



Geant4 @ NCBJ, 2022

2. Podstawy czyli wyprawa do wnętrza kodu

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. Źródło. Fizyka. Wizualizacja.
5. Detektory typu „primitive scorer”, „probe”. (Phasespace).
6. Detektory użytkownika („Hits”).
7. Obiekty typu „UserAction” jako detektory. Histogramy i n-tuple. Niepewność statystyczna w obliczeniach Monte Carlo. Geant4 na klastrze CiŚ.

* „Monte Carlo First Run
(wykład bonusowy)

Przegląd zagadnień pozostawionych na przyszłość:

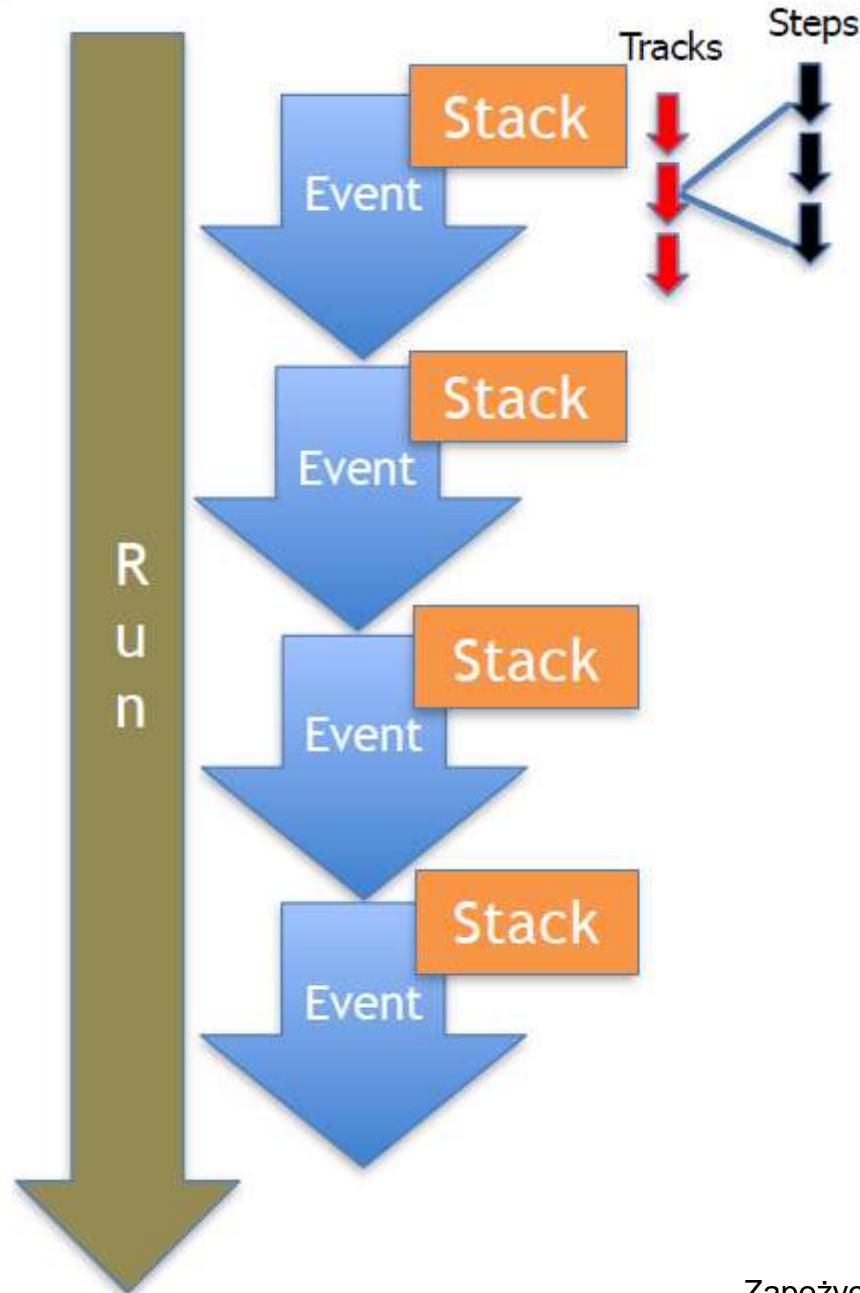
- wielowątkowość („multithreading”),
- własne interfejsy („messengers”),
- interfejs Roota (histogramy, n-tuple), interfejs python
- redukcja wariancji, „physics biasing”, „event biasing”, „geometrical biasing”,
- fotony optyczne, fizyka hadronowa, procesy i cząstki użytkownika,
- obcięcia energetyczne 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.

Żeby to wszystko osiągnąć musimy poznać podstawową strukturę kodu Geant4

Geant4 – podstawowa struktura



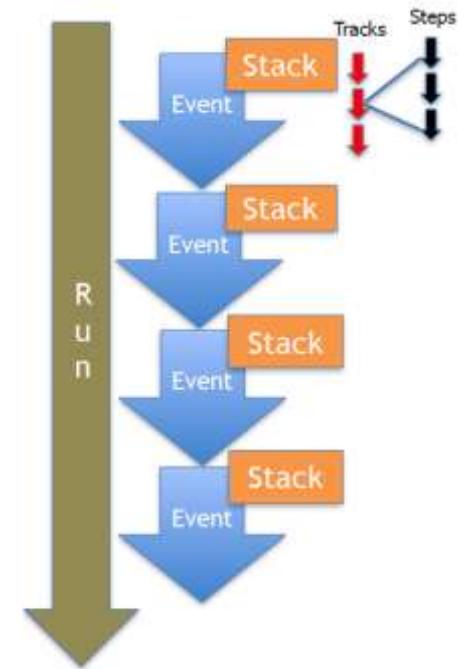
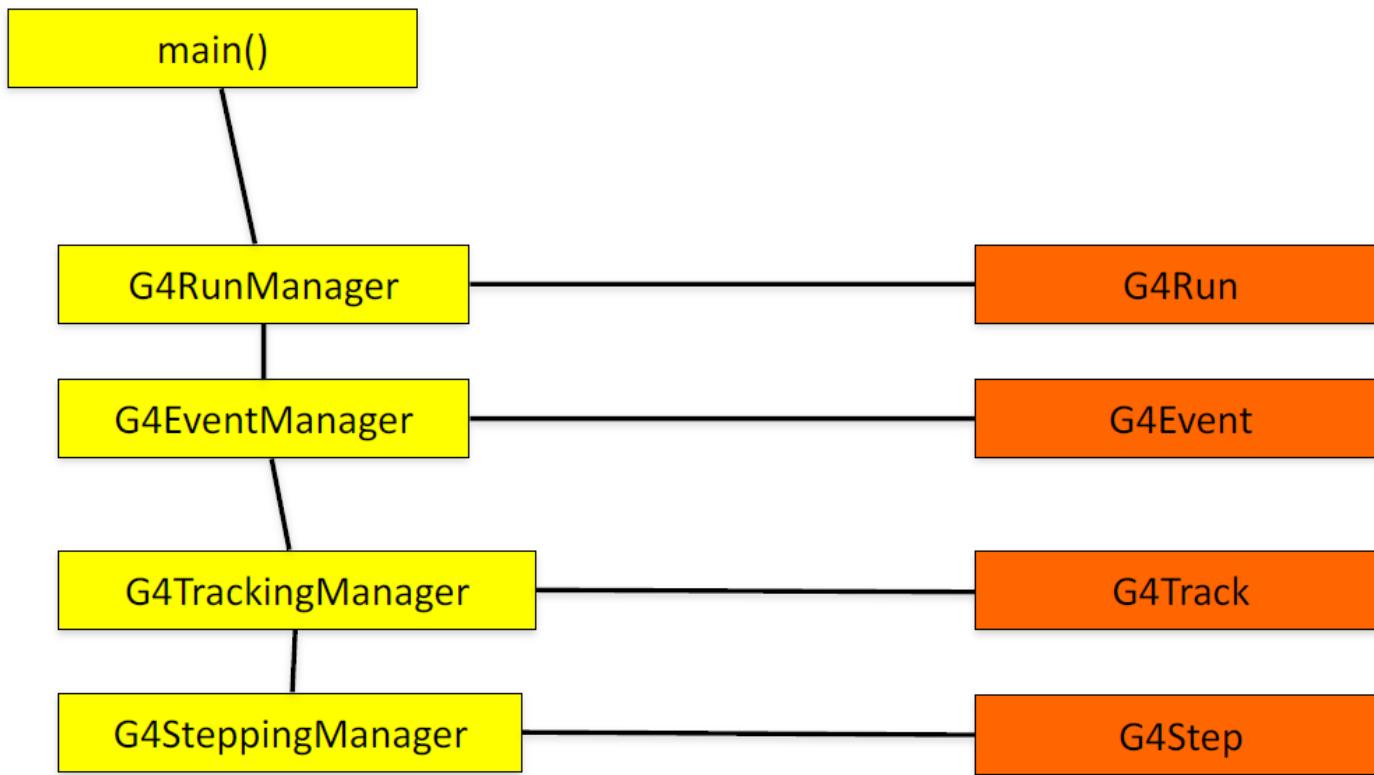
Zapożyczone od M. Asai za pośrednictwem J.Apostolakis

Geant4 – podstawowa struktura

G4RunManager

- „Główny rozgrywający” - zarządza jądrem Geant4
- Steruje przebiegiem symulacji:
 - dba o stworzenie obiektu klasy G4Run
 - powołuje do życia menadżerów niższych szczebli i wydaje im rozkazy, a oni powołują kolejnych podwładnych, itd.

Geant4 – podstawowa struktura



Zapożyczone od M. Asai za pośrednictwem J.Apostolakis

Geant4 – podstawowa struktura

G4RunManager

- „Główny rozgrywający” - zarządza jądrem Geant4
- Steruje przebiegiem symulacji:
 - dba o stworzenie obiektu klasy G4Run
 - powołuje do życia menadżerów niższych szczebli i wydaje im rozkazy, a oni powołują kolejnych podwładnych, itd.
- „Singleton”*, który musi być „powołany do życia” przez użytkownika
- Użytkownik **musi** stworzyć i zarejestrować do RunManagera trzy obiekty odpowiedzialne za:
 - opis geometrii, detektorów, pola (G4VUserDetectorConstruction)
 - konfigurację cząstek i procesów fizycznych (G4VUserPhysicsList lub G4VModularPhysicsList)
 - generację cząstek pierwotnych (G4VUserPrimaryGeneratorAction)
- Użytkownik **może** zarejestrować do RunManagera dodatkowe obiekty pozwalające na bezpośrednią interwencję (dostęp do danych lub modyfikację przebiegu symulacji) na kolejnych, coraz niższych (głębbszych) poziomach symulacji (G4VUserActionInitialization):
 - G4UserRunAction
 - G4UserEventAction
 - G4UserStackingAction
 - G4UserTrackingAction
 - G4UserSteppingAction

Konwencja nazw klas

Wszystko co zaczyna się od **G4V** („v” jak virtual) oznacza klasę abstrakcyjną (czysto wirtualną).

Służy ona tylko jako klasa bazowa dla klas pochodnych. Definiuje interfejs do komunikacji z jądrem Geanta.

Nie da się stworzyć obiektu klasy abstrakcyjnej.

Klasa w programowaniu obiektowym to jakby sam „przepis” (dla inżynierów: „dokumentacja konstrukcyjna”).

Obiekt to konkretny egzemplarz zbudowany według tego „przepisu”.

Np. klasa: samochód osobowy

obiekt: SEAT LEON o numerze VIN WVGZZZ5NZ8W031284

* Klasa skonstruowana w taki sposób, że da się stworzyć tylko jeden egzemplarz obiektu tej klasy. Za to dostępny z każdego miejsca w programie (globalny).

Geant4 – podstawowa struktura – funkcja main()

moja_mala_aplikacja_geant4.cc

```
#include "G4RunManager.hh"  
#include "MyDetectorConstruction.hh"  
#include "FTFP_BERT.hh"  
#include "MyActionInitialization.hh"
```

Dyrektywy kompilatora.

Dołączanie plików nagłówkowych z deklaracjami klas
(muszą być znane w momencie kompilacji).

```
int main(int argc,char** argv) {  
    G4RunManager* runManager = new G4RunManager;
```

- Operatorem „new” powołujemy „do życia” obiekt typu (klasy) G4RunManager.
- Od tego momentu istnieje on gdzieś w pamięci maszyny.
- Operator „new” zwraca adres tego miejsca w pamięci. Umieszczamy ten adres we wskaźniku „runManager” (gwiazdka „*” po nazwie typu w deklaracji zmiennej oznacza typ wskaźnikowy).
- Dzięki wskaźnikowi możemy posługiwać się nowo utworzonym obiektem, np. wywoływać jego funkcje składowe (co za moment zrobimy).
- Wskaźnik to więcej niż sam adres. To także informacja o tym co znajduje się pod tym adresem.
- Uwaga na marginesie. Typ wskaźnika określa klasę obiektów, na które możemy wskazywać tym wskaźnikiem. W szczególności nie da się użyć wskaźnika do obiektu klasy „int” (reprezentującej liczby całkowite) do pracy na obiekcie typu G4RunManager.

Geant4 – podstawowa struktura – funkcja main()

moja_mala_aplikacja_geant4.cc

```
#include "G4RunManager.hh"
#include "MyDetectorConstruction.hh"
#include "FTFP_BERT.hh"
#include "MyActionInitialization.hh"

int main(int argc,char** argv) {
    G4RunManager* runManager = new G4RunManager; →
    kernel = new G4RunManagerKernel();
eventManager = kernel->GetEventManager();
```

- Operator „new”, oprócz rezerwacji miejsca w pamięci pod nowo tworzony obiekt, uruchamia specjalną funkcję zwaną „konstruktorem klasy”.
- W konstruktorze runManagera tworzone są kolejne obiekty, m.in. tzw. kernel, a za jego pośrednictwem EventManager.

Geant4 – podstawowa struktura – funkcja main()

moja_mala_aplikacja_geant4.cc

```
#include "G4RunManager.hh"
#include "MyDetectorConstruction.hh"
#include "FTFP_BERT.hh"
#include "MyActionInitialization.hh"

int main(int argc,char** argv) {
    G4RunManager* runManager = new G4RunManager;

    runManager->SetUserInitialization(new MyDetectorConstruction());
    G4VModularPhysicsList* MyPhysicsList = new FTFP_BERT;
    runManager->SetUserInitialization(MyPhysicsList);
    runManager->SetUserInitialization(new MyActionInitialization());
```

- Po stworzeniu RunManagera, tworzymy obowiązkowe obiekty odpowiedzialne za geometrię układu, fizykę, źródło cząstek pierwotnych.
- Oczywiście tworzymy je operatorem „new” (który rezerwuje miejsce w pamięci i uruchamia ich konstruktory...)
- Wskaźniki do nowo utworzonych obiektów przekazywane są funkcji „SetUserInitialization”, która wchodzi w skład klasy „G4RunManager”.
Uwaga na marginesie: klasa G4RunManager posiada różne wersje funkcji „SetUserInitialization”, przeznaczone do pracy na argumentach różnych typów (tzw. przeładowanie nazwy funkcji).
- Funkcję „SetUserInitialization” wywołujemy na rzecz konkretnego obiektu klasy „G4RunManager”, do którego mamy wskaźnik (zmienna „runManager”). Służy do tego operator „->”.
- *Uwaga na marginesie. Przy tworzeniu „fizyki” ujawnia się tzw. „polimorfizm” – tworzymy obiekt klasy „FTFP_BERT” ale jego adres umieszczamy we wskaźniku do obiektów klasy „G4VModularPhysicsList”. Można tak zrobić, bo „FTFP_BERT” jest klasą pochodną (dziedziczy) od „G4VModularPhysicsList”. Inaczej mówiąc „G4VModularPhysicsList” jest klasą bazową dla „FTFP_BERT”. Klasa bazowa (w tym wypadku czysto abstrakcyjna) definiuje interfejs, dzięki któremu RunManager potrafi „rozmawiać” z obiektami klas pochodnych.*

Geant4 – podstawowa struktura – funkcja main()

moja_mala_aplikacja_geant4.cc

```
#include "G4RunManager.hh"
#include "MyDetectorConstruction.hh"
#include "FTFP_BERT.hh"
#include "MyActionInitialization.hh"

int main(int argc,char** argv) {
    G4RunManager* runManager = new G4RunManager; → kernel = new G4RunManagerKernel();
    eventManager = kernel->GetEventManager();

    runManager->SetUserInitialization(new MyDetectorConstruction());
    G4VModularPhysicsList* MyPhysicsList = new FTFP_BERT;
    runManager->SetUserInitialization(MyPhysicsList);
    runManager->SetUserInitialization(new MyActionInitialization()); → kernel->DefineWorldVolume(userDetector->Construct(), ...);

    runManager->Initialize(); → if(!geometryInitialized) InitializeGeometry();
                                if(!physicsInitialized) InitializePhysics();
```

- Po utworzeniu wszystkich obowiązkowych obiektów i zarejestrowaniu ich do RunManagera musimy wykonać *inicjalizację*.
Czyli wywołać funkcję „Initialize()” (będącą funkcją składową klasy „G4RunManager”) na rzecz obiektu wskazywanego przez wskaźnik...
- W trakcie inicjalizacji wywoływana jest m.in. funkcja „Construct()” należąca do klasy użytkownika „DetectorConstruction”, odpowiedzialna za „zbudowanie” geometrii i zwrócenie wskaźnika do objętości logicznej reprezentującej „świat”.

