



# Sensitive Detectors

Varisano Annagrazia

LNS-INFN

**Geant4 simulation code: theory and practical session**

X Seminar on Software for Nuclear, Subnuclear and Applied Physics

# Goal

- Learn how to retrieve information from the simulation:
  - Geant4 simulates the interactions of a particle with matter following its trajectory
  - There are several ways to get information from the interactions:
    - Develop a “**Sensitive Detector**” (*DS*), i.e. volume able to generate hits. The information can be retrieved by User hooks Classes.(G4UserEventAction, G4UserRunAction, G4UserSteppingAction,etc)
    - Use “**Primitive Scorer**” (*PS*): provided by Geant4. In fact, it provides a number of *PS* each of which accumulates a physical quantity for each event.
- Learn how to write information retrieved from the simulation on an external file, like ASCII files or ROOT files

# Part I: Sensitive Detectors (SD)

- ▶ A logical volume becomes sensitive if it has a pointer to a **sensitive detector (G4VSensitiveDetector)**
- ▶ To add sensitivity to a logical volume we must:
  1. Create an instance of a sensitive detector
  2. Register the sensitive detector to the SD manager
  3. Assign the pointer of your SD to the logical volume of your detector geometry

1. create instance

```
G4VSolid* boxSolid = new G4Box( "aBoxSolid", 1.0 * cm, 1.0 * cm, 1.0 * cm);
```

```
G4LogicalVolume* boxLog =  
    new G4LogicalVolume( boxSolid, matSilicon, "aBoxLog", 0, 0, 0);
```

```
G4VSensitiveDetector* sensitiveBox =  
    new MySensitiveDetector("/MyDetector");
```

```
G4SDManager* SDManager = G4SDManager::GetSDMPointer();
```

2. register to SD manager

```
SDManager -> AddNewDetector(sensitiveBox);
```

3. assign to logical volume

```
boxLog -> SetSensitiveDetector(sensitiveBox);
```

# Part I: Sensitive Detectors (SD)

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)

```
G4VSolid* boxSolid = new G4Box( "aBoxSolid", 1.0 * cm, 1.0 * cm, 1.0 * cm);
```

```
G4LogicalVolume* boxLog =  
    new G4LogicalVolume( boxSolid, matSilicon, "aBoxLog", 0, 0, 0);
```

**create instance**



```
G4VSensitiveDetector* sensitiveBox =  
    new MySensitiveDetector("/MyDetector");
```

**register to SD manager**



```
G4SDManager* SDManager = G4SDManager::GetSDMPointer();
```

```
SDManager -> AddNewDetector(sensitiveBox);
```

**assign to logical volume**



```
boxLog -> SetSensitiveDetector(sensitiveBox);
```

# Part II: User-defined sensitive detectors, Hits and Hits Collection

- ▶ 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



inheritance ↑

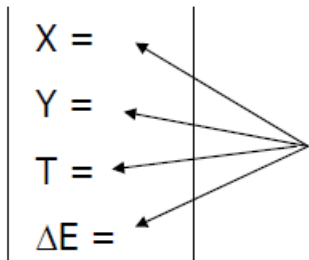


inheritance ↑



# Hit class - 1

- ▶ A hit is a snapshot of a physical interaction of a track in the sensitive region of the detector
- ▶ A “Hit” is like a “container”, a **empty box** which contains 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 (Typically: Momentum, Energy, Position, Volume, Particle type of a given track)

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

# Hit class - 1

- ▶ A h  
ser
- ▶ A “  
inf

In the simulation we can have many different sensitive detectors in the same setup (e.g. a calorimeter and a Si detector), for this reason, it is possible to define **many Hit classes** (all derived by **G4VHit**) storing different information

