

# GEANT4

Sungur Özkan

September 21, 2023

## Contents

- 1- How to install Geant4
- 2- About the virtual machine
- 3- Geant4 Interface
- 4- Creating a project
- 5- Detector Construction
- 6- Materials
- 7- Physics lists
- 8- Generating particles
- 9- Sensitive detectors
- 10- Storing hits in a ROOT file
- 11- Adding macro files
- 12- Cherenkov Radiator

## 1 How to install

The simplest way to get started with GEANT4 is by using a pre-installed virtual machine. Installing it directly on your computer can be a bit challenging. However, if you're working on larger projects, the virtual machine may not perform optimally due to its performance limitations. In such situations, you can choose to install GEANT4 directly on your Windows, MacOS, or Linux operating system.

Here is a virtual machine package that contains Geant4 version 11.1.2, operating on the Rocky Linux system. This package also includes several handy utility tools, such as visualization, analysis software, and development resources like ROOT, Qt5, and Python, already set up for you. This virtual environment is fully functional and supports various features like USB devices, displays, network connections (both cable and wireless), and more. You can find comprehensive information about the bundled packages on the website.

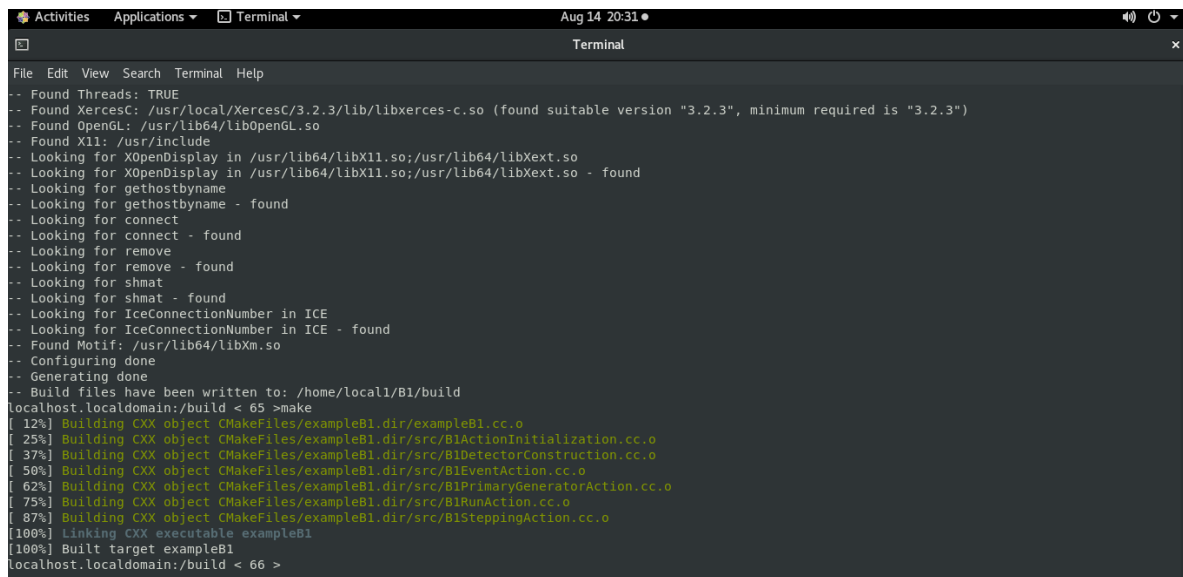
Here is the latest version of Geant4 if you want to install on your own system. This is a written installation guide and this is a video guide. Also, this is the official user documentation for application developers.

## 2 The virtual machine

In this guide, we'll use a virtual machine with Geant4 installed, located at `"/src/geant4.10.07.p03"` (note that the version may vary) within the virtual machine environment. To find the `"src"` folder: *Files* → *Other Locations* → *Computer*. Inside the `"geant4.10.07.p03"`, you'll find the `"examples"` folder. To make things easier, you can directly copy and paste the examples you want to run into your home directory, because it may not allow you to create a new folder inside an example, which we will need to do now.

To run the first example B1, follow these steps:

1. Launch the terminal.
2. Navigate to the B1 folder using the `'cd B1'` command.
3. Create a directory called `"build"` with `'mkdir build'` and access that directory using `'cd build'`.
4. Perform the initial compilation of the example using CMake by executing `'cmake ..'`. You should run this command each time you add a new file to the program.
5. Compile the example using `'make'`. You should run this command each time you make changes to the code.
6. Execute the example by running `'./exampleB1'`.



```
File Edit View Search Terminal Help
-- Found Threads: TRUE
-- Found XercesC: /usr/local/XercesC/3.2.3/lib/libxerces-c.so (found suitable version "3.2.3", minimum required is "3.2.3")
-- Found OpenGL: /usr/lib64/libOpenGL.so
-- Found X11: /usr/include
-- Looking for XOpenDisplay in /usr/lib64/libX11.so;/usr/lib64/libXext.so
-- Looking for XOpenDisplay in /usr/lib64/libX11.so;/usr/lib64/libXext.so - found
-- Looking for gethostbyname
-- Looking for gethostbyname - found
-- Looking for connect
-- Looking for connect - found
-- Looking for remove
-- Looking for remove - found
-- Looking for shmatt
-- Looking for shmatt - found
-- Looking for IceConnectionNumber in ICE
-- Looking for IceConnectionNumber in ICE - found
-- Found Motif: /usr/lib64/libXm.so
-- Configuring done
-- Generating done
-- Build files have been written to: /home/local1/B1/build
localhost.localdomain:/build < 65 > make
[ 12%] Building CXX object CMakeFiles/exampleB1.dir/exampleB1.cc.o
[ 25%] Building CXX object CMakeFiles/exampleB1.dir/src/B1ActionInitialization.cc.o
[ 37%] Building CXX object CMakeFiles/exampleB1.dir/src/B1DetectorConstruction.cc.o
[ 50%] Building CXX object CMakeFiles/exampleB1.dir/src/B1EventAction.cc.o
[ 62%] Building CXX object CMakeFiles/exampleB1.dir/src/B1PrimaryGeneratorAction.cc.o
[ 75%] Building CXX object CMakeFiles/exampleB1.dir/src/B1RunAction.cc.o
[ 87%] Building CXX object CMakeFiles/exampleB1.dir/src/B1SteppingAction.cc.o
[100%] Linking CXX executable exampleB1
[100%] Built target exampleB1
localhost.localdomain:/build < 66 >
```

Figure 1: After installation

### 3 Geant4 Interface

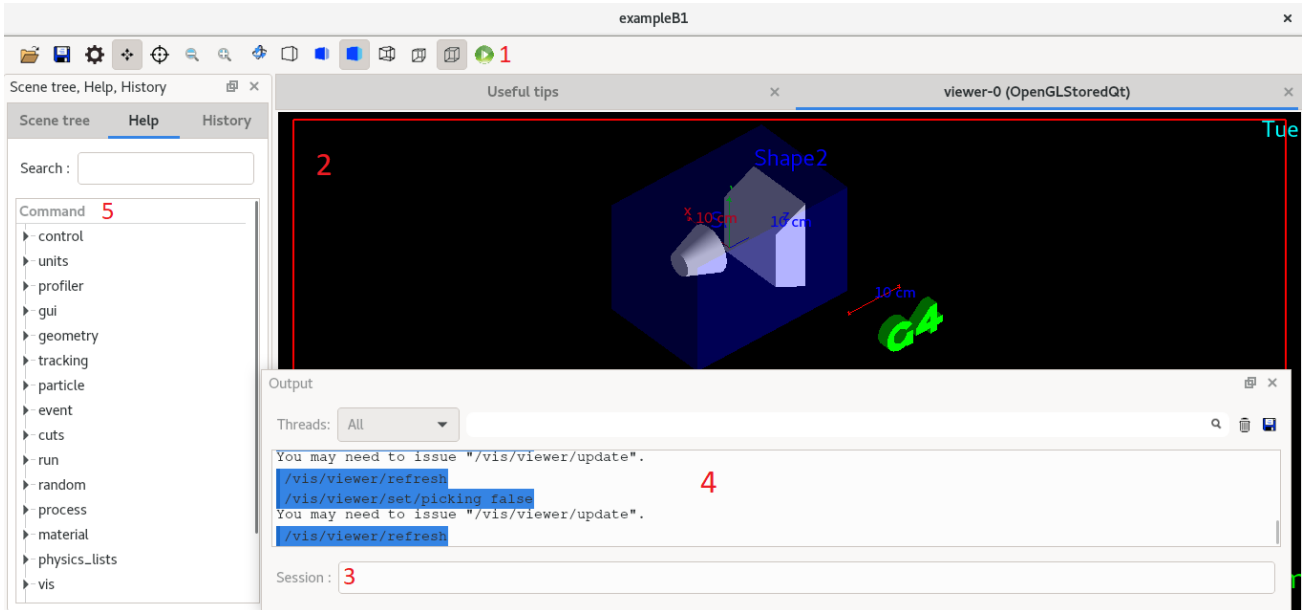


Figure 2: Example B1

To gain a clearer understanding of the Geant4 interface, let's take a closer look. It's important to note that we don't necessarily need to visualize this interface every time we run the program; we also have the option to run it in batch mode, which we'll explore in more detail later. But for now, let's familiarize ourselves with what's in front of us:

1. This symbol is used to run an event.
2. This is the interactive visualization of our geometry. We can rotate, take a closer look, etc.
3. This box is used to give commands about the program. We can interactively change the properties of the particle, run multiple events at once or even change the geometry.
4. This area displays essential information following the execution of an event, including the time taken and energy interactions between particles.
5. This tree structure provides a comprehensive list of available commands along with detailed information about each command.

