Geant 4

Detector Response

Acknowledgements:
A. Lechner,
J. Apostolakis, M. Asai, G. Cosmo, A. Howard

Maria Grazia Pia
Overview

Concepts
- Readout geometry
- Sensitive detector
- Hits
- Digis

Details
- Hit class
- Sensitive detector class
- Hits Collection class and its use

Basic features
- Digitizer module and digit
- Touchable
Basic concepts
Extracting information from the simulation

Once the mandatory classes (*DetectorConstruction*, *PhysicsList*, *PrimaryGeneration*) are implemented, the Geant4 application does not yet include functionality to extract information produced in the simulation.

A user **must provide his/her own code**
- To extract information relevant to the simulation application
- To describe the detector response

Geant4 concepts for such functionality are
- **Sensitive Detector** (optionally with *Readout Geometry*)
- **Hits** and **Hits Collections**
- **Digis** and **Digis Collections**

Maria Grazia Pia
Readout geometry

- Readout geometry is a **virtual** and **artificial** geometry which can be defined in parallel to the real detector geometry.
  - Tracks will be tracked in the “real” geometry, but the sensitive detector can have its own geometry for readout purposes.
    - E.g. to find the cell the current hit belongs to.
  - The readout geometry is **optional**; one may have more than one.
    - Each one should be associated to a sensitive detector.
    - But a sensitive detector is not required to have a readout geometry.
- **A step is not limited by** the boundary of the readout geometry.

During tracking, the G4TouchableHistory of the RO geometry according to the **PreStepPoint position** will be provided to the sensitive detector.

The Tracking Geometry
Built by G4VUserDetectorConstruction

A Readout Geometry
Built by G4VReadoutGeometry
How to create a ReadOut Geometry

- Derive your own concrete class from the `G4VReadoutGeometry` abstract base class

- The geometry setup is done in the same way as for the “real” tracking geometry:
  - Create solids, logical and physical volumes

- However, the materials used in the RO geometry are dummy materials
  - i.e. they are not used

- A sensitive detector for the RO geometry must be defined but is not used
  - This means that you need to declare the sensitive parts of the RO geometry by setting a non-NULL sensitive detector pointer to the logical volume

Maria Grazia Pia
Readout Geometry

```cpp
// header file: MyReadOutGeometry.hh
#include “G4VReadoutGeometry.hh“
...
class MyReadOutGeometry : public G4VReadoutGeometry {
public:
    MyReadOutGeometry();
    virtual ~MyReadOutGeometry();
    virtual G4VPhysicalVolume* Build(); // must return the physical world
    // volume of the readout geometry
private:
    ....
};
```

your RO geometry class
Hit

- Hit is a **user-defined** class, which derives from the G4VHit base class

- You can store various kind of information by implementing your own concrete Hit class

- Typically one records quantities like:
  - Position and time of the step
  - Momentum and energy of the track
  - Energy deposit in the step
  - Geometrical information
  - etc.
Example of a Hit class

// header file: MyHit.hh

#include “G4VHit.hh”

class MyHit : public G4VHit {

public:
    MyHit();
    virtual ~MyHit();
    ...
    inline void SetEnergyDeposit(double energy) { energyDeposit = energy; }
    inline double GetEnergyDeposit() { return energyDeposit; }
    ...

private:
    G4double energyDeposit;
    ...
};

Maria Grazia Pia
Sensitive Detector

A logical volume becomes sensitive if it has a pointer to a SensitiveDetector (SD)

A SensitiveDetector can be instantiated several times, where the instances are assigned to different logical volumes
- SD objects must have unique detector names
- A logical volume can only have one SD object attached
- But you can implement your detector to have multiple functionality

Two possibilities to make use of the SD functionality
- Create your own sensitive detector
  - Highly customizable
- Use Geant4 built-in tools
  - Primitive Scorers

Maria Grazia Pia
Creating your own sensitive detector

- A powerful way of extracting information from the physics simulation is to define your own SD.
- The ingredients of the scoring setup are:

