

# Simulation

Muography(w/ Dr. L. Cimmino)

Marwa Al Moussawi

Centre for Cosmology, Particle Physics and Phenomenology (CP3),  
Université Catholique de Louvain, Belgium

September 24, 2021

# Overview

## 1. CRY

## 2. Geant4

Detector Construction

Sensitive Detector and hits

Generator

## 3. Results

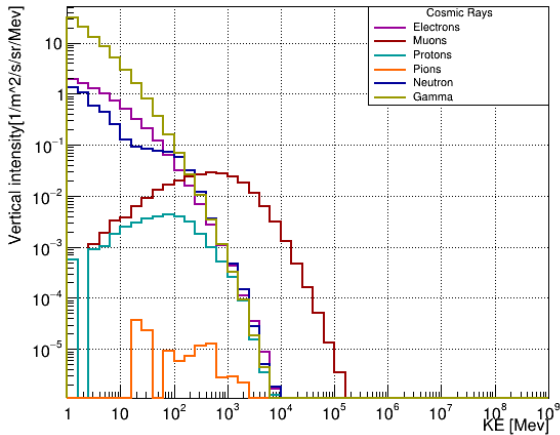
## Definition(Cosmic-ray Shower Library)

A Monte Carlo simulation used to follow :

- The tracks of all relevant secondary particles (neutrons, muons, gammas, electrons, and pions)
- Their fluxes at selectable altitudes(sea level, 2100 m, and 11300 m) and latitude
- The zenith and azimuth angle of those particles

```
returnNeutrons 1
returnProtons 1
returnGammas 1
returnKaons 1
returnPions 1
returnElectrons 1
returnMuons 1
date 7-1-2012 # month-day-year(solar activity effect)
latitude 48.0 # depend from the region(magnetic field effect)
altitude 0 # 0,2100,11300 m
subboxlength 0.16 # this quantity is chose with respect to your detector(maximum value = 300m)
```

Setup file CRY



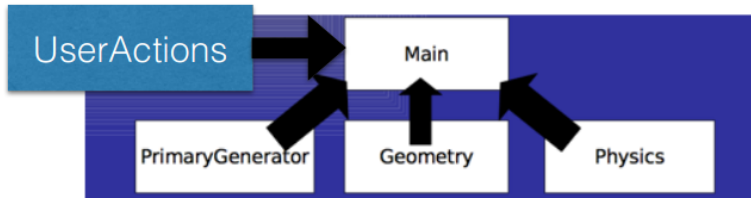
Flux of secondary at sea level

# Geant4

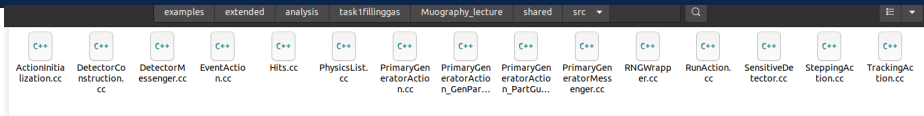
## Geant4: **GE**ometry **ANd** **T**racking

### Definition

- A platform for "the simulation of the passage of particles through matter" using Monte Carlo methods
- Includes facilities for handling geometry, tracking, detector response, run management, visualization and user interface
- We can linked to different software: CRY, Garfield++,..



# Basics



- **Action Initialisation:** Should contain one mandatory class to define my generator
- **Detector Construction:** Define my geometry and my sensitive detector
- **Event Action:** Collect my event to the Hit and used to fill my histogram
- **Hits:** Related to my hit information
- **Physics List:** To construct my process, define particle and make the cuts
- **Primary generator:** Define my generator
- **Run Action:** Save my variable and make the calculation
- **Sensitive Detector:** Define the quantity related to my hit by using the steps
- **Stepping and Tracking Action:** To know information on the particles (Track(parent..), position... ) inside your detector
- **ROOTManager:** Create my root output file

