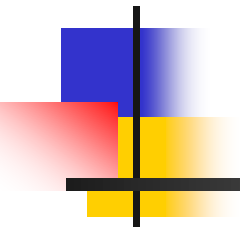
Geant4: how to retrieve information

Luciano Pandola

INFN



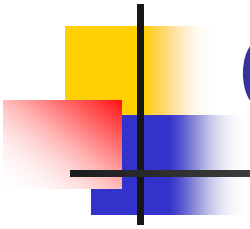
Part I: The main ingredients

Optional user classes - 1

- Five concrete base classes whose virtual member functions the user may override to gain control of the simulation at various stages
  - G4User**Run**Action
  - G4User**Event**Action
  - G4User**Tracking**Action
  - G4User**Stacking**Action
  - G4User**Stepping**Action

e.g. actions to be done at the beginning and end of each event

- Each member function of the base classes has a dummy implementation (**not** purely virtual)
  - Empty implementation: does nothing



Optional user classes - 2

- The user may implement the member functions he desires in his/her derived classes
  - E.g. one may want to perform some action at each tracking step
- Objects of user action classes must be **registered** to the **G4(MT)RunManager** via the ActionInitialization

```
runManager->SetUserAction(new
  MyActionInitialization);
```

MyActionInitialization (MT mode)

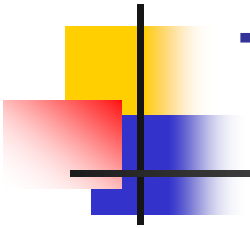
- Register thread-local user actions

```
void MyActionInitialization::Build() const
{
  //Set mandatory classes
  SetUserAction(new MyPrimaryGeneratorAction());
  // Set optional user action classes
  SetUserAction(new MyEventAction());
  SetUserAction(newMyRunAction());
}
```

- Register RunAction for the master

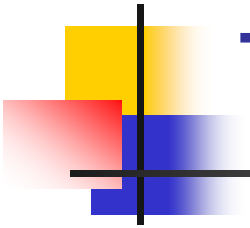
```
void MyActionInitialization::BuildForMaster() const
{
  // Set optional user action classes
 SetUserAction(newMyMasterRunAction());
}
```

The Run (G4Run)



- As an analogy with a real experiment, a run of Geant4 starts with '**Beam On**'
- Within a run, the User **cannot change**
  - The detector setup
  - The physics setting (processes, models)
- A Run is a collection of events with the same detector and physics conditions
- At the beginning of a Run, geometry is optimised for navigation and cross section tables are (re)calculated
- The G4RunManager class manages the processing of each Run, represented by:
  - **G4Run** class
  - **G4UserRunAction** for an optional User hook

The Event (G4Event)



- An Event is the basic unit of simulation in Geant4
- At the beginning of processing, primary tracks are generated and they are pushed into a stack
- A track is popped up from the stack one-by-one **and** '**tracked**'
  - Secondary tracks are also pushed into the stack
  - When the stack gets empty, the processing of the event is completed
- **`G4Event`** class represents an event. At the end of a successful event it has:
  - List of primary vertices and particles (as input)
  - Hits and Trajectory collections (as outputs)
- **`G4EventManager`** class manages the event.
- **`G4UserEventAction`** is the optional User hook

The Step (G4Step)

- **`G4Step`** represents a step in the particle propagation
- A G4Step object stores transient information of the step
  - In the tracking algorithm, G4Step is updated each time a process is invoked
- You can extract information from a step after the step is completed
  - Both, the **`ProcessHits()`** method of your sensitive detector and **`UserSteppingAction()`** of your step action class file get the pointer of G4Step
  - Typically , you may retrieve information in these functions (for example fill histograms in Stepping action)

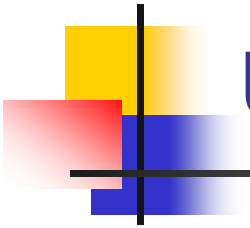
The Track (G4Track)

