Zastosowanie pakietu Geant4 w fizyce jądrowej Wykład 7

Aleksandra Fijałkowska

14 kwietnia 2020

Wyniki symulacji, **G4Event**

Istnieją trzy sposoby aby uzyskać dostęp do wyników symulacji. Pierwszy z rozważanych jest łatwy koncepcyjnie i nie wymaga wielkiej wiedzy o bibliotece Geant. Jest jednak uciążliwy na etapie implementacji i nie jest to rozwiązanie zalecane przez twórców biblioteki Geant.
Wyniki symulacji to najczęściej energia zdeponowana w detektorze, ładunek, liczba fotonów zaabsorbowana w jakimś elemencie.
Chcielibyśmy uzyskać dostęp do tych statystyk pod koniec każdego zdarzenia (Eventu).
Czym jest zdarzenie? Przypominam slajd z pierwszych zajęć.

Wyniki symulacji, **G4Event**

O evencie (zdarzeniu) na pierwszych zajęciach powiedzieliśmy sobie, że :

- Każdy Run składa się z określonej przez użytkownika liczby zdarzeń
- Even rozpoczyna się wysłaniem zdefiniowanych przez użytkownika cząstek pierwotnych
- Na początku zdarzenia wszystkie cząstki pierwotne umieszczane są na stosie a następnie transportowane przez geometrię
- Niektóre procesy, którym ulegają cząstki pierwotne, mogą powodować powstanie cząstek wtórnych (np. kreacja pary elektron-pozyton)
- Powstałe cząstki wtórne są odkładane na stos, a następnie jedna po drugiej transportowane przez detektor
- Po przetransportowaniu wszystkich cząstek przez geometrię program wykonuje polecenia określone w klasie `G4UserEventAction` (zapisanie danych do pliku) i kończy zdarzenie Zdarzeniem kieruje klasa G4EventManager.

Wyniki symulacji, **G4Event**

Zdarzenie reprezentuje klasa `G4Event`, do której dostęp mamy w klasie `EventAction`, która dziedziczy po klasie abstrakcyjnej `G4UserEventAction`. W kasie mamy dwie metody

```
virtual void BeginOfEventAction(const G4Event*);
virtual void EndOfEventAction(const G4Event*);
```

pierwsza z nich zostaje wywołana NA początku eventu, druga zaś na końcu. Jest to świetne miejsce aby uzyskać informacje o tym, co się w symulacji wydarzyło. Jednakże sama klasa `G4Event` nie daje bezpośredniego dostępu do żadnych statystyk.

Wyniki symulacji, **G4Step**

Klasa, która daje dostęp do właściwie wszystkiego, co się w symulacji wydarzyło to `G4Step`.
Zanim popatrzymy na metody, którymi dysponuje klasa `G4Step` warto omówić sposób dostania się do jej instancji.
Podobnie jak w przypadku klasy `EventAction` nasz kod implementuje klasę `SteppingAction`.
Klasa `SteppingAction` dziedziczy po abstrakcyjnej klasie bazowej `G4UserSteppingAction`.
Klasa ta posiada wirtualną metodę `UserSteppingAction(const G4Step* step)`, która jest wykonywana po każdym kroku (`Step`).

Implementując tą metodę możemy skorzystać ze wskaźnika do obiektu `G4Step* step` i dowiedzieć się, co w tym kroku się wydarzyło.

Wyniki symulacji, **G4Step**

Funkcje dostępne w klasie G4Step:

```
G4double  GetStepLength () const
//zwraca długośc kroku
G4double  GetTotalEnergyDeposit () const
//depozyt energii
G4double  GetNonIonizingEnergyDeposit () const
//depozyt energii w formie innej niż jonizacja
G4ThreeVector  GetDeltaPosition () const
//zmiana pozycji cząstki
G4ThreeVector  GetDeltaMomentum () const
//zmiana pędu cząstki
//.. i wiele innych (warto sprawdzić)
```

Są też cztery supermetody. Nie zwracają one bezpośrednich statystyk o kroku, ale obiekty, z których też można wyciągnąć ciekawe informacje.

```
const G4TrackVector *  GetSecondary () const
//zwraca wskaźnik do wektora trzymającego tory cząstek wtórnych,
//utworzonych w danym kroku
G4StepPoint* GetPreStepPoint ()
//zwraca PUNKT przed krokiem
G4StepPoint* GetPostStepPoint () const
//zwraca punkt po kroku
G4Track* GetTrack () const
//zwraca obiekt typu G4Track (co można by było tłumaczyć jako tor,
//ale nie jest to najlepsze tłumaczenie)
```

Wyniki symulacji, **G4Step**

Przykładowe metody klasy `G4Track`:

```
const G4ParticleDefinition* GetParticleDefinition() const
//znamy już typ G4ParticleDefinition, możemy się dowiedzieć jaka cząstka oddziałuje
const G4ThreeVector& GetPosition() const
//dokładna pozycja
G4VPhysicalVolume* GetVolume() const
//zwraca objętość fizyczną
//przypominam, że tworzymy ją w ostatnim kroku budowania geometrii
G4Material* GetMaterial() const
//zwraca materiał, w którym wystąpił krok,
//możemy więc wybrać depozyty energii w konkretnym materiale
G4double GetKineticEnergy() const
//zwraca energię kinetyczną cząstki PRZED krokiem
G4double GetTotalEnergy() const
//zwraca całkowitą energię cząstki PRZED krokiem
const G4ThreeVector& GetMomentumDirection() const
//zwraca kierunek cząstki przed krokiem
G4double GetLocalTime() const
//zwraca czas kroku
```

Metod jest więcej, zachęcam do zazponania się z dokumentacją. Wystarczy wpisać w wyszukiwarkę nazwę klasy, lub odszukać odpowiedni plik z kodzie Geant'a (co jest trudniejsze).

Podobne informacje można wyciągnąć z klasy `G4StepPoint`.

Zadanie

Znajdź całkowitą energię zdeponowaną w kręgosłupie fantomu.
Założenia:

- Ze środka fantomu (całej geometrii) emitowany jest pozyton o energii 587 keV (średnia energia pozytonu emitowanego w przemianie β ^{89}Sr) z izotropowym rozkładem kątowym.
- Jako wynik chcemy uzyskać listę depozytów energii w danym zdarzeniu w postaci pliku tekstowego zawierającego kolumnę z numerem zdarzenia oraz odpowiadającym mu całkowitym depozytem energii.
- W klasie **SteppingAction** stwórz zmienną, która będzie trzymała depozyt energii w kręgosłupie (np. **spineEnergyDep**).
- W każdym kroku zmienna **spineEnergyDep** musi być zwiększana o depozyt energii w tym kroku pod warunkiem, że krok miał miejsce w kręgosłupie. Algorytm ten należy napisać wewnątrz metody **void SteppingAction::UserSteppingAction(const G4Step* theStep)**.
- Depozyt energii można uzyskać dzięki metodzie klasy **G4Step**, **GetTotalEnergyDeposit()** (proszę nie zapominać o jednostkach!)