Scoring

Makoto Asai (Jefferson Lab)

Geant4 Tutorial Course

Contents

- Introduction to scoring

- Command-based scoring

- Sensitive detector vs. primitive scorer

- Basic structure of detector sensitivity

- Sensitive detector and hit

Contents

- Introduction to scoring

- Command-based scoring

- Sensitive detector vs. primitive scorer

- Basic structure of detector sensitivity

- Sensitive detector and hit

Extract useful information

- Given geometry, physics and primary track generation, Geant4 does proper physics simulation "silently".
  - You have to do something to extract information useful to you.
- There are two ways:
  - Built-in scoring commands
    - Most commonly-used physics quantities are available.
    - Define scoring mesh, scoring probes, or assign scorers to the tracking volume
  - Assign G4VSensitiveDetector to a volume to generate "hit".
    - Use user hooks (G4UserEventAction, G4UserRunAction) to get event / run summary
  - Merging over worker threads is automatically taken care.
- You may also use other user hooks (G4UserTrackingAction, G4UserSteppingAction, etc.)
  - You have full access to almost all information
  - Straight-forward, but do-it-yourself
  - A bit more complicated for multithreaded mode

Contents

- Introduction to scoring

- **Command-based scoring**

- Sensitive detector vs. primitive scorer

- Basic structure of detector sensitivity

- Sensitive detector and hit

Command-based scoring

- Command-based scoring functionality offers the various built-in scorers for commonly-used physics quantities such as dose, flux, etc.
  - Due to small performance overhead, it does not come by default.
- To use this functionality, access to the G4ScoringManager pointer after the instantiation of G4(MT)RunManager in your *main*().

```
#include "G4ScoringManager.hh"
int main()
{
   G4RunManager* runManager = new G4MTRunManager;
   G4ScoringManager* scoringManager =
                          G4ScoringManager::GetScoringManager();
   …
```

- All the UI commands of this functionality are in /score/ directory.
- /examples/extended/runAndEvent/RE03 and /examples/advanced/gorad are good examples

extended/runAndEvent/RE03

Three types of command-based scorers
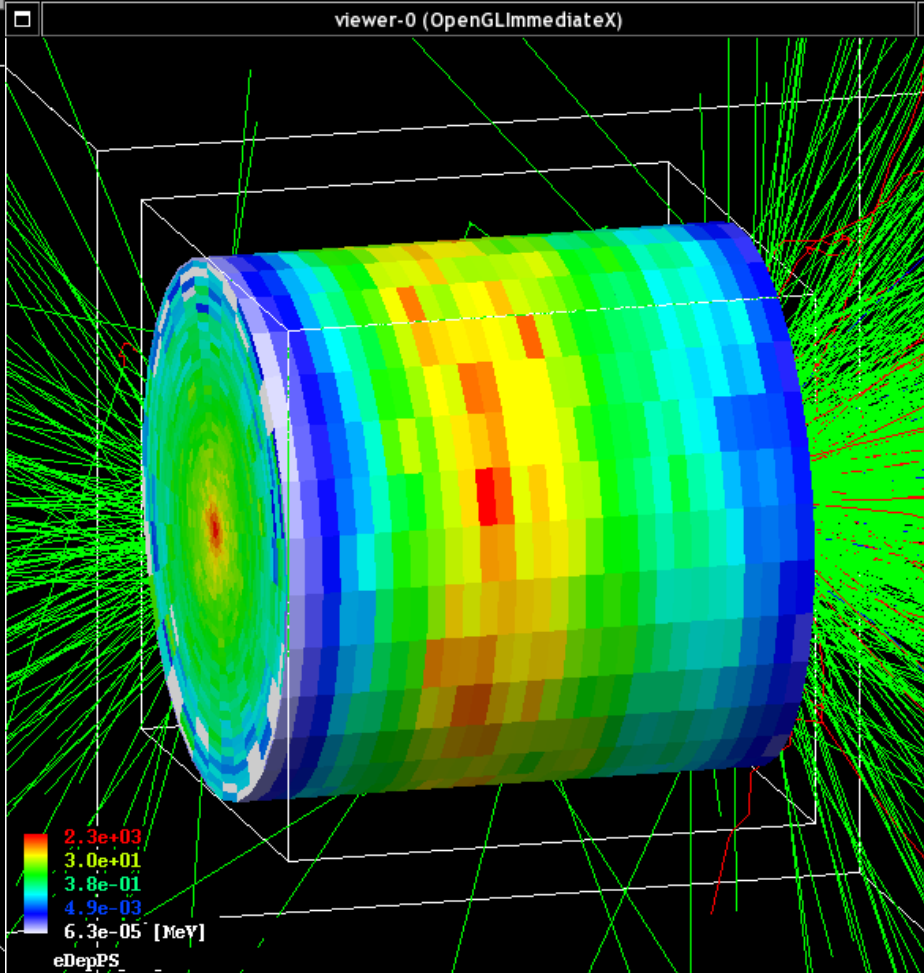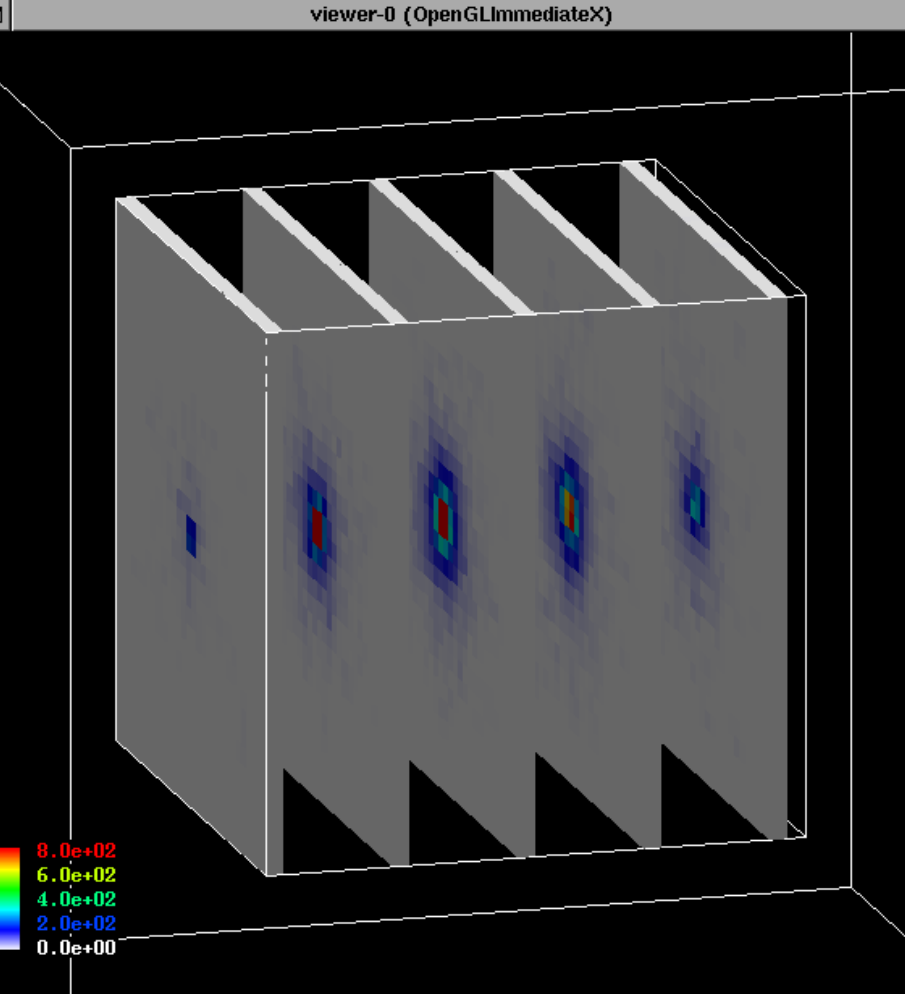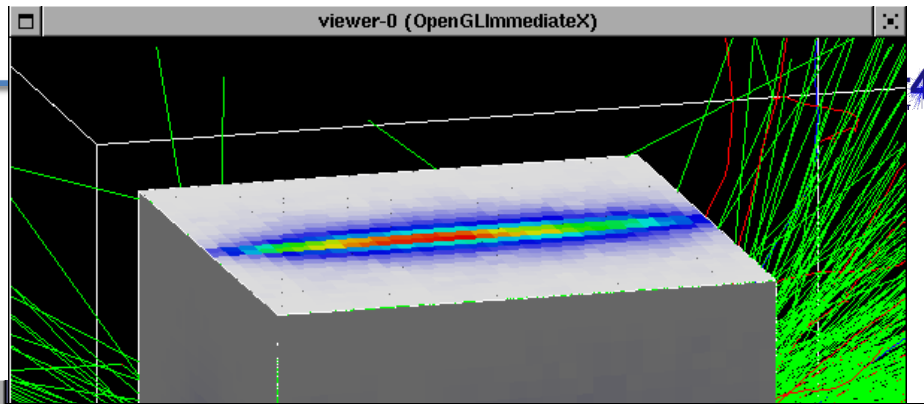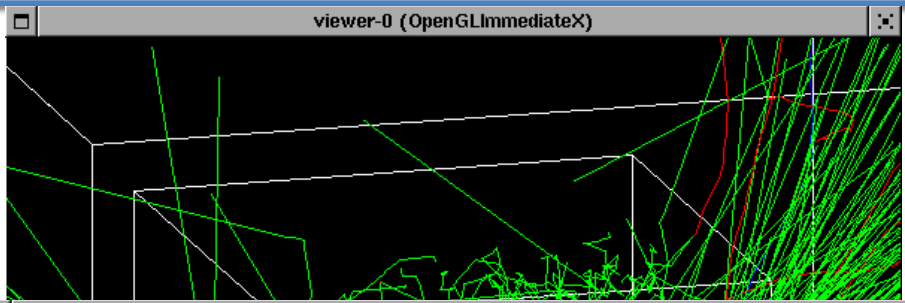
1) Scoring mesh
   – Define 3-D mesh (box or cylinder)
   – The mesh may overlap with real-world volumes
   – Assign arbitrary number of primitive scorers to mesh cell
2) Assigning scorers to a real-world logical volume
   – Declare a real-world logical volume as a detector
   – Assign arbitrary number of primitive scorers to the detector
   – If the volume is placed more than once, assigned scorers individually score for each physical volume
3) Scoring probe
   – A probe is a small cube that is located at arbitrary position. It may overlap with real-world volumes.
   – Assign arbitrary number of primitive scorers to the probe
   – If probe is placed more than once, assigned scorers individually score for each probe.

