esipap

European School of Instrumentation in Particle & Astroparticle Physics

esi

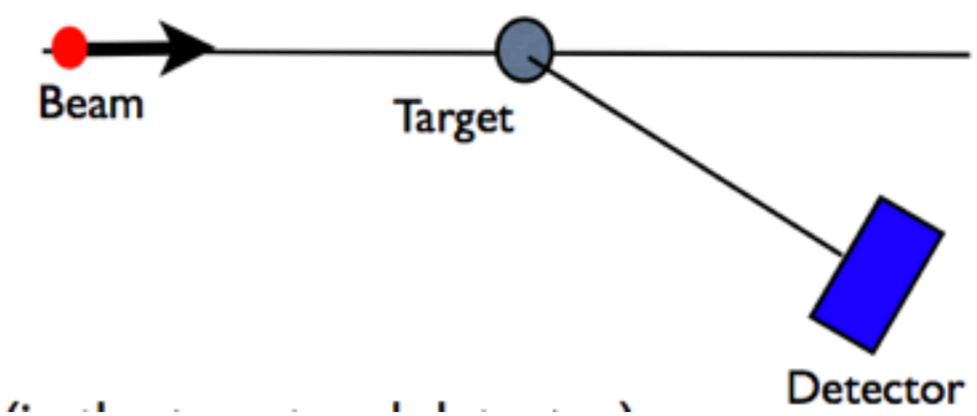
European Scientific Institute

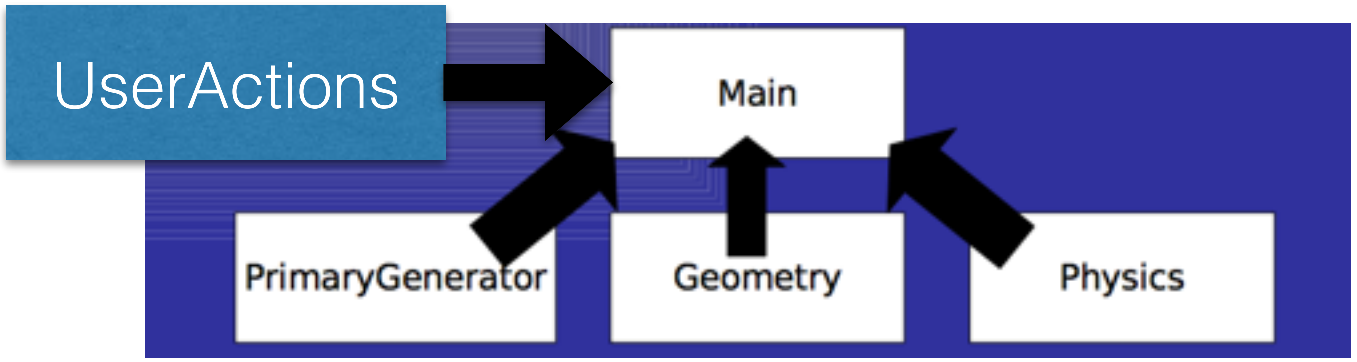
Detector Simulation
Sensitive Detectors and Hits
User Actions

Witek Pokorski
Alberto Ribon
CERN

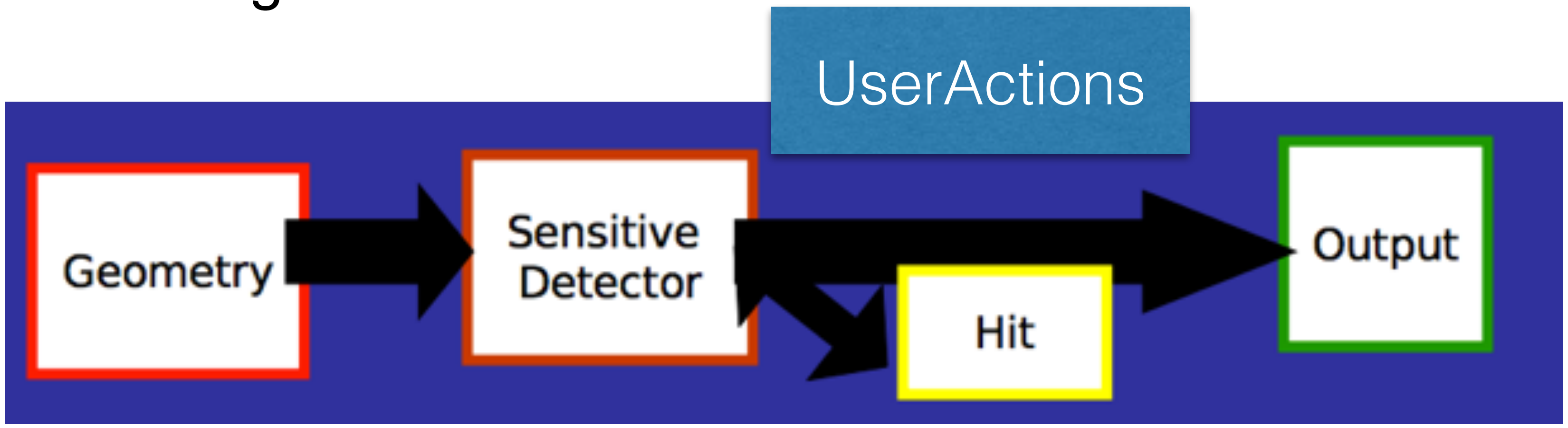
10-11.02.2014

What you need to make simulation?





and to get something out of it…



Introduction

- We know now how to create a detector
- We have started to look at the physics simulation
- In lecture we will learn how to extract information useful to you. Examples: energy released, number of particles, etc. Different methods:
  - Sensitive Detector
  - User Actions
  - Scoring

In This Part

- What is a Sensitive Detector?
- What is a G4Step?
- How to create a Sensitive Detector and use it

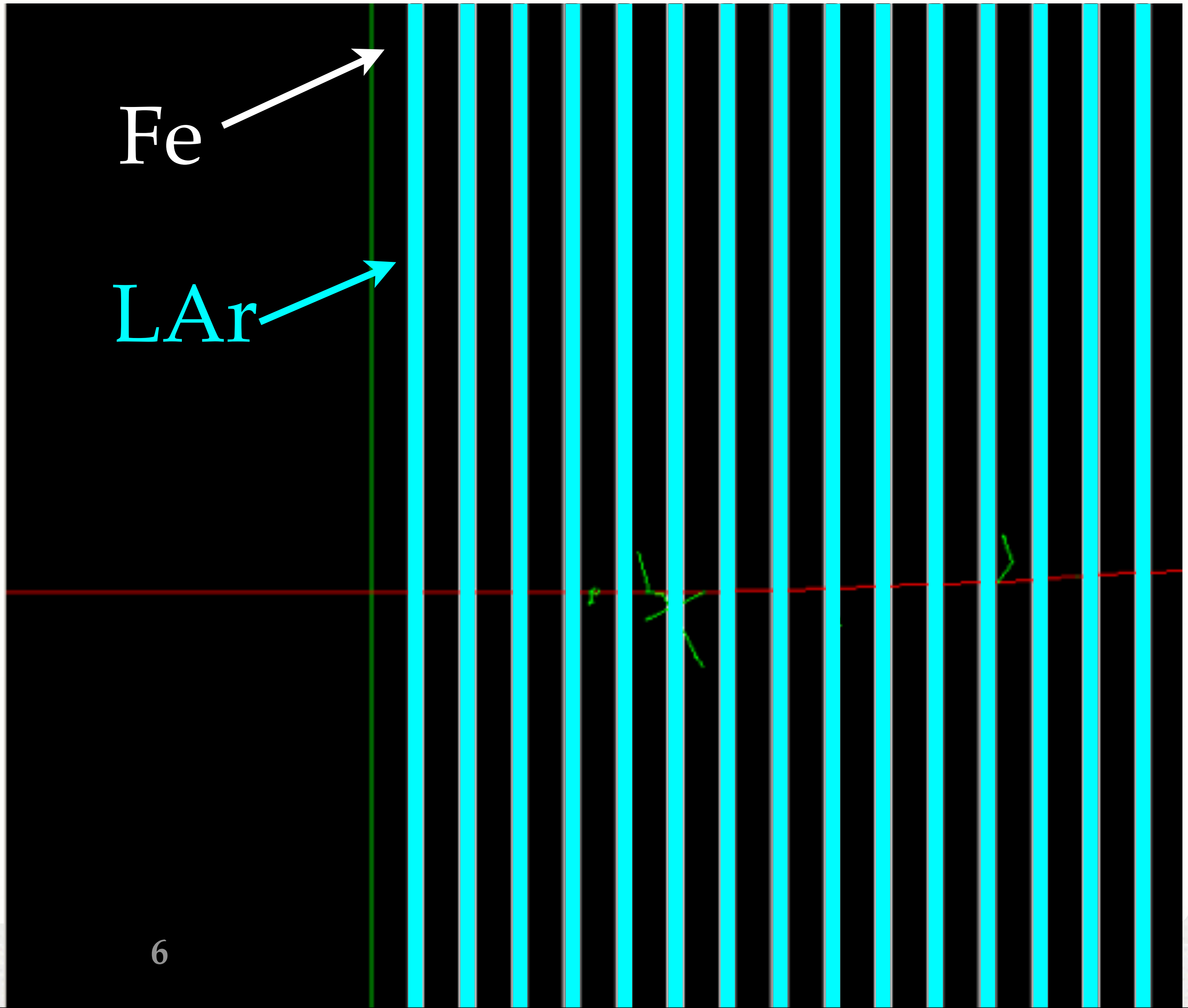
Sensitive Detector

- A SD can be used to simulate the "read-out" of your detector:
  - It is a way to declare a geometric element "sensitive" to the passage of particles
  - It gives the user a handle to collect quantities from these elements. For example: energy deposited, position, time information

Example: Hadcalo

- Hadronic Calorimeter consists of layers of absorber (Fe) and layers of active material (LAr)
  - We want to collect energy released in LAr layers
  - G4 tracks particles in the detector, when a particle passes through a detector declared sensitive the user's SD code is called

mu- @ 2 GeV



Sensitivedetector Class

- To create a SD you need to:

  1. Write your SensitiveDetector class

  2. Attach it to a logical volume

- Example: HadCaloSensitiveDetector class, inherits from G4VSensitiveDetector

  - G4VSensitiveDetector declares interface

Adding A Sd

- Basic strategy (in src/DetectorConstruction.cc):

```
G4LogicalVolume* hadLayerLogic = new G4LogicalVolume(hadLayerSolid,lar,"HadLayerLogic");

HadCaloSensitiveDetector* sensitive = new HadCaloSensitiveDetector("/HadClo");
G4SDManager* sdman = G4SDManager::GetSDMpointer();
sdman->AddNewDetector( sensitive );
hadLayerLogic->SetSensitiveDetector(sensitive);
```

- Each SD object must have a unique name.
  - Different logical volumes can share one SD object.
  - More than one SD object can be made from the same SD class with different detector name.

Sensitivedetector Class

```
#include "G4VSensitiveDetector.hh"
```

Add include

```
class HadCaloSensitiveDetector : public G4VSensitiveDetector
{
public:
  /// Constructor
  HadCaloSensitiveDetector(G4String SDname);
  /// Destructor
  ~HadCaloSensitiveDetector();

public:
  /// @name methods from base class G4VSensitiveDetector
  //@{
  /// Mandatory base class method : it must to be overloaded:
   G4bool ProcessHits(G4Step *step, G4TouchableHistory *ROhist);

  /// (optional) method of base class G4VSensitiveDetector
  void Initialize(G4HCofThisEvent* HCE);
  /// (optional) method of base class G4VSensitiveDetector
  void EndOfEvent(G4HCofThisEvent* HCE);
  //@}


private:
};
```

Sensitivedetector Class

```
#include "G4VSensitiveDetector.hh"
```

Base class

```
class HadCaloSensitiveDetector : public G4VSensitiveDetector
{
public:
  /// Constructor
  HadCaloSensitiveDetector(G4String SDname);
  /// Destructor
  ~HadCaloSensitiveDetector();

public:
  /// @name methods from base class G4VSensitiveDetector
  //@{
  /// Mandatory base class method : it must to be overloaded:
   G4bool ProcessHits(G4Step *step, G4TouchableHistory *ROhist);

  /// (optional) method of base class G4VSensitiveDetector
  void Initialize(G4HCofThisEvent* HCE);
  /// (optional) method of base class G4VSensitiveDetector
  void EndOfEvent(G4HCofThisEvent* HCE);
  //@}


private:
};
```

Sensitivedetector Class

```
#include "G4VSensitiveDetector.hh"


class HadCaloSensitiveDetector : public G4VSensitiveDetector
{
public:
  /// Constructor
  HadCaloSensitiveDetector(G4String SDname);
  /// Destructor
  ~HadCaloSensitiveDetector();

public:
  /// @name methods from base class G4VSensitiveDetector
  //@{
  /// Mandatory base class method : it must to be overloaded:
   G4bool ProcessHits(G4Step *step, G4TouchableHistory *ROhist);

  /// (optional) method of base class G4VSensitiveDetector
  void Initialize(G4HCofThisEvent* HCE);
  /// (optional) method of base class G4VSensitiveDetector
  void EndOfEvent(G4HCofThisEvent* HCE);
  //@}


private:
};
```

Constructor: SD are named!

Sensitivedetector Class

```
#include "G4VSensitiveDetector.hh"


class HadCaloSensitiveDetector : public G4VSensitiveDetector
{
public:
  /// Constructor
  HadCaloSensitiveDetector(G4String SDname);
  /// Destructor
  ~HadCaloSensitiveDetector();

public:
  /// @name methods from base class G4VSensitiveDetector
  //@{
  /// Mandatory base class method : it must to be overloaded:
   G4bool ProcessHits(G4Step *step, G4TouchableHistory *ROhist);

  /// (optional) method of base class G4VSensitiveDetector
  void Initialize(G4HCofThisEvent* HCE);
  /// (optional) method of base class G4VSensitiveDetector
  void EndOfEvent(G4HCofThisEvent* HCE);
  //@}


private:
};
```

Initialization: called at beginning of event

Note: G4HCofThisEvent will be discussed later today!

