



Interaction with the Geant4 kernel – part 2

Luciano Pandola

INFN – Laboratori Nazionali del Sud

Partially based on a presentation by G.A.P. Cirrone (INFN-LNS)



Part I: Sensitive Detectors



Sensitive Detector (SD)

- A **logical volume** becomes **sensitive** if it has a pointer to a **sensitive detector** (**G4VSensitiveDetector**)
 - A sensitive detector can be instantiated **several times**, where the instances are assigned to **different logical volumes**
 - Note that 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 many functionalities)
- **Two possibilities** to make use of the SD functionality:
 - Create **your own sensitive detector** (using class inheritance)
 - Highly **customizable**
 - Use Geant4 **built-in tools**: Primitive scorers

- 

```
boxLogical->SetSensitiveDetector(mySensitive);
```

} assign to logical volume

```
(or)
SetSensitiveDetector( "LVname" ,mySensitive);
```

} assign to logical volume (alternative)

Part II: Native Geant4 scoring





Extract useful information

- Geant4 provides a number of **primitive scorers**, each one **accumulating one physics quantity** (e.g. total dose) for an event
- This is alternative to the **customized sensitive detectors** (see later in this lecture), which can be used with full flexibility to gain complete control
- It is **convenient** to use primitive scorers **instead** of **user-defined sensitive detectors** when:
 - you are not interested in recording each individual step, but **accumulating physical quantities** for an event or a run
 - you have **not too many** scorers



G4MultiFunctionalDetector

- **G4MultiFunctionalDetector** is a concrete class derived from **G4VSensitiveDetector**
- It should be **assigned to a logical volume** as a **kind of** (ready-for-the-use) **sensitive detector**
- It takes an **arbitrary number** of **G4VPrimitiveSensitivity** classes, to define the **scoring quantities** that you need
 - Each **G4VPrimitiveSensitivity** accumulates **one physics quantity** for each physical volume
 - E.g. **G4PSDoseScorer** (a concrete class of **G4VPrimitiveSensitivity** provided by Geant4) **accumulates dose** for each cell
- By using this approach, **no need to implement sensitive detector** and **hit classes**!



G4VPrimitiveSensitivity

- Primitive **scorers** (classes derived from **G4VPrimitiveSensitivity**) **have to be registered** to the **G4MultiFunctionalDetector**
 - **->RegisterPrimitive()**, **->RemovePrimitive()**
- They are designed to **score one kind of quantity** (surface flux, total dose) and to **generate one hit collection** per event
 - automatically named as **<MultiFunctionalDetectorName>/<PrimitiveScorerName>**
 - **hit collections** can be **retrieved** in the **EventAction** or **RunAction** (as those generated by sensitive detectors)
 - **do not share** the same **primitive scorer object** among **multiple G4MultiFunctionalDetector** objects (results may mix up!)

myCellScorer/TotalSurfFlux
myCellScorer/TotalDose

For example ...

```
MyDetectorConstruction::ConstructSDandField()
```

```
{
```

```
    G4MultiFunctionalDetector* myScorer = new  
    G4MultiFunctionalDetector("myCellScorer");
```

} instantiate multi-
functional detector

```
    myCellLog->SetSensitiveDetector(myScorer);
```

} attach to volume

```
    G4VPrimitiveSensitivity* totalSurfFlux = new  
        G4PSFlatSurfaceFlux("TotalSurfFlux");
```

```
    myScorer->RegisterPrimitive(totalSurfFlux);
```

} create a primitive
scorer (surface
flux) and register
it

```
    G4VPrimitiveSensitivity* totalDose = new  
        G4PSDoseDeposit("TotalDose");
```

```
    myScorer->RegisterPrimitive(totalDose);
```

