Geant4 @ NCBJ, 2022
5. Wyniki symulacji
-
detektor użytkownika

Przemysław Adrich

NARODOWE CENTRUM BADAŃ JĄDROWYCH
ŚWIERK

Plan kursu

Wykłady

1. Wstęp do Geant4. Rys historyczny. Zastosowania. Przegląd możliwości. Instalacja. Dokumentacja.
2. Podstawowa struktura kodu. Hierarchie klas. Klasy użytkownika (obowiązkowe, opcjonalne).
   ~~Interfejsy. System jednostek. Liczby losowe. Śledzenie przebiegu symulacji („verbosity”).~~
3. Geometria i materiały.
4. Detektory typu „primitive scorer”, „probe”.
5. **Detektory użytkownika.**
6. Obiekty typu „UserAction” jako detektory. (Phasespace). Histogramy i n-tuple.
7. Źródło. Fizyka. Wizualizacja.
8. Niepewność statystyczna w obliczeniach Monte Carlo. Geant4 na klastrze CiŚ.

* „Monte Carlo First Run” (wykład bonusowy)

Przegląd zagadnień pozostawionych na przyszłość:

- wielowątkowość („multithreading”),
- własne interfejsy („messengers”),
- interfejs Roota (histogramy, n-tuple), interfejs python
- redukcja wariancji, „physics biasing”, „event biasing”, „geometrical biasing”,
- fotony optyczne, fizyka hadronowa, procesy i cząstki użytkownika,
- obcięcia energetyczne zależne od cząstki, regionu geometrii,
- zmiany geometrii i detektorów w trakcie wykonania programu,
- pole EM,
- światy równoległe,
- trackInformation, eventInformation, runInformation
- „stacking”,
- fast simulation,
- import geometrii z CAD,
- periodic boundary conditions,
- specjalistyczne kody bazujące na Geant4 (G4Beamline, GAMOS, GATE …),
- …

Geant4 – wprowadzenie. Co jest potrzebne by zbudować aplikację?

- Geant4 jest zestawem bibliotek i interfejsów. Użytkownik musi zbudować własną aplikację.
- W tym celu trzeba:
  - Zdefiniować układ:
    - geometria, materiały
  - Zdefiniować fizykę, która ma być stosowana w symulacji:
    - Cząstki, procesy/modele procesów fizycznych
    - Progi produkcji
  - Określić od czego mają zaczynać się zdarzenia:
    - Generator cząstek pierwotnych (źródło)
  - **Zadbać o wydobycie użytecznych informacji**
- Opcjonalnie można również:
  - Wizualizować geometrię, trajektorie, wyniki fizyczne
  - Dodać interfejs użytkownika (np. graficzny)
  - Zdefiniować własne komendy, itd., itd.

Geant4. Dostęp i gromadzenie wyników symulacji

- Standardowo symulacja przebiega bez gromadzenia jakichkolwiek wyników.
- Użytkownik musi sam określić jakie dane i w jaki sposób chce gromadzić.
- Znam 3½ zalecanych sposobów zbierania wyników symulacji:

1. Proste liczniki (primitive scorer) obsługiwane z poziomu poleceń interfejsu użytkownika. Działają w geometrii równoległej, niezależnej od rzeczywistej geometrii modelu zdefiniowanego w `G4VUserDetectorConstruction`.

2. **Zdefiniowanie objętości logicznej (bryły poziomu `G4LogicalVolume`) jako elementu aktywnego i przypisanie jej detektora:**

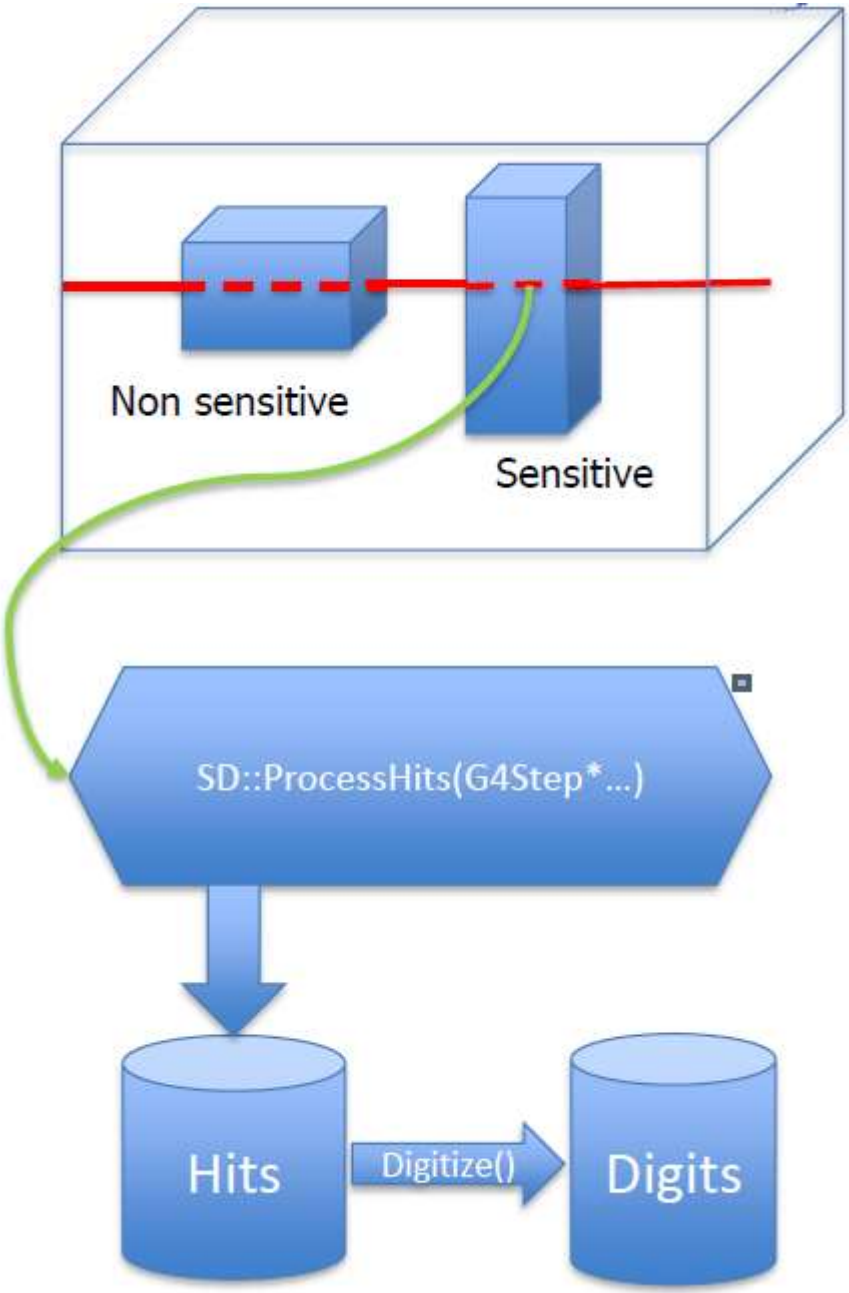
   2a. **Implementując własne klasy pochodne od `G4VSensitiveDetector` i `G4VHit`.**

   2b. Stosując gotową klasę `G4MultiFunctionalDetector`, która jest konkretną implementacją klasy `G4VSensitiveDetector` opartą o proste liczniki (obiekty typu `G4VPrimitiveScore`).

   W obu wypadkach, po zakończeniu eventu mamy dostęp, do pełnego zbioru hitów (`G4THitsCollection`) lub mapy wielkości akumulowanych w trakcie eventu (`G4THitsMap`) najczęściej indeksowanej numerem kopii obszaru logicznego. Użytkownik ma dostęp do tych danych korzystając z klasy `G4UserEventAction` lub własnej klasy pochodnej od `G4Run`.

3. Korzystając z własnych implementacji klas `G4UserTrackingAction`, **`G4UserSteppingAction`** i `G4UserStackingAction`. W ten sposób również mamy dostęp do pełnej informacji ale sami musimy zadbać o jej przetwarzanie w trakcie procesowania przypadku.

