Моделирование отклика детектора

- Любой логический объем в модели можно объявить детектирующим, или «чувствительным». При этом при прохождении частицы через данный объем моделируется срабатывание детектора.
- Детектирующих объемов одновременно может несколько. Обработка срабатываний при этом может происходить по разному.
- Дополнительно можно смоделировать оцифровку сигнала и электронный отклик детектора

Описание детектирующего объема (I)

- Создается класс-наследник класса G4VSensitiveDetector
- Описываются обязательные методы
  - **Initialize()** вызывается в начале каждого события
  - **ProcessHits()** вызывается на каждом шаге в детектирующем объеме. Позволяет получить информацию о характеристиках частицы в данной точке, о взаимодействии с веществом, и смоделировать срабатывание детектора
  - **EndOfEvent()** вызывается в конце события. Позволяет провести отбор срабатываний, и сохранить результаты

Описание детектирующего объема (II)

- Инициализируется объект класса G4SDManager

G4SDManager fSDM = G4SDManager::GetSDMpointer();*

- Объект, описывающий детектирующий объем, регистрируется в менеджере

fSDM->AddNewDetector(G4VSensitiveDetector)*

- Детектирующий объем ассоциируется с логическим объемом

logVolume->SetSensitiveDetector(G4VSensitiveDetector)*

Срабатывание

В Geant4 предусмотрен механизм сбора и сохранения информации о срабатываниях

- **G4VHit** – информация об одном срабатывании Объекты создаются в методе G4VSensitiveDetector::ProcessHits()
- **G4VHitsCollection** – класс-контейнер (коллекция) нескольких срабатываний
- **G4HCofThisEvent** – класс-контейнер разнородных коллекций срабатываний в событии

Пример класса G4VHit

```
class ExN04TrackerHit : public G4VHit
{
  public:

      ExN04TrackerHit();
      ~ExN04TrackerHit();
      void Draw() const;     // обязательная функция
      void Print() const;     // обязательная функция

  private:
      G4double edep;    // ионизационные потери
      G4ThreeVector pos;   // положение срабатывания в пространстве

  public:  // функции доступа к  членам класса
      inline void SetEdep(G4double de)      { edep = de; }
      inline G4double GetEdep() const      { return edep; }
      inline void SetPos(G4ThreeVector xyz)          { pos = xyz; }
      inline G4ThreeVector GetPos() const             { return pos; }
};
```

Инициализация коллекций срабатываний

```
void ExN04TrackerSD::Initialize(G4HCofThisEvent* HCE)
{
  static int HCID = -1;
  trackerCollection = new ExN04TrackerHitsCollection
                    ("SensitiveDetectorName", "collectionName");
  if(HCID<0)
  { HCID = GetCollectionID(0); }
  HCE->AddHitsCollection(HCID,trackerCollection);
}
```

Заполнение информации о срабатывании

```
G4bool ExN04TrackerSD::ProcessHits(G4Step* aStep,
                                   G4TouchableHistory*)
{
  G4double edep = aStep->GetTotalEnergyDeposit();
  if(edep==0.) return false;

  ExN04TrackerHit* newHit = new ExN04TrackerHit();
  newHit->SetEdep( edep );
  newHit->SetPos( aStep->GetPreStepPoint()->GetPosition() );
  trackerCollection->insert( newHit );

  return true;
}
```

Как получить основные характеристики частицы

- **Положение начала шага:** aStep->GetPreStepPoint()->GetPosition()
- **Положение конца шага:** aStep->GetPostStepPoint()->GetPosition()
  (эквивалентно aTrack->GetPosition())
- **Время:** aTrack->GetLocalTime() (от образования трека)
- aTrack->GetGlobalTime() (от начала события)
- **Тип частицы:** aTrack->GetParticleDefinition()->GetPDGEncoding()
- **Импульс:** aTrack->GetMomentum()
- **Направление:** aTrack->GetMomentumDirection()
- **Энергия:** aTrack->GetKinetic Energy()
- **Энерговыделение на шаге:** aStep->GetTotalEnergyDeposit()
  aStep->GetNonIonizingEnergyDeposit()
- **Физический объем:** aTrack->GetVolume()
- **ID трека:** aTrack->GetTrackID()
- **ID родительской частицы:** aTrack->GetParentID()
-
- aTrack = aStep->GetTrack()