Here are some practical commands you can use to interact with your simulation:

```
// Change the Particle
/gun/particle mu-

// Change the Energy of the Particle
/gun/energy 10 GeV

// Change the Starting Position
/gun/position 0.0 0.0 10.0 cm
```

## 4 Creating a project

To get started, we need a file named "CMakeLists.txt" as we use CMake for project compilation. In this file, we define the project name, include necessary packages for the interface and visualization, and set up an executable file. This is how the "CMakeLists.txt" file is structured::

```
cmake_minimum_required(VERSION 2.6 FATAL_ERROR)    # Set the minimum required CMake version

project(Simulation)    # Set the name of the project to "Simulation"

# Find and configure Geant4 package
find_package(Geant4 REQUIRED ui_all vis_all)

# Include Geant4 configuration files
include(${Geant4_USE_FILE})

# Glob source files and header files in the project directory
file(GLOB sources ${PROJECT_SOURCE_DIR}/*.cc)
file(GLOB headers ${PROJECT_SOURCE_DIR}/*.hh)

# Create an executable named "sim" from the source files
add_executable(sim sim.cc ${sources} ${headers})

# Link the "sim" executable with Geant4 libraries
target_link_libraries(sim ${Geant4_LIBRARIES})

# Create a custom target named "Simulation" that depends on the "sim" executable
add_custom_target(Simulation DEPENDS sim)
```

Next, we create the main file, which I called `sim.cc`. In a Geant4 project, the main file typically contains the main function, which serves as the entry point for your simulation. We use some classes to set up the basic structure of a Geant4 application, allowing us to configure and run the simulations interactively.

First, we have the `G4RunManager` class, which is the heart of Geant4. It is a central class that manages the execution and coordination of the simulation run. It acts as the main control point for setting up and running Geant4 simulations. The `G4RunManager` class is essential for handling the initialization, event generation, event processing, and cleanup phases of the simulation.

Next, we have the user interface. `G4UIExecutive` is a class that simplifies the setup and management of the user interface (UI) for interactive control and visualization of Geant4 simulations. It is used to initialize and start the user interface system, making it easier for users to interactively control the simulation and visualize the results. One of the main roles of `G4UIExecutive` is to handle the command-line arguments passed to the application. It recognizes command-line options related to the user interface and sets up the appropriate UI based on the user's choice.

Next, we have the visualisation manager. The `G4VisManager` class is responsible for initializing the visualization system, creating visualization drivers, and handling the visualization of the simulated events. It enables users to observe the interaction of particles with the detector materials and visualize various aspects of the simulation, such as particle trajectories, energy deposits, and detector geometry.

We can also start the visualisation manager for the empty project to run: `visManager->Initialize;`

Next, we have `G4UIManager`. It is a class that manages the user interface for interactive control and visualization of Geant4 simulations. It provides a bridge between the user and the simulation, allowing users to interactively control the behavior of the simulation, set parameters, and visualize the results in real-time. Combining all of these classes, the main file looks like this:

```

#include <iostream> // Include the standard C++ input/output library

#include "G4RunManager.hh" // Include Geant4's Run Manager
#include "G4UIManager.hh" // Include Geant4's User Interface Manager
#include "G4UIExecutive.hh" // Include Geant4's User Interface Executive
#include "G4VisManager.hh" // Include Geant4's Visualization Manager
#include "G4VisExecutive.hh" // Include Geant4's Visualization Executive

// Main function
int main(int argc, char** argv)
{
    // Create a Geant4 Run Manager
    G4RunManager *runManager = new G4RunManager();

    // Create a Geant4 User Interface Executive (for interactive mode)
    G4UIExecutive *ui = new G4UIExecutive(argc, argv);

    // Create a Geant4 Visualization Executive
    G4VisManager *visManager = new G4VisExecutive();
    visManager->Initialize(); // Initialize the visualization manager

    // Get the Geant4 User Interface Manager
    G4UIManager *UImanager = G4UIManager::GetUIpointer();

    // Start a Geant4 session for interactive commands
    ui->SessionStart();

    // Clean up and exit
    return 0;
}

```

With the CMakeLists.txt and main file sim.cc, we can run an empty Geant4 program.

1. Inside the folder, create a folder called build with "mkdir build"
2. Go to the build directory with "cd build"
3. Compile it with cmake "cmake .."
4. Compile with "make"
5. If you get the message "Linking executable sim.cc", then it's done. You can run it with the command "./sim.cc"

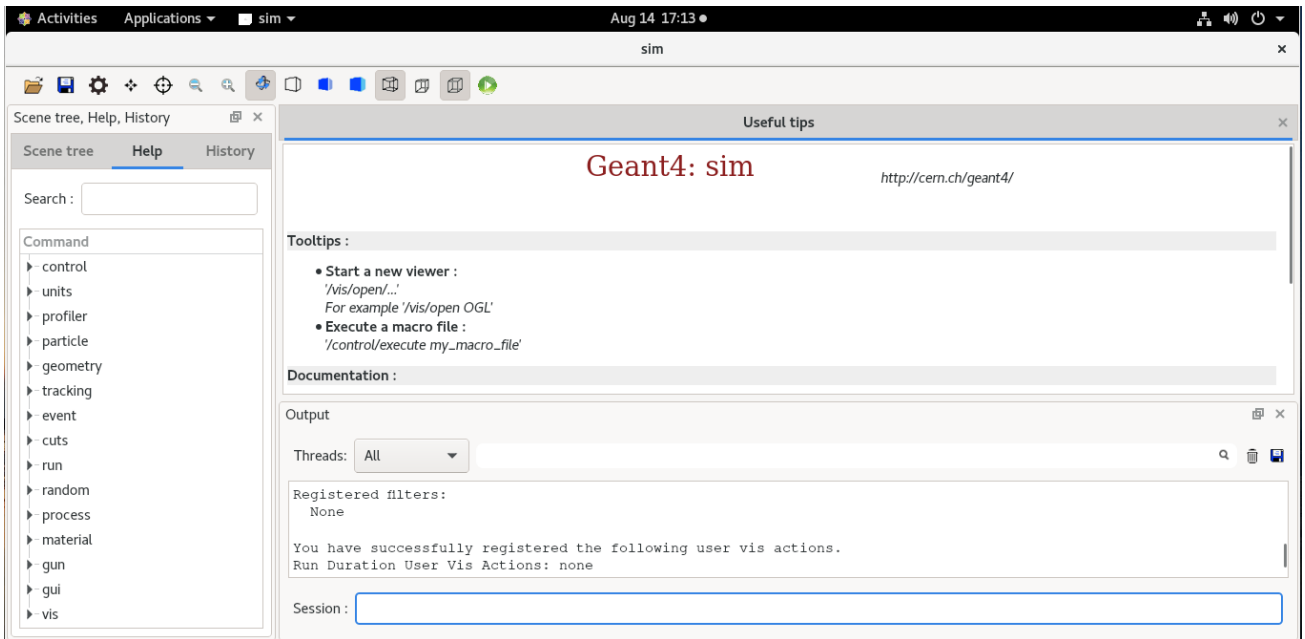


Figure 3: Empty Geant4

## 5 Detector Construction

Now, we can start the construction of our geometry. We begin with the header file named "construction.hh." Within this file, we create our own class called 'MyDetectorConstruction', which must inherit from 'G4VUserDetectorConstruction'. We include constructors and destructors. The central element of this class is the 'G4VPhysicalVolume', which serves as the return value of the 'Construct()' function. The 'Construct()' function is the core component responsible for building the entire detector geometry. Here is the header file:

```
#ifndef CONSTRUCTION_HH
#define CONSTRUCTION_HH

#include "G4VUserDetectorConstruction.hh" // Include the Geant4 user detector construction header
#include "G4VPhysicalVolume.hh" // Include the Geant4 physical volume header
#include "G4LogicalVolume.hh" // Include the Geant4 logical volume header

// Define the MyDetectorConstruction class, which inherits from G4VUserDetectorConstruction
class MyDetectorConstruction : public G4VUserDetectorConstruction
{
public:
    MyDetectorConstruction(); // Constructor

    ~MyDetectorConstruction(); // Destructor

    virtual G4VPhysicalVolume *Construct(); // Implementation of the Construct method
};

#endif
```

We continue with the source file construction.cc. We start with adding constructor and destructor. Then, we have the Construct() function, which will include all the important information. We will construct the detector geometry and materials in this function.

A detector geometry in Geant4 is made of a number of volumes. The largest volume is called the World volume. It must contain, with some margin, all other volumes in the detector geometry. The other volumes are created and placed inside previous volumes, included in the World volume. The most simple (and efficient) shape to describe the World is a box. Each volume is created by describing its shape and its physical characteristics, and then placing it inside a containing volume. To describe a volume's full properties, we use a logical volume. It includes the geometrical properties of the solid, and adds physical characteristics: the material of the volume; whether it contains any sensitive detector elements; the magnetic field; etc.

