



Interaction with the Geant4 kernel

Luciano Pandola
INFN-LNGS and LNS



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



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
 - G4UserRunAction
 - G4UserEventAction
 - G4UserTrackingAction
 - G4UserStackingAction
 - G4UserSteppingAction

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(new MyRunAction());
}
```

- Register RunAction for the **master**

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



Geant4 terminology: an overview

- The following keywords are often used in Geant4
 - Run, Event, Track, Step
 - Processes: At Rest, Along Step, Post Step
 - Cut (or production threshold)



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)

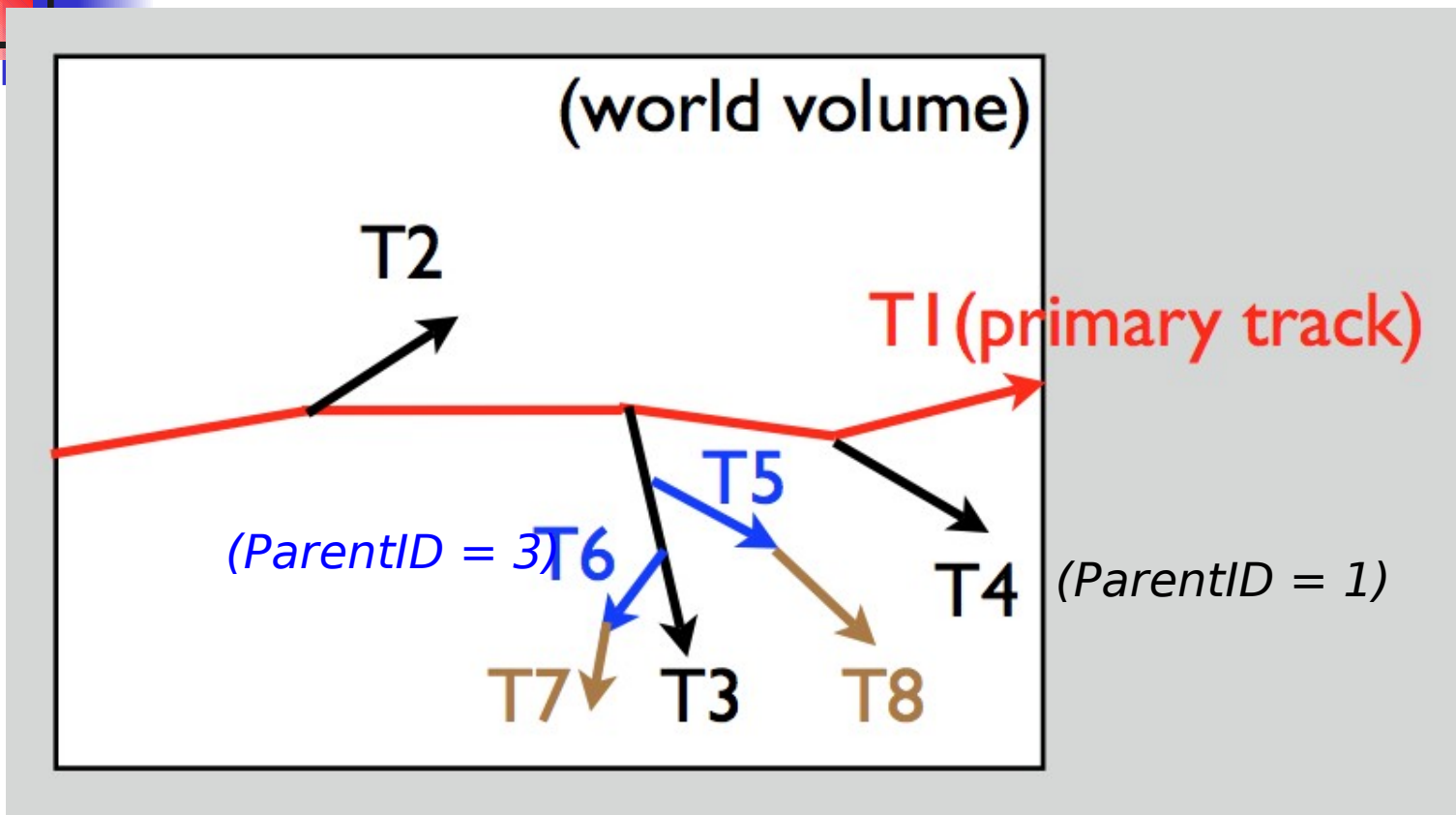
- The Track is a snapshot of a particle and it is represented by the G4Track class
 - It keeps 'current' information of the particle (i.e. energy, momentum, position, polarization, ..)
 - It is updated after every step
- The track object is deleted when
 - It goes outside the world volume
 - It disappears 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 User hook



Run, Event and Tracks

- One Run consists of
 - Event #1 (track #1, track #2,)
 - Event #2 (track #1, track #2,)
 -
 - Event #N (track #1, track #2,)

Example of an Event and Tracks



- Tracking order follows 'last in first out' rule:
T1 -> T4 -> T3 -> T6 -> T7 -> T5 -> T8 -> T2

* G4Track Information: Particle = e-, Track ID = 87, Parent ID = 1

Step#	X(mm)	Y(mm)	Z(mm)	KinE(MeV)	dE(MeV)	StepLeng	TrackLeng	NextVolume	ProcName
0	-1.87e+03	5.63	-5.52	0.0326	0	0	0	physicalTreatmentRoom	initStep
1	-1.87e+03	5.85	-4.72	0.032	0.000545	0.924	0.924	physicalTreatmentRoom	msc
2	-1.87e+03	5.92	-3.9	0.0317	0.00036	0.928	1.85	physicalTreatmentRoom	msc