- The Track is a **snapshot of a particle** and it is represented by the **`G4Track`** class
  - It keeps ‘current’ information of the particle (i.e. energy, momentum, position, polarization, ..)
  - It is updated after every step
- The track object is deleted when
  - It goes outside the world volume
  - It disappears in an interaction (decay, inelastic scattering)
  - It is slowed down to zero kinetic energy and there are no 'AtRest' processes
  - It is manually killed by the user
- No track object **persists** at the end of the event
- **`G4TrackingManager`** class manages the tracking
- **`G4UserTrackingAction`** is the optional hook for tracking

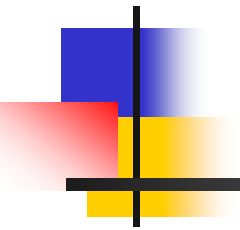
Example of usage of the hook user classes - 1

- **`G4UserRunAction`**
  - Has two methods (**`BeginOfRunAction()`** and **`EndOfRunAction()`**) and can be used e.g. to initialise, analyse and store histogram
  - Everything User want to know at this stage
- **`G4UserEventAction`**
  - Has two methods (**`BeginOfEventAction()`** and **`EndOfEventAction()`**)
  - One can apply an event selection, for example
  - Access the hit-collection and perform the event analysis

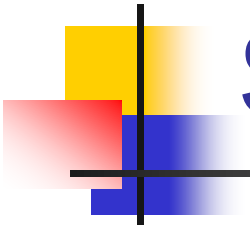
Example of usage of the hook user classes - 2



- **`G4UserStakingAction`**
  - Classify priority of tracks
- **`G4UserTrackingAction`**
  - Has two methods (**`PreUserTrakingAction()` `and PostUserTrackinAction()`**)
  - For example used to decide if trajectories should be stored
- **`G4UserSteppingAction`**
  - Has a method which is invoked at the end of a step



Part II: 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 → see next slides)
    - Highly **customizable**
  - Use Geant4 built-in tools: Primitive scorers

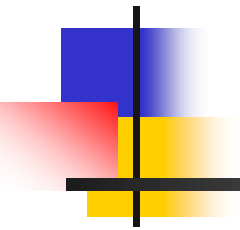
Adding sensitivity to a logical volume

- Create an instance of a sensitive detector
- Assign the pointer of your SD to the logical volume of your detector geometry
- Must be done in **`ConstructSDandField`**() of the user geometry class

```
G4VSensitiveDetector* mySensitive
   = new MySensitiveDetector(SDname="/MyDetector");
```
create instance

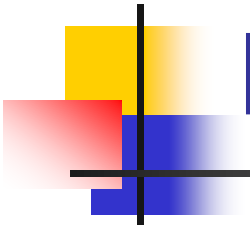
```
boxLogical->SetSensitiveDetector(mySensitive);
```
assign to logical volume

```
(or)
SetSensitiveDetector("LVname",mySensitive);
```
assign to logical volume (alternative)



Part III: 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 alterative 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 score object among multiple G4MultiFunctionalDetector objects (results may mix up!)

myCellScorer/TotalSurfFlux
myCellScorer/TotalDose

For example ...

```
MyDetectorConstruction::ConstructSDandField()
{
   G4MultiFunctionalDetector* myScorer = new
   G4MultiFunctionalDetector("myCellScorer");

    myCellLog->SetSensitiveDetector(myScorer);

    G4VPrimitiveSensitivity* totalSurfFlux = new
      G4PSFlatSurfaceFlux("TotalSurfFlux");
    myScorer->RegisterPrimitive(totalSurfFlux);
    G4VPrimitiveSensitivity* totalDose = new
      G4PSDoseDeposit("TotalDose");
    myScorer->RegisterPrimitive(totalDose);
}
```

