Geant4 hands-on

Marco Pinto and George Dedes

07-11-2017

1 Brief notes

The source code you will work on is based on the example B1 provided with Geant4. You can find it under examples/basic/B1.

2 Prepare your environment

The environment you are working on can be configured via a module system. In order to prepare your environment to run Geant4, you need to open a terminal and issue the following command:

module load geant4/10.03.p01

You should now be ready to use Geant4 in this terminal. Remember: if you close the terminal or open a new terminal, you need to issue the command again. You can avoid this by including the command in your bashrc configuration file.

3 First run

Trees for source and compiled codes can differ substantially from programmer to programmer and it is often a matter of taste. In this case, we have a **src** (source files), **include** (header files) and **build** (compiled code) directories, and a main file (simulation.cc). There is also some other files, namely the CMakeLists.txt and macro files (.mac). The former is a configuration file to compile your code and the latter are files used to automatize and tailor the launch of Geant4, when required.

3.1 Prepare for compilation

Go inside the build directory and do:

cmake ..

This step will launch CMake, which creates the necessary files to build your code in a compiler-independent manner. It does so by reading the instructions inside the CMakeLists.txt file and implementing the appropriate instructions based on the detected system compiler. The two dots tell CMake to look for the CMakeLists.txt file one level above. This step could be done at the same level where the CMakeLists.txt file is but this would create all the auxiliary files at that level. By making all steps inside the build directory, one just needs to delete the entire build directory to start over without touching the higher level directories.

You should see something like:

```
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
```

```
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found CLHEP Version 2.3.4.3
-- Found CLHEP: /software/opt/xenial/x86_64/clhep/2.3.4.3/lib/libCLHEP.so
-- Configuring done
-- Generating done
-- Build files have been written to:
```

To delete everything inside the build directory and therefore having a clean start, you can do (do it from the same level where the main and CMakeLists.txt files are):

rm -rf build/*

3.2 Compile your code

After successful completion of the CMake step, you should have now all the necessary files to compile your code inside the build directory. In the terminal, type:

make

You should see something like:

```
Scanning dependencies of target simulation
[ 14%] Building CXX object CMakeFiles/simulation.dir/simulation.cc.o
[ 28%] Building CXX object CMakeFiles/simulation.dir/src/RunAction.cc.o
[ 42%] Building CXX object CMakeFiles/simulation.dir/src/PrimaryGeneratorAction.cc.o
[ 57%] Building CXX object CMakeFiles/simulation.dir/src/DetectorConstruction.cc.o
[ 71%] Building CXX object CMakeFiles/simulation.dir/src/EventAction.cc.o
[ 85%] Building CXX object CMakeFiles/simulation.dir/src/SteppingAction.cc.o
[100%] Linking CXX executable simulation
[100%] Built target simulation
```

3.3 Run your code

If the code is compiled, you should have a executable called **simulation** inside the build directory. To run your code, type:

./simulation run.mac

The first instruction in the command runs the executable (**./simulation**) and the second gives some arguments to the executable, in this case the name of the macro file to be read (**run.mac**).

You should see a lot of verbosity and, if everything goes well, it should end in a similar way to this:

```
Run Summary
  Number of events processed : 1000
  User=0.27s Real=0.27s Sys=0s



--------------------End of Run------------------------
 The run consisted of 1000 proton of 200 MeV
 Cumulated dose in volume : 122.473 nGy rms = 12.216 nGy
------------------------------------------------------------
```

```
#
Graphics systems deleted.
Visualization Manager deleting...
G4 kernel has come to Quit state.
UserDetectorConstruction deleted.
UserPhysicsList deleted.
UserRunAction deleted.
UserPrimaryGenerator deleted.
RunManager is deleting RunManagerKernel.
EventManager deleted.
Units table cleared.
Total navigation history collections cleaned: 9
================== Deleting memory pools ===================
Pool ID '20G4NavigationLevelRep', size : 0.0115 MB
Pool ID '24G4ReferenceCountedHandleIvE', size : 0.000961 MB
Pool ID '17G4DynamicParticle', size : 0.0413 MB
Pool ID '7G4Event', size : 0.000961 MB
Pool ID '15G4PrimaryVertex', size : 0.000961 MB
Pool ID '17G4PrimaryParticle', size : 0.000961 MB
Pool ID '7G4Track', size : 0.0827 MB
Pool ID '18G4TouchableHistory', size : 0.000961 MB
Pool ID '15G4CountedObjectIvE', size : 0.000961 MB
Pool ID '10G4Fragment', size : 0.00288 MB
Pool ID '17G4ReactionProduct', size : 0.00288 MB
Number of memory pools allocated: 11; of which, static: 0
Dynamic pools deleted: 11 / Total memory freed: 0.15 MB
============================================================
G4Allocator objects are deleted.
UImanager deleted.
StateManager deleted.
RunManagerKernel is deleted. Good bye :)
```

If you saw something similar to these lines of verbosity, then congratulations: you have just run your Geant4 application.

4 Understand your code and do some exercises

Open the source code with a software of your choosing from menu under the section 'Development'. Geany and Code::Blocks IDE are good options.

NOTE: when compiling, if you are having some errors like:

```
error: invalid use of incomplete type  'class XXXXXX'
```

it usually means that you need to include some header files in your source code. Usually writing `#include "XXXXXX.hh"` does the trick.

