В предыдущей главе было указано, что кроме одного обязательного для реализации класса действий существует группа опциональных классов действий. Данные классы позволяют пользователю управлять процессом анализа и аккумуляции информации, которая сопутствует процессу моделирования.

Чтобы разобрать назначение тех или иных классов, рассмотрим процесс моделирования в Geant4. Моделирование в Geant4 начинается с Запуска (Run). Свойства геометрии и физических процессов фиксируются ядром Geant4 и становятся неизменными. Пользователь указывает, сколько событий он хочет рассмотреть и запускает программу на «расчет». Запуск начинается с момента старта первого события, а заканчивается моделированием всех вторичных частиц последнего. После завершения запуска пользователь может вновь менять по своему усмотрению геометрию и свойства используемых физических процессов.

Как говорилось ранее, запуск состоит из событий. Понятие события рассматривалось в предыдущей главе. Напомним, события состоят из треков, и моделирование события заканчивается с моделированием последнего трека вторичной частицы в данном событии. Треки моделируются по одному, независимо и последовательно.

Последовательность моделирования треков задается с помощью реализации стека треков. Сам по себе трек содержит все свойства частицы в процессе моделирования прохождения сквозь вещество. Каждый трек знает имя частицы, хранит информацию о её времени жизни и изменении кинетической энергии и т. п.

Все треки состоят из шагов. Шаг — это мельчайшая единица моделирования. Он характеризует расстояние (время), за которое произошло изменение состояния частицы. Это может быть переход из одного объема в

другой, потери энергии на ионизацию, вылет за исследуемую область моделирования и т. п.

Пользователь напрямую не управляет ни одной из этих условных единиц моделирования, однако ему предоставлены классы действий, связанные со своей соответствующей единицей. В рамках данных классов пользователь может анализировать информацию в процессе моделирования, а подчас и вносить некоторые корректировки в процесс моделирования.

Шаги

Шаг является мельчайшей единицей моделирования и самой информативной единицей анализа. Для взаимодействия с информацией на шагах следует унаследовать класс G4UserSteppingAction. У данного класса есть несколько виртуальных методов, однако особого внимания достоин один из них,

```
virtual void UserSteppingAction(const G4Step*){;}
```

а именно:

Этот метод вызывается в конце каждого шага. В качестве аргумента данный метод принимает константный указатель на текущий шаг. Рассмотрим информацию, которую можно извлечь из этого шага.

```
G4StepPoint* GetPreStepPoint() const;
G4StepPoint* GetPostStepPoint() const;
```

Это два самых универсальных метода, содержащих информацию о состоянии частицы в начале и конце шага соответственно. С помощью этих методов, а также информации, содержащейся в объекте G4StepPoint, можно узнать такие вещи, как позиция, объем, в котором находилась/оказалась частица, её энергия и т. п.

К примеру, можно проверить, изменился ли в этом шаге объем:

```
if (aStep->GetPostStepPoint()->GetPhysicalVolume() != nullptr)
   if (aStep->GetPostStepPoint()->GetPhysicalVolume() !=
      aStep->GetPreStepPoint()->GetPhysicalVolume())
      G4cout << "True\n";
```

Примечание: *Стоит обратить внимание на проверку существования объема. Дело в том, что в случае вылета частицы за пределы области моделирования, объема в финальной точке не существует, что может привести к ошибке времени выполнения.*

Существует группа методов, позволяющая определить изменения во времени, позиции и энергии:

```
G4double GetDeltaTime() const;
G4ThreeVector GetDeltaPosition() const;
G4double GetTotalEnergyDeposit() const;
```

К следующим полезным методами класса G4Step стоит отнести:

```
G4int GetNumberOfSecondariesInCurrentStep() const;
const G4TrackVector* GetSecondary() const ;
```

Первый из методов возвращает количество вторичных треков, образовавшихся на текущем шаге. Второй метод дает указатель на std::vector<G4Track*>() скрытый под typedef G4TrackVector, следовательно с данным указателем можно работать как с обычным указателем на вектор.