Sensitivedetector Class

```
#include "G4VSensitiveDetector.hh"


class HadCaloSensitiveDetector : public G4VSensitiveDetector
{
public:
  /// Constructor
  HadCaloSensitiveDetector(G4String SDname);
  /// Destructor
  ~HadCaloSensitiveDetector();

public:
  /// @name methods from base class G4VSensitiveDetector
  //@{
  /// Mandatory base class method : it must to be overloaded:
   G4bool ProcessHits(G4Step *step, G4TouchableHistory *ROhist);

  /// (optional) method of base class G4VSensitiveDetector
  void Initialize(G4HCofThisEvent* HCE);
  /// (optional) method of base class G4VSensitiveDetector
  void EndOfEvent(G4HCofThisEvent* HCE);
  //@}


private:
};
```

Finalize: called at end of event

Note: G4HCofThisEvent will be discussed later today!

Sensitivedetector Class

```
#include "G4VSensitiveDetector.hh"


class HadCaloSensitiveDetector : public G4VSensitiveDetector
{
public:
  /// Constructor
  HadCaloSensitiveDetector(G4String SDname);
  /// Destructor
  ~HadCaloSensitiveDetector();

public:
  /// @name methods from base class G4VSensitiveDetector
  //@{
  /// Mandatory base class method : it must to be overloaded:
   G4bool ProcessHits(G4Step *step, G4TouchableHistory *ROhist);

  /// (optional) method of base class G4VSensitiveDetector
  void Initialize(G4HCofThisEvent* HCE);
  /// (optional) method of base class G4VSensitiveDetector
  void EndOfEvent(G4HCofThisEvent* HCE);
  //@}


private:
};
```

Called for each G4Step in sensitive volume

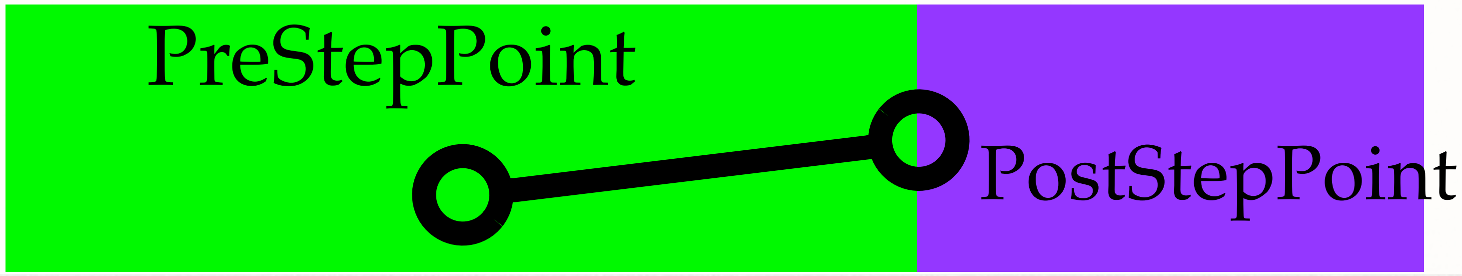
Sensitivedetector Class

```
#include "G4VSensitiveDetector.hh"


class HadCaloSensitiveDetector : public G4VSensitiveDetector
{
public:
  /// Constructor
  HadCaloSensitiveDetector(G4String SDname);
  /// Destructor
  ~HadCaloSensitiveDetector();

public:
  /// @name methods from base class G4VSensitiveDetector
  //@{
  /// Mandatory base class method : it must to be overloaded:
   G4bool ProcessHits(G4Step *step, G4TouchableHistory *ROhist);

  /// (optional) method of base class G4VSensitiveDetector
  void Initialize(G4HCofThisEvent* HCE);
  /// (optional) method of base class G4VSensitiveDetector
  void EndOfEvent(G4HCofThisEvent* HCE);
  //@}


private:
};
```

G4Step: What is it?

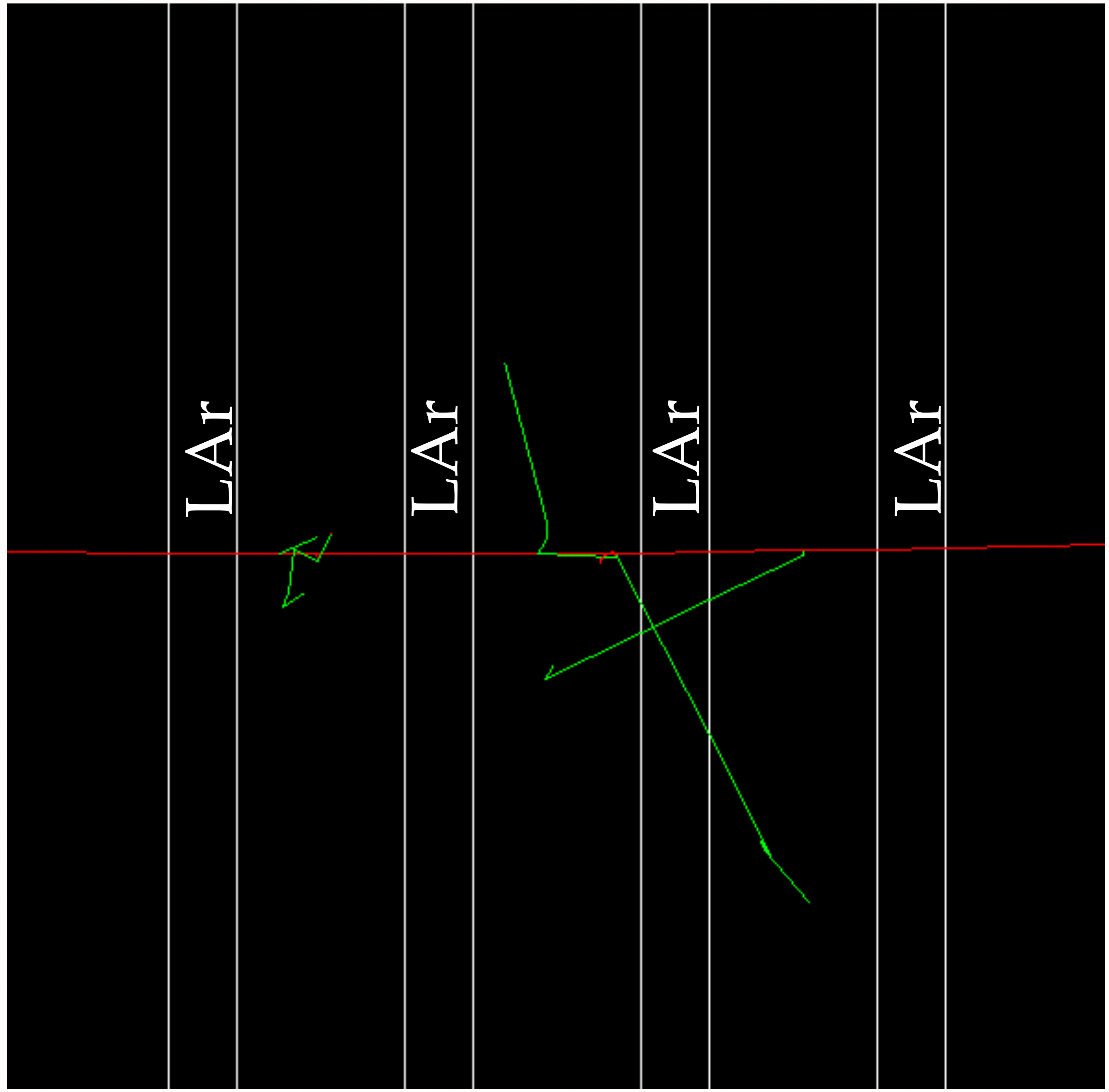
G4Step

- Snapshot of the interaction of a G4Track (particle) with a volume
- A G4Step can be seen as a “segment” delimited by two points
- It contains “delta” information (energy loss along the step, time-of-flight, etc)
- Each point knows the volume (and material) associated to it
- A step never spans across boundaries: geometry or physics define the end points
  - If the step is limited by a boundary, the post-step point stands on the boundary and it logically belongs to the next volume
  - Get the volume information from the PreStepPoint