3	-1.87e+03	5.89	-3.65	0.0289	0.00013	0.3	2.15	physicalTreatmentRoom	eIoni
:----- List of 2ndaries - #SpawnInStep= 1(Rest= 0,Along= 0,Post= 1), #SpawnTotal= 1 -----									
:	-1.87e+03	5.89	-3.65	0.00258					e-
:----- EndOf2ndaries Info -----									
4	-1.87e+03	5.81	-2.87	0.0279	0.00104	0.928	3.08	physicalTreatmentRoom	msc
5	-1.87e+03	5.35	-2.11	0.0273	0.000654	0.928	4.01	physicalTreatmentRoom	msc
6	-1.87e+03	5.01	-1.28	0.0248	0.00249	0.928	4.94	physicalTreatmentRoom	msc
7	-1.87e+03	5.03	-0.37	0.0231	0.00163	0.928	5.87	physicalTreatmentRoom	msc
8	-1.87e+03	4.78	0.503	0.022	0.00109	0.928	6.79	physicalTreatmentRoom	msc
9	-1.87e+03	4.64	1.35	0.0202	0.00184	0.928	7.72	physicalTreatmentRoom	msc
10	-1.87e+03	4.68	2.26	0.0181	0.00204	0.928	8.65	physicalTreatmentRoom	msc
11	-1.87e+03	4.63	2.46	0.0165	0.000345	0.231	8.88	physicalTreatmentRoom	eIoni
:----- List of 2ndaries - #SpawnInStep= 1(Rest= 0,Along= 0,Post= 1), #SpawnTotal= 2 -----									
:	-1.87e+03	4.63	2.46	0.00133					e-
:----- EndOf2ndaries Info -----									
12	-1.87e+03	4.6	2.49	0.0125	0	0.0383	8.92	physicalTreatmentRoom	eIoni
:----- List of 2ndaries - #SpawnInStep= 1(Rest= 0,Along= 0,Post= 1), #SpawnTotal= 3 -----									
:	-1.87e+03	4.6	2.49	0.00402					e-
:----- EndOf2ndaries Info -----									

* G4Track Information: Particle = e-, Track ID = 242, Parent ID = 87

Step#	X(mm)	Y(mm)	Z(mm)	KinE(MeV)	dE(MeV)	StepLeng	TrackLeng	NextVolume	ProcName
0	-1.87e+03	6.1	5.41	0.00138	0	0	0	physicalTreatmentRoom	initStep
1	-1.87e+03	6.11	5.39	0.000253	0.00112	0.0481	0.0481	physicalTreatmentRoom	msc
2	-1.87e+03	6.12	5.39		0 0.000253	0.0088	0.0569	physicalTreatmentRoom	eIoni

Example:

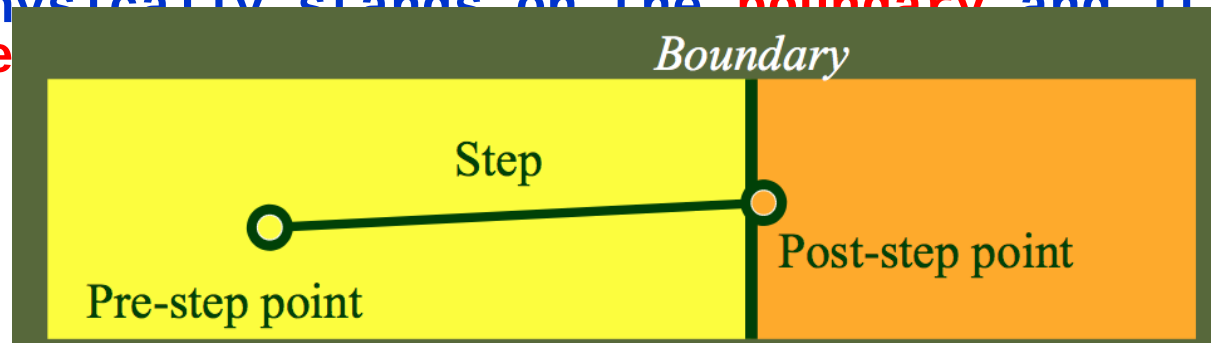
retrieving information from tracks

// retrieving information from tracks (given the G4Track object "track"):

