



Scoring II

Makoto Asai (Jefferson Lab)
Geant4 Tutorial Course

Contents



- Sensitive detector vs. primitive scorer
- Basic structure of detector sensitivity
- Sensitive detector and hit
- Add a new scorer / filter



Contents



- Sensitive detector vs. primitive scorer
- Basic structure of detector sensitivity
- Sensitive detector and hit
- Add a new scorer / filter



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 three ways:
 - Built-in scoring commands
 - Most commonly-used physics quantities are available.
 - Use scorers in the tracking volume
 - Create scores for each event
 - Create own Run class to accumulate scores
 - Assign **G4VSensitiveDetector** to a volume to generate “hit”.
 - Use user hooks (G4UserEventAction, G4UserRunAction) to get event / run summary
- You may also use user hooks (G4UserTrackingAction, G4UserSteppingAction, etc.)
 - You have full access to almost all information
 - Straight-forward, but do-it-yourself

In Kernel 2 talk

This talk

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

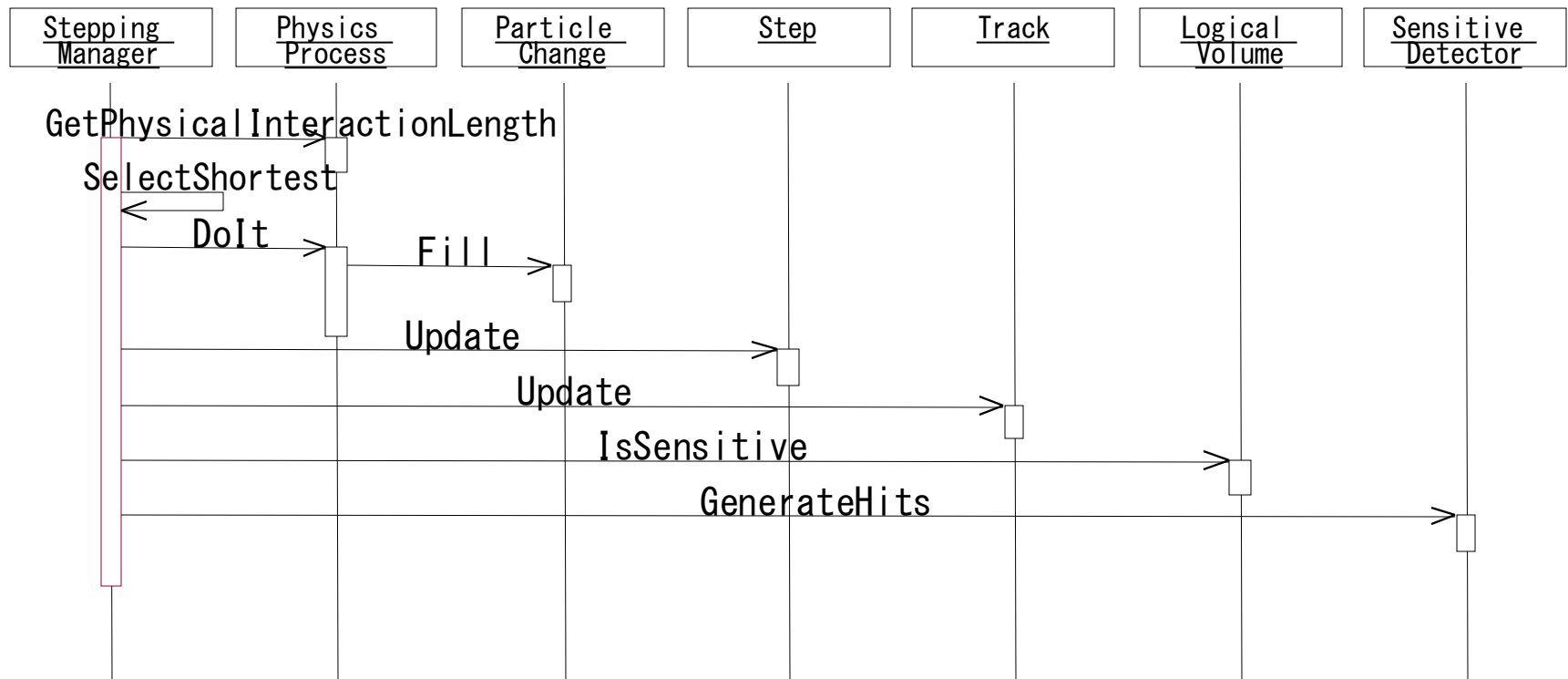


- Sensitive detector vs. primitive scorer
- Basic structure of detector sensitivity
- Sensitive detector and hit
- Add a new scorer / filter



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 as described in the previous talk.