X =  
Y =  
T =  
 $\Delta E =$

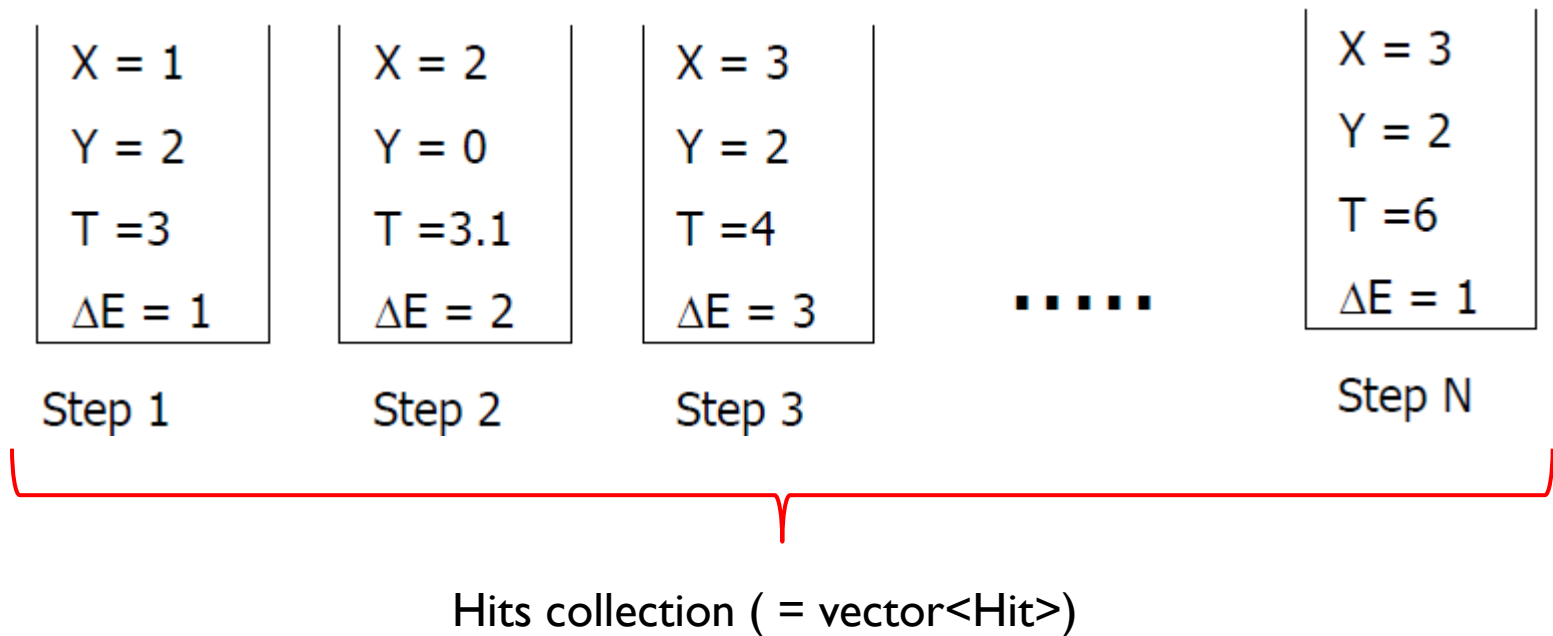
```
Class Hit1 :  
public G4VHit  
  
X =  
Y =  
T =  
 $\Delta E =$ 
```

```
Class Hit2 :  
public G4VHit  
  
Z =  
Pos =  
Dir =
```