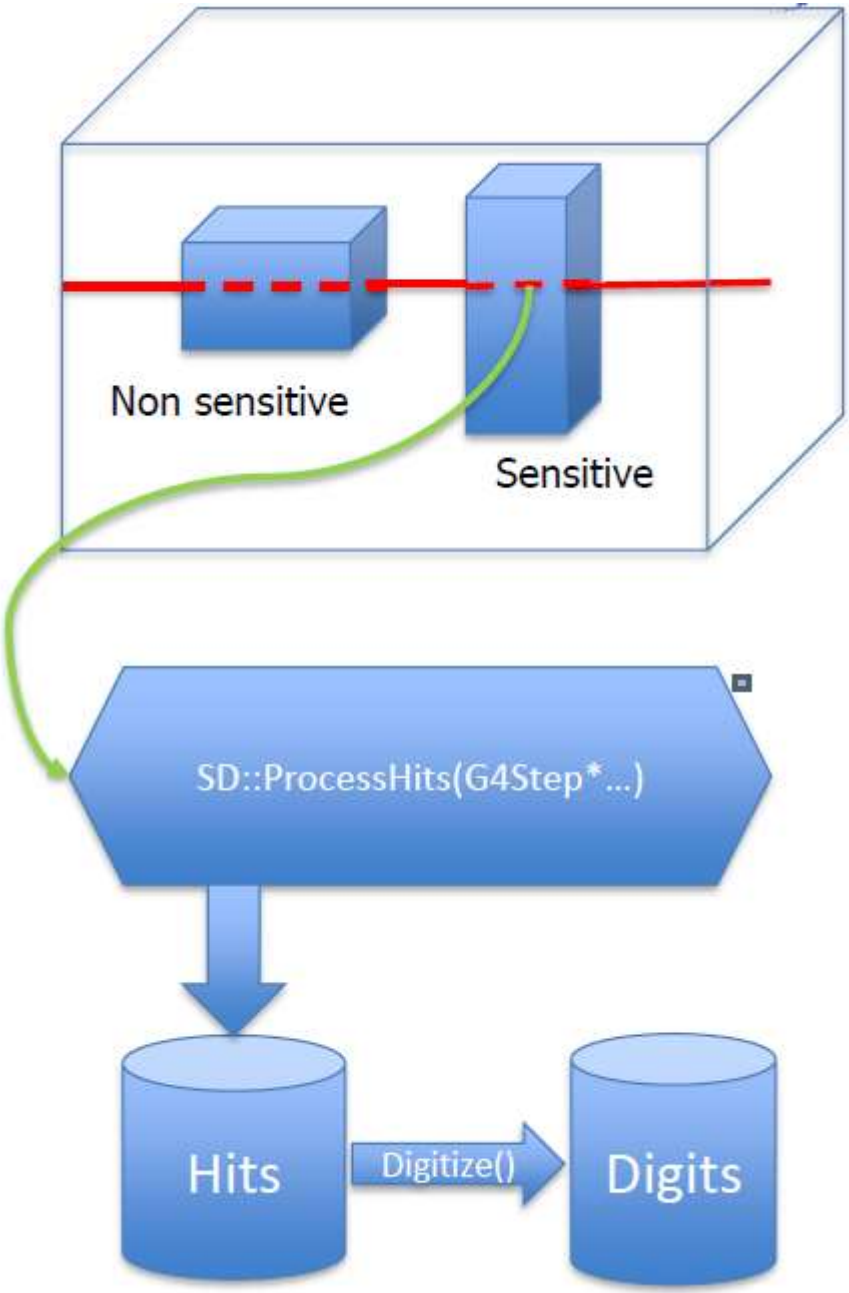
Element aktywny (detektor)



Koncepcja jest bardzo prosta

- Jądro geanta propaguje (transportuje) cząstkę przez geometrię
- Kiedy trafi na bryłę zdefiniowaną jako aktywna (sensitive – czuła), wywołuje specjalną funkcję użytkownika (detektor) powiązaną z tą bryłą:
  - po każdym kroku (w tej bryle),
  - na wejściu dostaje komplet informacji o tym co się zdarzyło z cząstką w trakcie kroku.

Element aktywny (detektor)



- Element aktywny (detektor; „sensitive detector”) – obiekt klasy pochodnej od `G4VSensitiveDetector`, przypisany do bryły na poziomie LogicalVolume
- Metoda (funkcja składowa) `ProcessHits` jest wywoływana po każdym kroku cząstki wykonanym w bryle, której LogicalVolume ma przypisany detektor
- `ProcessHits` dostaje na wejściu obiekt typu `G4Step` zawierający pełną informację o tym co wydarzyło się z cząstką podczas właśnie wykonanego kroku
- `ProcessHits` generuje „hity” - obiekty klasy pochodnej od `G4VHit` zawierające wybrane informacje np.: zdeponowaną energię, pozycję, czas, rodzaj cząstki, …
- Wszystkie hity wytworzone w evencie, składowane są w specjalnych kontenerach („pojemnikach”), do których użytkownik ma dostęp.
- Przetwarzanie informacji z hitów zebranych w evencie należy zrobić w `userEventAction->EndOfEventAction(currentEvent)`
- Gromadzenie/analizę wyników zbiorczych z całego runu należy zrobić w `userRunAction`
- Opcjonalnie, hity mogą być poddane „digitalizacji” (wyrażeniu informacji np. jako wartość prądu czy napięcia, tak jak miałoby to miejsce w rzeczywistym detektorze) i zapisane w formacie identycznym do używanego przez rzeczywisty system akwizycji danych eksperymentu.

Mocno zalecane (choć można po swojemu)

Tworzenie detektora

- *Detektor(y) powinien być stworzony i powiązany z objętością logiczną w funkcji składowej* `ConstructSDandField()` *klasy DetectorConstruction*

Geometria – jak zrobić własne DetctorConstruction?

```
// DetectorConstruction.hh

#ifndef DetectorConstruction_h
#define DetectorConstruction_h 1

#include "G4VUserDetectorConstruction.hh"
#include "G4NistManager.hh"
class G4LogicalVolume;
class G4VPhysicalVolume;
class G4Box;
class G4Tubs;

class DetectorConstruction : public G4VUserDetectorConstruction
{
  public:
    DetectorConstruction();
    virtual ~DetectorConstruction();
    virtual G4VPhysicalVolume* Construct();
    void ConstructSDandField();

    void SetWorldMaterial(G4String newMaterial);
    void SetupPhantom();
    void SetPDDPhantomNLayers(G4int newNLayers);
    G4int GetPDDPhantomNLayers();
    void SetPDDPhantomLayerThickness(G4double newPDDPhantomLayerThickness);

  private:
  ...
};
#endif
```

Interfejs publiczny

Tu definiujemy geometrię

Tu definiujemy detektor(y)

Implementacja. Prywatna.

Tworzenie detektora

- *Detektor(y) powinien być stworzony i powiązany z objętością logiczną w funkcji składowej* `ConstructSDandField()` *klasy DetectorConstruction*

```
void DetectorConstruction::ConstructSDandField()
{
  G4VSensitiveDetector* SD = new MySD("/MySDname", "MyHitsCollection");
  G4SDManager::GetSDMpointer()->AddNewDetector(SD);
  SetSensitiveDetector("MyLVName", SD);
  // lub
  // SetSensitiveDetector(MyLVptr, SD);
  // lub
  // MyLVptr->SetSensitiveDetector(SD);
}
```

- *Każdy detektor musi mieć unikalną nazwę.*
- *Można utworzyć kilka obiektów reprezentujących detektory tego samego typu ale każdy musi mieć inną nazwę,*
- *Detektor należy zarejestrować do menadżera detektorów* (`G4SDManager`)*,*
- *Przypisanie detektora do objętości logicznej można zrobić na różne sposoby: albo za pomocą nazwy albo za pomocą wskaźników.*
- *Ten sam detektor może być przypisany do kilku różnych objętości logicznych (informacja o objętości logicznej jest zawarta w G4Step)*
- *Ale jedna objętość logiczna może mieć przypisany tylko jeden detektor. Za to może on produkować hity kilku różnych rodzajów.*

Klasa bazowa G4VSensitiveDetector

```
#include "G4VSensitiveDetector.hh"
...
class MySD : public G4VSensitiveDetector {
public:
        MySD(const G4String& name, const G4String& hitsCollectionName);
        virtual ~MySD();

        virtual void Initialize(G4HCofThisEvent* hce);
        virtual G4bool ProcessHits(G4Step* step, G4TouchableHistory* history);
        virtual void EndOfEvent(G4HCofThisEvent* hce);

        void SetFilter (G4VSDFilter *value)
};
```

Detektor **musi** być zdefiniowany w klasie pochodnej od **G4VSensitiveDetector**

Klasa bazowa narzuca interfejs składający się, m.in., z trzech funkcji wywoływanych przez jądro Geanta w trakcie przetwarzania eventu:

- Na początku eventu: **Initialize(...)**
- Po wykonaniu kroku (o ile w powiązanej objętości): **ProcessHits(...)**
- Na końcu eventu: **EndOfEvent(...)**

Funkcja składowa **SetFilter** umożliwia przypisanie detektorowi filtra cząstek: jednego z wbudowanych lub własnego.

