Классы действий

Как было сказано ранее, в Geant4 существуют два основных класса инициализации свойств моделирования: первый отвечает на вопрос ГДЕ осуществляется моделирование(геометрия), а второй - КАК и по каким законам это моделирование происходит (физические процессы). Кроме того, существует группа классов «Действий» над частицей в процессе моделирования. Главным среди них является класс, определяющий КАКИМ ОБРАЗОМ, генерируются частицы, а его реализация становится обязательным шагом при написании программы для моделирования взаимодействия частиц с веществом на основе Geant4. В добавлении к нему существует группа необязательных классов действий, позволяющих извлекать пользователю информацию на различных этапах моделирования, анализировать её и аккумулировать.

Инициализация всех классов действий отличается от способа, используемого для геометрии и физических процессов. Дело в том, что, начиная с версии Geant4.10, была реализована поддержка многопоточности. С этого момента геометрия и физические процессы остались общей информацией, существующей в единственном экземпляре на мастер-потоке, а объекты классов действий стали уникальными для каждого потока - рабочего.

В связи с этим в Geant4 появился класс G4VUserActionInitialization, основным назначением которого является создание объектов классов действий для рабочих потоков.

Рассмотрим основные классы действий и способ их инициализации.

К классам действий относятся:

- G4VUserPrimaryGeneratorAction
- G4UserRunAction
- G4UserEventAction
- G4UserStackingAction
- G4UserTrackingAction
- G4UserSteppingAction

Их инициализация осуществляется за счет **protected** методов SetUserAction(). Данные методы вызываются в методе Build() потомка G4VUserActionInitialization для создания объектов рабочих потоков.

Например, потомок классах G4VUserActionInitialization может выглядеть

```
class my_Actions : public G4VUserActionInitialization {
public:
   void Build() const override {
      SetUserAction(new my_run_action);
      SetUserAction(new my_stack_action);
      SetUserAction(new my_event_action);
      SetUserAction(new my_track_action);
      SetUserAction(new my_step_action);

      SetUserAction(new my_primary_gen);
   }
};
```

следующим образом:

где my_* - потомки классов действий, объекты которых инициализируются для рабочих потоков.

Если же необходимо (допустимо только для UserRunAction) создать объект только для мастер-потока, то следует вызвать метод SetUserAction() в методе BuildForMaster() вместо метода Build(), однако в рамках данной книги этот способ рассмотрен не будет.

Генерация первичных событий

Обязательным для реализации в программе моделирования является только один класс действий — G4VUserPrimaryGeneratorAction.

Данный абстрактный класс содержит один чисто виртуальный метод:

```
virtual void GeneratePrimaries(G4Event* anEvent) = 0;
```

Однако, чтобы понять принцип работы данного класса, рассмотрим некоторые понятия, связанные с генерацией первичных частиц в Geant4.

Первичная частица и её трек, а также все вторичные частицы и их треки, образовавшиеся в процессе взаимодействия первичной частицы с веществом называются событием.

Первичное положение частицы в пространстве математической модели называется вершиной.

Класс G4VUserPrimaryGeneratorAction отвечает за связь между создаваемыми пользователем первичными частицами и их запуском в рамках модели. Сам по себе данный класс никаких первичных частиц не создает. Для этой цели в Geant4 служит базовый класс G4VPrimaryGenerator и группа его потомков. Для простоты опустим архитектуру класса G4VPrimaryGenerator и сразу перейдем непосредственно к рассмотрению одного из его потомков G4ParticleGun — самый простой и универсальный генератор частиц.

G4ParticleGun позволяет одновременно настраивать как первичную частицу, так и её вершину. Для это в данного классе представлены следующие методы:

для первичной частицы:

1. Установка типа первичной частицы

```
void SetParticleDefinition (G4ParticleDefinition * aParticleDefinition);
```

2. Кинетическая энергия первичной частицы

```
void SetParticleEnergy(G4double aKineticEnergy);
```

```
void SetParticleMomentum(G4double aMomentum);
```

3. Импульс частицы

либо

```
void SetParticleMomentum(G4ParticleMomentum aMomentum);
```

где в качестве аргумента G4ParticleMomentum может использоваться объект класса G4ThreeVector.

4. Направление импульса

```
void SetParticleMomentumDirection(G4ParticleMomentum aMomentumDirection);
```

5. Заряд частицы (если доступно)

```
void SetParticleCharge(G4double aCharge);
```

6. Поляризация (если доступно)

```
void SetParticlePolarization(G4ThreeVector aVal);
```

Для вершины первичной частицы представлены следующие методы:

1. Позиция вершины

```
void SetParticlePosition(G4ThreeVector aPosition);
```

2. Стартовое время

```
void SetParticleTime(G4double aTime);
```

Кроме всего прочего, класс G4ParticleGun имеет три варианта конструкторов:

```
G4ParticleGun();
G4ParticleGun(G4int numberofparticles);
G4ParticleGun(G4ParticleDefinition * particleDef, G4int numberofparticles = 1);
```

где numberofparticles отвечает за количество частиц, запускаемых из первичной вершины с текущими начальными свойствами в рамках одного события.

Теперь более подробно рассмотрим методы 2,3,4 для первичной частицы.

Если направление и энергия первичной частицы не заданы с помощью методов, то ей будет присвоено направление вдоль оси X (1,0,0) с энергией в 1000 МэВ.