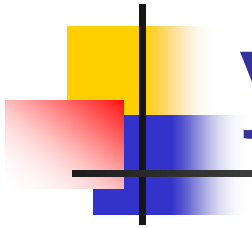
instantiate multi-functional detector

attach to volume

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

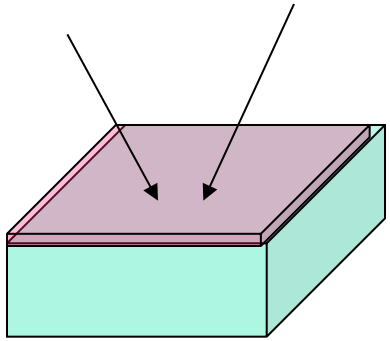


- Concrete Primitive Scorers (→ Application Developers Guide 4.4.5)
  - Track length
    - G4PSTrackLength, G4PSPassageTrackLength
  - Deposited energy
    - G4PSEnergyDepsit, G4PSDoseDeposit
  - Current/Flux
    - G4PSFlatSurfaceCurrent, G4PSSphereSurfaceCurrent,G4PSPassageCurrent, G4PSFlatSurfaceFlux, G4PSCellFlux, G4PSPassageCellFlux
  - Others
    - G4PSMinKinEAtGeneration, G4PSNofSecondary, G4PSNofStep, G4PSCellCharge

A closer look at some scorers

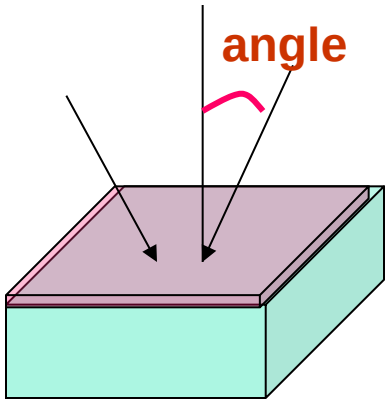
SurfaceCurrent :

Count number of injecting particles at defined surface.



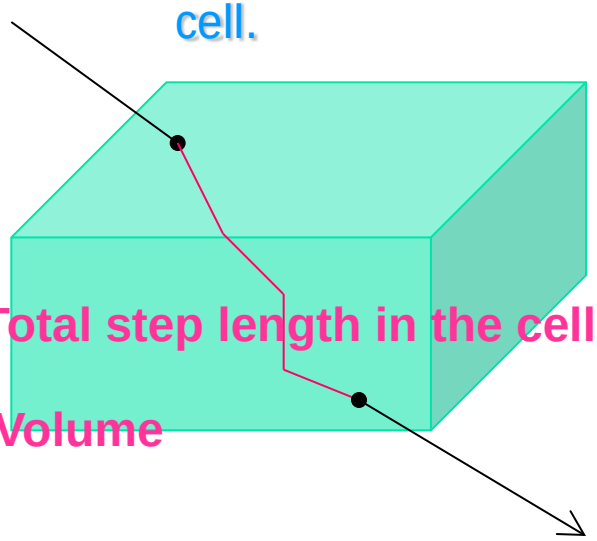
CellFlux :

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



SurfaceFlux :

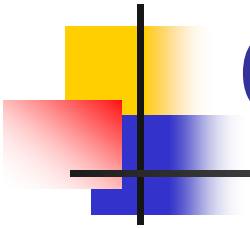
Sum up 1/cos(angle) of injecting particles at defined surface



L : Total step length in the cell

V : Volume

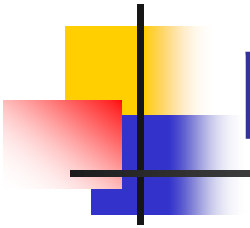
V : Volume



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)

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 **`G4THitsMap<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");
```

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

