Useful Geant4 Information

Benjamin Derieg

August 10, 2022

Introduction and Assumed Knowledge

My goal is to provide some background knowledge on Geant4 because its documentation is a little vague. However, I assume that the reader has at least a basic knowledge of C++ (including inheritance and how to use cmake). No knowledge of Geant4 is needed other than knowing it's a particle physics simulation program where the user implements a provided set of abstract classes to accomplish his/her goals. When doing so, it's easiest to start from an example, but even these examples can sometimes be quite confusing. (If the reader is developing a Geant4 application from scratch, I advise the use of these[1] videos by Dr. Mustafa Schmidt.)

Here, I will refer to the ICRP110_HumanPhantoms application for examples, but any example will probably do. I will make some changes to this ICRP110_HumanPhantoms example, and my modified code can be found on GitHub[2].

How Geant4 Works

A Geant4 application consists of a hierarchy of nested executions. I'll list them in order of most broad to most specific here:

- **Run:** A run is where some particles are generated and they interact with the world until no more interactions are to be had. A Geant4 application could (and probably does) consist of multiple runs. An application's main function must specify a run manager, which tells Geant4 what it should do on each run. To do this, the run manager calls a user-implemented child of the G4VUserActionInitialization class. The run manager also must specify the physics list to be used (which physics processes Geant4 should consider), and should call a user-implemented child of the G4VUserDetectorConstruction class (this tells Geant4 how to construct the world).
- **Event:** An event is a set of all the interactions produced by one primary particle generated by the General Particle Source. Nothing needs to be done here—Geant4 automatically handles everything—but useful information could be accessed in other classes if desired.
- **Step:** A step is any interaction (i.e., Geant4 lets the physics run until there is some interaction, where a step happens). Once again, nothing needs to be done here—Geant4 will handle things on its own. However, useful information can be extracted from a step (in other classes) such as the particle types, momenta, energies, etc.
- **Track:** While not really a 'process,' it should be noted that each particle is tracked with a G4Track object. This information can be extracted from the G4Step class.

Modification Example

Starting from the ICRP110_HumanPhantoms, let's work to build a spherical spacecraft around the existing phantom, then implement a multifunctional detector in the phantom which can detect hits and give us much more control over what kind of output we produce (the goal is to output a file with: the primary particle from a given event, the initial energy of this particle, and the absorbed dose by all secondaries).

[1] https://www.youtube.com/watch?v=Lxb4WZyKeCElist=PLLybgCU6QCGWgzNYOV0SKen9vqg4KXeVLindex=1
[2] https://github.com/bderieg/GCRSimulator

Create a Spherical Aluminum Shield

Remember that world construction is done in the user-implemented child of the G4VUserDetectorConstruction class (in this case the ICRP110PhantomConstruction) class. Specifically, the virtual Construct function is called by Geant4 to construct the world.

The current structure of the ICRP110PhantomConstruction::Construct function is the creation of a mandatory world volume, followed by the construction of the phantom which is placed inside the world volume. To construct an aluminum sphere around the phantom, we similarly just have to create it and then place it into the existing world volume.

Before we start let's define a few useful variables. At the end of the Construct function, we define:

```
G4double shieldInnerRadii = 1.54*m;
G4double shieldOuterRadii = 1.55*m;
G4double pi = 3.14159265358979323846;

G4NistManager* nist = G4NistManager::Instance();
G4Material* shieldMat = nist -> FindOrBuildMaterial("G4_Al");
```

Notice that Geant4 defines its own basic types (i.e., G4double). One advantage to this is that when we're using the Geant4 units class, we can assign units to our variables (here we defined the radii as having units of meters. In the last two lines, we use Geant4's built in function to get common materials, such as aluminum.

Now let's construct the shield itself. This consists of three steps (each involving a different type of Geant4 object). First, we define the geometry of the shield (of course, the meaning of all these arguments can be found in the documentation):

```
G4Sphere* shield = new G4Sphere("shield", shieldInnerRadii, shieldOuterRadii,
                                0, 2*pi, 0, pi);
```

Now we create a G4LogicalVolume from this geometry. This actually creates the object such that it can be used in various ways, but does not yet place it in the physical world:

```
logicShield = new G4LogicalVolume(shield, shieldMat, "logicShield", 0, 0, 0);
```

Notice also that we do not define logicShield here. You should define it as a private variable in the header file so that it can be accessed by other functions; this will come in handy later.

Now we can define a G4VPhysicalVolume, which actually places our definition in the world to interact with particles (notice the argument, logicWorld, which places the shield in the world):

```
G4VPhysicalVolume* physShield = new G4PVPlacement(0, G4ThreeVector(),
                                                  logicShield, "physShield",
                                                  logicWorld, false, 0);
```

Yay! There's now a sphere around the human.

Attach a Dose Counter to the Phantom

The existing scoring functionality defined in macro files is good, but doesn't have quite the control we might want. We can only tally the specific quantities offered by the basic scoring manager. Therefore, we here create a new class which will interface with the phantom; then, in the run action classes, we can analyze the results from each run and store whatever information we desire (in this case the dose provided by each generated particle).