Также не трудно предположить, что раз через шаг можно получить доступ к трекам вторичных частиц, то можно получить доступ к треку самой рассматриваемой частицы. Для этого предназначен метод:

```
G4Track* GetTrack() const;
```

Треки

Трек содержит максимально полную информацию. Он содержит информацию о типе частицы, процессе, в результате которого она образовалась и ID родителя. За счет трека можно определить, в каком объеме находится

частица. В треке содержится значение кинетической энергии, а также время существования частицы от момента создания и т. п.

Для получения этой информации следует воспользоваться следующими методами для определения:

кинетической энергии:

```
G4double G4Track::GetKineticEnergy() const;
```

типа частицы (Также из этого объекта можно узнать имя частицы):

```
G4ParticleDefinition* GetDefinition() const;
const G4String& GetParticleName() const;     //для определения имени из объекта
                                             //G4ParticleDefinition
```

времени от начала события или от создания трека:

```
G4double GetGlobalTime() const;              // Время от начала события

G4double GetLocalTime() const;               // Время от начала трека
```

Также можно получить информацию о модели и процессе, в следствие которых была создана частица

```
const G4String& GetCreatorModelName() const;     // имя модели
G4int GetCreatorModelID() const;                 //id модели
G4int GetParentID() const;                       //id родителя
G4VProcess* GetCreatorProcess() const;           //указатель на процесс-создатель
```

Можно получить информацию о поляризации, длине трека и т. д. и т. п.

В зависимости от поставленной задачи информацию из трека удобно получать одним из двух способов:

1. Как было показано выше, через шаг.
2. За счет соответствующего класса действий G4UserTrackAction

Класс G4UserTrackAction содержит два метода для перегрузки:

```
virtual void PreUserTrackingAction(const G4Track*){;}
virtual void PostUserTrackingAction(const G4Track*){;}
```

где метод PreUserTrackingAction вызывается в начале каждого трека (даже для первой первичной частицы в событии он вызывается уже после начала события), а метод PostUserTrackingAction в конце события.

Стек при работе с треками

Еще один класс действий - G4UserStackingAction, является чисто техническим классом для определения порядка обработки треков и т. п. В рамках данной книги данный класс рассмотрен не будет.

События

Неоднократно подчеркивалось, что событие представляет собой объединение треков первичных частиц, образованных в рамках однократного вызова метода void G4VUserPrimarygeneratorAction::GeneratePrimaries(G4Event*) и появляющихся в следствие их моделирования треков вторичных частиц.

Классом действий, связанным с G4Event, является G4UserEventAction. Аналогично классу действий над треками класс G4userEventAction имеет два метода для перегрузки

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

Метод BeginOfEventAction вызывается в начале каждого события, а метод EndOfEventAction в конце.

G4Event в отличии от G4Step и G4Track не хранит особой информации о процессе моделирования. К полезным стоит отнести методы получения информации о первичной вершине:

```
G4int GetNumberOfPrimaryVertex() const;                    //количество вершин
G4PrimaryVertex* GetPrimaryVertex(G4int i=0)  const;       //доступ к вершине по номеру
```

Может оказаться полезным метод

```
G4int GetEventID() const;
```

который возвращает ID текущего события. К примеру, его можно использовать, чтобы выводить сообщение о начале i-го события с целью оценки оставшегося времени выполнения

```
G4int printable = 10;
if(anEvent->GetEventID()%printable==0)
   G4cout<<anEvent->GetEventID()<<G4endl;
```

(каждое 10-ое событие на экран будет выводиться его номер)

В остальном роль класса G4UserEventAction в программе определяется конечным пользователем и его конкретной программой.

Запуски

Самой крупной единицей моделирования является запуск. С началом запуска начинается цикл моделирования, а конец запуска совпадает с завершением цикла.