To create a simple box, you only need to define its name and its extent along each of the Cartesian axes.

```
G4Box *solidWorld = new G4Box("solidWorld", 3.*m, 1.*m, 1.*m);
```

This creates a box named "solidWorld" with the extent from -3.0 meters to +3.0 meters along the X axis, from -1.0 to 1.0 meters in Y, and from -1.0 to 1.0 meters in Z. Note that the G4Box constructor takes as arguments the halves of the total box size. In Geant4, we can create boxes, cylinders, cones, spheres etc.

You can check this page to find all information in this link

Then we define something called logical volume. It refers to a representation of a geometrical shape with associated material properties used to describe the composition and geometry of a specific component. In the context of a box, we fill the box with this logical volume. For example, to create a simple logical volume filled with argon gas (we have to define the argon gas before, which I'll cover in the next chapter):

```
G4LogicalVolume *logicWorld = new G4LogicalVolume(solidWorld, Ar, "logicWorld");
```

To place a volume, we have to define the physical volume. A "physical volume" represents an instance of a logical volume placed at a specific position and orientation in 3D space within the simulation geometry.

```
G4VPhysicalVolume* physWorld
= new G4PVPlacement(0,                      // no rotation
                    G4ThreeVector(0*m, 0*m, 0*m), // translation position
                    logicWorld,             // its logical volume
                    "physWorld",           // its name
                    0,                      // its mother (logical) volume
                    false,                  // no boolean operations
                    0);                    // its copy number
```



This places the logical volume logicWorld at the origin unrotated. The resulting physical volume is named “physWorld” and has a copy number of 0. This is the World volume, which is the largest volume created, and which contains all other volumes. This volume obviously cannot be contained in any other. So, it must be created as a G4PVPlacement with a null mother pointer. It also must be unrotated, and it must be placed at the origin of the global coordinate system.

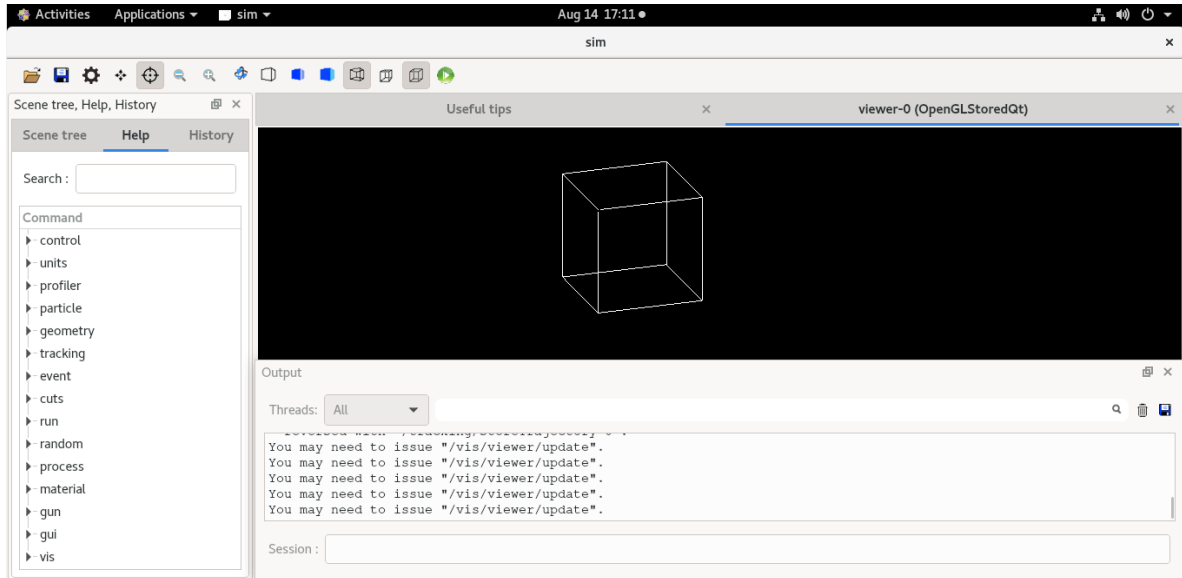


Figure 4: Mother volume

## 6 Materials

In nature, general materials (chemical compounds, mixtures) are made of elements, and elements are made of isotopes. Therefore, these are the three main classes designed in Geant4. Each of these classes has a table as a static data member, which is for keeping track of the instances created of the respective classes. All three objects automatically register themselves into the corresponding table on construction, and should never be deleted in user code.

The G4Element class describes the properties of the atoms: atomic number, number of nucleons, atomic mass, shell energy, as well as quantities such as cross sections per atom, etc.

The G4Material class describes the macroscopic properties of matter: density, state, temperature, pressure, as well as macroscopic quantities like radiation length, mean free path,  $dE/dx$ , etc. The G4Material class is the one which is visible to the rest of the toolkit, and is used by the tracking, the geometry, and the physics. It contains all the information relative to the eventual elements and isotopes of which it is made, at the same time hiding the implementation details.

In the example below, liquid argon is created, by specifying its name, density, mass per mole, and atomic number.

```
G4double z, a, density;
density = 1.390*g/cm3;
a = 39.95*g/mole;
G4Material* lAr = new G4Material(name="liquidArgon", z=18., a, density);
```

The pointer to the material, lAr, will be used to specify the matter of which a given logical volume is made:

```
G4LogicalVolume* myLbox = new G4LogicalVolume(aBox,lAr,"Lbox",0,0,0);
```

In the example below, the water, H<sub>2</sub>O, is built from its components, by specifying the number of atoms in the molecule.

```
G4double z, a, density;
G4String name, symbol;
G4int ncomponents, natoms;

a = 1.01*g/mole;
G4Element* elH = new G4Element(name="Hydrogen",symbol="H" , z= 1., a);

a = 16.00*g/mole;
G4Element* elO = new G4Element(name="Oxygen" ,symbol="O" , z= 8., a);

density = 1.000*g/cm3;
G4Material* H2O = new G4Material(name="Water",density,ncomponents=2);
H2O->AddElement(elH, natoms=2);
H2O->AddElement(elO, natoms=1);
```

There's a simpler way to do this, it's to use G4NistManager class. With this class, we can reach the GEANT4 database and find pre-defined elements and materials. For example, let's define gold.

```
G4NistManager *nist = G4NistManager::Instance();
G4Element *Au = nist->FindOrBuildElement("Au");
G4Material *Gold = new G4Material("Gold", 19.320*g/cm3, 1);
Gold->AddElement(nist->FindOrBuildElement("Au"), 1);
```

For example, to define the air molecule;

```
G4Material *Air = nist->FindOrBuildMaterial("G4_AIR");
```

You can find the element and material database [here](#). We need to finish the function with return physWorld. The file will look something like this:

```
#include "construction.hh" // Include the header file for MyDetectorConstruction

// Constructor for MyDetectorConstruction class
MyDetectorConstruction::MyDetectorConstruction()
{}

// Destructor for MyDetectorConstruction class
MyDetectorConstruction::~MyDetectorConstruction()
{}

// Implementation of the Construct method
G4VPhysicalVolume *MyDetectorConstruction::Construct()
```

```

{
    // Defining materials
    // Detector construction
    return physWorld; // Return the constructed physical world volume
}

```

Finally, we have to include the construction in the main file. We have to include the header file, also tell the run manager to initialize the geometry.

```
runManager->SetUserInitialization(new MyDetectorConstruction());
```

## 7 Physics lists

Physics lists in Geant4 are collections of models and algorithms that define how particles interact with matter and with each other during particle transport simulations. These lists encapsulate the various physical processes that particles undergo, ranging from electromagnetic interactions to hadronic (strong nuclear force) interactions, decay processes, and more. In essence, physics lists determine how particles behave as they travel through different materials in your simulation. The choice of a physics list can significantly affect the accuracy, speed, and level of detail of your simulation results.

To use a physics list in your Geant4 simulation, you typically create a class that inherits from `G4VModularPhysicsList`, where you can register the physics processes you want to include. You can also modify or extend existing physics lists to suit your simulation needs.

I will create two files: `physics.cc` and `physics.hh`. `Physics.hh` will look like this:

```

#ifndef PHYSICS_HH
#define PHYSICS_HH

// Include the necessary Geant4 header files for physics
#include "G4VModularPhysicsList.hh"
#include "G4EmStandardPhysics.hh"
#include "G4OpticalPhysics.hh"

// Define the MyPhysicsList class, which inherits from G4VModularPhysicsList
class MyPhysicsList : public G4VModularPhysicsList
{
public:
    // Constructor for MyPhysicsList class
    MyPhysicsList();

    // Destructor for MyPhysicsList class
    ~MyPhysicsList();
};

#endif

```

G4EmStandardPhysics is to simulate electromagnetic interactions of particles with matter. It helps Geant4 simulate how particles like electrons and photons interact with the atoms and molecules of different materials. G4OpticalPhysics is to simulate the behavior of photons in optical processes, such as those encountered in scintillation, Cherenkov radiation, and other light-related phenomena. This physics class is essential for accurately modeling light propagation and interactions in your simulation. There are many more physics lists, but it's important to use only the ones you need, because it takes more time to calculate with more physics lists.

