activated.

```
$ 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:





Examples 2: CT Scanner

In this CT scanner example, a water box is simulated associated to a CT curved detector. Only 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
```

The water box phantom is loaded:

```
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')
```

Then CT curved detector is built:

```
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)
```

Initialization of 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:

Device	Computation Time [s]
GeForce GTX 1050 Ti	128
GeForce GTX 980 Ti	52
Quadro P400	404
Xeon X-2245 8 cores / 16 threads	132

Examples 3: Voxelized Phantom Generator

A tool creating voxelized phantom is provided by GGEMS. Only basic shapes are available such as tube, box and sphere. The output format is MHD, and the range material data file is created in same time than the voxelized volume.

```
$ 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
```

First step is to create global volume storing all other voxelized objets. Dimension, voxel size, name of output volume, format data type and material are defined.

```
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()
```

Then a voxelized volume can be drawn in the global volume. A box object is built with the command lines below:

```
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()
```

Examples 4: Dosimetry

In dosimetry example, a cylinder is simulated computing absorbed dose inside it. Different results such as dose, energy deposited... can be plotted. An external source, using GGEMS X-ray source is simulated generating 2e8 particles.

```
$ python dosimetry_photon.py [-h] [-d DEVICE] [-b BALANCE] [-n N_PARTICLES] [-s SEED] [-v VE
-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
```

First, the cylinder phantom is loaded:

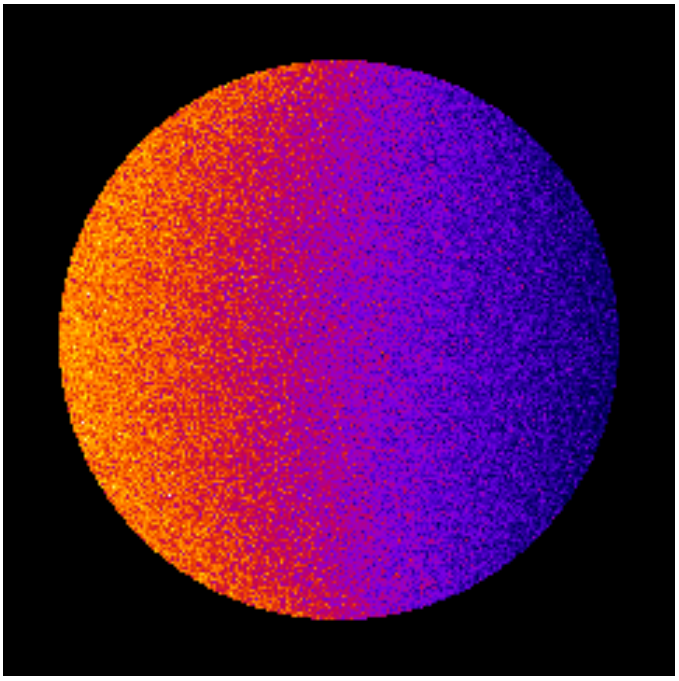
```
phantom = GGEMSVoxelizedPhantom('phantom')
phantom.set_phantom('data/phantom.mhd', 'data/range_phantom.txt')
phantom.set_rotation(0.0, 0.0, 0.0, 'deg')
phantom.set_position(0.0, 0.0, 0.0, 'mm')
```

Then dosimetry object is associated to the previous phantom, storing all data during particle tracking:

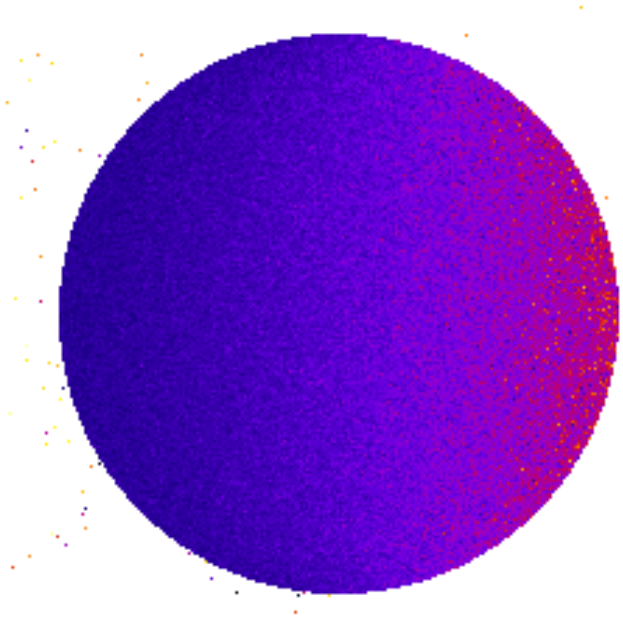
```
dosimetry = GGEMSDosimetryCalculator('phantom')
dosimetry.set_output('data/dosimetry')
dosimetry.set_dosel_size(0.5, 0.5, 0.5, 'mm')
dosimetry.water_reference(False)
dosimetry.minimum_density(0.1, 'g/cm3')
dosimetry.uncertainty(True)
dosimetry.photon_tracking(True)
dosimetry.edep(True)
dosimetry.hit(True)
dosimetry.edep_squared(True)
```

And finally an external source using GGEMSXRaySource is created:

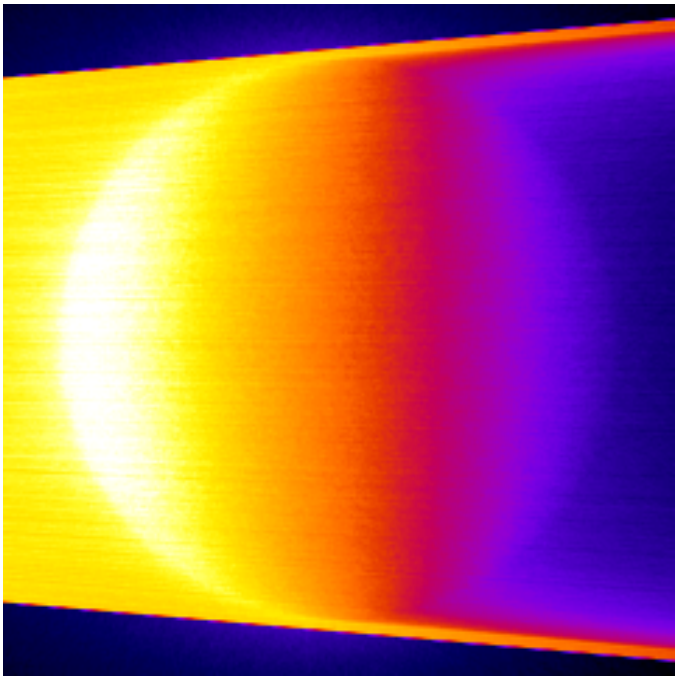
```
point_source = GGEMSXRaySource('point_source')
point_source.set_source_particle_type('gamma')
point_source.set_number_of_particles(200000000)
point_source.set_position(-595.0, 0.0, 0.0, 'mm')
point_source.set_rotation(0.0, 0.0, 0.0, 'deg')
point_source.set_beam_aperture(5.0, 'deg')
point_source.set_focal_spot_size(0.0,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



Dose absorbed by cylinder phantom



Uncertainty dose computation



Photon tracking in phantom

Performance:

Device	Computation Time [s]
GeForce GTX 1050 Ti	253
GeForce GTX 980 Ti	65
Quadro P400	1228
Xeon X-2245 8 cores / 16 threads	570

Examples 5: World Tracking

In world tracking example, a cylinder is simulated computing absorbed dose inside it, a CBCT flat panel detector is also defined storing photon count. And finally a world is defined 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 VERB
-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
```

First the world is defined, and all output are generated

```
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)
```

The created cylinder phantom is loaded and dosimetry module is associated to the cylinder phantom, and all output are activated