} create a primitive
scorer (total dose)
and register it

```
}
```

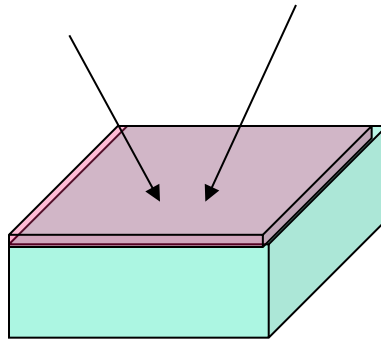


Some primitive scorers that you may find useful

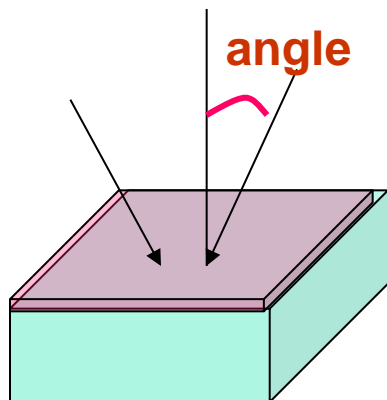
- Concrete Primitive Scorers (→ Application Developers Guide 4.4.5)
 - Track length
 - G4PSTrackLength, G4PSPassageTrackLength
 - Deposited energy
 - G4PSEnergyDeposit, G4PSDoseDeposit
 - Current/Flux
 - G4PSFlatSurfaceCurrent, G4PSSphereSurfaceCurrent, G4PSPassageCurrent, G4PSFlatSurfaceFlux, G4PSCellFlux, G4PSPassageCellFlux
 - Others
 - G4PSMinKinEAtGeneration, G4PSNofSecondary, G4PSNofStep, G4PSCellCharge

A closer look at some scorers

SurfaceCurrent :
Count number of
injecting particles
at defined surface.

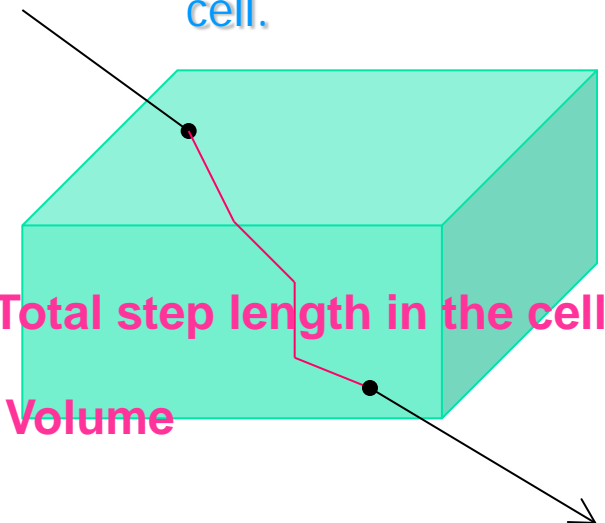


CellFlux :
Sum of L / V of
injecting particles
in the geometrical
cell.



SurfaceFlux :
Sum up
 $1/\cos(\text{angle})$ of
injecting particles
at defined surface

L : Total step length in the cell
 V : Volume



V : Volume



G4VSDFilter

- A **G4VSDFilter** can be **attached** to **G4VPrimitiveSensitivity** to define **which kind of tracks** have to **be scored** (e.g. one wants to know surface flux of **protons only**)
 - **G4SDChargeFilter** (accepts only **charged** particles)
 - **G4SDNeutralFilter** (accepts only **neutral** particles)
 - **G4SDKineticEnergyFilter** (accepts tracks in a defined range of **kinetic energy**)
 - **G4SDParticleFilter** (accepts tracks of a **given particle type**)
 - **G4VSDFilter** (base class to create user-customized filters)



For example ...

```
MyDetectorConstruction::ConstructSDandField()
```

```
{  
    G4VPrimitiveSensitivity* protonSurfFlux  
    = new G4PSFlatSurfaceFlux("pSurfFlux");  
    G4VSDFilter* protonFilter = new  
        G4SDParticleFilter("protonFilter");  
    protonFilter->Add("proton");  
  
    protonSurfFlux->SetFilter(protonFilter);  
  
    myScorer->RegisterPrimitive(protonSurfFlux);  
}
```

} create a primitive
scorer (**surface
flux**), as before

} create a **particle
filter** and add
protons to it

} **register** the **filter**
to the primitive
scorer

register the **scorer** to the
multifunc detector (as
shown before)

How to retrieve information - part 1



- At the end of the day, one wants to **retrieve** the information from the scorers
 - **True** also for the **customized** hits collection
- Each scorer **creates a hit collection**, which is **attached** to the **G4Event** object
 - Can be retrieved and read at the **end of the event**, using an integer ID
 - Hits collections mapped as **G4HitsMap<G4double>*** so can loop on the individual entries
 - **Operator += provided** which automatically sums up hits (no need to loop)