Geant4 – podstawowa struktura – funkcja main()

moja_mala_aplikacja_geant4.cc

```
#include "G4RunManager.hh"
#include "MyDetectorConstruction.hh"
#include "FTFP_BERT.hh"
#include "MyActionInitialization.hh"

int main(int argc,char** argv) {
    G4RunManager* runManager = new G4RunManager;

    runManager->SetUserInitialization(new MyDetectorConstruction());
    G4VModularPhysicsList* MyPhysicsList = new FTFP_BERT;
    runManager->SetUserInitialization(MyPhysicsList);
    runManager->SetUserInitialization(new MyActionInitialization());

    runManager->Initialize();

    G4UIExecutive* UI = new G4UIExecutive(argc, argv);
    UI->SessionStart();
```

- Po inicjalizacji RunManagera możemy utworzyć interfejs użytkownika (ponownie wkracza do akcji operator „new”)
- ..i np. uruchomić sesję interaktywną, w której możemy sterować wykonaniem symulacji za pomocą odpowiednich poleceń.

Geant4 – podstawowa struktura – funkcja main()

moja_mala_aplikacja_geant4.cc

```
#include "G4RunManager.hh"
#include "MyDetectorConstruction.hh"
#include "FTFP_BERT.hh"
#include "MyActionInitialization.hh"

int main(int argc,char** argv) {
    G4RunManager* runManager = new G4RunManager;

    runManager->SetUserInitialization(new MyDetectorConstruction());
    G4VModularPhysicsList* MyPhysicsList = new FTFP_BERT;
    runManager->SetUserInitialization(MyPhysicsList);
    runManager->SetUserInitialization(new MyActionInitialization());

    runManager->Initialize();

    G4UIExecutive* UI = new G4UIExecutive(argc, argv);
    UI->SessionStart(); → /run/beamOn 100
```

- Przez analogię z eksperymentami fizyki wysokich energii na akceleratorach (nawet jeśli akurat symulujemy, np. źródło radioizotopowe ^{60}Co albo dawki od promieniowania kosmicznego), symulacja startuje w momencie wywołania funkcji BeamOn() („włącz wiązkę”) obiektu klasy G4RunManager.
- Z poziomu interfejsu użytkownika wywołaniu temu odpowiada wydanie polecenia „/run/beamOn” z parametrem określającym liczbę cząstek pierwotnych (źródła).

Geant4 – podstawowa struktura – funkcja main()

moja_mala_aplikacja_geant4.cc

```
#include "G4RunManager.hh"
#include "MyDetectorConstruction.hh"
#include "FTFP_BERT.hh"
#include "MyActionInitialization.hh"

int main(int argc,char** argv) {
    G4RunManager* runManager = new G4RunManager;

    runManager->SetUserInitialization(new MyDetectorConstruction());
    G4VModularPhysicsList* MyPhysicsList = new FTFP_BERT;
    runManager->SetUserInitialization(MyPhysicsList);
    runManager->SetUserInitialization(new MyActionInitialization());

    runManager->Initialize();

    G4UIExecutive* UI = new G4UIExecutive(argc, argv);
    UI->SessionStart();

    delete UI;
    delete runManager;
    return 0;
}
```

- Po zakończeniu sesji interaktywnej usuwamy obiekt obsługujący interfejs użytkownika oraz RunManagera.
- Do kasowania obiektów służy operator „delete”.
- Operator „delete” najpierw wywołuje, na rzecz niszczonego obiektu, specjalną funkcję składową klasy zwaną „destruktorem”.
- Po zakończeniu pracy destruktora zwalniany jest obszar pamięci zajmowany przez obiekt.

Geant4 – podstawowa struktura – funkcja main()

moja_mala_aplikacja_geant4.cc

```
#include "G4RunManager.hh"
#include "MyDetectorConstruction.hh"
#include "FTFP_BERT.hh"
#include "MyActionInitialization.hh"

int main(int argc,char** argv) {
    G4RunManager* runManager = new G4RunManager; → kernel = new G4RunManagerKernel();
    eventManager = kernel->GetEventManager();

    runManager->SetUserInitialization(new MyDetectorConstruction());
    G4VModularPhysicsList* MyPhysicsList = new FTFP_BERT;
    runManager->SetUserInitialization(MyPhysicsList);
    runManager->SetUserInitialization(new MyActionInitialization()); → kernel->DefineWorldVolume(userDetector->Construct(), ...)

    runManager->Initialize(); → if(!geometryInitialized) InitializeGeometry();
    UI->SessionStart(); → if(!physicsInitialized) InitializePhysics();

    G4UIExecutive* UI = new G4UIExecutive(argc, argv);
    UI->SessionStart(); → /run/beamOn 100

    delete UI;
    delete runManager;
    return 0;
}
```

Wnioski:

- Większość wykonywanego kodu nie jest bezpośrednio widoczna
- Bardzo dużo dzieje się „automagicznie” ;)

Geant4 – czym jest Run?

Po komendzie „/run/beamOn 100” tworzony jest Run.

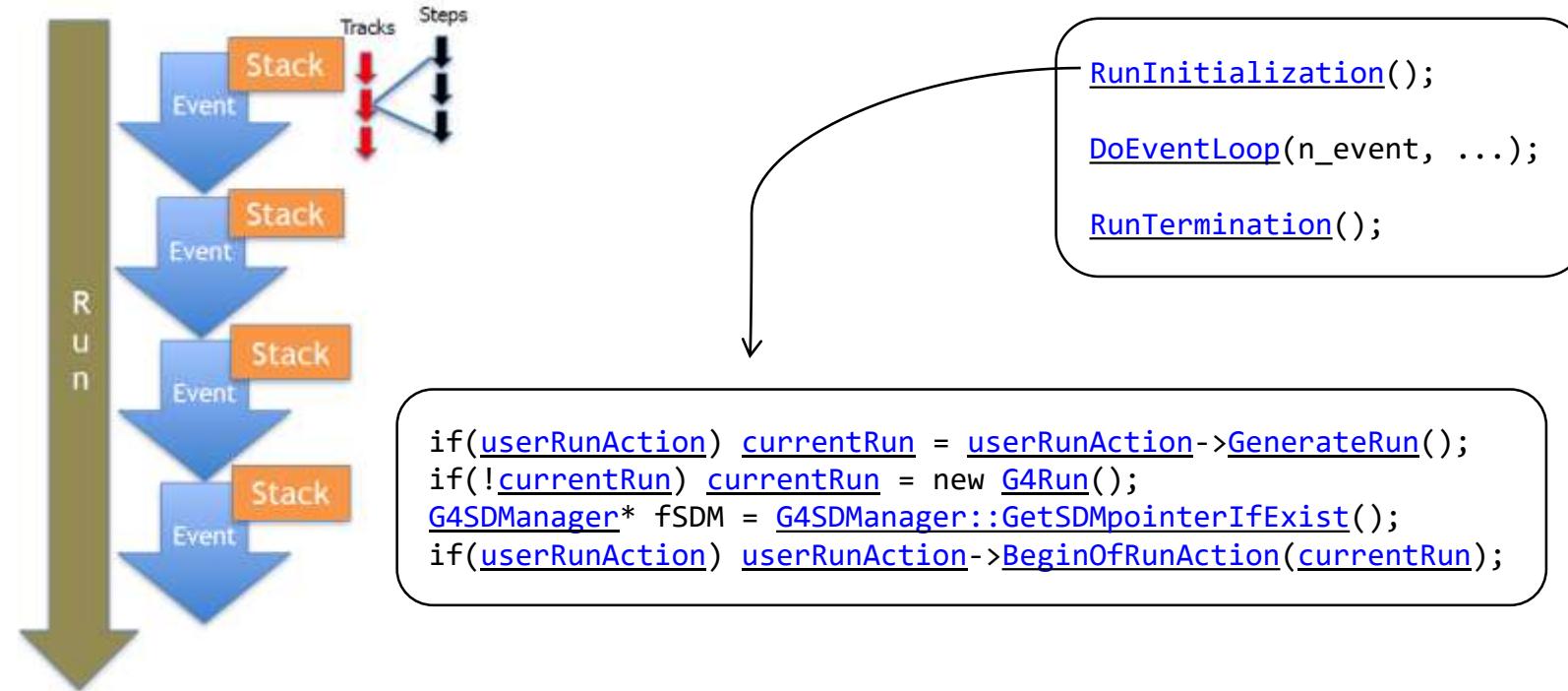
- O utworzenie Runu i zarządzanie nim dba G4RunManager.
- Run jest reprezentowany przez obiekt klasy **G4Run** (lub klasy użytkownika pochodnej od G4Run).
- W trakcie typowego Runu wykonywana jest pojedyncza pętla eventów. (Eventy są przetwarzane sekwencyjnie, jeden po drugim. Chyba, że napisaliśmy aplikację z obsługą wielowątkowości ... wtedy jest N wątków przetwarzających eventy równolegle).
- Na samym początku Runu przygotowywane („zamykane”) są struktury geometryczne i konfiguracja procesów fizycznych
 - geometria jest optymalizowana dla jak najwydajniejszej nawigacji,
 - obliczane są tabele przekrojów czynnych dla materiałów użytych w geometrii.
- Run składa się z konkretnej konfiguracji i zbioru eventów
- Z definicji, przed wystartowaniem runu, użytkownik musiał już powołać do życia:
 - układ (geometria),
 - fizykę (częstki i procesy),
 - źródło cząstek pierwotnych.

I nie wolno ich zmieniać w trakcie trwania runu!

Zapożyczone od M. Asai za pośrednictwem J.Apostolakis

Geant4 – podstawowa struktura

/run/beamOn 100 → G4RunManager::GetRunManager() -> BeamOn(G4int n_event, ...);

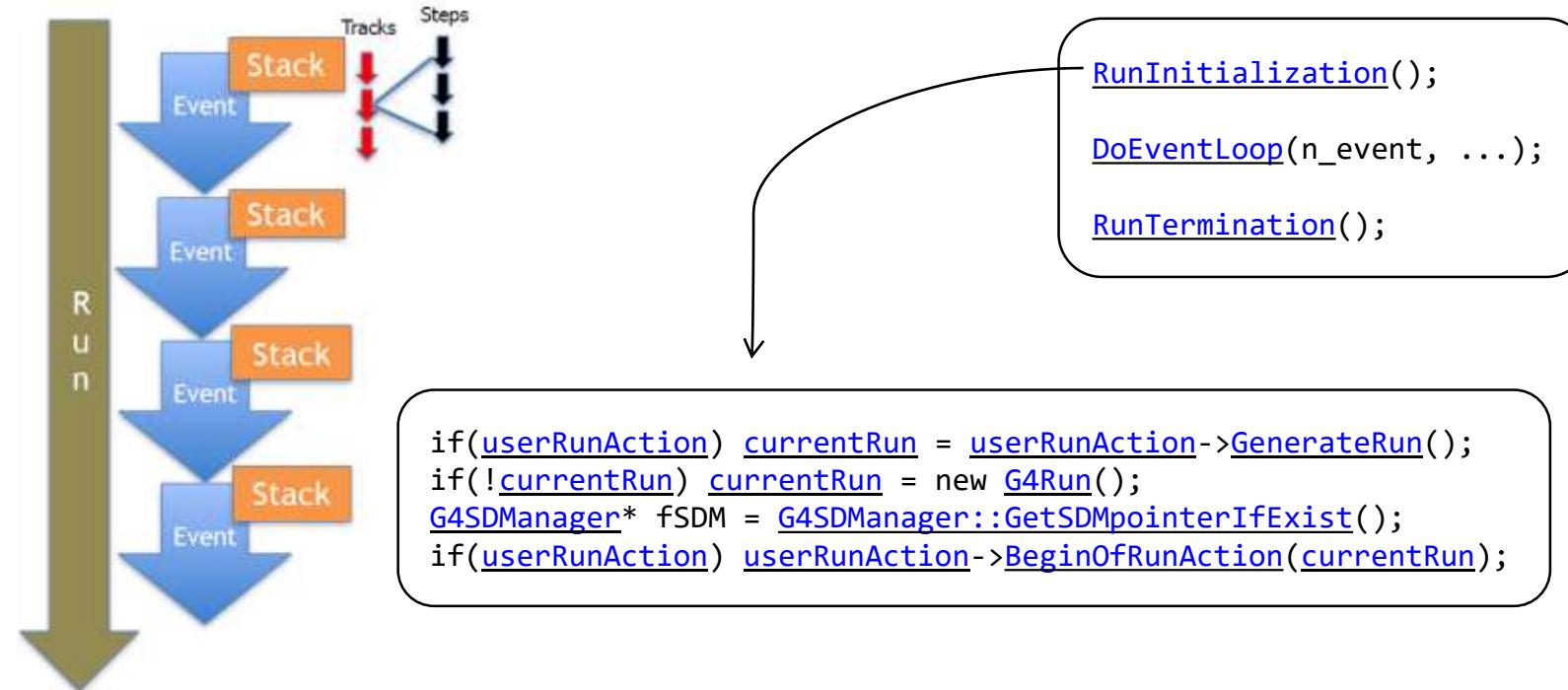


W fazie inicjalizacji

- Tworzony jest obiekt reprezentujący *Run*. Albo standardowego typu *G4Run* albo typu rozszerzonego przez użytkownika (na bazie *G4Run*). Rozszerzenie może polegać np. na dodaniu struktur danych do akumulowania jakiś wielkości fizycznych z wszystkich eventów.
- Pobierany jest wskaźnik do menadżera „Sensitive Detector” (niezerowy jeśli użytkownik stworzył aktywne rejony geometrii lub/i zadeklarował użycie detektorów typu *Primitive Scorer*).
- Jeśli istnieje, zostaje uruchomiona opcjonalna funkcja dostarczona przez użytkownika „*BeginOfRunAction*”. Otrzyma ona wskaźnik do obiektu reprezentującego *Run* w momencie tuż przed rozpoczęciem pętli eventów.