Define a scoring mesh

- To define a scoring mesh, the user must specify the followings.
  1. Shape and name of the 3D scoring mesh.
     - Currently, box and cylinder are available.
  2. Size of the scoring mesh.
     - Mesh size must be specified as "half width" like the arguments of G4Box / G4Tubs.
  3. Number of bins for each axes.
     - Note that too many bins causes immense memory consumption.
  4. Specify position and rotation of the mesh.
     - If not specified, the mesh is positioned at the center of the world volume without rotation.

  ```
  # define scoring mesh
  /score/create/boxMesh boxMesh_1
  /score/mesh/boxSize 100. 100. 100. cm
  /score/mesh/nBin 30 30 30
  /score/mesh/translate/xyz 0. 0. 100. cm
  ```

- The mesh geometry is completely independent to the real material geometry.

Scoring quantities

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

/score/quantity/xxxxx   <scorer_name> <unit>

Filter

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

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

Same primitive scorers with different filters may be defined.

/score/close   ⬅   **Close the mesh when defining scorers is done.**

Drawing a score

- Projection

  /score/drawProjection <mesh_name> <scorer_name> <color_map>

- Slice

  /score/drawColumn <mesh_name> <scorer_name> <plane> <column>

    <color_map>

- Available only for box or cylindrical mesh.

- Color map

 - By default, linear and log-scale color maps are available.

  - Minimum and maximum values can be defined by

    /score/colorMap/setMinMax command. Otherwise, min and max values are

    taken from the current score.