- Wbudowane filtry: `G4SDChargedFilter`, `G4SDNeutralFilter`, `G4SDParticleFilter`, `G4SDKineticEnergyFilter`, `G4SDParticleWithEnergyFilter`.
- Własny filtr może być dowolnie złożony (do podjęcia decyzji dostanie na wejściu obiekt typu `G4Step`)

Hit

Zasadniczo rozróżnia się dwa rodzaje hitów

1. Z detektora śledzącego (trajektorię cząstki)
    - Nowy hit tworzony jest po każdym kroku każdej cząstki w detektorze,
    - Hity tworzone są w `MySD::ProcessHits()`,
    - Hit typowo zawiera informację o: pozycji i czasie, stracie energii, ID cząstki, ID detektora

2. Z detektora kalorymetrycznego (tj. mierzącego całkowitą energię cząstki)
    - Hity tworzone są jednokrotnie w `MySD::Initialize()`. Na początku są puste (wyzerowane),
    - Hity są aktualizowane w `MySD::ProcessHits()` (do bieżącej wartości dodawana jest energia zdeponowana w aktualnym kroku),
    - Na końcu eventu hit typowo zawiera: całkowitą energię zdeponowaną w komórce detektora podczas eventu, ID komórki
    - Zamiast energii można oczywiście zbierać inną wielkość ale istotne jest, że na końcu eventu w hicie mamy sumę wkładów do tej wielkości z wszystkich kroków wykonanych przez cząstki w danym detektorze.

Hity składowane są w odpowiednich kontenerach…

Kontenery w C++ (biblioteka standardowa)

Kontener – klasa przechowująca inne obiekty. W C++ istnieją różne standardowe kontenery. Dynamiczna alokacja.

Vector – kontener danych typu uporządkowanego (trochę jakby dynamiczna tablica)

- dostęp do elementów poprzez ich indeks,
- iteracja po elementach w dowolnym kierunku,
- dodawanie i usuwanie elementów z końca.

```
#include <vector>
// vector<TypDanych> nazwa(początkowa długość);
vector<int> mojWektor(100);
mojWektor[5] = 11;
z = mojWektor.at(5);
```

Map – kontener danych składających się z par (klucz, wartość).

- dostęp do wartości pary poprzez wartość klucza,
- iteracja po parach.

```
#include <map>
// map<TypKlucza, TypWartości> nazwa;
map<String, int> mojaMapa;
mojaMapa["X"] = 12345;    // Jeśli klucz „X” wcześniej nie istniał to zostanie stworzony
mojaMapa["Y"] = 54321;
cout << mojaMapa["X"] << endl;
```

Wybrane metody ułatwiające pracę z kontenerami

```
kontener.size();              // ilość elementów w kontenerze
kontener.empty()              // true jeśli kontener pusty
kontener.clear();             // usuwa wszystkie elementy
wektor.push_back(wartość);    // dopisuje wartość na końcu wektora
wektor.pop_back();            // usuwa ostatni element z wektora
wektor.insert(pozycja,wartość); // wstawia element na konkretną pozycję
wektor.front();               // pierwszy element wektora
wektor.back();                // ostatni element wektora
mapa.find(klucz);             // sprawdza czy element istnieje
```

Iterator kontenera

Iterator – jakby wskaźnik ułatwiający pracę z kontenerami.

Iterator wektora – wskazuje na element wektora

```
vector<TypDanych>::iterator it = mojWektor.begin();
while ( it != mojWektor.end() ) {
  *it = .... // zmiana wskazywanej wartości
  it++;      // przejście do następnego elementu wektora
}
```

Iterator mapy – wskazuje na parę. Para jest obiektem posiadającym pola first (klucz) oraz second (wartość)

```
map<String, int> mojaMapa;
mojaMapa["X"] = 12345;
mojaMapa["Y"] = 54321;

map<string, int >::iterator it = mojaMapa.begin();
while ( it != mojaMapa.end() ) {
  cout << (*it).first << " - " << (*it).second << endl;
  it++; // przejście do następnego elementu mapy
}
it = mojaMapa.find("Y");  // ustawienie iteratora na konkretną parę
mojaMapa.erase("X");      // usunięcie z mapy konkretnej pary
```

Geant4 – kontenery hitów

`G4VHitsCollection` – wspólna abstrakcyjna klasa bazowa dwóch rodzajów kontenerów: **`G4THitsCollection`** i **`G4THitsMap`**

- *Nazewnictwo wprowadza lekkie zamieszanie.*
- *Uwaga na marginesie. Wszystko co zaczyna się od* **`G4T`** *oznacza klasę szablonową (od „template”).*

`G4THitsCollection` – szablonowa klasa wektorowa na elementy, które są **wskaźnikami do obiektów typu pochodnego od `G4VHit`**

- Innymi słowy, `G4THitsCollection` wymusza implementację własnej klasy Hit.
- W przypadku segmentacji detektora:
  - hity z detektora śledzącego powinny posiadać własne identyfikatory (np. ID kopii objętości logicznej);
  - hity z detektora kalorymetrycznego mogą być identyfikowane indeksem wektora (pozycja w wektorze = numer kopii objętości logicznej)

`G4THitsMap` – szablonowa klasa mapy par (klucz, wskaźnik do obiektu jakiegoś konkretnego typu).

- Klucz najczęściej jest numerem kopii objętości logicznej.
- Wskazywany obiekt może być dowolnego typu. Nie musi być pochodny od `G4VHit`. Może być np. zwykłą liczbą typu podstawowego (np. double).
- Proste liczniki wbudowane (primitive scorer) używają `G4THitsMap` (do przechowywania wartości typu `G4double` z kluczem typu `G4int`)

Implementacja klasy Hit – schemat

Klasa Hit zasadniczo służy do budowy prostych obiektów. Najczęściej składających się z:

- prywatnych pól danych, które chcemy gromadzić
- prostych, publicznych funkcji do zapisywania/odczytywania danych składowych

To nie jest sztywna reguła.
Nie musi koniecznie tak być

Data member	G4type fData;	G4double fEdep;
Set function	void SetData(G4type data);	void SetEdep(G4double edep):
Get function	G4type GetData() const;	G4double GetEdep() const;

```
class MyHit
{
  public:
    MyHit();
    void SetEdep(G4double edep);
    G4double GetEdep() const;
  private:
    // some data members; eg.
    G4double fEdep; // energy deposit
};
```

Uwaga na marginesie.
*W Geant4 nazwa rozpoczynająca się od „**f**” oznacza pole danych klasy (od ang. field), inaczej składnik danych klasy.*
Zaleca się przestrzegać tej konwencji.

Implementacja klasy Hit (dla detektora śledzącego)

https://geant4-userdoc.web.cern.ch/Doxygen/examples_doc/html/ExampleB2.html
https://geant4-userdoc.web.cern.ch/Doxygen/examples_doc/html/classB2_1_1TrackerHit.html
https://geant4-userdoc.web.cern.ch/Doxygen/examples_doc/html/TrackerHit_8hh_source.html
https://geant4-userdoc.web.cern.ch/Doxygen/examples_doc/html/TrackerHit_8cc_source.html

MyHit.hh

```
#include "G4VHit.hh"
class MyHit : public G4VHit
{
  public:
    MyHit(some_arguments);
    virtual ~MyHit();
    virtual void Draw();
    virtual void Print();
  private:
    // some data members
    G4ThreeVector fPosition;
    G4double fdE;
    G4double fTime;
    ...
  public:
   // some set/get methods
   void setPosition(G4ThreeVector);
   G4ThreeVector getPosition();
   ...
};

#include “G4THitsCollection.hh”
typedef G4THitsCollection<MyHit> MyHitsCollection;

inline void MyHit::setPosition(G4ThreeVector pos)
{
  fPosition = pos;
}
```

Deklaracje danych, które chcemy gromadzić

Deklaracje funkcji dostępu do danych (interfejs)

Dla wygody definiowana jest nowa, krótka nazwa dla typu kontenera hitów

Implementacja klasy Hit (dla kalorymetru)

CalorHit.hh

https://geant4-userdoc.web.cern.ch/Doxygen/examples_doc/html/ExampleB4.html
https://geant4-userdoc.web.cern.ch/Doxygen/examples_doc/html/classB4c_1_1CalorHit.html
https://geant4-userdoc.web.cern.ch/Doxygen/examples_doc/html/CalorHit_8hh_source.html
https://geant4-userdoc.web.cern.ch/Doxygen/examples_doc/html/CalorHit_8cc_source.html

```
#include "G4VHit.hh"
#include "G4THitsCollection.hh"

class CalorHit : public G4VHit
{
  public:
    CalorHit();
    ~CalorHit() override;

  private:
    G4double fEdep = 0.;

  public:
   void Add(G4double de);
   G4double GetEdep() const;
};

inline void CalorHit::Add(G4double de) {
   fEdep += de;
}

inline G4double CalorHit::GetEdep() const {
  return fEdep;
}
```

Uwaga na marginesie. Słowo „override” oznacza, że deklarujemy funkcję wirtualną, która przysłoni analogiczną funkcję z klasy bazowej. Kompilator zgłosi błąd jeśli w klasie bazowej nie ma takiej funkcji. Mechanizm używany do wykrywania zmian interfejsu w klasach bazowych.