Geant4 – podstawowa struktura

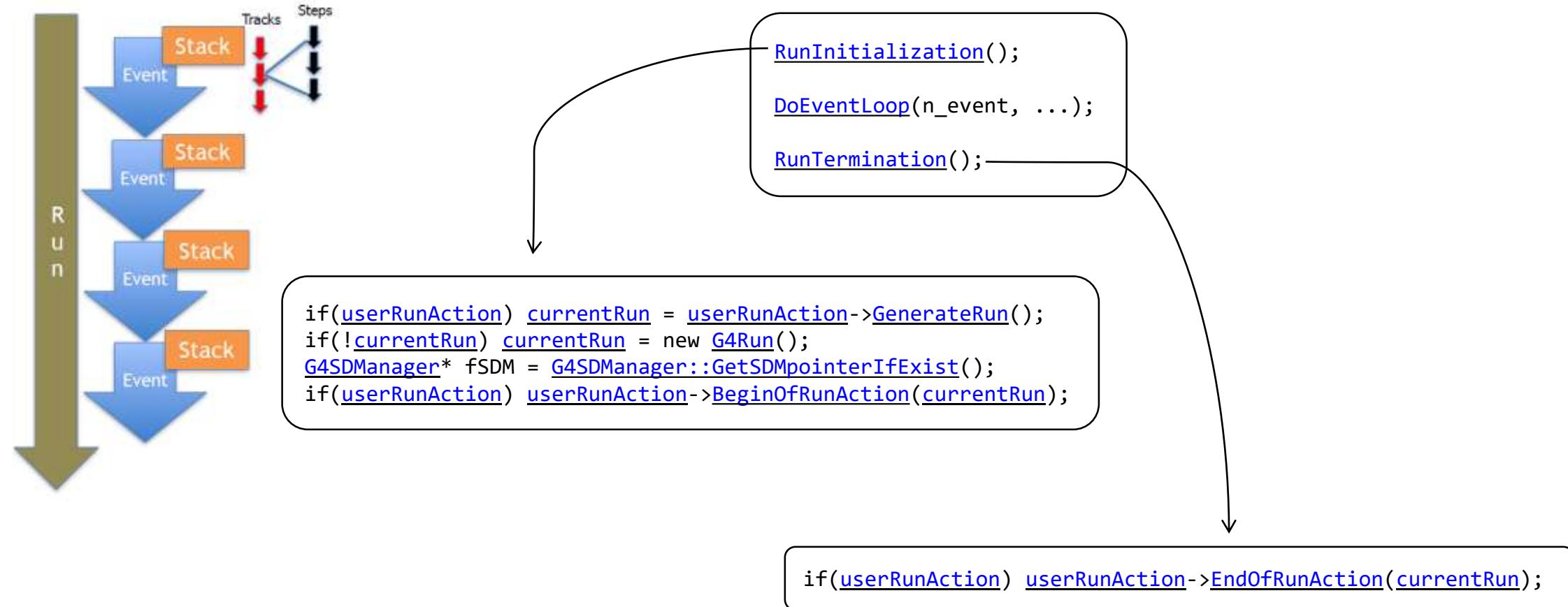
/run/beamOn 100 → G4RunManager::GetRunManager()->BeamOn(G4int n_event, ...);



Wykonanie pętli eventów – o tym za moment...

Geant4 – podstawowa struktura

/run/beamOn 100 → G4RunManager::GetRunManager()->BeamOn(G4int n_event, ...);



W fazie zakończenia

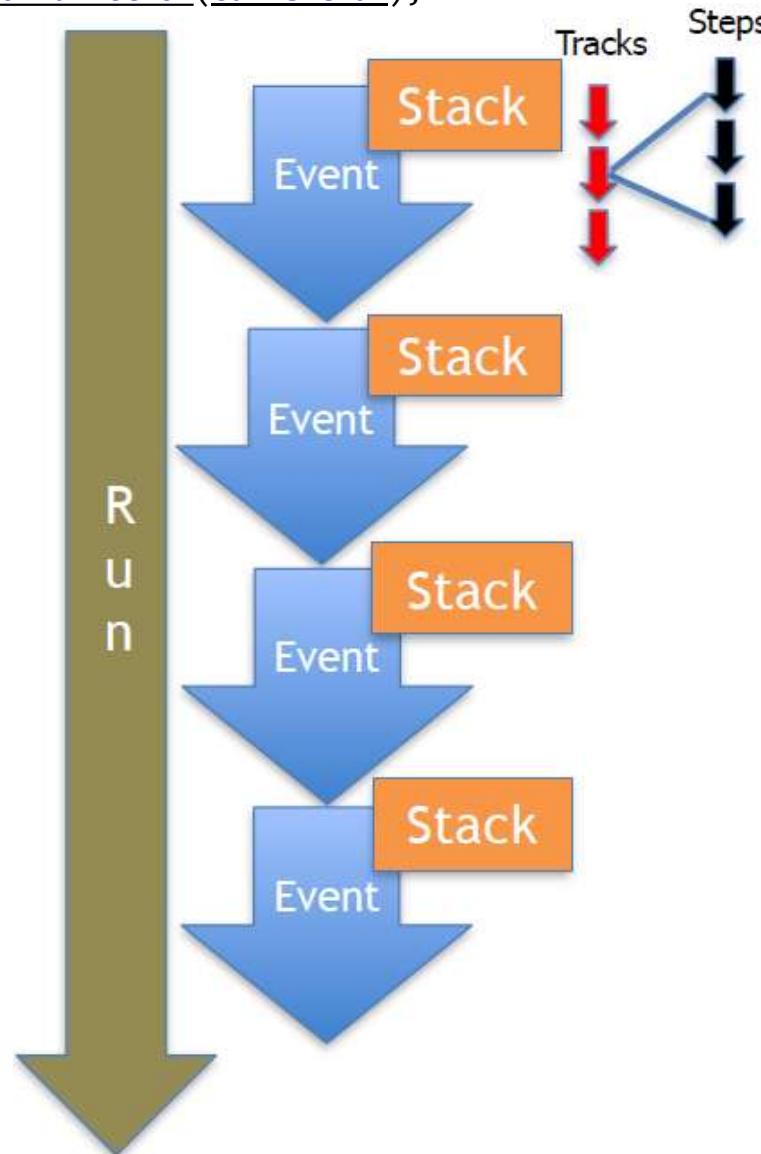
- *Jeśli istnieje, zostaje uruchomiona opcjonalna funkcja dostarczona przez użytkownika „EndOfRunAction”. Otrzyma ona wskaźnik do obiektu reprezentującego Run w momencie tuż po zakończeniu pętli eventów. Można ją wykorzystać np. do obliczania średnich z wielkości fizycznych akumulowanych po wszystkich eventach (np. średnia liczba mionów na poziomie morza wytworzonych przez proton promieniowania kosmicznego o energii X padający na atmosferę ziemską).*

Geant4 – podstawowa struktura

[userRunAction->BeginOfRunAction\(currentRun\);](#)

Tutaj istnieje już detektor, fizyka, generator cząstek i mamy do nich pełny dostęp. Możemy np.:

- tworzyć nazwy plików odzwierciedlające bieżące ustawienia detektora, wiązki, aktywnych procesów, itp.
- tworzyć odpowiednie histogramy, itd.



[userRunAction->EndOfRunAction\(currentRun\);](#)

- Prosta analiza runu, obliczanie średnich po wszystkich eventach, itp.
- Zapis i zamykanie plików

Geant4 – podstawowa struktura. G4UserRunAction

Właściwie to skąd się bierze userRunAction ?

Oraz

```
userRunAction->GenerateRun()
userRunAction->BeginOfRunAction(currentRun);
userRunAction->EndOfRunAction(currentRun);
```

Geant4 – podstawowa struktura. G4UserRunAction

Właściwie to skąd się bierze userRunAction ?

moja_mala_aplikacja_geant4.cc

```
#include "G4RunManager.hh"
#include "MyDetectorConstruction.hh"
#include "FTFP_BERT.hh"
#include "MyActionInitialization.hh"

int main(int argc,char** argv) {
    G4RunManager* runManager = new G4RunManager;

    runManager->SetUserInitialization(new MyDetectorConstruction());
    G4VModularPhysicsList* MyPhysicsList = new FTFP_BERT;
    runManager->SetUserInitialization(MyPhysicsList);
    runManager->SetUserInitialization(new MyActionInitialization());
```

- Tutaj, oprócz stworzenia generatora cząstek pierwotnych, można tworzyć tzw. akcje użytkownika różnych poziomów. Muszą to być obiekty klas pochodnych od:
 - G4UserRunAction
 - G4UserEventAction
 - G4UserStackingAction
 - G4UserTrackingAction
 - G4UserSteppingAction

Geant4 – podstawowa struktura. G4UserRunAction

Właściwie to skąd się bierze userRunAction ?

moja_mala_aplikacja_geant4.cc

```
runManager->SetUserInitialization(new MyActionInitialization());
```

- Jest to obiekt klasy pochodnej od G4VUserActionInitialization. Deklaracja tej klasy może wyglądać np. tak:

```
class MyActionInitialization : public G4VUserActionInitialization
{
public:
    MyActionInitialization();
    virtual ~MyActionInitialization();
    virtual void Build() const;
};
```

- W klasie MyActionInitialization trzeba zaimplementować funkcję Build() (odziedziczoną po klasie bazowej G4VUserActionInitialization), w której tworzymy i rejestrujemy do RunManagera „akcje użytkownika”:

```
MyActionInitialization::Build() {
    SetUserAction(new MyPrimaryGeneratorAction() );
    SetUserAction(new MyUserRunAction() );
}
```



Tutaj wywoływana jest funkcja:

```
G4RunManager::GetRunManager()->SetUserAction(G4UserRunAction *);
```

Geant4 – podstawowa struktura. G4UserRunAction

Właściwie to skąd się bierze userRunAction ?

moja_mala_aplikacja_geant4.cc

```
runManager->SetUserInitialization(new MyActionInitialization());
```

- W klasie MyActionInitialization trzeba zaimplementować funkcję Build() (odziedziczoną po klasie bazowej G4VUserActionInitialization), w której tworzymy i rejestrujemy do RunManagera „akcje użytkownika”:

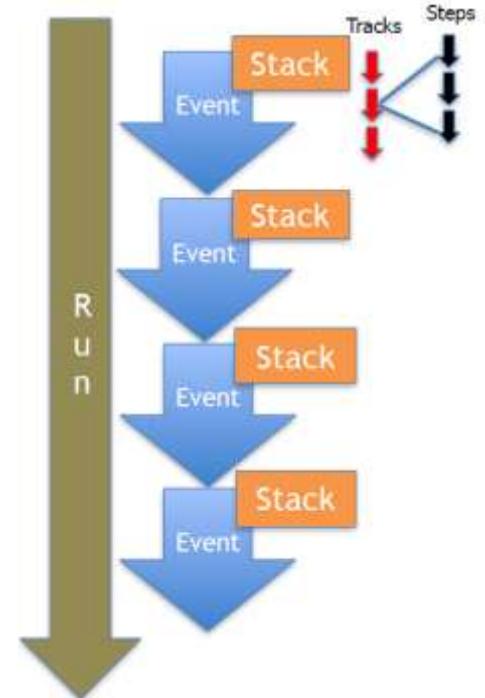
```
MyActionInitialization::Build() {
    SetUserAction(new MyPrimaryGeneratorAction() );
    SetUserAction(new MyUserRunAction() );
}
```

Deklaracja klasy MyUserRunAction może wyglądać np. tak:

```
class MyUserRunAction : public G4UserRunAction
{
public:
    MyUserRunAction();
    virtual ~MyUserRunAction();
    virtual G4Run* GenerateRun();
    virtual void BeginOfRunAction(const G4Run* );
    virtual void EndOfRunAction(const G4Run* );
};
```

Geant4 – podstawowa struktura

/run/beamOn 100 → G4RunManager::GetRunManager()->BeamOn(G4int n_event, ...);



RunInitialization();

DoEventLoop(n_event, ...);

RunTermination();

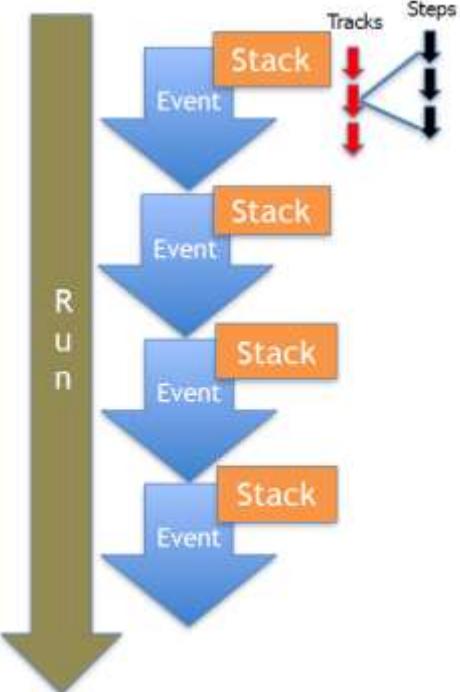
**Wykonanie pętli eventów
– o tym teraz!**

Geant4 – podstawowa struktura. Pętla eventów.

G4RunManager::GetRunManager()()->BeamOn(G4int n_event, ...);

DoEventLoop(n_event, ...);

```
for(G4int i_event=0; i_event<n_event; i_event++ ) {  
    ProcessOneEvent(i_event);  
    ...  
}
```



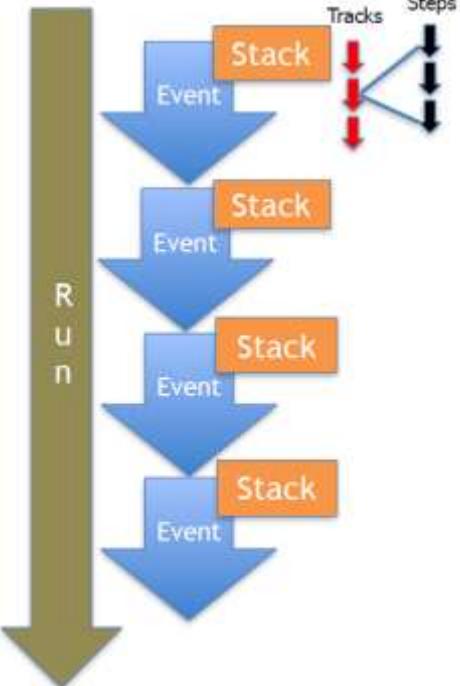
Geant4 – podstawowa struktura. Pętla eventów.

```
G4RunManager::GetRunManager()->BeamOn(G4int n_event, ...);
```

```
DoEventLoop(n_event, ...);
```

```
for(G4int i_event=0; i_event<n_event; i_event++ ) {  
    ProcessOneEvent(i_event);  
    ...  
}
```

```
{  
    currentEvent = GenerateEvent(i_event);  
    eventManager->ProcessOneEvent(currentEvent);  
    AnalyzeEvent(currentEvent);  
    UpdateScoring();  
}
```



Geant4 – czym jest Event?