Creating the Detector

The first step is to actually create the detector and attach it to the phantom. There are two types of detectors we could use, each with distinct advantages and disadvantages, but here we will use the G4MultiFunctionalDetector. Fortunately, this class is already packaged and ready to go.

Again, in the ICRP110PhantomConstruction class, we implement the virtual ConstructSDandField function. This is called by Geant4 in the construction phase to create the detector. Inside of this function, we first create the detector:

```
G4MultiFunctionalDetector* phantomDetector =
                new G4MultiFunctionalDetector("phantomDetector");
```

Then we add it to the detector manager so that Geant4 knows it exists:

```
G4SDManager::GetSDMpointer()->AddNewDetector(phantomDetector);
```

And we attach it to the phantom's logical volume:

```
logicVoxel->SetSensitiveDetector(phantomDetector);
```

Now, to tell Geant4 which sort of quantity we want to score, we create a scorer and attach it to the detector:

```
G4VPrimitiveScorer* doseCounter = new G4PSDoseDeposit("doseCounter");
phantomDetector->RegisterPrimitive(doseCounter);
```

Record Events and Store Information

We now need to create two new classes which will process and store information when an event occurs. The first of these is an implementation of the G4Run class (let's call it ICRP110PhantomRun). In the constructor for this class, let's get the ID of the detector we want to use (note that it is a global variable so we can use it later):

```
G4SDManager* SDM = G4SDManager::GetSDMpointer();
totalDoseID = SDM->GetCollectionID("phantomDetector/doseCounter");
```

Now let's implement the virtual RecordEvent(const G4Event* evt) function to define what to do on an event. Inside this function, first let's get the hits map from the current event:

```
G4HCofThisEvent* HCE = evt->GetHCofThisEvent();
eventTotalDose = (G4THitsMap<G4double>*)(HCE->GetHC(totalDoseID));
```

This hits map contains dose information and was populated by the detector (note that eventTotalDose is global). We also get information on the primary particle, which we stored earlier:

```
G4String eventPrimaryName = evt->GetPrimaryVertex()->GetPrimary()
                            ->GetParticleDefinition()->GetParticleName();
G4double eventPrimaryKE = evt->GetPrimaryVertex()->GetPrimary()
                            ->GetKineticEnergy();
```

Now we accumulate all the dose from this hits map into one variable:

```
std::map<G4int, G4double*>* eventDoseMap = eventTotalDose->GetMap();
std::map<G4int, G4double*>::iterator itr;
G4double newDose = 0.0;
for (itr = eventDoseMap->begin(); itr != eventDoseMap->end(); itr++) {
    newDose += *(itr->second);
}
```

This information could obviously be stored in various ways, but here we store it as a map where the keys are pairs of the primary name and kinetic energy, and the values are the total dose from the event:

```
std::pair<G4String, G4double> keyPair;
keyPair.first = eventPrimaryName;
keyPair.second = eventPrimaryKE;
totalDoses[keyPair] += newDose;
```

A function can then also be made to retrieve this map. Of course, we need to make a class now to actually retrieve and store it. This can be done with an implementation of the G4UserRunAction class (let's call it ICRP110PhantomRunAction). Here, we first implement the mandatory virtual GenerateRun() function. Just return a run:

```
return (new ICRP110PhantomRun());
```

Now we implement the virtual EndOfRunAction(const G4Run* aRun) function. Inside the function, we first get the run and retrieve the information we stored about the events therein:

```
ICRP110PhantomRun* theRun = (ICRP110PhantomRun*)aRun;
std::map<std::pair<G4String, G4double>, G4double> totalDoses = theRun
                                                    ->GetDoseDeposits();
```

Now we can just iterate over this map and output it to a file in whatever format is desired. Congratulations! We should now have a file with (for all events) the primary particle name, its kinetic energy, and the total dose deposited by all its secondaries.

IMPORTANT NOTE: The default unit IS NOT gray. It needs to be converted to gray before outputting. A G4double can be converted to a different unit by dividing by the desired unit.

Modify the Dose Counter

This is great, but does not work very well for a voxelized phantom because the G4PSDoseDeposit class calculates the dose with respect to the voxel, which is not what we want (we want to calculate the dose with respect to the organ). To do this, we first have to pass information about the organ in the event.

In the construction class, notice the method called ReadPhantomData. This populates a size_t for the material IDs. We need to do the same thing for the organ IDs. So declare a new variable called size_t fOrganIDs in the header. Then, near the end of the ReadPhantomData method, create this with:

```
fOrganIDs = new size_t[fNVoxels];
```

To populate this, look at the ReadPhantomDataFile method. Next to where mateID_out is declared, declare another variable called G4int organID_out. Set this to have the value of OrgID. Then at the end of the method, set:

```
fOrganIDs[nnew] ]= organID_out;
```

Back in the construct function, near the end, set:

```
param -> SetOrganIndices(fOrganIDs);
```

This calls a method which we will now make in the ICRP110PhantomNestedParameterisation() class.