Deklaracje danych, które chcemy gromadzić

Deklaracje funkcji dostępu do danych (interfejs)

Sumujemy depozycję energii w detektorze w trakcie eventu

G4Allocator – wydajna kreacja/destrukcja obiektów

- Tworzenie i usuwanie obiektów to złożone i obciążające operacje
  - za każdym razem program/system musi przydzielić/zwolnić obszar pamięci (aktualizacja tablic alokacji, itd.)
  - może to powodować problemy z wydajnością, w szczególności w przypadku obiektów, które są tworzone/usuwane bardzo często:
    - np. hity (również trajektorie i punkty trajektorii)
- W celu złagodzenia tego problemu, Geant4 dostarcza specjalną klasę `G4Allocator`
  - rezerwuje ciągły obszar pamięci,
  - w ramach wcześniej zarezerwowanego obszaru przydziela/zwalnia miejsce dla obiektu **danej konkretnej klasy** – ta operacja jest szybka bo wielkość przydzielanego bloku jest stała i z góry znana (analogia: tablica jednowymiarowa - komórki pamięci przydzielone jednokrotnie, używane wielokrotnie bez potrzeby każdorazowego powtarzania operacji rezerwowania/zwalniania pamięci)
  - Operatory `new` i `delete` klasy Hit powinny używać `G4Allocator`

Dla zaawansowanych:

- *G4Allocator musi być lokalny dla wątku (musi być typu thread-local)*
- *Obiekty tworzone przez G4Allocator muszą zostać usunięte w ramach tego samego wątku. Ale inne wątki mogą się odnosić do takich obiektów.*

G4Allocator – jak używać? (schemat)

MyHit.hh

```
#include "G4VHit.hh"
#include "G4Allocator.hh"
class MyHit : public G4VHit
{
  public:
  MyHit(some_arguments);
  inline void* operator new(size_t);
  inline void  operator delete(void *aHit);
  . . .
};
extern G4ThreadLocal G4Allocator<MyHit>* MyHitAllocator;
inline void* MyHit::operator new(size_t)
{
  if (!MyHitAllocator) MyHitAllocator = new G4Allocator<MyHit>;
  return (void*)MyHitAllocator->MallocSingle();
}
inline void MyHit::operator delete(void* aHit)
{ MyHitAllocator->FreeSingle((MyHit*)aHit); }

#include “G4THitsCollection.hh”
typedef G4THitsCollection<MyHit> MyHitsCollection;
```

Zalecana deklaracja własnych, szybkich, operatorów new i delete.

Dla wygody definiowana jest nowa nazwa dla typu kontenera hitów

MyHit.cc

```
#include "MyHit.hh"
G4ThreadLocal G4Allocator<MyHit>* MyHitAllocator = 0;
```

Wskaźnik do alokatora wspólny (globalny) dla wszystkich obiektów klasy MyHit

Przykład implementacji detektora śledzącego

Przykład implementacji detektora śledzącego – 1 (deklaracja klasy)

MyDetector.hh

```
#include "G4VSensitiveDetector.hh"
#include "MyHit.hh"
class G4Step;
class G4HCofThisEvent;
class MyDetector : public G4VSensitiveDetector
{
  public:
    MyDetector(G4String name);
    virtual ~MyDetector();
    virtual void Initialize(G4HCofThisEvent* HCE);
    virtual G4bool ProcessHits(G4Step*aStep, G4TouchableHistory*ROhist);
    virtual void EndOfEvent(G4HCofThisEvent* HCE);
  private:
    MyHitsCollection* hitsCollection;
    G4int collectionID;
};
```

Wskaźnik do kontenera hitów (naszego własnego typu)

Identyfikator kontenera

Przykład implementacji detektora śledzącego – 2 (definicja konstruktora)

```
MyDetector::MyDetector(G4String detector_name, G4String collection_name)
                  :G4VSensitiveDetector(detector_name),
                  collectionID(-1)
{
    collectionName.insert(collection_name);
}
```

Zmienną do przechowywania numeru identyfikacyjnego kontenera hitów inicjalizujemy na -1. Bo jeszcze go nie znamy.
Identyfikator kontenera przydzieli G4SDManager w chwili rejestrowania nowego detektora (w DetectorConstruction::ConstructSDandField).

`collectionName` *to składnik danych klasy bazowej* `G4VSensitiveDetector`*.*
Jest to kontener typu pochodnego od vector.

- W konstruktorze definiujemy nazwę detektora i nazwę kontenera hitów (kolekcji) obsługiwanego przez ten detektor.
- Jeśli detektor produkuje więcej niż jeden rodzaj hitów to dla każdego trzeba zdefiniować oddzielną nazwę kontenera.

Przykład implementacji detektora śledzącego – 3 (definicja funkcji `Initialize`)

```
void MyDetector::Initialize(G4HCofThisEvent* HCE)
{
  if(collectionID<0) collectionID = GetCollectionID(0);
  hitsCollection = new MyHitsCollection(SensitiveDetectorName, collectionName[0]);
  HCE->AddHitsCollection(collectionID,hitsCollection);
}
```

Pobieramy identyfikator kontenera hitów przydzielony przez G4SDManager. (`GetCollectionID` *jest funkcją składową klasy bazowej* `G4VSensitiveDetector`).

Tworzymy kontener na nasze własne hity, które wyprodukujemy podczas właśnie się rozpoczynającego eventu.
Nazwę kontenera już stworzyliśmy w konstruktorze.

Rejestrujemy nasz kontener na liście wszystkich innych kontenerów bieżącego eventu (`G4HCofThisEvent`).

- Metoda `Initialize()` jest wywoływana na początku każdego eventu.
  - Pobieramy w niej numer identyfikacyjny kontenera hitów (stały podczas runu):
    - `GetCollectionID()` *jest obciążającą operacją i nie powinna być wywoływana dla każdego eventu*
    - `GetCollectionID()` *jest dostępna dopiero kiedy obiekt typu SensitiveDetector jest stworzony (*`new`*) i zarejestrowany do* `G4SDManager`*. Dlatego metoda* `GetCollectionID()` *nie może być wywołana w konstruktorze naszego detektora.*

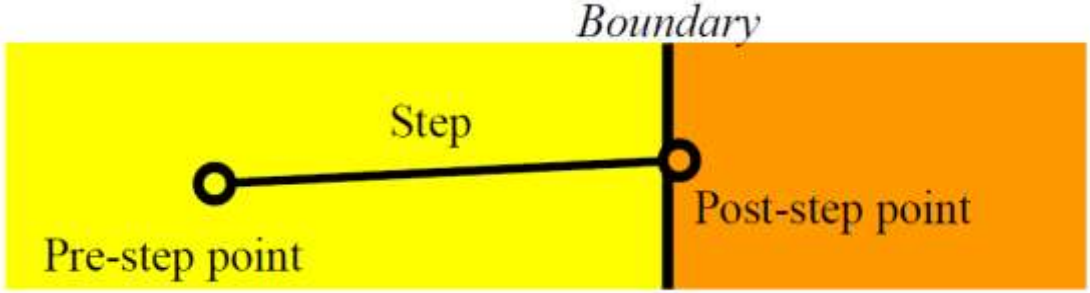
  - Tworzymy obiekt typu kontener naszych hitów (`MyHitsCollection`) i dołączamy go do listy wszystkich kontenerów bieżącego eventu (`G4HCofThisEvent`), którą dostaliśmy jako argument (od G4SDManager, który, na polecenie Event Managera, uruchamia inicjalizację detektorów na początku eventu).

Przykład implementacji detektora śledzącego – 4 (definicja funkcji `ProcessHits`)

```
G4bool MyDetector::ProcessHits(G4Step* aStep, G4TouchableHistory* ROhist)
{
  MyHit* aHit = new MyHit();
  ...
  // some set methods
  ...
  hitsCollection->insert(aHit);
  return true;
}
```

- Metoda `ProcessHits()` jest wywoływana dla **każdego kroku** wykonanego w objętości logicznej, do której „podpięty” jest nasz detektor (chyba, że zadziałał filtr cząstek).
- W tej metodzie produkujemy nowy hit na podstawie danych zawartych w obiekcie `G4Step` otrzymywanym jako parametr.
- Wytworzony i wypełniony danymi hit dołączamy do naszej kolekcji (kontenera).
- Zwracana wartość logiczna nie jest obecnie używana (być może będzie w przyszłych wersjach Geanta).