- ▶ 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 EndOfEvent

# Hit Collection- 1

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



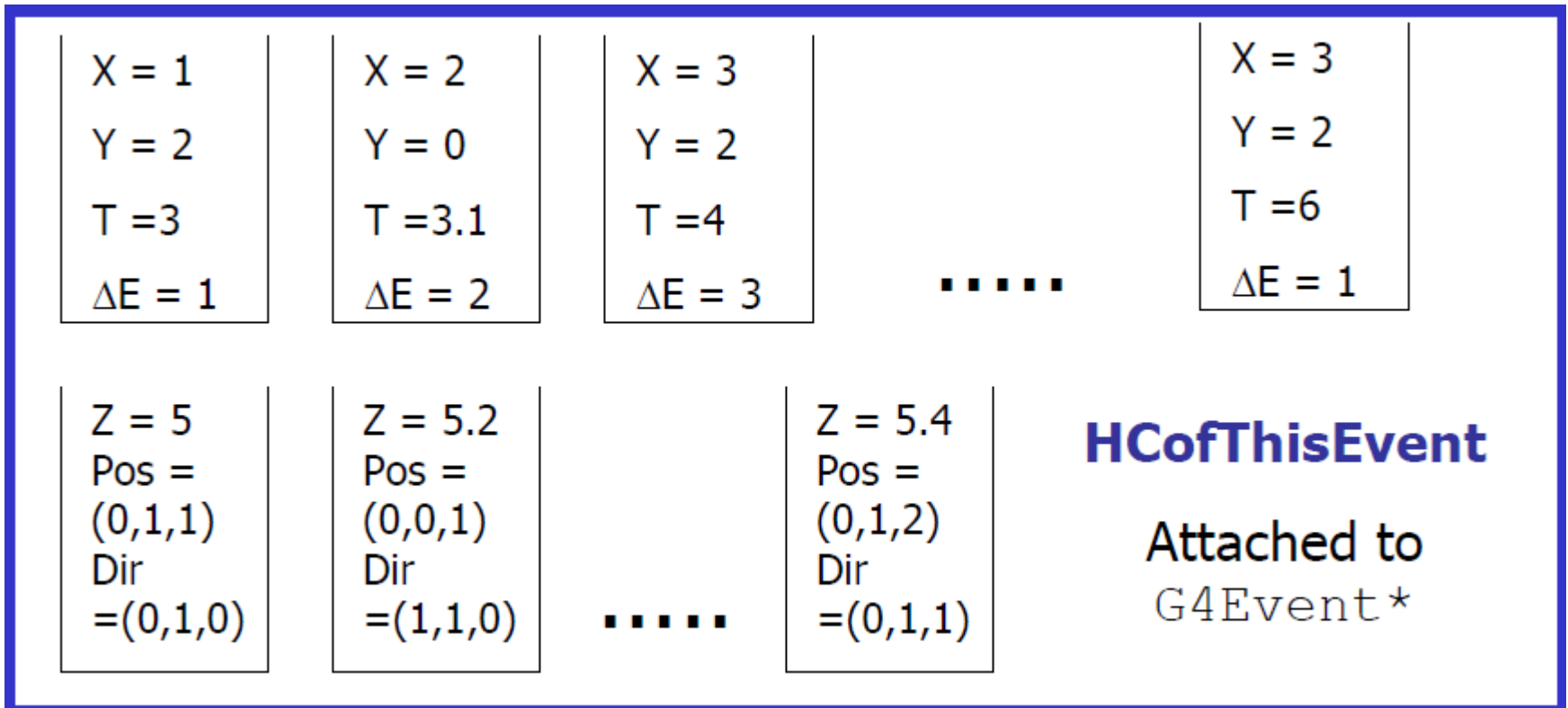


# Hit Collection- 2

- ▶ 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 accessed 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)

# Hit Collection- 3

► many kinds of Hits (and Hits Collections)



# Hit Collection- 3

## ► many kinds of Hits (and Hits Collections)

- 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
- 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

X = 3

Y = 2

T = 6

$\Delta E = 1$

**HCoThisEvent**

Attached to  
G4Event\*

# SD and Hits

- ▶ The principal goal of the sensitive detector is to manage and create hit objects through these three virtual methods (see also next slide)
  - ▶ **Initialize()**
  - ▶ **ProcessHits()** (Invoked for each step if step starts in logical volume having the SD attached)
- Using information from particle steps, a sensitive detector either
  - constructs, fills and stores one (or more) **hit object**
  - accumulates values to existing hit
- ▶ **EndOfEvent()**

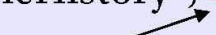
# Sensitive Detector (SD)

Base SD class



```
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"
```

*your SD class*

```
...  
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 SD 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).
  - 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 principal task of this method is to **generate hit(s)** or to accumulate data to existing hit objects, by using information from the current step.

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



## SD implementation: *EndOfEvent()*

- This EndOfEvent() method is invoked at the end of each event.

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

# Processing hit information - 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"); } retrieve  
                                                index  
  
    G4HCofThisEvent* HCE = event-> GetHCofThisEvent(); } retrieve all hits  
                                                collections  
  
    MyHitsCollection* hitsColl = 0;  
    if(HCE) hitsColl = (MyHitsCollection*)(HCE->GetHC(index)); } retrieve hits  
                                                collection by index  
  
    if(hitsColl) {  
        int numberHits = hitsColl->entries();  
    }  
}
```

