Zastosowanie pakietu Geant4 w fizyce jądrowej Wykład 6

Aleksandra Fijałkowska

22 listopada 2018

Ogłoszenia

Przypominam, że zajęcia w dniach 23 i 29 listopada NIE ODBĘDĄ się.
????????

Tworzenie własnego **Sensitive Detectors**

Proste liczniki (**Primitive Scorers**) są wbudowaną w bibliotekę Geant4 prostą implementacją **sensitive detectors**. Jeden licznik potrafi zliczać jeden, konkretny wynik (np. depozyt energii). Często chcemy jednak zbadać bardziej skomplikowane cechy detektora i symulacji (np chcemy jednocześnie depozyt energii, czas oddziaływania, typ cząstki itp).
W takiej sytuacji musimy stworzyć kawałek kodu, który po pierwsze wydobędzie te dane a po drugie umieści je w jakimś rozsądnym obiekcie (wyniki działania prymitywnych liczników były zwykłymi liczbami zmiennoprzecinkowymi typu **double**).

Będziemy potrzebować dwóch typów klas:

- klasę wywiedzioną z klasy bazowej **G4VSensitiveDetector**, zawierającą implementację czysto wirtualnej metody **G4bool ProcessHits(G4Step*aStep,G4TouchableHistory*ROhist)** (w niej wyciągamy interesujące wyniki symulacji)
- klasę wywiedzioną z klasy **G4VHit**, w której będziemy trzymać wyniki symulacji

G4VSensitiveDetector przypomnienie

- Implementując klasę **G4VSensitiveDetector** oraz czysto wirtualną metodę **G4bool ProcessHits(G4Step*aStep,G4TouchableHistory*ROhist)**.
  Warto zwrócić uwagę, że metoda **ProcessHits** również wykorzystuje fakt dostępu do obiektu **G4Step** oraz wszystkich jego publicznych metod, więc od strony implementacji metody **UserSteppingAction(const G4Step*)** i **G4bool ProcessHits(G4Step*aStep,G4TouchableHistory*ROhist)** są dość podobne.
  Interesujące dane otrzymane z kroku są zapisywane w klasie dziedziczącej po **G4VHit**, zastosowanie tej metody wymaga zaimplementowana zarówno **G4VSensitiveDetector**, jak i **G4VHit**.
  Zwyczajowo tworzy się dwie klady o zbliżonej nazwie, np. **NaISD** i **NaIHit**.
  Dostęp do hitów (umieszczonych jako zbiór w **G4THitsCollection**) uzyskuje się w klasie implementującej **G4UserEventAction** (w naszym projekcie nazywa się **EventAction**, w metodzie **void EndOfEventAction(const G4Event*)**.
  Wymogiem wykorzystania **G4THitsCollection** jest zadeklarowanie **G4Allocators** dla implementacji klasy **G4VHit**, a także operatora **new()** i **delete()**.

Sposób implementacji 2

Fragment pliku nagłówkowego klasy G4VSensitiveDetector:
class description:
This is the abstract base class of the sensitive detector. The user's sensitive detector which generates hits must be derived from this class. In the derived class constructor, name(s) of hits collection(s) which are made by the sensitive detector must be set to „collectionNameśtring vector.

```
class G4VSensitiveDetector
{
  public:
      G4VSensitiveDetector(G4String name);
      G4VSensitiveDetector(const G4VSensitiveDetector &right);
      // Constructors. The user's concrete class must use one of these
      // constructors by the constructor initializer of the derived class.
      // The name of the sensitive detector must be unique.

      virtual ~G4VSensitiveDetector();
      virtual void Initialize(G4HCofThisEvent*);
      virtual void EndOfEvent(G4HCofThisEvent*);
      //  These two methods are invoked at the begining and at the end of each
      // event. The hits collection(s) created by this sensitive detector must
      // be set to the G4HCofThisEvent object at one of these two methods.
```