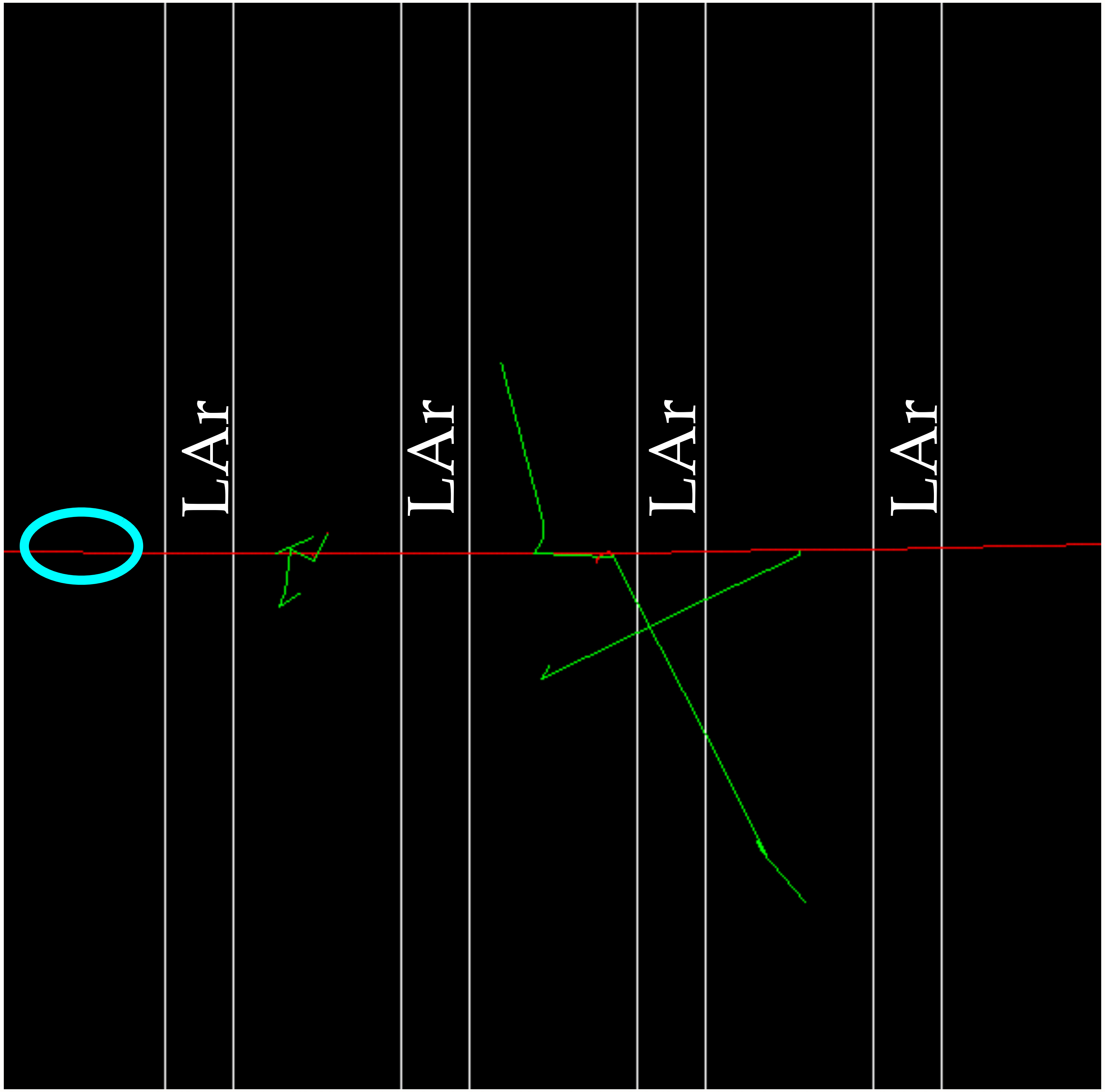
G4Step

The muon track passes through the calorimeter



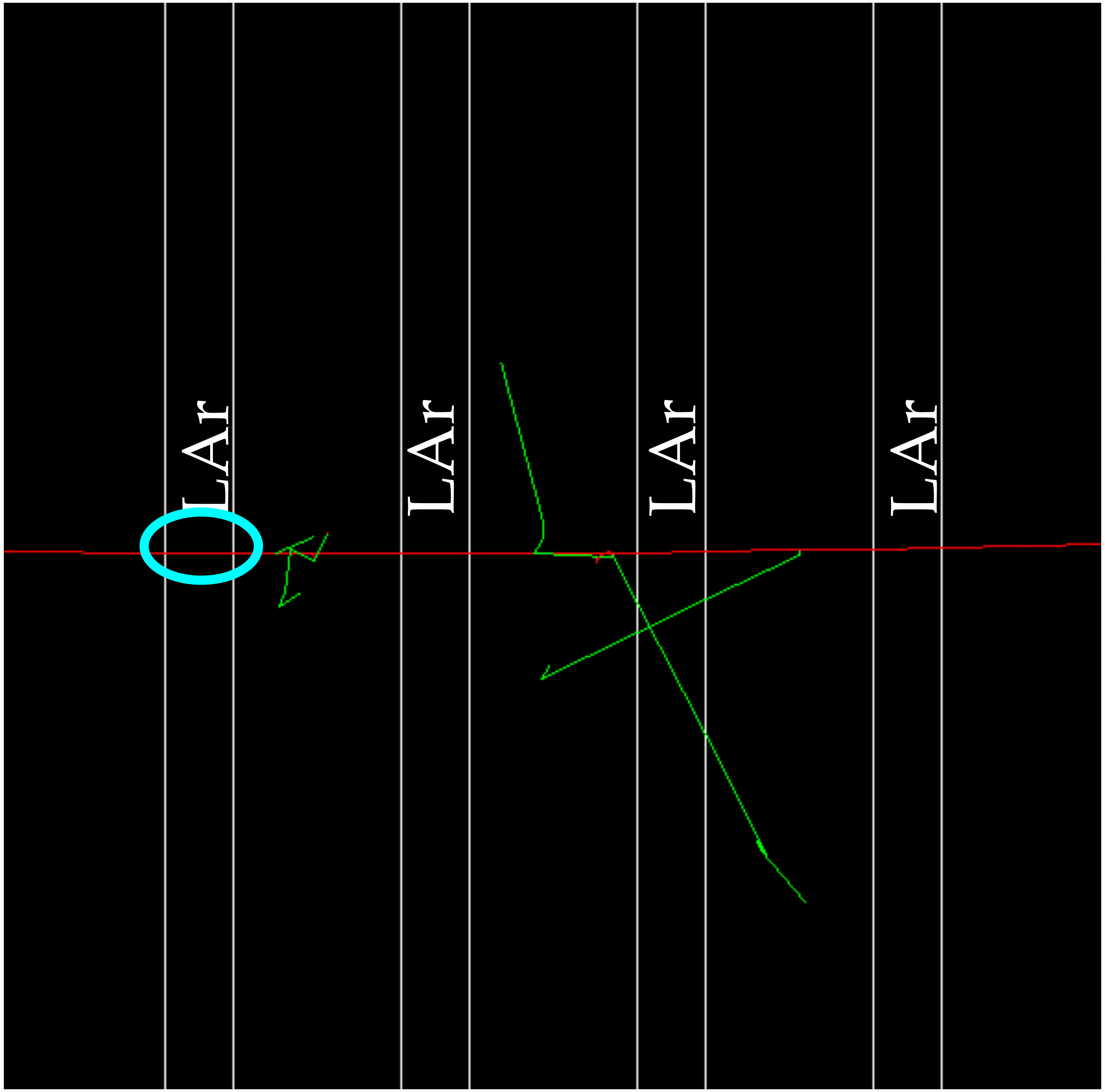
G4Step

A Step in Fe:
SD is ignored



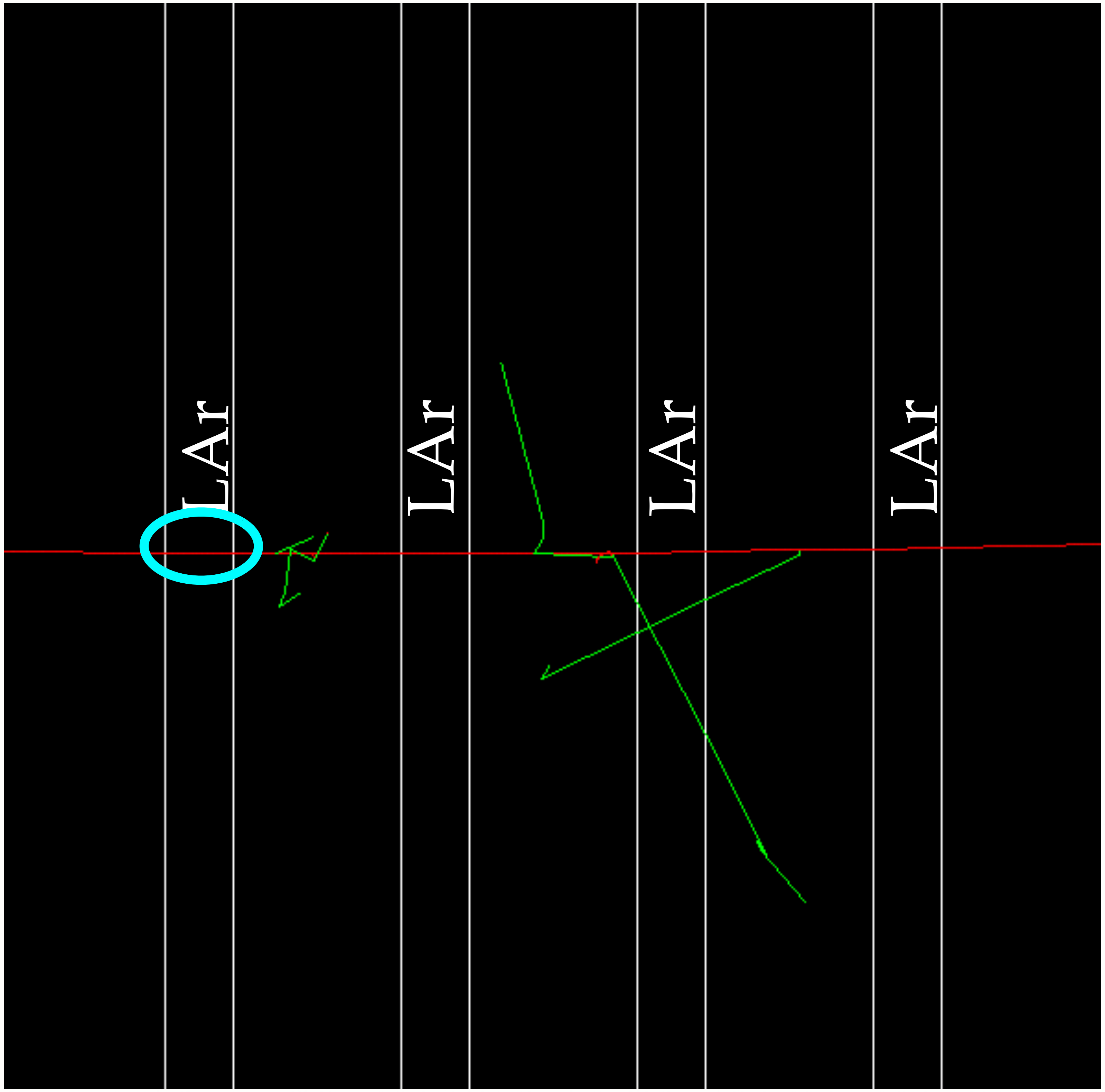
G4Step

A Step in LAr:
It's sensitive thus ::ProcessHits(...) will be called



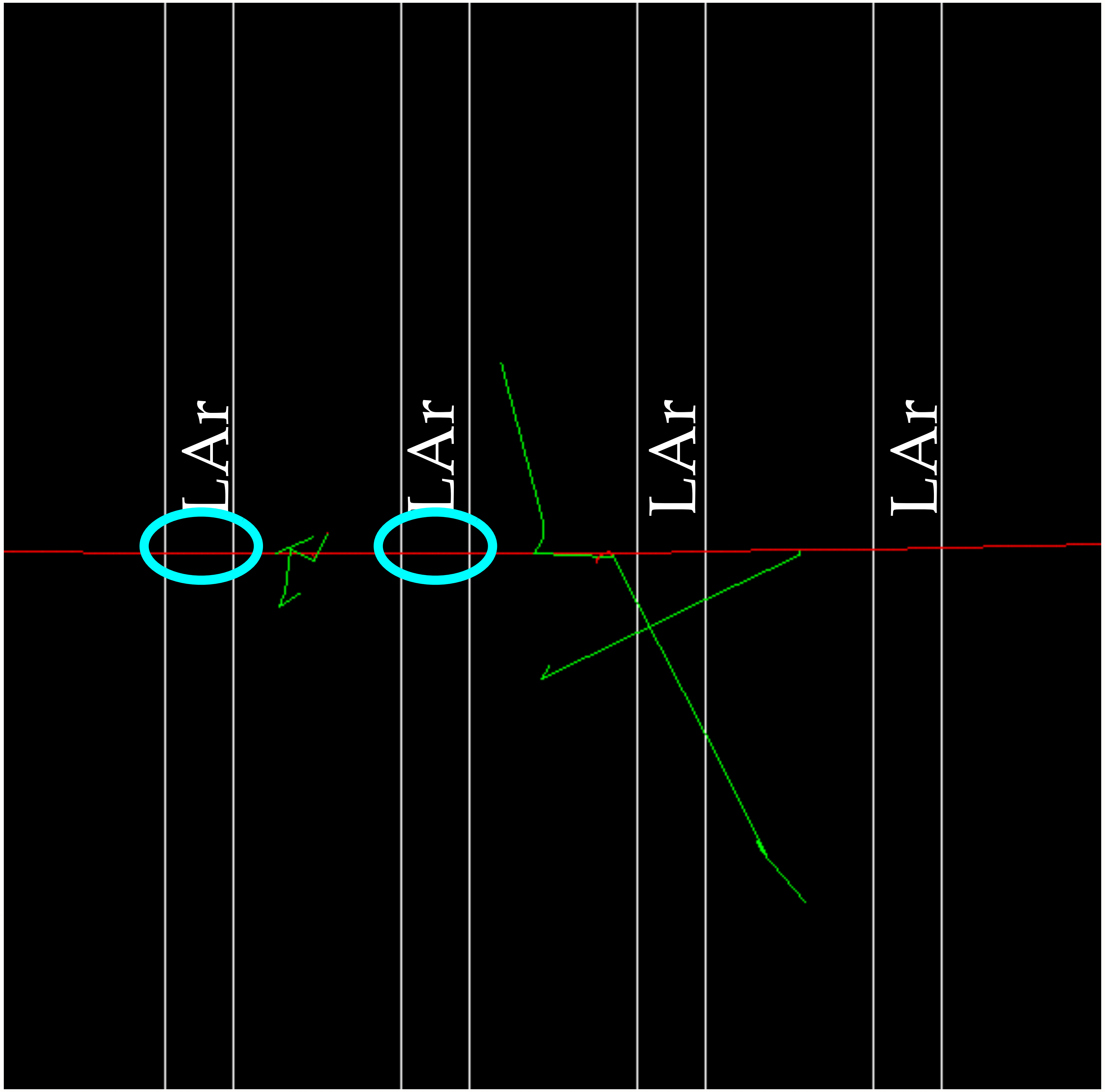
G4Step

For all these G4Steps ::ProcessHits(...) will be called



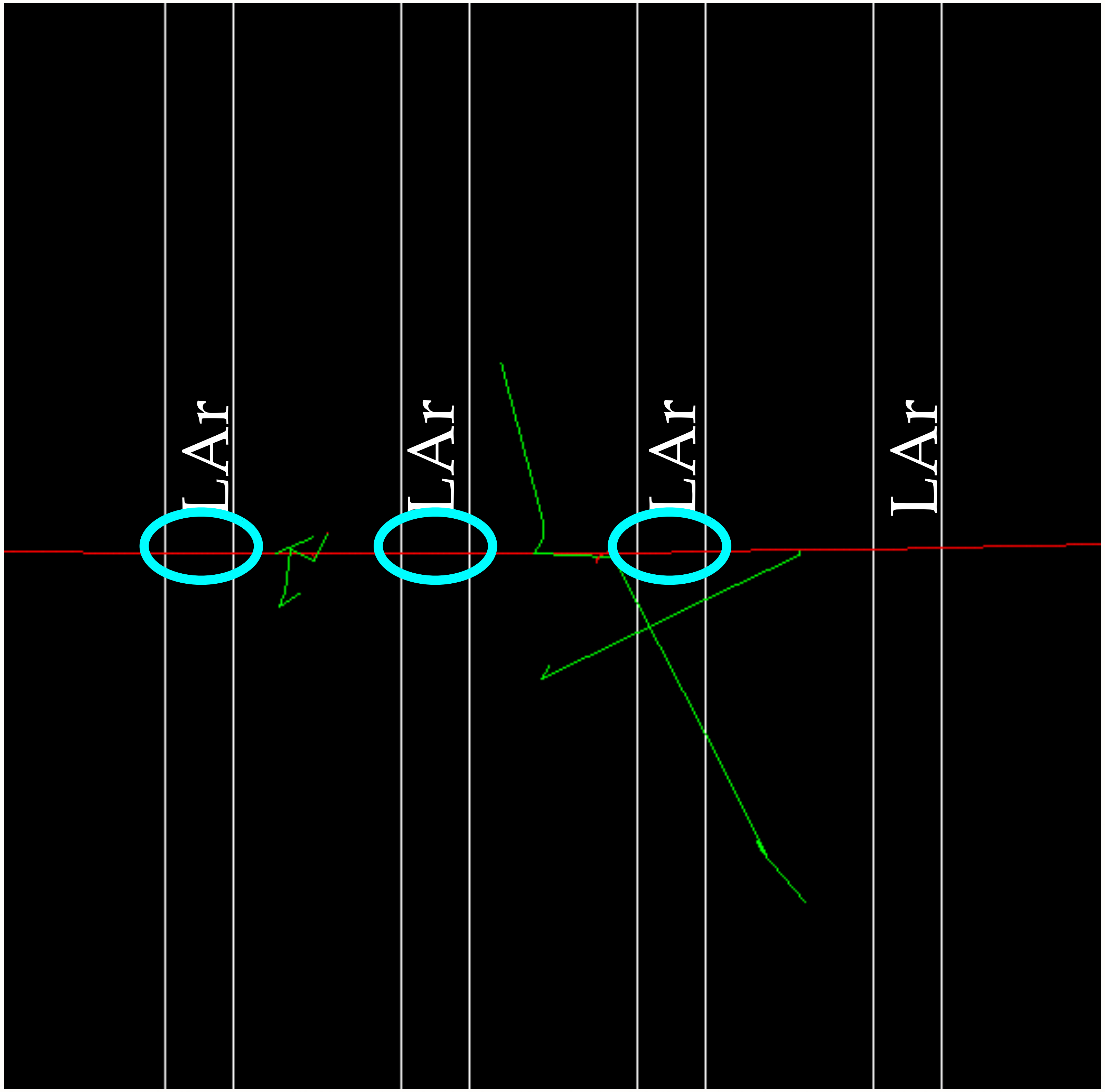
G4Step

For all these G4Steps ::ProcessHits(...) will be called



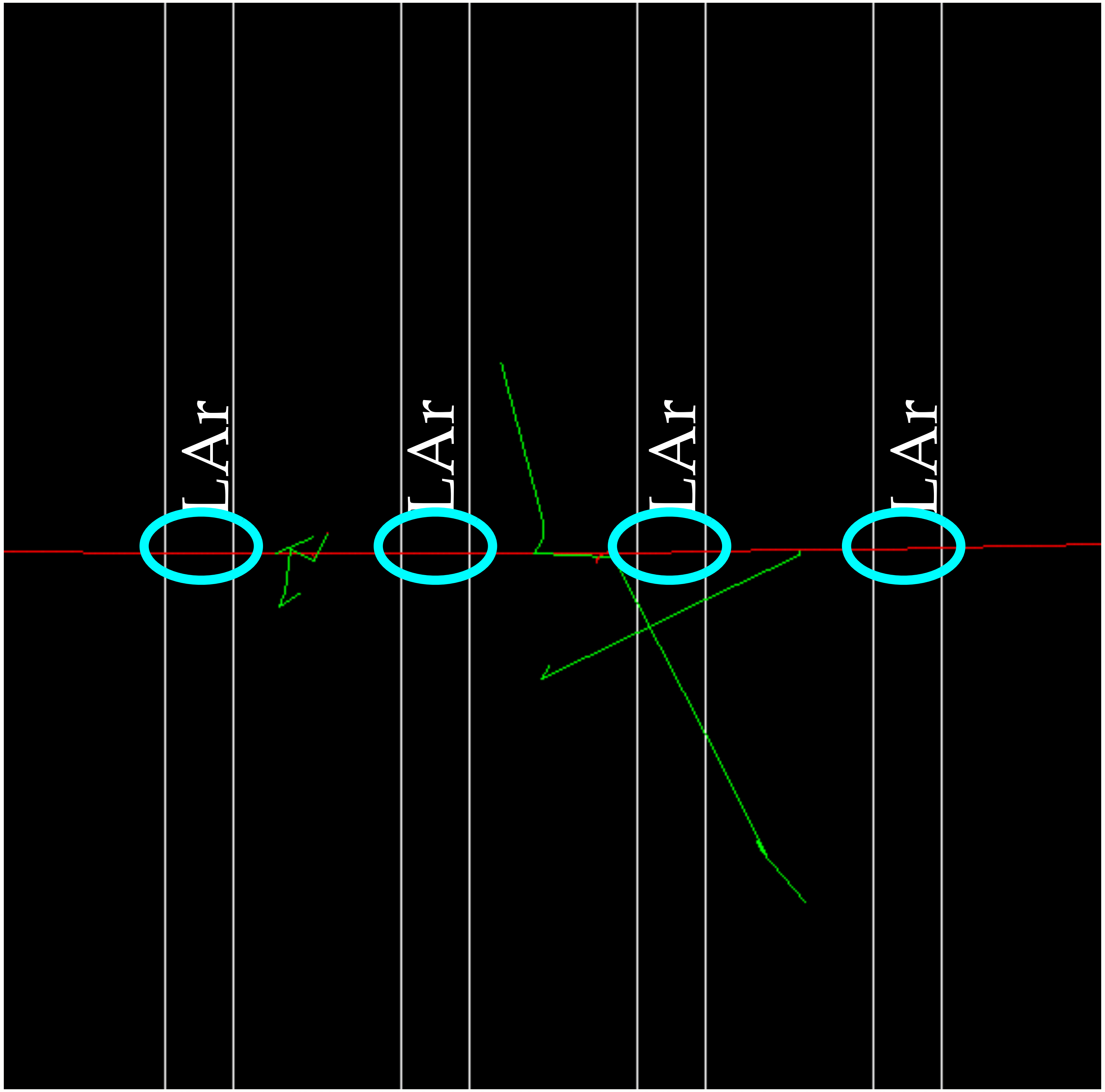
G4Step

For all these G4Steps ::ProcessHits(...) will be called



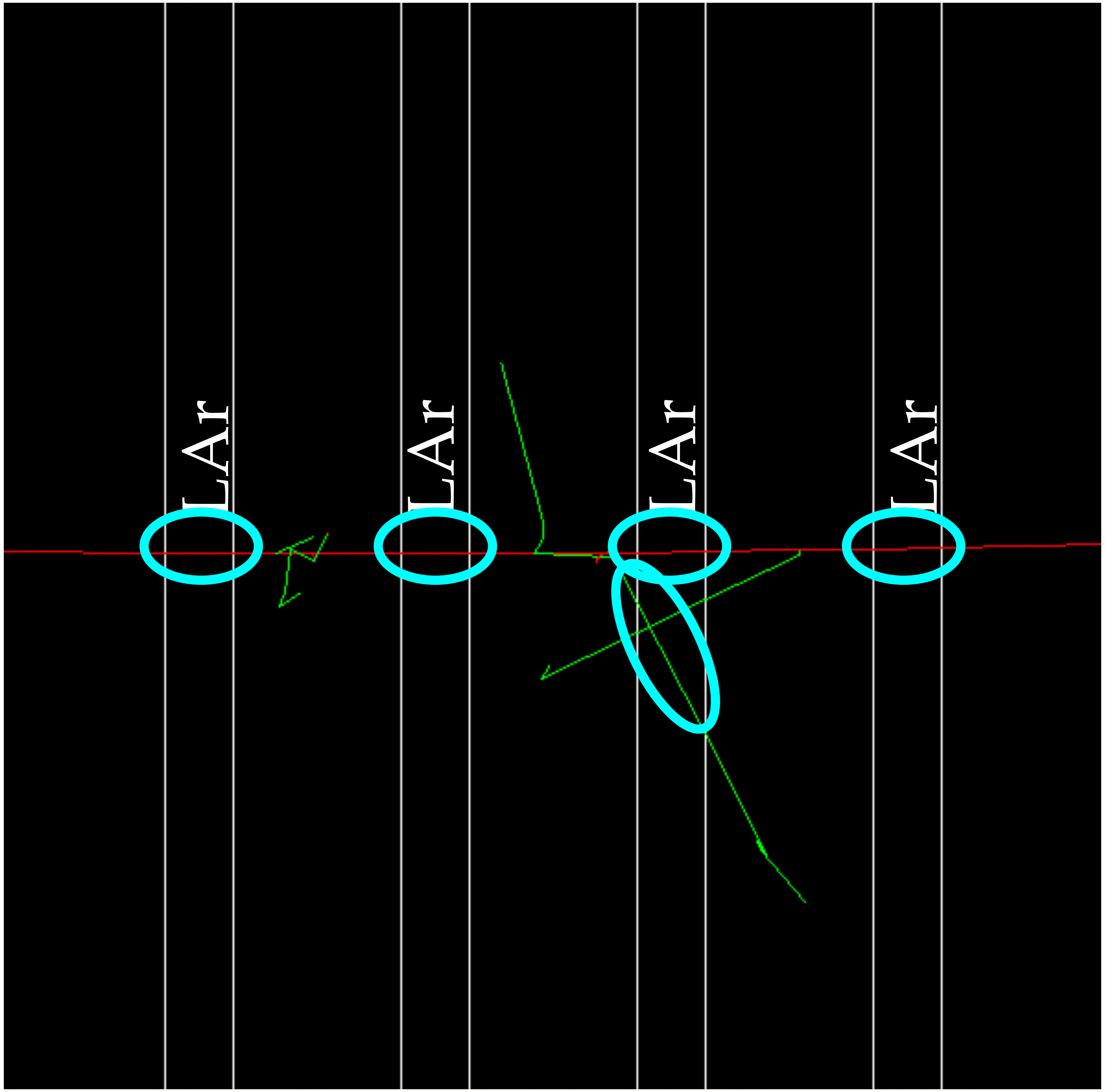
G4Step

For all these G4Steps ::ProcessHits(...) will be called



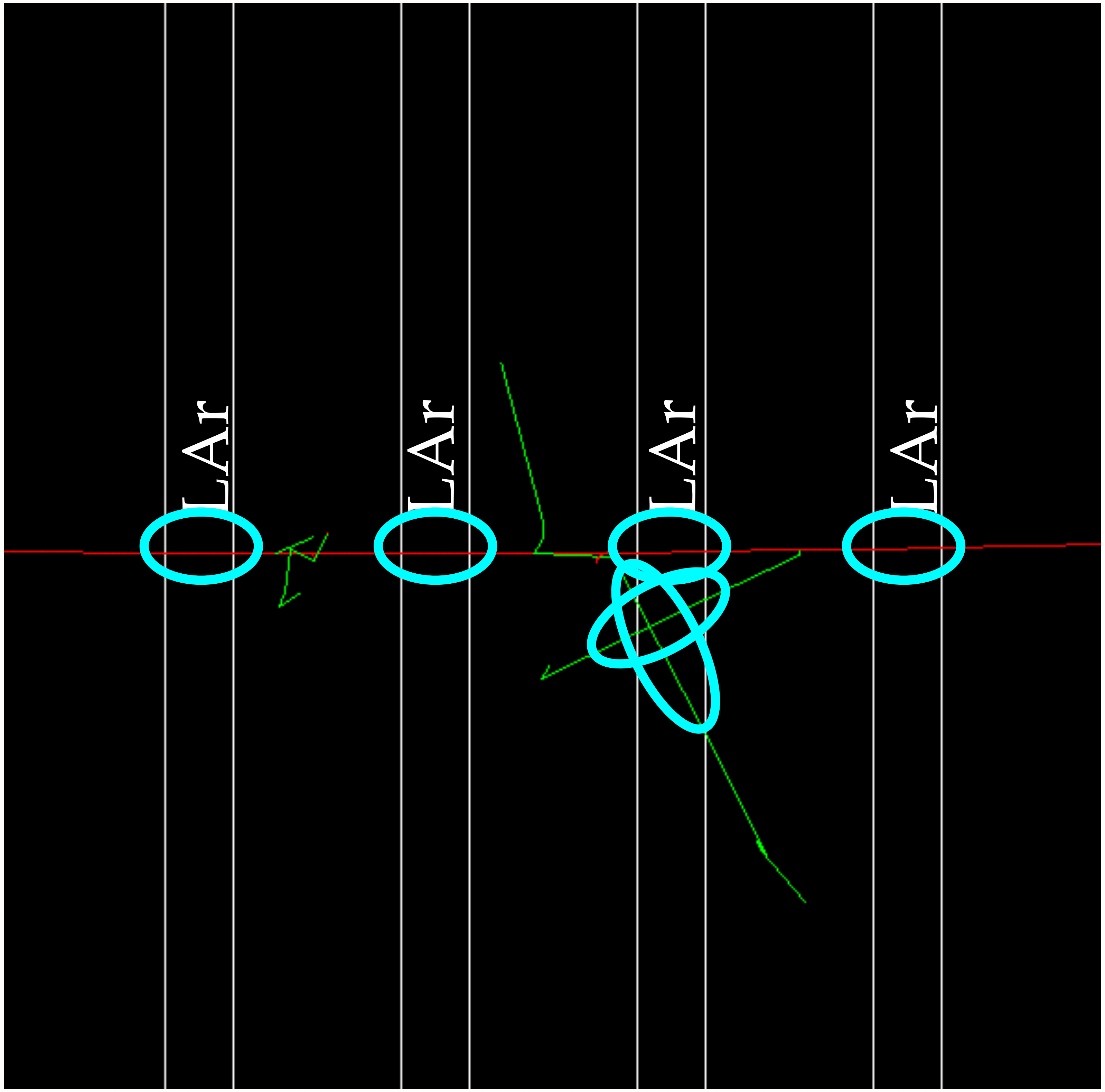
G4Step

For all these G4Steps ::ProcessHits(...) will be called