```
if(track -> GetTrackID() != 1) {  
    G4cout << "Particle is a secondary" << G4endl;  
  
    // Note in this context, that primary hadrons might loose their identity  
    if(track -> GetParentID() == 1)  
        G4cout << "But parent was a primary" << G4endl;  
  
    G4VProcess* creatorProcess = track -> GetCreatorProcess();  
  
    if(creatorProcess -> GetProcessName() == "LowEnergyIoni") {  
        G4cout << "Particle was created by the Low-Energy " <<  
            << "Ionization process" << G4endl;  
    }  
}
```

The Step in Geant4

- The G4Step has the information about the two points (pre-step and post-step) and the 'delta' information of a particle (energy loss on the step,)
- Each point knows the volume (and the material)
 - In case a step is limited by a volume boundary, the end point physically stands on the boundary and it logically belongs to the volume it came from



- G4SteppingManager class manages processing a step; a 'step' is represented by the G4Step class
- G4UserSteppingAction is the optional User hook



The G4Step object

- A G4Step object contains
 - The two endpoints (pre and post step) so one has access to the volumes containing these endpoints
 - Changes in particle properties between the points
 - Difference of particle energy, momentum,
 - Energy deposition on step, step length, time-of-flight, ...
 - A pointer to the associated G4Track object
- G4Step provides many Get methods to access these information or object instances
 - G4StepPoint* GetPreStepPoint(),

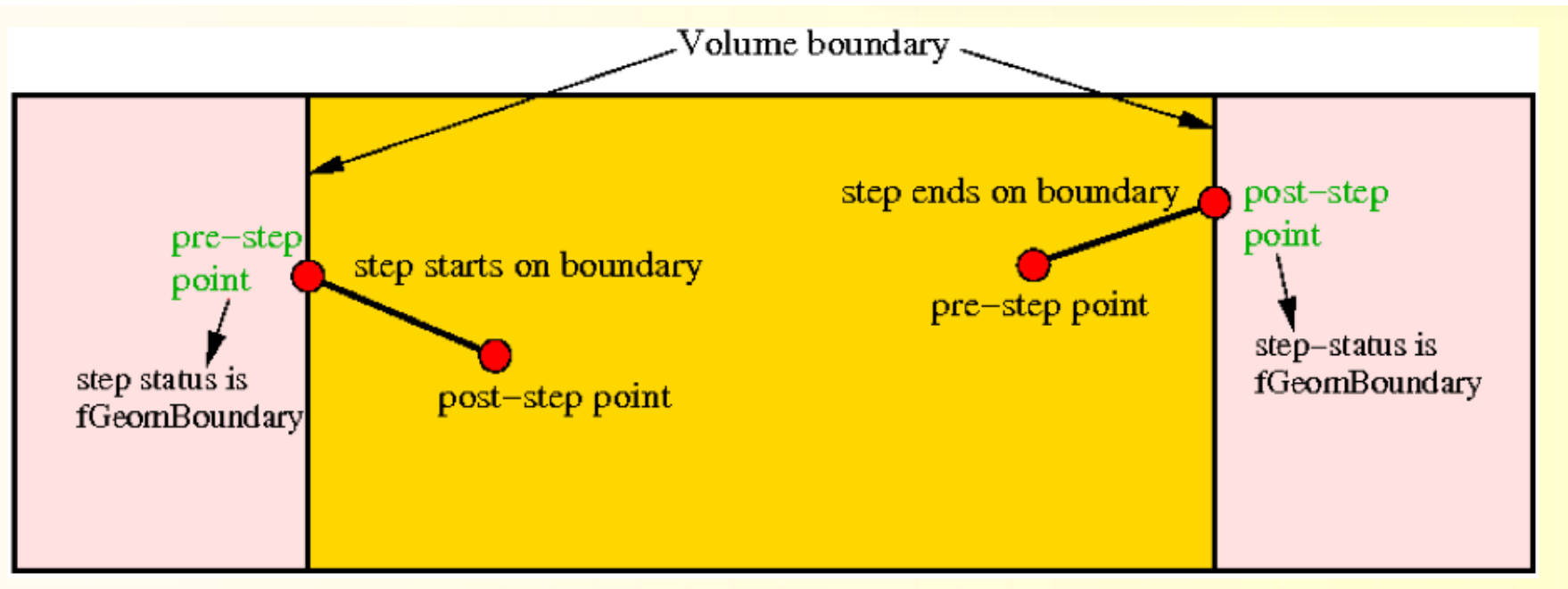


The geometry boundary

- To check, if a step ends on a boundary, one may compare if the physical volume of pre and post-step points are equal
- One can also use the step status
 - Step Status provides information about the process that restricted the step length
 - It is attached to the step points: the pre has the status of the previous step, the post of the current step
 - If the status of POST is "fGeometryBoundary" the step ends on a volume boundary (does not apply to word volume)
 - To check if a step starts on a volume boundary you can also use the step status of the PRE-step point

Step concept and boundaries

Illustration of step starting and ending on boundaries





Geant4 terminology: an overview

	Object	Description
Run	G4Run	Largest unit of simulation, that consist of a sequence of events: If a defined number of events was processed a run is finished.