- **Event**/"zdarzenie" jest podstawową jednostką symulacji w Geant4. Reprezentuje zbiór „śladów” (częstek).
 - Na początku eventu generowane są cząstki pierwotne (reprezentowane przez obiekty typu [G4PrimaryVertex](#)) i umieszczane na stosie.
 - Cząstka z góry stosu jest zdejmowana i „trackowana” („śledzona”) – rozgrywana jest „historia jej życia” zgodnie z rozkładami prawdopodobieństw poszczególnych procesów fizycznych.
 - Jakiekolwiek cząstki wtórne odkładane są na stosie.
 - „Trackowanie”/„śledzenie” trwa dopóty na stosie są cząstki.
 - Przetwarzanie eventu kończy się kiedy stos zostaje opróżniony.
- Zdarzenie jest reprezentowane przez obiekt klasy [G4Event](#). Po ukończeniu przetwarzania eventu, obiekt ten zawiera:
 - Listę pierwotnych położeń („primary vertex”) i cząstek („primary particle”) (oraz ich początkowe kierunki, energie, itd.)
 - (opcjonalnie) Obiekty reprezentujące dane wyjściowe: zbiory (kolekcje) wytworzonych „hitów” i „trajektorii” ([G4VHitsCollection](#), [G4TrajectoryContainer](#))
- Obiekt klasy [G4EventManager](#) koordynuje przetwarzanie eventu.
- Użytkownik może zarejestrować własne funkcje, które będą („automagicznie”) uruchamiane na początku i na końcu przetwarzania eventu i będą mieć dostęp do aktualnego obiektu G4Event.
([G4UserEventAction](#))

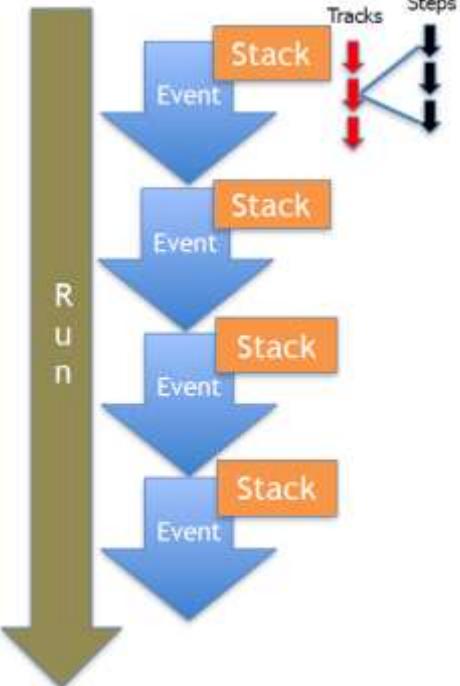
Zapożyczone od M. Asai za pośrednictwem J.Apostolakis

Geant4 – podstawowa struktura. Pętla eventów.

```
G4RunManager::GetRunManager()->BeamOn(G4int n_event, ...);
```

```
DoEventLoop(n_event, ...);
```

```
for(G4int i_event=0; i_event<n_event; i_event++ ) {  
    ProcessOneEvent(i_event);  
    ...  
}
```



```
{  
    currentEvent = GenerateEvent(i_event);  
    eventManager->ProcessOneEvent(currentEvent);  
    AnalyzeEvent(currentEvent);  
    UpdateScoring();  
}
```

```
G4Event* anEvent = new G4Event(i_event);  
userPrimaryGeneratorAction->GeneratePrimaries(anEvent);  
return anEvent;
```

- Tworzony jest nowy, „pusty” obiekt typu G4Event reprezentujący event.
- Wskaźnik do „pustego” eventu przekazywany jest do funkcji użytkownika odpowiedzialnej za kreację cząstek pierwotnych i dodanie ich do pustego eventu (o tym w innym wykładzie).
- Zwracany jest wskaźnik (ściślej wartość wskaźnika) do nowo utworzonego eventu (return anEvent).

Uwagi na marginesie. Cykl życia obiektu.

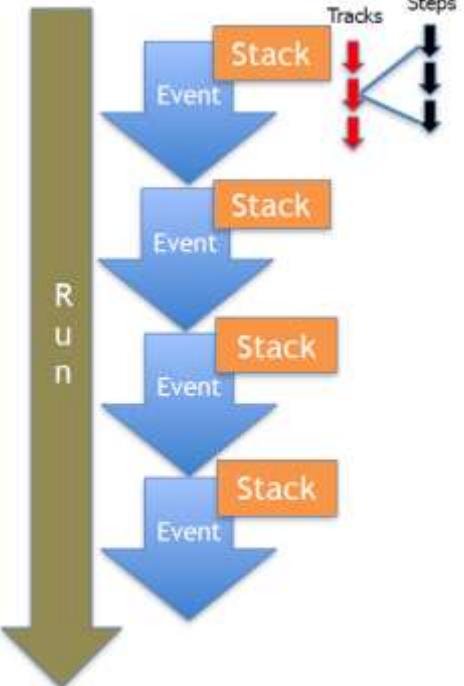
- Obiekt stworzony operatorem new nie jest obiektem lokalnym funkcji, w której został stworzony (wskaźnik anEvent jest zmienną/obiektem lokalną). To oznacza, że będzie istniał nawet po wyjściu z funkcji (wskaźnik anEvent zostanie automatycznie usunięty).
- Obiekt stworzony operatorem new istnieje w pamięci dopóki nie zostanie wykasowany operatorem delete. Może to mieć miejsce w zupełnie innej funkcji, a nawet klasie, niż ta, w której obiekt został stworzony.
- Do posługiwania się takim obiektem wystarczy znajomość jego adresu w pamięci (to daje wskaźnik).
- Zniszczenie wskaźnika nie niszczy samego obiektu! Ażkolwiek możemy utracić do niego dostęp.

Geant4 – podstawowa struktura. Pętla eventów.

```
G4RunManager::GetRunManager()->BeamOn(G4int n_event, ...);
```

```
DoEventLoop(n_event, ...);
```

```
for(G4int i_event=0; i_event<n_event; i_event++ ) {  
    ProcessOneEvent(i_event);  
    ...  
}
```



```
{  
    currentEvent = GenerateEvent(i_event);  
    eventManager->ProcessOneEvent(currentEvent);  
    AnalyzeEvent(currentEvent);  
    UpdateScoring();  
}
```

```
G4Event* anEvent = new G4Event(i_event);  
userPrimaryGeneratorAction->GeneratePrimaries(anEvent);  
return anEvent;
```

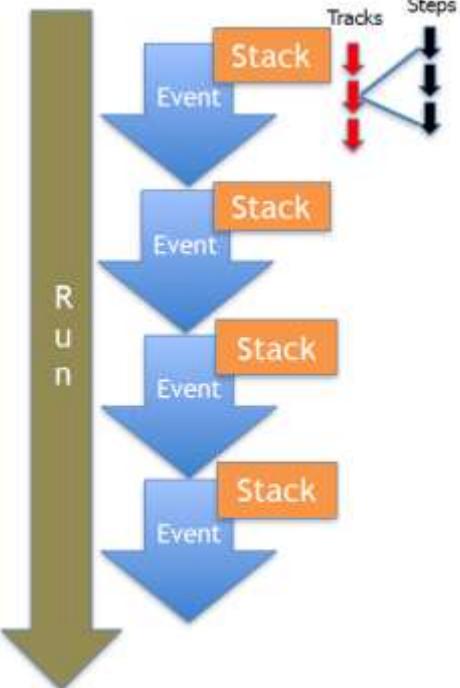
- Nowo utworzony event (wskaźnik do niego) przekazujemy menadżerowi eventów. Ścisłej, jednej z funkcji menadżera eventów odpowiedzialnej za wykonanie pojedynczego eventu.

Geant4 – podstawowa struktura. Pętla eventów.

```
G4RunManager::GetRunManager()->BeamOn(G4int n_event, ...);
```

```
DoEventLoop(n_event, ...);
```

```
for(G4int i_event=0; i_event<n_event; i_event++ ) {  
    ProcessOneEvent(i_event);  
    ...  
}
```



```
{  
    currentEvent = GenerateEvent(i_event);  
    eventManager->ProcessOneEvent(currentEvent);  
    AnalyzeEvent(currentEvent);  
    UpdateScoring();  
}
```

```
G4Event* anEvent = new G4Event(i_event);  
userPrimaryGeneratorAction->GeneratePrimaries(anEvent);  
return anEvent;
```

```
if(userEventAction) userEventAction->BeginOfEventAction(currentEvent);  
/* Przepisanie cząstek pierwotnych na stos */  
while( ( track = trackContainer->PopNextTrack() ) != 0 ) {  
    trackManager->ProcessOneTrack( track );  
    G4TrackVector * secondaries = trackManager->GimmeSecondaries();  
    /* Sortowanie cząstek wtórnych na odpowiednie stosy */  
}  
if(userEventAction) userEventAction->EndOfEventAction(currentEvent);
```

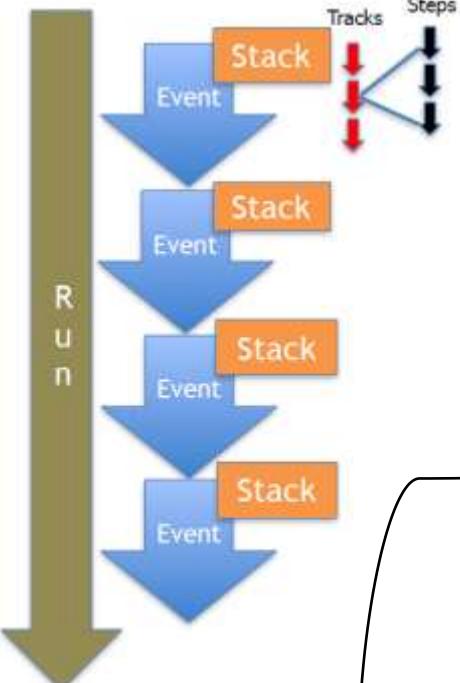
- Jeśli istnieją, wykonywane są akcje użytkownika na początku i końcu przetwarzania eventu.
- Cząstki pierwotne umieszczane są na stosie.
- Z wierzchu stosu zdejmowana jest cząstka i przetwarzana przez menadżera track'ów (o tym za chwilę).
- Powstałe cząstki wtórne są dokładane na stos.
- Przetwarzanie trwa dopóki stos nie zostanie opróżniony (pętla while).

Geant4 – podstawowa struktura. Pętla eventów.

```
G4RunManager::GetRunManager()->BeamOn(G4int n_event, ...);
```

```
DoEventLoop(n_event, ...);
```

```
for(G4int i_event=0; i_event<n_event; i_event++ ) {  
    ProcessOneEvent(i_event);  
    ...  
}
```



```
{  
    currentEvent = GenerateEvent(i_event);  
    eventManager->ProcessOneEvent(currentEvent);  
    AnalyzeEvent(currentEvent);  
    UpdateScoring();  
}
```

```
G4Event* anEvent = new G4Event(i_event);  
userPrimaryGeneratorAction->GeneratePrimaries(anEvent);  
return anEvent;
```

```
currentRun->RecordEvent(anEvent);
```

```
if(userEventAction) userEventAction->BeginOfEventAction(currentEvent);  
/* Przepisanie cząstek pierwotnych na stos */  
while( ( track = trackContainer->PopNextTrack() ) != 0 ) {  
    trackManager->ProcessOneTrack( track );  
    G4TrackVector * secondaries = trackManager->GimmeSecondaries();  
    /* Sortowanie cząstek wtórnych na odpowiednie stosy */  
}  
if(userEventAction) userEventAction->EndOfEventAction(currentEvent);
```

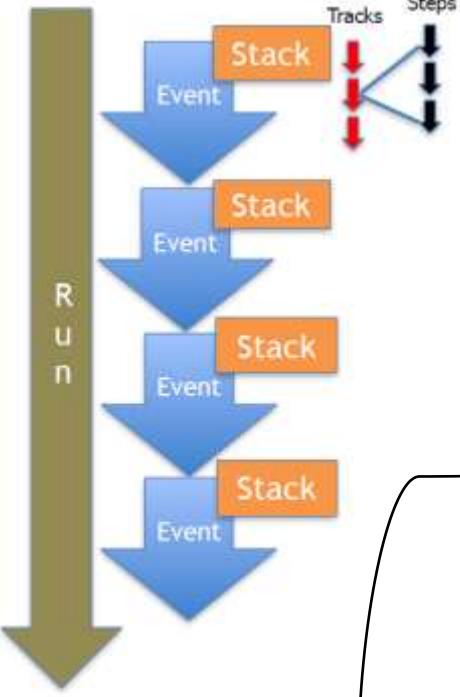
- Np. zapis trajektorii cząstek na potrzeby późniejszej wizualizacji
- Użytkownik może rozbudować funkcję RecordEvent o własne działania.

Geant4 – podstawowa struktura. Pętla eventów.

```
G4RunManager::GetRunManager()->BeamOn(G4int n_event, ...);
```

```
DoEventLoop(n_event, ...);
```

```
for(G4int i_event=0; i_event<n_event; i_event++ ) {  
    ProcessOneEvent(i_event);  
    ...  
}
```



```
{  
    currentEvent = GenerateEvent(i_event);  
    eventManager->ProcessOneEvent(currentEvent);  
    AnalyzeEvent(currentEvent);  
    UpdateScoring();  
}
```

```
G4Event* anEvent = new G4Event(i_event);  
userPrimaryGeneratorAction->GeneratePrimaries(anEvent);  
return anEvent;
```

```
if(userEventAction) userEventAction->BeginOfEventAction(currentEvent);  
/* Przepisanie cząstek pierwotnych na stos */  
while( ( track = trackContainer->PopNextTrack() ) != 0 ) {  
    trackManager->ProcessOneTrack( track );  
    G4TrackVector * secondaries = trackManager->GimmeSecondaries();  
    /* Sortowanie cząstek wtórnych na odpowiednie stosy */  
}  
if(userEventAction) userEventAction->EndOfEventAction(currentEvent);
```

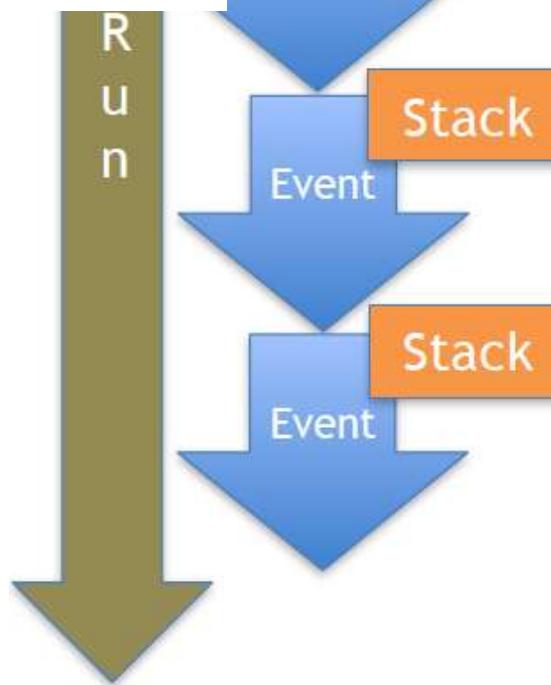
Aktualizacja wartości akumulowanych w detektorach typu Primitive Scorer (jeśli zostały utworzone przed rozpoczęciem run'u)

Geant4 – podstawowa struktura. Pętla eventów.

- Tworzenie własnych struktur danych dla eventu

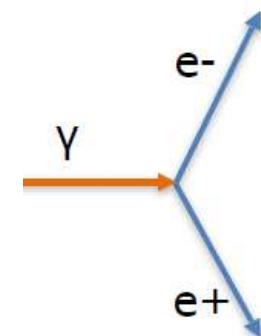


- Analiza eventu (mamy dostęp do cząstek pierwotnych i danych zebranych w tym evencie z detektorów)



Geant4 – czym jest Track?

