Task 4c Geant4 native scoring (multi-functional detectors)

This task will show how to retrieve basic physics properties using multi-functional detector and primitive scorers, as well as how to use histograms for analysis/data storage.

INTRODUCTION

Command-based scoring is simple to use but its biggest strength (possibility to acquire data in grids independent of detector description) brings also certain disadvantages - it's not easy to setup the dimensions correctly; and for complex volume geometries, it's virtually impossible.

Another approach, based on sensitive detectors, allows to collect information in the logical volumes themselves. Task 4c shows a light-weight approach to this: G4MultiFunctionalDetector, a sensitive detector class already included in Geant4.

Getting prepared

In this task, you will use the same source code as in tasks 4a and 4b. Therefore, make sure that it compiles and runs correctly.
Important: please **make sure to comment out** the line `G4ScoringManager::GetScoringManager();` in main.cc, which you added for task4b. Also **check that you have commented out the particle killing** in the StackingAction class, otherwise you will obtain results (numerically) different from expected.

PRACTICE

Exercise 4c.1: Assign multi-functional detector

In this exercise, you will create a multi-functional detector and primitive scorers in DetectorConstruction::ConstructSDandField() method in order to score energy in different layers of the calorimeter. In a way, it's repeating what we have already seen in task 4.b. However, you will realize that detailed coordinate knowledge won't be required. How to do that:

1) Create two G4MultiFunctionalDetector instances in DetectorConstruction::ConstructSDandField() - remember that a sensitive detector is always assigned to a logical volume. We have two such volumes, separately for absorber and scintillator layers. So, we also need two detectors.

2) Create two G4PSEnergyDeposit instances (as primitive scorers, again for both layer types) and register them with the corresponding multi-functional detector. Use detector's RegisterPrimitive method.

3) Assign the sensitive detectors to their appropriate volumes. Luckily, method SetSensitiveDetector() allows to do that using a logical volume name (a string, not a variable name!) and a pointer to the detector instance.

4) Register the instances of the G4MultiFunctionalDetector to the SensitiveDetectorManager, by using its method AddNewDetector(). For instance:

```
 G4SDManager* sdManager = G4SDManager::GetSDMpointer();
 sdManager->AddNewDetector(...);
```

Try to compile and run the code.

If everything worked fine, you should see output like this:

```
G4SDManager::AddNewCollection : the collection <absorber/energy> is registered at 1
New sensitive detector <absorber> is registored at /
G4SDManager::AddNewCollection : the collection <scintillator/energy> is registered at 2
New sensitive detector <scintillator> is registored at /
```

Exercise 4c.2: Retrieve and digest the hit collection

Unfortunately, the data collected from the scorers are not automatically processed and stored anywhere. They are temporarily kept as "hit collections" (or "maps") in the G4HCofThisEvent object assigned to an event until the event ends and a new one is started.

We will need to process the hits at the end of each event. To enable this, first add proper `#include` directive (of EventAction.hh) in ActionInitialization.cc and add an instance of EventAction in ActionInitialization::Build()

```
SetUserAction(new EventAction());
```

Now, how do you get to those hits? Find the definition of the EventAction::EndOfEventAction method and update it in the following steps:

1) Get the G4HCofThisEvent from the G4Event instance, using the GetHCofThisEvent() method.

2) Retrieve the collection IDs which are associated with the hits collections, i.e. fAbsorberId and fScintillatorId. G4SDManager enables this using the GetCollectionID method (we have already prepared a pointer to the manager, called sdm). In order to retrieve the collection ID, you need to know the names (= strings) of the two collections: what are their names? (Hint: look at exercise 4c.1) Since the collection IDs are the same throughout a run, you can retrieve it only once (i.e. in the first event).
Notice that the variables fAbsorberId and fScintillatorId are defined in EventAction.hh and they are initialized to -1.

3) Get the hits collection with the given ID (store it in local variable hitMapA), by looking into the G4HCofThisEvent object. Individual hits-collection are retrieved as G4HCofThisEvent::GetHC(id). Unfortunately, this interface returns a pointer to the base type of all hit collections and therefore, we have to cast this pointer to G4THitsMap<G4double>*, like this:

```
G4THitsMap<G4double>* hitMapA = dynamic_cast<G4THitsMap<G4double>*>(...insert code to obtain the HC...)
```

4) Look how we loop over the hit map. The map behaves like std::map and therefore, it is possible to iterate over its key-value pairs using the range-for loop. Inside such loop, the key is always stored as pair.first, the value in pair.second.

Try to compile and run the code. Does it work?

If successful, you should see something like this in the output:

```
EventAction: absorber energy scorer ID: 0
EventAction: scintillator energy scorer ID: 1
```

Exercise 4c.3: Write results in a histogram

Finally, we would like to write the data somewhere. We have 20 layers in total and for each of them, information about the deposited energy. Although this is not a typical case for histograms, technically we can easily use them to quantify what we obtained.

We will use the g4analysis tools of Geant4. Most of the code we have already prepared for you. Therefore, your task is very easy:

1) Create the histogram in the constructor RunAction::RunAction(). The analysis manager offers a method CreateH1 that requires 5 parameters: name (set it to "eDep"), display title, number of bins (this will be 20), lower limit (should be 50, in centimetres) and upper limit (should be 60).
Notice: the unit of measurements in Geant4 are just aliases to convert numbers into the internal default units. For instance, "* cm" means "* 10", as the default length unit is millimeters. If you use the unit of measurement in the `CreateH1()`, you will eventually get an histogram between 500 and 600: this could be still ok (provided that you fill it consistently), but it is not what we want here!. So, you should give no units of measurements when calling `CreateH1()` (= histogram bins will in cm).