Event	G4Event	Basic simulation unit in Geant4: If a defined number of primary tracks and all resulting secondary tracks were processed an event is over.
Track	G4Track	A track is NOT a collection of steps: It is a snapshot of the status of a particle after a step was completed (but it does NOT record previous steps). A track is deleted, if the particle leaves world, has zero kinetic energy,
Step	G4Step	Represents a particle step in the simulation and includes two points (pre-step point and post-step point).



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

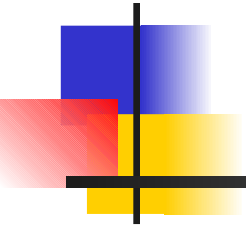
- Has two methods (BeginOfEventAction() and EndOfEventAction())
- One can apply an event selection, for example
- Access the hit-collection and perform the event analysis



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: Retrieving information from steps and tracks





Example: check if step is on boundaries

// in the source file of your user step action class:

```
#include "G4Step.hh"
```

```
UserStepAction::UserSteppingAction(const G4Step* step) {
```

```
    G4StepPoint* preStepPoint = step -> GetPreStepPoint();
```

```
    G4StepPoint* postStepPoint = step -> GetPostStepPoint();
```

// Use the GetStepStatus() method of G4StepPoint to get the status of the

// current step (contained in post-step point) or the previous step

// (contained in pre-step point):

```
    if(preStepPoint -> GetStepStatus() == fGeomBoundary) {
```

```
        G4cout << "Step starts on geometry boundary" << G4endl;
```

```
    }
```

```
    if(postStepPoint -> GetStepStatus() == fGeomBoundary) {
```

```
        G4cout << "Step ends on geometry boundary" << G4endl;
```

// You can retrieve the material of the next volume through the

// post-step point :

```
        G4Material* nextMaterial = step -> GetPostStepPoint()->GetMaterial();
```

```
    }
```

```
}
```

Example: step information in SD

// in source file of your sensitive detector:

```
MySensitiveDetector::ProcessHits(G4Step* step,  
                                G4TouchableHistory*) {
```

```
// Total energy deposition on the step (= energy deposited by energy loss  
// process and energy of secondaries that were not created since their  
// energy was < Cut):
```

```
G4double energyDeposit = step -> GetTotalEnergyDeposit();
```

```
// Difference of energy , position and momentum of particle between pre-  
// and post-step point
```