- Track jest jakby „zdjęciem” cząstki – reprezentuje jej chwilowy stan. ($\vec{x}, \vec{p}, \text{PDG}, \overset{\rightarrow}{\sigma}, q, \dots$)
 - Wielkości fizyczne przechowywane w obiekcie typu G4Track odzwierciedlają chwilowy stan cząstki w trakcie symulacji.
Nie przechowuje przeszłych wartości.
 - Step (krok) stanowi informację o zmianie track'u (“delta”).
Track nie jest zbiorem Stepów.
Track jest aktualizowany w serii kolejnych kroków.
- Każdy wytworzony w symulacji obiekt Track jest chwilowy i znika (jest usuwany z pamięci) kiedy reprezentowana przez niego cząstka:
 - dotrze do granic „świata”, (world volume),
 - zniknie w wyniku reakcji, np. $\gamma \rightarrow e^- + e^+$
 - osiągnie zerową energię kinetyczną (i nie ma procesów typu „AtRest”),
 - zostanie zlikwidowana przez użytkownika (tzn. w części kodu użytkownika).
- Wszystkie obiekty Track w końcu znikają. Na końcu Eventu nie ma żadnego.
 - Do zapamiętywania chwilowych danych z obiektu G4Track użytkownik musi użyć obiektu klasy G4Trajectory.
- **G4TrackingManager** zarządza przetwarzaniem śladu cząstki (reprezentowanego przez obiekt klasy **G4Track**).
- Klasa **G4UserTrackingAction** umożliwia dostęp do obiektu G4Track tuż po tym jak został utworzony i tuż przed jego finalnym kasowaniem.



Geant4 – czym jest Track? (Czym jest cząstka w Geant4?)

Cząstka w Geant4 jest reprezentowana przez obiekty ułożone w trójwarstwową hierarchię:

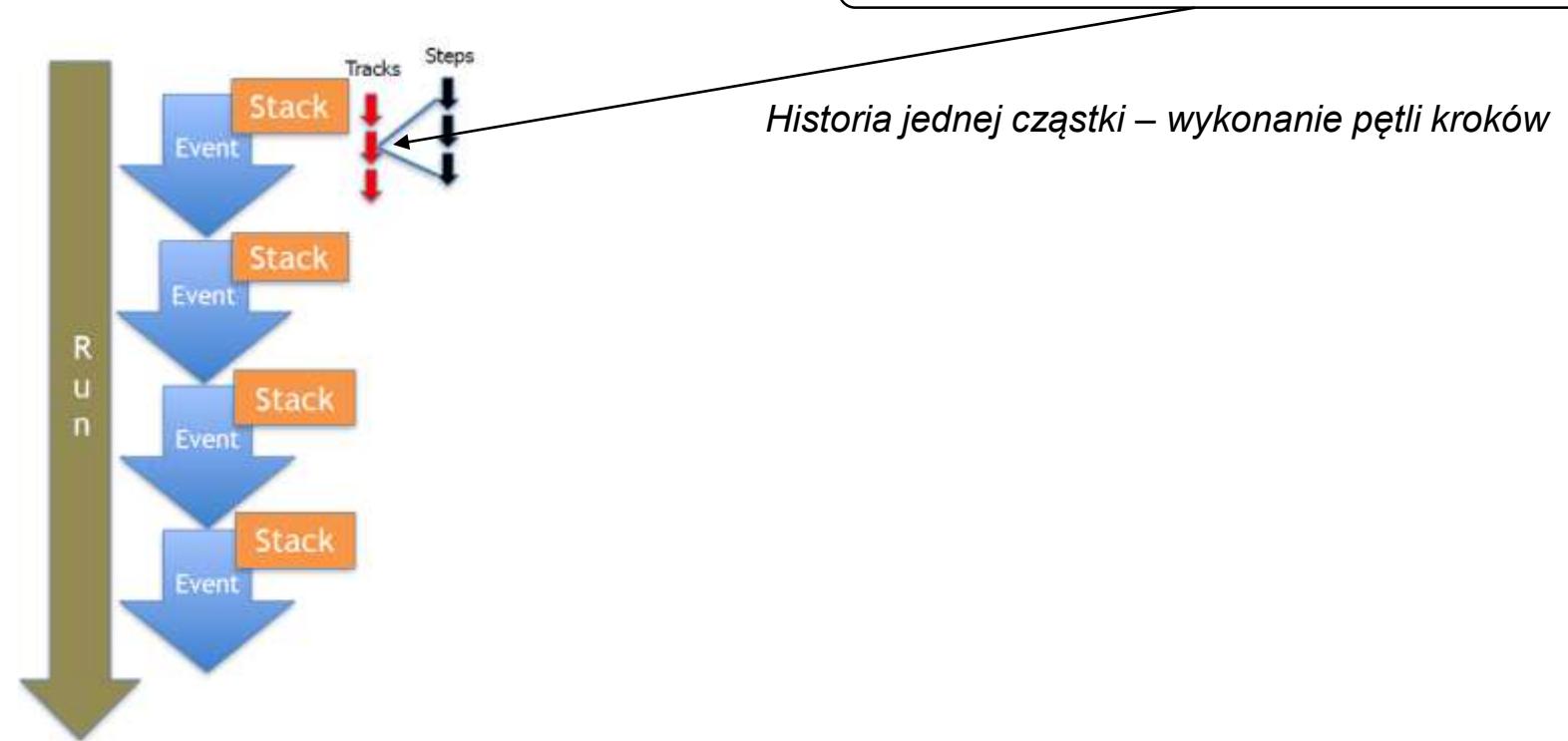
- **G4Track**
 - bieżąca pozycja i inne bieżące informacje geometryczne (np. bryła w której cząstka się aktualnie znajduje)
 - ten obiekt podlega „trackowaniu” – transportowaniu, krok po kroku, przez geometrię.
- **G4DynamicParticle**
 - „dynamiczne” (zmienne) wielkości fizyczne cząstki: pęd, energia, polaryzacja,..
 - każdy obiekt **G4Track** posiada w swoim wnętrzu unikalny obiekt **G4DynamicParticle** reprezentujący dynamiczne wielkości tej konkretnej, indywidualnej cząstki.
 - jest niszczony wraz z macierzystym obiektem typu G4Track.
- **G4ParticleDefinition**
 - „statyczne” (niezmienne) wielkości opisujące cząstkę: ładunek, masa, czas życia, kanały rozpadu, itd.
 - Wszystkie obiekty typu **G4DynamicParticle** reprezentujące własności cząstek tego samego rodzaju współdzielą jeden obiekt typu **G4ParticleDefinition**. Np. wszystkie G4DynamicParticle reprezentujące zmienne własności elektronów współdzielą jeden obiekt klasy definiującej statyczne cechy elektronu: **G4Electron** (która jest klasą pochodną od G4ParticleDefinition)
 - **G4ProcessManager** przechowuje spis wszystkich procesów fizycznych właściwych dla cząstek danego rodzaju

Zapożyczone od M. Asai za pośrednictwem J.Apostolakis

Geant4 – podstawowa struktura

`eventManager->ProcessOneEvent(currentEvent);`

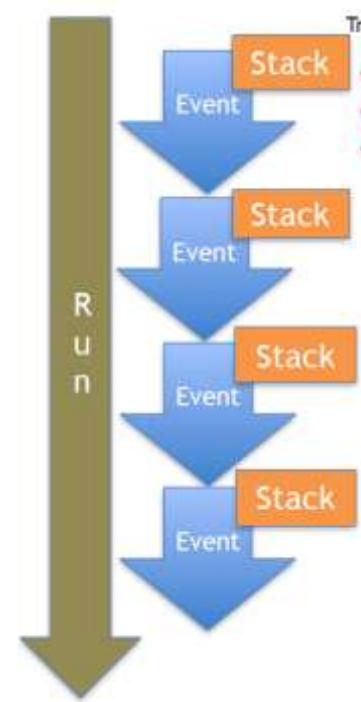
`trackManager->ProcessOneTrack(track);`



Geant4 – podstawowa struktura

```
eventManager->ProcessOneEvent(currentEvent);
```

```
trackManager->ProcessOneTrack( track );
```



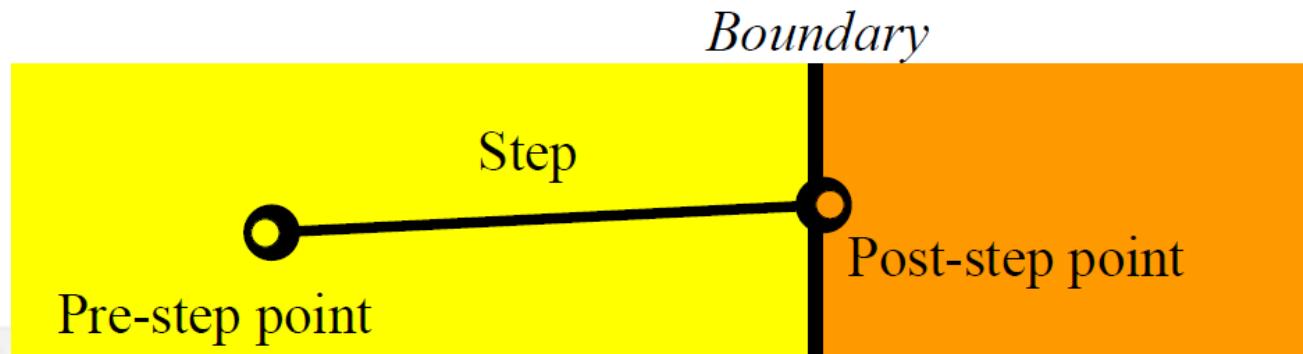
SteppingManager inicjalizuje obiekt G4Step, używany w pętli kroków

- *PreStepPoint i PostStepPoint wypełniane są danymi z obiektu track otrzymanego z TrackManagera (początkowe położenie, itp.)*

```
fpSteppingManager->SetInitialStep(fpTrack);
if( fpUserTrackingAction != 0 ) {fpUserTrackingAction->PreUserTrackingAction(fpTrack);}
// Give SteppingManger the maximum number of processes
fpSteppingManager->GetProcessNumber();
// Give track the pointer to the Step
fpTrack->SetStep(fpSteppingManager->GetStep());
// Inform beginning of tracking to physics processes
fpTrack->GetDefinition()->GetProcessManager()->StartTracking(fpTrack);
// Track the particle Step-by-Step while it is alive
while( (fpTrack->GetTrackStatus() == fAlive) || (fpTrack->GetTrackStatus() == fStopButAlive)){
    fpTrack->IncrementCurrentStepNumber();
    fpSteppingManager->Stepping();
}
// Inform end of tracking to physics processes
fpTrack->GetDefinition()->GetProcessManager()->EndTracking();
if( fpUserTrackingAction != 0 ) {fpUserTrackingAction->PostUserTrackingAction(fpTrack);}
```

Geant4 – czym jest Step?

- 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.).
- Punkt początkowy i końcowy kroku zawierają informację o obiekcie geometrycznym, w którym się znajdują.
- Na długości kroku wpływają procesy fizyczne, którym podlega cząstka. Krok może też być ograniczony „sztucznym” limitem (w gestii użytkownika) lub granicą obiektów geometrycznych.
- Jeśli długość kroku została ograniczona granicą obiektów geometrycznych, to punkt końcowy fizycznie jest położony na granicy ale logicznie należy już do następnej bryły.
 - Taki krok posiada информацию o materiałach obu – umożliwia to symulację procesów zachodzących na granicy ośrodków (odbicie, refrakcja (załamanie), emisja promieniowania przejścia)
- Krok jest reprezentowany przez obiekt klasy [**G4Step**](#).
- Punkt początkowy i końcowy są reprezentowane przez obiekty klasy [**G4StepPoint**](#)
- Obiekt klasy [**G4SteppingManager**](#) steruje wykonaniem kroku (i aktualizacją stanu cząstki).

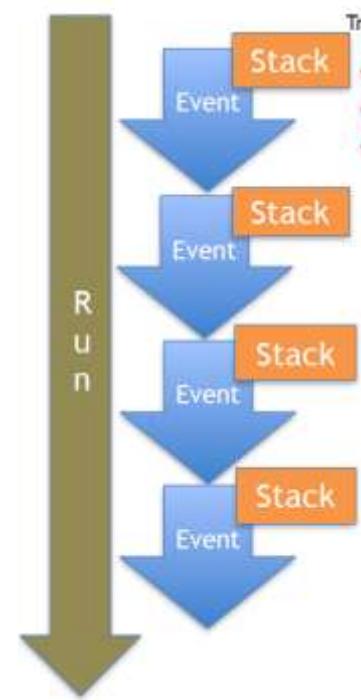


Zapożyczone od M. Asai za pośrednictwem J.Apostolakis

Geant4 – podstawowa struktura

```
eventManager->ProcessOneEvent(currentEvent);
```

```
trackManager->ProcessOneTrack( track );
```

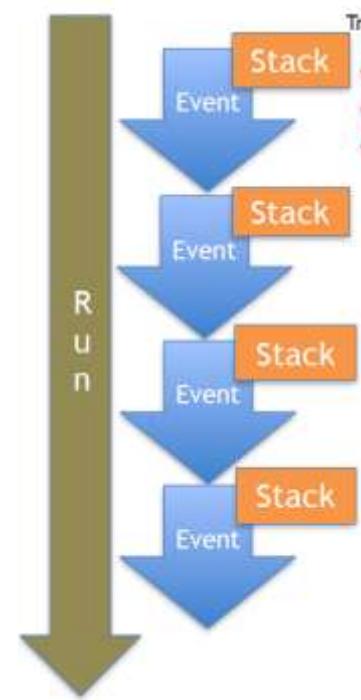


```
fpSteppingManager->SetInitialStep(fpTrack);
if( fpUserTrackingAction != 0 ) {fpUserTrackingAction->PreUserTrackingAction(fpTrack);}
// Give SteppingManger the maximum number of processes
fpSteppingManager-> Uruchamianie funkcji użytkownika (o ile została zarejestrowana do RunManagera).
// Give track the p fpTrack->SetStep(fp;
// Inform beginning   • Mamy dostęp do danych cząstki na samym początku historii jej „życia”.
// Track the particle Step-by-Step while it is alive   • Tutaj Track posiada już wskaźnik do pierwszego, jeszcze niewykonanego, kroku.
// Inform end of tracking to physics processes   • Możemy np. zdecydować o wykasowaniu tej cząstki.
fpTrack->GetDefinition()->GetProcessManager()->EndTracking(fpTrack);
if( fpUserTrackingAction != 0 ) {fpUserTrackingAction->PostUserTrackingAction(fpTrack);}
```

Geant4 – podstawowa struktura

```
eventManager->ProcessOneEvent(currentEvent);
```

```
trackManager->ProcessOneTrack( track );
```

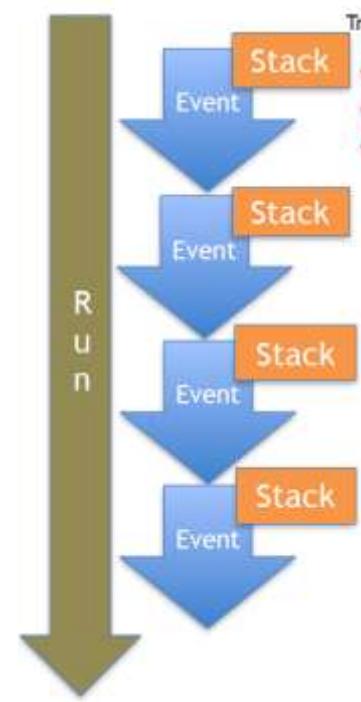


```
fpSteppingManager->SetInitialStep(fpTrack);
if( fpUserTrackingAction != 0 ) {fpUserTrackingAction->PreUserTrackingAction(fpTrack);}
// Give SteppingManger the maximum number of processes
fpSteppingManager->GetProcessNumber();
// Give track the pointer to the Step
fpTrack->SetStep(fpSteppingManager->GetStep());
// Inform beginning of tracking to physics processes
fpTrack->GetDefinition()->GetProcessManager()->StartTracking(fpTrack);
// Trac
while :Alive)){
    Kolejne inicjalizacje niezbędne do wykonania pętli kroków:
    f
    f
}
// In
fpTra
if( fpUserTrackingAction != 0 ) {fpUserTrackingAction->PostUserTrackingAction(fpTrack);}
```

Geant4 – podstawowa struktura

[eventManager->ProcessOneEvent\(currentEvent\);](#)

[trackManager->ProcessOneTrack\(track \);](#)



```
fpSteppingManager->SetInitialStep(fpTrack);
if( fpUserTrackingAction != 0 ) {fpUserTrackingAction->PreUserTrackingAction(fpTrack);}
// Give SteppingManger the maximum number of processes
fpSteppingManager->GetProcessNumber();
// Give track the pointer to the Step
fpTrack->SetStep(fpSteppingManager->GetStep());
// Inform beginning of tracking to physics processes
fpTrack->GetDefinition()->GetProcessManager()->StartTracking(fpTrack);
// Track the particle Step-by-Step while it is alive
while( (fpTrack->GetTrackStatus\(\) == fAlive) || (fpTrack->GetTrackStatus\(\) == fStopButAlive)){
    fpTrack->IncrementCurrentStepNumber\(\);
    fpSteppingManager->Stepping\(\);
}
// Inform end of tracking to physics processes
fpTrack->GetDefinition()->GetProcessManager()->EndTracking(fpTrack);
if( fpUserTrackingAction != 0 ) {fpUserTrackingAction->PostUserTrackingAction(fpTrack);}
```

Wykonanie pętli kroków.