Метод SetParticleMomentum(G4ParticleMomentum aMomentum) в качестве аргумента принимает G4ThreeVector (ибо G4ParticleMomentum есть typedef от G4ThreeVector). Направление вектора будет соответствовать направлению частицы, а его длинна — энергии из расчета, что единица длинны это 1 МэВ.

Однако для лучшей читаемости кода рекомендуется использовать пару методов SetParticleEnergy(G4double) и SetParticleMomentumDirection(G4ParticleMomentum), где первый отвечает за кинетическую энергию частицы, а второй только за её направление без влияния на энергию.

В качестве примера рассмотрим моноэнергетический точечный изотропный гамма - источник.

Настройки G4ParticleGun будут выглядеть следующим образом:

```
gun = new G4ParticleGun(1);
gun->SetParticleDefinition(G4Gamma::GammaDefinition());
gun->SetParticlePosition(G4ThreeVector(0, 0, 0));
gun->SetParticleEnergy(661 * keV);
gun->SetParticleMomentumDirection(G4ThreeVector(G4UniformRand(), G4UniformRand(),
G4UniformRand()));
```

где G4UniforRand() возвращает случайное дробное число от 0 до 1.

Теперь рассмотрим, как передать заданные настройки для моделирования. Метод GeneratePrimaries(G4Event*anEvent) класса G4VUserPrimaryGenerationAction вызывается единожды на каждом отдельно взятом событии. Для того, чтобы передать настройки моделирования первичной

частицы, следует вызвать метод GeneratePrimaryVertex(G4Event*) для используемого генератора.

Соответственно, возвращаясь к выше указанным настройкам, метод GeneratePrimaryVertex(G4Event*) примет следующую форму.

```
void PrimaryPat::GeneratePrimaries(G4Event* anEvent) {
   auto gun = new G4ParticleGun(1);
   gun->SetParticleDefinition(G4Gamma::GammaDefinition());
   gun->SetParticlePosition(G4ThreeVector(0, 0, 0));
   gun->SetParticleEnergy(661 * keV);
   gun->SetParticleMomentumDirection(G4ThreeVector(G4UniformRand(), G4UniformRand(),
G4UniformRand()));

   gun->GeneratePrimaryVertex(anEvent);
}
```

Стоит отметить что, если в качестве аргумента конструктора G4ParticleGun указать значение больше, чем 1, все запускаемые в рамках одного события частицы будут иметь одинаковые свойства (включая направление, а новые случайные значения будут разыгрываться только для частиц, запускаемых на следующем событии).

Пример использования энергетического спектра для генератора первичного излучения.

Как было показано выше, на каждом отдельном событии генератор имеет вполне конкретное значение кинетической энергии частицы. Посмотрим, как можно решить большинство типовых задач, в которых энергия задана функциональной зависимостью.

Пусть энергия частицы(e) на отрезке [A, B] задана плотностью распределения foo(x), где foo(x) описана функцией

```
G4double foo(G4double e);
```

создадим контейнер типа std::vector<G4double> для формирования ключей конечной функции распределения.

```
keys = new std::vector<G4double>();
```

Выбрав шаг *d*e необходимой точности, перейдем к дискретному распределению следующим образом:

записываем соответствующие значения функции в контейнер keys в выбранных точках,

в переменной sum накапливаем интеграл, для последующего перехода к

```
G4double buf, sum{0}, i;
for (i = a; i < b;i += de) {
   buf = foo(i);
   sum += buf;
   keys->push_back(buf);
}
```

вероятностям

и преобразуем контейнер, переходя к вероятностям

Таким образом, мы получили контейнер с вероятностями для дискретной величины. Осуществим интегрирование в этом же цикле для перехода к функции распределения вероятности:

```
for (double &key : *keys)
   key /= sum;
*keys->begin()/=sum;
for (auto item = ++keys->begin(); item!=keys->end();item++)
   ((*item /= sum) += *(item-1));
```

и построим контейнер типа std::map<G4double,G4double>, который будет содержать в качестве ключей значения функции распределения, а в качестве значений выбранные ранее значения энергии. В результате получим

```
keys = new std::vector<G4double>();
result = new std::map<G4double, G4double>();

G4double buf, sum{0}, i;
for (i = a; i < b;i += de) {
   buf = foo(i);
   sum += buf;
   keys->push_back(sum);
}

*keys->begin()/=sum;
for (auto item = ++keys->begin(); item!=keys->end();item++)
   ((*item /= sum) += *(item-1));

auto item = keys->begin();
for (i = a; i < b;i += de)
   result->emplace(*item++, i);
```

Можно заметить, что если сначала осуществить интегрирование, а уже потом нормировку, код заметно упростится:

```
keys = new std::vector<G4double>();
result = new std::map<G4double, G4double>();

G4double buf, sum{0}, i;
for (i = a; i < b; i += de) {
   buf = foo(i);
   sum += buf;
   keys->push_back(sum);

}

auto item = keys->begin();
for (i = a; i < b; i += de)
   result->emplace(*item++ / sum, i);
```

Теперь для установки значения энергии по заданному закону достаточно воспользоваться методом

```
void SetParticleEnergy(G4double aKineticEnergy);
```

в следующей форме:

```
gun->SetParticleEnergy(result->upper_bound(G4UniformRand())->second);
```