```
G4double deltaEnergy = step -> GetDeltaEnergy();
```

```
G4ThreeVector deltaPosition = step -> GetDeltaPosition();
```

```
G4double deltaMomentum = step -> GetDeltaMomentum();
```

```
// Step length
```

```
G4double stepLength = step -> GetStepLength();
```

```
}
```



Something more about tracks

- After each step the track can change its state
- The status can be (in red can only be set by the track)

Track Status	Description
fAlive	The particle is continued to be tracked
fStopButAlive	Kin. Energy = 0, but AtRest process will occur
fStopAndKill	Track has lost identity (has reached world boundary, decayed, ...), Secondaries will be tracked
fKillTrackAndSecondaries	Track and its secondary tracks are killed
fSuspend	Track and its secondary tracks are suspended (pushed to stack)
fPostponeToNextEvent	Track but NOT secondary tracks are postponed to the next event (secondaries are tracked in current event)



Particles in Geant4

- A particle in general has the following three sets of properties:
 - **Position/geometrical info**
 - G4Track class (representing a particle to be tracked)
 - **Dynamic properties: momentum, energy, spin, ..**
 - G4DynamicParticle class
 - **Static properties: rest mass, charge, life time**
 - G4ParticleDefinition class
- All the G4DynamicParticle objects of the same kind of particle share the same G4ParticleDefinition



Particles in Geant4

Class	What does it represent?	What does it contain?
G4Track	Represents a particle that travels in space and time	Information relevant to tracking the particle, e.g. position, time, step,..., and <i>dynamic information</i>
G4DynamicParticle	Represents a particle that is subject to interactions with matter	Dynamic information, e.g. particle momentum, kinetic energy, ..., and <i>static information</i>
G4ParticleDefinition	Defines a physical particle	Static information, e.g. particle mass, charge, ... Also physics processes relevant to the particle



Examples: particle information from step/track

```
#include "G4ParticleDefinition.hh"
#include "G4DynamicParticle.hh"
#include "G4Step.hh"
#include "G4Track.hh"

// Retrieve from the current step the track (after PostStepDoIt of step is
// completed):
G4Track* track = step -> GetTrack();

// From the track you can obtain the pointer to the dynamic particle:
const G4DynamicParticle* dynParticle = track -> GetDynamicParticle();

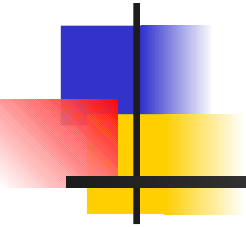
// From the dynamic particle, retrieve the particle definition:
G4ParticleDefinition* particle = dynParticle -> GetDefinition();

// The dynamic particle class contains e.g. the kinetic energy after the step:
G4double kinEnergy = dynParticle -> GetKineticEnergy();

// From the particle definition class you can retrieve static information like
// the particle name:
G4String particleName = particle -> GetParticleName();

G4cout << particleName << ": kinetic energy of "
        << kinEnergy/MeV << " MeV"
        << G4endl;
```

Part III: 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**

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");  
  
boxLogical->SetSensitiveDetector(mySensitive);  
(or)  
SetSensitiveDetector("LVname", mySensitive);
```

} create instance

} assign to logical volume

} assign to logical volume (alternative)



Part IV: 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 score 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

```
}
```

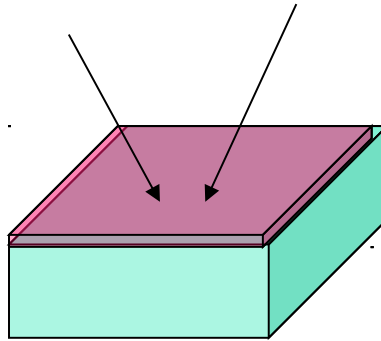


Some primitive scorers that you may find useful

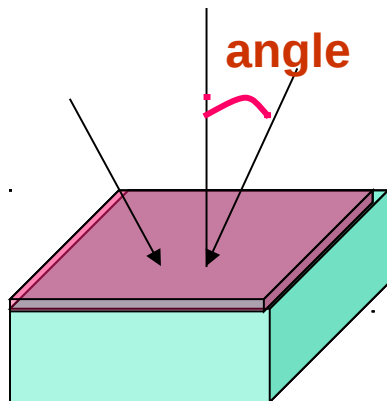
- Concrete Primitive Scorers (→ Application Developers Guide 4.4.5)
 - **Track length**
 - G4PSTrackLength, G4PSPassageTrackLength
 - **Deposited energy**
 - G4PSEnergyDeposit, G4PSDoseDeposit
 - **Current/Flux**
 - G4PSFlatSurfaceCurrent, G4PSSphereSurfaceCurrent, G4PSPassageCurrent, G4PSFlatSurfaceFlux,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

A closer look at some scorers

SurfaceCurrent :
Count number of
injecting particles
at defined surface.