<table>
<thead>
<tr>
<th>Base class</th>
<th>Concrete class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensitive Detector</td>
<td>MySensitiveDetector</td>
</tr>
<tr>
<td>Readout Geometry</td>
<td>MyReadoutGeometry</td>
</tr>
<tr>
<td>Hit</td>
<td>MyHit</td>
</tr>
<tr>
<td>G4VSensitiveDetector</td>
<td></td>
</tr>
<tr>
<td>G4VReadoutGeometry</td>
<td></td>
</tr>
<tr>
<td>G4VHit</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Template class</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hits Collection</td>
<td>G4THitsCollection&lt;your hit class&gt;</td>
</tr>
</tbody>
</table>

- Derive your own concrete classes from the base classes and customize them according to your needs.

Maria Grazia Pia
Basic strategy to retrieve information - 1

Assume that you have already created the detector geometry
- Shape and size (Solid) of your detector, Material
- Logical volumes
- Physical volumes

Implement a sensitive detector and assign an instance of it to the logical volume of your detector geometry setup
- Then this volume becomes sensitive
- The sensitive detector will become “active” for each particle step, if the step starts inside this logical volume

Optionally: implement a readout geometry and attach it to the sensitive detector

Maria Grazia Pia
Adding sensitivity to a logical volume

- Create an instance of a sensitive detector
- Register the sensitive detector to the SD manager
- Assign the pointer of your SD to the logical volume of your detector geometry

```cpp
G4VSolid* boxSolid = new G4Box("aBoxSolid", 1.* cm, 1.* cm, 1.* cm);
G4LogicalVolume* boxLog =
    new G4LogicalVolume(boxSolid, materialSilicon, "aBoxLog", 0, 0, 0);
G4VSensitiveDetector* sensitiveBox = new MySensitiveDetector("MyDetector");
G4SDManager* SDManager = G4SDManager::GetSDMPointer();
SDManager -> AddNewDetector(sensitiveBox);
boxLog -> SetSensitiveDetector(sensitiveBox);`
Basic strategy to retrieve information - 2

Then, create **Hit** objects in your sensitive detector using information from particle steps

**Hit** is a snapshot of the physical interaction of a track or an accumulation of interactions of tracks in the sensitive or interesting region of your detector
   - *You should create hit class(es) according to your needs*

Use **Touchable** of the **Readout Geometry** to retrieve geometrical information associated with hits

Store your hits in **Hit Collections**
   - hit collections are automatically associated to the G4Event object

Finally, process the information associated with hits in User Action classes (G4UserEventAction, G4UserRunAction etc.) to obtain a summary of the relevant event/run features

Maria Grazia Pia
Using built-in scorers

Alternatively, you can use a predefined sensitive detector `G4MultiFunctionalDetector` and primitive scorers:

<table>
<thead>
<tr>
<th>Concrete class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensitive Detector Primitive Scorers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Template class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hits Collection</td>
</tr>
</tbody>
</table>

- Each primitive scorer stores **one physics quantity** for each **physical volume** (accumulated over an event)
- Many scorers are provided by Geant4
  - energy deposit, flux, ...

Maria Grazia Pia
Basic strategy to retrieve information

Assume that you have already created the detector geometry
- Shape and size (Solid) of your detector, Material
- Logical volumes
- Physical volumes

Assign an instance of the Geant4 multifunctional detector (G4MultiFunctionalDetector) to the logical volume of your detector geometry set-up

Register instances of the required primitive scorers to your multifunctional detector

Finally, process the content of hit maps

Maria Grazia Pia
Sensitive Detector and Hits in detail
**Sensitive Detector**

- A user can provide his/her own implementation of the detector and its response

- To create a sensitive detector, derive your own concrete class from the `G4VSensitiveDetector` abstract base class
  - The principal purpose of the sensitive detector is to create hit Objects

- Overload the following methods:
  - `Initialize()`
  - `ProcessHits()` *(invoked for each step, if step starts in logical volume)*
  - `EndOfEvent()`
class G4VSensitiveDetector {  
  public:
  ...
  virtual void Initialize(G4HCofThisEvent*);
  virtual void EndOfEvent(G4HCofThisEvent*);
  ...
  virtual G4bool ProcessHits(G4Step* ,G4TouchableHistory*) = 0;
  ...
};

#include "G4VSensitiveDetector.hh"

class MySensitiveDetector : public G4VSensitiveDetector {
  public:
    MySensitiveDetector(G4String name);
    virtual ~MySensitiveDetector();
    virtual void Initialize(G4HCofThisEvent*HCE);
    virtual G4bool ProcessHits(G4Step* step,G4TouchableHistory* ROhist);
    virtual void EndOfEvent(G4HCofThisEvent*HCE);
  private:
    MyHitsCollection * hitsCollection;
    int collectionID;
Using information from particle steps, a sensitive detector either

- constructs one or more hits objects
  or
- accumulates values to existing hits
  - e.g. a tracker detector stores position, time of hit etc.
  - a calorimeter stores energy deposit etc.

Hit objects can be supplied information in the `ProcessHits` method of the concrete class of the SD

- This function has pointers to the current G4Step object and the G4TouchableHistory of the RO geometry (if defined) as arguments
Sensitive Detector - Constructor

- Specify a Hit Collection (by its unique name) for each type of hits considered in the sensitive detector
- Insert the name(s) in the collectionName vector