In the source file, we register this lists in the constructor.

```
#include "physics.hh" // Include the header file for MyPhysicsList

// Constructor for MyPhysicsList class
MyPhysicsList::MyPhysicsList()
{
    // Register standard electromagnetic physics processes
    RegisterPhysics(new G4EmStandardPhysics());

    // Register optical physics processes
    RegisterPhysics(new G4OpticalPhysics());
}

// Destructor for MyPhysicsList class
MyPhysicsList::~MyPhysicsList()
{}
```

Physics lists are registered, but we have to include the lists in the main file by including the header file physics.hh. We should also tell this to RunManager by adding this line to the main file;

```
runManager->SetUserInitialization(new MyPhysicsList());
```

At this point, we can also visualize the geometry by adding some commands in the main file:

```
UImanager->ApplyCommand("/vis/open OGL"); // Initialize OpenGL
UImanager->ApplyCommand("/vis/drawVolume"); // Draw all volumes
```

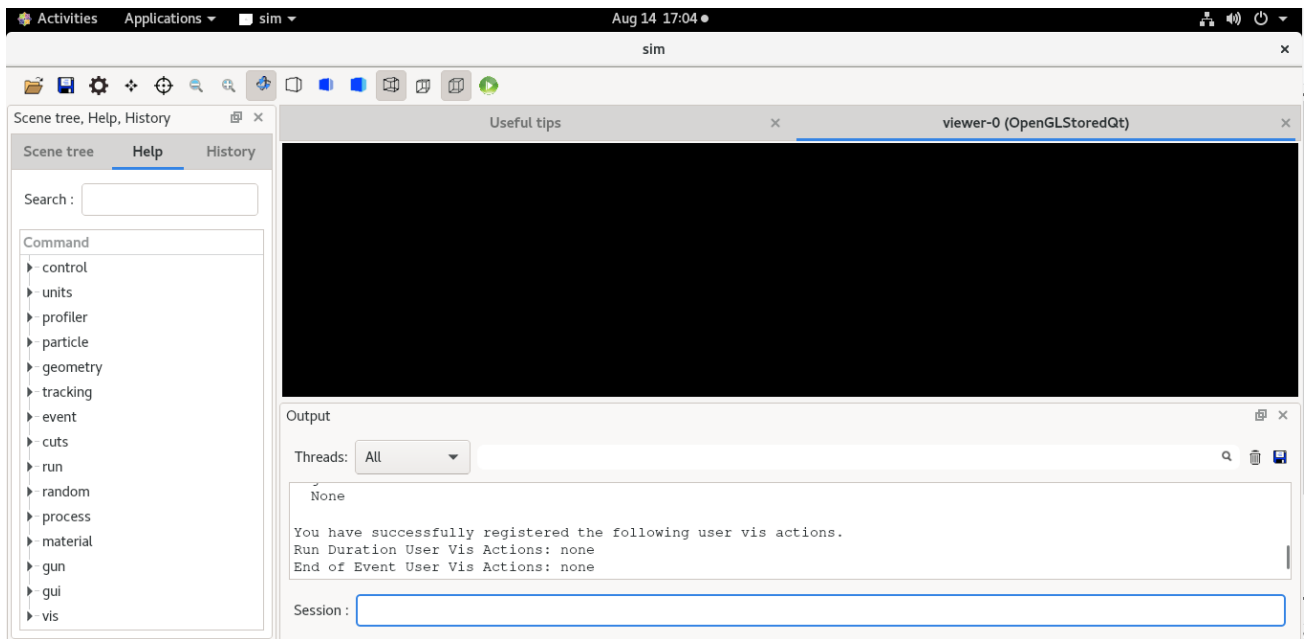


Figure 5: OpenGL initialized

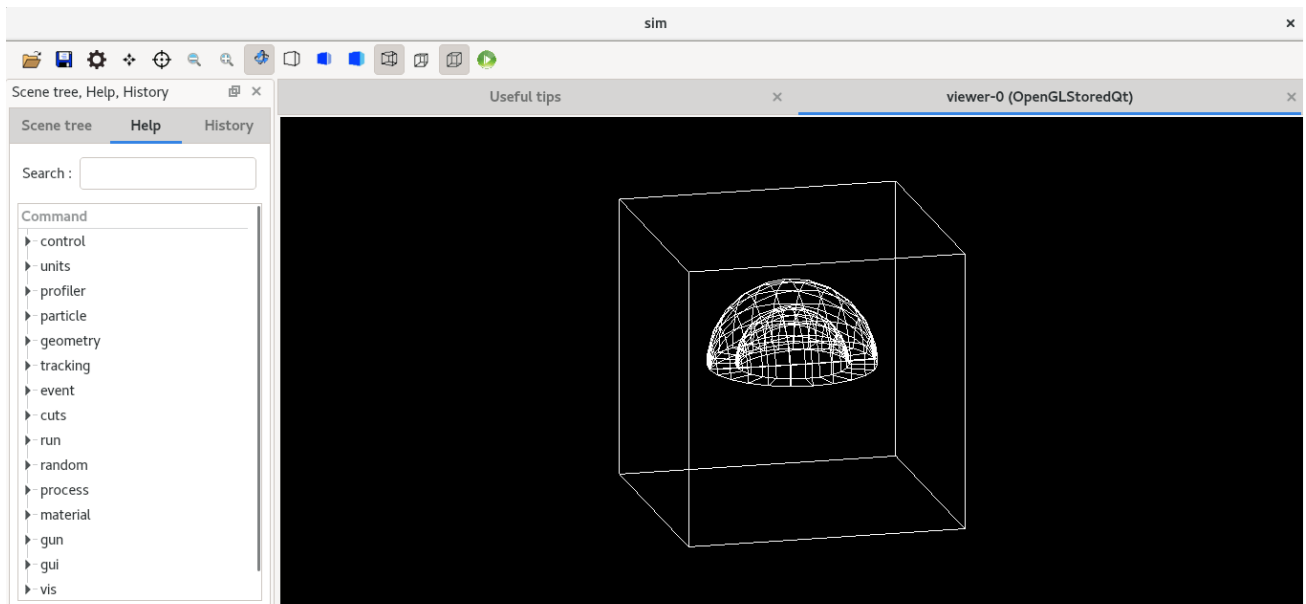


Figure 6: Volume drawn

## 8 Generating particles

In Geant4, generating particles involves creating instances of particle objects and setting their properties before initiating a simulation. Geant4 provides the `G4ParticleGun` class for generating primary particles. We have to define two classes, one for action initialization, one for primary generator. I will create the two files "action.cc" and "action.hh" for action initialization. We have to inherit from a class called "G4VUserActionInitialization" which is a class in Geant4 that you use to initialize user-defined actions for your simulation.

In the header file, I defined a class called "MyActionInitialization", which inherits from the class "G4VUserActionInitialization":

```
#ifndef ACTION_HH
#define ACTION_HH

// Include the necessary Geant4 header file for user action initialization
#include "G4VUserActionInitialization.hh"

// Define the MyActionInitialization class, which inherits from G4VUserActionInitialization
class MyActionInitialization : public G4VUserActionInitialization
{
public:
    // Constructor for MyActionInitialization class
    MyActionInitialization();

    // Destructor for MyActionInitialization class
    ~MyActionInitialization();

    // Implementation of the Build method required by G4VUserActionInitialization
    virtual void Build() const;
};

#endif
```

And the source file:

```
#include "action.hh" // Include the header file for MyActionInitialization

// Constructor for MyActionInitialization class
MyActionInitialization::MyActionInitialization()
{}

// Destructor for MyActionInitialization class
MyActionInitialization::~~MyActionInitialization()
{}

```

```
// Build method definition
void MyActionInitialization::Build() const
{
    // This is an empty method.
    // In Geant4, this is where you would typically initialize
    // and set up user-defined actions for the simulation.
    // Depending on the simulation's requirements, you would
    // create and configure instances of user-defined actions
}
```

For now, the build function is empty, because we need the particle gun. For that, I will create two files "generator.cc" and "generator.hh" where we have to inherit from a class called "G4VUserPrimaryGeneratorAction".

In the header file, I defined a class called "MyPrimaryGenerator", which inherits from the class "G4VUserPrimaryGeneratorAction". Also, some header files should be included such as "G4SystemOfUnits.hh" for units, "G4ParticleTable.hh" for particle names, etc.

```
#ifndef GENERATOR_HH
#define GENERATOR_HH

// Include necessary Geant4 header files
#include "G4VUserPrimaryGeneratorAction.hh"
#include "G4ParticleGun.hh" // Geant4 class for generating primary particles
#include "G4SystemOfUnits.hh" // Geant4 unit system
#include "G4ParticleTable.hh" // Geant4 class for particle properties

// Define the MyPrimaryGenerator class, which inherits from G4VUserPrimaryGeneratorAction
class MyPrimaryGenerator : public G4VUserPrimaryGeneratorAction
{
public:
    // Constructor for MyPrimaryGenerator class
    MyPrimaryGenerator();

    // Destructor for MyPrimaryGenerator class
    ~MyPrimaryGenerator();

    // Implementation of the GeneratePrimaries method required by G4VUserPrimaryGeneratorAction
    virtual void GeneratePrimaries(G4Event*);

private:
    G4ParticleGun *fParticleGun; // Pointer to a Geant4 ParticleGun object
};

#endif
```

The GeneratePrimaries method here is a part of the user-defined "Primary Generator Action." It's a crucial function that generates primary particles and initializes their properties before they're introduced into the simulation. Essentially, it's responsible for creating the initial particles that will travel through your simulated environment.