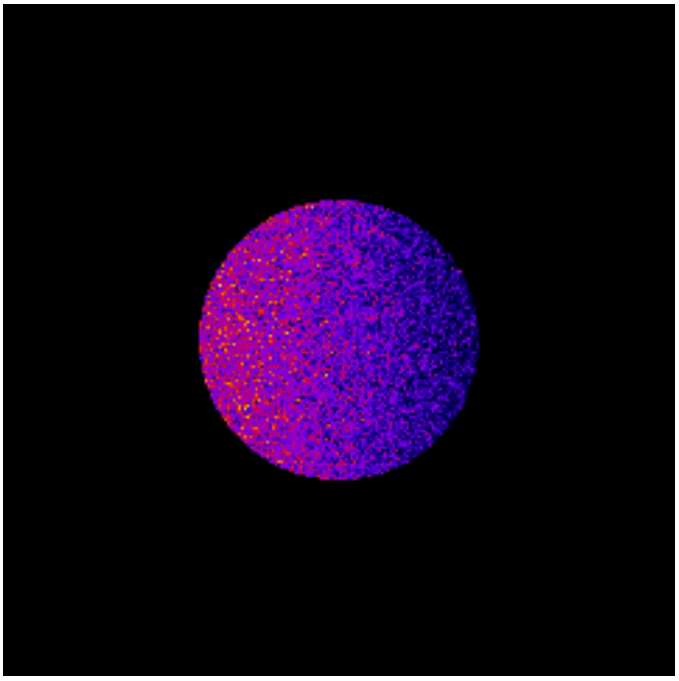
```
phantom = GGEMSVoxelizedPhantom('phantom')
phantom.set_phantom('data/phantom.mhd', 'data/range_phantom.txt')
phantom.set_rotation(0.0, 0.0, 0.0, 'deg')
phantom.set_position(0.0, 0.0, 0.0, 'mm')

dosimetry = 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 is built

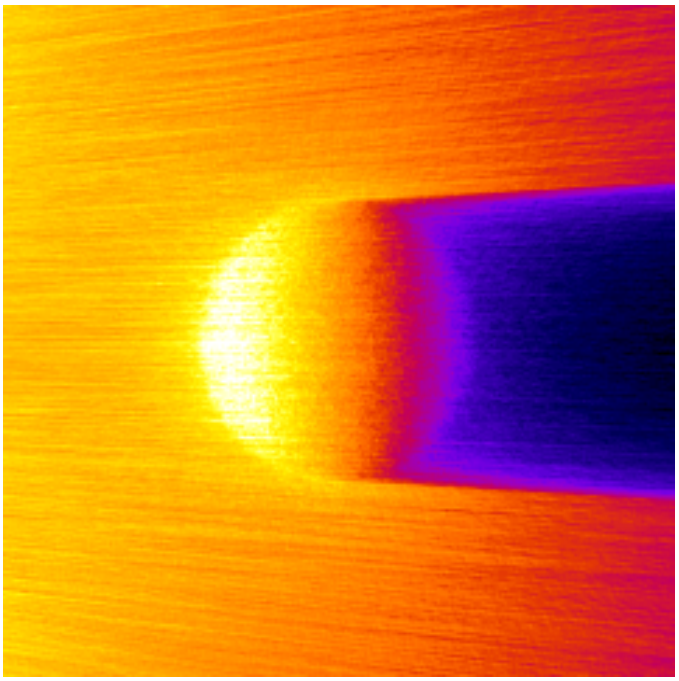
```
cbct_detector = GGEMSCTSystem('custom')
cbct_detector.set_ct_type('flat')
cbct_detector.set_number_of_modules(1, 1)
cbct_detector.set_number_of_detection_elements(400, 400, 1)
cbct_detector.set_size_of_detection_elements(1.0, 1.0, 10.0, 'mm')
cbct_detector.set_material('Silicon')
cbct_detector.set_source_detector_distance(1500.0, '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')
```

And finally an external source using GGEMSXRaySource is created:

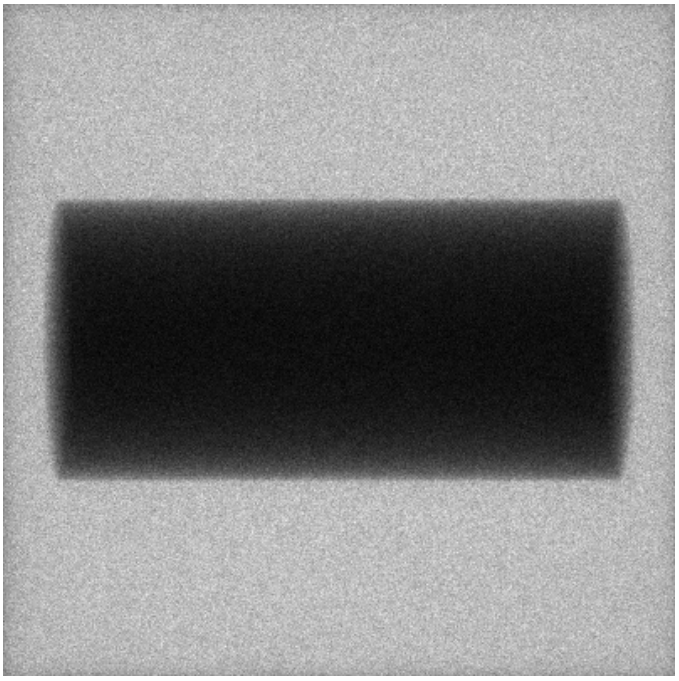
```
point_source = GGEMSXRaySource('point_source')
point_source.set_source_particle_type('gamma')
point_source.set_number_of_particles(100000000)
point_source.set_position(-900.0, 0.0, 0.0, 'mm')
point_source.set_rotation(0.0, 0.0, 0.0, 'deg')
point_source.set_beam_aperture(12.0, 'deg')
point_source.set_focal_spot_size(0.0,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