Przykład implementacji detektora śledzącego – 5 (wydobycie informacji z G4Step)



- *Step* (krok) ma dwa punkty: początkowy („Pre-step point”) i końcowy („Post-step point”)
- Krok reprezentuje zmianę stanu cząstki przy przejściu od punktu początkowego do końcowego (strata energii w trakcie kroku, czas przelotu (time-of-flight), itd.).
- Jeśli długość kroku została ograniczona granicą obiektów geometrycznych, to punkt końcowy fizycznie jest położony na granicy ale logicznie należy już do następnej bryły.
- Każdy obiekt `G4StepPoint` zawiera:
  - pozycję w globalnym układzie współrzędnych („świata”),
  - czas globalny i lokalny,
  - materiał,
  - obiekt `G4TouchableHistory`.
- Obiekt `G4TouchableHistory` jest wektorem zawierającym informację o hierarchii geometrycznej, do której należy punkt:
  - numer kopii objętości logicznej,
  - transformacja geometryczna pomiędzy objętościami matką a córką.

- Dane geometryczne nt. objętości logicznej, w której był wykonany krok, należy pobierać z `PreStepPoint`. Bo `PostStepPoint` może już logicznie należeć do innej objętości, zaś informacja geometryczna w `G4Track` jest identyczna do tej zawartej w `PostStepPoint`.

Przykład implementacji detektora śledzącego – 6 (wydobycie informacji z G4Step)

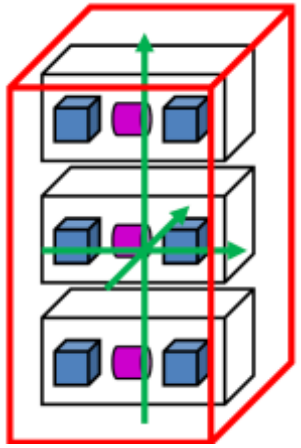
```
G4Step* aStep;
G4double EnergyDeposit = aStep->GetTotalEnergyDeposit();
G4double NonIonizingEnergyDeposit = aStep->GetNonIonizingEnergyDeposit();

G4Track* Track = aStep->GetTrack();
G4double StepLength = aStep->GetStepLength();
G4bool IsFirstStepInVolume = aStep->IsFirstStepInVolume();
G4bool IsLastStepInVolume = aStep->IsLastStepInVolume();

G4StepPoint* preStepPoint = aStep->GetPreStepPoint();
G4TouchableHistory* theTouchable = (G4TouchableHistory*)(preStepPoint->GetTouchable());

G4int copyNo = theTouchable->GetVolume()->GetCopyNo();
G4int motherCopyNo = theTouchable->GetVolume(1)->GetCopyNo();
G4int grandMotherCopyNo = theTouchable->GetVolume(2)->GetCopyNo();

G4ThreeVector worldPos = preStepPoint->GetPosition();
G4ThreeVector localPos =
              theTouchable->GetHistory()->GetTopTransform().TransformPoint(worldPos);
```



Przykład implementacji detektora śledzącego – 7 (definicja funkcji `ProcessHits`)

```
G4bool MyDetector::ProcessHits(G4Step* aStep, G4TouchableHistory* ROhist)
{
  MyHit* aHit = new MyHit();
  G4StepPoint* preStepPoint = aStep->GetPreStepPoint();
  G4TouchableHistory* theTouchable = (G4TouchableHistory*)(preStepPoint->GetTouchable());
  G4ThreeVector worldPos = preStepPoint->GetPosition();
  G4ThreeVector localPos =
                theTouchable->GetHistory()->GetTopTransform().TransformPoint(worldPos);
  G4double kinEnergy = aStep->GetTrack()->GetDynamicParticle()->GetKineticEnergy();
  aHit->SetPosition(localPos);
  aHit->SetKineticEnergy(kinEnergy);
  ...
  hitsCollection->insert(aHit);
  return true;
}
```

- Metoda `ProcessHits()` jest wywoływana dla **każdego kroku** wykonanego w objętości logicznej, do której „podpięty” jest nasz detektor (chyba, że zadziałał filtr cząstek).
- W tej metodzie produkujemy nowy hit na podstawie danych zawartych w obiekcie `G4Step` otrzymywanym jako parametr. Nie ma obowiązku tworzyć hitu (np. jeśli cząstka nas nie interesuje lub nie było depozycji)
- Wytworzony i wypełniony danymi hit dołączamy do naszej kolekcji (kontenera).
- Zwracana wartość logiczna nie jest obecnie używana (być może będzie w przyszłych wersjach Geanta).
- Dane geometryczne nt. objętości logicznej, w której był wykonany krok, należy pobierać z **`PreStepPoint`**.

Przykład implementacji detektora kalorymetrycznego

Przykład implementacji kalorymetru – 1 (deklaracja klasy)

CalorimeterSD.hh

https://geant4-userdoc.web.cern.ch/Doxygen/examples_doc/html/classB4c_1_1CalorimeterSD.html
https://geant4-userdoc.web.cern.ch/Doxygen/examples_doc/html/CalorimeterSD_8hh_source.html
https://geant4-userdoc.web.cern.ch/Doxygen/examples_doc/html/CalorimeterSD_8cc_source.html

```
#include "G4VSensitiveDetector.hh"
#include "CalorHit.hh"
class G4Step;
class G4HCofThisEvent;

class CalorimeterSD : public G4VSensitiveDetector
{
  public:
    CalorimeterSD(const G4String& name,
                  const G4String& hitsCollectionName,
                  G4int nofCells);
    ~CalorimeterSD() override;

    // methods from base class
    void Initialize(G4HCofThisEvent* hitCollection) override;
    G4bool ProcessHits(G4Step* step, G4TouchableHistory* history) override;
    void EndOfEvent(G4HCofThisEvent* hitCollection) override;

  private:
    CalorHitsCollection* fHitsCollection = nullptr;
    G4int fNofCells = 0;
 };
```

Przykład implementacji kalorymetru – 2 (konstruktor)

CalorimeterSD.cc

```
CalorimeterSD::CalorimeterSD(const G4String& name,
                             const G4String& hitsCollectionName,
                             G4int nofCells)
: G4VSensitiveDetector(name),
  fNofCells(nofCells)
{
  collectionName.insert(hitsCollectionName);
}
```

Przykład implementacji kalorymetru – 3 (definicja funkcji `Initialize`)

CalorimeterSD.cc

```
void CalorimeterSD::Initialize(G4HCofThisEvent* hce)
{
  // Create hits collection
  fHitsCollection = new CalorHitsCollection(SensitiveDetectorName, collectionName[0]);

  // Add this collection in hce
  auto hcID = G4SDManager::GetSDMpointer()->GetCollectionID(collectionName[0]);
  hce->AddHitsCollection( hcID, fHitsCollection );


  // Create hits
  // fNofCells for cells + one more for total sums
  for (G4int i=0; i < fNofCells + 1; i++ ) {
    fHitsCollection->insert(new CalorHit());
  }
}
```

Tworzymy kontener na nasze własne hity, które wyprodukujemy podczas właśnie się rozpoczynającego eventu.
Nazwę kontenera już stworzyliśmy w konstruktorze.

Pobieramy identyfikator kontenera hitów przydzielony przez G4SDManager.

Rejestrujemy nasz kontener na liście wszystkich innych kontenerów bieżącego eventu (`G4HCofThisEvent`).

Przykład implementacji kalorymetru – 4 (definicja funkcji `ProcessHits`)

CalorimeterSD.cc

```
G4bool CalorimeterSD::ProcessHits(G4Step* step, G4TouchableHistory*)
{
  // energy deposit
   auto edep = step->GetTotalEnergyDeposit();
   if ( edep==0.) return false;

   auto touchable = (step->GetPreStepPoint()->GetTouchable());
   // Get calorimeter cell id
   auto layerNumber = touchable->GetReplicaNumber(1);

   // Get hit accounting data for this cell
   auto hit = (*fHitsCollection)[layerNumber];
   if ( ! hit ) {
     G4ExceptionDescription msg;
     msg << "Cannot access hit " << layerNumber;
     G4Exception("CalorimeterSD::ProcessHits()", "MyCode0004", FatalException, msg);
   }

   // Get hit for total accounting
   auto hitTotal = (*fHitsCollection)[fHitsCollection->entries()-1];

   // Add values
   hit->Add(edep);
   hitTotal->Add(edep);

   return true;
 }
```

Przykład implementacji kalorymetru – 5 (definicja funkcji `EndOfEvent`)

CalorimeterSD.cc

```
G4bool CalorimeterSD::EndOfEvent(G4HCofThisEvent*)
{
  if ( verboseLevel>1 ) {
    auto nofHits = fHitsCollection->entries();
    G4cout << "Hits Collection: in this event there are " << nofHits << " hits." << G4endl;
    for ( std::size_t i=0; i<nofHits; ++i ) (*fHitsCollection)[i]->Print();
  }
}
```

- Funkcja (metoda) `EndOfEvent` jest wywoływana na końcu przetwarzania eventu
  - ale jeszcze przed wywołaniem metody `UserEndOfEventAction`,
  - jest wywoływana nawet jeśli zostało wymuszone przerwanie przetwarzania eventu.