За запуски отвечает класс G4Run, а соответствующим классом действий является G4UserRunAction. В данном классе доступны два метода для перегрузки

```
virtual void BeginOfRunAction(const G4Run* aRun);
virtual void EndOfRunAction(const G4Run* aRun);
```

вызываемых в начале и конце каждого запуска.

Также в данном классе доступен метод

```
virtual G4Run* GenerateRun();
```

позволяющий генерировать вместо стандартных объектов G4Run объекты его потомков.

Чтоже касается остальных функций данного класса, то, как и в случае с G4UserEventAction, они зависят от реализации конкретной программы.

Сбор информации с шагов в запуске

До сих пор были рассмотрены только личные особенности конкретных классов действий, однако не говорилось о взаимодействии между ними. Дело в том, что универсальной цепочки связи между ними в базовых классах нет. Все базовые классы действий содержат лишь конструкторы по умолчанию. Однако за счет особенностей наследования пользователь волен сам определить, какой класс и как будет связан с другими.

Ранее был рассмотрен класс G4VUserActionInitialization. В его методе Build() осуществляется генерация всех необходимых объектов для конкретного потока. Изменив конструкторы классов действий таким образом, что один из них будет принимать указатель на другой, можно получить необходимую связь между ними.

Допустим, мы хотим узнать, сколько протонов образуется в результате моделирования:

Нас не интересует, на каком треке и в рамках какого события образовался протон, а нужен лишь факт образования. Следовательно, мы будем на шагах проверять факт образования вторичной частицы с именем «proton» и передавать об этом информацию напрямую в запуск. Отсюда легко заметить, что нам необходимо осуществить связь между G4UserSteppingAction и G4UserRunAction. Для этого стоит передать указатель на потомка G4UserRunAction в объект потомка G4UserSteppingAction с помощью

```
#include "G4UserSteppingAction.hh"
#include "RunAction.hh"


class SteppingAction : public G4UserSteppingAction
{
public:
   explicit SteppingAction(RunAction*);

   void UserSteppingAction(const G4Step*) override;
private:
   RunAction* runAction;
};
```

конструктора:

и осуществить связь от потомка G4UserSteppingAction к потомку G4UserRunAction в методе Build()

```
void Action::Build()const {
   SetUserAction(new PrimaryPat);              //генератор первичных частиц

  auto runAction = new RunAction;
   SetUserAction(new SteppingAction(runAction));
   SetUserAction(runAction);
}
```

Теперь для решения поставленной задачи достаточно создать в RunAction целочисленную переменную - счетчик и метод по инкрементации её значения

```
class RunAction : public G4UserRunAction {
public:
   void EndOfRunAction(const G4Run *aRun) override;

   void BeginOfRunAction(const G4Run *aRun) override;

private:
   G4int counter;
public:
   void AddEvent() { counter++; }
};
```

В методе BeginOfRunAction значение счетчика будет обнуляться:

```
void RunAction::BeginOfRunAction(__attribute__((unused)) const G4Run *aRun) {
   counter = 0;
}
```

а в методе EndOfRunAction выводиться в консоль

```
 void RunAction::EndOfRunAction(__attribute__((unused)) const G4Run *aRun) {
    G4cout << "Result = " << counter << '\n';
 }
```

Наконец, само событие по образованию протонов будет отслеживаться в методе

```
void SteppingAction::UserSteppingAction(const G4Step * aStep) {
   if (aStep->GetNumberOfSecondariesInCurrentStep() != 0)
      for (auto &item: *aStep->GetSecondary())
         if (item->GetDefinition()->GetParticleName() == "proton")
            runAction->AddEvent();
}
```

В зависимости от особенностей той или иной задачи условия и связи между классами будут абсолютно разными. Универсального решения всех задач не существует.

Примечание: *не следует забывать, что в случае расчета в многопоточном режиме финальный результат, выводимый в EndOfRunAction, соответствует тем результатам, которые были получены на конкретном рабочем потоке. Для получения общего результата по всем потокам данные нужно «сшивать».*