In the ICRP110PhantomNestedParameterisation() class, make this new method and in it put (also declaring a private global variable called fOrganIndices):

```
fOrganIndices = orgInd;
```

Also make an accessor for this called GetOrganIndex(G4int copyNo) that just returns fOrganIndices[copyNo].

Then, at the end of the ComputeMaterial method here, put

```
mate -> SetName(std::to_string(GetOrganIndex(copyID)));
```

This will change the name of the material to the organ index number.

Now we must make a new PrimitiveScorer. Let's just modify the code from the built-in G4PSDoseDeposit class. Make a copy in your own directory or something and call it G4PSDoseDepositMod. Replace the code in ProcessHits with:

```
G4double edep = aStep->GetTotalEnergyDeposit();
if ( edep == 0. ) return FALSE;

G4double orgID = std::stoi(aStep->GetPreStepPoint()->GetMaterial()->GetName());
G4double organMass = IDToMass(orgID);
if (organMass == 0.) return FALSE;

G4double dose = 0.0;
dose = edep / (organMass*g);

G4int index = GetIndex(aStep);
EvtMap->add(index,dose);

return TRUE;
```

The function call IDToMass should just be a function also in this class that converts organID to mass (these can be found in build/ICRPdata/OrganMasses.dat), like so:

```
G4double G4PSDoseDepositMod::IDToMass(G4int id) {
        if (id==0) { return 0; }
```

```
        if (id==1) { return 7; }
        if (id==2) { return 7; }
        if (id==3) { return 11.03; }
        if (id==4) { return 28.41; }
....
```

Converting to Equivalent Dose

Instead of outputting raw dose, we could also output equivalent dose. First, in the G4PSDoseDepositMod class, make the following function to give the relative biological effectiveness of some step:

```
G4double G4PSDoseDepositMod::GetRBE(G4Step* aStep) {
        G4double e = 2.718281828459045;

        G4String particleName = aStep -> GetTrack() -> GetParticleDefinition()
                              -> GetParticleName();
        G4double KE = aStep -> GetPreStepPoint() -> GetKineticEnergy();
        G4int baryonNumber = aStep -> GetTrack() -> GetParticleDefinition()
                              -> GetBaryonNumber();

        if (particleName == "neutron") {
                if (KE < (1*MeV)) {
                        return ( 2.5 + ( 18.2*
                            pow(e,((-pow(std::log(KE),2))/6)) ) );
                } else if (KE < (50*MeV)) {
                        return ( 5.0 + ( 17.0*
                            pow(e,((-pow(std::log(2*KE),2))/6)) ) );
                } else {
                        return ( 2.5 + ( 3.25*
                            pow(e,((-pow(std::log(0.04*KE),2))/6)) ) );
                }
        } else if ((particleName == "proton")
                        || (particleName == "pi+")
                        || (particleName == "pi-")) {
                return 2.0;
        } else if (baryonNumber > 1) {
                return 20.0;
        } else {
                return 1.0;
        }
}
```

Now in the ProcessHits function, just multiply by this factor for each step.

Record All Energy Deposit

Another thing we might want to do is output all locations of energy deposit to a file for every hit inside the phantom. This is most easily done with a sensitive detector.

Creating the Sensitive Detector

First we have to declare a sensitive detector in the ConstructSDandField() method like the other detector. This is done like so:

```
ICRP110PhantomDetector* phantomSensitiveDetector =
                new ICRP110PhantomDetector("SensitiveDetector");
G4SDManager::GetSDMpointer() -> AddNewDetector(phantomSensitiveDetector);
logicVoxel -> SetSensitiveDetector(phantomSensitiveDetector);
```

Processing Hits

Now we must implement our own detector class, derived from the G4VSensitiveDetector class. This has a virtual method called ProcessHits(G4Step*, G4TouchableHistory*). Inside this method, a line can be output to a file for each hit after accessing information from the given step for each hit. This method is called internally by Geant4 on every hit.

Add Messengers

Instead of editing the source code every time we want to change something, we can add a messenger so that it takes a command from a macro file and we don't have to recompile. This is done by first adding a G4GenericMessenger to some header file. For example:

```
G4GenericMessenger* outputMessenger;
```

Then, in the constructor, we create it (here, for example, "/output/" is the folder for the messenger and "Run Action" is the description):

```
outputMessenger = new G4GenericMessenger(this, "/output/", "Run Action");
```

Then we just set the messenger to have some property that can be declared in the macro file. It automatically detects the variable type depending on what is passed (here, for example, "primariesFileName" is the name which will appear in the macro file, the second argument is the variable to use, and the third argument is the description):

```
outputMessenger -> DeclareProperty("primariesFileName", primariesFileName,
                                   "Name of output file for primaries");
```

This variable can then be used later with the variable value passed from the macro. In the macro file, we do something like this:

```
/output/primariesFileName a_file_name.dat
```

Notes

- The way this example works is to output one line a bunch of times to a file throughout the running of the simulation. Don't forget to delete files at the beginning of the application so files aren't appended to from previous simulations.
- Any questions contact Ben Derieg (bderieg@deriegfamily.com)