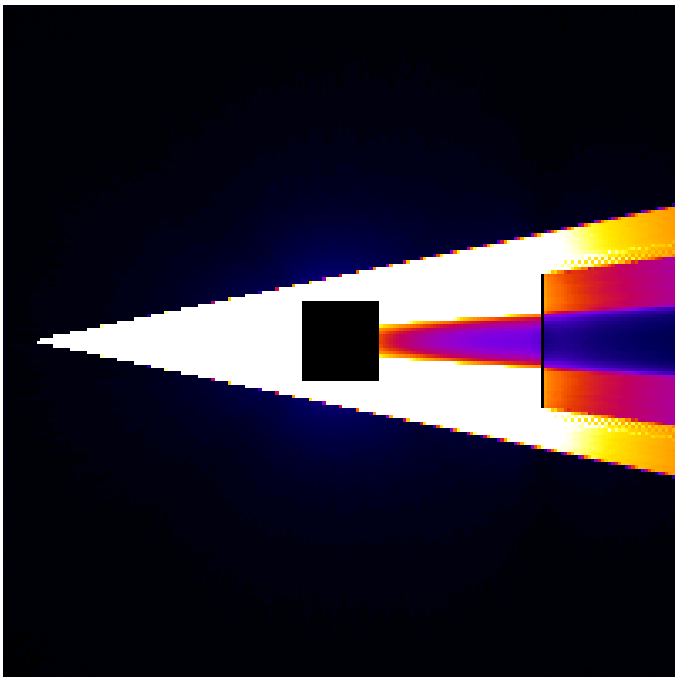
Dose absorbed by cylinder phantom



Photon tracking in phantom



Cylinder projection on flat panel detector



World photon tracking

Performance:

Device	Computation Time [s]
GeForce GTX 1050 Ti	363
GeForce GTX 980 Ti	390
Quadro P400	908
Xeon X-2245 8 cores / 16 threads	284

Release Notes

Supported and Tested Platforms

GGEMS has been tested only on 64 bits architecture.