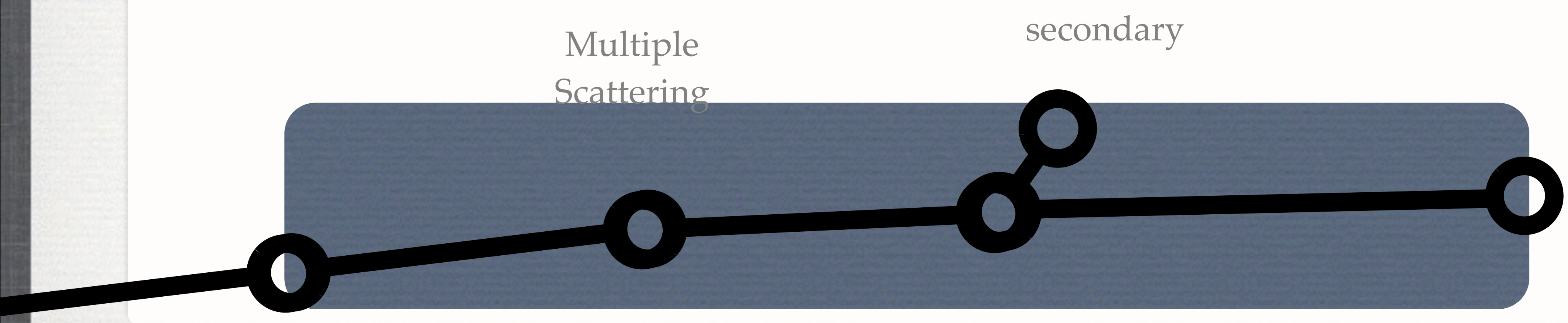
G4Step

For all these G4Steps ::ProcessHits(...) will be called



A Note On G4Step

- A G4Step is delimited by:
  - Geometry boundaries
  - A physics process (non continuous)
- G4Track is constant during step, G4 guarantees step is never too long (i.e. Edep does not change too much Ekin G4Track)



Getting Information From G4Steps

- G4Step can be interrogated to get information about physics process and volumes:

```
G4bool HadCaloSensitiveDetector::ProcessHits(G4Step *step, G4TouchableHistory *)
{
```

```
  G4TouchableHandle touchable = step->GetPreStepPoint()->GetTouchableHandle();
  G4int copyNo = touchable->GetVolume(0)->GetCopyNo();
```

Get volume where G4Step is remember:
Use PreStepPoint! PostStep "belongs" to next volume