G4ParticleGun is a class that's like a tool you use to "shoot" particles into your simulation. So, when you write `G4ParticleGun *fParticleGun;`, you're saying, "I'm creating a remote control called `fParticleGun` that I can use to control and manage my particle-shooting tool."

Now, we can move on to the source file where we can finally decide the properties of the particle gun.

```
#include "generator.hh"

MyPrimaryGenerator::MyPrimaryGenerator() // Constructor
{
    fParticleGun = new G4ParticleGun(1); // Create a new instance of G4ParticleGun with 1 particle
}

MyPrimaryGenerator::~MyPrimaryGenerator() // Destructor
{
    // Delete the fParticleGun object to free up memory
    delete fParticleGun;
}
```

So far, we're setting up the "particle shooting tool" for this generator. The argument (1) means you're creating one particle at a time. After that, it's cleaning up the resources used by the "particle shooting tool" when the `MyPrimaryGenerator` object is destroyed. In the remaining part of the source code, we decide which particle we want; the energy, the momentum and the position of the particle. In the context of Geant4, the momentum threevector of the particle represents the direction of the particle.

```
void MyPrimaryGenerator::GeneratePrimaries(G4Event *anEvent)
{
    // Get the particle table
    G4ParticleTable *particleTable = G4ParticleTable::GetParticleTable();

    // Define the particle (proton particle)
    G4String particleName = "proton";
    G4ParticleDefinition *particle = particleTable->FindParticle("proton");

    // Set initial position and momentum direction
    G4ThreeVector pos(0., 0., 0.);
    G4ThreeVector mom(0., 0., 1.); // positive z-direction

    // Configure the particle gun properties
    fParticleGun->SetParticlePosition(pos);
    fParticleGun->SetParticleMomentumDirection(mom);
    fParticleGun->SetParticleMomentum(100. * GeV);
    fParticleGun->SetParticleDefinition(particle);

    // Generate the primary vertex for the event
    fParticleGun->GeneratePrimaryVertex(anEvent);
}
```



Finally, we have to go back to the action initialization function and implement the particle gun. Going back to the action.cc file, you basically add these commands to the Build function:

```
MyPrimaryGenerator *generator = new MyPrimaryGenerator();  
SetUserAction(generator);
```

Now, we can use the particle gun, but we should also tell Geant4 to draw the trajectory of the particles, and refresh it before each event. For that, we go back to the main file:

```
UImanager->ApplyCommand("/vis/scene/add/trajectories smooth"); // draw trajectories  
UImanager->ApplyCommand("/vis/viewer/set/autoRefresh true"); // refresh
```

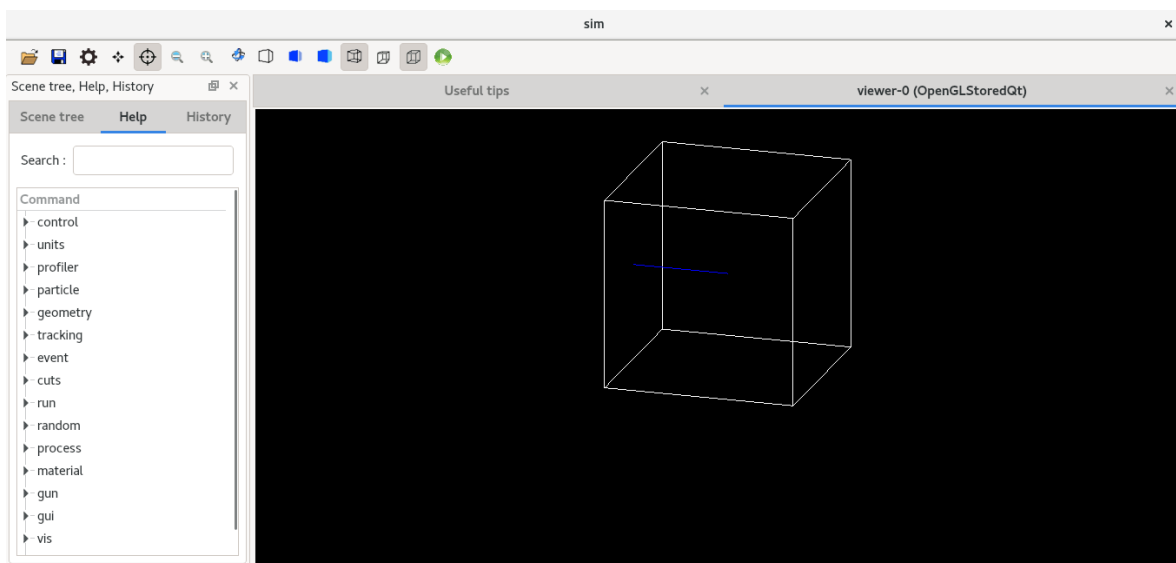


Figure 7: Particle generated from origin

## 9 Sensitive Detectors

Sensitive detectors in Geant4 are components that you define to track and accumulate information about the interactions of particles with specific detector volumes in your simulation. They play a crucial role in simulating how particles deposit energy, create secondary particles, or trigger signals in your detectors. In Geant4, you create sensitive detectors by defining classes that inherit from the "G4VSensitiveDetector" class. You then connect these sensitive detectors to specific logical volumes in your simulation. When a particle interacts with a volume attached to a sensitive detector, Geant4 calls the sensitive detector's functions to record the interaction details.

First thing to do when constructing a sensitive detector is to define the detector volume. I will create a 1cm\*1cm\*2cm box and cover a surface of the mother volume with these detector volumes.

Inside the MyDetectorConstruction::Construct() function:

```
// Define the mother volume
G4NistManager *nist = G4NistManager::Instance();
G4Material *worldMat = nist->FindOrBuildMaterial("G4_AIR");

// A cube with 1*m sides
G4Box *solidWorld = new G4Box("solidWorld", 0.5*m, 0.5*m, 0.5*m);
G4LogicalVolume *logicWorld = new G4LogicalVolume(solidWorld, worldMat, "logicWorld");
G4VPhysicalVolume *physWorld = new G4PVPlacement(0, G4ThreeVector(0., 0., 0.), logicWorld,
"physWorld", 0, false, 0, true);

// Define a 1cm*1cm*2*cm box
G4Box *solidDetector = new G4Box("solidDetector", 0.005*m, 0.005*m, 0.01*m);
```

For the logical volume, we need to define it both in the MyDetectorConstruction class in the header file and the Construct() function, because we will need to access this logical volume when building sensitive detectors. This is in the header file:

```
private:
    G4LogicalVolume *logicDetector;
```

Let's go back to the Construct() function:

```
logicDetector = new G4LogicalVolume(solidDetector, worldMat, "logicDetector");
```

Now we will place these detector volumes on the surface of the mother volume using a for loop. The position vector of the detector volumes is a bit tricky to understand.

```
for(G4int i = 0; i < 100; i++)
{
    for (G4int j = 0; j < 100; j++)
    {
        G4VPhysicalVolume *physDetector = new G4PVPlacement(0,
        G4ThreeVector(-0.5*m+(i+0.5)*m/100, -0.5*m+(j+0.5)*m/100, 0.49*m), logicDetector,
        "physDetector", logicWorld, false, j+i*100, true);
    }
}
```