Sposób implementacji 3

Fragment pliku nagłówkowego klasy G4VSensitiveDetector, cd.:

```
  protected:
      virtual G4bool ProcessHits(G4Step*aStep,G4TouchableHistory*ROhist) = 0;
      //  The user MUST implement this method for generating hit(s) using the
      // information of G4Step object. Note that the volume and the position
      // information is kept in PreStepPoint of G4Step.

      virtual G4int GetCollectionID(G4int i);
      //  This is a utility method which returns the hits collection ID of the
      // "i"-th collection. "i" is the order (starting with zero) of
      //the collection whose name is stored to the collectionName
      // protected vector.

      G4CollectionNameVector collectionName;
      //  This protected name vector must be filled at the constructor of
      //the user's concrete class for registering the name(s) of hits
      // collection(s) being created by this particular sensitive detector.
...
#endif
```

Sposób implementacji 4

Klasa Sensitive Detector w praktyce (kroki)

- Wywołać konstruktor klasy matki czyli **G4VSensitiveDetector**
- W kostruktorze klasy stworzyć nazwę „naszej" HitsCollection i dodać ją do wektora **collectionName**
- W metodzie **Initialize()** utworzyć obiekt **HitCollection** i dodać ją do obiektu **G4HCOfThisEvent**
- Zaimplementować metodę **ProcessHits()** – w niej tworzymy „hity" i dodajemy je do kolekcji

Sposób implementacji 4

Klasa Sensitive Detector w praktyce (kroki)

```
class PMTSD : public G4VSensitiveDetector
{
  public:
    PMTSD(G4String name);
    virtual ~PMTSD();
    virtual void Initialize(G4HCofThisEvent* );
    virtual G4bool ProcessHits(G4Step* aStep, G4TouchableHistory* );
    void SetPMTDeph(G4int pmtDephVal) {pmtDeph = pmtDephVal;}
    void SetModuleDeph(G4int moduleDephVal) {moduleDeph = moduleDephVal;}

  private:
    //seria pomocniczych metod
    G4int GetIndex(const G4Step* aStep, int deph);
    G4double GetEnergyDeposit(const G4Step* aStep);
    G4double GetHitTime(const G4Step* aStep);
    PMTHitsCollection* pmtHitCollection;//opis troche dalej
    G4int moduleDeph;
};
```

Sposób implementacji 5

Tworzenie kolekcji i dodawanie jej do **G4HCofThisEvent**

```
PMTSD::PMTSD(G4String name)
  : G4VSensitiveDetector(name), moduleDeph(0), pmtDeph(0)
{
  collectionName.insert("pmtHitCollection");
}
PMTSD::~PMTSD() {}

void PMTSD::Initialize(G4HCofThisEvent* hitsCE)
{
  pmtHitCollection = new PMTHitsCollection
                         (SensitiveDetectorName,collectionName[0]);
  static G4int hitCID = -1;
  if (hitCID<0) {
    hitCID = G4SDManager::GetSDMpointer()->GetCollectionID(collectionName[0]);
  }
  hitsCE->AddHitsCollection( hitCID, pmtHitCollection);
}
```

Sposób implementacji 6

Uzyskiwanie danych z kroku i umieszczanie ich w obiekcie typu **G4VHit**

```
G4bool PMTSD::Process(G4Step* aStep, G4TouchableHistory* )
{
    G4ParticleTable* particleTable = G4ParticleTable::GetParticleTable();
    G4ParticleDefinition* neutron = particleTable->FindParticle("neutron");
    if(aStep->GetTrack()->GetDefinition() != neutron)
   return false;

    G4double energyDep = GetEnergyDeposit(aStep);
    if(energyDep <= 0)
       return false;

    G4int moduleIndex = GetIndex(aStep, moduleDeph);
    G4double hitTime = GetHitTime(aStep);
    PMTHit* aHit = new PMTHit(moduleIndex, energyDep, hitTime);
    pmtHitCollection->insert( aHit );
    return true;
}

G4int PMTSD::GetIndex(const G4Step* aStep, int deph)
{ ... }

G4double PMTSD::GetEnergyDeposit(const G4Step* aStep)
{ ... }

G4double PMTSD::GetHitTime(const G4Step* aStep)
{ ... }
```