```cpp
class G4VSensitiveDetector {
    ...
    protected:
    G4CollectionNameVector collectionName;
    // This protected name vector must be filled in
    // the constructor of the concrete class for
    // registering names of hits collections
    ...
};
```

```cpp
MySensitiveDetector::MySensitiveDetector(G4String detectorUniqueName)
    : G4VSensitiveDetector(detectorUniqueName),
    collectionID(1)
{
    collectionName.insert("collection_name");
}
```
The Initialize() method is invoked at the beginning of each event

**Construct** all hits collections and insert them in the G4HCofThisEvent object, which is passed as argument to Initialize()

The **AddHitsCollection** method of G4HCofThisEvent requires the collection ID

The unique collection ID can be obtained with GetCollectionID():

- GetCollectionID() cannot be invoked in the constructor of this SD class
- it is required that the SD is instantiated and registered to the SD manager first

Hence, we defined a private data member (collectionID), which is set at the first call of the Initialize() function

```cpp
void MySensitiveDetector::Initialize(G4HCofThisEvent* HCE) {
  if (collectionID < 0)
    collectionID = GetCollectionID(0); // Argument : order of collection
    // as stored in the collectionName vector
  hitsCollection = new MyHitsCollection(sensitiveDetectorName, collectionName[0]);
  HCE -> AddHitsCollection(collectionID, hitsCollection);
}
```
This ProcessHits() method is invoked for every step in the volume(s) which hold a pointer to this SD.

The principal mandate of this method is to generate hit(s) or to accumulate data to existing hit objects, by using information from the current step.

- Note: Geometry information must be derived from the “PreStepPoint”.

```cpp
void MySensitiveDetector::ProcessHits(G4Step* step, G4TouchableHistory*ROhist) {
    MyHit* hit = new MyHit();
    ...
    // associate physics information, e.g.:
    G4double energyDeposit = step->GetTotalEnergyDeposit();
    hit->SetEnergyDeposit(energyDeposit); // See implementation of our Hit class
    ...
    hitsCollection->insert(hit);
    return true;
}
```
This `EndOfEvent()` method is invoked at the end of each event before the `EndOfEvent` function of the `G4UserEventAction` class.