Write scores to a file

- Single score

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

- All scores

  /score/dumpAllQuantitiesToFile <mesh_name> <file_name>

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

More than one scoring meshes

- You may define more than one scoring mesh.
  - And, you may define arbitrary number of primitive scorers to each scoring mesh.
- Mesh volumes may overlap with other meshes and/or with mass geometry.
- A step is limited on every boundary.
- Please be cautious of too many meshes, too granular meshes and/or too many primitive scorers.
  - Memory consumption
  - Computing speed
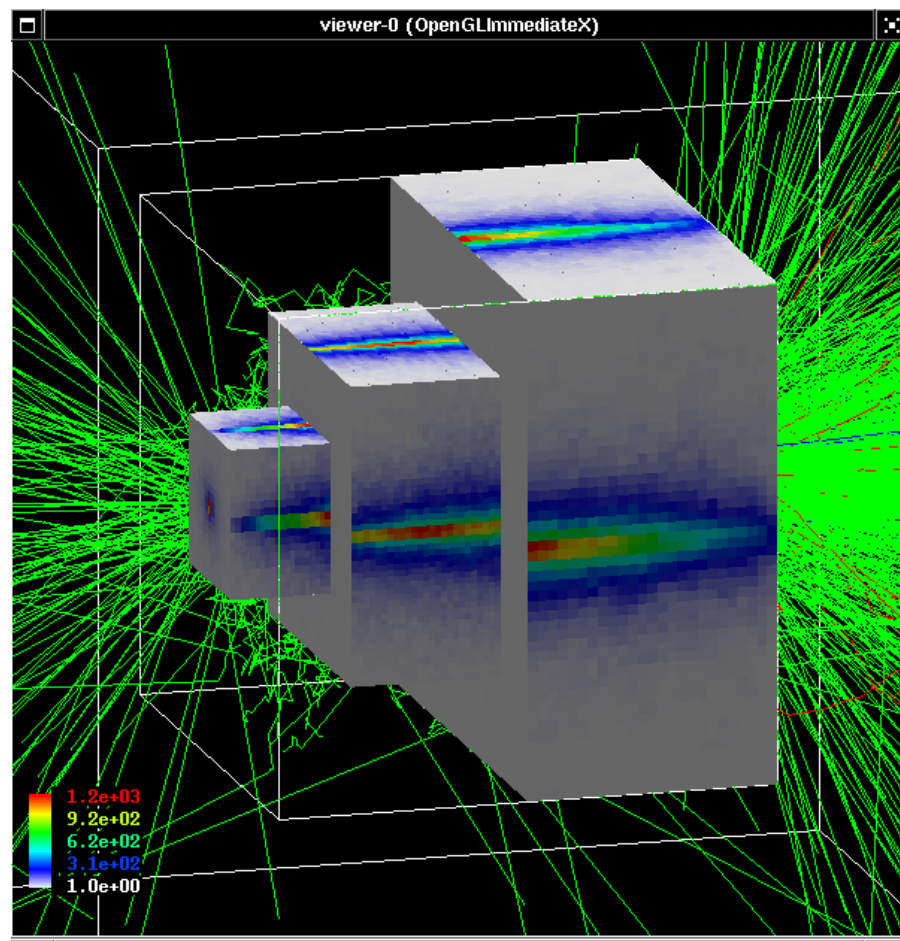
Define scorer to a tracking volume
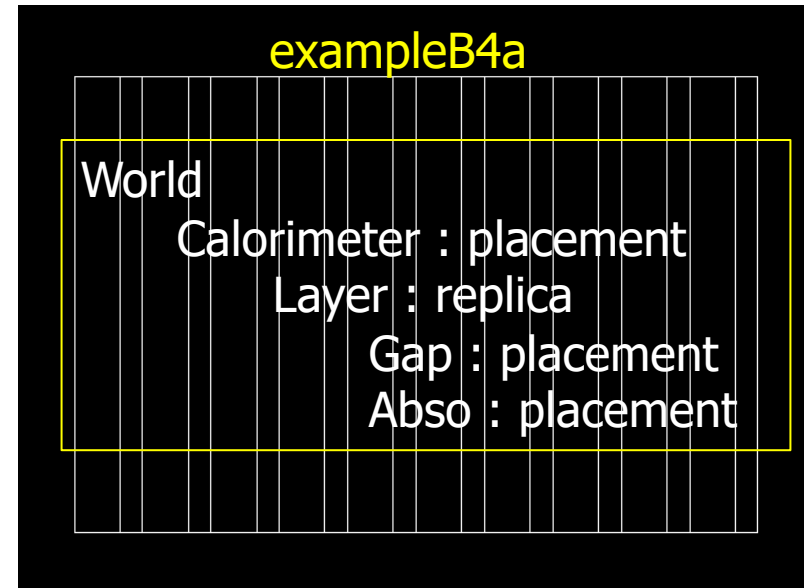
- Define a scorer to a logical volume.

    `/score/create/realWorldLogVol <LV_name> <anc_lvl>`

- One can define arbitrary scoring quantities and filters.
    - Same recipe as scoring mesh.
    - Scores are automatically merged other worker threads and written to a file.
    - Drawing is not yet supported.
- All physical volumes that share the same *<LV_name>* have the same primitive scorers but score separately.
    - Copy number of the physical volume is the index.
    - If the physical volume is placed only once, but its (grand-)mother volume is replicated, use the *<anc_lvl>* parameter to indicate the ancestor level where the copy number should be taken.

Command-based real-world scorer
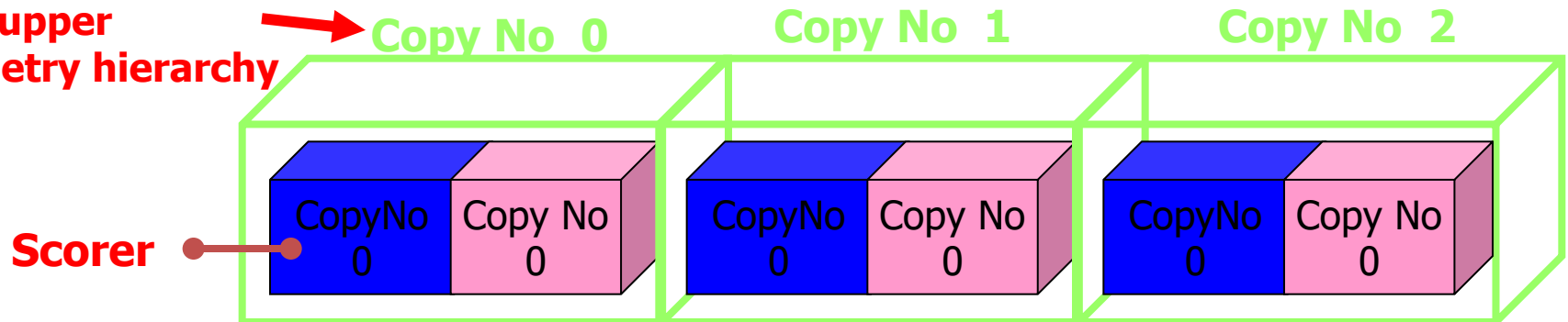
- Do not use this /score/create/realWorldLogVol command to a mother logical volume.

  – For example of this exampleB4, "Layer" is fully filled with "Gap" and "Abso" daughter volumes. You won't see any energy deposition in "Layer" volume.

/score/create/realWorldLogVol *Gap 1*
/score/quantity/energyDeposit *eDepGap* MeV
/score/create/realWorldLogVol *Abso 1*
/score/quantity/energyDeposit *eDepAbs* MeV
/score/close

exampleB4a

World
    Calorimeter : placement
      Layer : replica
        Gap : placement
        Abso : placement

If this is not set, given "Gap" and "Abso" are placed with copy number 0, energy deposition and track length are accumulated for all layers.

Command-based probe scorer

- User may locate scoring "probes" at arbitrary locations. A "probe" is a virtual cube, to which any Geant4 primitive scorers could be assigned.

- Given these probes are in an artificial "parallel world", probes may overlap to the volumes defined in the mass geometry.

- If probes are located more than once, all probes have the same scorers but score separately.

- In addition, the user may optionally set a material to the probe. Once a material is set to the probe, it overwrites the material(s) defined in the mass geometry when a track enters the probe cube.

  – Because of this overwriting, physics quantities that depend on material or density, e.g. energy deposition, would be measured accordingly to the specified material.

  – This affects to the simulation results. Use sparsely.