Defining a sensitive detector

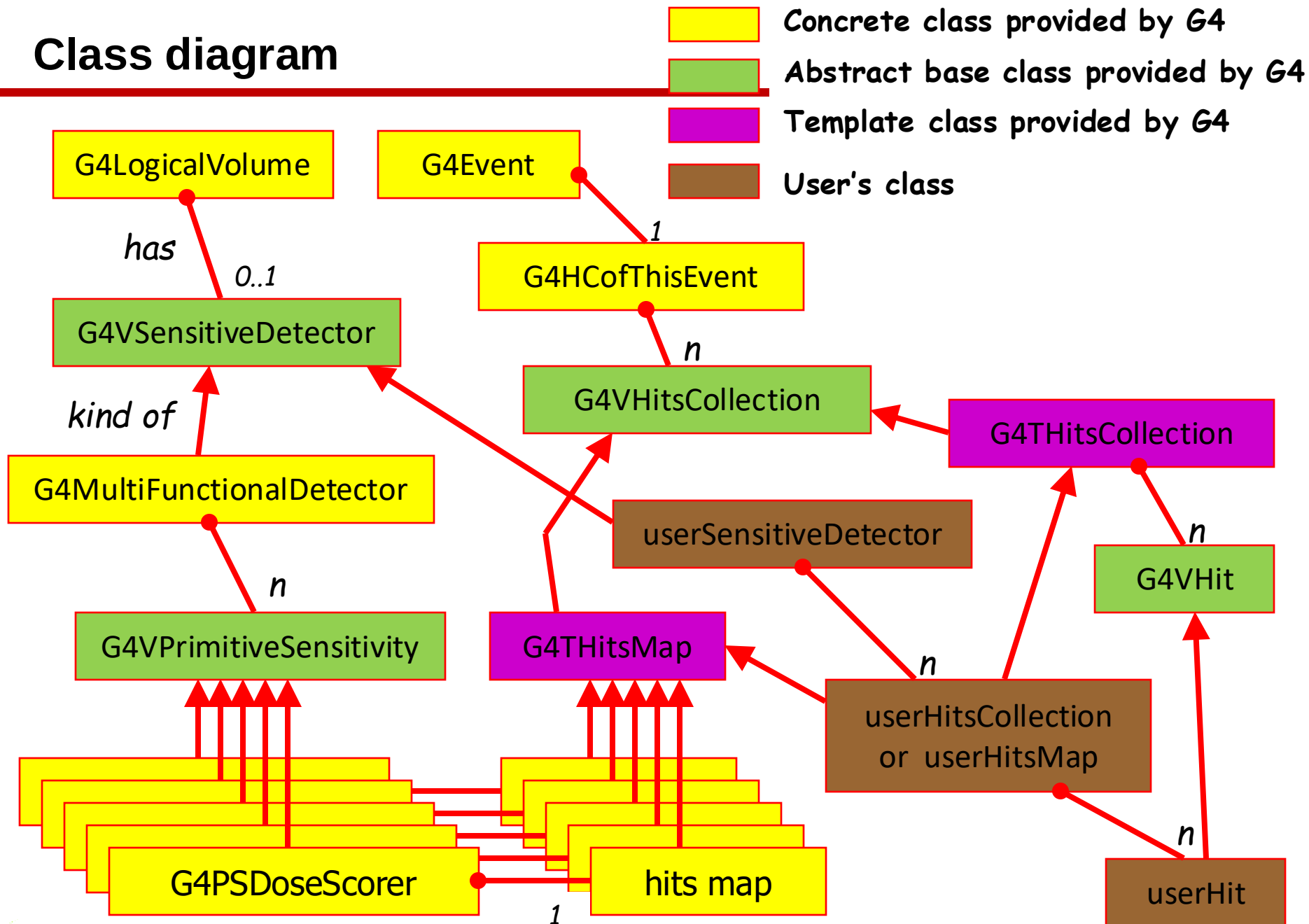
- Basic strategy

In the `ConstructSDandField()` method of your detector construction class

```
G4VSensitiveDetector* pSensitiveDet
= new MyDetector("/mydet");
G4SDManager::GetSDMpointer()
    ->AddNewDetector(pSensitiveDet);
SetSensitiveDetector("myLogicalVolume", pSensitiveDet);
```

