Geant4

Osvaldo Miguel Colín

Universidad Iberoamericana

What is the use of Monte Carlo techniques?

- Numerical solution of a (complex) macroscopic problem, by simulating smaller interactions.
- It's not only constrained to physics, it can be used for many other types of research such as: finances, traffic or even social studies.
  - It can apply for numeric problems as well (e.g. Integrals) not necessarily probabilistic problems.
- Generate configurations or arrangements.
  - People like to have a good view hence they will chose seats with the best view.
  - People prefer to sit in the aisle seat so the can access the bathroom easier.
  - Only one person can occupy any given seat.
- The analytic solution can be impossible or extremely difficult to solve, i.e. MC is more effective.

What is the use of Monte Carlo techniques?

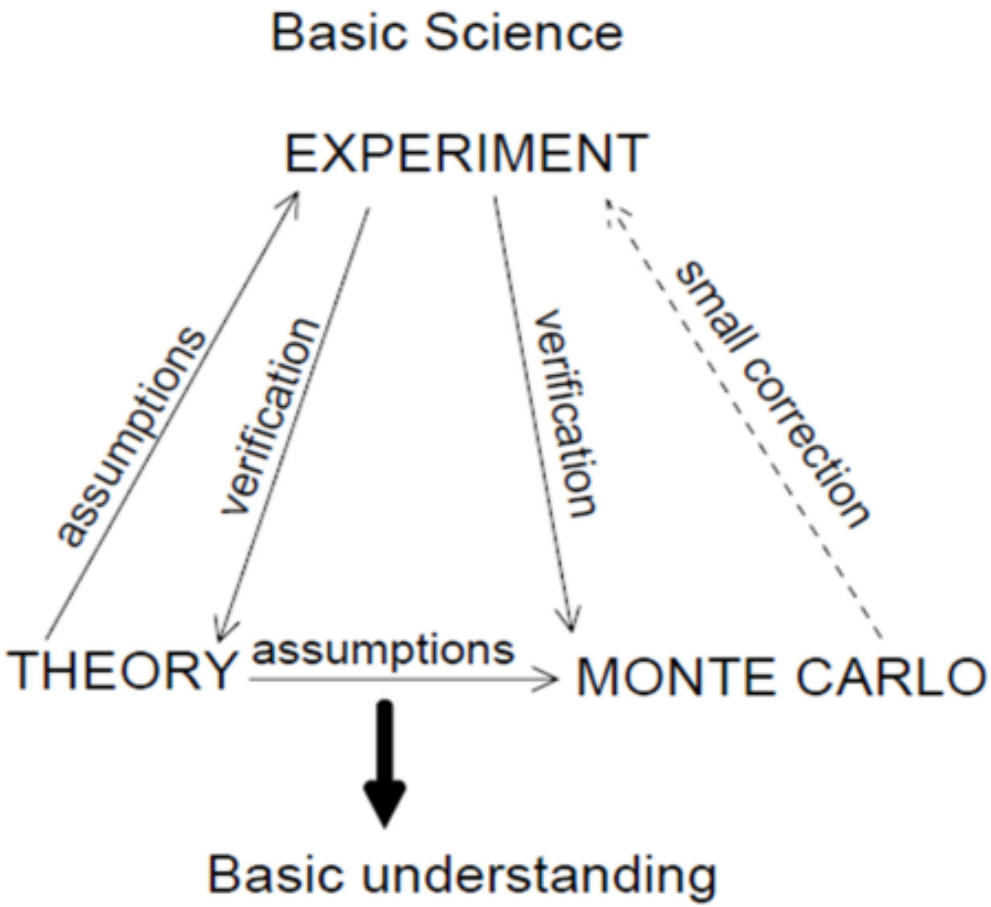
- In MC you don't consider laws to be laws, rather they are considered working assumptions.
- Here we use reverse engineering if you will. By generating random distributions according to our assumptions, rules and so on, we can find "laws" or patterns that better explain these behaviors.
    - Use MC to build a theory of a complex system by comparing it with real experiments.

Monte Carlo in science

- In physics, elementary laws are known (usually), MC is therefore used to predict the outcome of larger experiments.
- It can be used to prove or disprove a theory, and/or to provide small corrections to the theory.

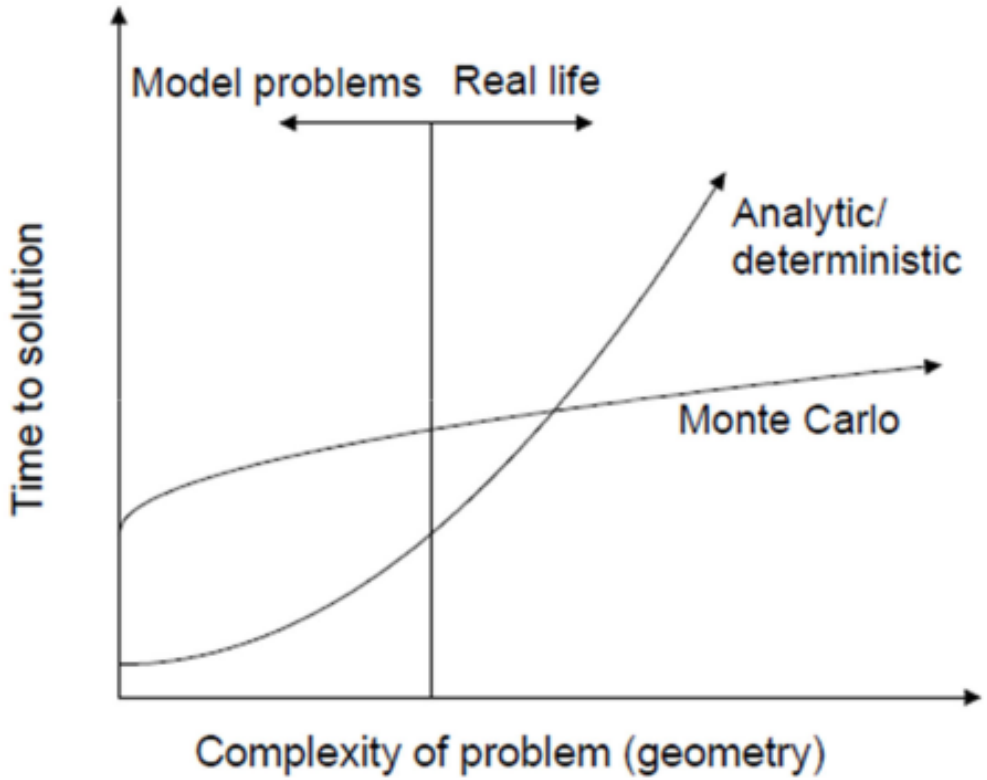
In this course we will use Monte Carlo for particle tracking.

Monte Carlo in science



Monte Carlo vs Analytic Solutions

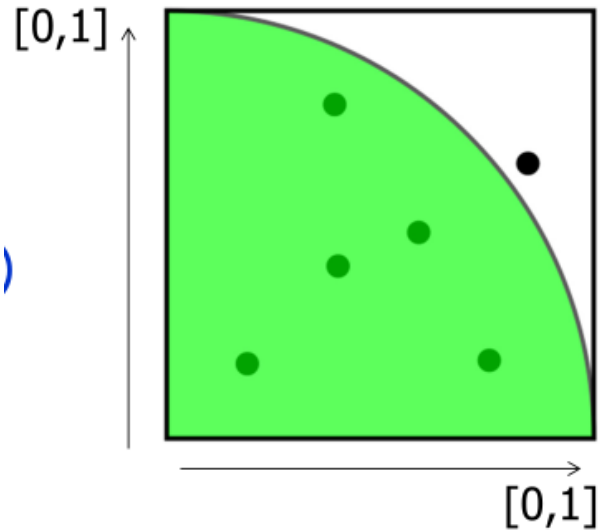
Simple. If you value you life and wish to find a solution before dying, then use Monte Carlo.

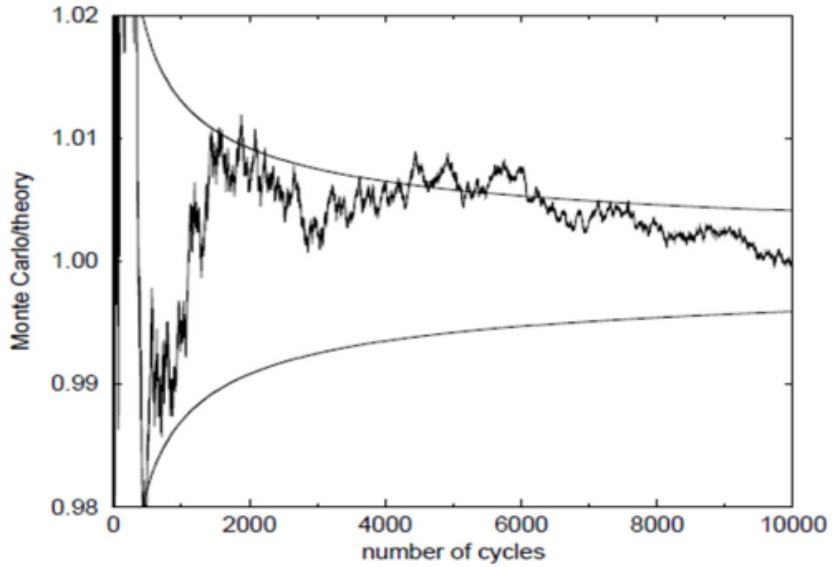


Simple MC application: Task 0

To calculate π you do the following:

- (N) Shoot N points in a fourth of a circle of radius 1.
- (n) Count how many of the points satisfy $x^2 + y^2 \leq 1$.
- $n/N = \pi/4$

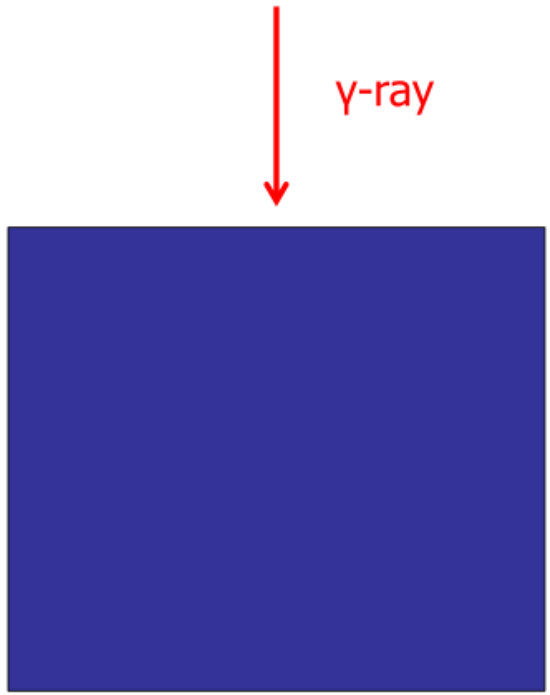




- Convergence as $\frac{1}{\sqrt{N}}$

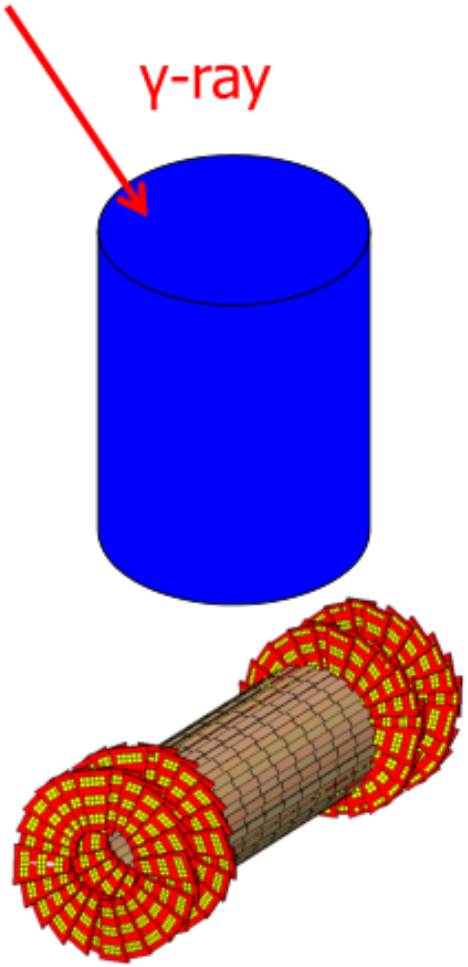
Common applications in physics

- Track a γ-ray in a semi-infinite detector and determine the energy spectrum deposited.
- All the physics we need are in textbooks (Compton scattering, photoelectric, etc..)
- Yet, the analytic solution is a nightmare.



Common applications in physics

- Track a γ-ray in a finite detector (e.g. a NaI).
- Analytic computation is nearly impossible.
  - Monte Carlo for the win.
- Now being realistic, modern detectors are much more complex than a simple cylinder, and have various components.



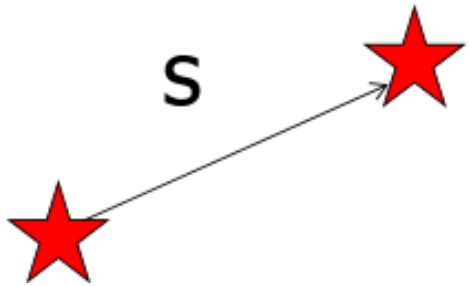
- The distance between two subsequent interactions (s) have the following distribution:

$$p(s) = \mu e^{-\mu s}$$

Where μ is the property of the medium (supposed to homogeneous) and of the physics considered.

- If a medium is not homogeneous then:

$$p(s) = \mu(s) e^{-\int_0^s \mu(s')ds'}$$



- Transition between two homogeneous materials goes as:

$$\mu(z) = \theta(b-z)\mu_1 + \theta(b-z)\mu_2$$

- μ is proportional to the total cross section and depends on the density of the material:

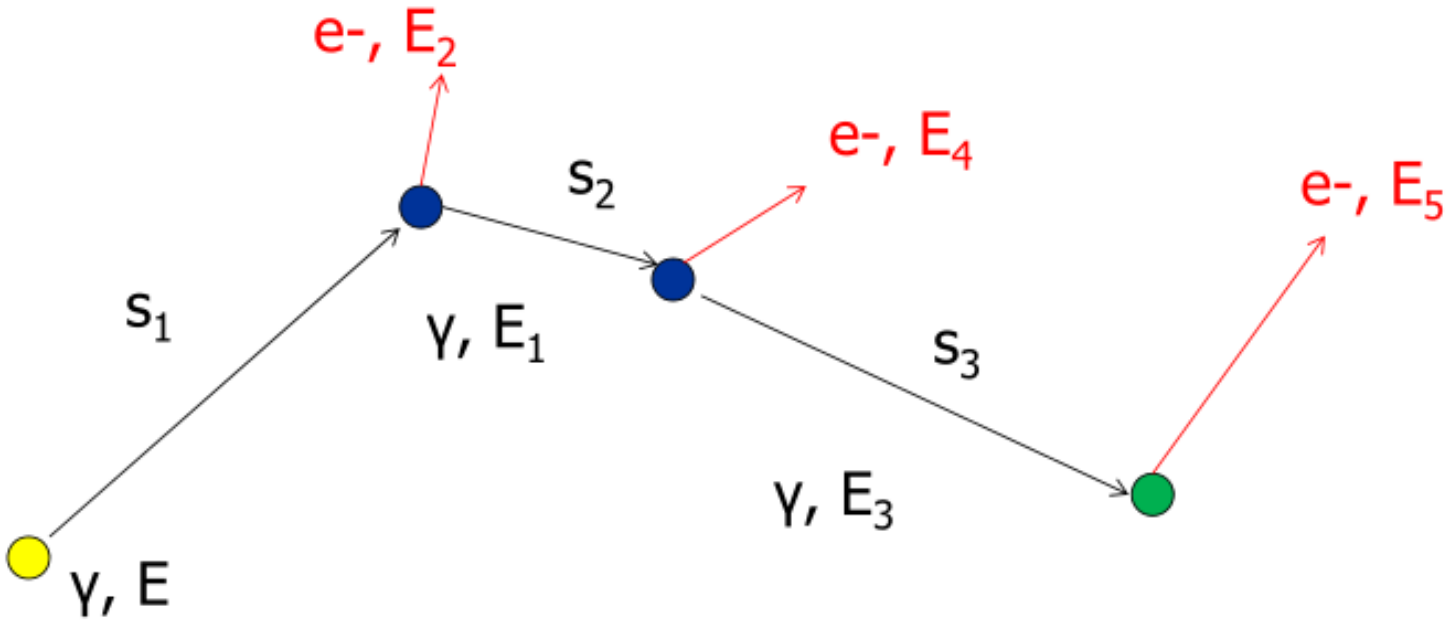
$$\mu = N\sigma = N\sum_i \sigma_i = \sum_i \mu_i$$

- All competing processes contribute with their own μ_i
- Each process takes place with probability μ_i/μ i.e. proportional to the total cross section.