*cast*

- 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 - example

```
void MyEventAction::EndOfEventAction(const G4Event* event) {
```

```
// index is a data member representing the hits collection index of the
```

- 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.

```
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  
    }  
}
```

*cast*

**loop over  
individual hits,  
retrieve the data**

# Strategy

1. Create your detector geometry
  - ▶ Solids, logical volumes, physical volumes
2. 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
3. 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**
  - ▶ Use Touchable of the read-out geometry to retrieve geometrical info associated with this
4. **Store** hits in hits collections (automatically associated to the G4Event object)
5. 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

# Native Geant4 scoring

- Alternatively to user-defined sensitive detectors, primitive scorers provided by Geant4 can be used
- Geant4 provides a number of **primitive scorers**, each one accumulating one physics quantity (e.g. total dose) for an event
- 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
- ▶ 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 score object among multiple G4MultiFunctionalDetector objects (results may mix up!)

# Example

```
MyDetectorConstruction::Construct()
```

```
{ ...
```

```
G4LogicalVolume* myCellLog = new G4LogicalVolume(...);
```

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

```
G4SDManager::GetSDMpointer() -> AddNewDetector(myScorer);
```

```
myCellLog->SetSensitiveDetector(myScorer); } Attach to volume
```

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

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

```
G4VPrimitiveSensitivity* totalDose = new
```

```
G4PSDoseDeposit("TotalDose");
```

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

```
}
```

Instantiate  
multifunctional  
detector  
and register in the  
SD manager

create a primitive scorer  
(surface flux) and register it

create a primitive scorer (total  
dose) and register it



# Some primitive scorers that you may find useful

- Concrete Primitive Scorers (-> Application Developers Guide 4.4.6)

- Track length

- G4PSTrackLength, G4PSPassageTrackLength

- Deposited energy

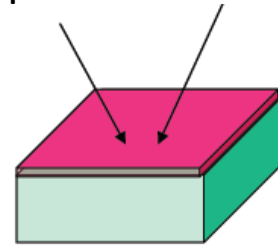
- G4PSEnergyDeposit, G4PSDoseDeposit

- Current/Flux

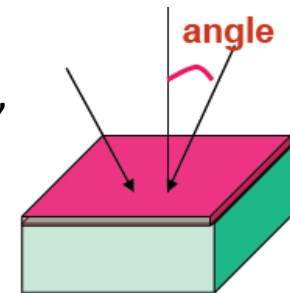
- G4PSFlatSurfaceCurrent, G4PSSphereSurfaceCurrent, G4PSPassageCurrent, G4PSFlatSurfaceFlux, G4PSCellFlux, G4PSPassageCellFlux

- Others

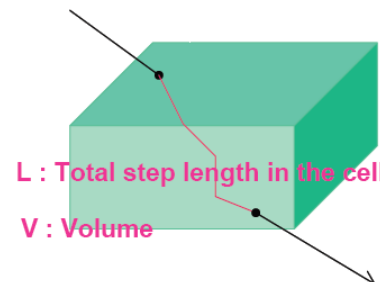
- G4PSMinKinEAtGeneration, G4PSNofSecondary, G4PSNofStep, G4PSCellCharge



**SurfaceCurrent :**  
Count number of injecting particles at defined surface.



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



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

# 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)

# Example

- MyDetectorConstruction::Construct()

```
{
```

```
G4VPrimitiveSensitivity* protonSurfFlux = new  
G4PSFlatSurfaceFlux("pSurfFlux");
```

create a primitive scorer  
(surface flux), as before

```
G4VSDFilter* protonFilter = new  
G4SDParticleFilter("protonFilter");
```

create a particle filter and  
Add protons to it

```
protonFilter->Add("proton");
```

```
protonSurfFlux->SetFilter(protonFilter);
```

register the filter to the primitive  
scorer

```
myScorer->Register(protonSurfFlux);
```

```
}
```

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

# UI commands for scoring

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

```
#include "G4ScoringManager.hh"
int main()
{
    G4RunManager* runManager = new G4RunManager;
    G4ScoringManager* scoringManager = G4ScoringManager::GetScoringManager();
    ...
}
```

- All of the UI commands of this functionality is in **/score/** directory.