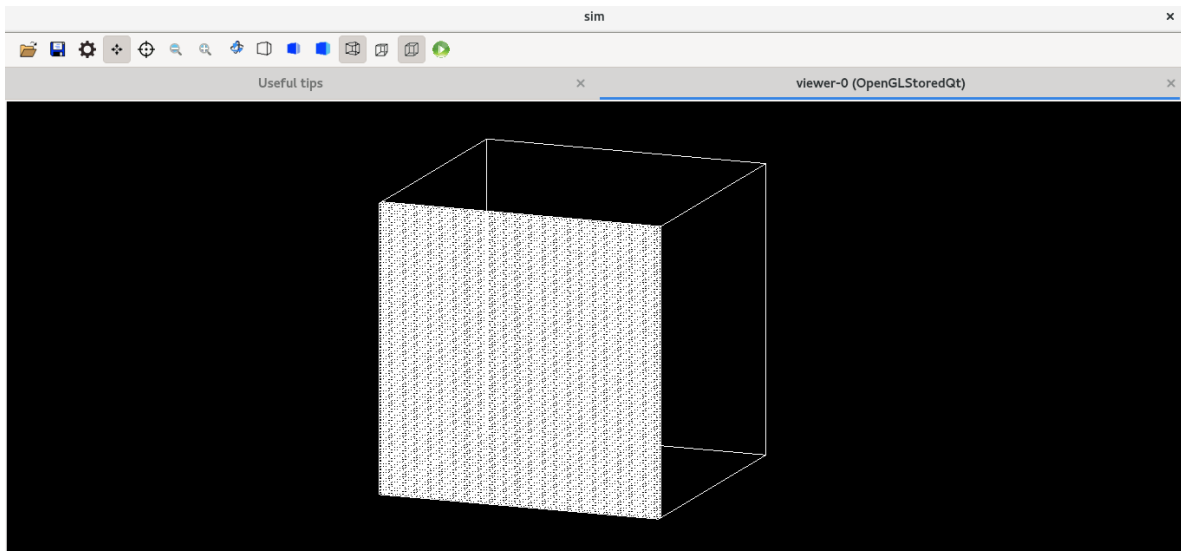


Figure 8: 100x100 detectors on the surface

The next step involves implementing the sensitive detectors within these small volumes. As mentioned earlier, this is achieved by creating a class that inherits from the "G4VSensitiveDetector" class. To accomplish this, we will generate two separate files named "detector.cc" and "detector.hh". Let's start with the header file.

```
#ifndef DETECTOR_HH
#define DETECTOR_HH

#include "G4VSensitiveDetector.hh" // Include the base sensitive detector class
#include "G4RunManager.hh" // Include the run manager class

// Define your custom sensitive detector class
class MySensitiveDetector : public G4VSensitiveDetector
{
public:
    MySensitiveDetector(G4String); // Constructor
    ~MySensitiveDetector(); // Destructor

private:
    // This virtual function is where you define how to process hits
    virtual G4bool ProcessHits(G4Step *, G4TouchableHistory *);
};

#endif
```

"G4Step\*" and "G4TouchableHistory\*" are the parameters of the "ProcessHits" function. It tells the function what information it will receive when it's called.

G4Step\*: This parameter represents the step of the particle, describing things like where it started, where it ended up, its energy, and other properties.

G4TouchableHistory\*: This parameter provides information about the geometry context of the interaction, including details about which volume the interaction occurred in and its position within the geometry hierarchy.

Let's continue with the source file:

```
#include "detector.hh" // Include the header file for this class

// Constructor implementation
MySensitiveDetector::MySensitiveDetector(G4String name) : G4VSensitiveDetector(name)
{
    // Constructor of the base class is called with the provided name
    // This helps initialize the sensitive detector with a unique identifier
}

// Destructor implementation
MySensitiveDetector::~MySensitiveDetector()
{}

// Function to process hits (particle interactions)
G4bool MySensitiveDetector::ProcessHits(G4Step *aStep, G4TouchableHistory *ROhist)
```

Now, we should go back to our construction files and define a function called "ConstructSDandField()". In this function, we set up the sensitive detector and connect with the logical volumes. We need to define this function privately in the MyDetectorConstruction class in the header file construction.hh:

```
private:
    G4LogicalVolume *logicDetector;
    virtual void ConstructSDandField();
```

Lastly, we can implement the detector within the source file "construction.cc". It's important to recognize that this function is distinct from the "Construct()" function present in the "construction.cc" file.

```
void MyDetectorConstruction::ConstructSDandField()
{
    // Create a new instance of the MySensitiveDetector class
    MySensitiveDetector *sensDet = new MySensitiveDetector("SensitiveDetector");

    // Attach the sensitive detector instance to the logical volume logicDetector
    logicDetector->SetSensitiveDetector(sensDet);
}
```

Now, let's revisit our sensitive detector and determine how to handle the recorded hits. This task will be accomplished within the detector's source file, specifically in the ProcessHits function.

```
G4bool MySensitiveDetector::ProcessHits(G4Step *aStep, G4TouchableHistory *ROhist)
{
    // Get the track associated with this interaction
    G4Track *track = aStep->GetTrack();

    // Set the track status to stop and kill
    // This stops further tracking of the particle after it enters the detector
    track->SetTrackStatus(fStopAndKill);

    // Get pre-step and post-step points of the interaction

    // The position where the particle enters the detector
    G4StepPoint *preStepPoint = aStep->GetPreStepPoint();

    // The position where the particle exits the detector
    G4StepPoint *postStepPoint = aStep->GetPostStepPoint();

    // Extract the position of the particle before the interaction
    G4ThreeVector posPhoton = preStepPoint->GetPosition();

    // We can also get the position of the detector

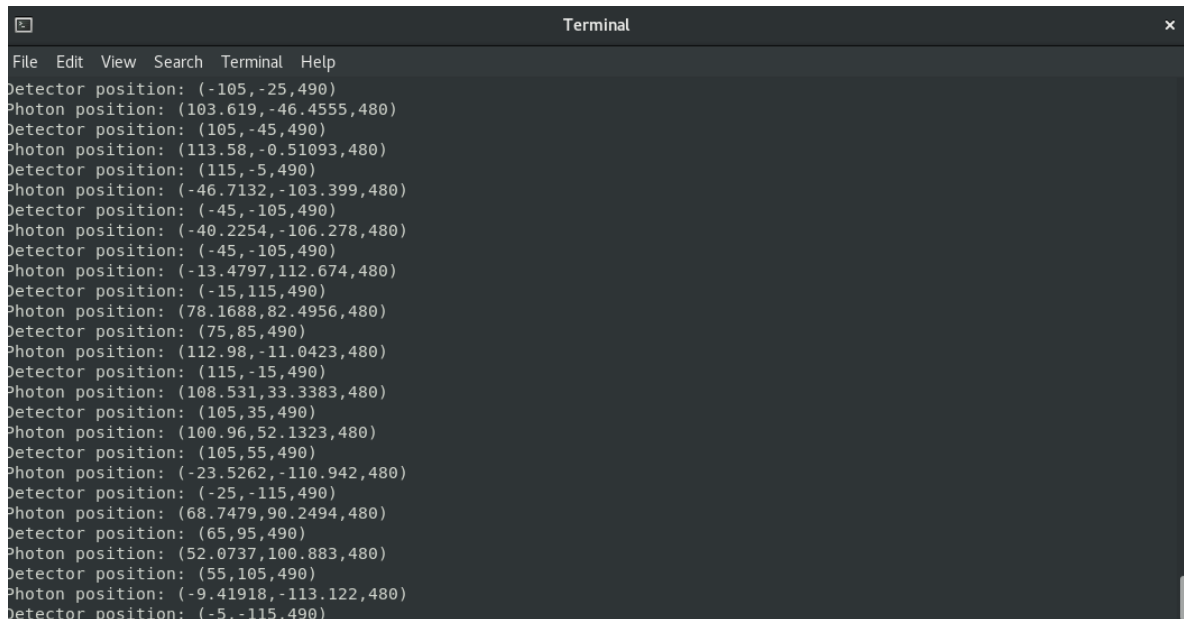
    // Extracting geometry context information for the particle interaction
    const G4VTouchable *touchable = aStep->GetPreStepPoint()->GetTouchable();

    // Extract the copy number associated with the volume
    G4int copyNo = touchable->GetCopyNumber();

    // Get the physical volume where the interaction occurred
    G4VPhysicalVolume *physVol = touchable->GetVolume();

    // Get the translation (position) of the volume in the global coordinate system
    G4ThreeVector posDetector = physVol->GetTranslation();

    // Print out the position of the detector in the global coordinate system
    G4cout << "Detector position: " << posDetector << G4endl;
}
```

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The window displays a list of 24 coordinate pairs, each representing a Cherenkov radiation hit. Each pair consists of a detector position and a photon position, both in parentheses and separated by a comma. The coordinates are numerical values, some with decimal points and some with scientific notation (e.g., 480).

```
Detector position: (-105,-25,490)
Photon position: (103.619,-46.4555,480)
Detector position: (105,-45,490)
Photon position: (113.58,-0.51093,480)
Detector position: (115,-5,490)
Photon position: (-46.7132,-103.399,480)
Detector position: (-45,-105,490)
Photon position: (-40.2254,-106.278,480)
Detector position: (-45,-105,490)
Photon position: (-13.4797,112.674,480)
Detector position: (-15,115,490)
Photon position: (78.1688,82.4956,480)
Detector position: (75,85,490)
Photon position: (112.98,-11.0423,480)
Detector position: (115,-15,490)
Photon position: (108.531,33.3383,480)
Detector position: (105,35,490)
Photon position: (100.96,52.1323,480)
Detector position: (105,55,490)
Photon position: (-23.5262,-110.942,480)
Detector position: (-25,-115,490)
Photon position: (68.7479,90.2494,480)
Detector position: (65,95,490)
Photon position: (52.0737,100.883,480)
Detector position: (55,105,490)
Photon position: (-9.41918,-113.122,480)
Detector position: (-5,-115,490)
```

Figure 9: Position outputs of a Cherenkov radiation

## 10 Storing hits in a ROOT file

ROOT is a widely used framework in particle physics and scientific research for storing, analyzing, and visualizing data. It offers efficient data storage, high-performance input/output, physics libraries, and powerful visualization tools. We can store, analyze and visualize the data of our Geant4 simulations in ROOT. For that purpose, we need to write a class which inherits from the class "G4UserRunAction". It is a Geant4 class that allows you to customize actions that should be performed at the beginning and end of each run (a series of events) in a Geant4 simulation. To accomplish this, we will generate two separate files named "run.cc" and "run.hh". Let's start with the header file.

```
#ifndef RUN_HH
#define RUN_HH

#include "G4UserRunAction.hh" // Include the Geant4 user run action header

// Define the custom run action class inheriting from G4UserRunAction
class MyRunAction : public G4UserRunAction
{
public:
    MyRunAction(); // Constructor declaration
    ~MyRunAction(); // Destructor declaration

    // Begin of run action method declaration
    virtual void BeginOfRunAction(const G4Run*);
    // End of run action method declaration
    virtual void EndOfRunAction(const G4Run*);
};

#endif
```