Particle Tracking: Basic Recipe

- Divide the trajectory of the particle in "steps".
  - Straight free-flight tracks between consecutive interactions.
  - Steps can also be limited by geometric boundaries.
- Decide the step length s, by sampling according to $p(s) = \mu e^{-\mu s}$ (Non uniform random numbers, ask the professor so he can wow you of how this works), with the proper μ (Remember, this depends on the material and the physics involved).
- Decide which interaction takes place at the end of the step, according to μ_i/μ.
- Produce the final state according to the physics of the interaction.
  - Update direction of the primary particle.
  - Store somewhere the possible secondary particles, to be tracked later on.

Basic Recipe



- Follow all secondaries, until they are absorbed or leave the volume.
- Notice: μ depends on energy (cross sections do!)

Basic Recipe

- This basic recipe works fine for γ-rays and other neutral particles.
- This doesn't work so well for $e^{\pm}$: the cross section (ionization & bremsstrahlung) is very high, so the steps between two consecutive interactions are very small.
  - CPU intensive: viable for low energies and this material.
- Even worse: in each interaction only a small fraction of energy is lost, and the angular displacement is small.
  - A lot of time is spent to simulate interactions having small effect.
  - The interactions of γ are "catastrophic": large change in energy and direction.

Mixed Monte Carlo

- Simulate explicitly (i.e. force step) interactions only if energy is lost (or change of direction) is above a threshold W_0
  - Detailed simulation
  - "hard" interaction (like γ interactions)
- The effect of all sub-threshold interactions is described statistically.
  - Condensed simulation
  - "soft" interactions
- Hard interactions occur much less than soft interactions.
  - you recover the fully detailed simulation when $W_0 = 0$

Mixed Monte Carlo

- Has some technical tricks:
  - Since energy is lost along the step due to soft interactions, the sampled step s cannot be too long ($s < s_{max}$).
- Parameter μ_h between hard collisions

$$\mu_h = N \int_{W_0}^{E} \frac{d\sigma}{dW}(E) dW$$

  - Has $\mu_h << \mu$ because the differential cross section is strongly peaked at low W (= soft secondaries)
  - "soft" interactions
- Much longer step length

- Stopping power due to soft collisions (dE/dx)

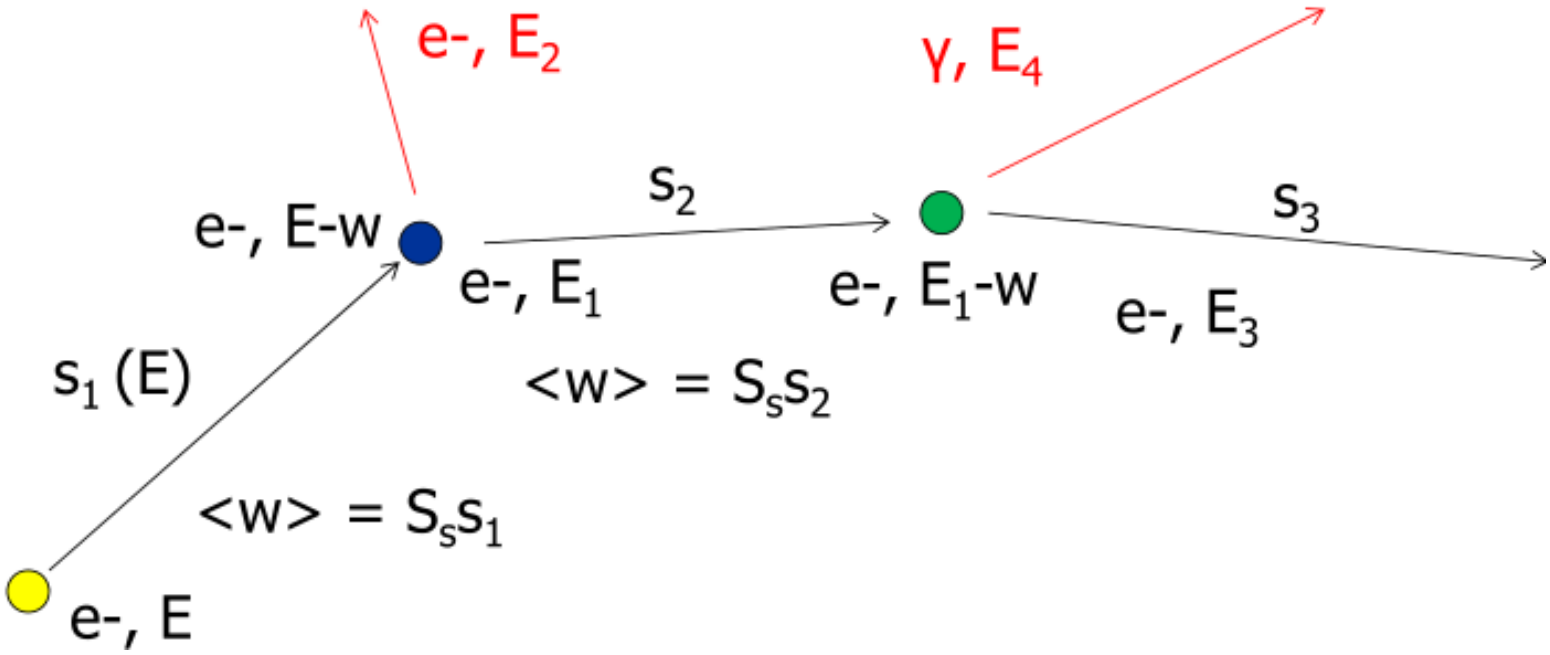
$$S_s = N \int_0^{W_0} W \frac{d\sigma}{dW}(E) dW$$

- Average energy lost along the step: $< w >= sS_s$
  - Must be $< w ><< E$
- Fluctuations around the average value $< w >$ have to be taken into account.
  - Appropriate random sampling of w with mean value $< w >$ and variance (straggling).

Extended recipe

- Decide the step length s, by sampling according to $p(s) = \mu_h e^{-\mu_h s}$
- Calculate the cumulative effect of the soft interactions along the step: sample the energy loss w, with $< w >= sS_s$, and the displacement.
- Update energy and directions of the primary particle at the end of the step $E_{i+1} = E - w$.
- Decide which interaction takes place at the end of the step, according to $\mu_{i,h}/\mu_h$
- Produce the final state according to the physics of the interaction ($\frac{d^2\sigma}{d\Omega dE}$).

Extended Recipe



- Follow all secondaries, until they are absorbed or leave the volume.

Toolkit and User Application

- It's a toolkit!
  - You can't run it as you would any other application.
  - You have to make use of the tools in order for it to run.
- Consequences:
  - There is no such thing as "Geant4 defaults".
  - You must provide the necessary information to configure you simulation.
  - You must deliberately choose which Geant4 tools to use.

- What you must do:
  - Describe your experimental set-up.
  - Provide the primary particles input to your simulation.
  - Decide which particles and physics models you want to use out of those available in Geant4 and the precision of your simulation (cuts to produce and track secondary particles).
- You may also want:
  - To interact with Geant4 kernel to control your simulation.
  - To visualize your simulation configuration or results.
  - To produce histograms, tuples, etc.. to be further analyzed.

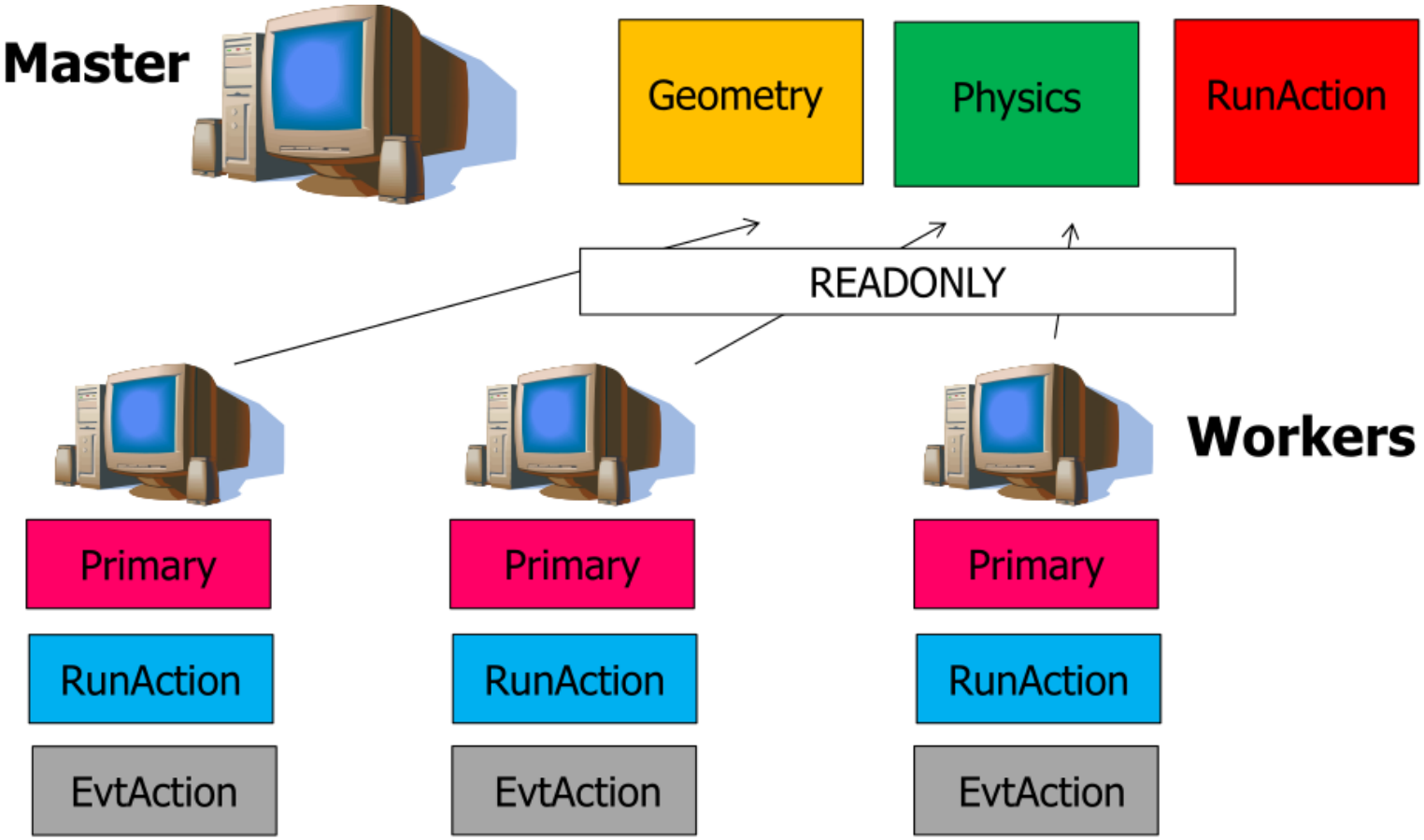
Main Geant4 Capabilities

- Transportation of a particle ‘step-by-step‘ taking into account all possible interactions with materials and fields.
- The transport ends if the particle
  - is slowed down to zero kinetic energy (and doesn't have any interaction at rest)
  - disappears in some interaction (annihilated)
  - reaches the end of the simulation volume.
- Geant4 allows the user to access the transportation process and retrieve the results (User Actions):
  - at the beginning and end of the transport (event).
  - at the end of each step in transportation.
  - if the particle reaches a sensitive detector.
  - Others...

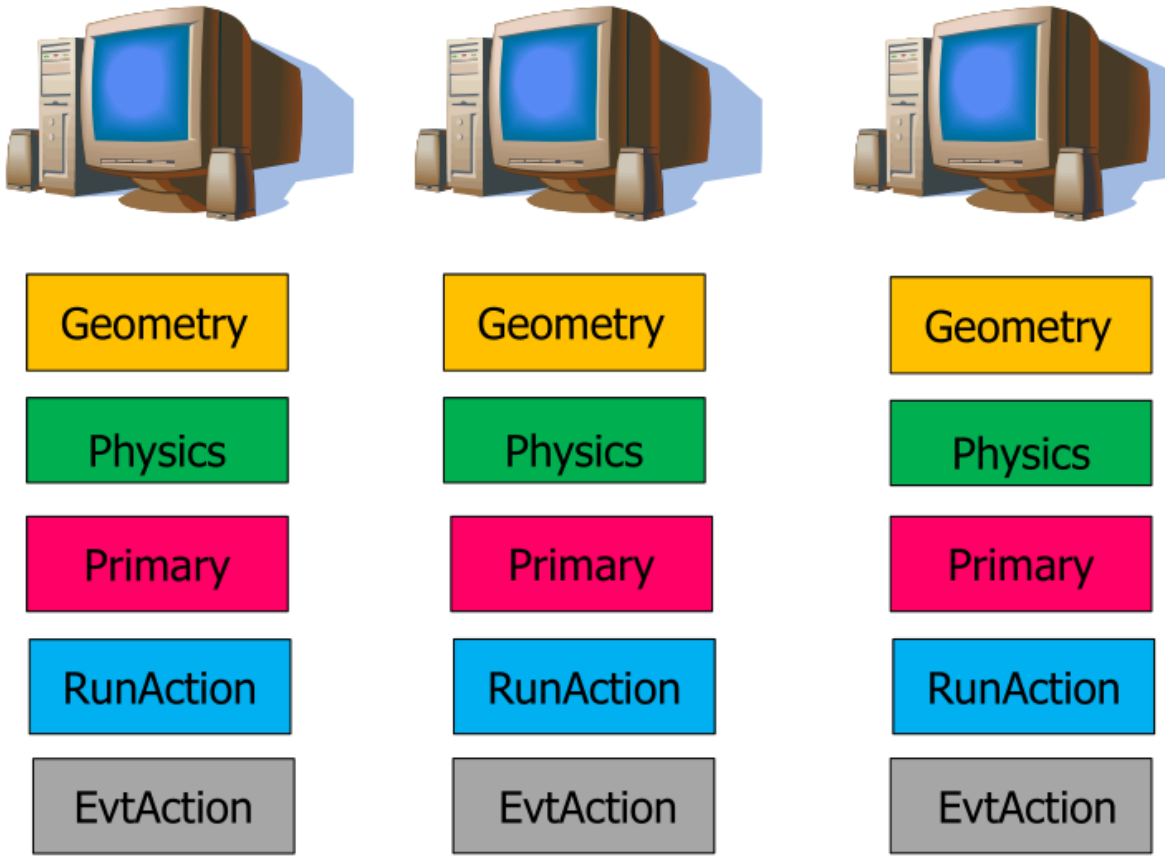
Multi-thread mode

- Geant4 10.0 (released Dec, 2013) supports multi-thread approach for multi-core machines.
  - Simulation is automatically split on an event-by-event basis.
    - different events are processed by different cores
  - Can fully profit of all cores available on modern machines. This means we obtain a substantial boost in our simulations.
  - Unique copy (master) of geometry and physics.
    - All cores have them with read-only access.
- Backwards compatible with the sequential mode.
  - Multi-thread programming requires thread-safe operations.

Multi-thread



Parallelism



How to work with Geant4

In order to use Geant4, you proceed as you would with any external library. In other words, you simply have to create an empty project using cmake and link the Geant4 library with your code.