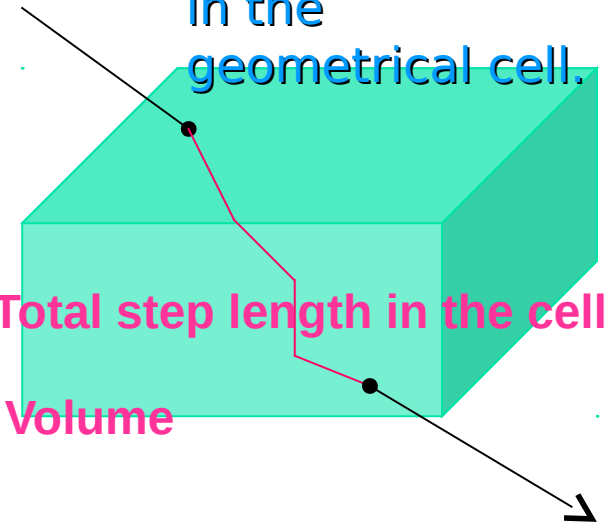


CellFlux :
Sum of L / V of
injecting
particles
in the
geometrical cell.



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 `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



Command-based scoring

Under development!

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)

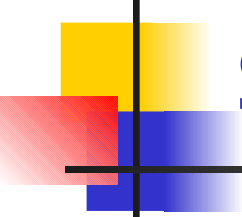


Part V: Write information on output files

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)



Output stream (G4cout)

- G4cout is a iostream object defined by Geant4.
 - The usage of this objects is exactly the same as the ordinary `std::cout` except that the output streams will be handled by G4UImanager
 - `G4endl` is the equivalent of `std::endl` to end a line
- Output strings may be displayed on another window or stored in a file
- One can also use the file streams (`std::ofstream`) provided by the C++ libraries

Output on screen – an example

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

    evtNb = eventAction -> Trasporto();

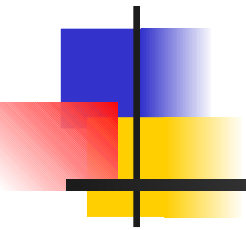
    G4String particleName = aStep -> GetTrack() -> GetDynamicParticle() -> GetDefinition() -> GetParticleName();
    G4String volumeName = aStep -> GetPreStepPoint() -> GetPhysicalVolume() -> GetName();
    G4double particleCharge = aStep -> GetTrack() -> GetDefinition() -> GetAtomicNumber();
    G4double 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 on screen – an example

```
---> 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
```



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 another
- 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); }      Start numbering of
                                   histograms from
                                   ID=1
    // Creating histograms
    man->CreateH1("h", "Title", 100, 0., 800*MeV); } ID=1
    man->CreateH1("hh", "Title", 100, 0., 10*MeV); } ID=2

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



Fill histograms and close

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



Histograms - 1

- 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");
```



Histograms - 2

- Can change parameters of an existing histogram
- Can fill with a weight
- Methods to scale, retrieve, get rms and mean

```
G4bool SetH1Title(G4int id, const G4String& title);  
G4bool SetH1XAxisTitle(G4int id, const G4String& title);  
G4bool SetH1YAxisTitle(G4int id, const G4String& title);
```

```
G4bool FillH1(G4int id, G4double value, G4double weight =  
1.0);
```

```
G4bool ScaleH1(G4int id, G4double factor);
```

```
G4int GetH1Id(const G4String& name, G4bool warn = true) const;
```



Histograms - 3

- UI support available, to change parameters (e.g. file name) at run-time

```
/analysis/setFileName name           # Set name for the
      histograms and ntuple file
/analysis/setHistoDirName name       # Set name for the
histograms directory
/analysis/setNtupleDirName name      # Set name for the
histograms directory
/analysis/setActivation true|false  # Set activation option
/analysis/verbose level              # Set verbose level