Пример доступа к информации о срабатываниях

```
G4SDManager* fSDM = G4SDManager::GetSDMpointer();
G4RunManager* fRM = G4RunManager::GetRunManager();
G4int collectionID =
        fSDM->GetCollectionID("collection_name");
 const G4Event* currentEvent =
        fRM->GetCurrentEvent();
G4HCofThisEvent* HCofEvent =
        currentEvent->GetHCofThisEvent();
MyHitsCollection* myCollection = (MyHitsCollection*)
  (HC0fEvent->GetHC(collectionID));
```

Универсальные детекторы

- G4MultifunctionalDetector — разновидность G4VSensitiveDetector, в которой вместо кода пользователя используются стандартные объекты-счетчики (G4PrimitiveScorer)
- Каждый счетчик сохраняет значения определенной физической величины для каждого срабатывания (шага)
- Есть возможность добавлять фильтры для отбора сохраняемых значений по характеристикам частицы

Счетчики

- G4PSTrackLength
- G4PSEnergyDeposit
- G4PSDoseDeposit
- G4PSFlatSurfaceCurrent
- G4PSSphereSurfaceCurrent
- G4PSCellFlux
- G4PSNofSecondary
- G4PSCellCharge
- ...

всего более 50

Фильтры

- Наследник класса G4VSDFilter
- Метод *G4bool Accept(const G4Step*)* - *если true, то счетчик считает информацию о данном шаге*
- Стандартные фильтры:
- **G4SDChargedFilter** - все заряженные
- **G4SDNeutralFilter** - все нейтральные
- **G4SDParticleFilter** - по типу частиц
- **G4SDKineticEnergyFilter** - по диапазону энергии
- **G4SDParticleWithEnergyFilter** - по типу частиц и диапазону энергии

Пример

examples/extended/runAndEvent/RE06/

```
void RE06DetectorConstruction::SetupDetectors()
{
  G4String filterName, particleName;

  G4SDParticleFilter* gammaFilter =
    new G4SDParticleFilter(filterName="gammaFilter",particleName="gamma");
  G4SDParticleFilter* epFilter = new G4SDParticleFilter(filterName="epFilter");
  epFilter->add(particleName="e-");
  epFilter->add(particleName="e+");

    G4MultiFunctionalDetector* det = new G4MultiFunctionalDetector("detector");

    G4VPrimitiveScorer* primitive;
    primitive = new G4PSEnergyDeposit("eDep",copyNo);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSNofSecondary("nGamma",copyNo);
    primitive->SetFilter(gammaFilter);
    det->RegisterPrimitive(primitive);

    G4SDManager::GetSDMpointer()->AddNewDetector(det);
    logicalVolume->SetSensitiveDetector(det);
}
```

Оцифровка сигнала

- Физические величины, полученные при срабатывании детектора, могут быть преобразованы в электронный сигнал, аналогичный непосредственно регистрируемому системами сбора данных
- Позволяет
  - ***моделировать работу АЦП и время-цифровых преобразователей***
  - ***моделировать схему сбора данных***
  - ***моделировать триггер***
  - ***моделировать наложение событий (pile-up)***
  - ***производить данные в формате, аналогичном формату электроники считывания***

Пример получения информации из счетчика

```
G4THitsMap<G4double>* evtMap = (G4THitsMap<G4double>*)(HCE->GetHC(collectionID));
std::map<G4int,G4double*>::iterator itr = evtMap->GetMap()->begin();
for(; itr != evtMap->GetMap()->end(); itr++)
    {
      G4int key = (itr->first);          // номер копии логического объема
      G4double val = *(itr->second);     // значение физ. величины
       .....
    }
```

- Каждому модулю электроники соответствует объект-наследник класса G4VDigitizerModule, определяемого пользователем
  - обязательный метод **Digitize**()
- Оцифрованный сигнал хранится в объекте-наследнике класса G4VDigi, определяемого пользователем
- Отдельные оцифрованные сигналы могут объединяться в коллекции (G4VDigiCollection)
- Управление оцифровкой осуществляется объектом класса G4DigiManager

Работа с модулями оцифровки сигнала

Инициализация

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
MyDigitizer * myDM = new MyDigitizer("/myDet/myCal/myEMdigiMod" );
fDM->AddNewModule(myDM);
```

Доступ к объекту и оцифровка (конструирование объектов-наследников G4VDigi)

```
MyDigitizer * myDM =
     fDM->FindDigitizerModule( "/myDet/myTDCdigiMod" );
 myDM->Digitize();
```