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

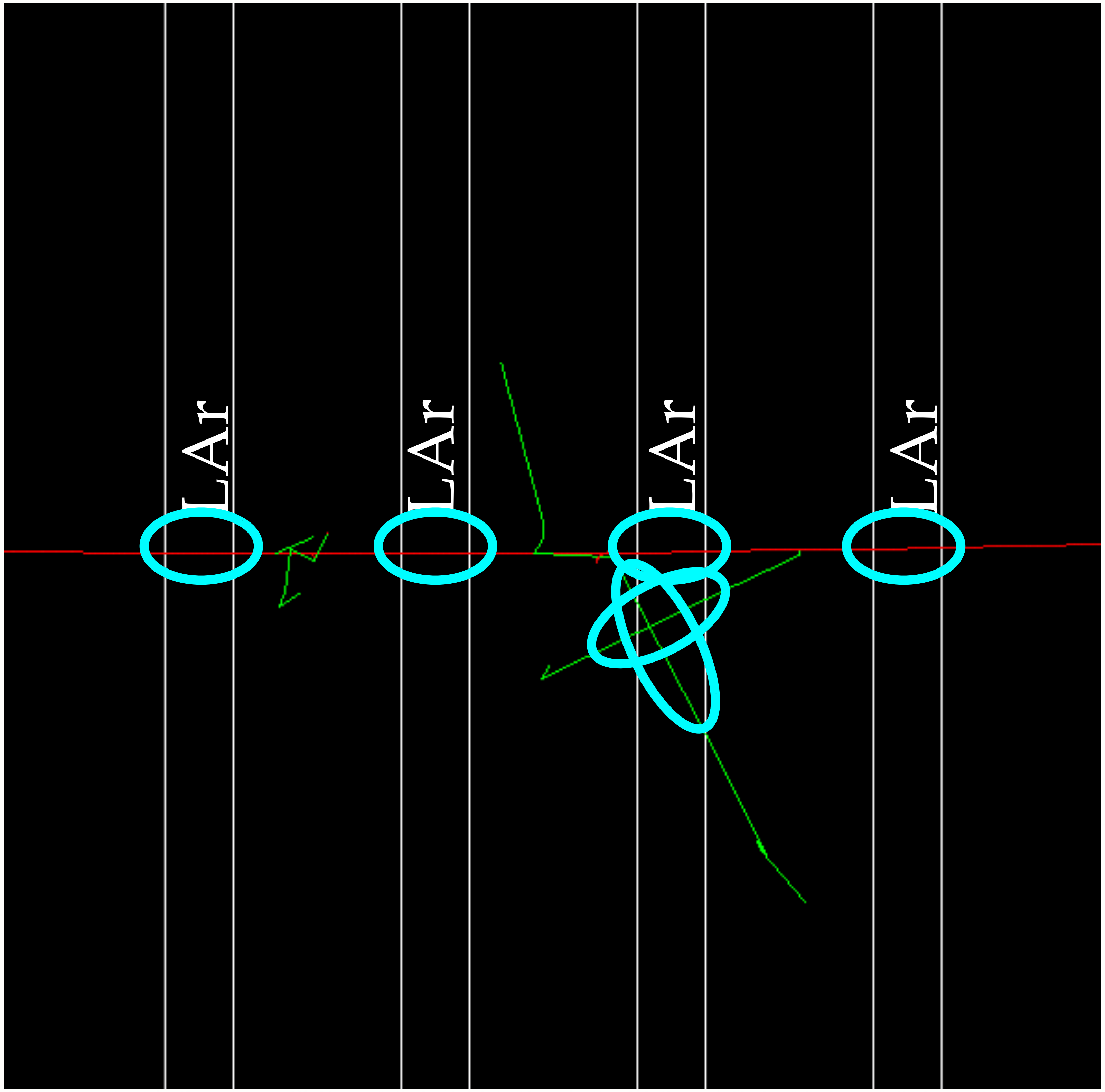
Get energy deposited along G4Step (i.e. ionization)

In This Part

- What is G4Hit?
- How to use hits

Reminder: G4Step

For all these G4Steps ::ProcessHits(...) will be called



What Hits Are

- Hits are created in Sensitive Detector to store user quantities
- Hits are collected in a container and “registered” in Geant4
  - Hits become available to all components of the application
- 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
- Hits should be identified: they have an id that uniquely identifies them

Hits

- You need to write your own Hit class: inherits from G4VHit
- Hits must be stored in a collection of hits instantiated from G4THitsCollection template class

```
#include "G4VHit.hh"
#include "G4Allocator.hh"
#include "G4THitsCollection.hh"
```

← Headers files

```
class HadCaloHit : public G4VHit {
public:
  HadCaloHit(const G4int layer);
  ~HadCaloHit();
  void Print();
  void          AddEdep(const double e){ eDep += e; }

  G4double      GetEdep()        const { return eDep;}
  G4int         GetLayerNumber() const { return layerNumber; }
private:
  const G4int   layerNumber;
  G4double      eDep;
};

// Define the "hit collection" using the template class G4THitsCollection:
typedef G4THitsCollection<HadCaloHit> HadCaloHitCollection;
```

Hits

- You need to write your own Hit class: inherits from G4VHit

- Hits must be stored in a collection of hits instantiated from G4THitsCollection template class

```
#include "G4VHit.hh"
#include "G4Allocator.hh"
#include "G4THitsCollection.hh"

class HadCaloHit : public G4VHit {
public:
   HadCaloHit(const G4int layer);
  ~HadCaloHit();
   void Print();
   void            AddEdep(const double e){ eDep += e; }

   G4double        GetEdep()        const { return eDep;}
   G4int           GetLayerNumber() const { return layerNumber; }
private:
  const G4int    layerNumber;
  G4double       eDep;
};

// Define the "hit collection" using the template class G4THitsCollection:
typedef G4THitsCollection<HadCaloHit> HadCaloHitCollection;
```

Base class

Hits

- You need to write your own Hit class: inherits from G4VHit
- Hits must be stored in a collection of hits instantiated from G4THitsCollection template class

```
#include "G4VHit.hh"
#include "G4Allocator.hh"
#include "G4THitsCollection.hh"

class HadCaloHit : public G4VHit {
public:
   HadCaloHit(const G4int layer);
  ~HadCaloHit();
   void Print();
   void          AddEdep(const double e){ eDep += e; }

   G4double      GetEdep()        const { return eDep;}
   G4int         GetLayerNumber() const { return layerNumber; }
private:
  const G4int   layerNumber;
  G4double      eDep;
};

// Define the "hit collection" using the template class G4THitsCollection:
typedef G4THitsCollection<HadCaloHit> HadCaloHitCollection;
```

Create a new Hit: the ID is the layer index

Hits

- You need to write your own Hit class: inherits from G4VHit

- Hits must be stored in a collection of hits instantiated from G4THitsCollection template class

```cpp
#include "G4VHit.hh"
#include "G4Allocator.hh"
#include "G4THitsCollection.hh"

class HadCaloHit : public G4VHit {
public:
  HadCaloHit(const G4int layer);
  ~HadCaloHit();
  void Print();
  void        AddEdep(const double e){ eDep += e; }

  G4double      GetEdep()        const { return eDep;}
  G4int         GetLayerNumber() const { return layerNumber; }
private:
  const G4int   layerNumber;
  G4double      eDep;
};

// Define the "hit collection" using the template class G4THitsCollection:
typedef G4THitsCollection<HadCaloHit> HadCaloHitCollection;
```

Hit interface: print on screen

Hits

- You need to write your own Hit class: inherits from G4VHit
- Hits must be stored in a collection of hits instantiated from G4THitsCollection template class

Our specific hit interface

```
#include "G4VHit.hh"
#include "G4Allocator.hh"
#include "G4THitsCollection.hh"

class HadCaloHit : public G4VHit {
public:
  HadCaloHit(const G4int layer);
  ~HadCaloHit();
  void Print();
  void           AddEdep(const double e){ eDep += e; }

  G4double       GetEdep()        const { return eDep;}
  G4int          GetLayerNumber() const { return layerNumber; }
private:
  const G4int    layerNumber;
  G4double       eDep;
};

// Define the "hit collection" using the template class G4THitsCollection:
typedef G4THitsCollection<HadCaloHit> HadCaloHitCollection;
```

Hits

- You need to write your own Hit class: inherits from G4VHit
- Hits must be stored in a collection of hits instantiated from G4THitsCollection template class

```
#include "G4VHit.hh"
#include "G4Allocator.hh"
#include "G4THitsCollection.hh"

class HadCaloHit : public G4VHit {
public:
  HadCaloHit(const G4int layer);
  ~HadCaloHit();
  void Print();
  void          AddEdep(const double e){ eDep += e; }

  G4double      GetEdep()        const { return eDep;}
  G4int         GetLayerNumber() const { return layerNumber; }
private:
  const G4int   layerNumber;
  G4double      eDep;
};

// Define the "hit collection" using the template class G4THitsCollection:
typedef G4THitsCollection<HadCaloHit> HadCaloHitCollection;
```

The Hit container, just add this line

Warning: more advanced code (memory management optimization) not shown here, optional but highly recommended

How To Declare Hits

- A hits collection has a name, this name must be declared in SensitiveDetector constructor
- SD has a data member: collectionName, add your name to this vector of names
  - A SD can declare more than one hits collection!

```
HadCaloSensitiveDetector::HadCaloSensitiveDetector(G4String SDname)
  : G4VSensitiveDetector(SDname)
{
    G4cout<<"Creating SD with name: "<<SDname<<G4endl;
  // 'collectionName' is a protected data member of base class G4VSensitiveDetector.
  // Here we declare the name of the collection we will be using.
  collectionName.insert("HadCaloHitCollection");

  // Note that we may add as many collection names we would wish: ie
  // a sensitive detector can have many collections.
}
```

Our hits collection name!

How To Create A Hc (Hits Container)

- Every event a new hit collection (HC) has to be created and added to current event collection of hits

- Every HC has two names: the SD name that created it and the name of collection. This pair is unique

    - Geant4 uses also an identifier (a number) to uniquely identify your collection, you need to use this ID to register/retrieve the collection

```
void HadCaloSensitiveDetector::Initialize(G4HCofThisEvent* HCE)
{
   hitCollection = new HadCaloHitCollection(GetName(), collectionName[0]);

   static G4int HCID = -1;
   if (HCID<0) HCID = GetCollectionID(0); // <<-- this is to get an ID for collectionName[0]
   HCE->AddHitsCollection(HCID, hitCollection);
```

How To Create A Hc (Hits Container)

- Every event a new hit collection (HC) has to be created and added to current event collection of hits

- Every HC has two names: the SD name that created it and the name of collection. This pair is unique

    - Geant4 uses also an identifier (a number) to uniquely identify your collection, you need to use this ID to register/retrieve the collection

```
void HadCaloSensitiveDetector::Initialize(G4HCofThisEvent* HCE)
{
    hitCollection = new HadCaloHitCollection(GetName(), collectionName[0]);
    static G4int HCID = -1;
    if (HCID<0) HCID = GetCollectionID(0); // <<-- this is to get an ID for collectionName[0]
    HCE->AddHitsCollection(HCID, hitCollection);
```

Create a Hit Collection: GetName() returns SD name ("/HadCalo") , collectionName is a vector: [0] is the first (and only in our case) element ("HadCaloHitCollection")

How To Create A Hc (Hits Container)

- Every event a new hit collection (HC) has to be created and added to current event collection of hits

- Every HC has two names: the SD name that created it and the name of collection. This pair is unique

  - Geant4 uses also an identifier (a number) to uniquely identify your collection, you need to use this ID to register/retrieve the collection

```
void HadCaloSensitiveDetector::Initialize(G4HCofThisEvent* HCE)
{
   hitCollection = new HadCaloHitCollection(GetName(), collectionName[0]);
   static G4int HCID = -1;
   if (HCID<0) HCID = GetCollectionID(0); // <<-- this is to get an ID for collectionName[0]
   HCE->AddHitsCollection(HCID, hitCollection);
```

GetCollectionID(0) is an heavy operation, you should avoid to do it every event!
GetCollectionID(0) returns the unique ID associated to the hit collection!

How To Create A Hc (Hits Container)

- Every event a new hit collection (HC) has to be created and added to current event collection of hits

- Every HC has two names: the SD name that created it and the name of collection. This pair is unique
    - Geant4 uses also an identifier (a number) to uniquely identify your collection, you need to use this ID to register/retrieve the collection

```
void HadCaloSensitiveDetector::Initialize(G4HCofThisEvent* HCE)
{
  hitCollection = new HadCaloHitCollection(GetName(), collectionName[0]);
  static G4int HCID = -1;
  if (HCID<0) HCID = GetCollectionID(0); // <<-- this is to get an ID for collectionName[0]
  HCE->AddHitsCollection(HCID, hitCollection);
```

Register the hits collection object in the Hits Collections of This Event (G4HCofThisEvent)

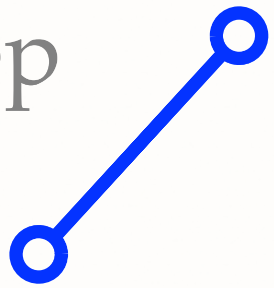
How To Create And Fill Hits

- Every time ProcessHits is called you can (if needed) create a hit and add it to the hits collection

```
G4bool HadCaloSensitiveDetector::ProcessHits(G4Step *step, G4TouchableHistory *)
{
      HadCaloHit* aHit = new HadCaloHit(layerIndex);
     hitCollection->insert(aHit);
    aHit->AddEdep( edep );
    return true;
}
```