```
G4HCofThisEvent* HCE = event->GetHCofThisEvent();
```

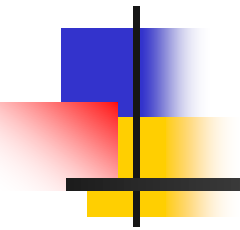
Get **all HC** available in this event

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

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

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

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

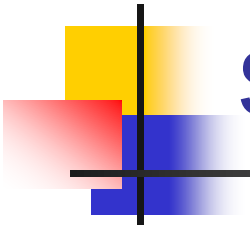


Part IV: Write information on output files

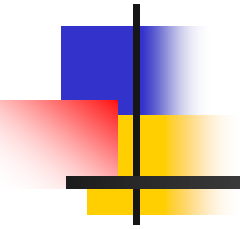
Introduction: data analysis with Geant4

- For a long time, Geant4 did not attempt to provide/support **any data analysis** tools
  - The focus was given (and is given) to the central mission as a Monte Carlo simulation toolkit
  - As a general rule, the user is expected to provide her/his own code to output results to an appropriate analysis format
- Basic classes for data analysis have recently been implemented in Geant4 (g4analysis)
  - Support for histograms and ntuples
  - Output in **ROOT**, **XML**, **HBOOK** and **CSV** (ASCII)
  - Appropriate only for easy/quick analysis: for advanced tasks, the user must write his/her own code and to use an external analysis tool

Introduction: how to write simulation results

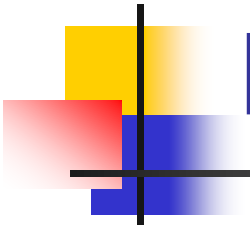


- Formatted (= human-readable) **ASCII files**
  - Simplest possible approach is comma-separated values (.csv) files
  - The resulting files can be opened and analyzed by tools such as: Gnuplot, Excel, OpenOffice, Matlab, Origin, ROOT, PAW, ...
- **Binary files** with complex analysis objects (Ntuples)
  - 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 , PAW, AIDA-compliant (PI, JAS3 and OpenScientist)

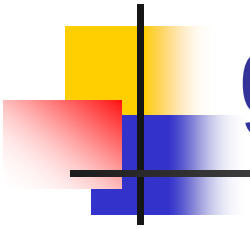


G4analysis tools

Native Geant4 analysis classes



- A basic analysis interface is available in Geant4 for histograms (1D and 2D) and ntuples
  - Make life easier because they are MT-compliant (no need to worry about the interference of threads)
- Unique interface to support different output formats
  - ROOT, AIDA XML, CSV and HBOOK
  - Code is the same, just change one line to switch from one to an other
- Everything done via the public analysis interface **`G4AnalysisManager`**
  - Singleton class: Instance()
  - **UI commands** available for creating histograms at run-time and setting their properties



g4analysis

- Selection of output format is hidden in a user-defined .hh file
- **All** the rest of the code unchanged
  - Unique interface

```
#ifndef MyAnalysis_h
#define MyAnalysis_h 1

#include "g4root.hh"
//#include "g4xml.hh"
//#include "g4csv.hh"  // can be used only with ntuples

#endif
```

Open file and book histograms

```
#include "MyAnalysis.hh"

void MyRunAction::BeginOfRunAction(const G4Run* run)
{
  // Get analysis manager
  G4AnalysisManager* man = G4AnalysisManager::Instance();
  man->SetVerboseLevel(1);
  man->SetFirstHistoId(1);

  // Creating histograms
  man->CreateH1("h","Title", 100, 0., 800*MeV);
  man->CreateH1("hh","Title",100,0.,10*MeV);

 // Open an output file
  man->OpenFile("myoutput");
}
```

Start numbering of histograms from ID=1

ID=1

ID=2

Open output file

Fill histograms and close

```
#include "MyAnalysis.hh"
void MyEventAction::EndOfEventAction(const G4Run* aRun)
{
  G4AnalysisManager* man = G4AnalysisManager::Instance();
  man->FillH1(1, fEnergyAbs);
  man->FillH1(2, fEnergyGap);
}
void MyRunAction::EndOfRunAction(const G4Run* aRun)
{
  G4AnalysisManager* man = G4AnalysisManager::Instance();
  man->Write();
  man->CloseFile();
}
MyRunAction::~MyRunAction()
{
  delete G4AnalysisManager::Instance();
}
```

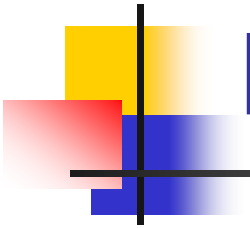
ID=1

ID=2

Histograms

- Support linear and log scales and un-even bins
- **`CreateH2()`** for 2D histograms