Let's continue with the source file.

```
#include "run.hh"

MyRunAction::MyRunAction() // Constructor
{}

MyRunAction::~MyRunAction() // Destructor
{}

void MyRunAction::BeginOfRunAction(const G4Run*)
{
    // This method is called at the beginning of each run

    // Get the analysis manager instance
    G4AnalysisManager *man = G4AnalysisManager::Instance();

    // Open an output ROOT file
    man->OpenFile("output.root");

    // Create an ntuple named "Hits" with relevant columns
    man->CreateNtuple("Hits", "Hits");
    man->CreateNtupleIColumn("fEvent"); // Event number
    man->CreateNtupleDColumn("fX"); // X-coordinate
    man->CreateNtupleDColumn("fY"); // Y-coordinate
    man->CreateNtupleDColumn("fZ"); // Z-coordinate
    man->FinishNtuple(0); // Finish defining the ntuple structure
}

void MyRunAction::EndOfRunAction(const G4Run*)
{
    // This method is called at the end of each run

    // Get the analysis manager instance
    G4AnalysisManager *man = G4AnalysisManager::Instance();

    // Write the data to the output ROOT file
    man->Write();

    // Close the output ROOT file
    man->CloseFile();
}
```

At this point, we start to implement it in the program. First thing to do, we need to implement it to the action initialization files. We need to include "run.hh" in the file "action.hh". Then, we go to the function MyActionInitialization::Build() in "action.cc" and add these lines:

```
// Create an instance of the MyRunAction class
MyRunAction *runAction = new MyRunAction();

// Set the created MyRunAction object as a user action for the run
SetUserAction(runAction);
```

Now, the ROOT output file can be created, but it's empty. We will go ahead and fill the file with the data.

To use the analysis manager and run manager, we need to include the header files "g4root.hh" and "G4RunManager.hh" in our detector header file "detector.hh". Then, go back to the ProcessHits function in the source file and implement the root analysis with these lines:

```
// Get the current event ID using Geant4's RunManager and Event classes
G4int evt = G4RunManager::GetRunManager()->GetCurrentEvent()->GetEventID();

// Get the instance of the Analysis Manager
G4AnalysisManager *man = G4AnalysisManager::Instance();

// Fill the ntuple columns with event-specific data
man->FillNtupleIColumn(0, evt);           // Fill the 0th column with event ID
man->FillNtupleDColumn(1, posDetector[0]); // Fill the 1st column with X-coordinate
man->FillNtupleDColumn(2, posDetector[1]); // Fill the 2nd column with Y-coordinate
man->FillNtupleDColumn(3, posDetector[2]); // Fill the 3rd column with Z-coordinate

// Add a row to the ntuple to indicate data for the current event has been filled
man->AddNtupleRow(0);
```

Now, our simulation data will be stored in the file "output.root" in the build folder. To access a ROOT file in the virtual machine, write the command "root output.root". To see the data, you can open the TBrowse in root with the command "new TBrowse".

Here are some data visualizations of a Cherenkov radiation:



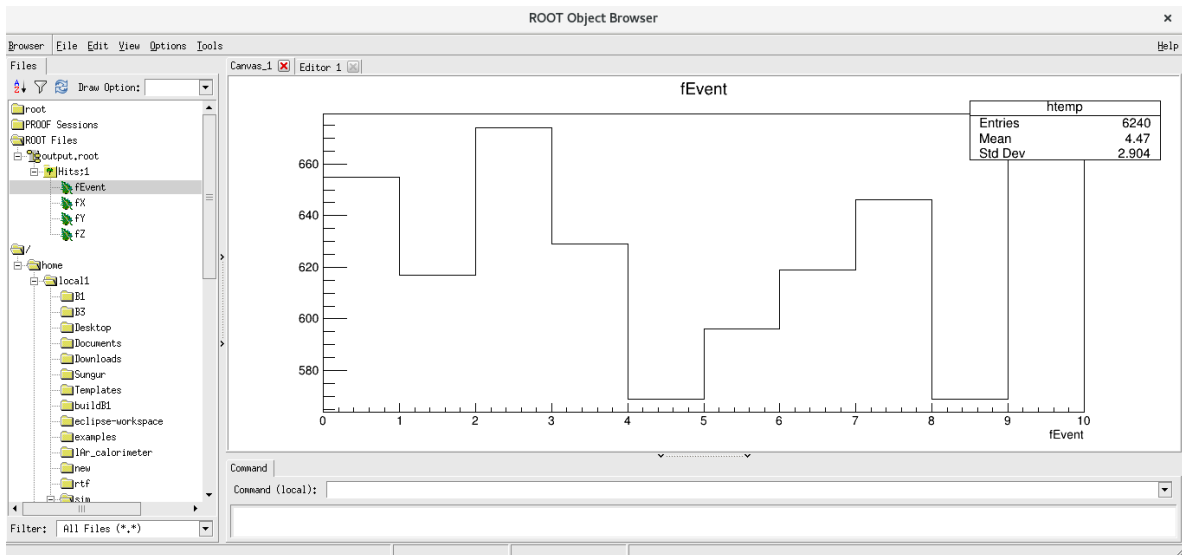


Figure 10: Number of entries for 10 events

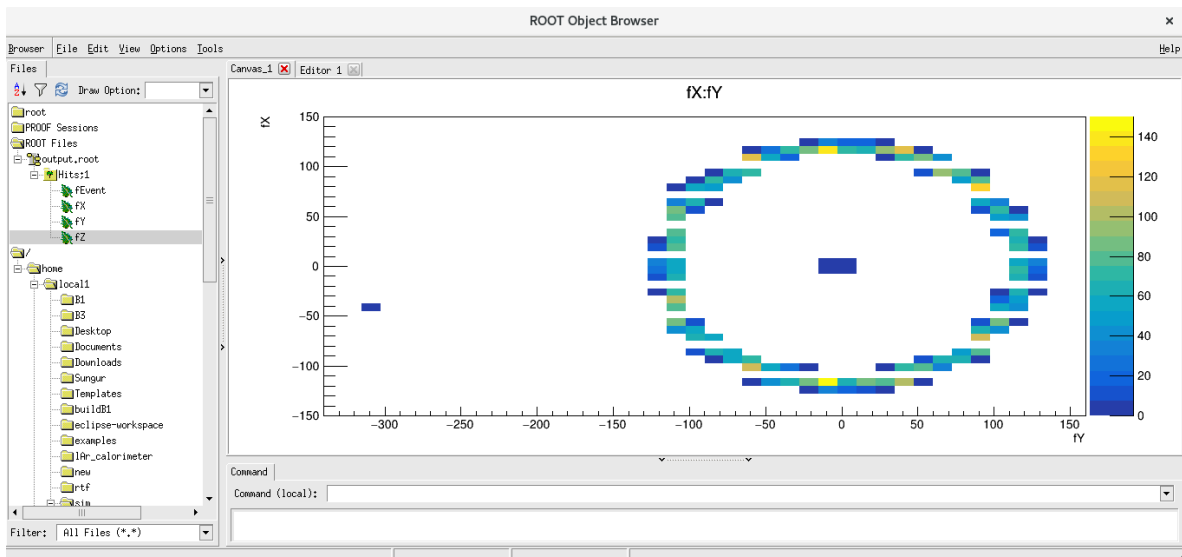


Figure 11: X and Y coordinates

## 11 Adding macro files

Geant4 macro files are text files that contain a series of commands and settings used to control and configure Geant4 simulations. These macro files provide a way to interact with Geant4 without the need to recompile the source code of your simulation. Some common uses of Geant4 macro files include defining the geometry of your experimental setup, setting particle properties, configuring visualization, running simulations, recording data, and creating custom commands or functions for specific tasks or analysis during the simulation.

As an example, let's set up visualization and run macro files. To create a visualization macro, we'll separate the "vis" commands from the main file and save them in a new file named "vis.mac." This allows us to manage visualization settings separately.

```
/vis/open OGL // Open a 3D OpenGL visualization window.

/vis/drawVolume // Display the geometry volumes in the visualization window.

/vis/viewer/set/viewpointVector 1 1 1 // Set the viewpoint vector for the visualization.

/vis/scene/add/trajectories smooth // Add trajectories to the scene with a smooth display style.

/vis/viewer/set/autoRefresh true // Enable automatic refreshing of the visualization.

/vis/scene/endOfEventAction accumulate // Accumulate visualization at the end of each event.

/vis/scene/add/scale 10 cm // Add a 10 cm scale to the visualization scene for reference.

/vis/scene/add/axes // Add axes (typically X, Y, and Z) to show the coordinate system.

/vis/scene/add/eventID // Display the event ID in the visualization for event tracking.
```

Now, we need to execute the macro in the main file by simply adding this:

```
UImanager->ApplyCommand("/control/execute vis.mac");
```

Prior to including the run macro, I will modify the main file to behave differently depending on how it's executed. When executed with the run macro, it will perform the simulation without visualization and provide output data which is called batch mode. If executed without the run macro, it will run in the interactive mode.