How to retrieve information – part 2

```
//needed only once
G4int collID = G4SDManager::GetSDMpointer()
    ->GetCollectionID("myCellScorer/TotalSurfFlux");

G4HCofThisEvent* HCE = event->GetHCofThisEvent();

G4THitsMap<G4double>* evtMap =
    static_cast<G4THitsMap<G4double>*>
    (HCE->GetHC(collID));

std::map<G4int,G4double*>::iterator itr;
for (itr = evtMap->GetMap()->begin(); itr !=
    evtMap->GetMap()->end(); itr++) {
    G4double flux = *(itr->second);
    G4int copyNb = *(itr->first);
}
```

Get **ID** for the collection (given the name)

Get **all HC** available in this event

Get the HC with the **given ID** (need a cast)

Loop over the **individual entries** of the HC: the key of the map is the copyNb, the other field is the real content



Command-based scoring

Thanks to the newly developed **parallel navigation**, an **arbitrary scoring mesh geometry** can be defined which is **independent to the volumes** in the mass geometry. Also, G4MultiFunctionalDetector and primitive scorer classes now offer the **built-in scoring** of most-common quantities

UI **commands** for scoring → no C++ required, apart from instantiating G4ScoringManager in main()

- Define a scoring mesh
 - /score/create/boxMesh <mesh_name>
 - /score/open, /score/close
- Define mesh parameters
 - /score/mesh/boxsize <dx> <dy> <dz>
 - /score/mesh/nbin <nx> <ny> <nz>
 - /score/mesh/translate,
- Define primitive scorers
 - /score/quantity/eDep <scorer_name>
 - /score/quantity/cellFlux <scorer_name>
 - currently **20 scorers** are available
- Define filters
 - /score/filter/particle <filter_name>
 - <particle_list>
 - /score/filter/kinE <filter_name>
 - <Emin> <Emax> <unit>
 - currently **5 filters** are available
- Output
 - /score/draw <mesh_name>
 - <scorer_name>
 - /score/dump, /score/list



How to learn more about built-in scoring

Have a look at the **dedicated extended examples** released with Geant4:

[examples/extended/runAndEvent/RE02](#)
(use of primitive scorers)

[examples/extended/runAndEvent/RE03](#)
(use of UI-based scoring)

Part III: User-defined sensitive detectors: Hits and Hits Collection





The ingredients of user SD

- A **powerful** and **flexible** way of extracting information from the physics simulation is to **define your own SD**
- Derive **your own concrete classes** from the base classes and **customize** them according to your needs

	Concrete class	Base class
Sensitive Detector	MySensitiveDetector	G4VSensitiveDetector
Hit	MyHit	G4VHit
		Template class
Hits collection		G4THitsCollection<MyHit* >



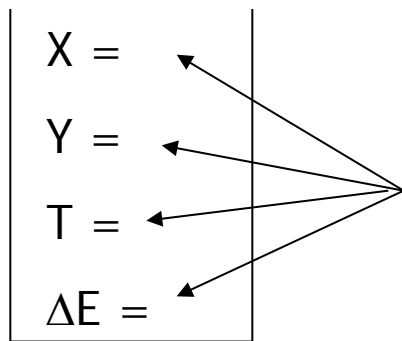
Hit class - 1

- Hit is a **user-defined class** which derives from the base class **G4VHit**. Two **virtual methods**
 - Draw()
 - Print()
- You can **store various types of information** by implementing your own concrete Hit class
- Typically, one may want to record information like
 - Position, time and ΔE of a step
 - Momentum, energy, position, volume, particle type of a given track
 - Etc.



Hit class - 2

A “Hit” is like a “**container**”, a **empty box** which will store the **information** retrieved step by step



The Hit **concrete class** (derived by **G4VHit**) **must** be **written by the user**: the user must decide **which variables and/or information** the hit should **store** and **when** store them

The Hit objects are **created** and **filled** by the **SensitiveDetector** class (invoked at each step in **detectors defined as sensitive**). **Stored** in the “**HitCollection**”, attached to the **G4Event**: can be **retrieved** at the end of the event



Hit class - 3

Example

// header file: MyHit.hh

#include "G4VHit.hh"

class **MyHit** : **public** **G4VHit** {

public:

MyHit();

virtual ~MyHit();

...

public methods to
handle data member

inline void SetEnergyDeposit(G4double energy) { **energyDeposit** = energy; }

inline G4double GetEnergyDeposit() { **return** **energyDeposit**; }

... // more get and set methods

private:

G4double **energyDeposit**;

... // more data members

};



data member (private)



Geant4 Hits

Since in the simulation one may have **different sensitive detectors** in the same setup (e.g. a calorimeter and a Si detector), it is possible to define **many Hit classes** (all derived by **G4VHit**) storing **different information**

X =
Y =
T =
ΔE =

Class Hit1 :
public G4VHit

Z =
Pos =
Dir =

Class Hit2 :
public G4VHit



Hits Collection - 1

At each step in a **detector** defined as **sensitive**, the method **ProcessHit()** of the user **SensitiveDetector** class is invoked: it must **create**, **fill** and **store** the Hit objects

X = 1
Y = 2
T = 3
$\Delta E = 1$

Step 1

X = 2
Y = 0
T = 3.1
$\Delta E = 2$

Step 2

X = 3
Y = 2
T = 4
$\Delta E = 3$

Step 3

...

X = 3
Y = 2
T = 6
$\Delta E = 1$

Step N

Hits collection (= vector<Hit>)



Hits Collection - 2

- Once created in the sensitive detectors, objects of the concrete hit class **must be stored** in a **dedicated collection**
 - **Template class** `G4THitsCollection<MyHit>`, which is actually an **array of `MyHit*`**
- The hits collections can be accesses in **different phases** of tracking
 - At the **end of each event**, through the `G4Event` (*a-posteriori event analysis*)
 - During **event processing**, through the Sensitive Detector Manager `G4SDManager` (*event filtering*)



The HCofThisEvent

Remember that you may have **many kinds of Hits**
(and Hits Collections)

X = 1
Y = 2
T = 3
 $\Delta E = 1$

X = 2
Y = 0
T = 3.1
 $\Delta E = 2$

X = 3
Y = 2
T = 4
 $\Delta E = 3$

.....

X = 3
Y = 2
T = 6
 $\Delta E = 1$

Z = 5
Pos =
(0,1,1)
Dir
=(0,1,0)

Z = 5.2
Pos =
(0,0,1)
Dir
=(1,1,0)

.....

Z = 5.4
Pos =
(0,1,2)
Dir
=(0,1,1)

HCofThisEvent

Attached to
G4Event*



Hits Collections of an event

- A **G4Event** object has a **G4HCofThisEvent** object at the end of the event processing (if it was successful)
 - The **pointer** to the **G4HCofThisEvent** object can be retrieved using the **G4Event::GetHCofThisEvent()** method
- The **G4HCofThisEvent** stores all **hits collections** created within the event
 - Hits collections are **accessible** and can be **processed** e.g. in the **EndOfEventAction()** method of the User **Event Action** class



SD and Hits

- Using **information** from **particle steps**, a sensitive detector either
 - **constructs**, **fills** and **stores** one (or more) **hit object**
 - **accumulates** values to existing hits
- Hits objects can be **filled** with information in the **ProcessHits()** method of the SD concrete user class → **next slides**
 - This method has **pointers** to the current **G4Step** and to the **G4TouchableHistory** of the ReadOut **geometry** (if defined)



Sensitive Detector (SD)

- A specific feature to Geant4 is that a user can provide his/her **own implementation** of the **detector** and **its response** → **customized**
- 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** (see also next slide):
 - **Initialize()**
 - **ProcessHits()** (Invoked for **each step** if step starts in logical volume having the SD attached)
 - **EndOfEvent()**



Sensitive Detector

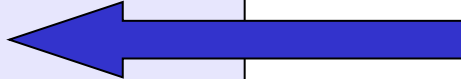
```
class G4VSensitiveDetector {  
  public:                                     abstract base class  
    ...  
    virtual void Initialize(G4HCofThisEvent*);  
    virtual void EndOfEvent(G4HCofThisEvent*);  
  protected:  
    virtual G4bool ProcessHits(G4Step* ,  
                                G4TouchableHistory*) = 0;  
    ...  
};
```

pure virtual method →

```
// header file: MySensitiveDetector.hh  
#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;  
    G4int collectionID;  
};
```

User
concrete
SD class





SD implementation: constructor

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

```
MySensitiveDetector::MySensitiveDetector(G4String detectorUniqueName)  
    : G4VSensitiveDetector(detectorUniqueName),  
      collectionID(-1) {  
  
    collectionName.insert("collection_name");  
}
```

Base class



```
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  
    ...  
};
```



SD implementation: Initialize()