- **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
```

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)

# Define a **scoring mesh**

- To **define** a scoring mesh, the user has to specify the followings.
  - **Shape and name** of the 3D scoring mesh. Currently, **box** is the only available shape.
  - **Size** of the scoring mesh. Mesh size must be specified as "half width" similar to the arguments of G4Box.
  - **Number of bins for each axes**. Note that too many bins cause immense memory consumption.

```
# define scoring mesh
/score/create/boxMesh boxMesh_1
/score/mesh/boxSize 100. 100. 100. cm
/score/mesh/nBin 30 30 30
```

Optionally, position and rotation of the mesh. If not specified, the mesh is positioned at the center of the world volume without rotation.

```
# Translation/Rotation of scoring mesh
/score/translate 0. 0. 10. cm
/score/mesh/rotate/rotateZ 45. deg
```

The mesh geometry can be **completely independent** from the real material geometry.

# Scoring quantities

- A mesh may have **arbitrary number of scorers**. Each scorer scores **one** physics quantity (xxxxx).
  - energyDeposit \* **Energy** deposit scorer.
  - cellCharge \* Cell charge scorer.
  - cellFlux \* **Cell flux** scorer.
  - passageCellFlux \* Passage cell flux scorer
  - doseDeposit \* **Dose deposit** scorer.
  - nOfStep \* Number of step scorer.
  - nOfSecondary \* Number of secondary scorer.
  - trackLength \* Track length scorer.
  - passageCellCurrent \* Passage cell current scorer.
  - passageTrackLength \* Passage track length scorer.
  - flatSurfaceCurrent \* Flat surface current Scorer.
  - flatSurfaceFlux \* Flat surface flux scorer.
  - nOfCollision \* Number of collision scorer.
  - population \* Population scorer.
  - nOfTrack \* Number of track scorer.
  - nOfTerminatedTrack \* Number of terminated tracks scorer.

`/score/quantity/xxxxx <scorer_name>`

# Filter

- Each scorer may take a **filter**.
  - charged \* Charged particle filter.
  - neutral \* Neutral particle filter.
  - kineticEnergy \* Kinetic energy filter.  
/score/filter/kineticEnergy <fname> <eLow> <eHigh> <unit>
  - particle \* Particle filter.  
/score/filter/particle <fname> <p1> ... <pn>
  - particleWithKineticEnergy \* Particle with kinetic energy filter.

```
/score/quantity/energyDeposit    eDep
/score/quantity/nOfStep         nOfStepGamma
/score/filter/particle          gammaFilter    gamma
/score/quantity/nOfStep         nOfStepEMinus
/score/filter/particle          eMinusFilter    e-
/score/quantity/nOfStep         nOfStepEPlus
/score/filter/particle          ePlusFilter    e+
/score/close
```

Same primitive scorers  
with different filters  
may be defined.



Close the mesh when defining scorers is done.

# Write scores to a file

- Single score

```
/score/dumpQuantityToFile <mesh_name> <scorer_name> <file_name>
```

- All scores