Summary

G4Step



Summary

Summary

Repeat for each step in the event

Summary

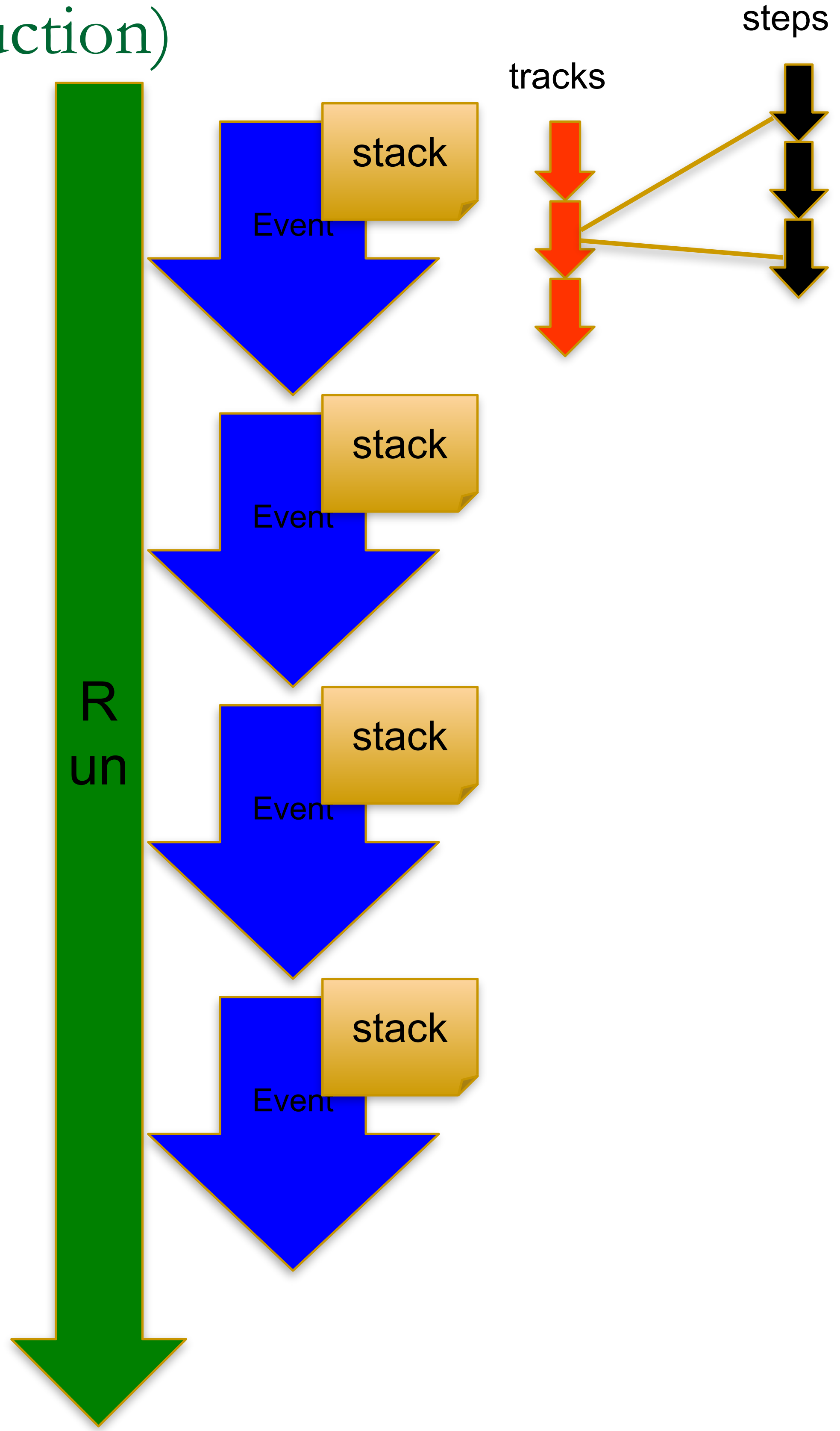
End of the event

In This Part

- User Actions

User actions (slide from Introduction)

- how to control your simulation?
  - G4UserRunAction
    - BeginOfRunAction, EndOfRunAction
  - G4UserEventAction
    - BeginOfEventAction, EndOfEventAction
  - G4UserStackingAction
    - ClassifyNewTrack, NewStage, PrepareNewEvent
  - G4UserTrackingAction
    - PreUserTrackingAction, PostUserTrackingAction
  - G4UserSteppingAction
    - BeginOfStepAction, EndOfStepAction
- fully customizable (empty by default)
- allow user to take actions depending on his specific case
  - simulated only relevant particles
  - save specific information, fill histograms
  - speed-up simulation by applying different limits



Extract information from G4 internal objects

- Simulation is successively split into
- Run consists of
- Event(s), consists of
- Particle(s) transported in
- Steps through detector setup,
- depositing energy (ionization),
- and creating secondaries

- Corresponding / related Objects
- G4RunManager, G4Run
- G4Event
- G4Track, G4DynamicParticle
- G4Step, G4StepPoint
- G4Trajectory
- G4Stack

User Actions

- User at each moment has possibility to take control or access information via UserAction classes
  - G4UserRunAction Actions for each Run
  - G4UserEventAction Actions for each Event
  - G4UserTrackingAction Actions for each Track
  - G4UserSteppingAction Actions for each Step
  - G4UserStackingAction Tracks Stack management

RunManager in Geant4

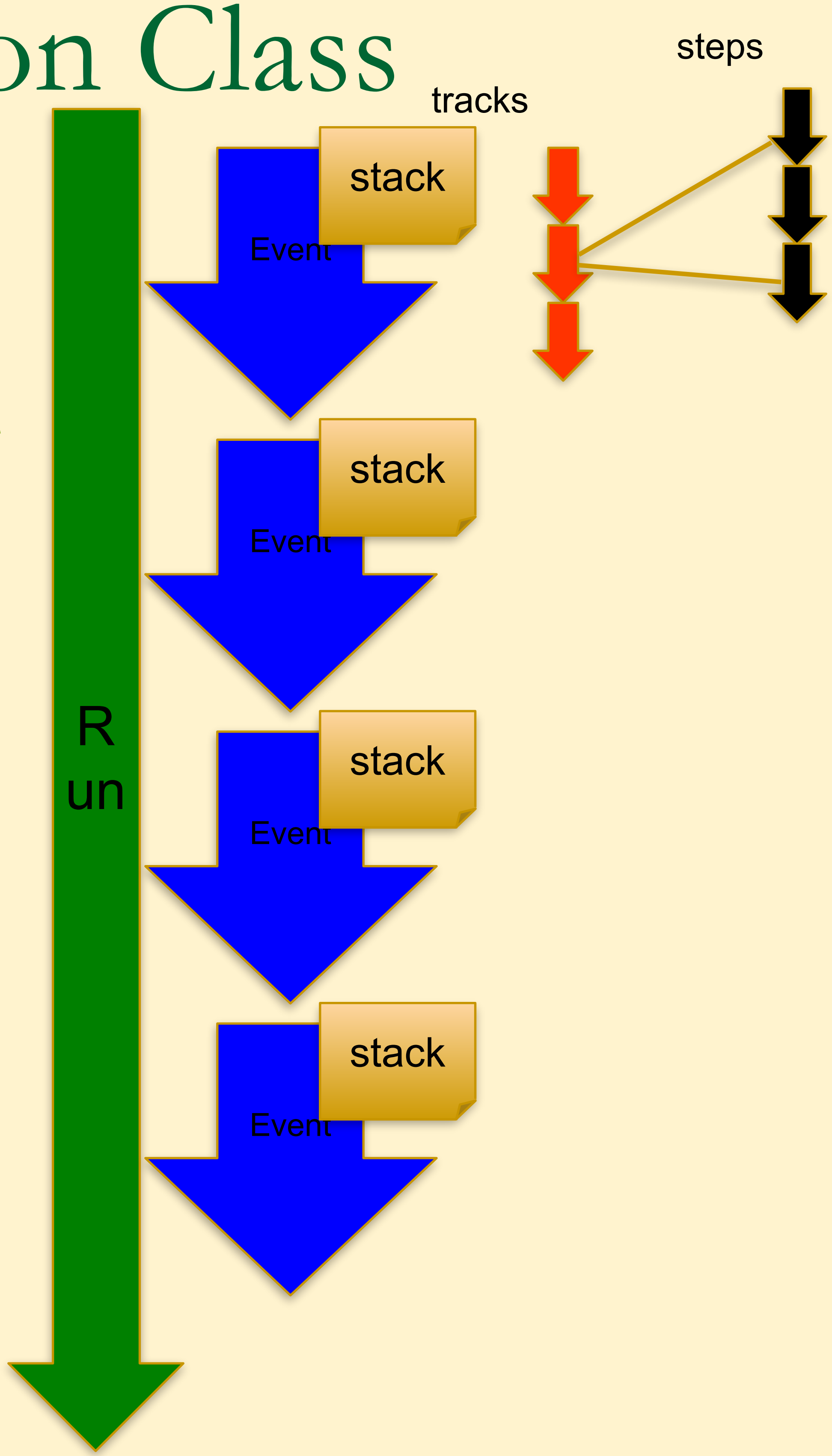
- G4RunManager class manages processing a run
  - Must be created by user
  - May be user derived class
  - Must be singleton
- User must register in RunManager using
  - SetUserInitialization() method
    - Geometry
    - Physics
  - SetUserAction() method
    - Event generator
  - Optional UserAction objects

Run in Geant4

- Run is a collection of events
  - A run consists of one event loop
  - Starts with a /run/beamOn command.
- Within a run, conditions do not change, i.e. the user cannot change
  - detector setup
  - settings of physics processes
- At the beginning of a run, geometry is optimized for navigation and cross-section tables are calculated according to materials appear in the geometry and the cut-off values defined.
- Run is represented by G4Run class or a user-defined class derived from G4Run.
  - A run class may have a summary results of the run.
- G4RunManager is the manager class
- G4UserRunAction is the optional user hook.

Optional User **Run** Action Class

- G4UserRunAction
  - G4Run* GenerateRun()
    - Instantiate user-customized run object
  - void BeginOfRunAction(const G4Run*)
    - Define histograms
  - void EndOfRunAction(const G4Run*)
    - Analyze the run
    - Store histograms

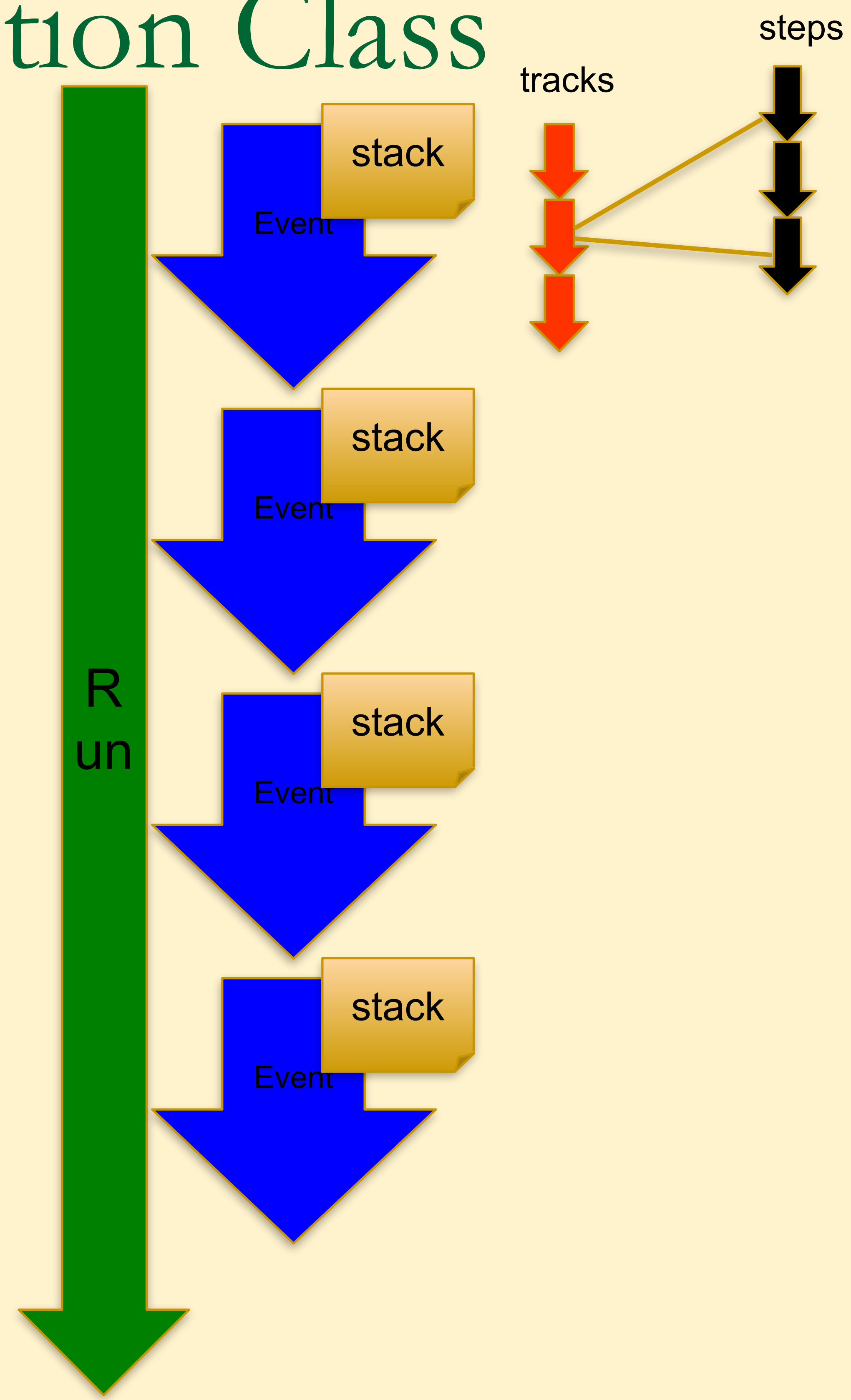


Event in Geant4

- An event is the basic unit of simulation in Geant4.
- At beginning of processing, primary tracks are generated. These primary tracks are pushed into a stack.
- A track is popped up from the stack one by one and "tracked". Resulting secondary tracks are pushed into the stack.
  - This "tracking" lasts as long as the stack has a track.
- When the stack becomes empty, processing of one event is over.
- G4Event class represents an event. It has the following objects at the end of its (successful) processing.
  - List of primary vertices and particles (as input)
  - Hits and Trajectory collections (as output)
- G4EventManager class manages processing an event.
- G4UserEventAction is the optional user hook.