Steps:

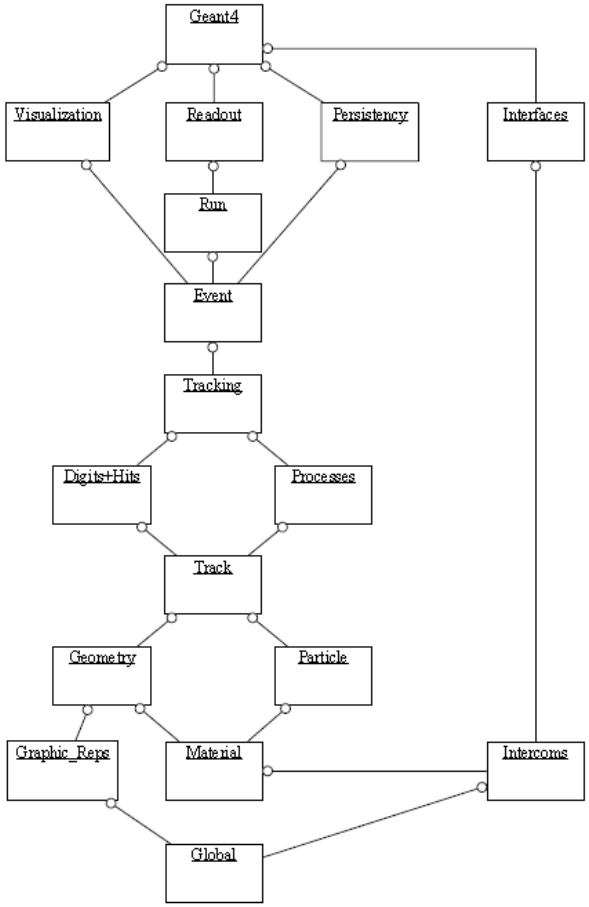
- Create an empty C++ project.
- Initialize Geant4 in main().
- Describe the geometry, primary particles, physics and other functionality.
- Compile the code
- Run Forest!! Run!

Modular Architecture

Geant4 consist of many modules:

- Run: management of the runs.
- Event: management of the events.
- Tracking: particle tracks in the geometry.
- Processes: physics attached to particles.
- Particle: elementary and other particles.
- Geometry: description of the detector.
- Material: all material properties.
- Interfaces: communication with the user.
- Visualization: graphical representation of the geometry and tracks.
- ... and much more.

Modular Architecture



Source Structure

Official basic/B1 example:

```
2,4K  4 Dic 14:48 CMakeLists.txt
475B  4 Dic 14:48 GNUmakefile
2,8K  4 Dic 14:48 History
7,5K  4 Dic 14:48 README
4,0K  4 Dic 14:48 exampleB1.cc
226B  4 Dic 14:48 exampleB1.in
 35K  4 Dic 14:48 exampleB1.out
272B  4 Dic 14:49 include
338B  4 Dic 14:48 init_vis.mac
553B  4 Dic 14:48 run1.mac
448B  4 Dic 14:48 run2.mac
272B  4 Dic 14:49 src
3,8K  4 Dic 14:48 vis.mac
```

The text file CMakeLists.txt is the CMake script containing commands which describe how to build the exampleB1 application

contains main() for the application

Header files

```
2,2K  4 Dic 14:48 B1ActionInitialization.hh
2,4K  4 Dic 14:48 B1DetectorConstruction.hh
2,4K  4 Dic 14:48 B1EventAction.hh
2,7K  4 Dic 14:48 B1PrimaryGeneratorAction.hh
2,5K  4 Dic 14:48 B1RunAction.hh
2,4K  4 Dic 14:48 B1SteppingAction.hh
```

Source files

```
2,9K  4 Dic 14:48 B1ActionInitialization.cc
7,7K  4 Dic 14:48 B1DetectorConstruction.cc
2,6K  4 Dic 14:48 B1EventAction.cc
4,3K  4 Dic 14:48 B1PrimaryGeneratorAction.cc
5,8K  4 Dic 14:48 B1RunAction.cc
3,2K  4 Dic 14:48 B1SteppingAction.cc
```

Macro file containing the commands

UNIVERSIDAD IBEROAMERICANA CIUDAD DE MÉXICO ®

How to add a new class

1. Select a class to inherit from (if there is one).
2. Find a good name for your class (Don't be the guy that calls it "MyAwesomeClass", really - don't be that guy.).
3. Create a header file inside the include folder (include/)
   - name it using the class name with a .hh extension.
   - define the class (inheriting from the base classes, if any).
   - declare methods to override and other methods.
4. Create a source file inside the src folder (src/)
   - name it using the class name with a .cc extension.
   - include the header file (#include).
   - define the class methods.

Typical header file

MyDetectorConstruction.hh

```
#ifndef MYDETECTORCONSTRUCTION_HH
#define MYDETECTORCONSTRUCTION_HH

#include <G4VUserDetectorConstruction.hh>

class MyDetectorConstruction
  : public G4VUserDetectorConstruction
{
public:
    MyDetectorConstruction();
    ~MyDetectorConstruction();

    G4VPhysicalVolume* Construct() override;
    void ConstructSDandField() override;

private:
    G4LogicalVolume* fLogVolume { nullptr };
};
#endif
```

Header protection (against multiple inclusions)

Include other header files (the base class + ...)

New class definition (corresponding to the file name!)

Inherit from a base class

Constructor (optional)

Virtual destructor (optional)

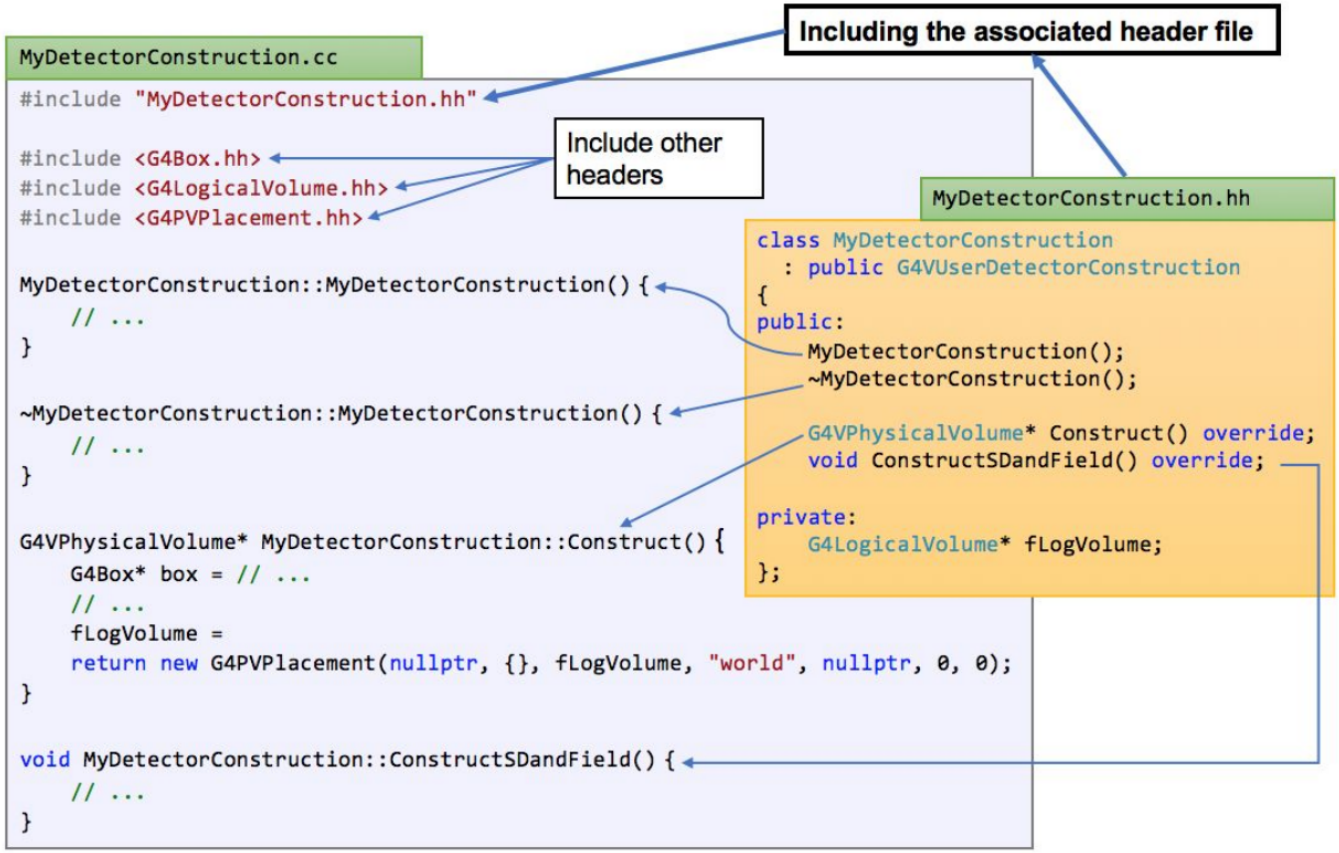
A member field

Override virtual functions

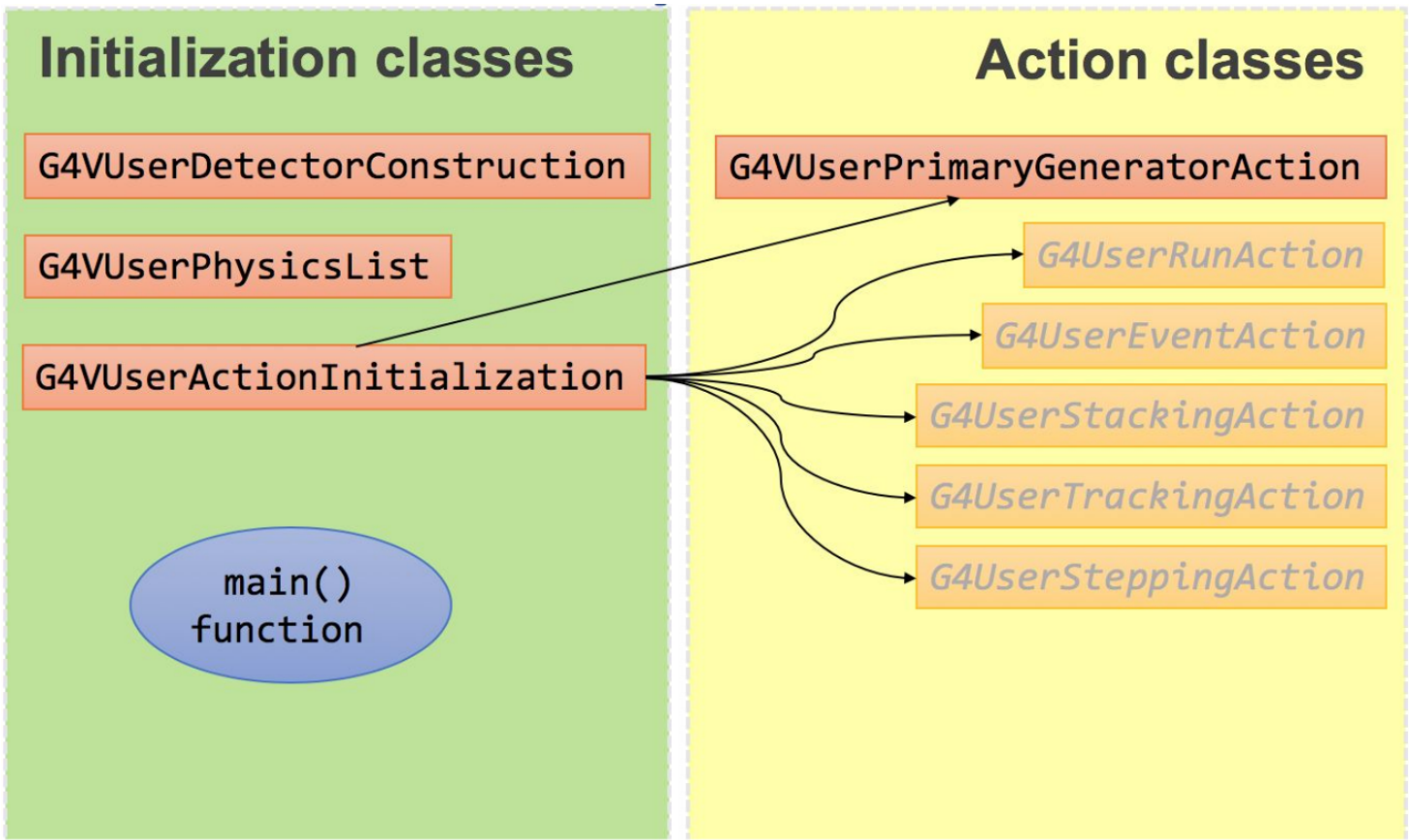
G4VUserDetectorConstruction.hh

```
// ...
class G4VUserDetectorConstruction
{
  public:
    G4VUserDetectorConstruction();
    virtual ~G4VUserDetectorConstruction();

  public:
    virtual G4VPhysicalVolume* Construct() = 0;
    virtual void ConstructSDandField();
  // ...
}
```

Typical source file



Classes



* Gray are optional.

Primary generator action

G4VUserPrimaryGeneratorAction.hh

```
// ...
class G4VUserPrimaryGeneratorAction
{
  public:
    G4VUserPrimaryGeneratorAction();
    virtual ~G4VUserPrimaryGeneratorAction();
  public:
    virtual void GeneratePrimaries(G4Event* anEvent) = 0;
    // ...
}
```

Defines the source of simulated particles.

- particle type.
- kinematic properties.
- additional information.

Physics list

G4VUserPhysicsList.hh

```
// ...
class G4VUserPhysicsList
{
  public:
    G4VUserPhysicsList();
    virtual ~G4VUserPhysicsList();
  public:
    virtual void ConstructParticle() = 0;
    virtual void ConstructProcess() = 0;
    virtual void SetCuts();
    // ...
}
```

- Define all necessary particles.
- Define all necessary processes and assign them to proper particles.
- Define particles production threshold.

Detector construction

G4VUserDetectorConstruction.hh

```
// ...
class G4VUserDetectorConstruction
{
  public:
    G4VUserDetectorConstruction();
    virtual ~G4VUserDetectorConstruction();

  public:
    virtual G4VPhysicalVolume* Construct() = 0;
    virtual void ConstructSDandField();
    // ...
}
```

- Define the geometry of your model.
  - All materials
  - All volumes & placements
- Optionally define fields.
- Optionally define sensitive detectors.

User interaction

Communicate with your application at three levels:

- Hard-Coded: application with no interaction.
- Batch mode: controlled by a macro.
- Interactive mode: with real-time user response.

User interaction

G4VUserActionInitialization.hh

```
// ...
class G4VUserActionInitialization
{
  public:
    G4VUserActionInitialization ();
    virtual ~G4VUserActionInitialization ();
  public:
    virtual void Build() const = 0;
    virtual void BuildForMaster() const;
    // ...
}
```

Optional actions as hooks:

- G4UserRunAction.
- G4UserEventAction.
- G4UserStackingAction.
- G4UserTrackingAction.
- G4UserSteppingAction.

Visualization

- View and debug your geometry.
- View and study the tracks.
- Product publication-ready graphics.
- Export events and geometry to text files.

main()

Geant4 does not provide a main function for you. Remember this is just a toolkit. All C++ programs need one, hence we define one.