```
/score/dumpAllQuantitiesToFile <mesh_name> <file_name>
```

- By default, values are written in CSV.
- By creating a concrete class derived from **G4VScoreWriter** base class, the user can define his own file format.
  - Example in [/examples/extended/runAndEvent/RE03](#)
  - User's score writer class should be registered to G4ScoringManager.

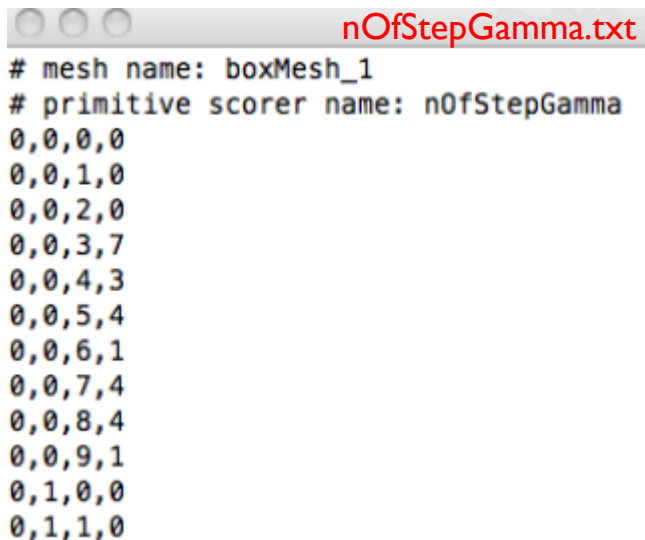


# Example of file output in command-based scores

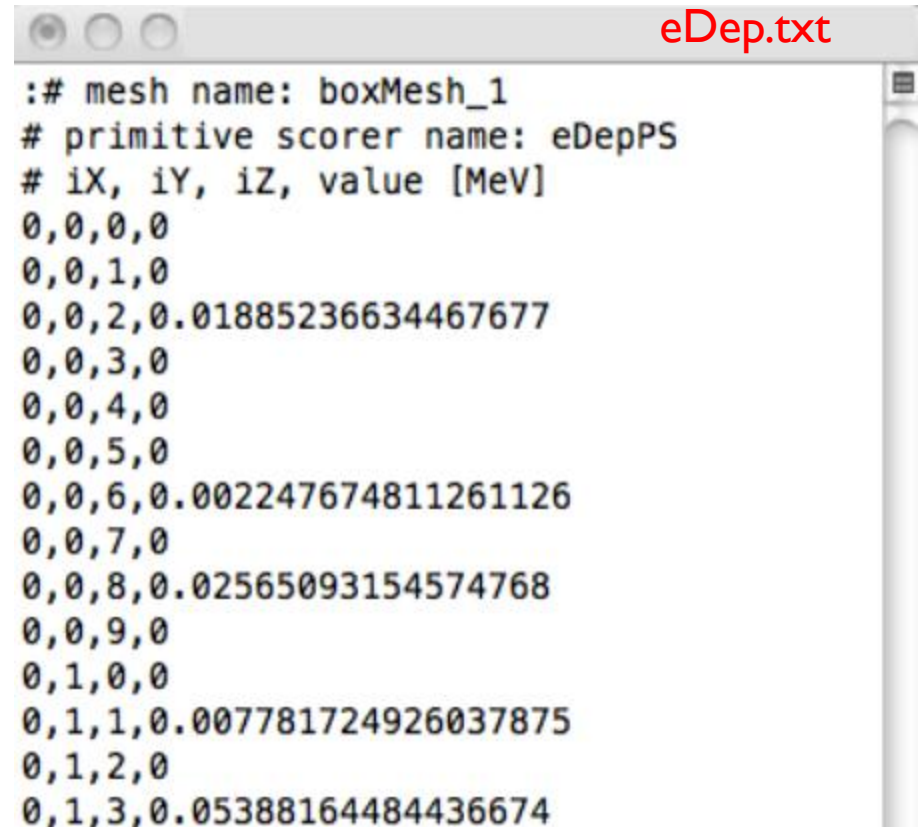
# Dump scores to a file

/score/dumpQuantityToFile boxMesh\_1 nOfStepGamma nOfStepGamma.txt

/score/dumpQuantityToFile boxMesh\_1 eDep eDep.txt



```
# mesh name: boxMesh_1
# primitive scorer name: nOfStepGamma
0,0,0,0
0,0,1,0
0,0,2,0
0,0,3,7
0,0,4,3
0,0,5,4
0,0,6,1
0,0,7,4
0,0,8,4
0,0,9,1
0,1,0,0
0,1,1,0
```



```
:# mesh name: boxMesh_1
# primitive scorer name: eDepPS
# iX, iY, iZ, value [MeV]
0,0,0,0
0,0,1,0
0,0,2,0.01885236634467677
0,0,3,0
0,0,4,0
0,0,5,0
0,0,6,0.002247674811261126
0,0,7,0
0,0,8,0.02565093154574768
0,0,9,0
0,1,0,0
0,1,1,0.007781724926037875
0,1,2,0
0,1,3,0.05388164484436674
```

# Writing information on an external file

For a long time, Geant4 did not provide any native data analysis tool. As a general rule, the user was supposed provide his own code to output results to an appropriate analysis format and to use an external analysis tool.

In the latest geant4 releases, a few basic classes for data analysis have been implemented:

- Support for histograms and ntuples
- Output in ROOT, XML and CSV (ASCII)
- **Simplest using text (ASCII) files (human-readable)**
  - Simplest possible approach = comma-separated values (.csv files)
  - The resulting files can be analyzed by tools such as: Gnuplot, Excel, OpenOffice, Matlab, Origin ..
- **Advanced using ntuple files**
  - Allows to control what plot you want with modular choice of conditions and variables
    - Ex: energy of electrons knowing that (= cuts): (1) position/location, (2) angular window, (3) primary/secondary ...
  - Tools: Root

# Output stream (G4cout)

- G4cout is a iostream object defined by Geant4. The usage of this object is exactly the same as the ordinary `std::cout` except that the output streams will be handled by G4UImanager.
- Output strings may be displayed on another window or stored in a file.

Example:

```
void SteppingAction::UserSteppingAction(const G4Step* aStep)
{

    evtNb = eventAction -> Trasporto();

    G4String particleName = aStep -> GetTrack() -> GetDynamicParticle() -> GetDefinition() -> GetParticleName();
    G4String volumeName = aStep -> GetPreStepPoint() -> GetPhysicalVolume() -> GetName();
    G4double particleCharge = aStep -> GetTrack() -> GetDefinition() -> GetAtomicNumber();
    G4double PDG=aStep->GetTrack()->GetDefinition()->GetAtomicMass();

    G4Track* theTrack = aStep->GetTrack();
    G4double kineticEnergy = theTrack -> GetKineticEnergy();
    G4int trackID = aStep -> GetTrack() -> GetTrackID();
    G4double edep = aStep->GetTotalEnergyDeposit();
    G4String materialName = theTrack->GetMaterial()->GetName();
```

```
G4cout    << "Energy deposited--->" << " " << edep << " "
           << "Charge--->" << " " << particleCharge << " "
           << "Kinetic Energy --->" << " " << kineticEnergy << " "
           << G4endl;
```

# Output stream (G4cout)

```
G4cout  << "Energy deposited--->" << " " << edep << " "  
        << "Charge--->" << " " << particleCharge << " "  
        << "Kinetic Energy --->" << " " << kineticEnergy << " "  
        << G4endl;
```

```
---> Begin of Event: 0  
Energia depositata---> 9.85941e-22 Carica---> 6 Energia Cinetica---> 160  
Energia depositata---> 8.36876 Carica---> 6 Energia Cinetica---> 151.631  
Energia depositata---> 8.63368 Carica---> 6 Energia Cinetica---> 142.998  
Energia depositata---> 5.98509 Carica---> 6 Energia Cinetica---> 137.012  
Energia depositata---> 4.73055 Carica---> 6 Energia Cinetica---> 132.282  
Energia depositata---> 0.0225575 Carica---> 6 Energia Cinetica---> 132.259  
Energia depositata---> 1.47468 Carica---> 6 Energia Cinetica---> 130.785  
Energia depositata---> 0.0218983 Carica---> 6 Energia Cinetica---> 130.763  
Energia depositata---> 5.22223 Carica---> 6 Energia Cinetica---> 125.541  
Energia depositata---> 7.10685 Carica---> 6 Energia Cinetica---> 118.434  
Energia depositata---> 6.62999 Carica---> 6 Energia Cinetica---> 111.804  
Energia depositata---> 6.50997 Carica---> 6 Energia Cinetica---> 105.294  
Energia depositata---> 6.28403 Carica---> 6 Energia Cinetica---> 99.0097  
Energia depositata---> 5.77231 Carica---> 6 Energia Cinetica---> 93.2374  
Energia depositata---> 5.2333 Carica---> 6 Energia Cinetica---> 88.0041  
Energia depositata---> 3.9153 Carica---> 6 Energia Cinetica---> 84.0888  
Energia depositata---> 14.3767 Carica---> 6 Energia Cinetica---> 69.7121  
Energia depositata---> 14.3352 Carica---> 6 Energia Cinetica---> 55.3769
```

# ASCII file saving

In order to allow an object to save data to a file you must:

- Add to the include list the <fstream> header file
- Put into the class declaration (file .hh) an ofstream object:

```
std::ofstream myFile;
```

- Open the file, in the class constructor, or into a specific method:

```
myFile.open("filename.out");
```

- To append data to an existing file, you must specify `std::ios::app`
- Inside a regularly called method (e.g. inside a virtual method of an *User Class*), write your data (i.e. G4double, G4int, G4String,...) to file, in the same fashion of G4cout:

```
if (myFile.is_open()) // Check that file is opened  
{  
    myFile << kineticEnergy << '\t' << dose << G4endl;  
    ...  
}
```

- Finally close the file, in the class destructor, or into a specific method:

```
myFile.close();
```

# ROOT

- ROOT is an Object Oriented Data Analysis Framework.
- It is heavily used in High Energy Physics.
- Freely available.
- <http://root.cern.ch/>

## Using ROOT in your C++ Geant4 code

it is necessary to add in the header (.hh file) of a specific class devoted to analysis the needed include ROOT files, i.e.:

Mandatory headers

NTuples, 1-D(float) & 3-D(double) histograms

Graphic objects

```
#include "TROOT.h"
#include "TFile.h"
#include "TNtuple.h"
#include "TTree.h"
#include "TH1F.h"
#include "TH3D.h"
#include "TCanvas.h"
#include "TGraph.h"
#include "TAxis.h"
#include "TLegend.h"
#include "TLegendEntry.h"
#include "TLegend.h"
#include "TStyle.h"
```

# ROOT

Declare the needed **ROOT** objects in your class header:

- **TFile** \*theTFile; // ROOT file
- **TH1F** \*histoEnergyDepositedPerEvent; // I-D histogram
- **TNtuple** \*kinFragNtuple; // ntuple
- ...

Then create an instance for each object in the class constructor, or in a specific method (also in case you need to recreate the file more than one time per simulation):

```
theTFile = new TFile("myFileName", "RECREATE");
```

This will create the file myFileName.root containing an image of ROOT variables. The option "RECREATE" means that an existing file will be overwritten!

# ROOT

An **instance** of each defined object can be created, in the class **constructor** or in a specific **method** called once, via the new operator:

```
// Histogram containing the energy deposited in the FIRST slice of the
// detector, at each event;
histoEnergyDepositedPerEvent = new TH1F("EnergyPerEvent",
                                         "Energy, Counts",
                                         400,
                                         50.0,
                                         70.0);

kinFragNtuple = new TNtuple("kinFragNtuple",
                           "Kinetic energy by voxel & fragment",
                           "i:j:k:A:Z:kineticEnergy");
```

Now you can define a class method (remember to put declarations in .hh file!) to fill each ROOT object. Data are temporarily written to memory, then flushed to file:

```
////////////////////////////////////
// FillKineticFragmentTuple create an ntuple where the voxel indexes, the atomic number and mass and the kinetic
// energy of all the particles interacting with the phantom, are stored
void HadrontherapyAnalysisManager::FillKineticFragmentTuple(G4int i,
                                                           G4int j,
                                                           G4int k,
                                                           G4int A,
                                                           G4double Z,
                                                           G4double kinEnergy)
{
    kinFragNtuple -> Fill(i, j, k, A, Z, kinEnergy);
}
```



# Closing all ROOT objects

At the end of the simulation (i.e. at the end of a run) write & finalize the ROOT file.  
You can invoke the method to finalize the file:

- ✓ At the end of the RunAction (if you don't plan to have many runs)
- ✓ In the object destructor
- ✓ At the end of the *main*

```
////////////////////////////////////  
// Flush data & close the file  
void HadrontherapyAnalysisManager::flush()  
{  
  if (theTFile) ←  
  {  
    theTFile -> Write();  
    theTFile -> Close();  
  }  
}
```

It's a good programming practice to set this pointer to NULL in the class constructor

This will finalize and close the ROOT file, moreover it frees the memory

# Conclusions

- The final goal of any MC simulation is to retrieve physical information
- Geant4 provides a **powerful** and **flexible** system to retrieve and score information during the run based on:
  - Sensitive Detectors (attached to logical volumes)
  - Hits
  - Hits Collections (attached to the G4Event)
  - Require **concrete classes** written by the user to work
- An other possibility is to use built-in Geant4 scorers
  - Less work to do but much less flexible
  - Suggested only in case you need a limited amount of information and/or for a restricted scope

Thanks