Uwaga na marginesie
std::size_t is the unsigned integer type (of the result of the sizeof operator)
std::size_t can store the maximum size of a theoretically possible object of any type (including array).
std::size_t is commonly used for array indexing and loop counting. Programs that use other types, such as unsigned int, for array indexing may fail on, e.g. 64-bit systems.

Geant4. Dostęp i gromadzenie wyników symulacji

- Standardowo symulacja przebiega bez gromadzenia jakichkolwiek wyników.
- Użytkownik musi sam określić jakie dane i w jaki sposób chce gromadzić.
- Znam 3½ zalecanych sposobów zbierania wyników symulacji:

1. Proste liczniki (primitive scorer) obsługiwane z poziomu poleceń interfejsu użytkownika. Działają w geometrii równoległej, niezależnej od rzeczywistej geometrii modelu zdefiniowanego w `G4VUserDetectorConstruction`.

2. **Zdefiniowanie objętości logicznej (bryły poziomu `G4LogicalVolume`) jako elementu aktywnego i przypisanie jej detektora:**

   2a. Implementując własne klasy pochodne od `G4VSensitiveDetector` i `G4VHit`.

   2b. **Stosując gotową klasę `G4MultiFunctionalDetector`, która jest konkretną implementacją klasy `G4VSensitiveDetector` opartą o proste liczniki (obiekty typu `G4VPrimitiveScore`).**

   W obu wypadkach, po zakończeniu eventu mamy dostęp, do pełnego zbioru hitów (`G4THitsCollection`) lub mapy wielkości akumulowanych w trakcie eventu (`G4THitsMap`) najczęściej indeksowanej numerem kopii obszaru logicznego. Użytkownik ma dostęp do tych danych korzystając z klasy `G4UserEventAction` lub własnej klasy pochodnej od `G4Run`.

3. Korzystając z własnych implementacji klas `G4UserTrackingAction`, **`G4UserSteppingAction`** i `G4UserStackingAction`. W ten sposób również mamy dostęp do pełnej informacji ale sami musimy zadbać o jej przetwarzanie w trakcie procesowania przypadku.

Geometria – jak zrobić własne DetctorConstruction?

```
// DetectorConstruction.hh

#ifndef DetectorConstruction_h
#define DetectorConstruction_h 1

#include "G4VUserDetectorConstruction.hh"
#include "G4NistManager.hh"
class G4LogicalVolume;
class G4VPhysicalVolume;
class G4Box;
class G4Tubs;

class DetectorConstruction : public G4VUserDetectorConstruction
{
  public:
    DetectorConstruction();
    virtual ~DetectorConstruction();
    virtual G4VPhysicalVolume* Construct();
    void ConstructSDandField();

    void SetWorldMaterial(G4String newMaterial);
    void SetupPhantom();
    void SetPDDPhantomNLayers(G4int newNLayers);
    G4int GetPDDPhantomNLayers();
    void SetPDDPhantomLayerThickness(G4double newPDDPhantomLayerThickness);

  private:
  ...
};
#endif
```

Interfejs publiczny

Tu definiujemy geometrię

Tu definiujemy detektor(y)

Implementacja. Prywatna.

Tworzenie detektora z użyciem G4MultiFunctionalDetector

- *Detektor(y) powinien być stworzony i powiązany z objętością logiczną w funkcji składowej* `ConstructSDandField()` *klasy DetectorConstruction*

```
void DetectorConstruction::ConstructSDandField()
{
  G4MultiFunctionalDetector* MFD = new G4MultiFunctionalDetector("MyDetector");
  G4VPrimitiveScorer* primitive = new G4PSEnergyDeposit("eDep", 0);
  MFD->RegisterPrimitive(primitive);
  primitive = new G4PSTrackCounter("TrackCounter",1,0);
  MFD->RegisterPrimitive(primitive);

  G4SDManager::GetSDMpointer()->AddNewDetector(MFD);
  SetSensitiveDetector("MyLVName", MFD);
  // lub
  // SetSensitiveDetector(MyLVptr, MFD);
  // lub
  // MyLVptr->SetSensitiveDetector(MFD);
}
```

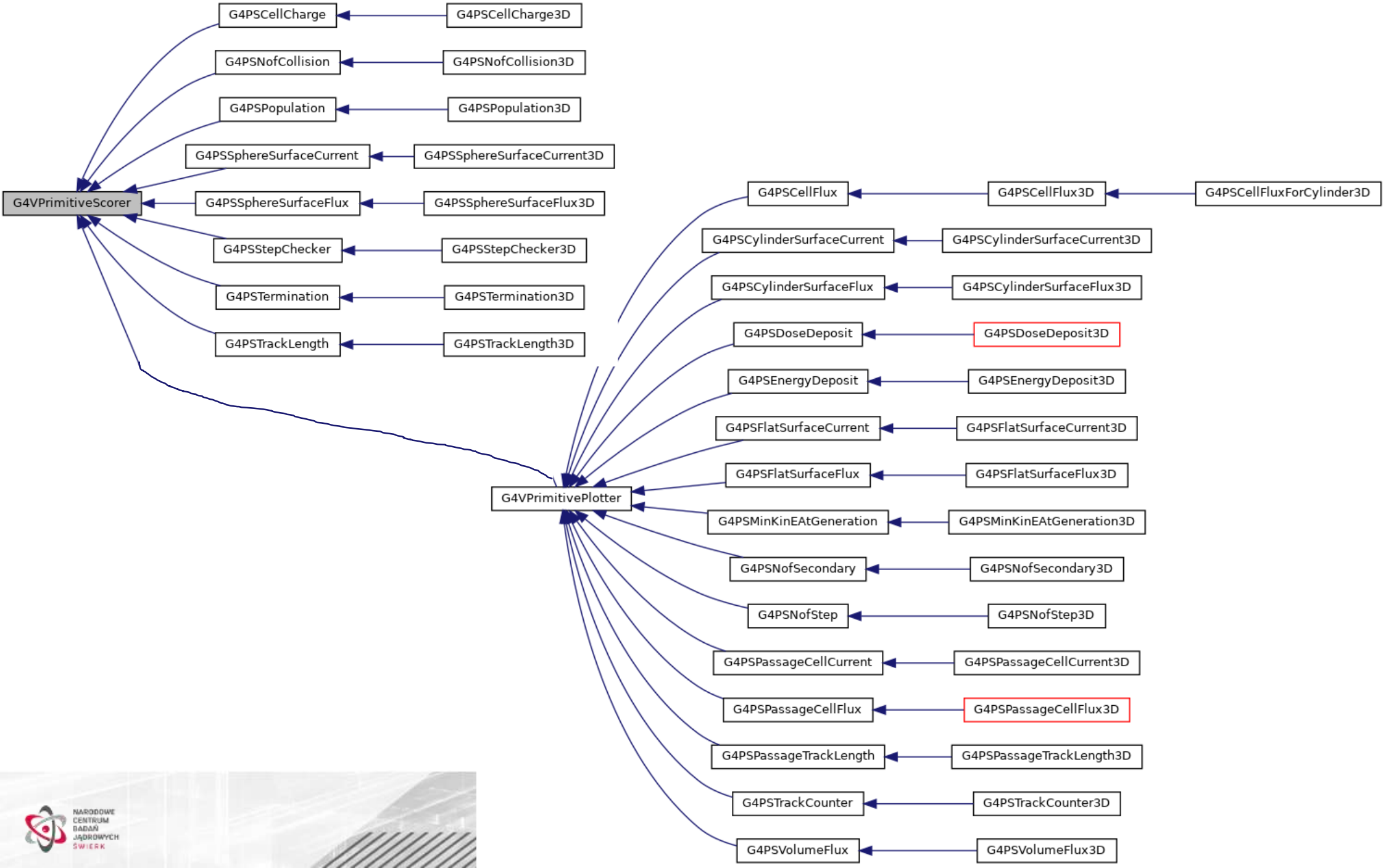
Poziom hierarchii geometrycznej do indeksowania mapy
0 – córka
1 – matka
2 – babcia