- Create a source .cc file in the root directory of the application.
- Define a main function.
- Inside the main function:
  - Initialize the run manager.
  - Initialize all your initialization classes.
  - Initialize user interface and/or visualization.

main()

myApplication.hh

```
#include <G4RunManager.hh>
#include <G4UIExecutive.hh>
#include <G4VisExecutive.hh>

#include "MyDetectorConstruction.hh"
#include "MyPhysicsList.hh"
#include "MyActionInitialization.hh"

int main(int argc, char** argv) {
    G4RunManager* runManager = new G4RunManager;
    runManager->SetUserInitialization(new MyDetectorConstruction);
    runManager->SetUserInitialization(new MyPhysicsList);
    runManager->SetUserInitialization(new MyActionInitialization);
    G4VisManager* visManager = new G4VisExecutive;
    visManager->Initialize();
    G4UIExecutive* ui = new G4UIExecutive(argc, argv);
    ui->SessionStart();
    delete ui;
    delete visManager;
    delete runManager;
}
```

Include Geant4 class headers

Include your class headers

Prepare Geant4 kernel

Set up your initialization classes

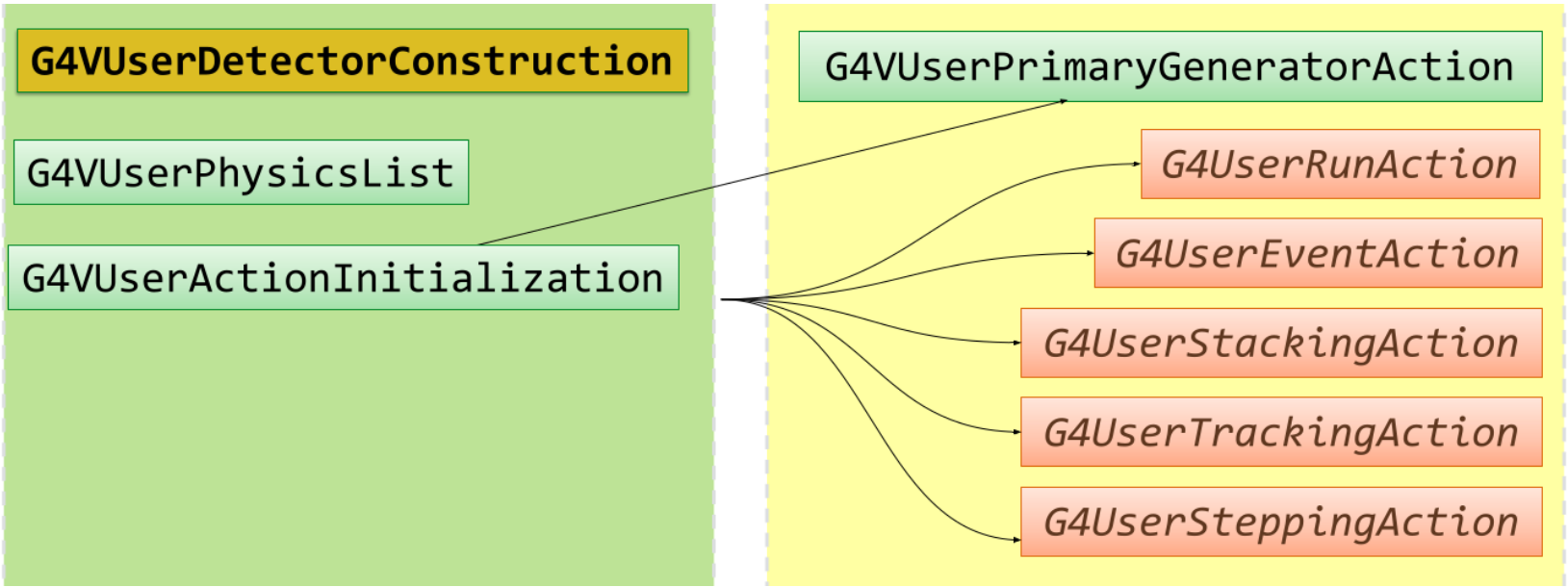
Enable visualization

Enable user interface

Interact with user

Final clean-up

User classes

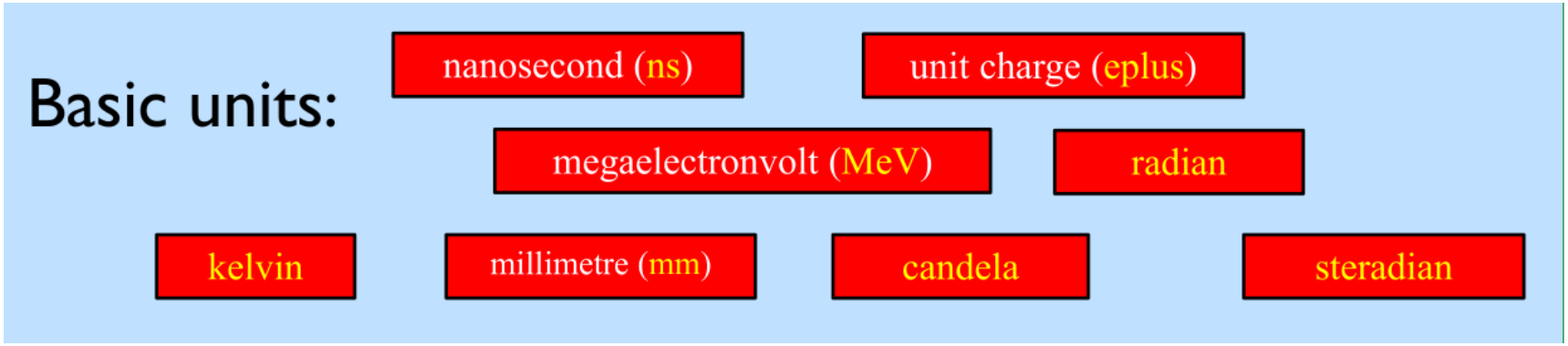


Units

- Always specify the units (mm, cm, m, etc..). These units can be found in the CLHEP library (included in Geant4).
- You can also define you own units, however this will not be shown here.
- In order to output data in terms of a specific unit. You must divide the value by the unit you're interested in using e.g: G4cout « dE / MeV « " (MeV)" « G4endl

System of Units

All units defined in Geant4 come from the basic ones.



Materials

Different levels of material description:

- Isotopes: G4Isotope
- Elements: G4Element
- Molecules, compounds and mixtures: G4Material

Attributes associated: temperature, pressure, state, density.
G4Isotope and G4Element: describes properties of atoms.
G4Material: describes the macroscopic properties of matter.

Creating Elements

Isotopes can be assembled into elements.

```
G4Isotope (const G4String& name,
           G4int     z,    // atomic number
           G4int     n,    // number of nucleons
           G4double  a );  // mass of mole
```

Not number of neutrons!

Do not forget unit (g/mole)

Elements:

```
G4Element (const G4String& name,
           const G4String& symbol, // element symbol
                 G4int     nIso ); // n. of isotopes
G4Element::AddIsotope(G4Isotope* iso,  // isotope
                      G4double relAbund); // fraction of atom per volume
```

Creating Elements

Elements can also be specified with a natural isotopic abundance.

```
G4Element (const G4String& name,
           const G4String& symbol,
           G4int      z,    // atomic number
           G4double   a );  // mass of mole
```

Do not forget unit (g/mole)

Single-element material

```
G4double z, a, density;
density = 1.390*g/cm3;
a = 39.95*g/mole;
G4Material* lAr = new G4Material("liquidAr", z=18, a, density);
```

Molecule material (composition by number of atoms)

```
a = 1.01*g/mole;
G4Element* elH = new G4Element("Hydrogen", symbol="H", z=1., a);

a = 16.00*g/mole;
G4Element* elO = new G4Element("Oxygen", symbol="O", z=8., a);

density = 1.000*g/cm3;
G4Material* H2O = new G4Material("Water", density, ncomponents=2);
H2O->AddElement(elH, natoms=2);
H2O->AddElement(elO, natoms=1);
```

By fraction of mass

```
a = 14.01*g/mole;
G4Element* elN  = new G4Element(name="Nitrogen",symbol="N", z= 7., a);
a = 16.00*g/mole;
G4Element* elO  =  new G4Element(name="Oxygen",symbol="O", z= 8., a);
density = 1.290*mg/cm3;
G4Material* Air = new G4Material(name="Air", density, ncomponents=2);
Air->AddElement(elN, 70.0*perCent);
Air->AddElement(elO, 30.0*perCent);
```

Mixing various elements and materials

```
G4Element* elC  = …;    // define "carbon" element
G4Material* SiO2 = …;   // define "quartz" material
G4Material* H2O = …;    // define "water" material
density = 0.200*g/cm3;

G4Material* aerogel = new G4Material("Aerogel",
                                     density, ncomponents=3);
aerogel->AddMaterial(SiO2,fractionmass=62.5*perCent);
aerogel->AddMaterial(H2O, fractionmass=37.4*perCent);
aerogel->AddElement (elC, fractionmass= 0.1*perCent);
```

Example: Gas

It's necessary to specify the temperature and pressure since this affects the energy variation along a distance this is known as thermal scattering.

```
G4double density = 27. * mg/cm3;
G4double temperature = 325. * kelvin;
G4double pressure = 50. * atmosphere;

G4Material* CO2 = new G4Material("CO2Gas", density,
    ncomponents=2, kStateGas, temperature, pressure);
CO2->AddElement(C, natoms = 1);
CO2->AddElement(O, natoms = 2);
```

Absolute vacuum doesn't exist (gas at very low density). Geant4 disallows creating materials with density equal to 0

```
G4double rho = 1.e-25*g/cm3;
G4double pr = 3.e-18*pascal;
G4Material* Vacuum = new G4Material("interGalactic",Z, A,   rho,
kStateGas, temperature, pr);
```

NIST material database

- It's easy to retrieve predefined elements or materials:

```
G4NistManager* manager = G4NistManager::Instance();
G4Material* H2O =    manager->FindOrBuildMaterial("G4_WATER");
G4Material* air =    manager->FindOrBuildMaterial("G4_AIR");
G4Material* vacuum = manager->FindOrBuildMaterial("G4_Galactic");
G4Element* Si = manager->FindOrBuildElement("Si");
```

- UI commands:

`/material/nist/printElement`	← print defined elements
`/material/nist/listMaterials`	← print defined materials

NIST material database

- NIST database for materials is imported inside Geant4
- UI commands specific for handling materials
- Precise values for the most relevant parameters:
  - Density
  - Mean excitation potential
  - Chemical bonds
  - Element composition
  - Isotope composition
  - Various corrections

Z		A	m	error	(%)	A_{eff}
14	Si	22	22.03453	(22)		28.0855(3)
		23	23.02552	(21)		
		24	24.011546	(21)		
		25	25.004107	(11)		
		26	25.992330	(3)		
		27	26.98670476	(17)		
		28	27.9769265327	(20)	92.2297 (7)	
		29	28.97649472	(3)	4.6832 (5)	
		30	29.97377022	(5)	3.0872 (5)	
		31	30.97536327	(7)		
		32	31.9741481	(23)		
		33	32.978001	(17)		
		34	33.978576	(15)		
		35	34.984580	(40)		
		36	35.98669	(11)		
		37	36.99300	(13)		
		38	37.99598	(29)		
		39	39.00230	(43)		
		40	40.00580	(54)		
		41	41.01270	(64)		
		42	42.01610	(75)		

- Natural isotope composition
- More than 3000 isotope masses

NIST material database

```
=====================================
###  Elementary Materials from the NIST Data
=====================================
 Z Name  ChFormula       density(g/cm^3)  I(eV)
=====================================
1   G4_H   H_2            8.3748e-05      19.2
2   G4_He                 0.000166322     41.8
3   G4_Li                 0.534           40
4   G4_Be                 1.848           63.7
5   G4_B                  2.37            76
6   G4_C                  2               81
7   G4_N   N_2            0.0011652       82
8   G4_O   O_2            0.00133151      95
9   G4_F                  0.00158029      115
10  G4_Ne                 0.000838505     137
11  G4_Na                 0.971           149
```

```
=====================================
###  Compound Materials from the NIST Data Base
=====================================
 N Name     ChFormula       density(g/cm^3)  I(eV)
=====================================
13  G4_Adipose_Tissue              0.92        63.2
          1    0.119477
          6    0.63724
          7    0.00797
          8    0.232333
         11    0.0005
         12    2e-05
         15    0.00016
         16    0.00073
         17    0.00119
         19    0.00032
         20    2e-05
         26    2e-05
         30    2e-05
4  G4_Air                         0.00120479  85.7
          6    0.000124
          7    0.755268
          8    0.231781
         18    0.012827
2  G4_CsI                         4.51        553.1
         53    0.47692
         55    0.52308
```

Describing our detector

- A detector geometry is made of many volumes.
- The largest volume is called World volume; this volume contains all other volumes.
- You must derive your detector construction from the G4VUserDetectorConstruction abstract class.
- Implementing the pure virtual method Construct()
  - Define shapes/solids required to describe the geometry
  - Construct all necessary materials
  - Construct and place volumes of you detector geometry
  - (Optional) Define "sensitivity" properties associated to volumes
  - (Optional) Associate magnetic field to detector regions
  - (Optional) Define visualization attributes for the detector elements

- Implement a class inheriting from the abstract base class G4VUserDetectorConstruction.

```
class MyDetector : public G4VUserDetectorConstruction {
public:
    virtual G4VPhysicalVolume* Construct();      // required

    virtual void ConstructSDAndField();          // optional
    // ...
};
```

- Create an instance of your implementation in the main program and add it to the run manager.

```
MyDetector* detector = new MyDetector();
runManager->SetUserInitialization(detector);
```

- Don't try to build all your geometry in just one class. This is OK if your geometry is simple, but if it's complex consider defining it in many classes.
- Don't delete your detector instance because the run manager does this automatically.

G4VUserDetectorConstruction

Construct()

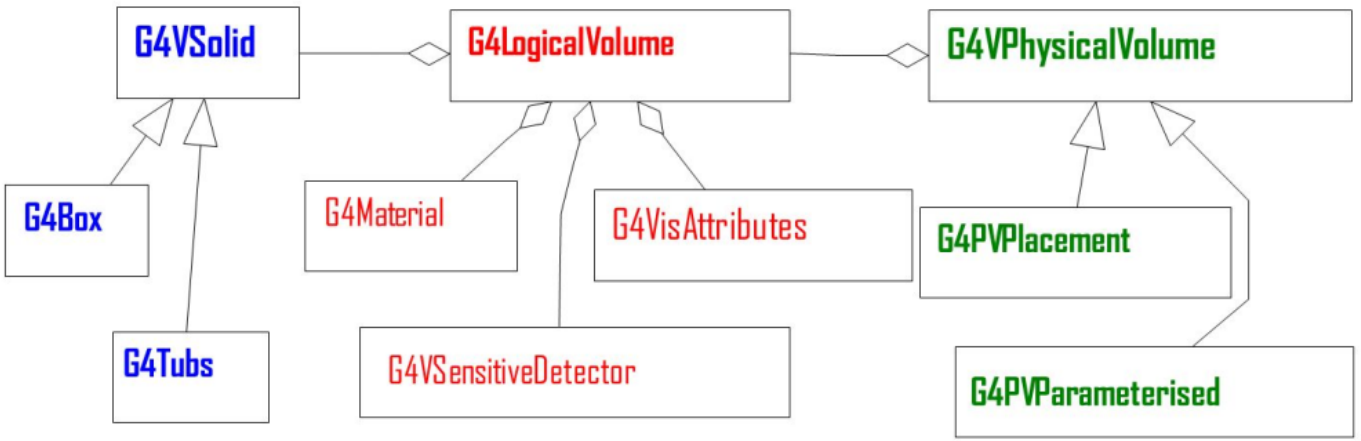
- Define materials
- Define solids and volumes of the geometry
- Build the tree hierarchy of volumes
- Define visualization attributes
- (The most important) MUST return the world physical volume

ConstructSDAndField()

- Assign magnetic field to volumes / regions
- Define sensitive detectors and assign them to volumes

Three conceptual layers

- G4VSolid: Shape and size
- G4LogicalVolume: Hierarchy of volumes, material, sensitivity and magnetic field
- G4VPhysicalVolume: Position, rotation. The same logical volume can be placed many times (Repeated modules)

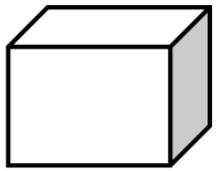


Detector geometry

- Basic strategy

```
G4VSolid* pBoxSolid =

  new G4Box("aBoxSolid",

        1.*m, 2.*m, 3.*m);
```

Solid: shape and size.



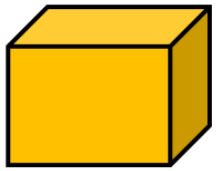
Step 1
Create the geom. object: box

Detector geometry

- Basic strategy

```
G4VSolid* pBoxSolid =
  new G4Box("aBoxSolid",
        1.*m, 2.*m, 3.*m);
G4LogicalVolume* pBoxLog =
   new G4LogicalVolume( pBoxSolid,
   pBoxMaterial, "aBoxLog", 0, 0, 0);
```

Logical volume : + material, sensitivity, etc.