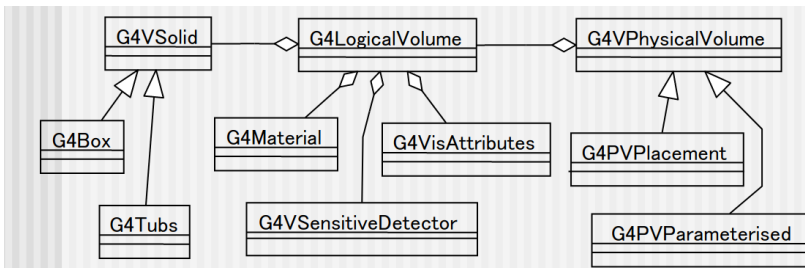
## How to Define a Detector Geometry

- A detector geometry in Geant4 is made of a number of volumes
- The largest volume is called the World volume:
  - Contain all other volumes in the detector geometry
  - The other volumes are created and placed inside it
  - The most simple (and efficient) shape to describe the World is a box
- Each volume is created by describing its shape and its physical characteristics, and then placing it inside a containing volume

# Create a Simple Volume

What do you need to do to create a volume?

- **G4VSolid:** Create a solid (shape, dimension)
- **G4LogicalVolume:** Create a logical volume (material, sensitivity, magnetic field, etc)
- **G4VPhysicalVolume:** Create a physical volume (position, rotation, copy...)



## Step 1

Create the  
geom. object :  
box

## Step 2

Assign properties  
to object :  
material

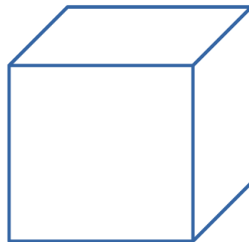
## Step 3

Place it in the  
coordinate  
system of  
mother volume

# Detector Construction

- **Shape and size**
- Logical volume, material, sensitivity, etc
- Physical volume, rotation and position

```
G4Box *USolid =  
    new G4Box("UBlock", //its name  
              (20/2)*cm, //its size  
              (20/2)*cm,  
              (30/2)*cm);
```



## Step 1

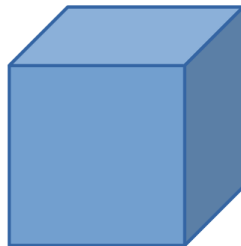
Create the  
geom. object :  
box



# Detector Construction

- Shape and size
- **Logical volume, material, sensitivity, etc**
- Physical volume, rotation and position

```
G4Box *USolid =  
    new G4Box("UBlock", //its name  
              (20/2)*cm, //its size  
              (20/2)*cm,  
              (30/2)*cm);  
  
G4Material* Uranium_mat = man->FindOrBuildMaterial("G4_U");  
G4LogicalVolume *ULog =  
    new G4LogicalVolume(USolid, //its solide  
                        Uranium_mat, //its material  
                        "logicUBlock"); //its name
```



## Step 1

Create the  
geom. object :  
box

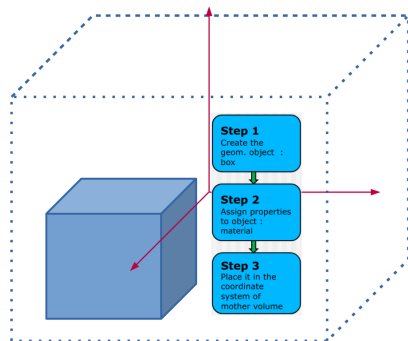
## Step 2

Assign properties  
to object :  
material

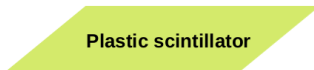
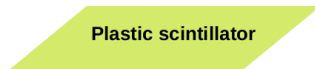
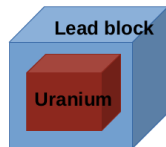
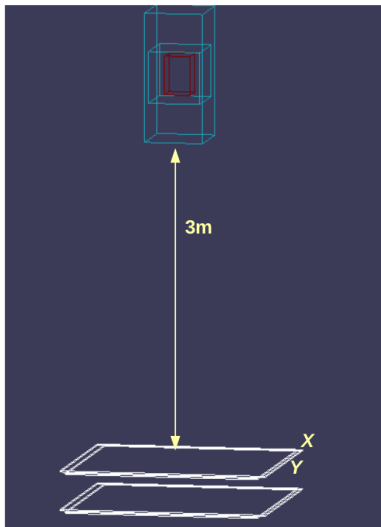
# Detector Construction