Optional User **Event** Action Class

- G4UserEventAction
  - void BeginOfEventAction(const G4Event*)
    - Event selection
      - Using information from event generator, vertices, primary particles
    - Optionally attach G4VUserEventInformation object
  - void EndOfEventAction(const G4Event*)
    - Output event information
    - Analyse event
      - Access to hits collection via G4Event::GetHCofThisEvent()
      - Acces digitisation collection via G4Event:: GetDCofThisEvent()
    - Fill histograms

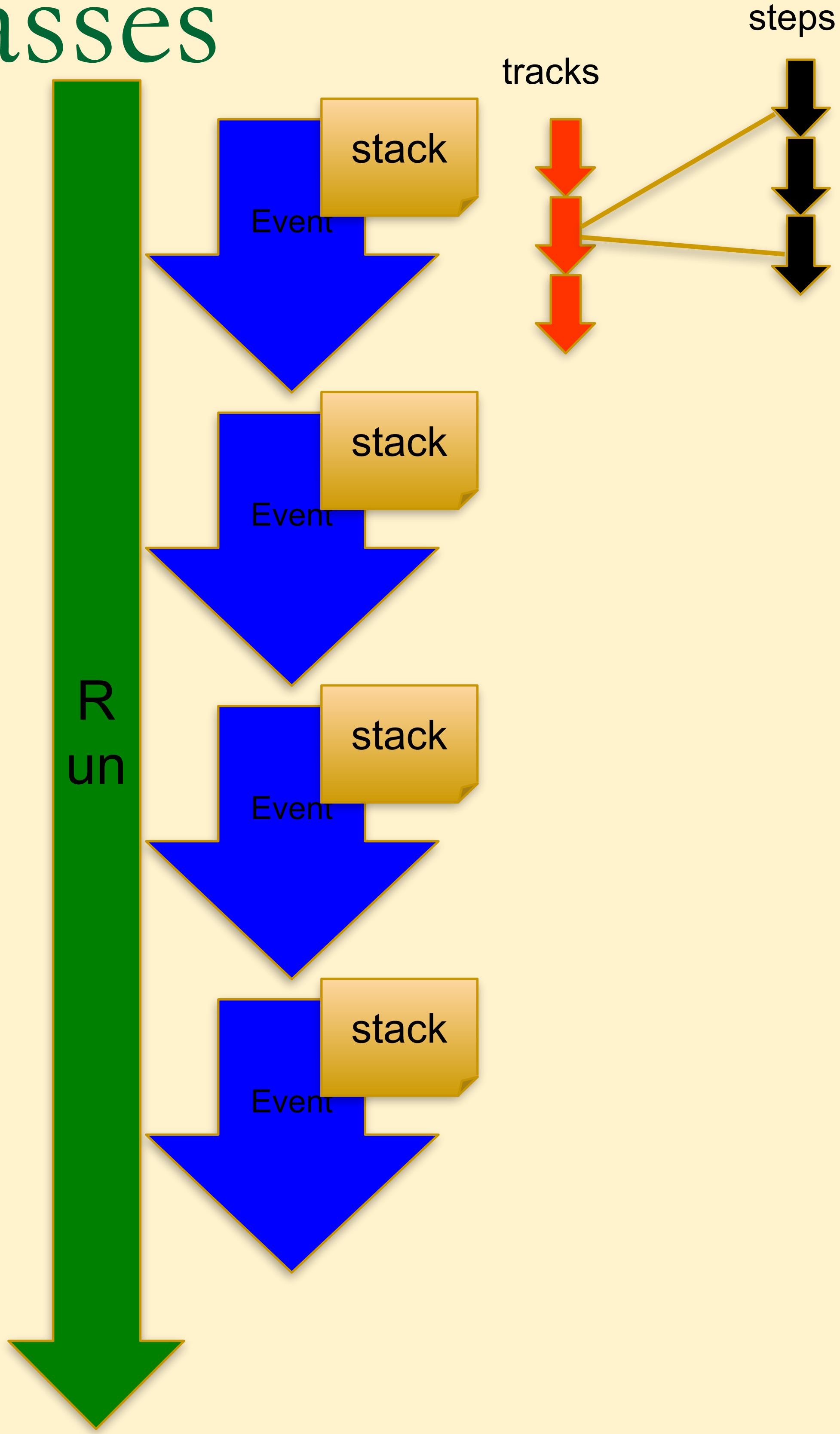


Track in Geant4

- Track is a snapshot of a particle.
  - It has physical quantities of current instance only. It does not record previous quantities.
  - Step is a “delta” information to a track. Track is not a collection of steps. Instead, a track is being updated by steps.
- Track object is deleted when
  - it goes out of the world volume,
  - it disappears (by e.g. decay, inelastic scattering),
  - it goes down to zero kinetic energy and no “AtRest” additional process is required, or
  - the user decides to kill it artificially.
- No track object persists at the end of event.
  - For the record of tracks, use trajectory class objects.
- G4TrackingManager manages processing a track, a track is represented by G4Track class.
- G4UserTrackingAction is the optional user hook.

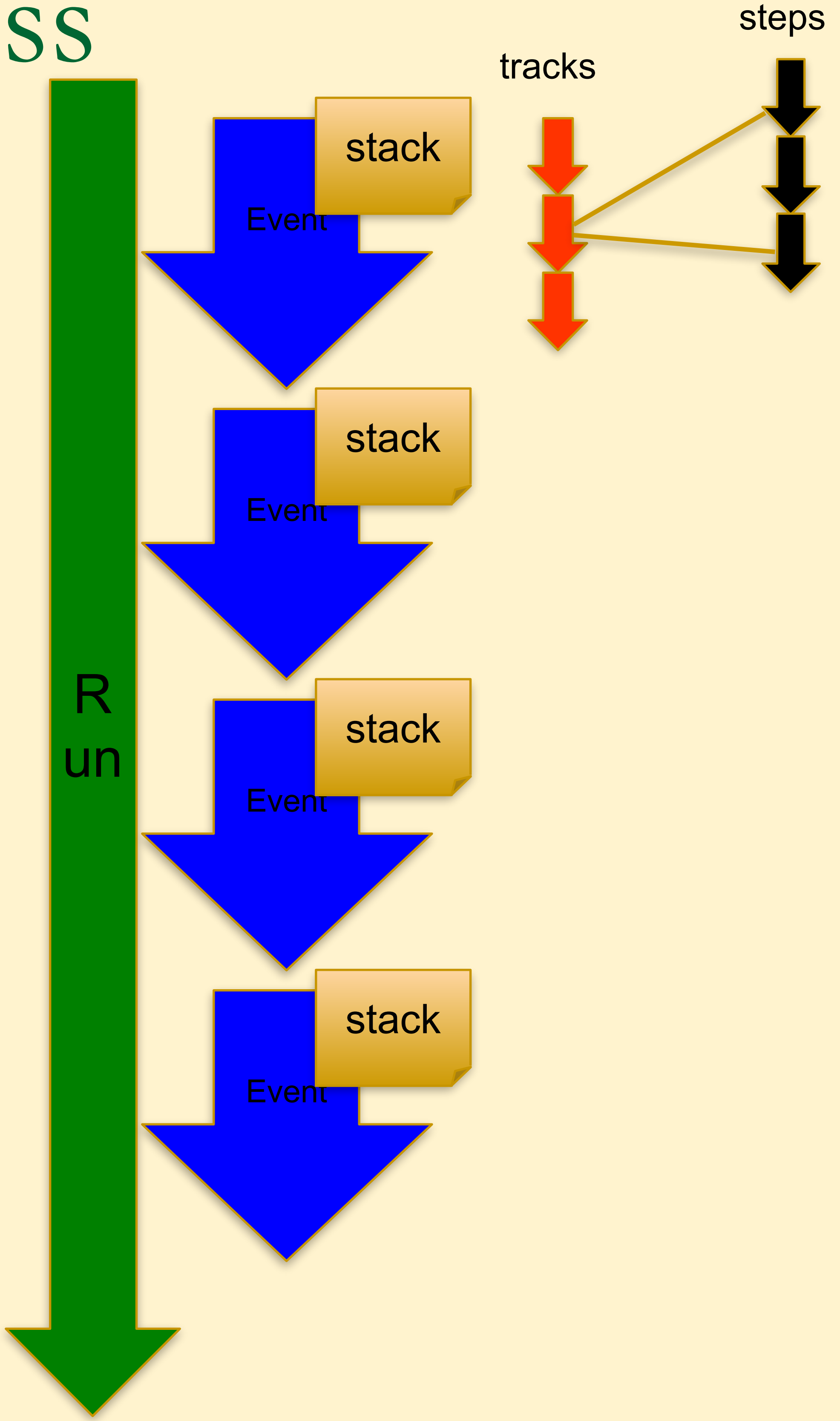
Tracking User Action Classes

- G4UserTrackingAction
  - void PreUserTrackingAction(const G4Track*)
    - Decide if trajectory should be stored or not
    - Create user-defined trajectory
  - void PostUserTrackingAction(const G4Track*)
    - Delete unnecessary trajectory



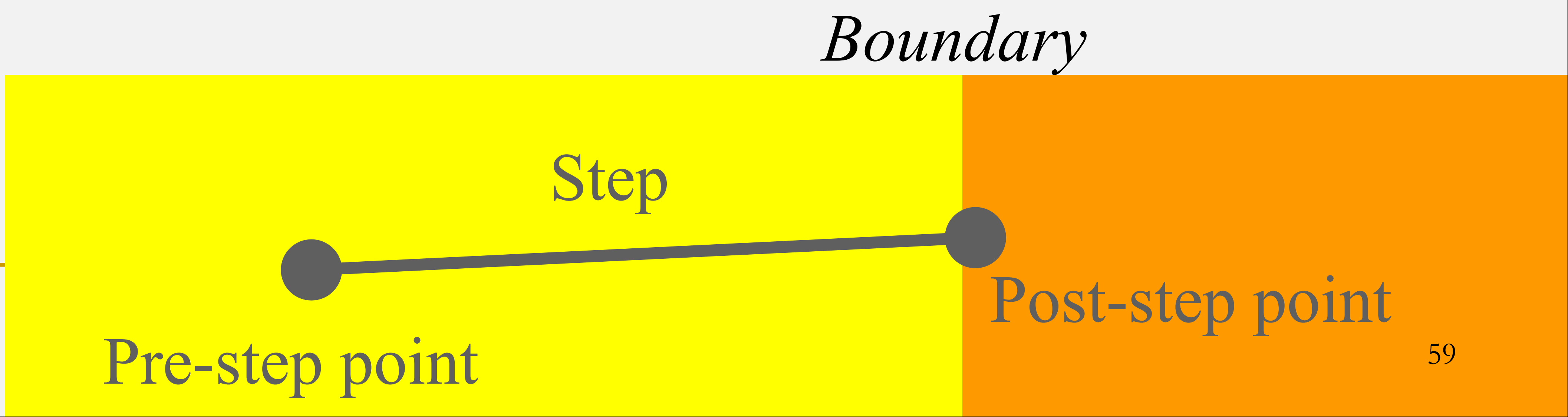
Stacking User Action Class

- G4UserStackingAction
  - G4ClassificationOfNewTrack ClassifyNewTrack(const G4Track*)
    - Invoked every time a new track is created, ie. Pushed to the stack
    - Classify a new track -- priority control
      - Urgent, Waiting, PostponeToNextEvent, Kill
  - Manipulate track stack,
    - void PrepareNewEvent()
      - Reset priority control
    - void NewStage()
      - Invoked when the Urgent stack becomes empty
      - Change the classification criteria
      - Event filtering (Event abortion)



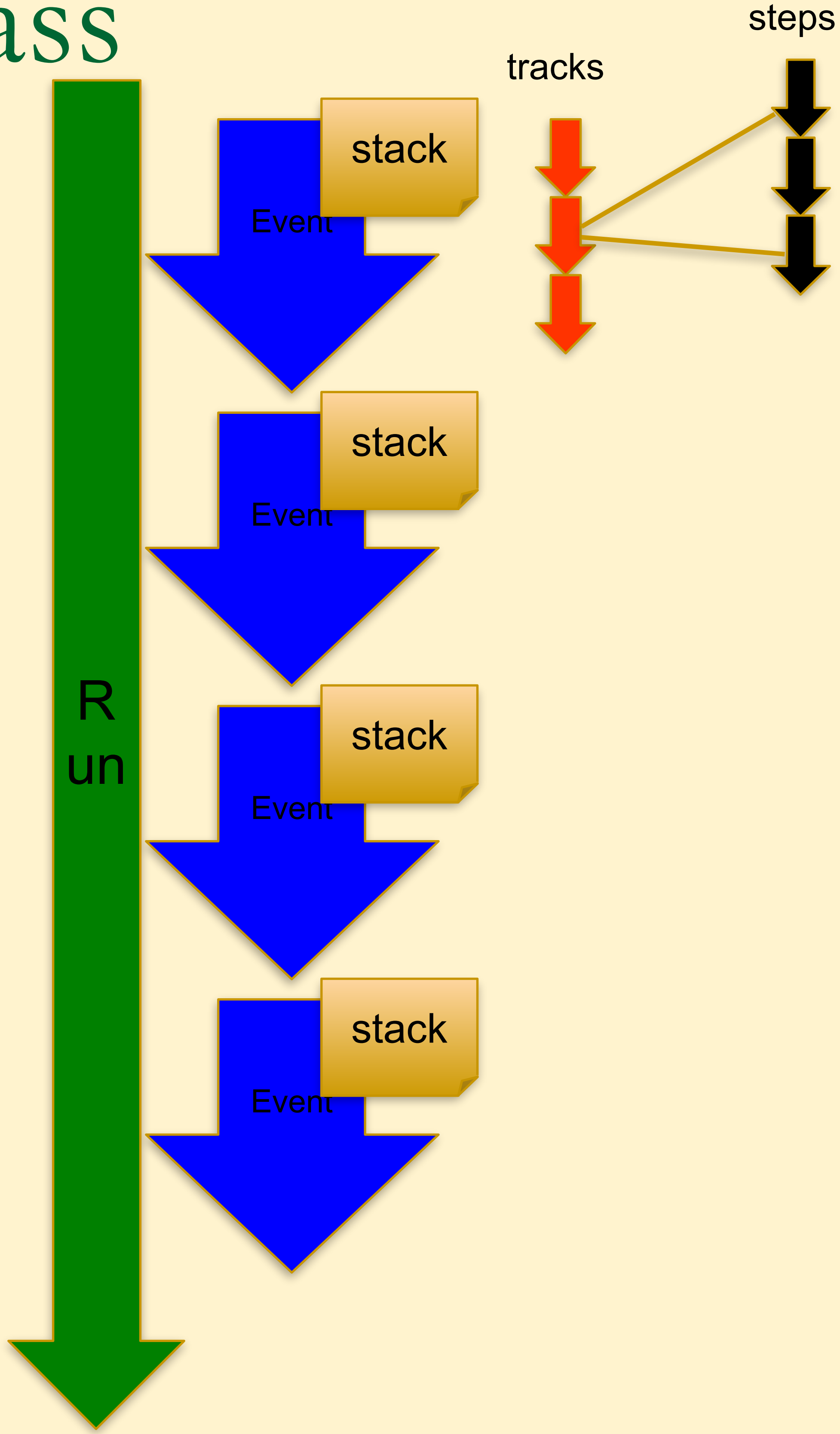
Step in Geant4

- Step has two points and also "delta" information of a particle (energy loss on the step, time-of-flight spent by the step, etc.).
  - Point is represented by G4StepPoint class
- 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.
  - Because one step knows materials of two volumes, boundary processes such as transition radiation or refraction could be simulated.
- G4SteppingManager class manages processing a step, a step is represented by G4Step class.
- G4UserSteppingAction is the optional user hook.