- 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



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



- Sensitive detector vs. primitive scorer
- Basic structure of detector sensitivity
- Sensitive detector and hit
- Add a new scorer / filter



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

```
#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.

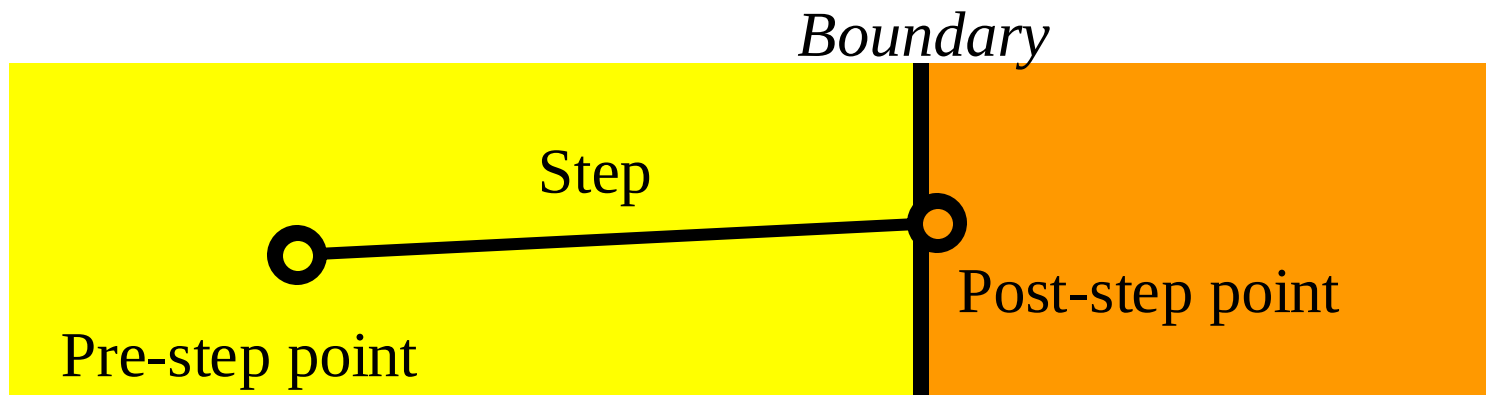
```
#include "G4VSensitiveDetector.hh"
#include "MyHit.hh"
class G4Step;
class G4HCofThisEvent;
class MyDetector : public G4VSensitiveDetector
{
public:
    MyDetector(G4String name);
    virtual ~MyDetector();
    virtual void Initialize(G4HCofThisEvent*HCE);
    virtual G4bool ProcessHits(G4Step*aStep,
                               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

Touchable

- 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();