- Shape and size
- Logical volume, material, sensitivity, etc
- **Physical volume, rotation and position**

```
G4Box *USolid =  
    new G4Box("UBlock", //its name  
              (20/2)*cm, //its size  
              (20/2)*cm,  
              (30/2)*cm);  
  
G4Material* Uranium_mat = man->FindOrBuildMaterial("G4_U");  
G4LogicalVolume *ULog =  
    new G4LogicalVolume(USolid, //its solide  
                        Uranium_mat, //its material  
                        "logicUBlock"); //its name  
  
G4ThreeVector posU = G4ThreeVector(0*cm, 0*cm, - 3*m);  
new G4PVPlacement(0, //no rotation  
                  posU, //at position  
                  ULog, //its logical volume  
                  "PhysUBlock", //its name  
                  logmother, //its mother volume  
                  false, //no boolean operation  
                  0); //copy number
```



# My geometry example



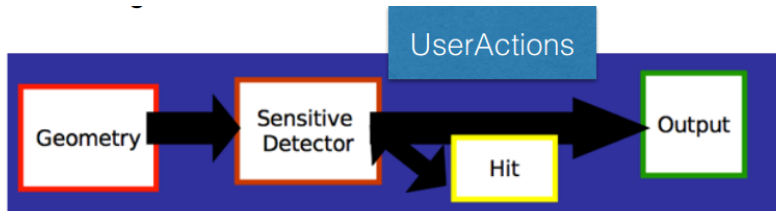
# Hit and sensitive detector

## What is a Sensitive Detector?

### SD

A **SD** can be used to simulate the “read-out” of your detector:

- A way to declare a geometric element “sensitive” to the passage of particles, for example: scintillator bar, gas-gap(for RPC)..
- Gives the user a handle to collect quantities from these elements. For example: energy deposited, position, time information ...



# Hit and sensitive detector

To create a SD you need to:

- **Write your SensitiveDetector class**
- Attach it to a logical volume

```
1 #ifndef SensitiveDetector_h
2 #define SensitiveDetector_h 1
3
4 #include "G4VSensitiveDetector.hh"
5 #include "Hits.hh"
6
7 class G4Step;
8 class G4HCoFThisEvent;
9 class G4TouchableHistory;
10
11 class StripSD : public G4VSensitiveDetector
12 {
13 public:
14   StripSD( const G4String &SDname );
15   virtual ~StripSD();
16
17   virtual void Initialize( G4HCoFThisEvent *hitcollection );
18   virtual G4bool ProcessHits( G4Step *step, G4TouchableHistory *history );
19   virtual void EndOfEvent( G4HCoFThisEvent* hitCollection );
20
21 private:
22   StripHitsCollection* fHitsCollection;
23   G4int fHCID;
24 };
25
26 #endif
```

- Add include:  
**G4VSensitiveDetector.hh**
- Constructor: SD are named
- Initialization: called at beginning of event
- Called for each **G4Step**(Snapshot of the interaction of a G4Track (particle) with a volume) in sensitive volume
- Finalize: called at end of event

# Hit and sensitive detector

To create a SD you need to:

- Write your SensitiveDetector class
- **Attach it to a logical volume**

```
void DetectorConstruction::ConstructSDandField()
{
    G4SDManager::GetSDMpointer()->SetVerboseLevel(1);
    G4String SDname;

    // approach with hits
    auto Strip = new StripSD(SDname="/Strip");
    G4SDManager::GetSDMpointer()->AddNewDetector(Strip);

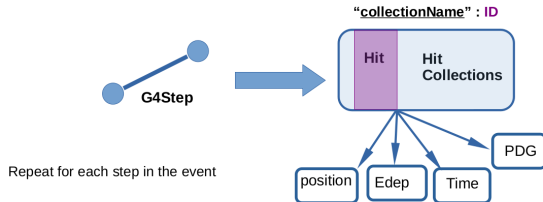
    GetScoringVolume()->SetSensitiveDetector(Strip);
}
```

- **GetScoringVolume:** return the logicalVolume for the plastic scintillator define in our example geometry