- Once a probe is defined, user can associate arbitrary number of primitive scorers and filters like the conventional scoring mesh.
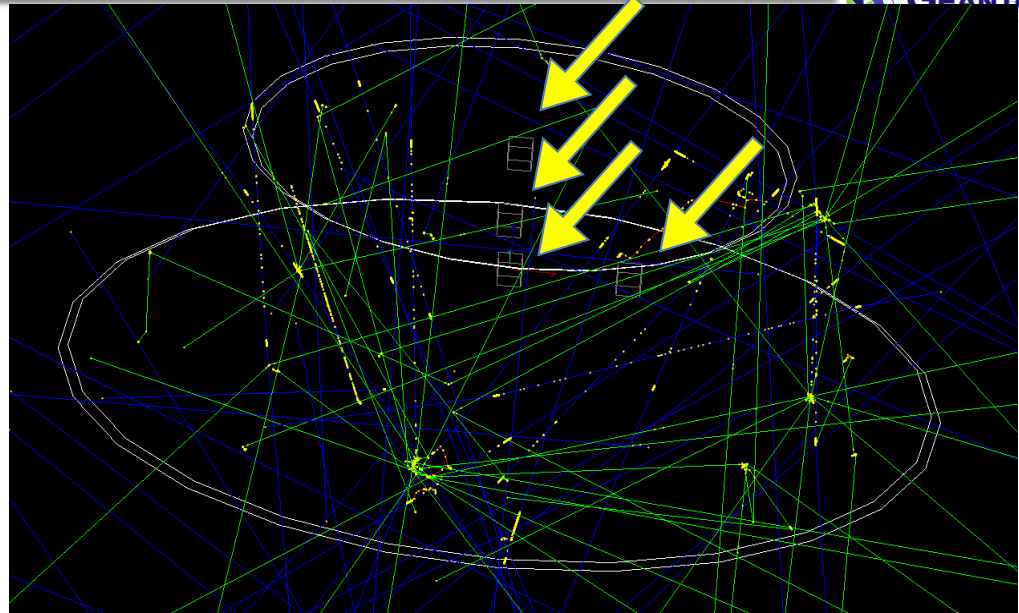
Scoring probe

/score/create/probe Probes 5. cm
/score/probe/material G4_WATER
/score/probe/locate 0. 0. 0. cm
/score/probe/locate 25. 0. 0. cm
/score/probe/locate 0. 25. 0. cm
/score/probe/locate 0. 0. 25. cm
/score/quantity/energyDeposit eDep MeV
/score/quantity/doseDeposit dose mGy
/score/quantity/volumeFlux volFlx
/score/quantity/volumeFlux protonFlux
/score/filter/particle protonFilter proton
/score/close
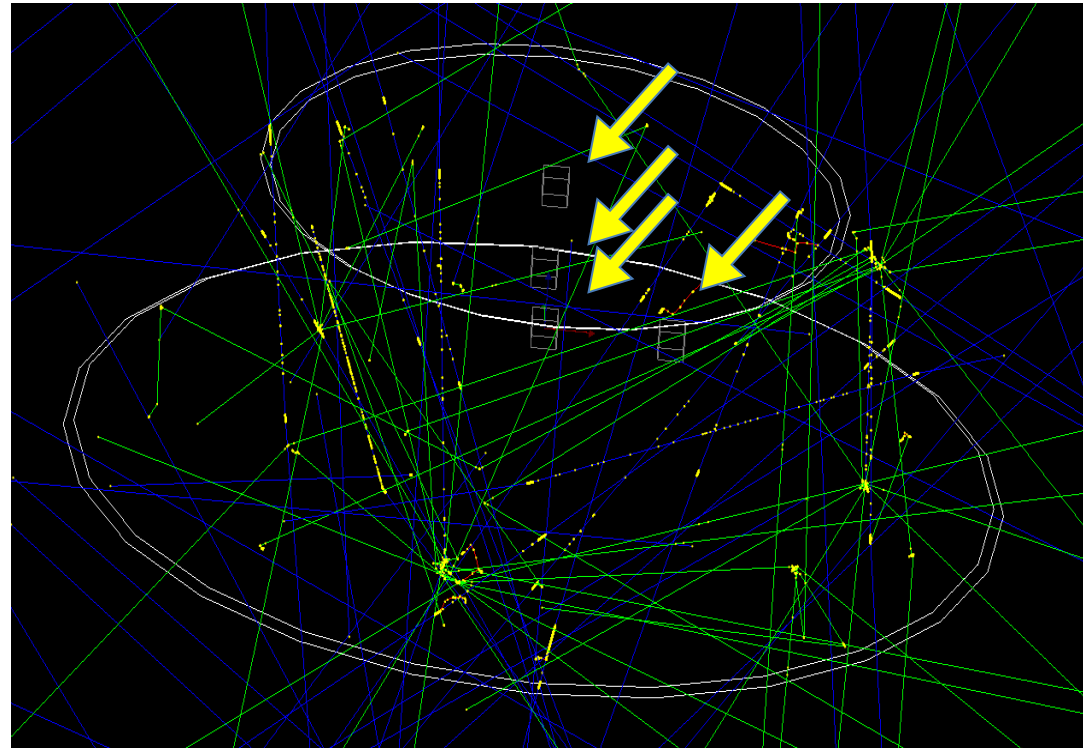
Note: To visualize the probes defined in a parallel world, the following command is required.
/vis/drawVolume worlds

1-D histogram directly filled by a primitive scorer

- Through a newly introduced interface class (G4TScoreHistFiller) a primitive scorer can directly fill a 1-D histogram defined by G4Analysis.

  – Track-by-track or step-by-step filling allows command-based histogramming such as energy spectrum.

- G4TScoreHistFiller template class must be instantiated in the user's code with his/her choice of analysis data format.

```
#include "G4AnalysisManager.hh"
#include "G4TScoreHistFiller.hh"
auto analysisManager = G4AnalysisManager::Instance();
analysisManager–>SetDefaultFileType("root");
auto histFiller = new G4TScoreHistFiller<G4AnalysisManager>;
```

- Primitive scorer must be defined in advance to setting a histogram.

- Histogram must be defined through /analysis/h1/create command in advance to setting it to a primitive scorer.

- This functionality is available only for primitive scorers defined in real-world scorer or probe scorer.

  – Not available for box or cylindrical mesh scorer due to memory consumption concern.
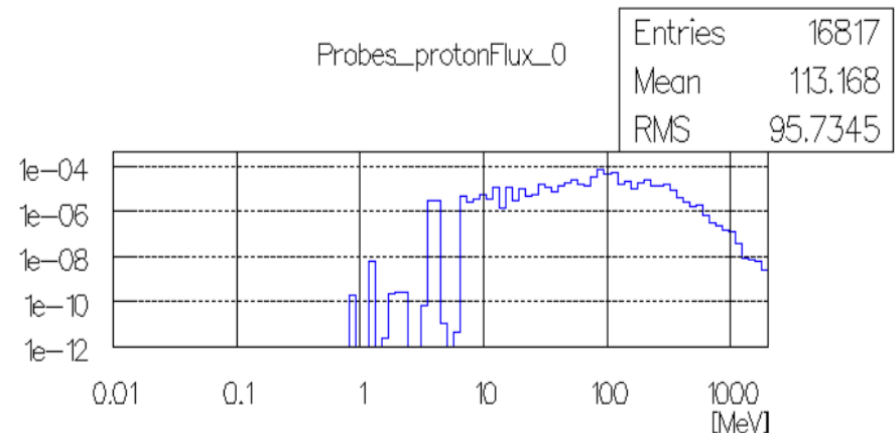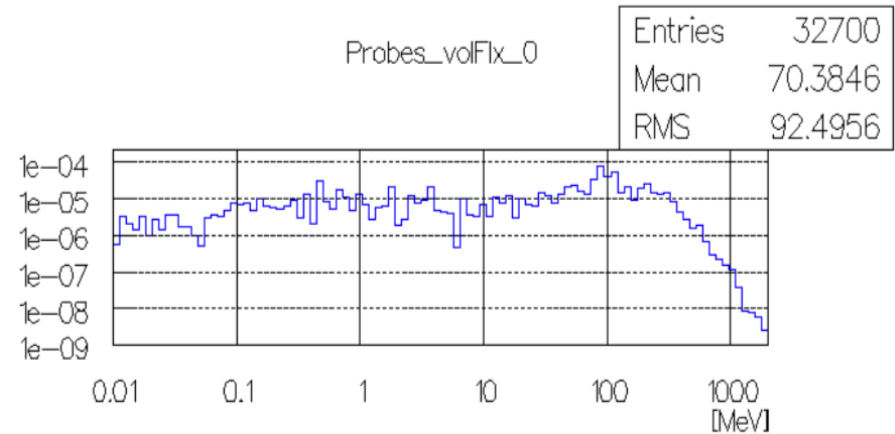
1-D histogram directly filled by a primitive scorer