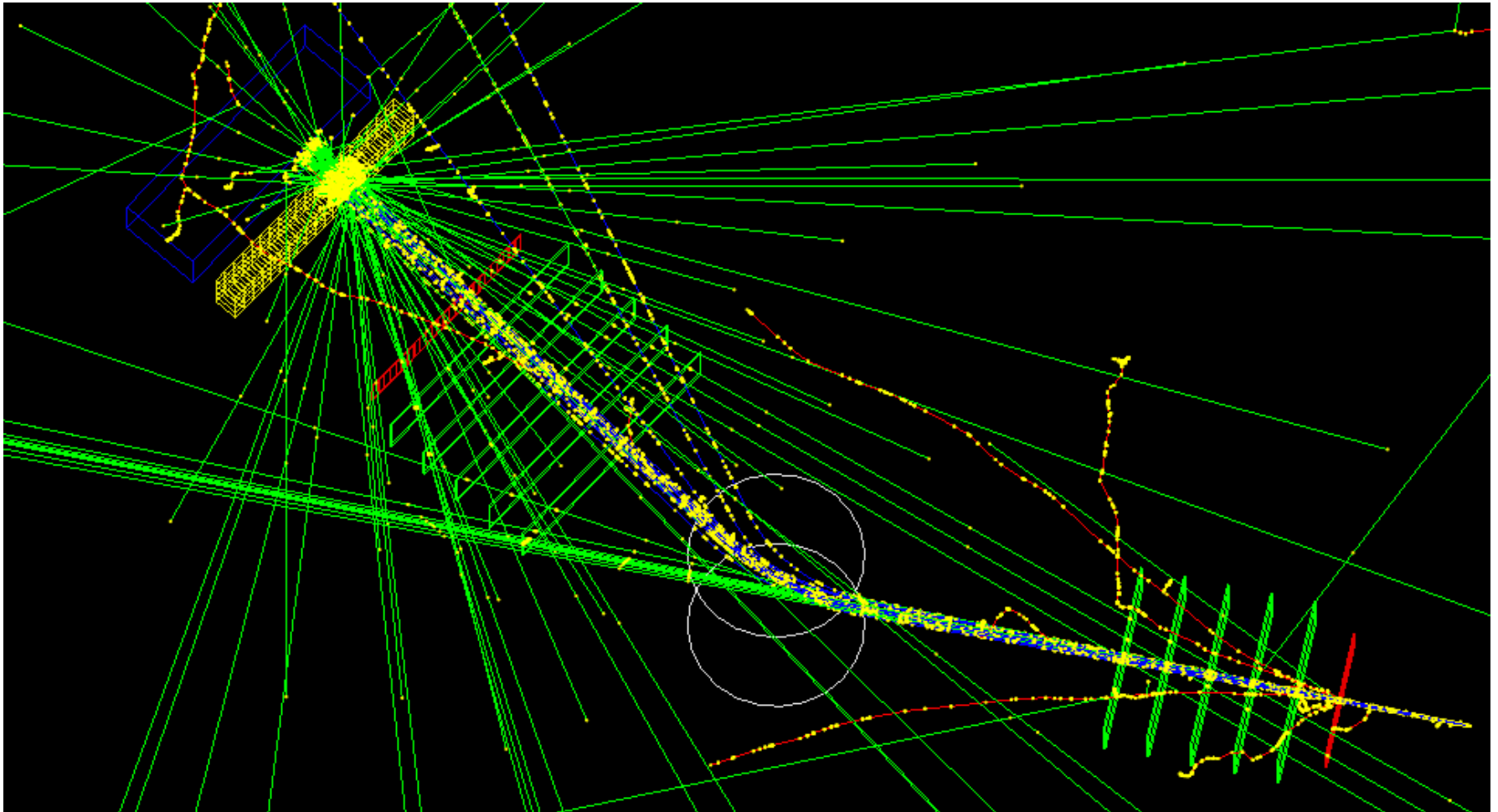
G4ThreeVector worldPos = preStepPoint->GetPosition();

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

```
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.

Contents



- Sensitive detector vs. primitive scorer
- Basic structure of detector sensitivity
- Sensitive detector and hit
- Add a new scorer / filter



Scorer base class

- G4VPrimitiveScorer is the abstract base of all scorer classes.
- To make your own scorer, you have to implement at least:
 - Constructor
 - Initialize()
 - Initialize G4THitsMap<G4double> map object
 - ProcessHits()
 - Get the physics quantity you want from G4Step, etc. and fill the map
 - Clear()
 - GetIndex()
 - Convert **three copy numbers** into an index of the map
- G4PSEnergyDeposit3D could be a good example.
- Create your own messenger class to define /score/quantity/<your_quantity> command.
 - Refer to G4ScorerQuantityMessengerQCmd class.

Creating your own scorer

- Though we provide most commonly-used scorers, you may want to create your own.
 - If you believe your requirement is quite common, just let us know, so that we will add a new scorer.
- G4VPrimitiveScorer is the abstract base class.

```
class G4VPrimitiveScorer
{
public:
    G4VPrimitiveScorer(G4String name, G4int depth=0);
    virtual ~G4VPrimitiveScorer();
protected:
    virtual G4bool ProcessHits(G4Step*,
                               G4TouchableHistory*) = 0;
    virtual G4int GetIndex(G4Step*);
public:
    virtual void Initialize(G4HCofThisEvent*);
    virtual void EndOfEvent(G4HCofThisEvent*);
    virtual void clear();
    ...
};
```

- Methods written in red will be discussed at “Scoring 2” talk.

Filter class

- G4VSDFilter

- Abstract base class which you can use to make your own filter

```
class G4VSDFilter
```

```
{
```

```
    public:
```

```
        G4VSDFilter(G4String name) ;
```

```
        virtual ~G4VSDFilter() ;
```

```
    public:
```

```
        virtual G4bool Accept(const G4Step*) const = 0;
```

```
    ...
```

- Create your own messenger class to define /score/filter/<your_filter> command.
 - Refer to G4ScorerQuantityMessenger class.