# Hit and sensitive detector

## Hit

- You need to write your own Hit class: inherits from G4VHit: **Hit.hh**
- Hits are created in Sensitive Detector to store user quantities:  
**SensitiveDetector.cc** & **EventAction.cc**
- 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, etc: **SensitiveDetector.cc**



```
//My specific variable
void SetStationID( G4int z ) { fStationID = z; };
G4int GetStationID() const { return fStationID; };

void SetEdep( G4double de ) { fEdep = de; };
void SetKe( G4double de ) { fKe = de; };
void SetPz( G4double dp ) { fPz = dp; };
void SetPhi( G4double dp ) { fPhi = dp; };

void AddEdep( G4double de ) { fEdep += de; };
G4double GetEdep() const { return fEdep; };
G4double GetKe() const { return fKe; };
G4double GetPz() const { return fPz; };
G4double GetPhi() const { return fPhi; };

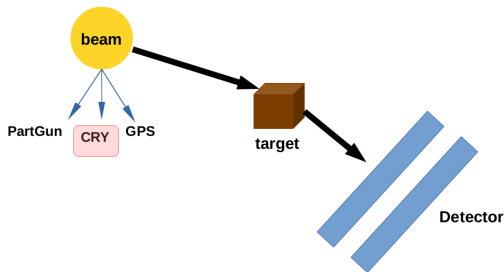
void SetPos( G4ThreeVector xyz ) { fPos = xyz; };
G4ThreeVector GetPos() const { return fPos; };

void SetTrackID( G4int id ) { fTrackID = id; };
G4int GetTrackID() const { return fTrackID; };
```

# Generator

- After we are done by define the sensitive part of our detector (collected the hit) and make the construction of our target, we need to chose our generator (muons) :

**PrimaryGeneratorAction.cc**



- Another generator can be integrated with Geant4: CORSIKA, EcoMug

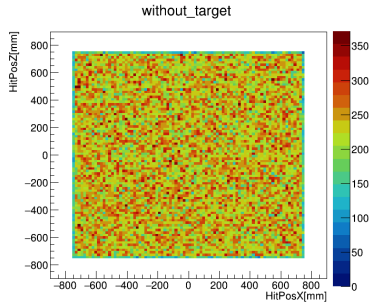


## Generators Geant4

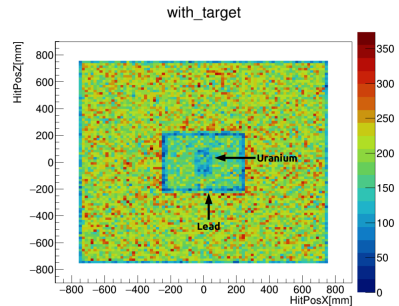
- **PartGun:** G4ParticleGun is a generator provided by Geant4. This class generates primary particle(s) with a given momentum and position
- **GPS:** The G4GeneralParticleSource (GPS) is part of the Geant4 toolkit for Monte-Carlo, high-energy particle transport, GPS allows the user to control the following characteristics of primary particles:
  - Spatial sampling (2D or 3D surfaces such as discs, spheres, and boxes)
  - Angular distribution (unidirectional, isotropic, cosine-law, beam or arbitrary..)
  - Spectrum (linear, exponential, power-law, Gaussian,..)
- **CRY:** Real flux generator (we need to link it to our Geant4 code):
  - Shape: flat surface
  - Energy: real spectrum of energy
  - Momentum direction: related to zenith and azimuthal angle
- We need to save all the information in a root output (using G4AnalysisManager, ROOTManager, HistoManager): **ROOTManager.cc** & **EventAction.cc**

# Results

- Using the Hit position information for the muon event that cross all the detector (using the id of plan)



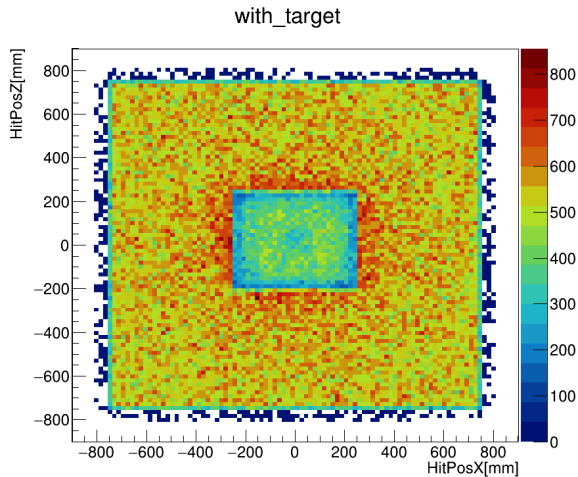
2D hit position for free sky



2D hit position using Partgun with rang of energy from 0 to 3GeV

- We can make the ratio to calculate the transmission, but we need to normalize for have the same number of events for the two cases

# Results



2D hit position using CRY as generator

The End