/score/create/probe Probes 5. cm

/score/probe/locate 0. 0. 0. cm

/score/quantity/volumeFlux volFlux

/score/quantity/volumeFlux protonFlux

/score/filter/particle protonFilter proton

/score/close

/analysis/h1/create volFlux Probes_volFlux

                100 0.01 2000. MeV ! log

/score/fill1D 1 Probes volFlux

/analysis/h1/create protonFlux Probes_protonF

                100 0.01 2000. MeV ! log

/score/fill1D 2 Probes protonFlux

N.B. If probe is placed more than once, *fill1D* command should be called to each *copyNo.*

    /score/fill1D 1 Probes volFlux 0

N.B. Direct histogram filling is available to the selected kinds of primitive scorers that are derived from *G4VPrimitivePlotter* abstract base class.

– E.g. volumeFlux, flatSurfaceFlux, doseDeposit, energyDeposit

Contents

- Introduction to scoring

- Command-based scoring

- **Sensitive detector vs. primitive scorer**

- Basic structure of detector sensitivity

- Sensitive detector and hit

Extract useful information

- Given geometry, physics and primary track generation, Geant4 does proper physics simulation "silently".
  - You have to do something to extract information useful to you.
- There are two ways:
  - Built-in scoring commands
    - Most commonly-used physics quantities are available.
    - Define scoring mesh, scoring probes, or assign scorers to the tracking volume
  - Assign G4VSensitiveDetector to a volume to generate "hit".
    - Use user hooks (G4UserEventAction, G4UserRunAction) to get event / run summary
  - Merging over worker threads is automatically taken care.
- You may also use other user hooks (G4UserTrackingAction, G4UserSteppingAction, etc.)
  - You have full access to almost all information
  - Straight-forward, but do-it-yourself
  - A bit more complicated for multithreaded mode

Sensitive detector vs. primitive scorer

Sensitive detector

- You have to implement your own detector and hit classes.
- One hit class can contain many quantities. A hit can be made for each individual step, or accumulate quantities.
- Basically, one hits collection is made per one detector.

- Hits collection is relatively compact.

Primitive scorer

- Many scorers are provided by Geant4. You can add your own.
- Each scorer accumulates one quantity for an event.

- G4MultiFunctionalDetector creates many collections (maps), i.e. one collection per one scorer.
- Keys of maps are redundant for scorers of same volume.

I would suggest to :

▸ Use primitive scorers

  ▸ if you are not interested in recording each individual step but accumulating some physics quantities for an event or a run, and

  ▸ if you do not have to have too many scorers.

▸ Otherwise, consider implementing your own sensitive detector.

Contents

- Introduction to scoring

- Command-based scoring

- Sensitive detector vs. primitive scorer

- **Basic structure of detector sensitivity**

- Sensitive detector and hit

Sensitive detector

- A G4VSensitiveDetector object can be assigned to G4LogicalVolume.

- In case a step takes place in a logical volume that has a G4VSensitiveDetector object, this G4VSensitiveDetector is invoked with the current G4Step object.
  - You can implement your own sensitive detector classes, or use scorer classes provided by Geant4.

```
Stepping          Physics          Particle          Step          Track          Logical          Sensitive
Manager           Process          Change                                          Volume           Detector

GetPhysicalInteractionLength
  SelectShortest
      <
        DoIt  >      Fill  >

              Update                        >
                    Update                              >
                      IsSensitive                                 >
                        GenerateHits                                          >
```
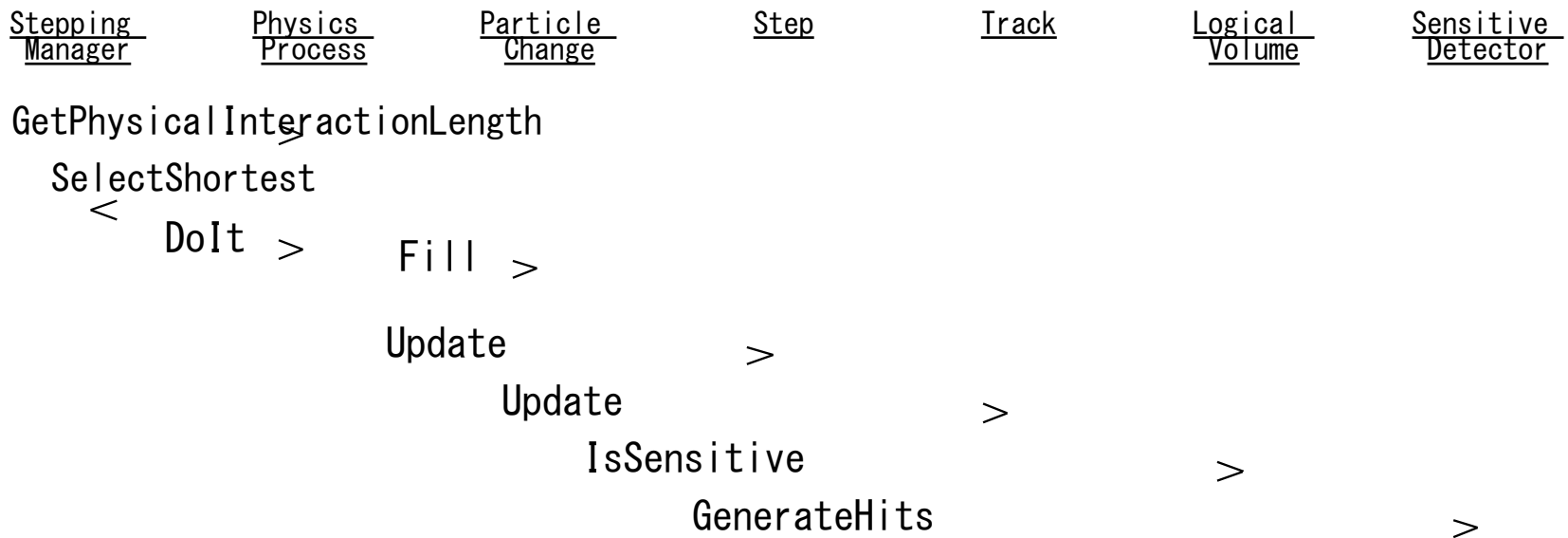
Defining a sensitive detector

- Basic strategy

  In the ConstructSDandField() method of your detector construction class

  ```
  G4VSensetiveDetector* pSensetiveDet
      = new MyDetector("/mydet");

  G4SDManager::GetSDMpointer()
                  ->AddNewDetector(pSensetiveDet);

  SetSensitiveDetector("myLogicalVolume",pSensetiveDet);
  ```

- Each detector object must have a unique name.