```cpp
void MySensitiveDetector::EndOfEvent(G4HCofThisEvent* HCE) {
    ...
}
```
Hits Collection

Once created in the sensitive detector, instances of the concrete hit class must be stored in a dedicated collection
- Template class **G4THitsCollection**
- the template parameter is the concrete hit class: G4THitsCollection<MyHit>

Hit collections can be accessed in various phases of the simulation
- At the end of each event through the G4Event object
  - to analyse the event and store useful information
- During event processing through the sensitive detector manager, G4SDManager
  - to perform event filtering
HitCollections and Event

- A **G4Event** object holds a **G4HCofThisEvent** object at the end of event processing.

- You can get the pointer to the G4HCofThisEvent object by using the **GetHCofThisEvent** method of G4Event.

- The **G4HCofThisEvent** object stores all hits collections created within the event.

- At the end of an event, you can process the hits of hits collections contained in G4HCofThisEvent in your implementation of the **EndOfEventAction** method of the **user EventAction** class.
Hit information processing

Retrieve the **pointer to a hit collection** with the **GetHC** method of **G4HCofThisEvent** collection by using the collection index

- Index numbers of a hit collection are unique and do not change for a run
- The index number can be obtained by
  
  \[G4SDManager::GetCollectionID(“detName/colName”);\]

**NOTE**

- If hit collection(s) were not created in a particular event, the pointers to the collection are NULL; pointers should be checked before attempting to access hit objects
- Hits collection pointers are of **G4VHitsCollection type** in G4HCofThisEvent
- After retrieving them, it is required to **cast** them to the types of individual concrete classes
Hit information processing

- Loop over the entries of a hit collection to access individual hits

- You can use the `[ ]` operator of the hits collection class to get the pointer to a hit object (corresponding to the specified index)

- Retrieve the information associated with hits
  - and process the information, if required

- Store the output in analysis objects *(optional)*
  - Histograms, tuples etc.
  - NOTE: in the exercises we use AIDA objects to store the simulation output

- Store hits objects through a persistency mechanism for further processing *(optional)*

Maria Grazia Pia
Example of processing hits

```cpp
void MyEventAction::EndOfEventAction(const G4Event* event) {
    // index is a data member, representing the hits collection index of the considered collection
    // It was initialized to 1 in the class constructor
    if (index < 0) index = G4SDManager::GetSDMpointer() ->GetCollectionID("myDet/myColl");
    G4HCofThisEvent* HCE = event -> GetHCofThisEvent();

    MyHitsCollection* hitsCollection = 0;
    if (HCE) hitsCollection = (MyHitsCollection*)(HCE -> GetHC(index));  // cast

    if (hitsCollection) {
        int numberHits = hitsCollection -> entries();

        for (int j = 0; j < numberHits; j++) {
            MyHit* hit = (*hitsCollection)[j];

            // Retrieve information from hit object, e.g.
            double energy = hit -> GetEnergyDeposit();
            // Further process and store information…
            // Fill histograms etc.
        }
    }
}
```
Digis

- A Hit is created when a particle step is inside a detector, and typically is responsible for physical observables.
- A **Digit** represents the output of a detector.
  - e.g. ADC/TDC count, trigger signal, ...
- It is created from the information from hits and/or other digits.
- Similarly as for hits, you need to implement customized digit class(es) of your own.
- The base class of digits is **G4VDigi**.
- Digit objects are stored in digit collections.
  - Template class **G4TDigiCollection**.
- G4Event has a **GDCofThisEvent** object, that is a container for digit collections.
- *Note the similarity with Hits.*
Digitization

- Digits are created in a **digitizer module**

- Create your digitizer module by inheriting from the base class **G4VDigitizerModule**
  - Overload the pure virtual method **Digitize()**: Create digit objects in this function and store them in digit collections

**NOTE**: In contrast to the ProcessHits() function, the Digitize() method is **NOT automatically called** by the Geant4 kernel
  - You should invoke this function yourself explicitly
    - e.g. in the user action classes
Touchable

- Each G4StepPoint object has
  - Position in world coordinate system
  - Global and local time
  - Material
  - G4TouchableHistory for geometrical information

- G4TouchableHistory object provides information on the geometrical hierarchy
  - copy number
  - transformation / rotation to its mother