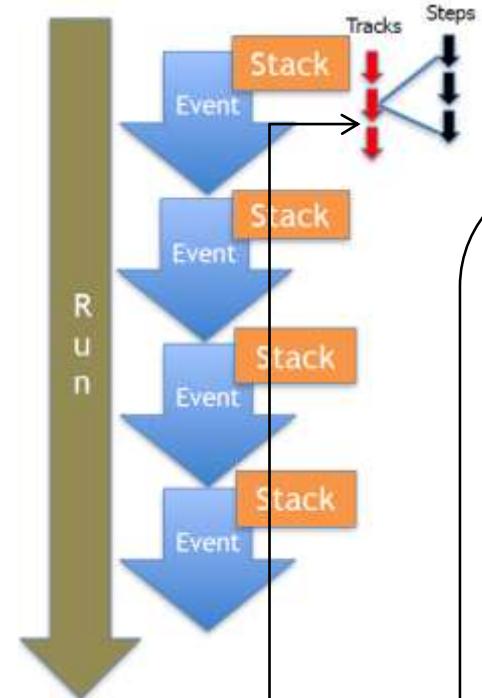
Pętla trwa dopóki cząstka jest „żywa”, tzn.:

- nie opuściła świata oraz
- nie została usunięta oraz
- ma niezerową energię kinetyczną, lub
- jest w spoczynku ale są procesy fizyczne, które jeszcze mogą zajść (np. rozpad)

Geant4 – podstawowa struktura

[eventManager->ProcessOneEvent\(currentEvent\);](#)

[trackManager->ProcessOneTrack\(track \);](#)



```
fpSteppingManager->SetInitialStep(fpTrack);
if( fpUserTrackingAction != 0 ) {fpUserTrackingAction->PreUserTrackingAction(fpTrack);}
// Give SteppingManger the maximum number of processes
fpSteppingManager->GetProcessNumber();
// Give track the pointer to the Step
fpTrack->SetStep(fpSteppingManager->GetStep());
// Inform beginning of tracking to physics processes
fpTrack->GetDefinition()->GetProcessManager()->StartTracking(fpTrack);
// Track the particle Step-by-Step while it is alive
while( (fpTrack->GetTrackStatus() == fAlive) || (fpTrack->GetTrackStatus() == fStopButAlive)){
    fpTrack->IncrementCurrentStepNumber();
    fpSteppingManager->Stepping();
}
// Inform end of tracking to physics processes
fpTrack->GetDefinition()->GetProcessManager()->EndTracking();
if( fpUserTrackingAction != 0 ) {fpUserTrackingAction->PostUserTrackingAction(fpTrack);}
```

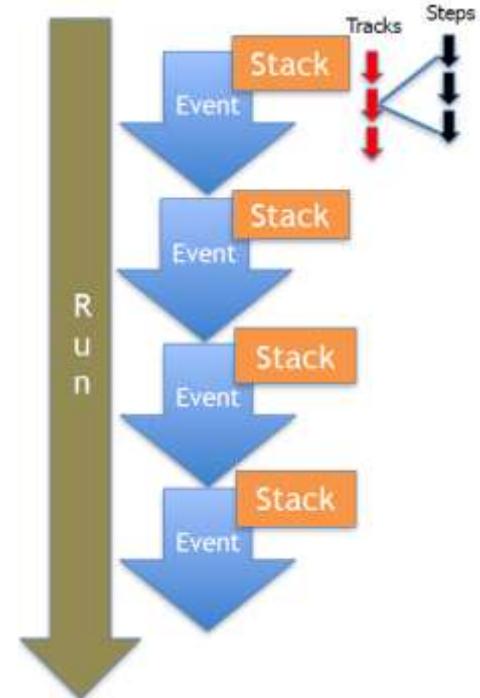
Uruchamianie funkcji użytkownika (o ile została zarejestrowana do RunManagera).

- Mamy dostęp do danych cząstki na samym końcu historii jej „życia”.
- Mamy też dostęp do wszystkich cząstek wtórnych wytworzonych po drodze.

Geant4 – podstawowa struktura. Wykonanie pojedynczego kroku.

```
trackManager->ProcessOneTrack( track );
```

```
SteppingManager->Stepping();
```



Inicjalizacja (lokalizacja cząstki w hierarchii geometrii, resetowanie proponowanej odległości oddziaływania dla każdego z aktywnych procesów)

```
fpTrack->GetDefinition()->GetProcessManager()->StartTracking(fpTrack);
```



```
{ ((*theProcessList)[idx])->StartTracking(aTrack); }
```



```
currentInteractionLength = -1.0;  
theNumberOfInteractionLengthLeft = -1.0;  
theInitialNumberOfInteractionLength = -1.0;
```

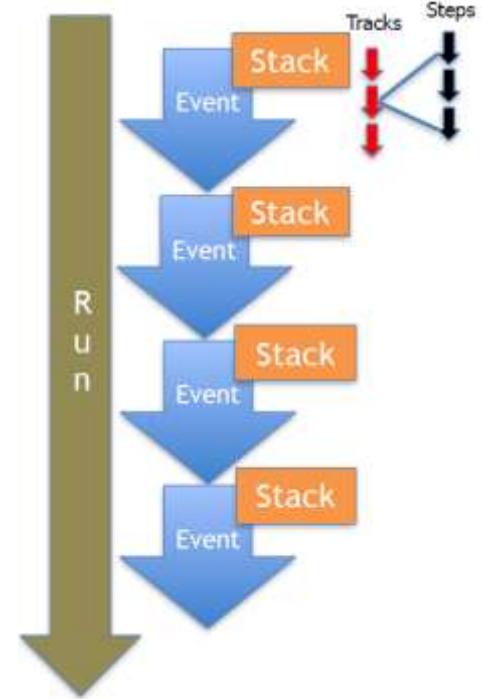
Geant4 – podstawowa struktura

trackManager->[ProcessOneTrack](#)(track);

SteppingManager->[Stepping\(\)](#);

Wyznaczenie długości kroku:

- Wszystkie procesy konkurują – każdy proponuje długość kroku (jako krotność swojej średniej drogi swobodnej).



Średnia droga swobodna, całkowity przekrój czynny

λ - średnia droga swobodna - średnia odległość jaką cząstki (danego rodzaju i o danej energii) pokonują w ośrodku pomiędzy kolejnymi punktami oddziaływań.

Średnia droga swobodna jest odwrotnością całkowitego makroskopowego przekroju czynnego

$$\lambda = \frac{1}{\Sigma}$$

Makroskopowy przekrój czynny

$$\Sigma = \sigma \cdot n$$

$\left[\frac{1}{m} \right]$

Mikroskopowy przekrój czynny [m^2] Gęstość centrów rozpraszania [m^{-3}]

$$n = \frac{N_A \rho}{A}$$

N_A – liczba Avogadro
 ρ – gęstość
 A – masa molowa

Sens fizyczny makroskopowego przekroju czynnego

$$\Sigma = \frac{P(dl)}{dl} - \text{prawdopodobieństwo reakcji na jednostkę drogi}$$

$$P(dl) = \frac{\text{Liczba cząstek z reakcją na odcinku } dl}{\text{Liczba cząstek padających}}$$

Próbkowanie długości kroku

Aby obliczyć prawdopodobieństwo $P_N(l)$, że cząstka przebędzie bez oddziaływania odległość l , wyznaczmy prawdopodobieństwo $P_N(l+dl)$ przejścia bez oddziaływania odległości $l+dl$. Będzie to iloczyn dwóch prawdopodobieństw:

$$P_N(l + dl) = P_N(l)P_N(dl)$$

$$P_N(dl) = 1 - P(dl) = 1 - \Sigma dl$$

Stąd mamy:

$$P_N(l + dl) = P_N(l)(1 - \Sigma dl)$$

$$P_N(l + dl) - P_N(l) = -P_N(l)\Sigma dl$$

$$\frac{P_N(l+dl)-P_N(l)}{dl} = -P_N(l)\Sigma$$

Co daje:

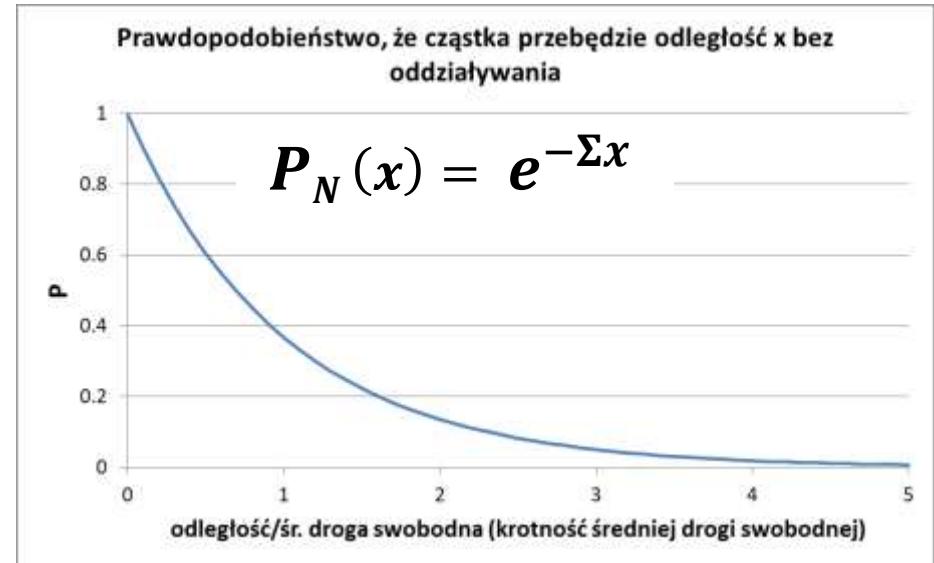
$$\frac{dP_N(l)}{dl} = -\Sigma P_N(l)$$

Rozwiążanie tego równania różniczkowego jest łatwe do znalezienia

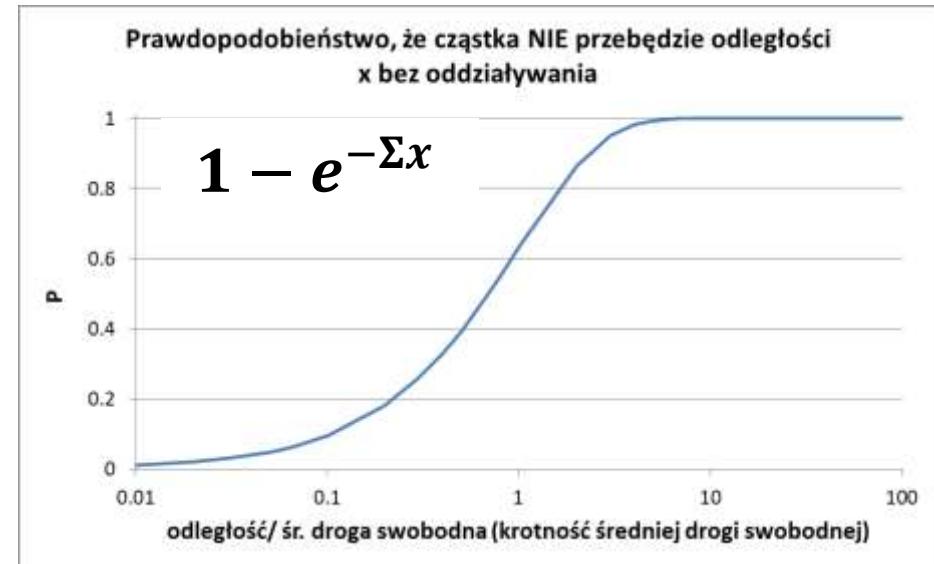
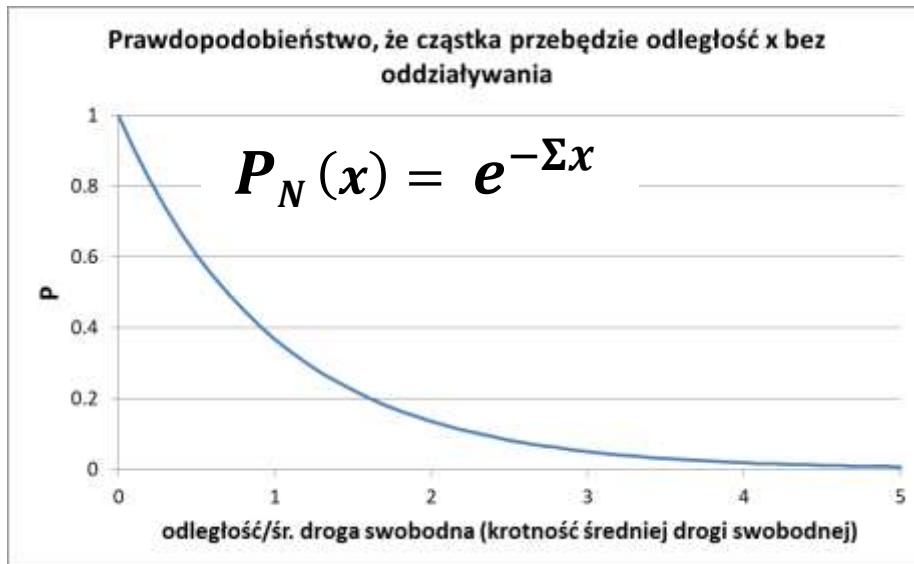
$$P_N(l) = e^{-\Sigma l} + Const \quad (\text{Stałą wyznaczamy z warunku } P_N(0) = 1, \text{ co daje } Const = 0)$$

Ostatecznie otrzymujemy znany wzór

$$P_N(l) = e^{-\Sigma l}$$



Próbkowanie długości kroku



η – (gr. „eta”) zmienna losowa z przedziału $(0, 1)$

$$\eta = 1 - e^{-\Sigma x}$$

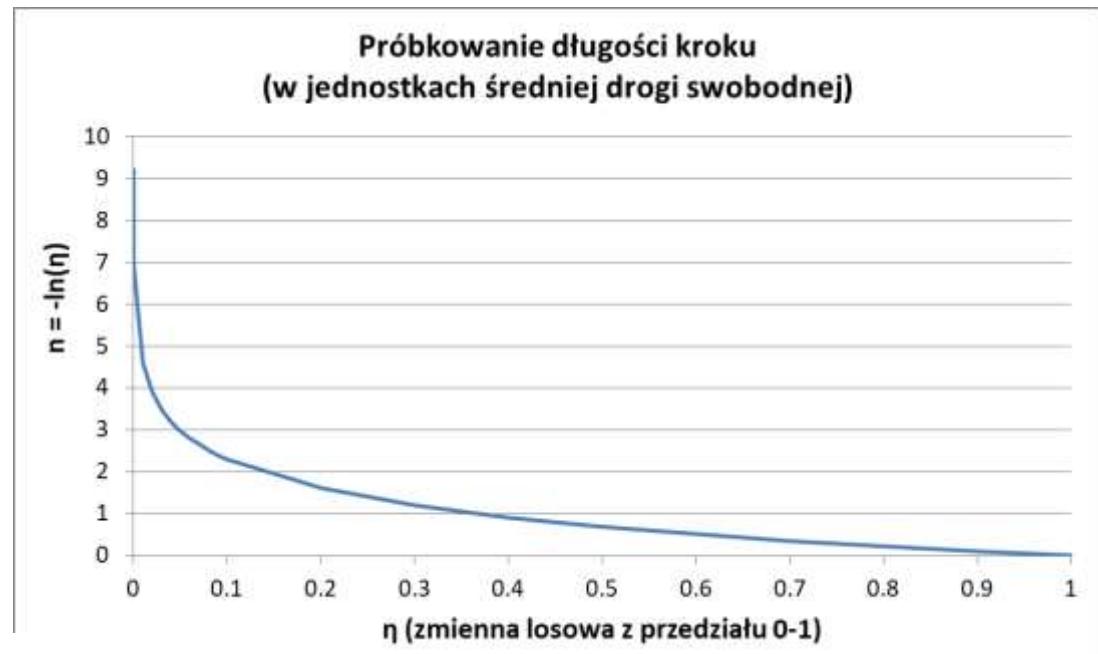
$$e^{-\Sigma x} = 1 - \eta$$

$$-\Sigma x = \ln(1 - \eta)$$

$$\frac{x}{\lambda} = -\ln(1 - \eta)$$

Rozkład zmiennej $(1 - \eta)$ jest taki sam jak zmiennej η więc możemy napisać:

$$n_\lambda = \frac{x}{\lambda} = -\ln(\eta)$$



Wyrażenie długości kroku w jednostkach średniej drogi swobodnej uniezależnia od własności materiału, w którym cząstka się porusza. A to znakomicie ułatwia życie przy propagacji w niejednorodnej geometrii.