  - More than one logical volumes can share one detector object.

  - More than one detector objects can be instantiated from one detector class with different detector name.

  - One logical volume cannot have more than one detector objects. But, one detector object can generate more than one kinds of hits.

    - e.g. a double-sided silicon micro-strip detector can generate hits for each side separately.

Class diagram

Concrete class provided by G4

Abstract base class provided by G4

Template class provided by G4

User's class

G4LogicalVolume

G4Event

has

0..1

G4VSensitiveDetector

kind of

G4MultiFunctionalDetector

1

G4HCofThisEvent

n

G4VHitsCollection

G4THitsCollection

userSensitiveDetector

n

G4VPrimitiveSensitivity

n

G4THitsMap

G4VHit

userHitsCollection
or   userHitsMap

n

G4PSDoseScorer

hits map

n

userHit

1

Hits collection, hits map
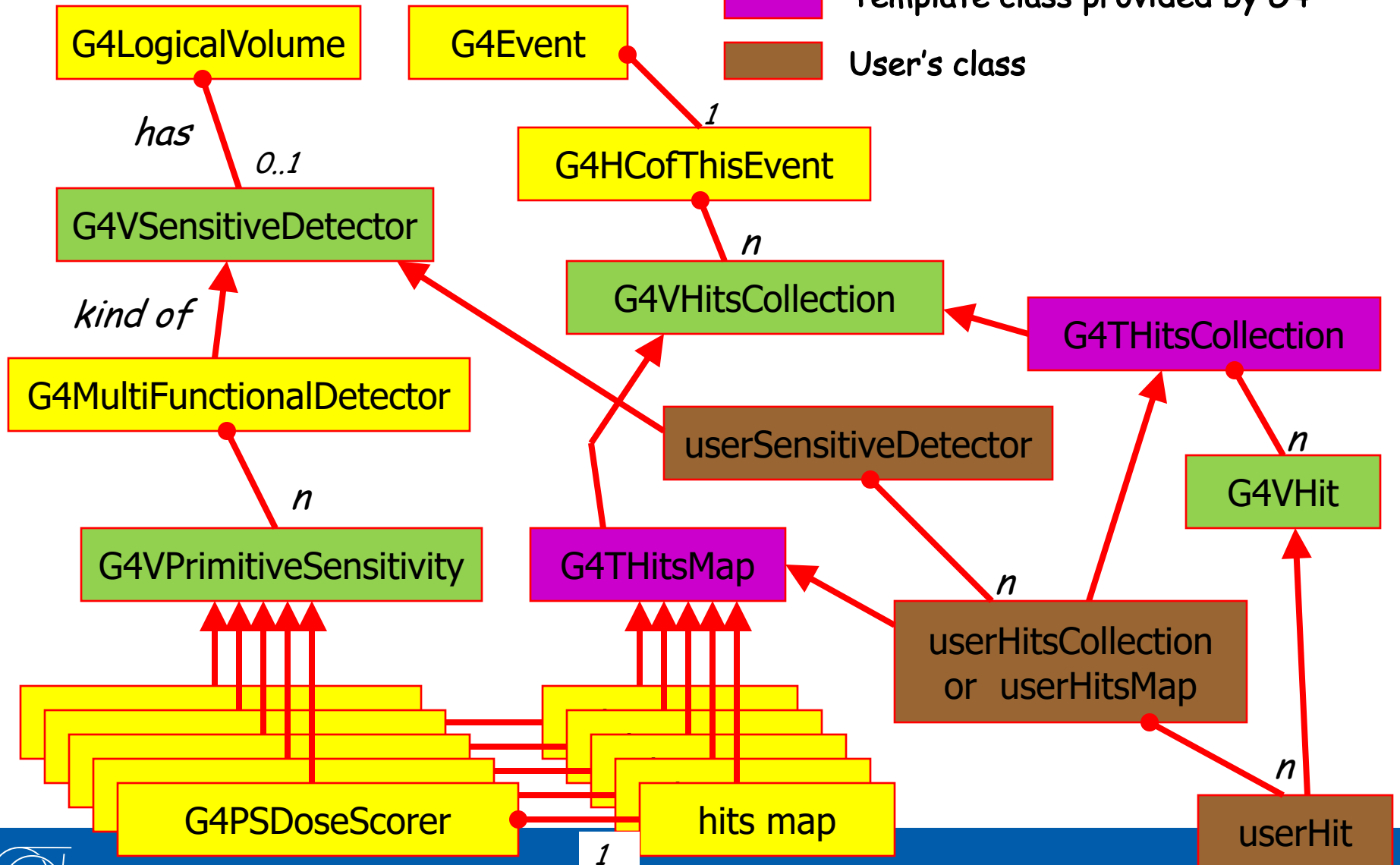
- G4VHitsCollection is the common abstract base class of both G4THitsCollection and G4THitsMap.

- G4THitsCollection is a template vector class to store pointers of objects of one concrete hit class type.

  - A hit class (deliverable of G4VHit abstract base class) should have its own identifier (e.g. cell ID).

  - In other words, G4THitsCollection requires you to implement your hit class.

- G4THitsMap is a template map class so that it stores keys (typically cell ID, i.e. copy number of the volume) with pointers of objects of one type.

  - Objects may not be those of hit class.

    - All of currently provided scorer classes use G4THitsMap with simple double (for an event) and G4StatDouble (for a run).

  - Since G4THitsMap is a template, it can be used by your sensitive detector class to store hits.

Contents

- Introduction to scoring

- Command-based scoring

- Sensitive detector vs. primitive scorer

- Basic structure of detector sensitivity

- Sensitive detector and hit

Hit class

- Hit is a user-defined class derived from G4VHit.

- You can store various types information by implementing your own concrete Hit class. For example:

    - Position and time of the step

    - Momentum and energy of the track

    - Energy deposition of the step

    - Geometrical information

    - or any combination of above

- Hit objects of a concrete hit class must be stored in a dedicated collection which is instantiated from G4THitsCollection template class.

- The collection will be associated to a G4Event object via G4HCofThisEvent.

- Hits collections are accessible

    - through G4Event at the end of event.

        - to be used for analyzing an event

    - through G4SDManager during processing an event.

        - to be used for event filtering.