Platforms:

- Linux: gcc-9.3, clang-11
- Windows: Visual C++ 19.28 (Visual Studio 2019), clang-11

More verified and tested configurations:

- Linux: gcc-7.5, clang-9/10
- Windows: clang-9/10

OpenCL devices:

- **Intel**
  - Xeon E5-2680, Xeon W-2245
  - HD Graphics 530
- **NVIDIA**
  - Quadro P400, P2000
  - GeForce GTX 980 Ti, 1050 Ti, 1080 Ti

What You Can Do in GGEMS

Applications:

- CT and CBCT systems
- Photon dosimetry

Sources:

- X-ray source using spectrum
- Point source and rectangular source derived for X-ray source

Volume:

- Voxelized phantom

Physical processes:

- Compton scattering
- Rayleigh scattering
- Photoelectric effect

Particles:

- Photon

Output:

- Raw file
- MHD file

Compilation Warnings

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

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.

Change log

CMAKE

- Example are now installed in GGEMS install path

C++

- Smart pointers removed and replaced by classic ‘new’ and ‘delete’ methods.
- For Windows user, options application can be handled by methods defined in GGEMSWinGetOpt.

GGEMS

- New classes GGEMSProfilerManager, GGEMSProfiler and GGEMSProfilerItem can be used to display details about elapsed time in OpenCL kernels.
- GGEMS can be run on multi-devices GPU and/or CPU.
- A new method in C++ and python handles the balance computation between each device.
- In GGEMSOpenCLManager, ‘clean’ method cleans all GGEMS C++ singletons.
- MHD file suffix is checked at the beginning of simulation.
- New OpenCL kernel ‘is_alive’ checks if particles are alive after each batch
- Problem reading material file on Linux windows is fixed
- C++ Singleton GGEMSManager is deleted and replaced by GGEMS class
- A security has been added to prevent infinite loop during tracking

Features

- For CT application, scatter histogram can be saved.
- New class GGEMSWorld stores data (fluence (photon tracking), energy deposit, energy deposite squared and momentum) outside navigator (phantom and detector).

Examples

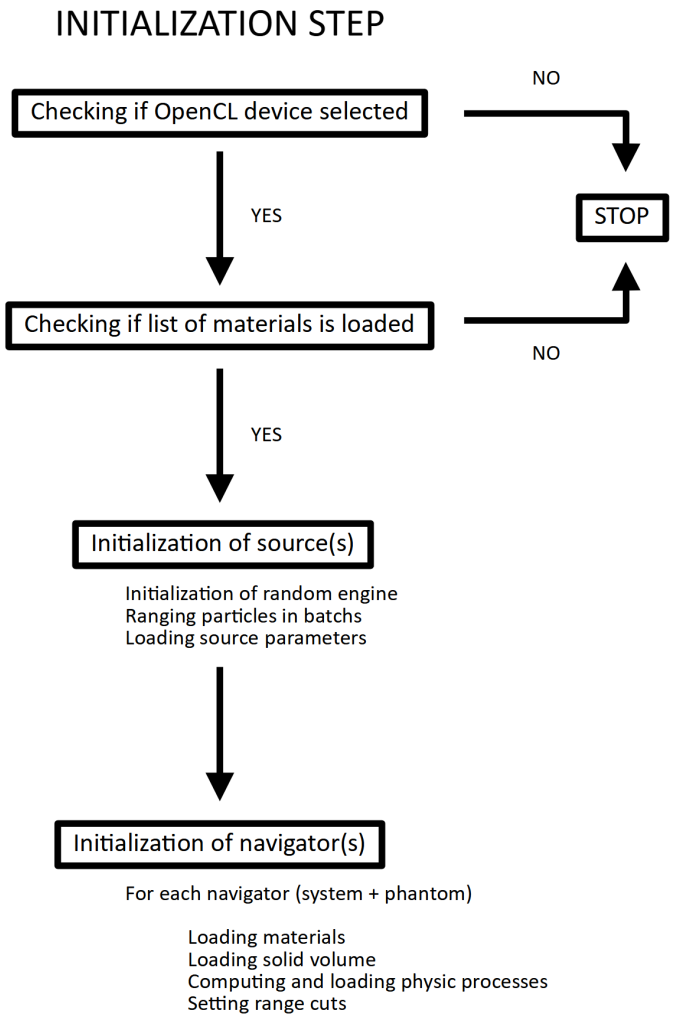
- New example 5_World_Tracking illustrating new GGEMSWorld feature