G4VHit – ostatni element

Pozostaje tylko stworzyć obiekt, który będzie trzymał interesujące dane. Klasą bazową jest **G4VHit**.
Geant4 wymusza, aby do obiektu **G4VHit** napisać także operator przypisania i konstruktor kopiujący.

```
class PMTHit : public G4VHit
{
  public:
    //przykładowe dane, które możemy wkładać do konstruktora
    PMTHit(G4int moduleIndexVal,
           G4double energyDepVal, G4double hitTimeVal);
    virtual ~PMTHit();
    PMTHit(const PMTHit &right);
    const PMTHit& operator=(const PMTHit &right);
    G4int operator==(const PMTHit &right) const;

    inline void *operator new(size_t);
    inline void operator delete(void *aHit);

    //tu powinien być zbiór Getterów i Setterów
  private:
    G4int pmtIndex;
    G4double energyDepVal;
    G4double hitTimeVal;
};
```

G4VHit – ostatni element

Plik nagłówkowy cd. (po skończonej deklaracji klasy, ale w pliku *.hh)

```
//WAZNE i OBOWIĄZKOWE, zawsze wyglada tak samo
typedef G4THitsCollection<PMTHit> PMTHitsCollection;

extern G4ThreadLocal G4Allocator<PMTHit>* PMTHitAllocator;

inline void* PMTHit::operator new(size_t){
  if(!PMTHitAllocator)
      PMTHitAllocator = new G4Allocator<PMTHit>;
  return (void *) PMTHitAllocator->MallocSingle();
}

inline void PMTHit::operator delete(void *aHit){
  PMTHitAllocator->FreeSingle((PMTHit*) aHit);
}
```

G4VHit – ostatni element

Plik źródłowy

```
//Stały punkt
G4ThreadLocal G4Allocator<PMTHit>* PMTHitAllocator=0;
//konstruktor
PMTHit::PMTHit(G4int modIndex, G4double enDep, G4double hitTime)
{
//wpisanie danych z konstruktora do zmiennych klasy
}
PMTHit::~PMTHit() {}
//konstuktor kopiujący
PMTHit::PMTHit(const PMTHit &right) : G4VHit()
{
   moduleIndex = right.moduleIndex;
   time = right.time;
   energyDep = right.energyDep;
}
//operator przypisania
const PMTHit& PMTHit::operator=(const PMTHit &right)
{
   moduleIndex = right.moduleIndex;
   time = right.time;
   energyDep = right.energyDep;
   return *this;
}
//operator porownania
G4int PMTHit::operator==(const PMTHit &right) const
{
  return (this==&right) ? 1 : 0;
}
//gettery i settery
```

Wykorzystanie wyników

Mamy już Sensitive Detector, który wykonuje obsługę Step-ów, interesujące dane umieszcza w obiektach typu G4VHit, powstaje kolekcja tych Hitów (uderzeń?), do której chcielibyśmy się dostać pod koniec Event-u.
Dalsza procedura jest wiec bardzo zbliżona do wersji z Prostym Licznikiem.
Modyfikujemy klasę **DetectorConstruction** oraz **EventAction** następująco:
DetectorConstruction:
Przykład wzięty z „prawdziwego" kodu:

```
if (!pmtSD)
{
    pmtSD = new PMTSD("/VANDLEDet/pmtSD");//konstruktor
    pmtSD->SetModuleDeph(3); //numer kopii
}
G4SDManager* SDman = G4SDManager::GetSDMpointer();
SDman->AddNewDetector(pmtSD);//rejestracja w G4SDManager
photocathLogic->SetSensitiveDetector(pmtSD); //przypisanie do log vol
```

Wykorzystanie wyników

EventAction:

```
//1. uzyskanie informacji o ID kolekcji
G4SDManager* SDman = G4SDManager::GetSDMpointer();
int pmtCollID=SDman->GetCollectionID("nazwaSD/nazwaKolekcji");

//2. wyciągnij kolekcje oddziaływań dla zdarzenia (eventu)
//   (hits collections of an event)
G4HCofThisEvent* hitsCE = anEvent->GetHCofThisEvent();

//3. z hitsHC wyciągnij konkretną, interesującą kolekcję
//   (odpowiadajcą danemu ID)
PMTHitsCollection* pmtHC = (PMTHitsCollection*)( hitsCE->GetHC(pmtCollID) );

//4. przeiteruj się po kolekcji uzyskując interesujące dane
G4int pmts = pmtHC->entries();
for(G4int i=0; i!=pmts; i++)
{
    G4int moduleIndex = (*pmtHC)[i]->GetModuleIndex();
    G4double energyDep = (*pmtHC)[i]->GetEnergyDeposit();
    G4double time = (*pmtHC)[i]->GetTime();
}
```

Dane można już wypisywać na ekran lub do pliku tekstowego. Warto jednak korzystać z plików binarnych (np. ROOT).