Kierunek cząstki
0 – wchodzi lub wychodzi
1 – wchodzi
2 – wychodzi

- *Każdy detektor musi mieć unikalną nazwę. Każdy licznik musi mieć unikalną nazwę.*
- *Do jednego* **`G4MultiFunctionalDetector`** *można zarejestrować wiele prostych liczników (primitive).*
- *Kontenery hitów są mapami. Są automatycznie tworzone i wypełniane.*
- *Nazwy kontenerów są automatycznie nadawane wg. wzorca „Nazwa detektora”/„nazwa licznika”, np. „MyDetector/eDep”.*
- *Detektor należy zarejestrować do menadżera detektorów* (**`G4SDManager`**).
- *Przypisanie detektora do objętości logicznej można zrobić na różne sposoby: albo za pomocą nazwy albo za pomocą wskaźników.*
- *Ale jedna objętość logiczna może mieć przypisany tylko jeden detektor.*

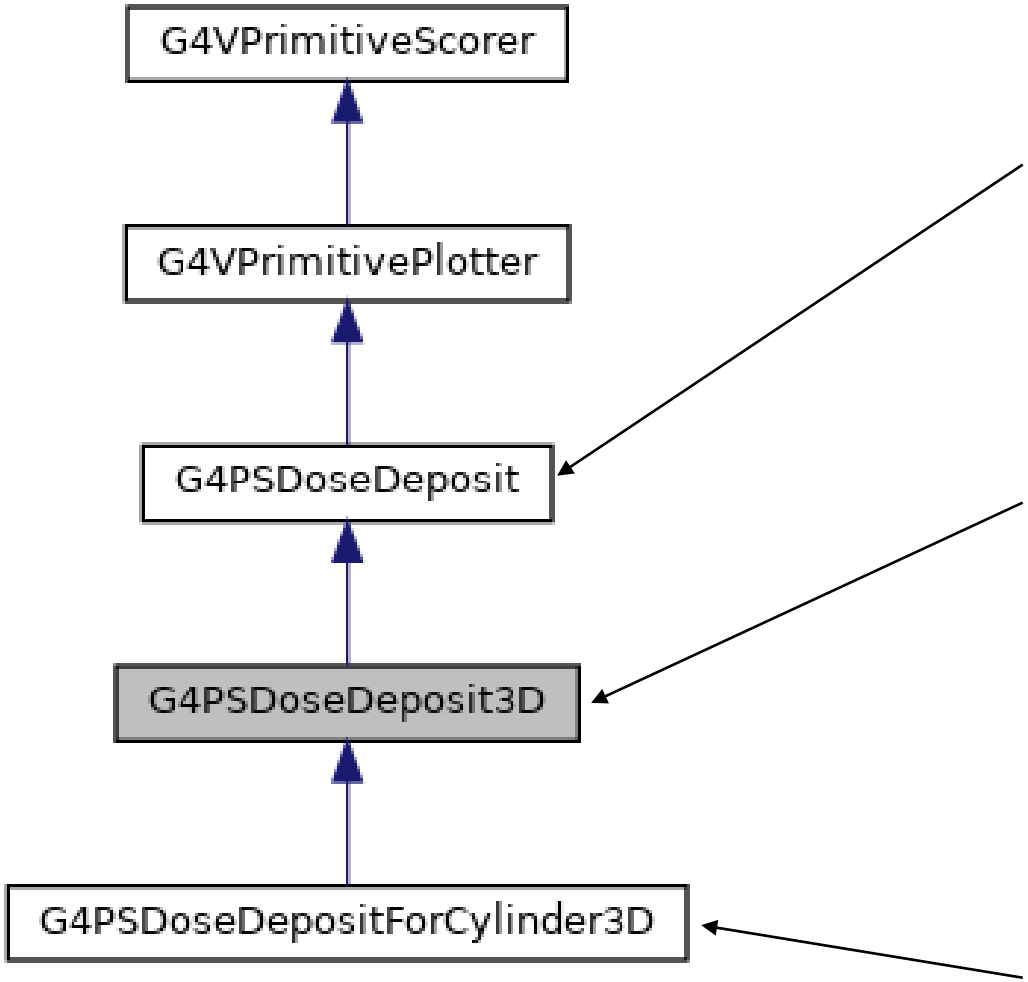
Tworzenie detektora z użyciem `G4MultiFunctionalDetector` - liczniki

https://geant4.kek.jp/Reference/v11.0.1/classG4VPrimitiveScorer.html



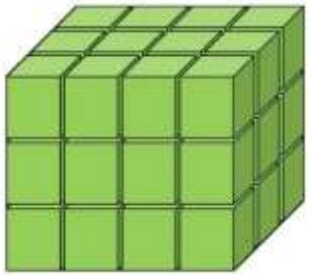
Tworzenie detektora z użyciem `G4MultiFunctionalDetector` - liczniki

https://geant4.kek.jp/Reference/v11.0.1/classG4VPrimitiveScorer.html

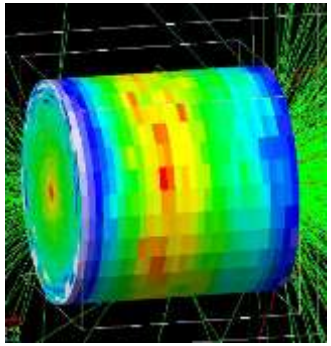


Tej klasy użyjemy jeśli zbieramy dawkę z pojedynczej objętości lub z prostej repliki (do identyfikacji kopii wystarczy prosty indeks)

Tej klasy użyjemy jeśli zbieramy dawkę na siatce ortogonalnej (trzykrotna replikacja w układzie kartezjańskim)



Tej klasy użyjemy jeśli zbieramy dawkę na siatce cylindrycznej (trzykrotna replikacja w układzie cylindrycznym)



Podsumowanie: Sensitive Detector VS Primitive Scorer

Sensitive Detector

- Użytkownik musi zaimplementować własne klasy dla detektora i hitów.
- Hit może jednocześnie gromadzić wiele wielkości fizycznych. Hit może być albo generowany osobno dla każdego kroku albo może akumulować dane podczas eventu.
- Zwykle tworzony jest tylko jeden kontener hitów dla detektora.
- Kontener hitów zwykle jest relatywnie zwięzły.

Primitive scorer

- Kilkanaście liczników dostępnych w bibliotece. Można dodać własne.
- Każdy licznik akumuluje jedną wielkość fizyczną podczas eventu. Informacja z poszczególnych kroków nie jest dostępna.
- Jedna siatka (`G4MultiFunctionalDetector`) generuje wiele kontenerów (map) – po jednym dla każdego licznika.
- Klucze map są nadmiarowe (w przypadku liczników powiązanych z tą samą siatką)

Często ten sam efekt da się uzyskać na oba sposoby. W związku z tym można spotkać następujące zalecenia:

- Stosuj proste liczniki (primitive scorers)
  - jeśli interesują Cię tylko wielkości akumulowane podczas całego eventu/runu i nie potrzebne Ci dane z poziomu pojedynczych kroków, oraz
  - nie potrzeba zbyt wielu liczników.
- W innych wypadkach przemyśl implementację własnego detektora.

Analiza i zapis danych z detektorów na dysk

Analiza i zapis danych z detektorów na dysk

Event

`HitCollectionsOfThisEvent`

Po zakończeniu przetwarzania eventu, w pamięci istnieje obiekt typu `G4Event` *z przypisanym zbiorem wyprodukowanych w trakcie eventu hitów (podzielonych na kolekcje/kontenery od poszczególnych detektorów).*

Na rzecz tego obiektu zostaną automatycznie wywołane dwie funkcje (oczywiście, o ile je wcześniej zarejestrujemy do RunManagera…), które możemy wykorzystać do przeanalizowania wyników i/lub zapisu ich na dysk.

