Geant 4

Detector Response

Anton Lechner, CERN

Acknowledgements to
J. Apostolakis, G. Cosmo, M. Asai, A. Howard

http://cern.ch/geant4

I. Geant4 scoring functionality

- Overview and repetition

Extracting information

- Once the mandatory classes (Geometry, Physics List, Primary Generator) are implemented, the Geant4 application does not yet include the functionality to extract and store information
  - A user must provide his/her own code to extract information relevant to the simulation application
- For retrieving application-specific information, Geant4 provides the following scoring functionality to users:
  - **Sensitive detector** (optional with **readout geometry**),
  - **Hits** and
  - **Hits collections**

Sensitive detector (SD)

- A logical volume becomes sensitive if it has a pointer to a sensitive detector
  - 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
- Register the sensitive detector to the SD manager
- Assign the pointer of your SD to the logical volume of your detector geometry

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

User-defined SD (1/3)

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

	Concrete Class	Base Class
Sensitive Detector	*MySensitiveDetector*	G4VSensitiveDetector
Readout Geometry	*MyReadoutGeometry* (optional)	G4VReadoutGeometry
Hit	*MyHit*	G4VHit
		Template Class
Hits Collection		G4THitsCollection<your hit class>

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

User-defined SD (2/3)

- Basic strategy to retrieve information:
  - Assume, you have already created the detector geometry
    - Shape and size (Solid) of your detector, Material
    - Logical volumes
    - Physical volumes
  - Implement a sensitive detector and assign an instance of it to the *logical volume* of your detector geometry set-up
    - Then this volume becomes sensitive
    - The sensitive detector will become “active” for each particle step, if the step starts inside this logical volume
  - Optionally: Implement a readout geometry and attach it to the sensitive detector

User-defined SD (3/3)

- Basic strategy to retrieve information (cont.)
  - Then, create hit objects in your sensitive detector using information from particle steps
    - Hit is a snapshot of the physical interaction of a track or an accumulation of interactions of tracks in the sensitive or interesting region of your detector: *You need to create hit class(es) according to your needs*
    - Use Touchable of the Readout geometry to retrieve geometrical information associated with hits
  - Store your hits in hit collections (hit collections are automatically associated to the G4Event object)
  - Finally, process the information associated with hits in user action classes (G4UserEventAction,G4UserRunAction) to obtain a summary of the event/run

Primitive Scorers (1/2)

- Alternatively, you can use a pre-defined sensitive detector (G4MultiFunctionalDetector) and primitive scorers:

	Concrete Class(es)
Sensitive Detector Primitive Scorers	G4MultiFunctionalDetector G4PSEnergyDeposit, G4PSTracklength, ...
	Template Class
Hits Collection	G4THitsMap<G4double>

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

Primitive Scorers (2/2)

- Basic strategy to retrieve information:
  - Assume, you have already created the detector geometry
    - Shape and size (Solid) of your detector, Material
    - Logical volumes
    - Physical volumes
  - Assign an instance of the Geant4 multi-functional detector (G4MultiFunctionalDetector) to the *logical volume* of your detector geometry set-up
  - Register instances of the required primitive scorers to your the multi-functional detector
  - Finally, process the content of hit maps and store the information

II. User-defined sensitive detectors

- Hits and Hits Collections
- Sensitive Detector and readout geometry

Hits and Hits Collections

Creating a customized hit class,
Storing hits in hits collections

Hit class

- Hit is a user-defined class, which derives from the base class G4VHit
- You can store various types of information by implementing your own concrete Hit class
- Typically, one keeps track of information like
  - Position and time of a step
  - Momentum and energy of the track
  - Energy deposition of the step
  - Geometrical information
  - Or any combination of above