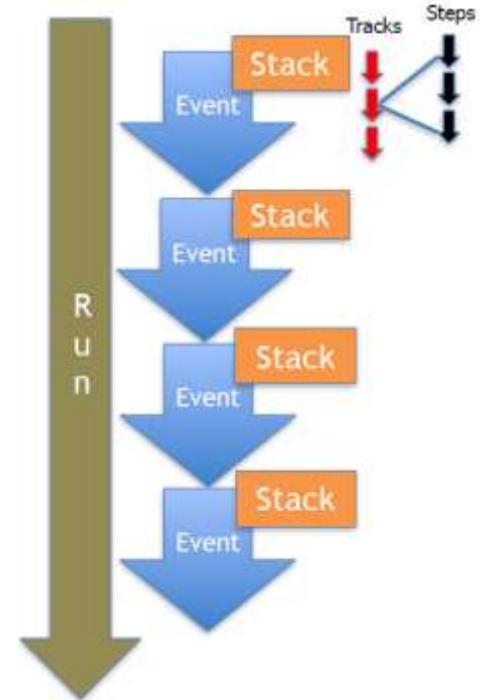
Geant4 – podstawowa struktura

trackManager->[ProcessOneTrack](#)(track);

SteppingManager->[Stepping\(\)](#);

Wyznaczenie długości kroku:

- Wszystkie procesy konkurują – każdy proponuje długość kroku (jako krotność swojej średniej drogi swobodnej). Od zaproponowanej długości odejmowana jest długość poprzedniego kroku (o ile był) – wszystko jako krotności średniej drogi swobodnej.
- Zwycięża ten proces, który zaproponował najkrótszą drogę (po przeliczeniu na „Physical Interaction Length” – czyli na odległość fizyczną/geometryczną).
- Chyba, że bliżej jest do granicy obiektów geometrycznych. Wtedy odległość od granicy określa długość kroku.



Geant4 – podstawowa struktura

```
trackManager->ProcessOneTrack( track );
```

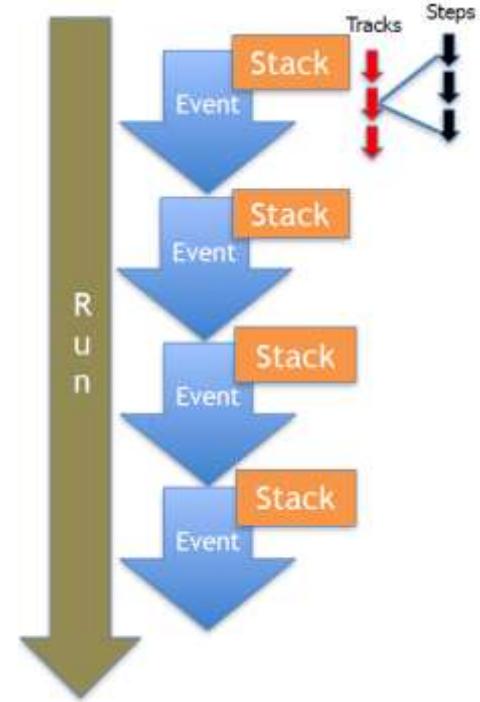
```
SteppingManager->Stepping();
```

Wyznaczenie długości kroku:

- Wszystkie procesy konkurują – każdy proponuje długość kroku (jako krotność swojej średniej drogi swobodnej). Od zaproponowanej długości odejmowana jest długość poprzedniego kroku (o ile był) – wszystko jako krotności średniej drogi swobodnej.
- Zwycięża ten proces, który zaproponował najkrótszą drogę (po przeliczeniu na „Physical Interaction Length” – czyli na odległość fizyczną/geometryczną).
- Chyba, że bliżej jest do granicy obiektów geometrycznych. Wtedy odległość od granicy określa długość kroku.

Wykonywanie procesów:

- Jeśli żywa cząstka jest w spoczynku, losowany i wykonywany jest proces typu „at rest” (annihilacja, rozpad...)



```
if( fTrack->GetTrackStatus() == fStopButAlive ){
    if( MAXofAtRestLoops>0 ){
        InvokeAtRestDoItProcs();
    } else {
        // Find minimum Step length demanded by active disc./cont. processes
        DefinePhysicalStepLength();
        // Invoke AlongStepDoIt
        InvokeAlongStepDoItProcs();
        // Update track by taking into account all changes by AlongStepDolt
        fStep->UpdateTrack();
        // Invoke PostStepDolt
        InvokePostStepDoltProcs();
        if( fSensitive != 0 ) { fSensitive->Hit(fStep); }
        if( fUserSteppingAction != 0 ) { fUserSteppingAction->UserSteppingAction(fStep); }
    }
}
```

Geant4 – podstawowa struktura

```
trackManager->ProcessOneTrack( track );
```

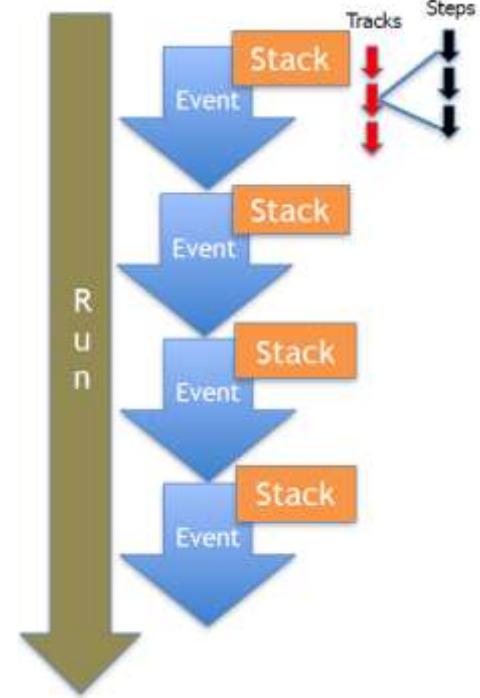
```
SteppingManager->Stepping();
```

Wyznaczenie długości kroku:

- Wszystkie procesy konkurują – każdy proponuje długość kroku (jako krotność swojej średniej drogi swobodnej). Od zaproponowanej długości odejmowana jest długość poprzedniego kroku (o ile był) – wszystko jako krotności średniej drogi swobodnej.
- Zwycięża ten proces, który zaproponował najkrótszą drogę (po przeliczeniu na „Physical Interaction Length” – czyli na odległość fizyczną/geometryczną).
- Chyba, że bliżej jest do granicy obiektów geometrycznych. Wtedy odległość od granicy określa długość kroku.

Wykonywanie procesów:

- Jeśli żywa cząstka jest w spoczynku, losowany i wykonywany jest proces typu „at rest” (annihilacja, rozpad...)
- Po kolej wykonywane są wszystkie procesy „ciągłe” wzdłuż kroku - „along step” (jonizacja, Cerenkov, wielokrotne rozpraszań, itd.)



```
if( fTrack->GetTrackStatus() == fStopButAlive ){
    if( MAXofAtRestLoops>0 ){
        InvokeAtRestDoItProcs();
    } else {
        // Find minimum Step length demanded by active disc./cont. processes
        DefinePhysicalStepLength();
        // Invoke AlongStepDoIt
        InvokeAlongStepDoItProcs();
        // Update track by taking into account all changes by AlongStepDoIt
        fStep->UpdateTrack();
        // Invoke PostStepDoIt
        InvokePostStepDoItProcs();
        if( fSensitive != 0 ) { fSensitive->Hit(fStep); }
        if( fUserSteppingAction != 0 ) { fUserSteppingAction->UserSteppingAction(fStep); }
    }
}
```

„GPIL race”; GPIL – Get Physical Interaction Length

Geant4 – podstawowa struktura

```
trackManager->ProcessOneTrack( track );
```

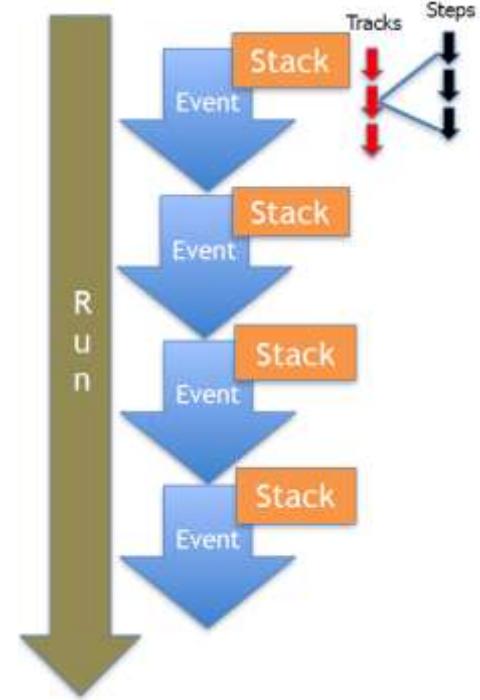
```
SteppingManager->Stepping\(\);
```

Wyznaczenie długości kroku:

- Wszystkie procesy konkurują – każdy proponuje długość kroku (jako krotność swojej średniej drogi swobodnej). Od zaproponowanej długości odejmowana jest długość poprzedniego kroku (o ile był) – wszystko jako krotności średniej drogi swobodnej.
- Zwycięża ten proces, który zaproponował najkrótszą drogę (po przeliczeniu na „Physical Interaction Length” – czyli na odległość fizyczną/geometryczną).
- Chyba, że bliżej jest do granicy obiektów geometrycznych. Wtedy odległość od granicy określa długość kroku.

Wykonywanie procesów:

- Jeśli żywa cząstka jest w spoczynku, losowany i wykonywany jest proces typu „at rest” (annihilacja, rozpad...)
- Po kolej wykonywane są wszystkie procesy „ciągłe” wzdłuż kroku - „along step” (jonizacja, Cerenkov, wielokrotne rozpraszań, itd.)
- Stan cząstki (Track – ślad) jest aktualizowany.



```
if( fTrack->GetTrackStatus() == fStopButAlive ){
    if( MAXofAtRestLoops>0 ){
        InvokeAtRestDoItProcs();
    } else {
        // Find minimum Step length demanded by active disc./cont. processes
        DefinePhysicalStepLength();
        // Invoke AlongStepDoIt
        InvokeAlongStepDoItProcs();
        // Update track by taking into account all changes by AlongStepDolt
        fStep->UpdateTrack();
        // Invoke PostStepDolt
        InvokePostStepDoltProcs();
        if( fSensitive != 0 ) { fSensitive->Hit(fStep); }
        if( fUserSteppingAction != 0 ) { fUserSteppingAction->UserSteppingAction(fStep); }
    }
}
```

Geant4 – podstawowa struktura

```
trackManager->ProcessOneTrack( track );
```

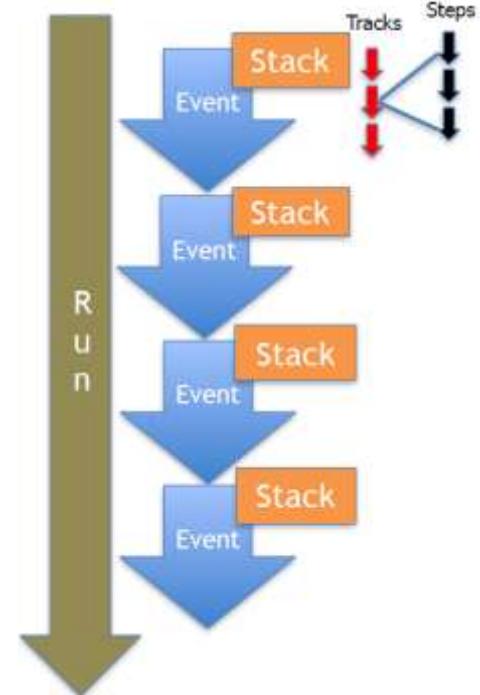
```
SteppingManager->Stepping\(\);
```

Wyznaczenie długości kroku:

- Wszystkie procesy konkurują – każdy proponuje długość kroku (jako krotność swojej średniej drogi swobodnej). Od zaproponowanej długości odejmowana jest długość poprzedniego kroku (o ile był) – wszystko jako krotności średniej drogi swobodnej.
- Zwycięża ten proces, który zaproponował najkrótszą drogę (po przeliczeniu na „Physical Interaction Length” – czyli na odległość fizyczną/geometryczną).
- Chyba, że bliżej jest do granicy obiektów geometrycznych. Wtedy odległość od granicy określa długość kroku.