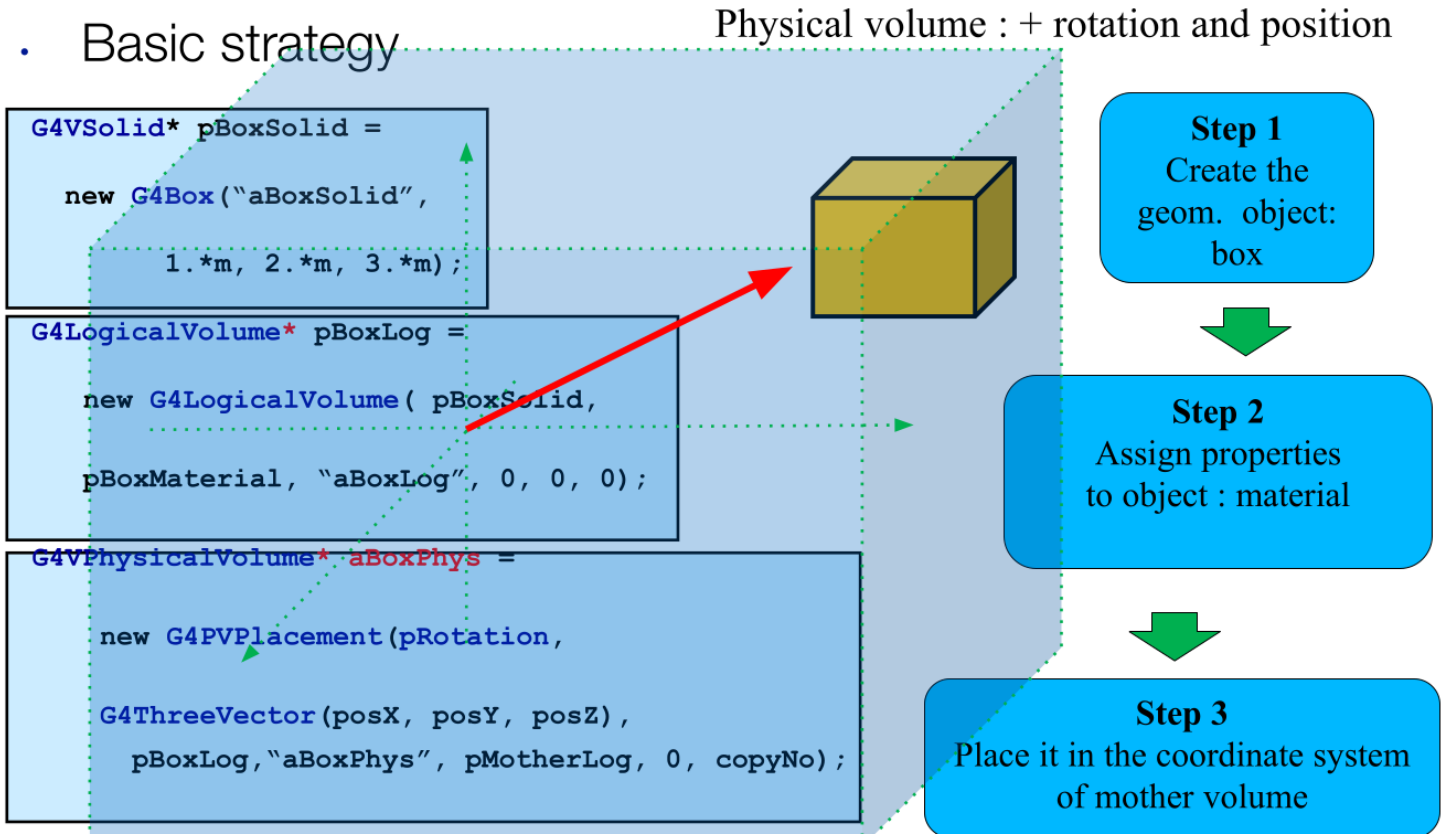


Step 1
Create the geom. object: box

Step 2
Assign properties to object : material

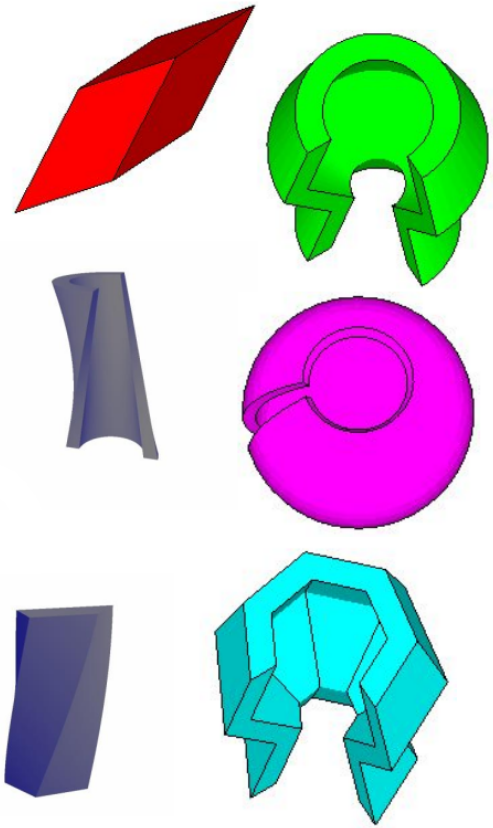
Detector geometry

- Basic strategy



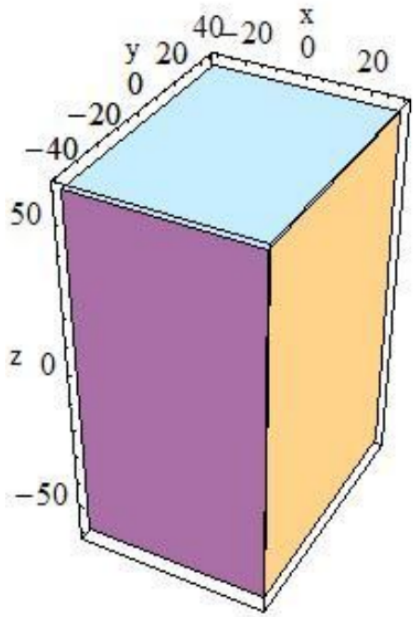
Solids

- CSG (Constructed Solid Geometry) solids: G4Box, G4Tubs, G4Cons, G4Trd, ...
- Specific solids (CSG like): G4Polycone, G4Polyhedra, G4Hype, ...
- BREP (Boundary REPresented) solids: G4BREPSolidPolycone, G4BSplineSurface, ...
- Boolean solids: G4UnionSolid, G4SubstractionSolid, ...



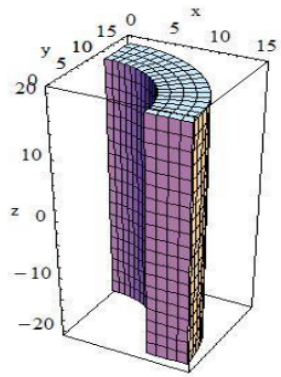
```
G4Box(const G4String& pname,  // name
            G4double  pX, // half-length in X
            G4double  pY, // half-length in Y
            G4double  pZ, // half-length in Z);
```

Note the half-length!

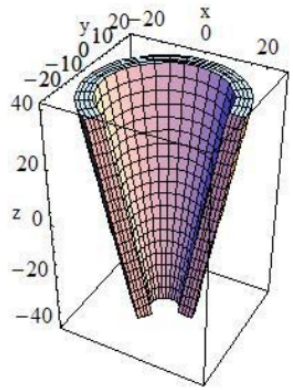


G4Tubs & G4Cons

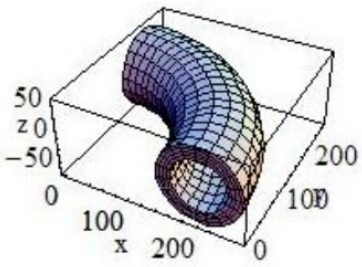
```
G4Tubs(const G4String& pname,  // name
             G4double  pRmin,  // inner radius (0)
             G4double  pRmax,  // outer radius
             G4double  pDz,    // Z half! length
             G4double  pSphi,  // starting Phi (0)
             G4double  pDphi); // segment angle (twopi)
```



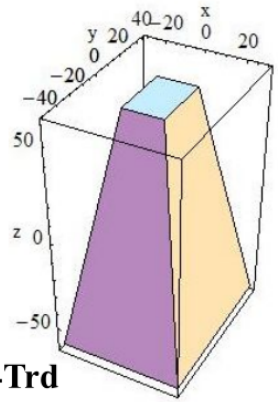

```
G4Cons(const G4String& pname,  // name
             G4double  pRmin1, // inner radius -pDz
             G4double  pRmax1, // outer radius -pDz
             G4double  pRmin2, // inner radius +pDz
             G4double  pRmax2, // outer radius +pDz
             G4double  pDz,    // Z half length
             G4double  pSphi,  // starting Phi
             G4double  pDphi); // segment angle
```



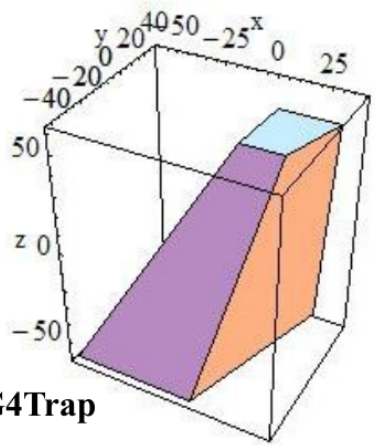
More CSG Solids



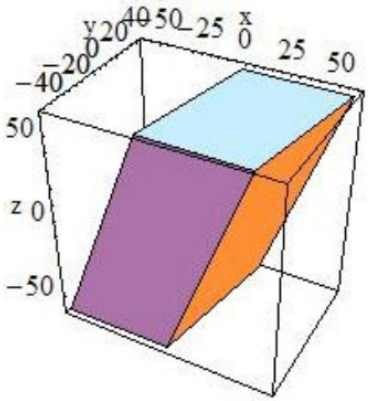
G4Torus



G4Trd

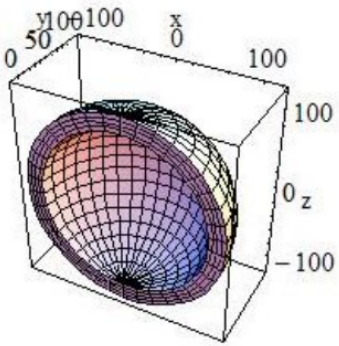


G4Trap

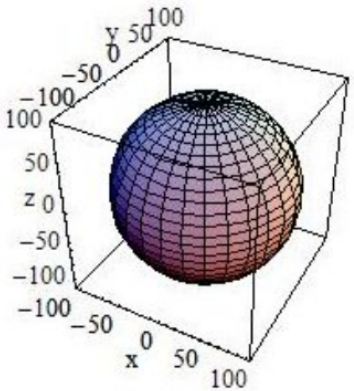


G4Para
(parallelepiped)

G4Sphere



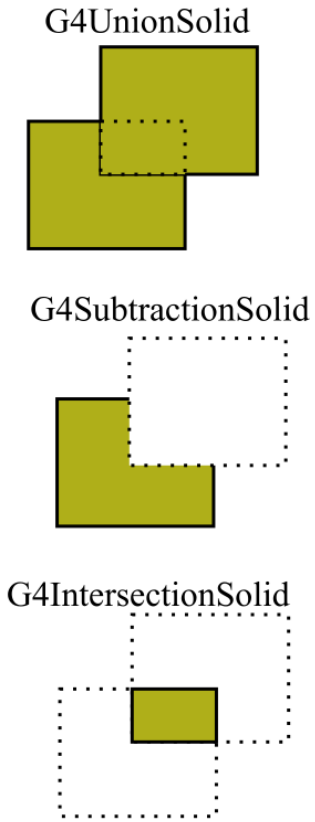
G4Orb
(full solid sphere)



Check Section 4.1.2 of Geant4 Application Developers Guide for all available shapes

Boolean Solids

- Solids can be combined using boolean operations.
  - G4UnionSolid, G4SubtractionSolid, G4IntersectionSolid
  - Requires: 2 solids, 1 boolean operation and an (optional) transformation for the 2nd solid.
  - 2nd solid is positioned relative to the coordinate system of the 1st solid.
  - Result of boolean operations becomes a solid, this solid is reusable with another boolean operation.
- Solids to be combined can be either CSG or other Boolean solids.



Boolean Solids - Example

```
G4VSolid* box = new G4Box("Box",50*cm,60*cm,40*cm);
G4VSolid* cylinder =
    new G4Tubs("Cylinder",0.,50.*cm,50.*cm,0.,twopi);

G4VSolid* union =
    new G4UnionSolid("Box+Cylinder", box, cylinder);

G4VSolid* subtract =
    new G4SubtractionSolid("Box-Cylinder", box, cylinder,
        0, G4ThreeVector(30.*cm,0.,0.));

G4RotationMatrix* rm = new G4RotationMatrix();
rm->RotateX(30.*deg);
G4VSolid* intersect =
    new G4IntersectionSolid("Box&&Cylinder",
        box, cylinder, rm, G4ThreeVector(0.,0.,0.));
```

Logical Volumes

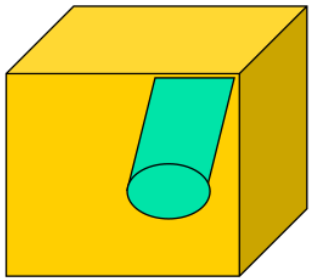
- Contains all information of volume except position:
  - Shape and dimension (G4VSolid)
  - Material, sensitivity, visualization attributes
  - Position of daughter volumes
  - Magnetic field, User limits
- Physical volumes of same type can share a logical volume.

```
G4LogicalVolume(G4VSolid* pSolid,
                G4Material* pMaterial,
                const G4String& name,
                G4FieldManager* pFieldMgr=0,
                G4VSensitiveDetector* pSDetector=0,
                G4UserLimits* pULimits=0,
                G4bool optimise=true);
```

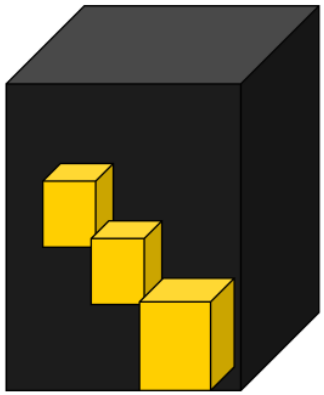
optional

Physical Volumes

- A physical volume is a positioned instance of a logical volume inside another logical volume (the containing logical volume is called the mother volume)
- Placement (G4PVPlacement)
- Repeated: a volume placed many times
  - can represent many volumes
  - reduces memory
  - G4PVReplica (= simple repetition)
  - G4PVParameterised (= more complex pattern)
  - G4PVDivision



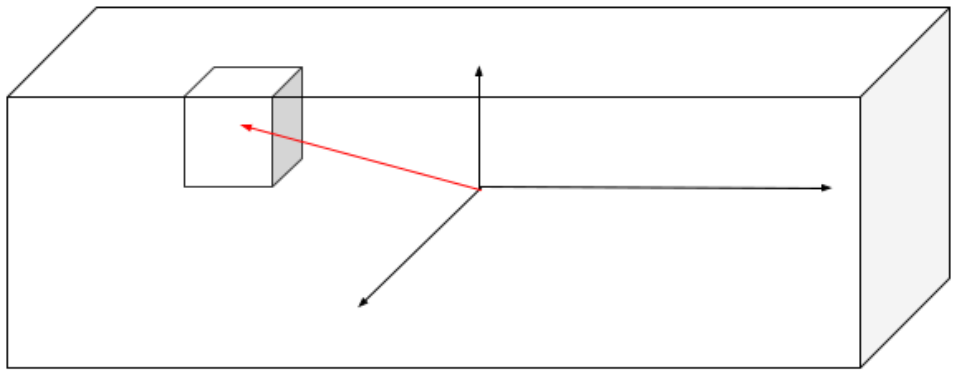
placement



repeated

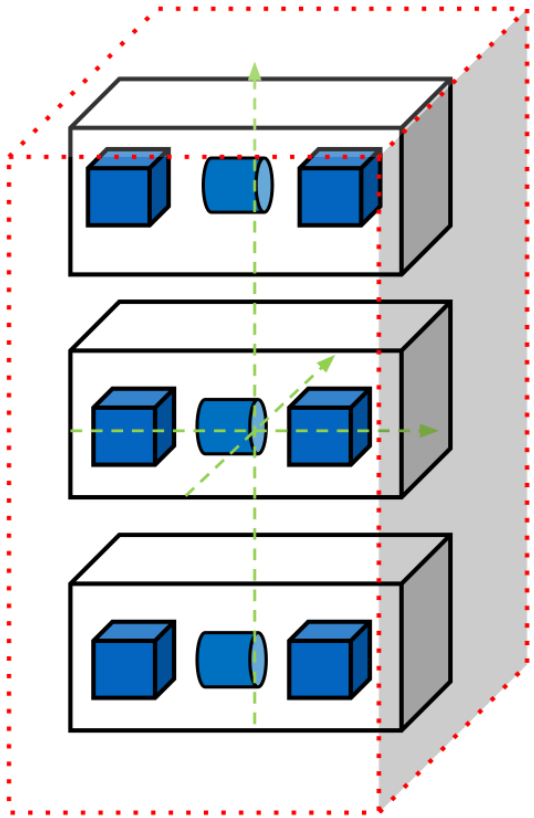
Geometry Hierarchy

- A volume is placed in its mother volume
  - Position and rotation of the daughter volume is described with respect to the local coordinate system of the mother volume
  - The origin of the mother's local coordinate system is at the center of the mother volume
  - Daughter volumes cannot protrude from the mother volume
  - Daughter volumes cannot overlap
- The logical volume of mother knows the daughter volumes is contains
  - It is uniquely defined to be their mother volume.