For the run macro file, I will keep it simple. But you can add many more things.

```
/gun/momentumAmp 5 GeV // Adjust the energy  
/run/beamOn 100 // Run 100 events
```

To achieve different purposes, use the following commands:

```
./sim // Interactive mode  
./sim run.mac // Batch mode
```

Additionally, relocate the particle gun configuration commands from the `GeneratePrimaries` method to the constructor of `MyPrimaryGenerator` in the "generator.cc" file. This modification makes these settings a part of the codebase while still allowing flexibility for adjustments via the macro file as needed.

The new main file looks like this:

```

#include <iostream>
#include "G4RunManager.hh"
#include "G4UImanager.hh"
#include "G4UIExecutive.hh"
#include "G4VisManager.hh"
#include "G4VisExecutive.hh"

#include "construction.hh" // Include your detector construction header
#include "physics.hh"      // Include your physics list header
#include "action.hh"       // Include your action initialization header

int main(int argc, char** argv)
{
    G4RunManager *runManager = new G4RunManager();

    // Set user-defined initialization classes
    runManager->SetUserInitialization(new MyDetectorConstruction());
    runManager->SetUserInitialization(new MyPhysicsList());
    runManager->SetUserInitialization(new MyActionInitialization());
    runManager->Initialize();

    G4UIExecutive *ui = 0;
    if(argc == 1)
    {
        // Create a user interface if no command line arguments are provided
        ui = new G4UIExecutive(argc, argv);
    }
    G4VisManager *visManager = new G4VisExecutive();
    visManager->Initialize();
    G4UImanager *UImanager = G4UImanager::GetUIpointer();

    if(ui)
    {
        // Execute visualization macro if the user interface is enabled
        UImanager->ApplyCommand("/control/execute vis.mac");
        ui->SessionStart();
    }
    else
    {
        G4String command = "/control/execute ";
        G4String fileName = argv[1];
        // Execute the provided macro file if no user interface is enabled
        UImanager->ApplyCommand(command + fileName);
    }

    return 0;
}

```

## 12 Cherenkov Radiator

This example shows the construction of a Cherenkov detector, its materials and it's nice to understand how to add optical parameters to materials.

```
G4VPhysicalVolume *MyDetectorConstruction::Construct()
{
    // Get the NIST manager for materials
    G4NistManager *nist = G4NistManager::Instance();

    // Define SiO2 material
    G4Material *SiO2 = new G4Material("SiO2", 2.201*g/cm3, 2);
    SiO2->AddElement(nist->FindOrBuildElement("Si"), 2);
    SiO2->AddElement(nist->FindOrBuildElement("O"), 2);

    // Define H2O material
    G4Material *H2O = new G4Material("H2O", 1.000*g/cm3, 2);
    H2O->AddElement(nist->FindOrBuildElement("H"), 2);
    H2O->AddElement(nist->FindOrBuildElement("O"), 1);

    // Find or build the Carbon (C) element
    G4Element *C = nist->FindOrBuildElement("C");

    // Create Aerogel material
    G4Material *Aerogel = new G4Material("Aerogel", 0.200*g/cm3, 3);
    Aerogel->AddMaterial(SiO2, 62.5*perCent);
    Aerogel->AddMaterial(H2O, 37.4*perCent);
    Aerogel->AddElement(C, 0.1*perCent);

    // Define optical properties
    G4double energy[2] = {1.239841939*eV/0.2, 1.239841939*eV/0.9};
    G4double rindexAerogel[2] = {1.1, 1.1};
    G4double rindexWorld[2] = {1.0, 1.0};
    G4double rindexWater[2] = {1.33, 1.33};

    // Create material properties table for Aerogel
    G4MaterialPropertiesTable *mptAerogel = new G4MaterialPropertiesTable();
    mptAerogel->AddProperty("RINDEX", energy, rindexAerogel, 2);

    // Set material properties for Aerogel
    Aerogel->SetMaterialPropertiesTable(mptAerogel);

    // Define the world material as G4_AIR
    G4Material *worldMat = nist->FindOrBuildMaterial("G4_AIR");
```

```

// Create material properties table for water
G4MaterialPropertiesTable *mptWater = new G4MaterialPropertiesTable();
mptWater->AddProperty("RINDEX", energy, rindexWater, 2);

// Set material properties for H2O (water)
H2O->SetMaterialPropertiesTable(mptWater);

// Create material properties table for the world material
G4MaterialPropertiesTable *mptWorld = new G4MaterialPropertiesTable();
mptWorld->AddProperty("RINDEX", energy, rindexWorld, 2);

// Set material properties for the world material
worldMat->SetMaterialPropertiesTable(mptWorld);

G4Box *solidWorld = new G4Box("solidWorld", 0.5*m, 0.5*m, 0.5*m);

// Create a logical volume for the world
G4LogicalVolume *logicWorld = new G4LogicalVolume(solidWorld, worldMat, "logicWorld");

// Place the world volume
G4VPhysicalVolume *physWorld = new G4PVPlacement(0, G4ThreeVector(0., 0., 0.),
logicWorld, "physWorld", 0, false, 0, true);

// Create a solid box for the radiator
G4Box *solidRadiator = new G4Box("solidRadiator", 0.4*m, 0.4*m, 0.01*m);

// Create a logical volume for the radiator
G4LogicalVolume *logicRadiator = new G4LogicalVolume(solidRadiator, Aerogel, "logicalRadiator")

// Place the radiator volume
G4VPhysicalVolume *physRadiator = new G4PVPlacement(0, G4ThreeVector(0., 0., 0.25*m),
logicRadiator, "physRadiator", logicWorld, false, 0, true);

// Create a solid box for the detector
G4Box *solidDetector = new G4Box("solidDetector", 0.005*m, 0.005*m, 0.01*m);

// Create a logical volume for the detector
logicDetector = new G4LogicalVolume(solidDetector, worldMat, "logicDetector");

```

```

    // Place the detector volume in a loop
    for(G4int i = 0; i < 100; i++)
    {
        for (G4int j = 0 ; j < 100; j++)
        {
            G4VPhysicalVolume *physDetector = new G4PVPlacement(0,
            G4ThreeVector(-0.5*m+(i+0.5)*m/100, -0.5*m+(j+0.5)*m/100, 0.49*m),
            logicDetector, "physDetector", logicWorld, false, j+i*100, true);
        }
    }
    return physWorld;
}

```

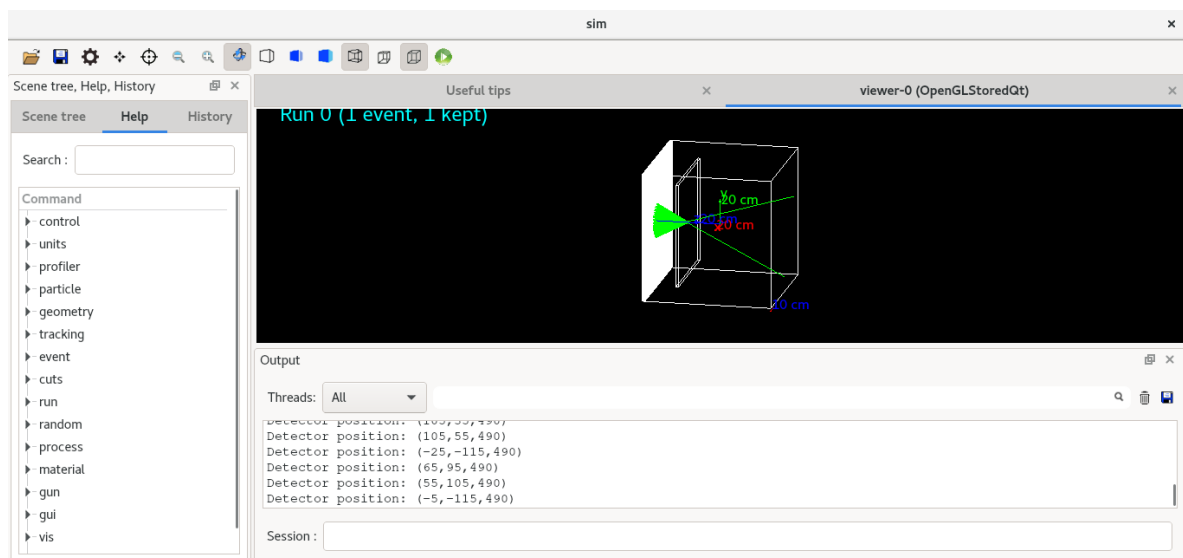


Figure 12: Cherenkov radiation

To explore additional examples and gain a deeper understanding of various concepts, please refer to the "examples" folder as previously mentioned. Within this folder, you'll find a diverse range of both fundamental and advanced examples. To better comprehend the purpose and learning objectives of each example, I recommend reviewing the accompanying README files associated with each one. These README files provide valuable insights into the content and educational opportunities offered by each example.