- 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

```
void MySensitiveDetector::Initialize(G4HCofThisEvent*HCE) {  
    if(collectionID < 0)  
        collectionID = GetCollectionID(0); // Argument : order of collect.  
                                           // as stored in the collectionName  
    hitsCollection = new MyHitsCollection  
        (SensitiveDetectorName, collectionName[0]);  
  
    HCE -> AddHitsCollection(collectionID, hitsCollection);  
}
```



SD implementation: ProcessHits()

- This **ProcessHits()** method is invoked for **every step** in the volume(s) which **hold a pointer to this SD** (= each volume defined as "**sensitive**")
- The **main 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**"

```
G4bool MySensitiveDetector::ProcessHits(G4Step* step,
                                         G4TouchableHistory* ROhist) {
    MyHit* hit = new MyHit();    // 1) create hit
    ...
    // some set methods, e.g. for a tracking detector:
    G4double energyDeposit = step -> GetTotalEnergyDeposit(); // 2) fill hit
    hit -> SetEnergyDeposit(energyDeposit); // See implement. of our Hit class
    ...
    hitsCollection -> insert(aHit); // 3) insert in the collection
    return true;
}
```



SD implementation: EndOfEvent()

- This EndOfEvent () method is invoked at the end of each event.
 - Note is invoked **before** the EndOfEvent function of the **G4UserEventAction** class

```
void MySensitiveDetector::EndOfEvent(G4HCofThisEvent* HCE) {  
}
```



Processing hit information - 1

- Retrieve the pointer of a hits collection with the **GetHC()** method of `G4HCofThisEvent` collection using the **collection index** (a `G4int` number)
- **Index** numbers of a hit collection are **unique** and don't change for a run. The number **can be obtained** by `G4SDManager::GetCollectionID("name")`;
- Notes:
 - if the collection(s) are **not created**, the pointers of the collection(s) are NULL: **check** before trying to access it
 - Need an **explicit cast** from `G4VHitsCollection` (see code)



Processing hit information - 2

- **Loop** through the entries of a hits collection to **access individual hits**
 - Since the HitsCollection is a vector, you can use the **[] operator** to get the hit object corresponding to a **given index**
- **Retrieve** the information **contained in this hit** (e.g. using the **Get/Set methods** of the concrete user Hit class) and **process it**
- **Store** the output in analysis objects



Process hit: example

```
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* hitsColl = 0;  
    if(HCE) hitsColl = (MyHitsCollection*)(HCE->GetHC(index));  
    if(hitsColl) {  
        int numberHits = hitsColl->entries();  
        for(int i1= 0; i1 < numberHits ; i1++) {  
            MyHit* hit = (*hitsColl)[i1];  
            // Retrieve information from hit object, e.g.  
            G4double energy = hit -> GetEnergyDeposit;  
            ... // Further process and store information  
        }  
    }  
}
```

retrieve index

retrieve all hits collections

retrieve hits collection by index

cast

loop over individual hits, retrieve the data



The HCofThisEvent

Remember that you may have **many kinds of Hits**
(and Hits Collections)

X = 1
Y = 2
T = 3
 $\Delta E = 1$

X = 2
Y = 0
T = 3.1
 $\Delta E = 2$

X = 3
Y = 2
T = 4
 $\Delta E = 3$

.....

X = 3
Y = 2
T = 6
 $\Delta E = 1$

Z = 5
Pos =
(0,1,1)
Dir
=(0,1,0)

Z = 5.2
Pos =
(0,0,1)
Dir
=(1,1,0)

.....

Z = 5.4
Pos =
(0,1,2)
Dir
=(0,1,1)

HCofThisEvent

Attached to
G4Event*



Recipe and strategy - 1

- Create your **detector geometry**
 - Solids, logical volumes, physical volumes
- Implement a **sensitive detector** and assign an instance of it to the ***logical volume*** of your geometry set-up
 - Then this volume becomes “**sensitive**”
 - Sensitive detectors are **active for each particle steps**, if the step starts in this volume



Recipe and strategy - 2

- Create **hits objects** in your sensitive detector using information from the particle step
 - You need to **create the hit class(es)** according to **your requirements**
- **Store** hits in hits collections (automatically associated to the **G4Event** object)
- Finally, **process the information** contained in the hit in user action classes (e.g. **G4UserEventAction**) to obtain **results** to be stored in the analysis object