Geometry Hierarchy

- One logical volume can be placed more than once
- The mother daughter relationship is contained in the G4LogicalVolume
- The world volume must be a unique physical volume which fully contains all other volumes (root volume of the hierarchy).
  - The world volume defines the global coordinate system. The origin of the global coordinate system is at the center of the world volume.
  - Position of a track is given with respect to the global coordinate system

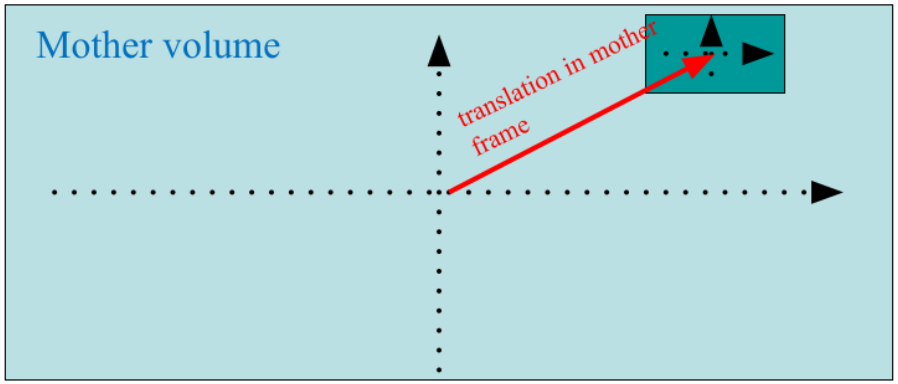


G4PVPlacement

- Single volume positioned relatively to the mother volume
  - In a frame rotated and translated relative to the coordinate system of the mother volume
- A few variants
  - Using G4Transform3D to represent the direct rotation and translation of the solid instead of the frame.
  - Specifying the mother volume as a pointer to its physical volume instead of its logical volume.

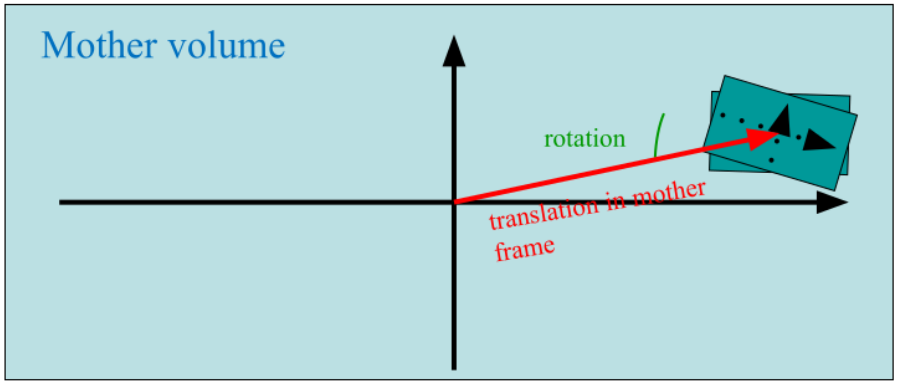
G4PVPlacement: Rotation of the mother frame

```
G4PVPlacement(G4RotationMatrix* pRot,        // rotation of mother frame
              const G4ThreeVector& tlate,    // position in mother frame
              G4LogicalVolume* pCurrentLogical,
              const G4String& pName,
              G4LogicalVolume* pMotherLogical,
              G4bool pMany,              // not used. Set it to false…
              G4int pCopyNo,             // unique arbitrary index
              G4bool pSurfChk=false ); // optional overlap check
```



G4PVPlacement: Rotation in the mother frame

```
G4PVPlacement(G4Transform3D(
            G4RotationMatrix &pRot,        // rotation in daughter frame
                 const G4ThreeVector &tlate),   // position in mother frame
              G4LogicalVolume *pDaughterLogical,
              const G4String &pName,
              G4LogicalVolume *pMotherLogical,
              G4bool pMany,                          // not used, set it to false…
              G4int pCopyNo,                         // unique arbitrary integer
              G4bool pSurfChk=false );               // optional overlap check
```

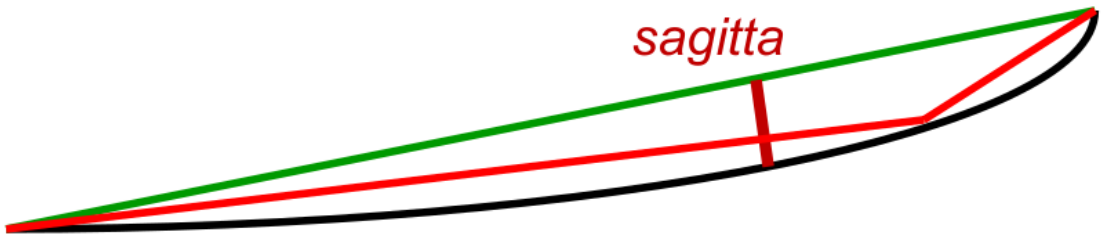


Tracking EM Fields

- Divide the trajectory of the particle in "steps"
  - Straight free-flight tracks between consecutive physics interactions
- In presence of EM fields, the free-flight part between interactions is not straight
  - Change of direction (B-field) or energy (E-field)
  - Effect of fields must be incorporated into the tracking algorithm which is very CPU-demanding
- Notice: most codes handle only weak fields
  - An e^- at rest will not accelerate, no syncrothron radiation, no avalanche

Tracking in fields

- In order to propagate a particle inside a field the equation of motion of the particle in the field is integrated numerically
- In general this is best done using a Runge-Kutta (RK) method for the integration of ordinary differential equations
- Once the curved path is calculated, Geant4 breaks it up into linear chord segments



- The chord segments are determined to closely approximate the curved path
  - In some cases, one step could be split in several helix-turns

Example: how to create a magnetic field: uniform

- Uniform field in the entire world volume: easy recipe

```
G4ThreeVector field(0,1.*tesla,0);
G4GlobalMagFieldMessenger* fMagFieldMessenger =
  new G4GlobalMagFieldMessenger(field)
```

- In general, one can customize the precision of the stepper and method used for the numerical integration of the equations

```
G4UniformMagField* magField = new G4UniformMagField(field);
G4FieldManager* fieldMgr =
G4TransportationManager::GetTransportationManager()
 ->GetFieldManager();
fieldMgr->SetDetectorField(magField);
fieldMgr->CreateChordFinder(magField);
```

register

Example: how to create a magnetic field: non-uniform

- Non-uniform field in the world volume
  - Create a class, derived from G4MagneticField implementing $f(\vec{x}, t)$

```
void MyField::GetFieldValue(const double
   Point[4], double *field) const
```

```
MyField* myField = new MyField();
G4FieldManager* fieldMgr =
G4TransportationManager::GetTransportationManager()
 ->GetFieldManager();
fieldMgr->SetDetectorField(myField);
fieldMgr->CreateChordFinder(myField);
```

Example: how to create a local magnetic field

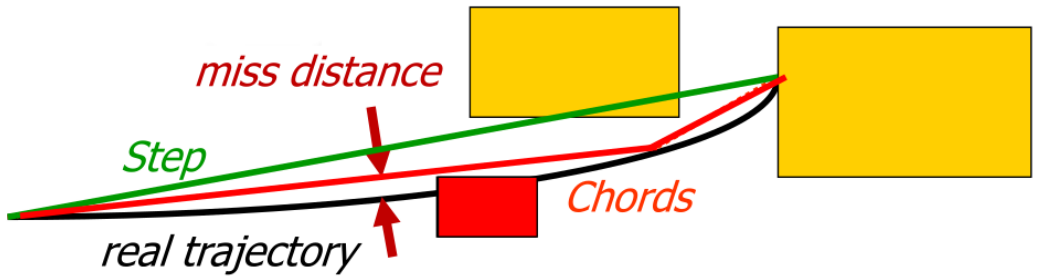
- It is possible to define a field inside a logical volume (and its daughters)
  - This can be done creating a local G4FieldManager and attaching it to a logical volume

```
MyField* myField = new MyField();
G4FieldManager* localFieldMgr =
            new G4FieldManager(myField);
G4bool allLocal = true;
logicVolWithField
 ->SetFieldManager(localFieldMgr, allLocal);
```

If true, field assigned to all daughters
If false, field assigned only to daughters w/o their own field manager

Customization

- A few parameters to customize the precision of the tracking in EM fields. Most critical: "miss distance"
  - Upper bound for the value of the sagitta (default: 3 mm)
  - May be highly CPU consuming



- Integration calculated by 4th-order Runge-Kutta (G4ClassicalRK4), robust and general purpose
  - If the field is not smooth (e.g. field map), lower-order (and faster) integrators can be appropriate
  - 3rd order G4SimpleHeum, 2nd order G4ImplicitEuler, 1st order G4ExplicitEuler

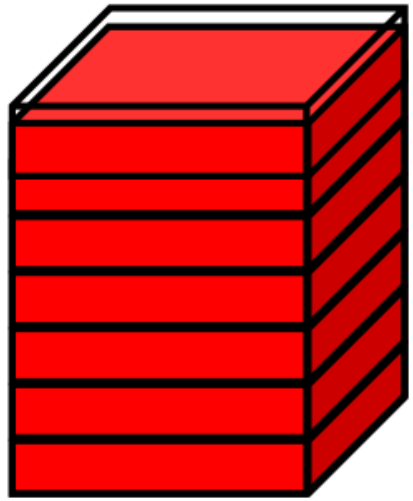
Physical volumes

- Placement volume (G4PVPlacement): one positioned volume
  - One physical volume represents one "real" volume
- Repeated volume: a volume placed many times
  - One physical volume represents any number of "real" volumes
- Parametrized (repetitions w.r.t. copy number)
- Replicas and Divisions
- Notice: a repeated volume is not equivalent to a loop of placements

Replicated volumes (G4PVReplica)

- The mother volume is completely filled with replicas, all having same size and shape
  - If you need gaps, use G4PVDivision instead (less CPU-efficient)
- Replication may occur along:
  - Cartesian axes (kXAxis, kYAxis, kZAxis)
  - Radial axis (cilindrical polar) (kRho) - onion rings
  - Phi axis (cylindrical polar) (kPhi) – cheese wedges

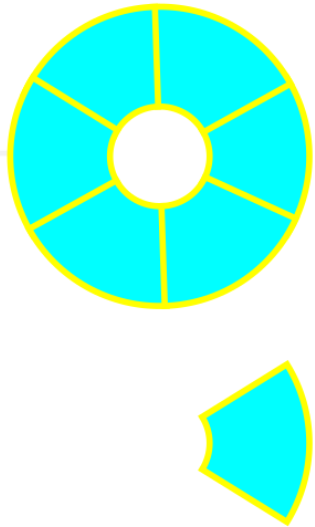
a daughter logical volume to be replicated



mother volume

G4PVReplica

```
G4PVReplica(const G4String &pName,
            G4LogicalVolume* pLogical,
            G4LogicalVolume* pMother,
            const EAxis pAxis,
            const G4int nReplicas,
            const G4double width,
            const G4double offset=0.);
```



Features and restrictions:

- CSG solids only
- G4PVReplica must be the only daughter
- Replicas may be placed inside other replicas
- Normal placement volumes may be placed inside replicas
- No volume can be placed inside a radial replication
- Parameterised volumes cannot be placed inside a replica

Replica: axes, width and offset

Center of n th daughter is given as:

- Cartesian axes - kXaxis, kYaxis, kZaxis

  `-width*(nReplicas-1)*0.5+n*width`

  Offset shall not be used

- Radial axis – kRho

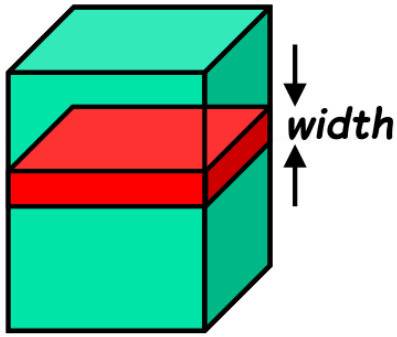
  `width*(n+0.5)+offset`

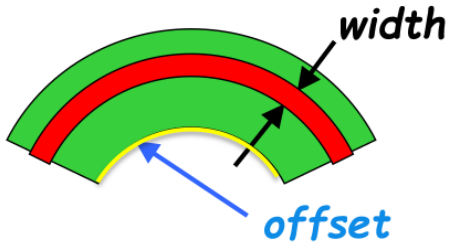
  Offset must be the inner radius of the mother

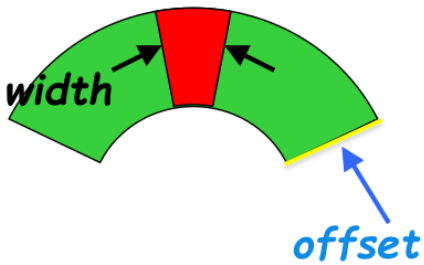
- Phi axis – kPhi

  `width*(n+0.5)+offset`

  Offset must be the starting angle of the mother

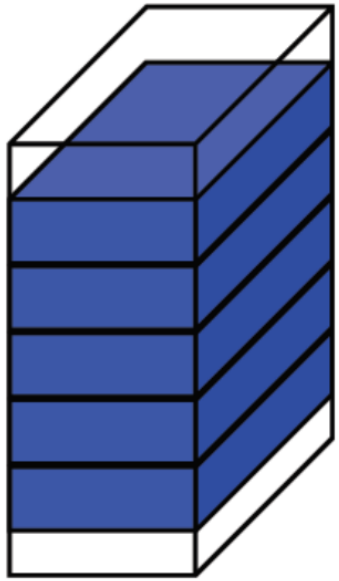






G4PVDivision

- The G4PVDivision is similar to the G4PVReplica but
  - Allows for gaps between mother and daughter volumes
  - Less CPU-effective than replica
- Shape of all daughter volumes must be the same as of the mother volume
- A number of shapes / axes patterns are supported, e.g.
  - G4Box : kXAxis, kYAxis, kZAxis
  - G4Tubs : kRho, kPhi, kZAxis
  - G4Cons : kRho, kPhi, kZAxis
  - ...