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
};
```

Hits collection

- Once created in the sensitive detector, instances of the concrete hit class must be stored in a dedicated collection:
  - Template class G4THitsCollection, where the template parameter is the concrete hit class: G4THitsCollection<MyHit>
- The hits collections can be accessed in different phases of physics tracking:
  - At the end of each event through the G4Event object (to analyse the event and store useful information)
  - During event processing through the sensitive detector manager, G4SDManager (to perform event filtering)

Sensitive Detector

Creating a sensitive detector and a readout geometry

Sensitive Detector (SD)

- A specific feature to Geant4 is that a user can provide his/her own implementation of the detector and its response
- To create a sensitive detector, derive your own concrete class from the G4VSensitiveDetector abstract base class
  - The principal purpose of the sensitive detector is to create hit objects
  - Overload the following methods (see also next slide):
    - Initialize()
    - ProcessHits() (Invoked for each step if step starts in logical vol.)
    - EndOfEvent()

SD:

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

abstract base class

pure virtual method

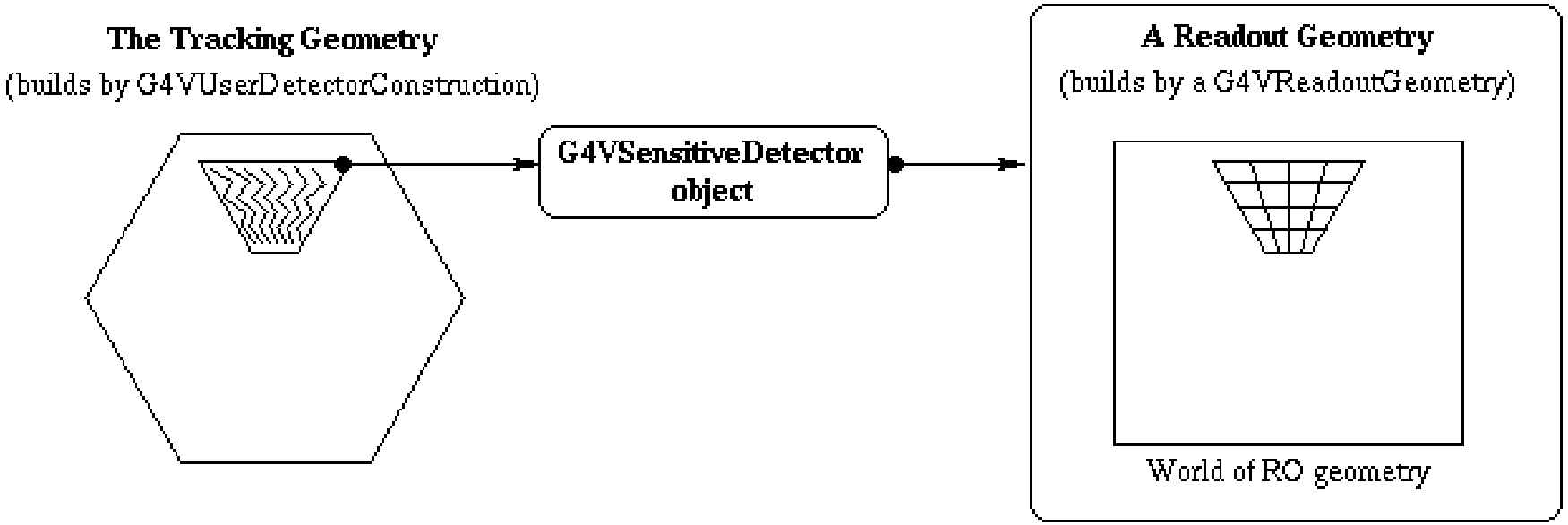
// header file: MySensitiveDetector.hh

```
#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;
};
```

your SD class

Readout geometry (1/3)

- In contrast to the “real ”geometry used for tracking, the readout geometry can be considered as a virtual and artificial geometry
  - It is defined in parallel to the real detector geometry



Readout geometry (2/3)

- Tracks will be traced in the "real" geometry, but the sensitive detector can have its own geometry for readout purposes, e.g. to find the cell the current hit belongs to. Note that
  - particle steps do not stop at RO geometry boundaries.
  - during tracking, the G4TouchableHistory of the RO geometry according to the *pre-step point position* will be provided to the sensitive detector.
- The RO geometry is optional
  - A SD does not require a readout geometry

Readout geometry (3/3)

- Derive your own concrete class from the G4VReadoutGeometry abstract base class to create a readout geometry
- The geometry set-up is done in the same way as for the “real“ tracking geometry:
  - Create solids, logical and physical volumes
  - However, the materials used in the RO geometry are dummy materials (i.e. they are not considered)
  - A sensitive detector for the RO geometry must be defined but is not used! That means, you need to declare the sensitive parts of the RO geometry by setting a non-NULL sensitive detector pointer to the logical volume.

RO geometry:

```
class  G4VReadoutGeometry {

 protected:
    virtual G4VPhysicalVolume* Build() = 0;
 ...
};
```

pure virtual method

abstract base class

your RO geometry class