/analysis/h1/create
      name title [nbin min max] [unit] [fcn] [binScheme]  #
Create 1D histogram
```



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); } Start numbering of
                               } ntuples from ID=1

    // Creating ntuple
    man->CreateNtuple("name", "Title"); } ID=1
    man->CreateNtupleDColumn("Eabs");
    man->CreateNtupleDColumn("Egap");
    man->FinishNtuple();

    man->CreateNtuple("name2", "title2"); } ID=2
    man->CreateNtupleIColumn("ID");
    man->FinishNtuple();
}
```



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); } ID=1,
    man->AddNtupleRow(1);                      columns 0, 1

    man->FillNtupleIColumn(2, 0, fID);
    man->AddNtupleRow(2); } ID=2,
                           column 0
}
```



Part VI: 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

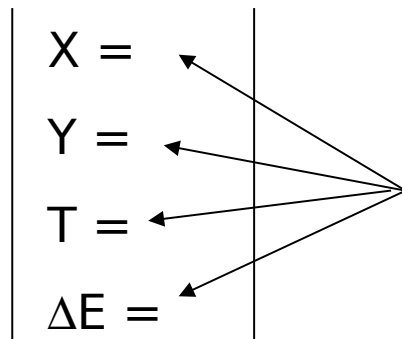
- Hit is a user-defined class which derives from the base class G4VHit. Two virtual methods
 - Draw()
 - Print()
- You can store various types of information by implementing your own concrete Hit class
- Typically, one may want to record information like
 - Position, time and ΔE of a step
 - Momentum,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



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

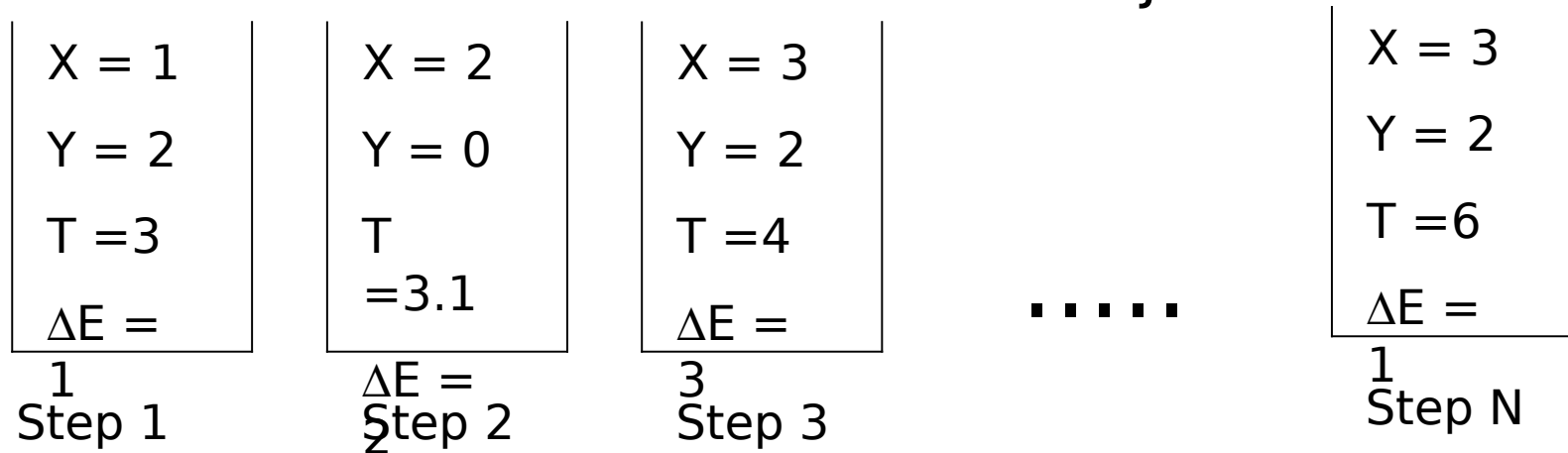
Z =
Pos =
Dir =

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





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

Z = 5.2

Pos =

(0,0,1)

Dir

=(1,1,0

)

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 = 5.4

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
    return true;                    collection
}
```



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

Z = 5.2

Pos =

(0,0,1)

Dir

=(1,1,0

)

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

)