```
G4int CreateH1(const G4String& name, const G4String& title,
               G4int nbins, G4double xmin, G4double xmax,
               const G4String& unitName = "none",
               const G4String& fcnName = "none",
               const G4String& binSchemeName = "linear");

G4int CreateH1(const G4String& name, const G4String& title,
               const std::vector<G4double>& edges,
               const G4String& unitName = "none",
               const G4String& fcnName = "none");
```



Ntuples

- g4tool supports ntuples
  - Any number of ntuples, each with any number of columns
  - The content can be **int / float / double**
- For more complex tasks (e.g. full functionality of ROOT TTrees) have to link ROOT directly
- Similar strategy as for histograms. Access happens through the common interface **`G4AnalysisManager`**
  - Saved on the same output file with histograms

Book ntuples

```
#include "MyAnalysis.hh"
void MyRunAction::BeginOfRunAction(const G4Run* run)
{
  // Get analysis manager
  G4AnalysisManager* man = G4AnalysisManager::Instance();
  man-> SetFirstNtupleId(1);

  // Creating ntuple
  man->CreateNtuple("name", "Title");
  man->CreateNtupleDColumn("Eabs");
  man->CreateNtupleDColumn("Egap");
  man->FinishNtuple();

  man->CreateNtuple("name2","title2");
  man->CreateNtupleIColumn("ID");
  man->FinishNtuple();
}
```

Start numbering of ntuples from ID=1

ID=1

ID=2

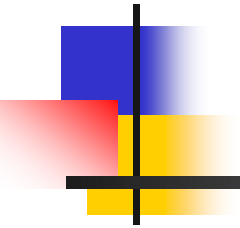
Fill ntuples

- File handling and general clean-up as shown for histograms

```
#include "MyAnalysis.hh"
void MyEventAction::EndOfEventAction(const G4Run* aRun)
{
  G4AnalysisManager* man = G4AnalysisManager::Instance();
  man->FillNtupleDColumn(1, 0, fEnergyAbs);
  man->FillNtupleDColumn(1, 1, fEnergyGap);
  man->AddNtupleRow(1);

  man->FillNtupleIColumn(2, 0, fID);
  man->AddNtupleRow(2);

}
```

ID=1, columns 0, 1

ID=2, column 0

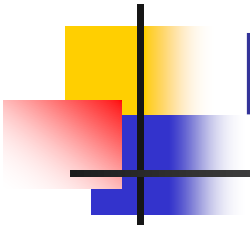


Part V: User-defined sensitive detectors: Hits and Hits Collection

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*>

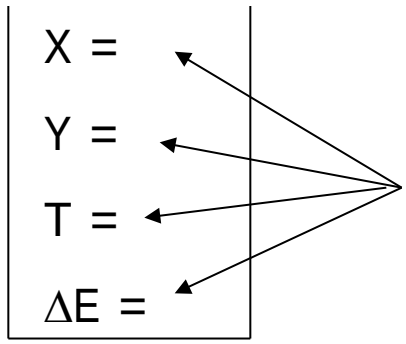


Hit class - 1

- 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”, an **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

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 EndOfEvent

Hit class - 3

Example

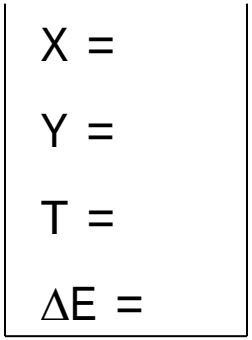
```
// header file: MyHit.hh

#include "G4VHit.hh"

class MyHit : public G4VHit {

 public:
   MyHit();
   virtual ~MyHit();
   ...