Implementation of Hit class

```cpp
#include "G4VHit.hh"
#include "G4Allocator.hh"
class MyHit : public G4VHit
{
  public:
      MyHit(some_arguments);
      inline void*operator new(size_t);
      inline void operator delete(void *aHit);
      virtual ~MyHit();
      virtual void Draw();
      virtual void Print();
  private:
      // some data members
  public:
      // some set/get methods
};

#include "G4THitsCollection.hh"
typedef G4THitsCollection<MyHit> MyHitsCollection;
```

G4Allocator

- Instantiation / deletion of an object is a heavy operation.
  - It may cause a performance concern, in particular for objects that are frequently instantiated / deleted.
    - E.g. hit, trajectory and trajectory point classes
- G4Allocator is provided to ease such a problem.
  - It allocates a chunk of memory space for objects of a certain class.
- Please note that G4Allocator works only for a concrete class.
  - It works only for "final" class.
  - Do NOT use G4Allocator for abstract base class.
- G4Allocator must be thread-local. Also, objects instantiated by G4Allocator must be deleted within the same thread.
  - Such objects may be referred by other threads.

Use of G4Allocator

```
MyHit.hh
    #include "G4VHit.hh"
    #include "G4Allocator.hh"
    class MyHit : public G4VHit
    {
      public:
          MyHit(some_arguments);
          inline void*operator new(size_t);
          inline void operator delete(void *aHit);
      . . .
    };
    extern G4ThreadLocal G4Allocator<MyHit>* MyHitAllocator;
    inline void* MyHit::operator new(size_t)
    {
        if (!MyHitAllocator)
            MyHitAllocator = new G4Allocator<MyHit>;
        return (void*)MyHitAllocator->MallocSingle();
    }
    inline void MyHit::operator delete(void* aHit)
    { MyHitAllocator->FreeSingle((MyHit*)aHit); }

MyHit.cc
    #include "MyHit.hh"
    G4ThreadLocal G4Allocator<MyHit>* MyHitAllocator = 0;
```

Sensitive Detector class

- Sensitive detector is a user-defined class derived from G4VSensitiveDetector.

```cpp
#include "G4VSensitiveDetector.hh"
#include "MyHit.hh"
class G4Step;
class G4HCofThisEvent;
class MyDetector : public G4VSensitiveDetector
{
  public:
      MyDetector(G4String name);
      virtual ~MyDetector();
      virtual void Initialize(G4HCofThisEvent*HCE);
      virtual G4bool ProcessHits(G4Step*aStep,
                            G4TouchableHistory*ROhist);
      virtual void EndOfEvent(G4HCofThisEvent*HCE);
  private:
      MyHitsCollection * hitsCollection;
      G4int collectionID;
};
```

Sensitive detector

- A tracker detector typically generates a hit for every single step of every single (charged) track.
    - A tracker hit typically contains
        - Position and time
        - Energy deposition of the step
        - Track ID
- A calorimeter detector typically generates a hit for every cell, and accumulates energy deposition in each cell for all steps of all tracks.
    - A calorimeter hit typically contains
        - Sum of deposited energy
        - Cell ID
- You can instantiate more than one objects for one sensitive detector class. Each object should have its unique detector name.
    - For example, each of two sets of detectors can have their dedicated sensitive detector objects. But, the functionalities of them are exactly the same to each other so that they can share the same class. See examples/basic/B5 as an example.

Step

- Step has two points and also "delta" information of a particle (energy loss on the step, time-of-flight spent by the step, etc.).

- Each point knows the volume (and 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 next volume.

- Note that you must get the volume information from the "PreStepPoint".
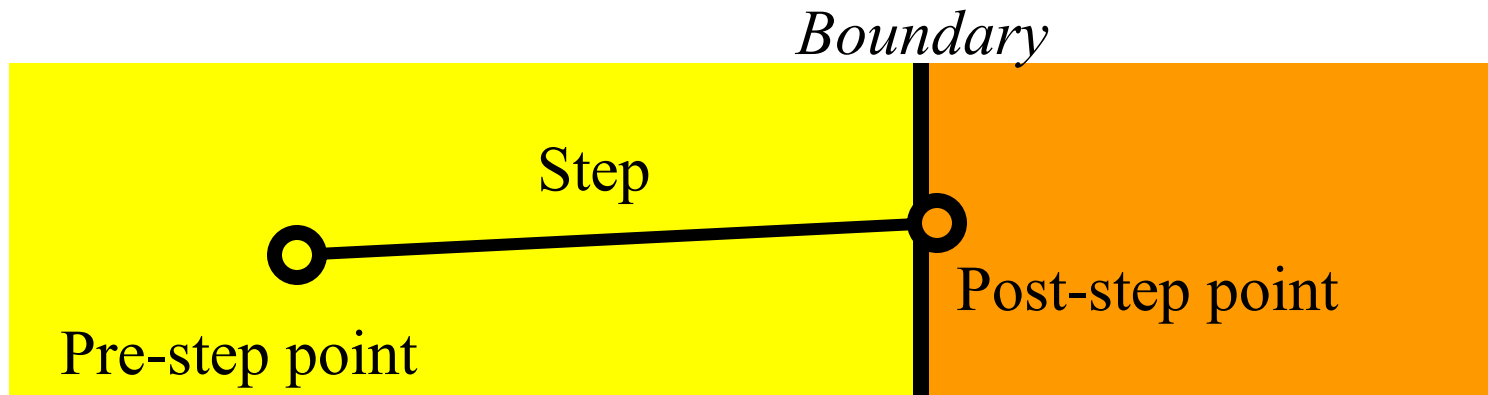
Step point and touchable

- As mentioned already, G4Step has two G4StepPoint objects as its starting and ending points. All the geometrical information of the particular step should be taken from "PreStepPoint".
  - Geometrical information associated with G4Track is identical to "PostStepPoint".
- Each G4StepPoint object has
  - Position in world coordinate system
  - Global and local time
  - Material
  - G4TouchableHistory for geometrical information
- G4TouchableHistory object is a vector of information for each geometrical hierarchy.
  - copy number
  - transformation / rotation to its mother