Also, uncomment the four lines initializing the analysis manager and the last line in the constructor that opens a file to be used as output.

2) Fill the histogram in EventAction::EndOfEventAction. Find the loops over hitMapA and hitMapS and in both(!) of them, update the histogram with the position (we already calculated this for you) and the weight equal to the deposited energy. Use the analysis object and its `FillH1` method. The number calculated here

```
 G4double x = 50.75 + pair.first;   // already in cm
```

is already in cm, so you don't need to use the unit of measurement in `FillH1()`.

Notice: in such a way only one histogram is created, containing simultaneously the energy released in the scintillator and in the absorber (one bin per layer).

3) Technical detail #1: you have to write the analysis objects, by uncommenting the two specific lines in the RunAction::~RunAction destructor

4) Technical detail #2: close the output file at the end of the application by uncommenting the two relevant lines in the main.cc (look for 4c.3 in the source code).

Notice: When running in sequential mode (but not in MT), the technical steps (3) and (4) (Write + CloseFile) could be done together in the main.cc. This does not work in MT-mode, as the Write() has to be called by all threads.

Compile and run. Does it work?

If yes, you should see a task4.root file in your build directory. You can either open it in ROOT yourself, or run the script:

```
% root calorimeter_histogram.C
```

Notice: the script expects to find a file called `task4.root` in the same directory and containing an histogram called `eDep`. Edit the script accordingly, if this is not the case.

If you have many runs within the same execution (i.e. many `/run/beamOn`), results will be accumulated in the same histogram.

Exercise 4c.4: Change the format output to CSV (optional)

Look into Analysis.hh and see that we use the G4RootAnalysisManager class as an alias for the G4AnalysisManager. The g4analysis package was written with universality in mind so you are not stuck to ROOT (or to any specific) output format, which might or might not be suitable for your analysis or project requirements. Just by changing the definition of the alias

```
using G4AnalysisManager = ...
```

you can switch to a different format: CSV (G4CvsAnalysisManager) or XML (G4XMLAnalysisManager).

Try switching from G4RootAnalysisManager to G4CvsAnalysisManager (do not forget the #includes). Then compile, run and look what happened. You should see that there is a task4_h1_eDep.csv file - in human-readable format.

If you have Python with matplotlib and numpy installed on your computer (it is also the case of the virtual machine), you can visualize the results from CSV using another script:

```
% python calorimeter_histogram.py
```

(Notice: also in this case, the file name `task4_h1_eDep.csv` is built in the assumption that the histogram is called `eDep`)

Don't forget to switch back to ROOT format once you're finished.

SOLUTIONS

Exercise 4c.1

```
// In DetectorConstruction.cc
#include "G4VPrimitiveScorer.hh"
#include "G4PSEnergyDeposit.hh"

// In DetectorConstruction::ConstructSDandField
G4MultiFunctionalDetector* absorberDetector = new G4MultiFunctionalDetector("absorber");
G4MultiFunctionalDetector* scintillatorDetector = new G4MultiFunctionalDetector("scintillator");

G4VPrimitiveScorer* absorberScorer = new G4PSEnergyDeposit("energy");
absorberDetector->RegisterPrimitive(absorberScorer);
G4VPrimitiveScorer* scintillatorScorer = new G4PSEnergyDeposit("energy");
scintillatorDetector->RegisterPrimitive(scintillatorScorer);

SetSensitiveDetector("absorber", absorberDetector);
SetSensitiveDetector("scintillator", scintillatorDetector);
sdManager->AddNewDetector(absorberDetector);
sdManager->AddNewDetector(scintillatorDetector);
```

Exercise 4c.2

```
// In EventAction::EndOfEventAction
G4HCofThisEvent* hcofEvent = event->GetHCofThisEvent();

// ...

if (fAbsorberId < 0)
{
    fAbsorberId = sdm->GetCollectionID("absorber/energy");
    G4cout << "EventAction: absorber energy scorer ID: " << fAbsorberId << G4endl;
}
if (fScintillatorId < 0)
{
    fScintillatorId = sdm->GetCollectionID("scintillator/energy");
    G4cout << "EventAction: scintillator energy scorer ID: " << fScintillatorId << G4endl;
}

// ...

if (fAbsorberId >= 0)
{
    G4THitsMap<G4double>* hitMapA = dynamic_cast<G4THitsMap<G4double>*>(hcofEvent->GetHC(fAbsorberId));

// ...

if (fScintillatorId >= 0)
{
    G4THitsMap<G4double>* hitMapS = dynamic_cast<G4THitsMap<G4double>*>(hcofEvent->GetHC(fScintillatorId));
```

Exercise 4c.3

```
// In RunAction::RunAction
analysisManager->CreateH1("eDep","depositedEnergy", 20, 50, 60);

// In EventAction::EndOfEventAction
analysis->FillH1(histogramId, x, energy / keV);

// ...
analysis->FillH1(histogramId, x, energy / keV);
```

Exercise 4c.4

```
// In Analysis.hh
#include <G4CsvAnalysisManager.hh>
//using G4AnalysisManager = G4RootAnalysisManager;
using G4AnalysisManager = G4CsvAnalysisManager;
```