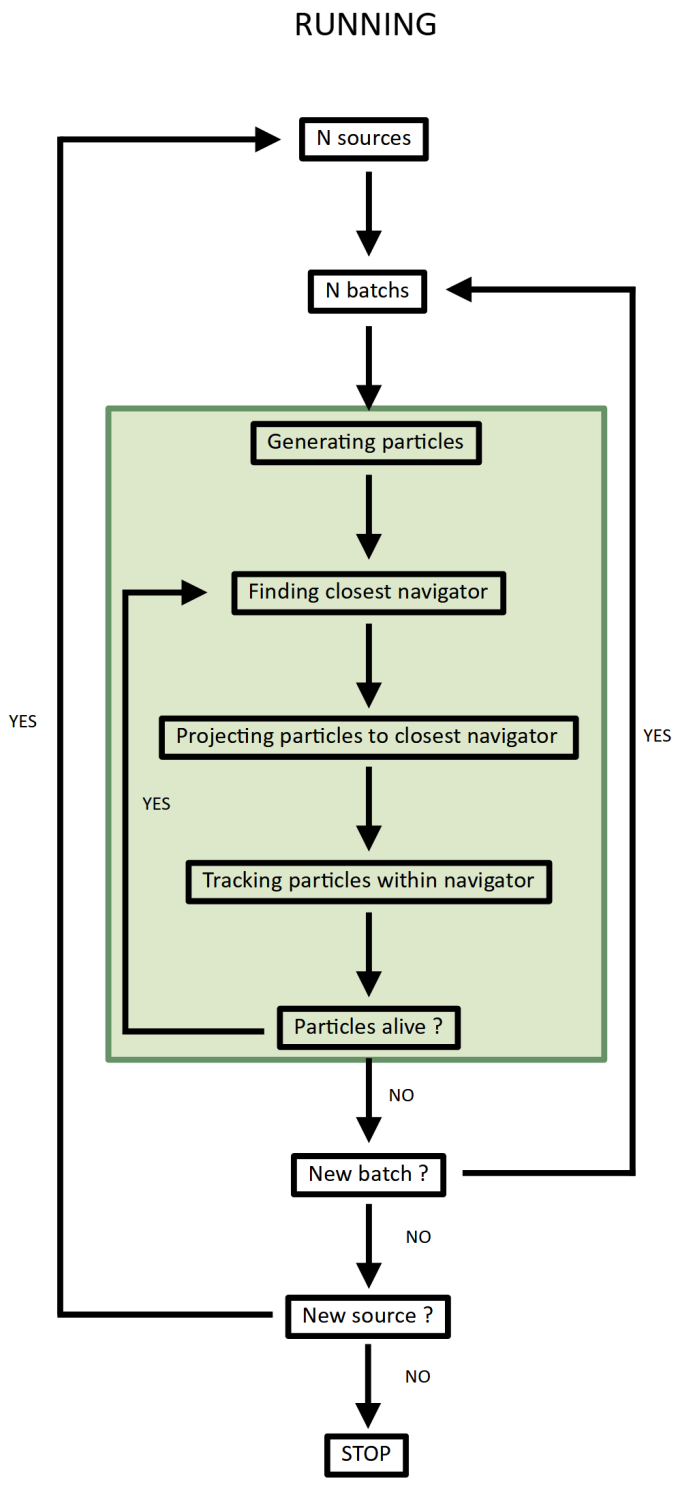
GGEMS Design

The GGEMSManager class has two important steps:

- Initialization
- Running

The following schemes summarize these steps.