Parametrized volumes (G4VPVParameterisation)

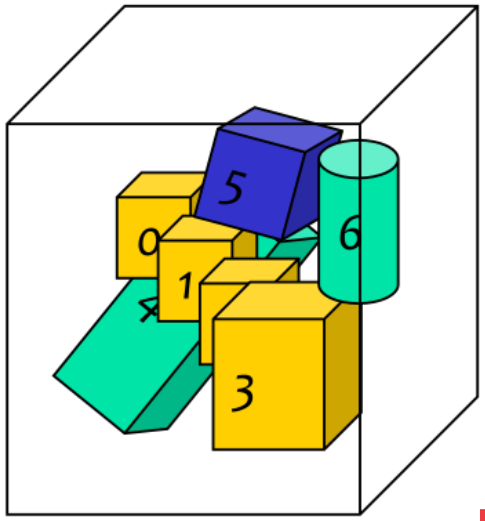
Repeated volumes can differ by size, shape, material and transformation matrix, that can all be parameterised by the user as a function of the copy number

User is asked to derive her/his own parameterisation class from the G4VPVParameterisation class implementing the methods:

```
void ComputeTransformation(const G4int copyNo,
                           G4VPhysicalVolume
                           *physVol) const;
void ComputeDimensions(G4Tubs& trackerLayer,
       const G4int copyNo, const
       G4VPhysicalVolume *physVol)  const;
```

Optional methods:

```
       ComputeMaterial(...)
       ComputeSolid(...)
```



Parametrized volumes

- All daughters must be fully contained in the mother
  - Daughters should not overlap to each other
- Limitations:
  - Applies to simple CSG solids only
  - Grand-daughter volumes allowed only for special cases
- Typical use-cases:
  - Complex detectors with large repetition of volumes, regular or irregular
  - Medical applications: the material in tissue is modeled as parametrixed voxels with variable density
  - Limited memory footprint

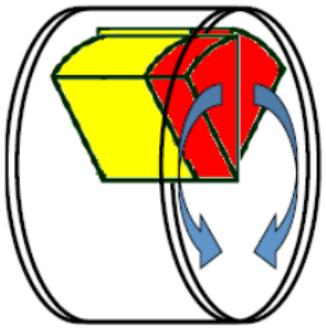
G4PVParametrized

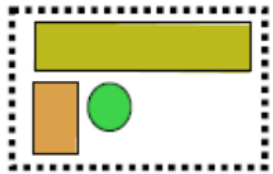
```
G4PVParameterised(const G4String& pName,
                  G4LogicalVolume* pLogical,
                  G4LogicalVolume* pMother,
                  const EAxis pAxis,
                  const G4int nReplicas,
                  G4VPVParameterisation *pParam
                  G4bool pSurfChk=false);
```

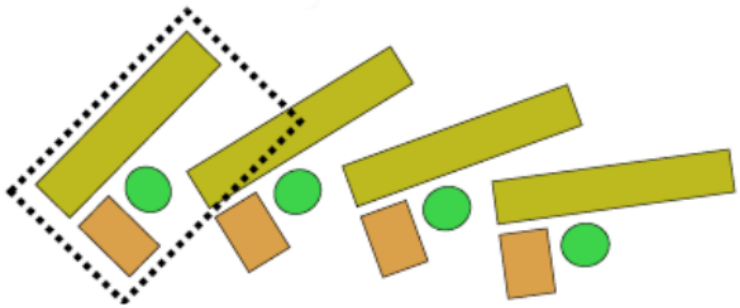
- Replicates the volume nReplicas times using the parameterization pParam, within the mother volume pMother
- pAxis specifies the tracking optimisation algorithm to apply:
  - kXAxis, kYAxis, kzAxis - 1D voxelisation algorithm
  - kUndefined - 3-D voxelisation algorithm
- Each replicated volume is a touchable detector element

Assembly & reflections

- Possible to represent a regular pattern of positioned volumes, composing a more or less complex structure
  - structures which are hard to describe with simple replicas or parameterised volumes
- Assembly volume (G4AssemblyVolume)
  - acts as an envelope for its daughter volumes

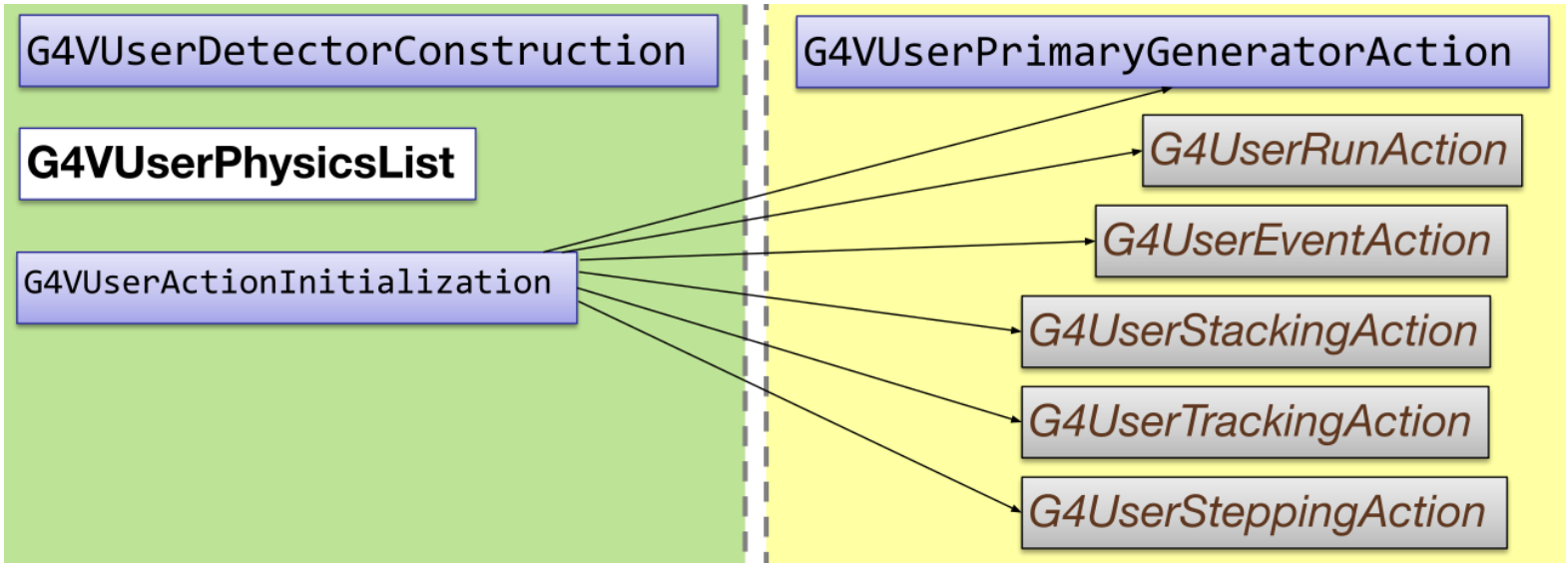






- G4ReflectedSolid (derived from G4VSolid)
  - Utility class representing a solid shifted from its original reference frame to a new mirror symmetric one

User classes

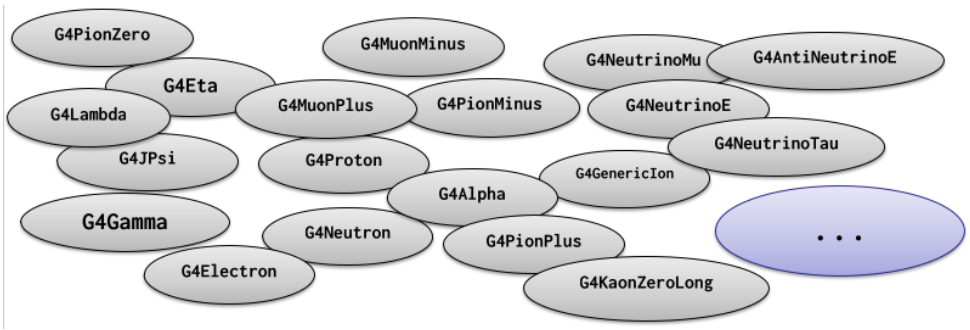


Physics

- Huge amount of different processes for various purposes (only a handful are relevant)
- Competing descriptions of the same physics phenomena (necessary to choose)
  - fundamentally different approaches
  - balance between speed and precision
  - different parameterizations
- Hypothetical processes and exotic physics

Definition of a Particle

Geant4 provides G4ParticleDefinition daughter class to represent a large number of elementary particles and nuclei, organized in six major categories: leptons, mesons, baryons, bosons, short-lived and ions.



User must define all particle types which might be used in the application: not only primary particles but also all other particles which may appear as secondaries generated by the used physics processes

Definition of a Particle

- Particle Data Group (PDG) particle
- Optical photons (different from gammas)
- Special particles: geantino and charged geantino
  - Only transported without interactions
  - Charged geantino also feels the EM field
- Short-lived particles ($\tau > 10^{-14} s$) are not transported by Geant4 (decay is applied)
- Light ions (as deuterons, tritons, alphas)
- Heavier ions are simply represented by G4Ions

Leptons and Bosons

Particle name	Class name	Name (in GPS...)	PDG
electron	G4Electron	e-	11
positron	G4Positron	e+	-11
muon +/-	G4MuonPlus G4MuonMinus	mu+ mu-	-13 13
tauon +/-	G4TauPlus G4TauMinus	tau+ tau-	-15 15
electron (anti)neutrino	G4NeutrinoE G4AntiNeutrinoE	nu_e anti_nu_e	12 -12
muon (anti)neutrino	G4NeutrinoMu G4AntiNeutrinoMu	nu_mu anti_nu_mu	14 -14
tau (anti)neutrino	G4NeutrinoTau G4AntiNeutrinoTau	nu_tau anti_nu_tau	16 -16
photon (γ, X)	G4Gamma	gamma	22
photon (optical)	G4OpticalPhoton	opticalphoton	(0)
geantino	G4Geantino	geantino	(0)
charged geantino	G4ChargedGeantino	chargedgeantino	(0)

Common Hadrons and Ions

Particle name	Class name	Name (in GPS...)	PDG
(anti)proton	G4Proton G4AnitProton	proton anti_proton	2212 -2212
(anti)neutron	G4Neutron G4AntiNeutron	neutron anti_neutron	2112 -2112
(anti)lambda	G4Lambda G4AntiLambda	lambda anti_lambda	3122 -3122
pion	G4PionMinus G4PionPlus G4PionZero	pi- pi+ pi0	-211 211 111
kaon	G4KaonMinus G4KaonPlus G4KaonZero G4KaonZeroLong G4KaonZeroShort	kaon- kaon+ kaon0 kaon0L kaon0S	-321 321 311 130 310
(anti)alpha	G4Alpha G4AntiAlpha	alpha anti_alpha	1000020040 -1000020040
(anti)deuteron	G4Deteuron G4AntiDeuteron	deuteron anti_deuteron	1000010020 -1000010020
Heavier ions	G4Ions	ion	100ZZZAAAI*

*ZZZ=proton number, AAA=nucleon number, I=excitation level

These define how particles interact with materials.
Responsibilities:

- decide when and where an interaction occurs
  - GetPhysicalInteractionLength...()
  - this requires a cross section
  - for the transportation processes, the distance to the nearest object
- generate the final state of the interaction
  - changes in momentum, generates secondaries, etc..
  - method: DoIt...()
  - this requires a model of the physics

G4VProcess class

- Abstract class as a base for all processes in Geant4
  - Used by all physics processes (also by the transportation, etc...
  - Defined in source/processes/management
- Define three kinds of actions:

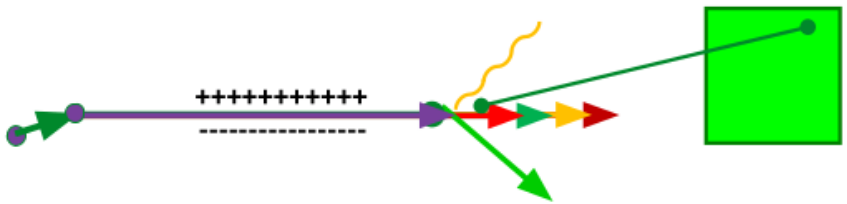
  - AtRest actions
    - Decay, e^+ annihilation ...
  - AlongStep actions
    - To describe continuous (inter)actions, occurring along the path of the particle, like ionization
  - PostStep actions
    - For describing point-like (inter)actions, like decay in flight, hadronic interactions ...

Example Processes

- Discrete process: Compton Scattering, hadronic inelastic, ...
  - step determined by cross section, interaction at end of step
- Continuous process: Čerenkov effect
  - photons created along step, roughly proportional to step length
- At rest process: muon capture at rest
  - interaction at rest
- Rest + discrete: positron annihilation, decay, ...
  - both in flight and at rest
- Continuous + discrete: ionization
  - energy loss is continuous
  - knock-on electrons (δ-ray) are discrete

Handling Multiple Processes

1. all particle is shot and "transported"
2. all processes associated to the particle propose a geometrical step length (depends on process cross-section)
3. The process proposing the shortest step "wins" and the particle is moved to destination (if shorter than "Safety")
4. All processes along the step are executed (e.g. ionization)
5. post step phase of the process that limited the step is executed. New tracks are "pushed" to the stack
6. If $E_{kin} = 0$ all at rest processes are executed; if particle is stable the track is killed. Else:
7. New step starts and sequence repeats...



Physics List

- One instance per application
  - registered in the run manager in main()
  - inheriting from G4VUserPhysicsList
- Responsibilities
  - all particle types (electron, proton, gamma, ...)
  - all processes (photoeffect, bremsstrahlung, ...)
  - all process parameters (...)
  - production cuts

3 ways to get a physics list

1. (Hard) Manual: Specify all particles and processes that may occur in the simulation.
2. (Easier) Physics Constructors: Combine your physics from pre-defined sets of particles and processes. Still you define your own class – modular physics list
3. (Easy) Reference physics lists: Take one of the pre-defined physics lists. You don't create any class

Method 1: G4VUserPhysicsList

Implement 3 methods:

```
class MyPhysicsList : public G4VUserPhysicsList {
public:
    // ...
    void ConstructParticle();  // pure virtual
    void ConstructProcess();   // pure virtual
    void SetCuts();
    // ...
}
```

Advantage: most flexible Disadvantages:

- most verbose
- most difficult to get right

G4VUserPhysicsList: Implementation

- ConstructParticle()
  - choose the particles you need in your simulation, define all of them here
- ConstructProcess()
  - for each particle, assign all the physics processes relevant to your simulation
- SetCuts()
  - set the range cuts for secondary production for processes with infrared divergence

ConstructParticle()

Due to the large number of particles can be necessary to instantiate, this method sometimes can be not so comfortable

```
void MyPhysicsList::ConstructParticle() {
    G4Electron::ElectronDefinition();
    G4Proton::ProtonDefinition();
    G4Neutron::NeutronDefinition();
    G4Gamma::GammaDefinition();
    ....
}
```

It is possible to define **all** the particles belonging to a **Geant4 category:**

- G4LeptonConstructor
- G4MesonContructor
- G4BaryonConstructor
- G4BosonConstructor
- G4ShortlivedConstructor
- G4IonConstructor

```
void MyPhysicsList::ConstructParticle() {
    // Construct all baryons
    G4BaryonConstructor bConstructor;
    bConstructor.ConstructParticle();
    // Construct all leptons
    G4LeptonConstructor lConstructor;
    lConstructor.ConstructParticle();
    // ...
}
```

ConstructProcess()

1. For each particle, get its process manager.

```
G4ProcessManager *elManager = G4Electron::ElectronDefinition()->GetProcessManager();
```

2. Construct all processes and register them.

```
elManager->AddProcess(new G4eMultipleScattering, -1, 1, 1);
elManager->AddProcess(new G4eIonisation, -1, 2, 2);
elManager->AddProcess(new G4eBremsstrahlung, -1, -1, 3);
elManager->AddDiscreteProcess(new G4StepLimiter);
```

```
AddTransportation();
```

SetCuts()

- Define all production cuts for gamma, electrons and positrons
  - Recently also for protons
- Notice: this is a production cut, not a tracking cut

Example

```
void StandardPhysics::ConstructParticle()
{
        // We are interested in gamma, electrons and possibly positrons
        G4Electron::ElectronDefinition();
        G4Positron::PositronDefinition();
        G4Gamma::GammaDefinition();
}

void StandardPhysics::ConstructProcess()
{
        // Transportation is necessary
        AddTransportation();

        // Electrons
        G4ProcessManager *elManager = G4Electron::ElectronDefinition()->GetProcessManager();
        elManager->AddProcess(new G4eMultipleScattering, -1, 1, 1);
        elManager->AddProcess(new G4eIonisation, -1, 2, 2);
        elManager->AddProcess(new G4eBremsstrahlung, -1, -1, 3);
        elManager->AddDiscreteProcess(new G4StepLimiter);

        // Positrons
        G4ProcessManager *posManager = G4Positron::PositronDefinition()->GetProcessManager();
        posManager->AddProcess(new G4eMultipleScattering, -1, 1, 1);
        posManager->AddProcess(new G4eIonisation, -1, 2, 2);
        posManager->AddProcess(new G4eBremsstrahlung, -1, -1, 3);
        posManager->AddProcess(new G4eplusAnnihilation, 0, -1, 4);
        posManager->AddDiscreteProcess(new G4StepLimiter);

        // Gamma
        G4ProcessManager *phManager = G4Gamma::GammaDefinition()->GetProcessManager();
        phManager->AddDiscreteProcess(new G4ComptonScattering);
        phManager->AddDiscreteProcess(new G4PhotoElectricEffect);
        phManager->AddDiscreteProcess(new G4GammaConversion);

        // TODO: Introduce Rayleigh scattering. It has large cross-section than Pair production
}

void StandardPhysics::SetCuts()
{
        // TODO: Create a messenger for this
        defaultCutValue = 0.03 * mm;
        SetCutsWithDefault();
}
```

Method 2: G4VModularPhysicsList

Similar structure as G4VUserPhysicsList:

```
class MyPhysicsList : public G4VModularPhysicsList {
public:
    MyPhysicsList();            // define physics constructors
    void ConstructParticle();   // optional
    void ConstructProcess();    // optional
    void SetCuts();             // optional
}
```

Differences to "manual":

- Particles and processes typically handled by physics constructors
- Transportation automatically included

Physics Constructors

A physics constructor is a module that is used to construct a modular physics list.