Wykonywanie procesów:

- Jeśli żywa cząstka jest w spoczynku, losowany i wykonywany jest proces typu „at rest” (annihilacja, rozpad...)
- Po kolei wykonywane są wszystkie procesy „ciągłe” wzdłuż kroku - „along step” (jonizacja, Cerenkov, wielokrotne rozpraszańie, itd.)
- Stan cząstki (Track – ślad) jest aktualizowany.
- Jeśli cząstka ma wciąż niezerową energię wykonywany jest proces dyskretny – „post step” (jeśli to on wyznaczył długość kroku). Po czym resetowana jest jego krotność średniej drogi swobodnej.
- Wykonywane są wszystkie procesy wymuszone typu „post step” (z flagą *Forced*).
- Stan cząstki (track) jest aktualizowany.



```
if( fTrack->GetTrackStatus() == fStopButAlive ){
    if( MAXofAtRestLoops>0 ){
        InvokeAtRestDoItProcs();
    } else {
        // Find minimum Step length demanded by active disc./cont. processes
        DefinePhysicalStepLength();
        // Invoke AlongStepDoIt
        InvokeAlongStepDoItProcs();
        // Update track by taking into account all changes by AlongStepDolt
        fStep->UpdateTrack();
        // Invoke PostStepDolt
        InvokePostStepDoltProcs();
        if( fSensitive != 0 ) { fSensitive->Hit(fStep); }
        if( fUserSteppingAction != 0 ) { fUserSteppingAction->UserSteppingAction(fStep); }
    }
}
```

```
void G4VProcess::ResetNumberOfInteractionLengthLeft() {
    theNumberOfInteractionLengthLeft = -std::log( G4UniformRand() );
    theInitialNumberOfInteractionLength = theNumberOfInteractionLengthLeft;
}
```

Geant4 – podstawowa struktura

```
trackManager->ProcessOneTrack( track );
```

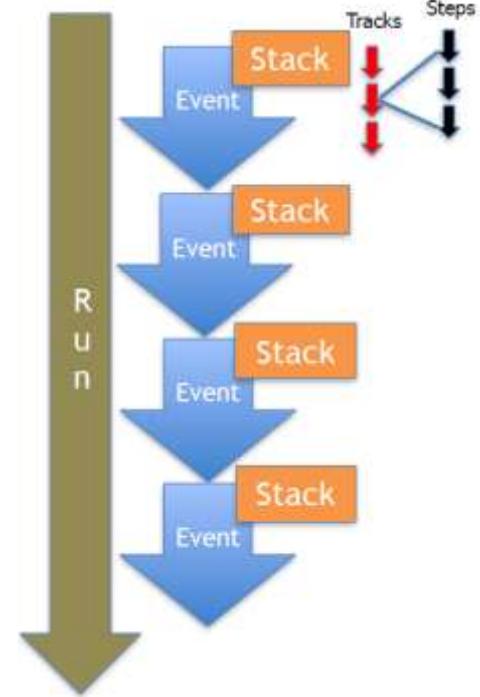
```
SteppingManager->Stepping\(\);
```

Wyznaczenie długości kroku:

- Wszystkie procesy konkurują – każdy proponuje długość kroku (jako krotność swojej średniej drogi swobodnej). Od zaproponowanej długości odejmowana jest długość poprzedniego kroku (o ile był) – wszystko jako krotności średniej drogi swobodnej.
- Zwycięża ten proces, który zaproponował najkrótszą drogę (po przeliczeniu na „Physical Interaction Length” – czyli na odległość fizyczną/geometryczną).
- Chyba, że bliżej jest do granicy obiektów geometrycznych. Wtedy odległość od granicy określa długość kroku.

Wykonywanie procesów:

- Jeśli żywa cząstka jest w spoczynku, losowany i wykonywany jest proces typu „at rest” (annihilacja, rozpad...)
- Po kolei wykonywane są wszystkie procesy „ciągłe” wzdłuż kroku - „along step” (jonizacja, Cerenkov, wielokrotne rozpraszanie, itd.)
- Stan cząstki (Track – ślad) jest aktualizowany.
- Jeśli cząstka ma wciąż niezerową energię wykonywany jest proces dyskretny – „post step” (jeśli to on wyznaczył długość kroku). Po czym resetowana jest jego krotność średniej drogi swobodnej.
- Wykonywane są wszystkie procesy wymuszone typu „post step” (z flagą *Forced*).
- Stan cząstki (track) jest aktualizowany.
- Jeśli cząstka jest w objętości czynnej detektora – wywoływana jest obsługa detektora (dostaje kompletny obiekt G4Step)



```
if( fTrack->GetTrackStatus() == fStopButAlive ){
    if( MAXofAtRestLoops>0 ){
        InvokeAtRestDoItProcs();
    } else {
        // Find minimum Step length demanded by active disc./cont. processes
        DefinePhysicalStepLength();
        // Invoke AlongStepDoIt
        InvokeAlongStepDoItProcs();
        // Update track by taking into account all changes by AlongStepDolt
        fStep->UpdateTrack();
        // Invoke PostStepDolt
        InvokePostStepDoltProcs();
        if( fSensitive != 0 ) { fSensitive->Hit(fStep); }
        if( fUserSteppingAction != 0 ) { fUserSteppingAction->UserSteppingAction(fStep); }
    }
}
```

Geant4 – podstawowa struktura

```
trackManager->ProcessOneTrack( track );
```

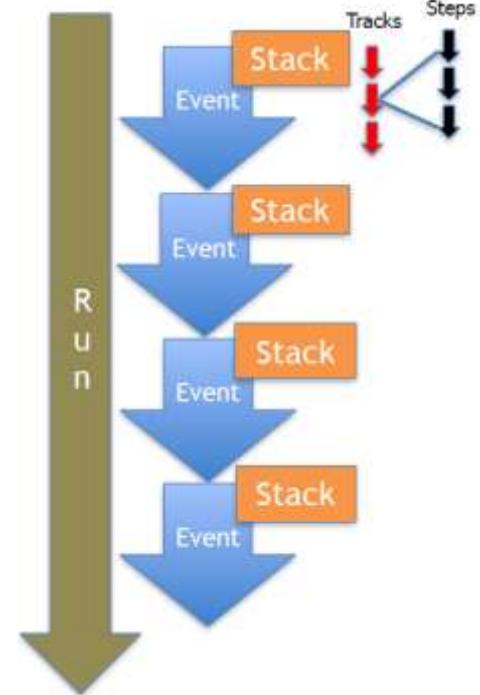
```
SteppingManager->Stepping\(\);
```

Wyznaczenie długości kroku:

- Wszystkie procesy konkurują – każdy proponuje długość kroku (jako krotność swojej średniej drogi swobodnej). Od zaproponowanej długości odejmowana jest długość poprzedniego kroku (o ile był) – wszystko jako krotności średniej drogi swobodnej.
- Zwycięża ten proces, który zaproponował najkrótszą drogę (po przeliczeniu na „Physical Interaction Length” – czyli na odległość fizyczną/geometryczną).
- Chyba, że bliżej jest do granicy obiektów geometrycznych. Wtedy odległość od granicy określa długość kroku.

Wykonywanie procesów:

- Jeśli żywa cząstka jest w spoczynku, losowany i wykonywany jest proces typu „at rest” (annihilacja, rozpad...)
- Po kolei wykonywane są wszystkie procesy „ciągłe” wzdłuż kroku - „along step” (jonizacja, Cerenkov, wielokrotne rozpraszanie, itd.)
- Stan cząstki (Track – ślad) jest aktualizowany.
- Jeśli cząstka ma wciąż niezerową energię wykonywany jest proces dyskretny – „post step” (jeśli to on wyznaczył długość kroku). Po czym resetowana jest jego krotność średniej drogi swobodnej.
- Wykonywane są wszystkie procesy wymuszone typu „post step” (z flagą *Forced*).
- Stan cząstki (track) jest aktualizowany.
- Jeśli cząstka jest w objętości czynnej detektora – wywoływana jest obsługa detektora (dostaje kompletny obiekt G4Step)
- Wywoływana jest UserSteppingAction (o ile została zarejestrowana do RunManagera).



```
if( fTrack->GetTrackStatus() == fStopButAlive ){
    if( MAXofAtRestLoops>0 ){
        InvokeAtRestDoItProcs();
    } else {
        // Find minimum Step length demanded by active disc./cont. processes
        DefinePhysicalStepLength();
        // Invoke AlongStepDoIt
        InvokeAlongStepDoItProcs();
        // Update track by taking into account all changes by AlongStepDolt
        fStep->UpdateTrack();
        // Invoke PostStepDolt
        InvokePostStepDoltProcs();
        if( fSensitive != 0 ) { fSensitive->Hit(fStep); }
        if( fUserSteppingAction != 0 ) { fUserSteppingAction->UserSteppingAction(fStep); }
    }
}
```

Uwaga na obciążenie procesora!

UserSteppingAction jest wykonywana w najbardziej wewnętrznej pętli...

Geant4 – podstawowa struktura

```
trackManager->ProcessOneTrack( track );
```

```
SteppingManager->Stepping\(\);
```

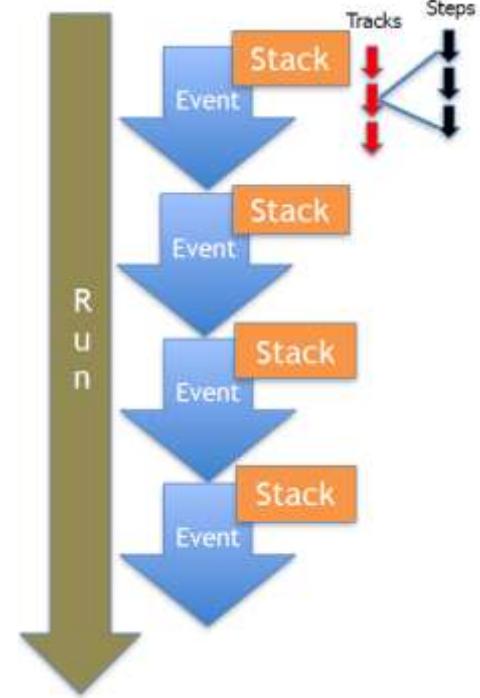
Wyznaczenie długości kroku:

- Wszystkie procesy konkurują – każdy proponuje długość kroku (jako krotność swojej średniej drogi swobodnej). Od zaproponowanej długości odejmowana jest długość poprzedniego kroku (o ile był) – wszystko jako krotności średniej drogi swobodnej.
- Zwycięża ten proces, który zaproponował najkrótszą drogę (po przeliczeniu na „Physical Interaction Length” – czyli na odległość fizyczną/geometryczną).
- Chyba, że bliżej jest do granicy obiektów geometrycznych. Wtedy odległość od granicy określa długość kroku.

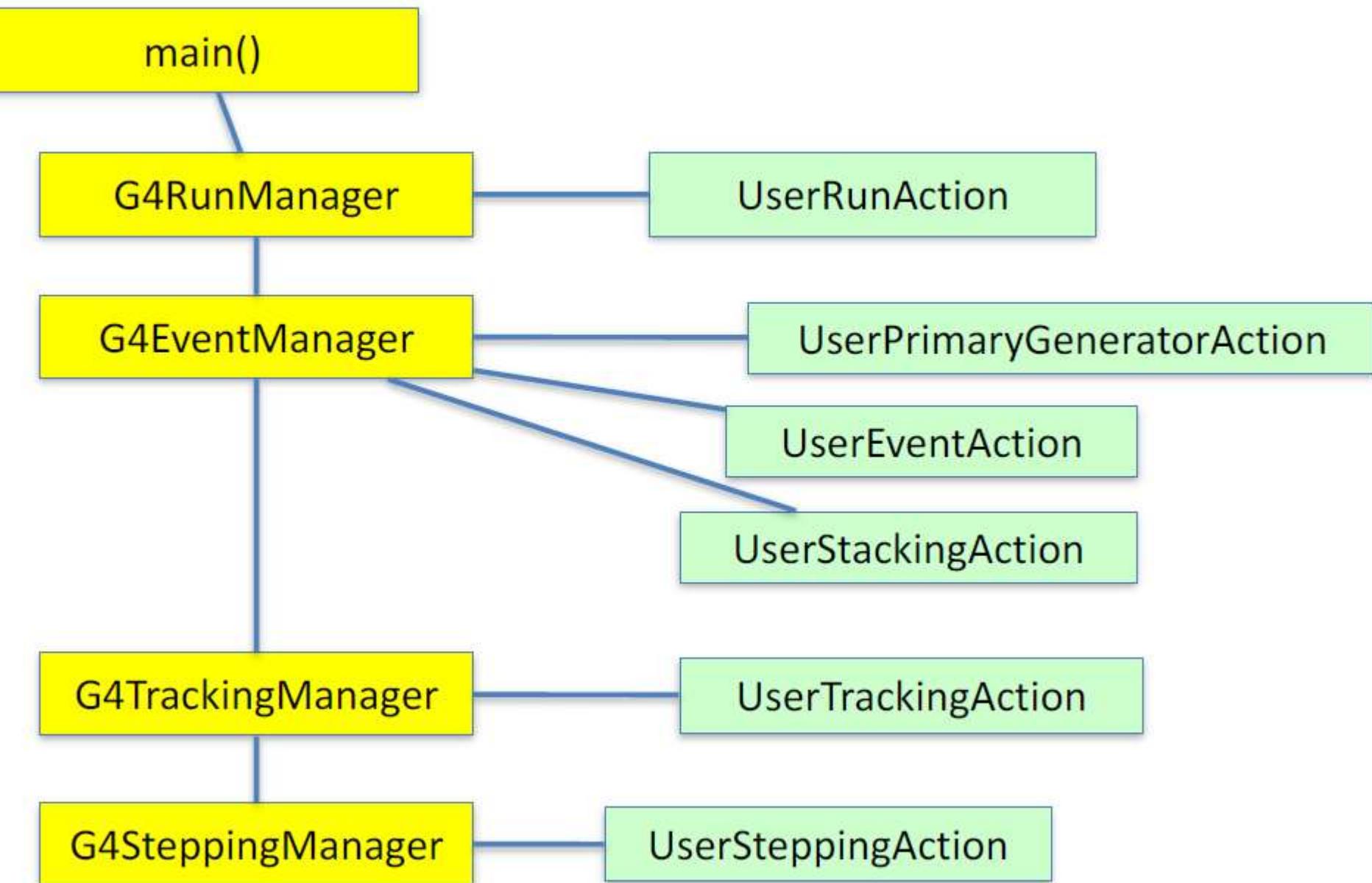
Wykonywanie procesów:

- Jeśli żywa cząstka jest w spoczynku, losowany i wykonywany jest proces typu „at rest” (annihilacja, rozpad...)
- Po kolei wykonywane są wszystkie procesy „ciągłe” wzdłuż kroku - „along step” (jonizacja, Cerenkov, wielokrotne rozpraszanie, itd.)
- Stan cząstki (Track – ślad) jest aktualizowany.
- Jeśli cząstka ma wciąż niezerową energię wykonywany jest proces dyskretny – „post step” (jeśli to on wyznaczył długość kroku). Po czym resetowana jest jego krotność średniej drogi swobodnej.
- Wykonywane są wszystkie procesy wymuszone typu „post step” (z flagą *Forced*).
- Stan cząstki (track) jest aktualizowany.
- Jeśli cząstka jest w objętości czynnej detektora – wywoływana jest obsługa detektora (dostaje kompletny obiekt G4Step)
- Wywoływana jest UserSteppingAction (o ile została zarejestrowana do RunManagera).
- Cząstki wtórne odkładane są na stosie (wywoływane jest UserStackingAction).
- Cząstki wtórne są tworzone jedynie jeśli ich energia przewyższa zadany próg. Jeśli nie, to ich energia jest dodawana do lokalnej depozycji energii.

```
if( fTrack->GetTrackStatus() == fStopButAlive ){
    if( MAXofAtRestLoops>0 ){
        InvokeAtRestDoItProcs();
    } else {
        // Find minimum Step length demanded by active disc./cont. processes
        DefinePhysicalStepLength();
        // Invoke AlongStepDoIt
        InvokeAlongStepDoItProcs();
        // Update track by taking into account all changes by AlongStepDolt
        fStep->UpdateTrack();
        // Invoke PostStepDolt
        InvokePostStepDoltProcs();
        if( fSensitive != 0 ) { fSensitive->Hit(fStep); }
        if( fUserSteppingAction != 0 ) { fUserSteppingAction->UserSteppingAction(fStep); }
    }
}
```



Geant4 – podstawowa struktura. Podsumowanie. Klasy użytkownika



Dziękuję za uwagę



NARODOWE
CENTRUM
BADAŃ
JĄDROWYCH
ŚWIERK

www.ncbj.gov.pl