INITIALIZATION STEP
Checking if OpenCL device selected
NO
STOP
YES
Checking if list of materials is loaded
NO
YES
Initialization of source(s)
Initialization of random engine
Ranging particles in batchs
Loading source parameters
Initialization of navigator(s)
For each navigator (system + phantom)
Loading materials
Loading solid volume
Computing and loading physic processes
Setting range cuts

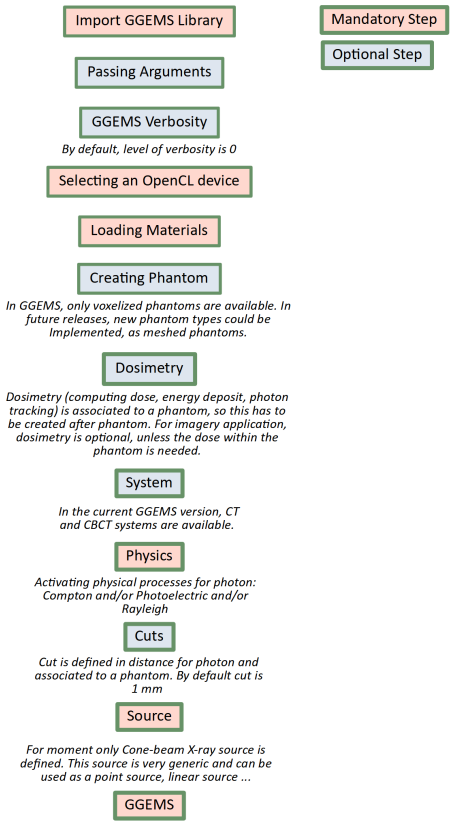


Make a GGEMS Project

GGEMS is designed as a library, and can be called using either python or C++. The performance are the same.

Template

GGEMS (C++ or python) macros are writting following this template:



Python

Before using python and GGEMS, check GGEMS 'python_module' is in your PYTHONPATH variable. PYTHONPATH has to point to the GGEMS library too.

Using GGEMS with python is very simple. A folder storing the project should be created. Inside this folder, write a python file importing GGEMS.

```
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)
```

Next step, an OpenCL device is selected.

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

Then a material database has to be loaded in GGEMS. The material file provided by GGEMS is in 'data' folder. This file can be copy and paste in your project, and a new material can be added.

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

The 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')
```

The photon physical processes are selecting using the process name, the concerning particle and the associated phantom (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')
```

Range cuts are defined in distance, particle type has to 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 particle tracking, if the energy particle is below to the cut the particle is killed and the energy is locally deposited.

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

GGEMS is called in python using the 'ggems_manager' variable. All verboses can be set to 'True' or 'False' depending on the amount of details the user needs. In 'tracking_verbose', the second parameters is the index of particle to track. All objects in GGEMS are initialized with the method 'initialize'. The GGEMS simulations starts with the method 'run'.

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

The last step, exit GGEMS properly by cleaning OpenCL:

```
ggems.delete()
opencl_manager.clean()
exit()
```

C++

Building a project from scratch using GGEMS library in C++ is a little more difficult. A small example is given using CMake.

Create a project folder (named 'my_project' for instance), then 'include' and 'src' folder can be created if custom C++ classes are written. A file named 'main.cc' is created for this example and 'CMakeLists.txt' file is also created. At this stage, the folder structure is:

```
<my_project>
|-- include\
|-- src\
|-- main.cc
|-- CMakeLists.txt
```

Compiling this project can be done using the following 'CMakeLists.txt' example:

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.13 FATAL_ERROR)

SET(ENV{CC} "clang")
SET(ENV{CXX} "clang++")

PROJECT(MYPROJECT LANGUAGES CXX)

FIND_PACKAGE(OpenCL REQUIRED)

SET(GGEMS_INCLUDE_DIRS "" CACHE PATH "Path to the GGEMS include directory")
SET(GGEMS_LIBRARY "" CACHE FILEPATH "GGEMS library")