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

   inline G4double GetEnergyDeposit() { return energyDeposit;}

  ... // more get and set methods

private:
   G4double energyDeposit;
  ... // more data members
};
```

public methods to handle data member

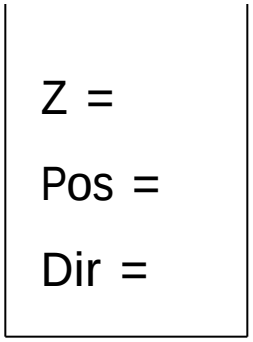
data member (private)

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



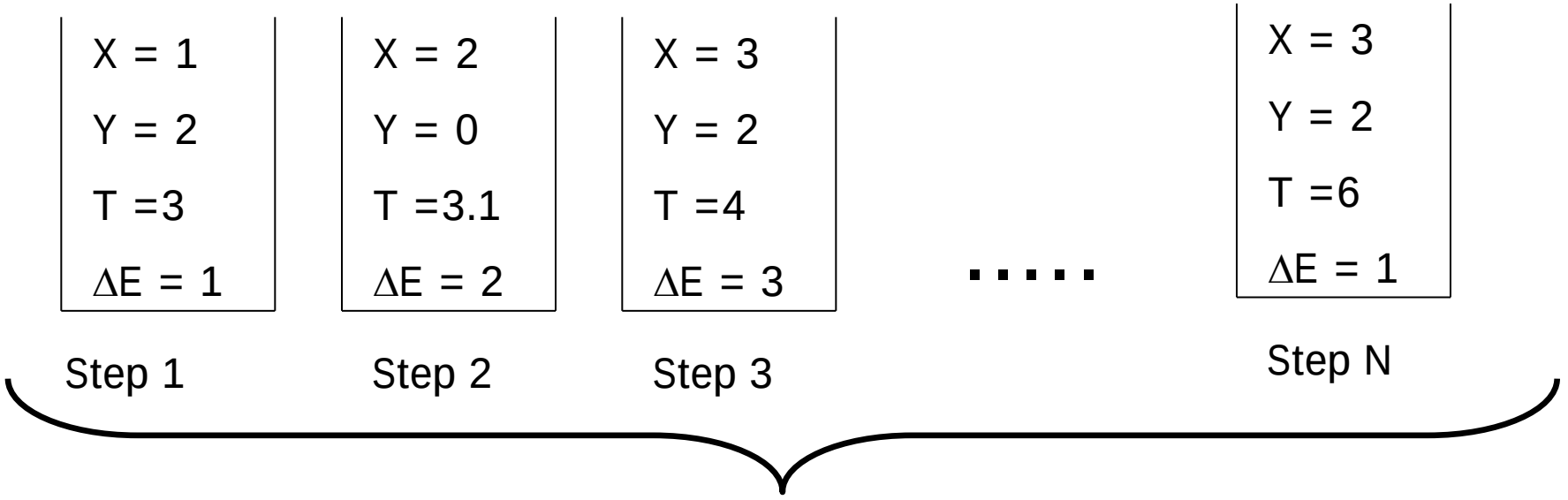
Class Hit1 :
public G4VHit



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 inkoved: it must **create**, **fill** and **store** the Hit objects

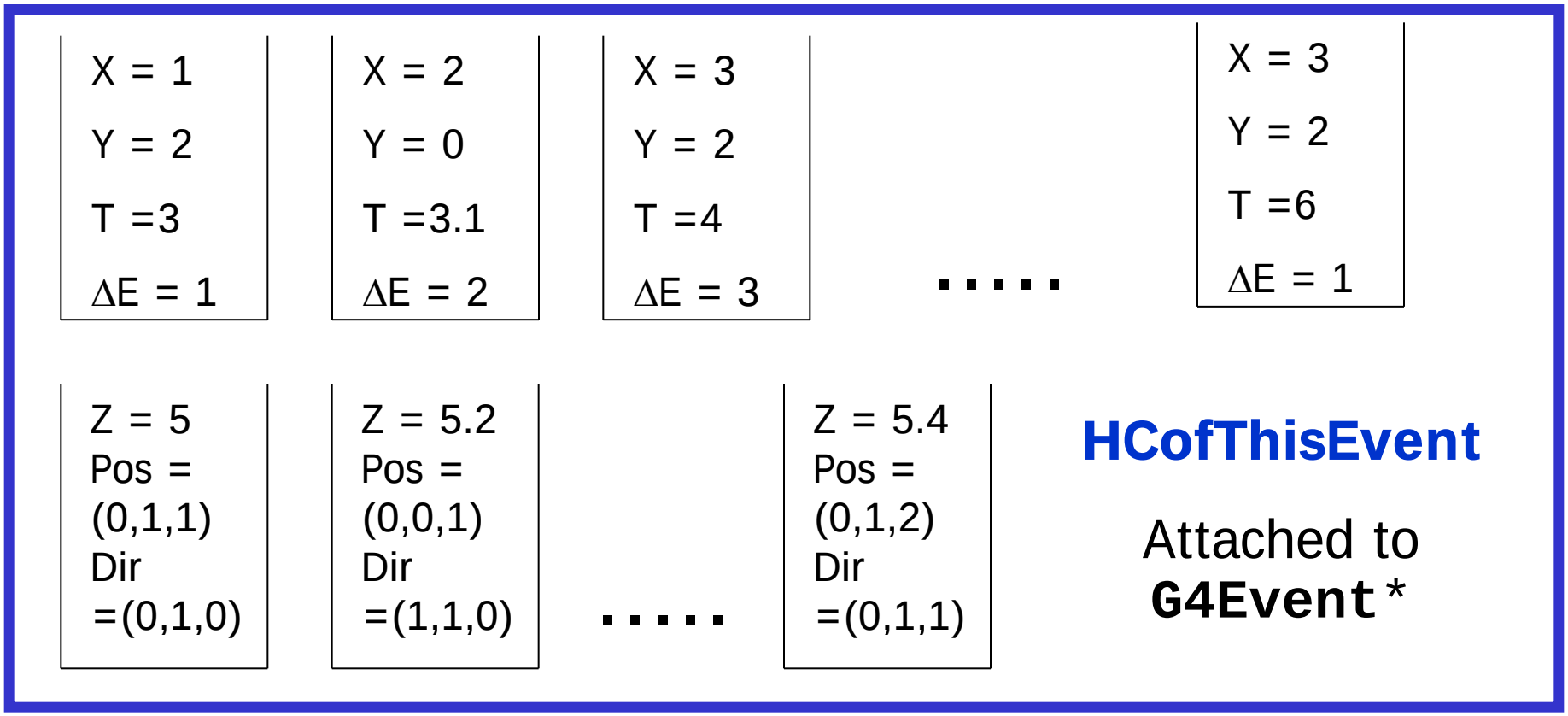


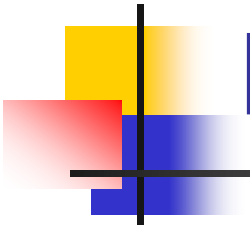
Hits Collection - 2

- 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

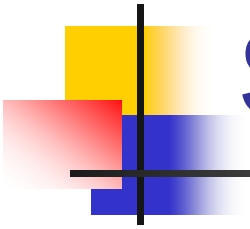
Remember that you may have **many kinds of Hits** (and Hits Collections)





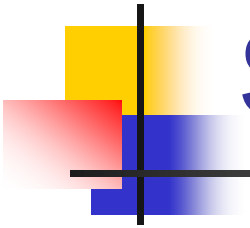
Hits Collections of an event

- 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 creted within the event
  - Hits collections are accessible and can be processes 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`**



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:
   ...
     virtual void Initialize(G4HCofThisEvent*);
     virtual void EndOfEvent(G4HCofThisEvent*);
 protected:
     virtual G4bool ProcessHits(G4Step* ,
                                G4TouchableHistory*) = 0;
   ...
};
```

abstract base class

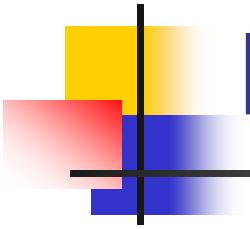
pure virtual method

```
// 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

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