```
// header file: MyReadOutGeometry.hh

#include "G4VReadoutGeometry.hh"
...
class MyReadOutGeometry : public G4VReadoutGeometry {

 public:
    MyReadoutGeometry();
    virtual ~MyReadoutGeometry();

    virtual G4VPhysicalVolume* Build();  // must return the physical world
                                         // volume of the readout geometry

 private:
  ....
 ;
```

SD and Hits (1/2)

- Using information from particle steps, a sensitive detector either
  - constructs one or more hits objects or
  - accumulates values to existing hits
- Examples:
  - The large experiments at CERN use tracker detectors. This type of detector typically registers a hit for every single step of charged particle traversing the detector. A hit associated with such a detector might thus contain:
    - Position and time of the hit,
    - Energy deposition of step,
    - the ID of the particle track

SD and Hits (2/2)

- Examples:
  - Calorimeters on the other hand accumulate the energy deposition of all steps inside a calorimeter cell for all particles traversing the cell. A calorimeter hit thus might contain:
    - Total energy deposition, the cell index
- Hit objects can be “filled” with information in the ProcessHits method of the concrete class of the SD
  - This functions has pointers to the current G4Step object and the G4TouchableHistory of the RO geometry (if defined) as arguments
  - Note that you must get the volume information from the “PreStepPoint”: See the lecture on interaction with the kernel for details

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 (Note: Execution of GetCollectionID() is time-consuming and thus this function should not be invoked at every event).

```
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.
- The principal mandate of this method is to generate hit(s) or to accumulate data to existing hit objects, by using information from the current step
- Note: Geometry information must be derived from the "PreStepPoint".

```
void 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(aHit);
  return true;
}
```

SD implementation: EndOfEvent()

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

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

}
```

Hits Collections of an event

Summary of an event

Hits Collections of an event

- A G4Event object has a G4HCofThisEvent object at the end of event processing (if event processing was successful).
  - You can get the pointer to the G4HCofThisEvent object by using the GetHCofThisEvent method of G4Event
- The G4HCofThisEvent object stores all hits collections created within the event:
  - In the end of an event, you can process the hits of hits collections contained in G4HCofThisEvent in your implementation of the EndOfEventAction method of the user event action class.

Processing hit information (1/2)

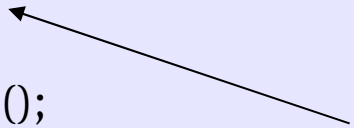
- Basic strategy of information processing:
  - Retrieve the pointer of a hits collection with the GetHC method of G4HCofThisEvent collection by using the collection index.
  - Index numbers of a hits collection are unique and do not changed for a run. The index number can be obtained by G4SDManager::GetCollectionID("detName/colName");
  - NOTE:
    - If the hits collection(s) were not created in a particular event the pointers to the collection are NULL and should thus be checked before attempting to access hit objects
    - Hits collection pointers have the type G4VHitsCollection in G4HCofThisEvent: After retrieving them, it is required to cast them to the types of individual concrete classes.

Processing hit information (2/2)

- Basic strategy of information processing (cont):
  - Loop through the entries of a hits collection to access individual hits
    - You can use the [ ] operator of the hits collection class to get the pointer to a hit object (corresponding to the specified index)
  - Retrieve the information associated with hits (and process the information if required)
  - Store the output in analysis objects
    - NOTE: In the afternoon exercises we use AIDA objects to store the simulation output

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
      }
  }
}
```



cast

Digits and Digitization

Digits

- Remember: A Hit is created when a particle step is inside a detector
- In contrast, a Digit represents the output of a detector:
  - E.g. ADC/TDC count, trigger signal, ...
  - It is created from the information from hits and/or other digits
  - Similarly as for hits, you need to implement customized digit class(es) by your own
  - The base clase of digits if G4VDigi
- Digit objects are stored in digit collections:
  - Template class G4TDigiCollection
- G4Event has a GDCofThisEvent object, that is a container class for digit collections

Note the similar concept as for hits

Digitization

- Digits are created in a digitizer module
  - Create your digitizer module by inheriting from the base class G4VDigitizerModule:
    - Overload the pure virtual method Digitize(): Create digit objects in this function and store them in digit collections
  - NOTE: In contrast to the ProcessHits() function (which creates and stores hits), the Digitize() method is NOT automatically called by the Geant4 Kernel:
    - You need to invoke this function by your own (e.g. in the user action classes)