Wykorzystanie klas ROOT'a

Aby skorzystać z możliwości, jakie daje ROOT należy przede wszystkim dodać go do listy wykorzystywanych bibliotek w CMakeList.txt

```
# root
find_package(ROOT)
include_directories(${ROOT_INCLUDE_DIRS})
```

oraz przy linkowaniu:

```
target_link_libraries(VANDLESim ${ROOT_LIBRARIES})
```

Następnie można stworzyć klasę (np **Output**), która stworzy plik z drzewem ROOT-a a następnie zapisze do niego wyniki naszej symulacji.

Wykorzystanie klas ROOT'a

Jak taka klasa mogłaby wyglądać?

```
#ifndef Output_H
#define Output_H
#include <string>
#include "TFile.h"
#include "TTree.h"
#include "PMTHit.h"

class Output {
public:
    Output(std::string fileName);
    ~Output();
    void AddHit(PMTHit &hit);
    //alternatywnie
    void AddHit(G4int modIndex, G4double enDep, G4double hitTime);

private:
    TFile* outputFile;
    TTree* outputTree;
    int modIndex;
    double enDep;
    double hitTime;
};
#endif
```

Klasa może być singletonem, wtedy trzeba konstruktor uczynić prywatnym i dodać publiczną statyczną metodę zwracającą instancję obiektu.

Wykorzystanie klas ROOT'a cd.

```
#include "Output.hh"
#include <iostream>
Output::Output(std::string filename)
{
    //stworzenie i otwarcie pliku
    outputFile = new TFile(filename.c_str(), "recreate");
    //stworzenie drzewa (może być wiecej niż jedno)
    outputTree = new TTree("outputTree", "outputTree");
    //stworzenie gałęzi i przepisanie im adresów
    outputTree->Branch("modIndex", &modIndex);
    outputTree->Branch("enDep", &enDep);
    outputTree->Branch("hitTime", &hitTime);
}
//w destruktorze zapisujemy i zamykamy plik
//jeśli obiekt jest singletonem warto stworzyć medotę do
//zamykania, aby destruktor pozostał prywatny
Output:: ~Output()
{
    outputFile->Write();
    outputFile->Close();
    delete outputFile;
}
//dodawanie hitów
void Output::AddHit(G4int modIndexV, G4double enDepV, G4double hitTimeV)
{
    modIndex = modIndexV;
    enDep = enDepV;
    hitTime = hitTimeV;
    //wpisz do drzewa!
    outputTree->Fill();
}
```

Wykorzystanie klas ROOT'a cd.

Do podglądania wyników można użyć **TBrowser()**, albo napisać prosty skrypt

```
void PlotTime()
{
    //otwarcie pliku
    TFile* f = new TFile("nazwaPliku");
    TTree* t=(TTree*)f->Get("nazwaDrzewa");
    //stworzenie histogramu (liczby przykładowe!)
    double xMin = 0;
    double xMax = 7000;
    int nBins = 1*(xMax-xMin);
    TH1F* time = new TH1F("time","time",nBins,xMin,xMax);
    //możemy stawiać bramki na innych gałęziach
    TCut cut = "modIndex == 39";

    t->Draw("hitTime>>time",cut);
    gPad->SetLogy();
    time->Draw();
}
```

Zadanie 1

- Znajdź całkowitą energię zdeponowaną w kręgosłupie fantomu oraz narysować rozkład czasu oddziaływań. Oszacować ilu średnio oddziaływaniom ulega kwant promieniowania γ w kręgosłupie.
- Narysuj rozkład liczby detektorów, które zarejestrowały promieniowanie jonizujące w jednym zdarzeniu
- Narysuj rozkład korelacji pomiędzy detektorami, które "wypaliły" (widmo 2D).