4.1 DetectorConstruction

Look at the class 'DetectorConstruction' and try to understand the geometry implemented. To help you with that, you can run the code with:

./simulation

This will trigger the visualization drivers and open GUI with the geometry. Compare what you see in the GUI with what is defined in the source code. The axes drawn in the GUI are x (red), y (green) and z (blue) and the their origin is at (0,0,0). Explore the GUI and check what you can do, namely using the right mouse button.

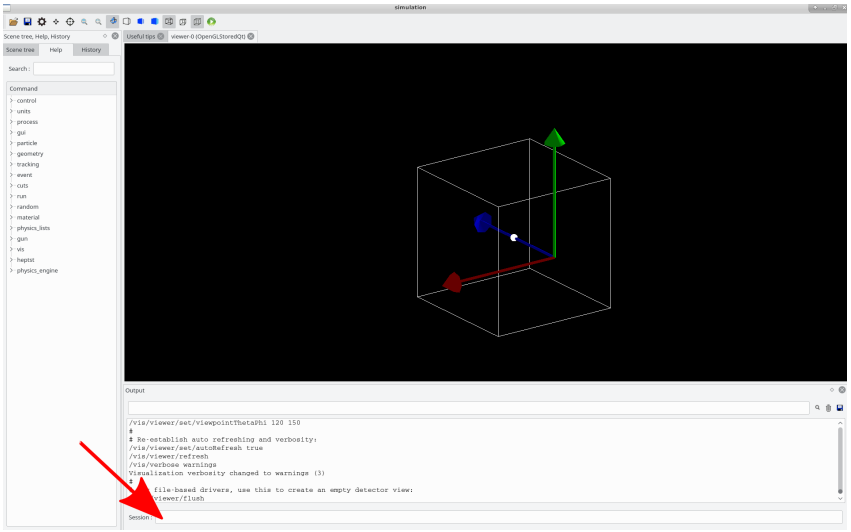
Task 1: change the detector shape placed at the center of the phantom from a sphere ('G4Orb') to a cylinder ('G4Tubs'). Make a cylinder with 3 cm radius (zero inner radius) and 5 cm long positioned at the center of the phantom. Search for 'G4Tubs' in a search engine, look for it in the Geant4 manual or check yesterday's lecture.

4.2 PrimaryGeneratorAction

Open the class 'PrimaryGeneratorAction' and try to understand it. This class defines the beam parameters. In this case, you have a 200 MeV proton beam traveling along the z-axis. The starting position for this beam is at -10 cm in the z-direction, and x and y are randomly sampled for each new event following a Gaussian function with zero and 1 cm as mean and sigma values, respectively.

To observe the beam in the GUI, issue the following command in the GUI:

/run/beamOn 10



This command will launch the simulation of 10 protons. Try with other number of protons but be careful to not overload the graphical system.

Task 2: create a pencil beam (i.e. a beam without dimensions) starting at (0,0,-5 cm). Compile the code with the changes (note: you just need to issue the command '**make**' in the build directory) and check the result with the GUI.

Task 3: create a square beam with an area of 5x5 cm^2 (x and y) starting at z = -10 cm and centered around (0,0). Compile the code with the changes and check the result with the GUI. Note: Geant4 has the function '**G4UniformRand()**' that gives a random number between 0 and 1. Then, you just need to make it in a way that will randomly create x and y in the interval of [-2.5 cm, 2.5 cm]. Remember that the default Geant4 unit for length is mm, so you must 'multiply' by cm in order to convert the values!

Task 4: change the simulated particle to 6-MeV photons. In Geant4, a photon is called '**gamma**'.

4.3 RunAction

This class will be called in the beginning and end of your run and it is often used to store data during the simulation. In this case, you can see there are two variables (fEdep and fEdep2) following this purpose.

At the construction time, this class will create new units for dose. Those will be then used internally by Geant4, namely when using the function 'G4BestUnit' with 'Dose' as unit category (see in the class some examples). This will allow to print units in a more user-friendly manner.

At the end of the simulation (see function `void RunAction::BeginOfRunAction( const G4Run* )`) the code will print some verbosity regarding the dose in the scoring volume, i.e. the detector placed inside the water phantom.

Task 5: run a simulation with the macro (**./simulation run.mac**) and look in the verbosity for the dose in the detector.

Task 6: run a simulation in which you print the total energy deposited and the energy deposited per primary particle. Print those values in MeV and Joule. **Note:** you can check the units registered in Geant4 by issuing the command `/units/list` in the GUI terminal.

4.4 SteppingAction

This class will be called in every step and it is often used to collect simulation information at a given time. At this moment, this class is only accessing the total energy deposited in the step and sending it to the class EventAction in order to accumulate it, if the step is performed inside the detector (the scoring volume).

Task 7: if the step is inside the detector, print:

- the position of the particle before and after the step
- the step length
- the kinetic energy of the particle (you need to use G4Track)
- the time of flight since the beginning of the event (look for global time in G4Track)
- the particle name (you need to go to G4Step -> G4Track -> G4ParticleDefinition)

Note: you have access to the G4Step in the SteppingAction class (the pointer 'step') and you can check its functions in this website.