INCLUDE_DIRECTORIES(${PROJECT_SOURCE_DIR}/include ${GGEMS_INCLUDE_DIRS})
INCLUDE_DIRECTORIES(SYSTEM ${OpenCL_INCLUDE_DIRS})
```

```
LINK_DIRECTORIES(${OpenCL_LIBRARY})

FILE(GLOB source ${PROJECT_SOURCE_DIR}/src/*.cc)

ADD_EXECUTABLE(my_project main.cc ${source})
TARGET_LINK_LIBRARIES(my_project ${OpenCL_LIBRARY} ${GGEMS_LIBRARY})
```

In main.cc file, GGEMS files are included:

```
#include "GGEMS/global/GGEMSOpenCLManager.hh"
#include "GGEMS/global/GGEMS.hh"
#include "GGEMS/materials/GGEMSMaterialsDatabaseManager.hh"
#include "GGEMS/physics/GGEMSRangeCutsManager.hh"
#include "GGEMS/physics/GGEMSProcessesManager.hh"
```

For silent GGEMS execution, the level is set to 0, otherwize 3 for maximum information.

```
GGcout.SetVerbosity(0);
GGcerr.SetVerbosity(0);
GGwarn.SetVerbosity(0);
```

Next step, an OpenCL device is selected. Here, device 0 is selected:

```
GGEMSOpenCLManager& opencl_manager = GGEMSOpenCLManager::GetInstance();
opencl_manager.DeviceToActivate(0);
```

Then a material database has to be loaded in GGEMS. The material file provided by GGEMS is in ‘data’ folder. This file can be copy and paste in your project, and a new material can be added.

```
GGEMSMaterialsDatabaseManager& material_manager = GGEMSMaterialsDatabaseManager::GetInstance
material_manager.SetMaterialsDatabase("materials.txt");
```

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

```
GGEMSProcessesManager& processes_manager = GGEMSProcessesManager::GetInstance();
processes_manager.SetCrossSectionTableNumberOfBins(220);
processes_manager.SetCrossSectionTableMinimumEnergy(1.0f, "keV");
processes_manager.SetCrossSectionTableMaximumEnergy(1.0f, "MeV");
```

The photon physical processes are selecting using the process name, the concerning particle and the associated phantom (or ‘all’ for all defined phantoms).

```
processes_manager.AddProcess("Compton", "gamma", "all");
processes_manager.AddProcess("Photoelectric", "gamma", "all");
processes_manager.AddProcess("Rayleigh", "gamma", "all");
```

In GGEMS, range cuts are defined in distance, particle type has to 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 particle tracking, if the energy particle is below to the cut, then the particle is killed and the energy is locally deposited.

```
GGEMSRangeCutsManager& range_cuts_manager = GGEMSRangeCutsManager::GetInstance();
range_cuts_manager.SetLengthCut("all", "gamma", 0.1f, "mm");
```

GGEMS C++ singleton is called with ‘ggems_manager’ variable. All verboses can be set to ‘True’ or ‘False’ depending on the amount of details the user needs. In ‘tracking_verbose’, the second parameters in the index of particle to track. All objects in GGEMS are initialized with the method ‘initialize’. The GGEMS simulations starts with the method ‘run’.

```
GGEMS ggems;
ggems.SetOpenCLVerbose(true);
ggems.SetMaterialDatabaseVerbose(true);
ggems.SetNavigatorVerbose(true);
ggems.SetSourceVerbose(true);
ggems.SetMemoryRAMVerbose(true);
```

```
ggems.SetProcessVerbose(true);
ggems.SetRangeCutsVerbose(true);
ggems.SetRandomVerbose(true);
ggems.SetProfilingVerbose(true);
ggems.SetTrackingVerbose(true, 0);
```

The last step, exit GGEMS properly by cleaning OpenCL C++ singleton

```
GGEMSOpenCLManager::GetInstance().Clean();
```

Portable Document

GGEMS documentation can be downloaded in PDF format from the following link:

GGEMS pdf