2

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*



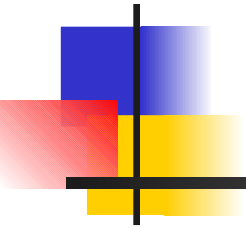
Recipe and strategy - 1

- 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



Backup



To write a new ASCII file: a recipe - 1

- Add to the include list of your class the `<fstream>` header file
 - This will allow to use the C++ libraries for stream on file
- Put into the class declaration (file .hh) an `ofstream` (=output file stream) object (or pointer):

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

 - In this way, the file object will be visible in all methods of the class
- Open the file, in the class constructor, or into a specific method:

```
myFile.open("filename.out", std::ios::trunc);
```

 - To append data to an existing file, you must specify `std::ios::app`

To write a new ASCII file: a recipe - 2

- Inside a regularly called method (e.g. inside a virtual method of an User Class), where appropriate, write your data (i.e. G4double, G4int, G4String,...) to the file, in the same fashion of G4cout:

```
if (myFile.is_open()) // Check that file is opened
{
    myFile << kineticEnergy/MeV << " " << dose << G4endl;
    ...
}
```

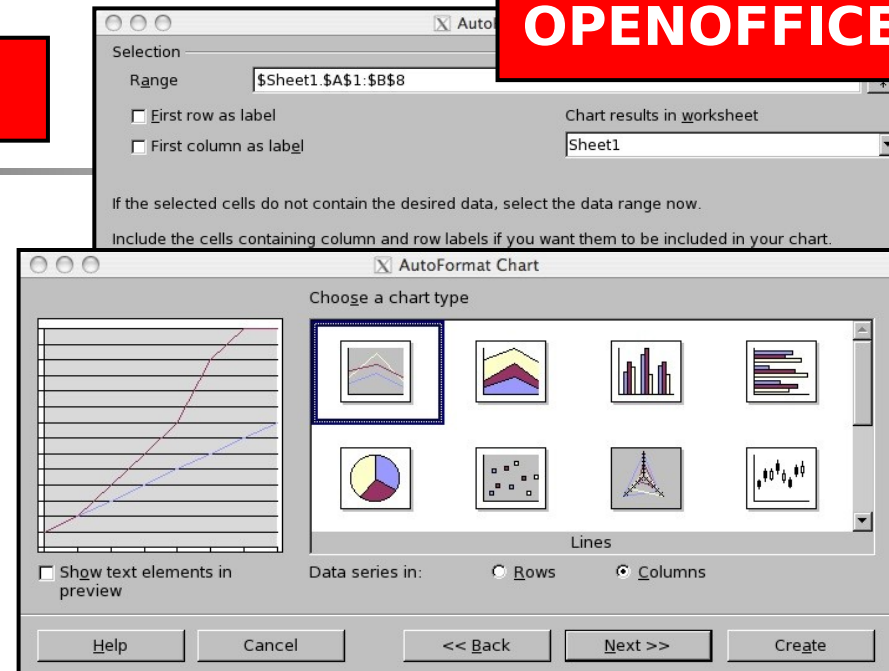
- This could be for instance the `EndOfEventAction()` of the `G4UserEventAction` user class
- Finally close the file, in the class destructor, or into a specific method:
`myFile.close();`

Plotting with tools

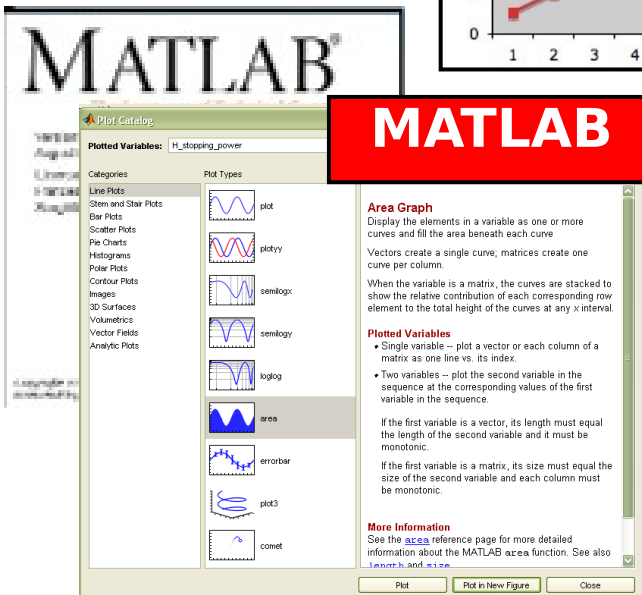
EXCEL



OPENOFFICE



MATLAB



GNU PLOT