Analiza i zapis danych – sposób 1

Przykład Basic 4c
B4c::EventAction

Event

`HitCollectionsOfThisEvent`

`UserEventAction()`

`EndOfEventAction``(const G4Event* evt)`

- *Pobieramy HitCollections:*
  `evt->GetHCofThisEvent()->GetHC(hcID);`
- *Sprawdzamy czy jest coś wartego zapisania*
- *Wypełniamy histogramy/ntuple*

Jeśli zarejestrujemy do RunManagera naszą klasę `UserEventAction` *to metoda* `EndOfEventAction` *będzie automatycznie wywoływana na końcu przetwarzania każdego eventu.*

Analiza i zapis danych – sposób 1

Przykład Basic 4c
B4c::EventAction

◆ EndOfEventAction()

void B4c::EventAction::EndOfEventAction (const G4Event * event)

Definition at line **104** of file **EventAction.cc**.

```
105 {
106   // Get hits collections IDs (only once)
107   if ( fAbsHCID == -1 ) {
108     fAbsHCID
109       = G4SDManager::GetSDMpointer()->GetCollectionID("AbsorberHitsCollection");
110     fGapHCID
111       = G4SDManager::GetSDMpointer()->GetCollectionID("GapHitsCollection");
112   }
113
114   // Get hits collections
115   auto absoHC = GetHitsCollection(fAbsHCID, event);
116   auto gapHC = GetHitsCollection(fGapHCID, event);
117
```

```
134   // Fill histograms, ntuple
135   //
136
137   // get analysis manager
138   auto analysisManager = G4AnalysisManager::Instance();
139
140   // fill histograms
141   analysisManager->FillH1(0, absoHit->GetEdep());
142   analysisManager->FillH1(1, gapHit->GetEdep());
143   analysisManager->FillH1(2, absoHit->GetTrackLength());
144   analysisManager->FillH1(3, gapHit->GetTrackLength());
145
146   // fill ntuple
147   analysisManager->FillNtupleDColumn(0, absoHit->GetEdep());
148   analysisManager->FillNtupleDColumn(1, gapHit->GetEdep());
149   analysisManager->FillNtupleDColumn(2, absoHit->GetTrackLength());
150   analysisManager->FillNtupleDColumn(3, gapHit->GetTrackLength());
151   analysisManager->AddNtupleRow();
152 }
```

Analiza i zapis danych – sposób 1

Przykład Basic 4c
B4c::EventAction
B4::RunAction

Event

`HitCollectionsOfThisEvent`

UserEventAction()

EndOfEventAction(const G4Event* evt)

- *Pobieramy HitCollections:*
  `evt->GetHCofThisEvent()->GetHC(hcID);`
- *Sprawdzamy czy jest coś wartego zapisania*
- *Wypełniamy histogramy/ntuple*

UserRunAction()

Konstruktor **UserRunAction**()

- *Tworzymy* `G4AnalysisManager`
- *Rejestrujemy histogramy/ntuple*

BeginOfRunAction(const G4Run* aRun)

- *Tworzymy nazwę pliku dyskowego*
- *Prosimy AnalysisManagera o otwarcie pliku dyskowego*

EndOfRunAction(const G4Run* aRun)

- *Prosimy AnalysisManagera by zapisał histogramy/ntuple do pliku*
- *Prosimy AnalysisManagera by zamknął plik*

Analiza i zapis danych – sposób 1

Przykład Basic 4c
B4::RunAction

RunAction()

B4::RunAction::RunAction ( )

Definition at line **43** of file **RunAction.cc**.

```
44  {
45    // set printing event number per each event
46    G4RunManager::GetRunManager()->SetPrintProgress(1);
47
48    // Create analysis manager
49    // The choice of the output format is done via the specified
50    // file extension.
51    auto analysisManager = G4AnalysisManager::Instance();
52
53    // Create directories
54    //analysisManager->SetHistoDirectoryName("histograms");
55    //analysisManager->SetNtupleDirectoryName("ntuple");
56    analysisManager->SetVerboseLevel(1);
57    analysisManager->SetNtupleMerging(true);
58      // Note: merging ntuples is available only with Root output
59
60    // Book histograms, ntuple
61    //
62
63    // Creating histograms
64    analysisManager->CreateH1("Eabs","Edep in absorber", 100, 0., 800*MeV);
65    analysisManager->CreateH1("Egap","Edep in gap", 100, 0., 100*MeV);
66    analysisManager->CreateH1("Labs","trackL in absorber", 100, 0., 1*m);
67    analysisManager->CreateH1("Lgap","trackL in gap", 100, 0., 50*cm);
68
69    // Creating ntuple
70    //
71    analysisManager->CreateNtuple("B4", "Edep and TrackL");
72    analysisManager->CreateNtupleDColumn("Eabs");
73    analysisManager->CreateNtupleDColumn("Egap");
74    analysisManager->CreateNtupleDColumn("Labs");
75    analysisManager->CreateNtupleDColumn("Lgap");
76    analysisManager->FinishNtuple();
77  }
```

Analiza i zapis danych – sposób 1

◆ BeginOfRunAction()

void B4::RunAction::BeginOfRunAction (const G4Run *)

Definition at line **87** of file **RunAction.cc**.

```
 88 {
 89   //inform the runManager to save random number seed
 90   //G4RunManager::GetRunManager()->SetRandomNumberStore(true);
 91
 92   // Get analysis manager
 93   auto analysisManager = G4AnalysisManager::Instance();
 94
 95   // Open an output file
 96   //
 97   G4String fileName = "B4.root";
 98   // Other supported output types:
 99   // G4String fileName = "B4.csv";
100   // G4String fileName = "B4.hdf5";
101   // G4String fileName = "B4.xml";
102   analysisManager->OpenFile(fileName);
103   G4cout << "Using " << analysisManager->GetType() << G4endl;
104 }
```

◆ EndOfRunAction()

void B4::RunAction::EndOfRunAction (const G4Run *)

Definition at line **108** of file **RunAction.cc**.

```
109 {
143   // save histograms & ntuple
144   //
145   analysisManager->Write();
146   analysisManager->CloseFile();
147 }
```

Histogramy w pamięci są resetowane po zamknięciu pliku

Analiza i zapis danych – sposób 1

Przykład Basic 4c
B4c::EventAction
B4::RunAction

Event

`HitCollectionsOfThisEvent`

`UserEventAction()`

`EndOfEventAction``(const G4Event* evt)`
- *Pobieramy HitCollections:*
  `evt->GetHCofThisEvent()->GetHC(hcID);`
- *Sprawdzamy czy jest coś wartego zapisania*
- *Wypełniamy histogramy/ntuple*

- ***Używanie G4AnalysisManagera nie jest obowiązkowe.***
- ***Można samemu obsługiwać pliki i zapisywać do nich we własnym formacie.***
  - *Tworzymy* `G4AnalysisManager`
  - *Rejestrujemy histogramy/ntuple*

`BeginOfRunAction``(const G4Run* aRun)`
- *Tworzymy nazwę pliku dyskowego*
- *Prosimy AnalysisManagera o otwarcie pliku dyskowego*

`EndOfRunAction``(const G4Run* aRun)`
- *Prosimy AnalysisManagera by zapisał histogramy/ntuple do pliku*
- *Prosimy AnalysisManagera by zamknął plik*

Analiza i zapis danych – sposób 2

Event

`HitCollectionsOfThisEvent`

Po zakończeniu przetwarzania eventu, w pamięci istnieje obiekt typu `G4Event` *z przypisanym zbiorem wyprodukowanych w trakcie eventu hitów (podzielonych na kolekcje/kontenery od poszczególnych detektorów).*

Na rzecz tego obiektu zostaną automatycznie wywołane dwie funkcje (oczywiście, o ile je wcześniej zarejestrujemy do RunManagera…), które możemy wykorzystać do przeanalizowania wyników i/lub zapisu ich na dysk.

Analiza i zapis danych – sposób 2

Event

`HitCollectionsOfThisEvent`

`UserRunAction()`

`GenerateRun``()`

- *Tworzymy nasz własny, rozszerzony obiekt pochodny od* `G4Run`
- `return MyRun;`

`MyRun()`

Konstruktor **`MyRun`**`()`

- *Tworzymy* `G4AnalysisManager`
- *Rejestrujemy histogramy/ntuple (o ile już nie istnieją)*
- *Tworzymy nazwę pliku dyskowego*
- *Prosimy AnalysisManagera o otwarcie pliku dyskowego*

`RecordEvent(const G4Event* evt)`

- *Pobieramy HitCollections:* `evt->GetHCofThisEvent()->GetHC(hcID);`
- *Sprawdzamy czy jest coś wartego zapisania*
- *Wypełniamy histogramy/ntuple*

`Destruktor ~MyRun()`

- *Prosimy AnalysisManagera by zapisał histogramy/ntuple do pliku*
- *Prosimy AnalysisManagera by zamknął plik*

Analiza i zapis danych – sposób 2

Event

`HitCollectionsOfThisEvent`

`UserRunAction()`

`GenerateRun()`

- *Tworzymy nasz własny, rozszerzony obiekt pochodny od* `G4Run`
- `return MyRun;`

`MyRun()`

Konstruktor **`MyRun()`**

Ten sposób ma zalety w przypadku pracy wielowątkowej

- ***mechanizmy sumowania wyników z równoległych wątków.***

`RecordEvent(const G4Event* evt)`

- *Pobieramy HitCollections:* `evt->GetHCofThisEvent()->GetHC(hcID);`
- *Sprawdzamy czy jest coś wartego zapisania*
- *Wypełniamy histogramy/ntuple*

`Destruktor ~MyRun()`

- *Prosimy AnalysisManagera by zapisał histogramy/ntuple do pliku*
- *Prosimy AnalysisManagera by zamknął plik*

Dziękuję za uwagę

NARODOWE CENTRUM BADAŃ JĄDROWYCH
ŚWIERK

www.ncbj.gov.pl