- G4TouchableHistory has information of geometrical hierarchy of the point.

```
G4Step* aStep;

G4StepPoint* preStepPoint = aStep->GetPreStepPoint();

G4TouchableHistory* theTouchable =

    (G4TouchableHistory*)(preStepPoint->GetTouchable());

G4int copyNo = theTouchable->GetVolume()->GetCopyNo();

G4int motherCopyNo

        = theTouchable->GetVolume(1)->GetCopyNo();

G4int grandMotherCopyNo

        = theTouchable->GetVolume(2)->GetCopyNo();

G4ThreeVector worldPos = preStepPoint->GetPosition();

G4ThreeVector localPos = theTouchable->GetHistory()

    ->GetTopTransform().TransformPoint(worldPos);
```
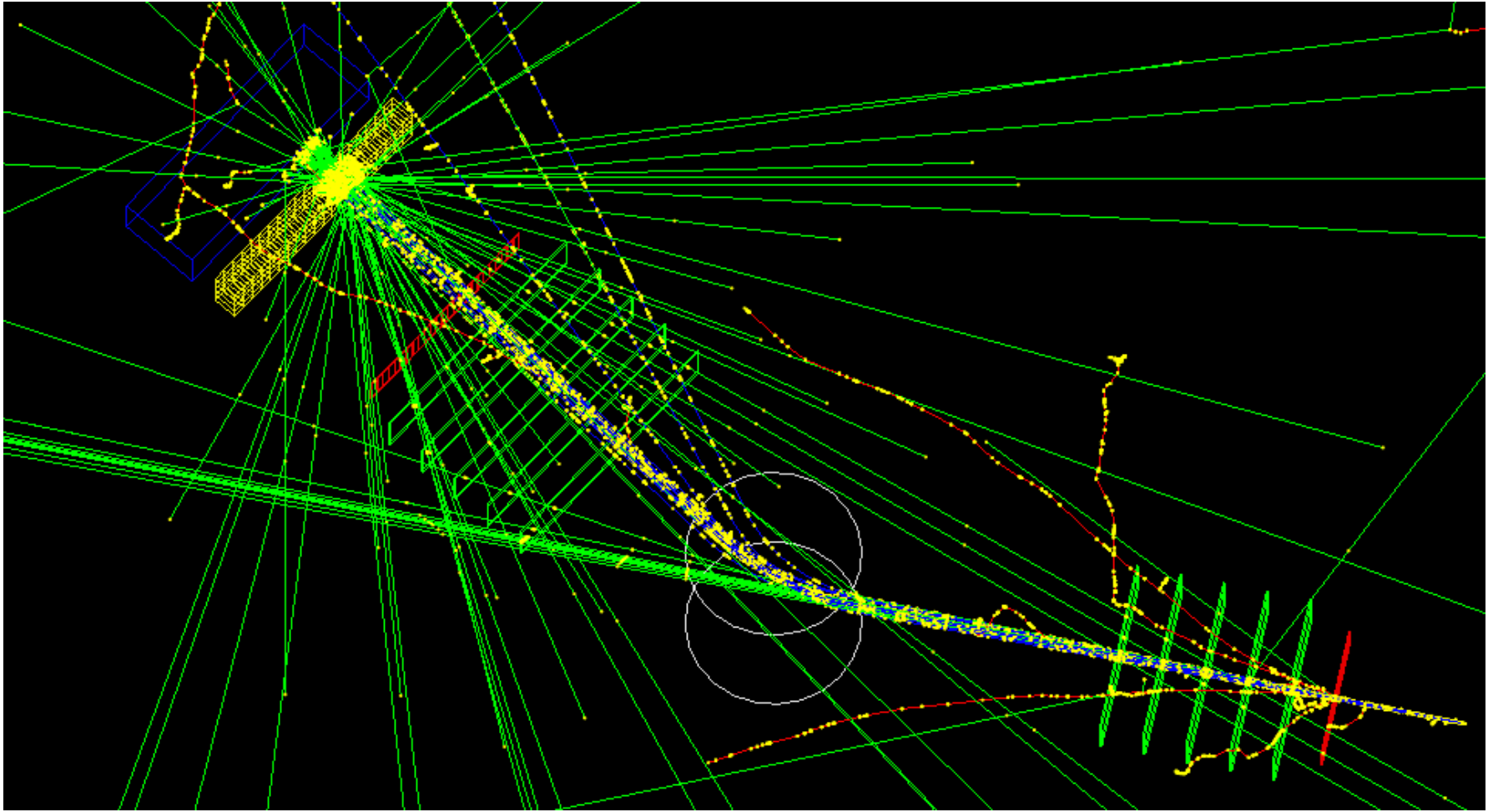
Best example for sensitive detector

- Refer to **/examples/basic/B5** example, which has several sensitive detectors.

Implementation of Sensitive Detector - 1

```cpp
MyDetector::MyDetector(G4String detector_name)
                :G4VSensitiveDetector(detector_name),
                 collectionID(-1)
{

  collectionName.insert("collection_name");

}
```

- In the constructor, define the name of the hits collection which is handled by this sensitive detector

- In case your sensitive detector generates more than one kinds of hits (e.g. anode and cathode hits separately), define all collection names.

Implementation of Sensitive Detector - 2

```
void MyDetector::Initialize(G4HCofThisEvent*HCE)
{
  if(collectionID<0) collectionID = GetCollectionID(0);
  hitsCollection = new MyHitsCollection
        (SensitiveDetectorName,collectionName[0]);
  HCE->AddHitsCollection(collectionID,hitsCollection);
}
```

- Initialize() method is invoked at the beginning of each event.
- Get the unique ID number for this collection.
  – GetCollectionID() is a heavy operation. It should not be used for every events.
  – GetCollectionID() is available after this sensitive detector object is constructed and registered to G4SDManager. Thus, this method cannot be invoked in the constructor of this detector class.
- Instantiate hits collection(s) and attach it/them to G4HCofThisEvent object given in the argument.
- In case of calorimeter-type detector, you may also want to instantiate hits for all calorimeter cells with zero energy depositions, and insert them to the collection.

Implementation of Sensitive Detector - 3

```
G4bool MyDetector::ProcessHits
  (G4Step*aStep,G4TouchableHistory*ROhist)
{
  MyHit* aHit = new MyHit();
  ...
  // some set methods
  ...
  hitsCollection->insert(aHit);
  return true;
}
```

- This ProcessHits() method is invoked for every steps in the volume(s) where this sensitive detector is assigned.

- In this method, generate a hit corresponding to the current step (for tracking detector), or accumulate the energy deposition of the current step to the existing hit object where the current step belongs to (for calorimeter detector).

- Don't forget to get geometry information (e.g. copy number) from "PreStepPoint".

- Currently, returning boolean value is not used.

Implementation of Sensitive Detector - 4

```
void MyDetector::EndOfEvent(G4HCofThisEvent*HCE)

{;}
```

- This method is invoked at the end of processing an event.
  - It is invoked even if the event is aborted.
  - It is invoked before UserEndOfEventAction.

G4HCofThisEvent

- A G4Event object has a G4HCofThisEvent object at the end of (successful) event processing. G4HCofThisEvent object stores all hits collections made within the event.

  - Pointer(s) to the collections may be NULL if collections are not created in the particular event.

  - Hits collections are stored by pointers of G4VHitsCollection base class. Thus, you have to cast them to types of individual concrete classes.

  - The index number of a Hits collection is unique and unchanged for a run. The index number can be obtained by

    G4SDManager::GetCollectionID("*detName/colName*"**);**

    - This is a heavy operation. Do it only once and keep the ID.

Use of G4HCofThisEvent in UserEventAction

```cpp
MyEventAction::MyEventAction() : fCollectionId(-1) {;}

void MyEventAction::EndOfEventAction(const G4Event* event)

{

  if(fCollectionId<0)

  { fCollectionId = G4SDManager::GetSDMpointer()

                    ->GetCollectionID("detName/colName"); }

  MyHitsCollection * hitsCollection = static_cast<MyHitsCollection*>

                    (event->GetHCofThisEvent()->GetHC(fCollectionId));

  if(hitsCollection != nullptr)

  {

    G4cout << "Number of hits : " <<  hitsCollection->entries() << G4endl;

    for(auto aHit : *hitsCollection)

    { aHit->Print(); }

  }
}
```