Stepping User Action Class

- G4UserSteppingAction
  - void UserSteppingAction(const G4Step*)
    - Change status of track
      - Kill / suspend / postpone the track
    - Draw the step (for a track not to be stored as a trajectory)



Recap – User action classes

- All needed UserAction classes
  - must be constructed in main()
  - must be provided to the RunManager using SetUserAction() method
- One mandatory User Action class
  - Event generator must be provided
  - Event generator class must be derived from G4VUserPrimaryGeneratorAction
- List of optional User Action classes
  - G4UserRunAction
  - G4UserEventAction
  - G4UserTrackingAction
  - G4UserSteppingAction
  - G4UserStackingAction

Geant4 ‘Scoring’

- Retrieving information from Geant4 using scoring
- Command-based scoring
- Add a new scorer/filter to command-based scoring
- Define scorers in the tracking volume
- Accumulate scores for a run

covered here

not covered here

Extract useful information - reminder

- Given geometry, physics and primary track generation, Geant4 does proper physics simulation “silently”.
  - You have to add a bit of code to extract information useful to you.
- There are three ways:
  - Assign G4VSensitiveDetector to a volume to generate “hit”.
    - Use user hooks (G4UserEventAction, G4UserRunAction) to get event / run summary

    Covered before
  - Built-in scoring commands
    - Most commonly-used physics quantities are available.

    Covered here
  - Use scorers in the tracking volume
    - Create scores for each event
    - Create own Run class to accumulate scores

    Not covered here
- You may also use user hooks (G4UserTrackingAction, G4UserSteppingAction, etc.)
  - You have full access to almost all information
  - Straight-forward, but do-it-yourself

  Covered before

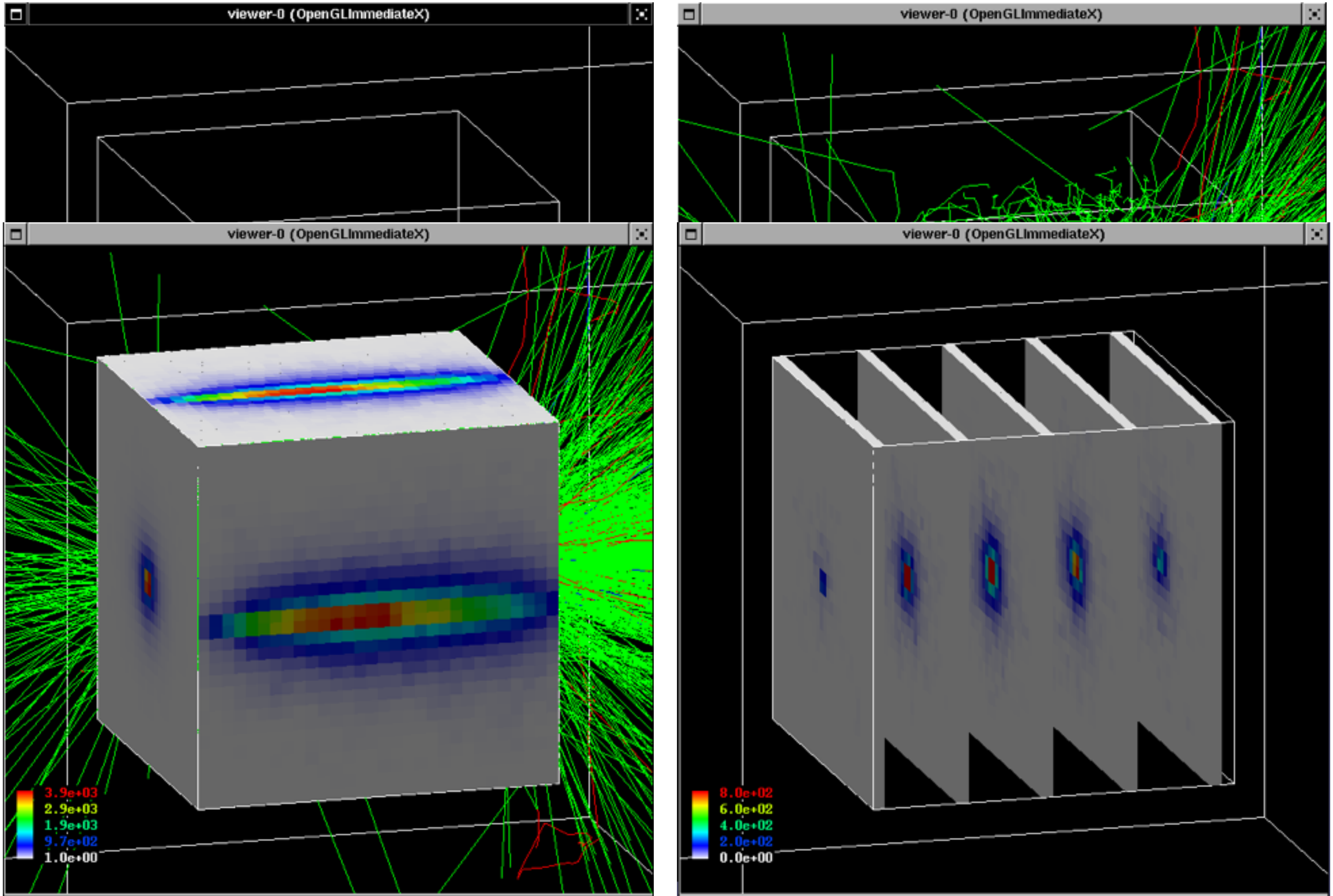
Command-based scoring

- Command-based scoring functionality offers the built-in scoring mesh and various scorers for commonly-used physics quantities such as dose, flux, etc.
- To use this functionality, access to the G4ScoringManager pointer after the instantiation of G4RunManager in your *main*().

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

- All of the UI commands of this functionality is in /score/ directory.
- /examples/extended/runAndEvent/RE03

/example/extended/runAndEvent/RE03



Define a scoring mesh

- To define a scoring mesh, the user has to specify the followings.
  1. Shape and name of the 3D scoring mesh. Currently, box is the only available shape.
     - Cylindrical mesh also available as a beta-release.
  2. Size of the scoring mesh. Mesh size must be specified as "half width" similar to the arguments of G4Box.
  3. Number of bins for each axes. Note that too many bins causes immense memory consumption.
  4. Optionally, 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
```

- The mesh geometry can be 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/quantitly/xxxxx   <scorer_name>

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/quantity/energyDeposit eDep
/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+
/score/close

Same primitive scorers with different filters may be defined.

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>

- 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.

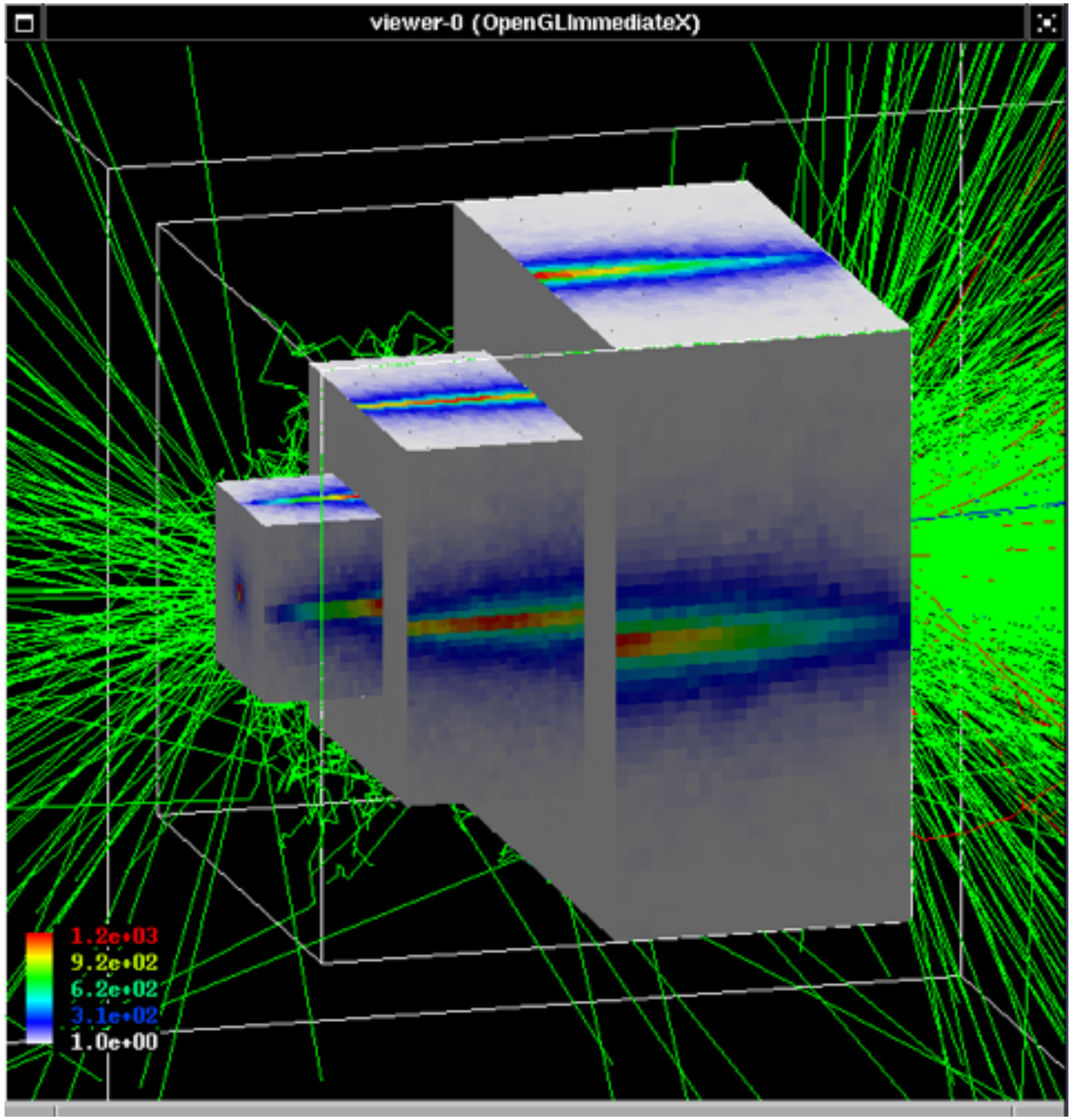
Write scores to a file

- 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 any boundary.
- Please be cautious of too many meshes, too granular meshes and/or too many primitive scorers.
  - Memory consumption
  - Computing speed



Summary

- Sensitive Detectors create ‘hits’
- User action classes allow user to control simulation or get information and results
  - Action classes for event generation, run, event, track, and step
- Ready-to-use scoring can be used to calculate different quantities (flux, etc)

Attaching User Information to selected Geant4 classes

Attaching user information to some Geant4 kernel classes

- Abstract classes
  - You can use your own class derived from provided base class
  - G4Run, G4VTrajectory, G4VTrajectoryPoint
    - Other examples: G4VHit, G4VDigit
- Concrete classes
  - You can attach a user information object
    - G4Event - G4VUserEventInformation
    - G4Track - G4VUserTrackInformation
    - G4PrimaryVertex - G4VUserPrimaryVertexInformation
    - G4PrimaryParticle - G4VUserPrimaryParticleInformation
    - G4Region - G4VUserRegionInformation
  - User information object is deleted when associated Geant4 object is deleted.
  - Objects are managed, but not used by Geant4

UserInformation classes (1)

- G4VUserEventInformation
  - Additional data user wants to store for the event
    - Only Print() method is required
  - User needs to register an instance in his G4UserEventAction class indirectly with G4Event
  - Using G4EventManager::SetUserInformation(G4VUserEventInformation * ..)
      - Cannot register directly in G4Event, as this is a const pointer
    - Get previously registered object using GetUserInformation() from G4Event or G4EventManager
  - Object is deleted when G4Event object is deleted

UserInformation classes (2)

- G4VUserTrackInformation
  - Data user want to keep for track, and not in trajectory
    - Only Print() method is required
  - Pointer to UserInformation object is kept in G4Track
    - should be set from G4UserTrackingAction indirectly via
    - G4TrackingManager::SetUserInformation(G4VUserTrackInformati on * ..)
      - Cannot register directly in G4Track, as this is a const pointer
    - Get previously registered object using GetUserInformation() from G4Track or G4TrackManager
  - Object is deleted when G4Track object is deleted

UserInformation classes (3)

- G4VUserPrimaryVertexInformation
  - Attach information to G4PrimaryVertex
- G4VUserPrimaryParticleInformation
  - Attach information to G4PrimaryParticle
- G4VUserRegionInformation
  - Attach information to G4Region
- Us Set/Get-UserInformation methods in G4PrimaryVertex, …, to attach object.