- Inherits from G4VPhysicsConstructor
- Defines ConstructorParticle() and ConstructProcess()
- GetPhysicsType()
  - enables switching physics of the same type, if possible.

Physics Constructors

- Huge set of pre-defined ones
  - EM: Standard, Livermore, Penelope
  - Hadronic inelastic: QGSP_BIC, FTFP_Bert, ...
  - Hadronic elastic: G4HadronElasticPhysics, ...
  - ... (decay, optical physics, EM extras, ...)
- You can implement your own by inheriting from the G4VPhysicsContructor class

Add physics constructor in the constructor:

```
MyModularList::MyModularList() {
    // Hadronic physics
    RegisterPhysics(new G4HadronElasticPhysics());
    RegisterPhysics(new G4HadronPhysicsFTFP_BERT_TRV());
    // EM physics
    RegisterPhysics(new G4EmStandardPhysics());
}
```

Method 3: Reference Physics Lists

Pre-defined physics lists

- already containing a complete set of particles and processes
- targeted at specific area of interest (HEP, medical physics, ...)
- constructed as modular physics list, built on top of physics constructors
- customizable (by calling appropriate methods before initialization)

List of Reference Physics Lists

Source code: $G4INSTALL/source/physics_lists/lists

FTF_BIC.hh	G4PhysListRegistry.hh	QGSP_BIC_AllHP.hh
FTFP_BERT.hh	G4PhysListStamper.hh	QGSP_BIC.hh
FTFP_BERT_HP.hh	INCLXXPhysicsListHelper.hh	QGSP_BIC_HP.hh
FTFP_BERT_TRV.hh	LBE.hh	QGSP_FTFP_BERT.hh
FTFP_INCLXX.hh	NuBeam.hh	QGSP_INCLXX.hh
FTFP_INCLXX_HP.hh	QBBC.hh	QGSP_INCLXX_HP.hh
G4GenericPhysicsList.hh	QGS_BIC.hh	Shielding.hh
G4PhysListFactoryAlt.hh	QGSP_BERT.hh	
G4PhysListFactory.hh	QGSP_BERT_HP.hh	

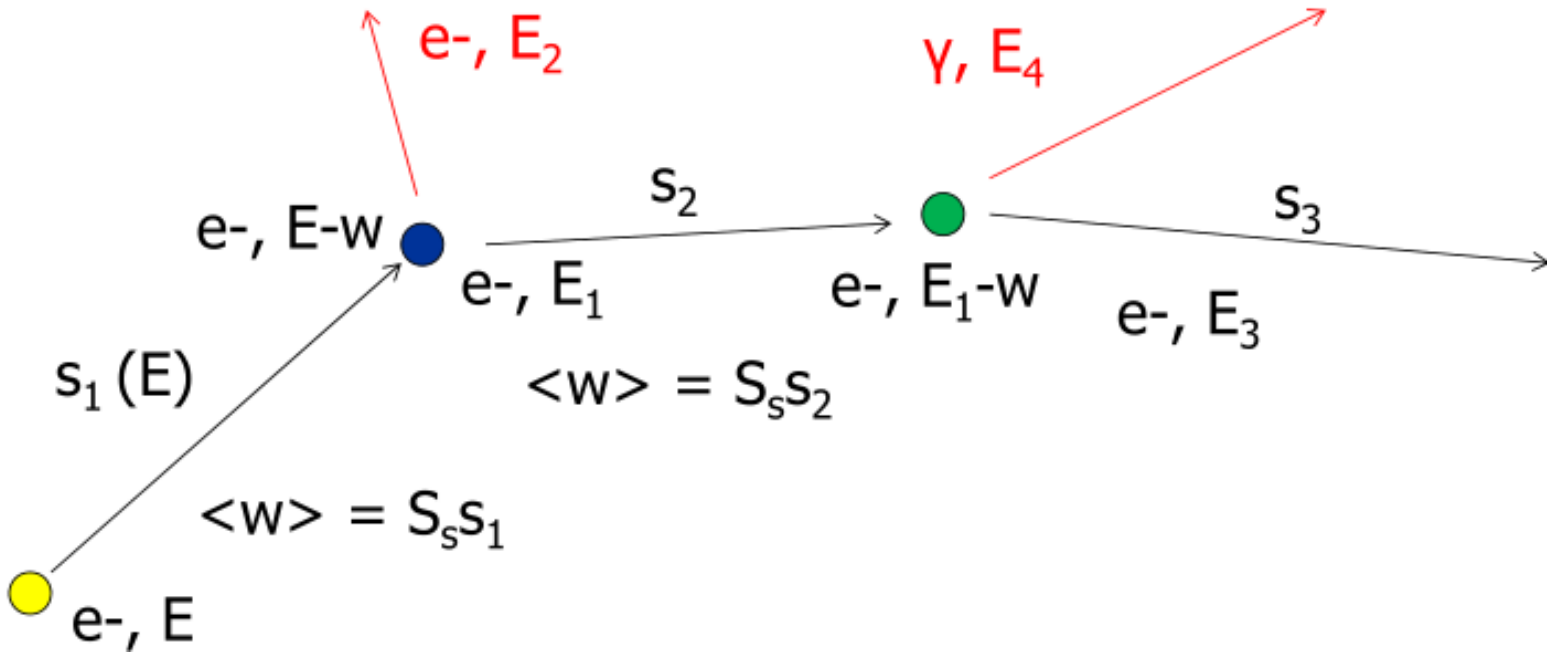
Mixed Monte Carlo (Reminder)

- Simulate explicitly (i.e. force step) interactions only if energy loss (or change of direction) is above threshold W_0
  - Detailed simulation
  - "hard" interactions (destructive or big changes in energy)
- The effect of all sub-threshold interactions is described statistically
  - Condensed simulation
  - "soft" interactions
- When W_0 is 0, Geant4 will treat all interactions as hard interactions (CPU intensive).

The G4VProcess (Reminder)

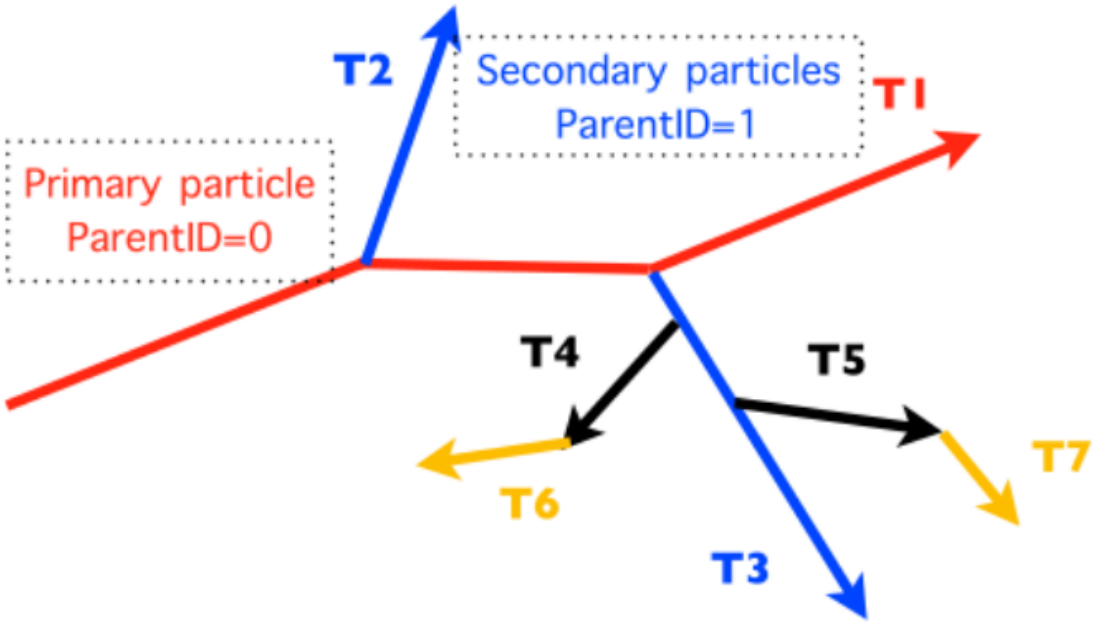
- Physics processes are derived from the G4VProcess base class
- Abstract class defining the common interface of all processes in Geant4, used by all physics processes
  - Detailed simulation
  - "hard" interactions (destructive or big changes in energy)
  - Three kinds of "actions":
    - AtRest actions: Decays, e^+ annihilation
    - AlongStep actions: To describe continuous interactions, occurring along the path of the particle, i.e. "soft" interactions
    - PostStep actions: To describe the point-like (inter)actions, like decay in flight, hadronic interactions, i.e. "hard" interactions

Particle tracking: mixed recipe (Reminder)



Geant4 Tracking

- Force step at geometry boundaries
- All AlongStep processes co-work, the PostStep compete (= only one selected)
- Call AtRest actions for particles at rest
- Secondaries saved at the top of the stack: tracking order follows 'last in first out' rule: T1 $\rightarrow$ T3 $\rightarrow$ T5 $\rightarrow$ T7 $\rightarrow$ T4 $\rightarrow$ T6 $\rightarrow$ T2



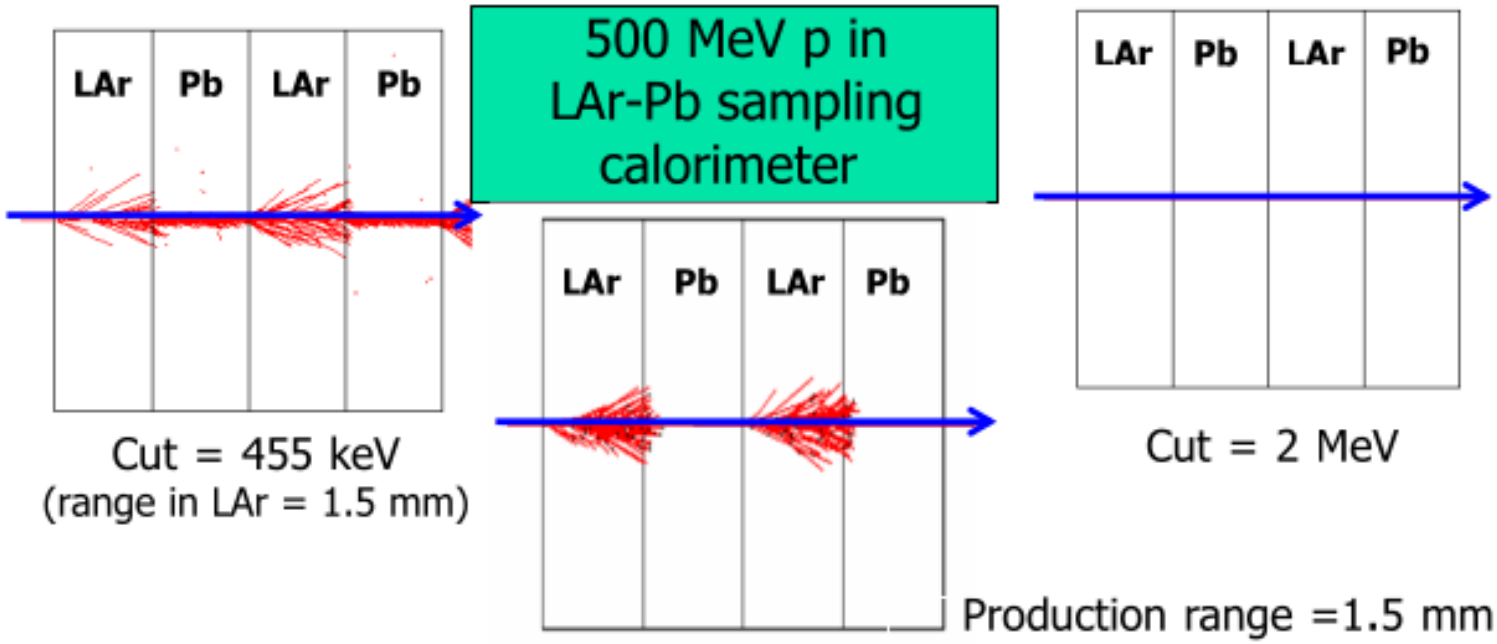
Geant4 Production Cuts

- The traditional Monte Carlo solution is to set a tracking cut-off in energy:
  - Particles are stopped when this energy is reached and the residual energy is deposited at that point
  - May yield cause imprecise stopping location and deposition of energy
- Geant4 does not have tracking cuts
  - All tracks are followed down to zero energy or until they leave the world volume or are destroyed in interactions
  - Could be implemented manually by the user
- Geant4 uses only a production cut (W_0)
  - Cuts deciding whether a secondary particle will be produced or not

Geant4 way of cuts: cut-in-range

- Geant4 solution: set a “range” production threshold
  - Particles unable to travel at least the range cut value are not produced
    - They contribute to the AlongStep
  - default $= 1$ mm
- One production threshold is uniformly set
  - Sets the ”spatial accuracy” of the simulation
- Production threshold is internally converted to the energy threshold W_0, depending on particle type and material

Cut-in-range



Threshold in range: 1.5 mm

455 keV electron energy in liquid Ar

2 MeV electron energy in Pb

SetCuts() Example

- Define all production cuts for gamma and electrons
  - Lowest W 0 is 990 eV (but can be changed)

In **G4VUserPhysicsList** class

```
void MyPhysicsList::SetCuts()
{
    //G4VUserPhysicsList::SetCuts();
    defaultCutValue = 0.5 * mm;
    SetCutsWithDefault();

    SetCutValue(0.1 * mm, "gamma");
    SetCutValue(0.01 * mm, "e+");
    G4ProductionCutsTable::GetProductionCutsTable()
        ->SetEnergyRange(100*eV, 100.*GeV);
}
```

Default value: 1.0 mm

Lower the possible W_0 from 990 eV **to 100 eV**

G4StepLimiter

- Alternative to define the level of tracking detail
- Why?
  - you want to see the exact track of the particle
  - you don‘t trust the chord finder for your magnetic field
- How?
  - Include G4StepLimiter process in your physics list
    - Formally seen as a physics process, competing with all others: always proposing the same step length
    - Can be done by using the Geant4 constructor G4StepLimiterPhysics in a modular physics list
  - Set “user limits” for the logical volumes of interest: SetUserLimits()

Cuts per region

- Complex detector may contain many different sub-detectors involving:
  - finely segmented volumes
  - position-sensitive materials (e.g. Si trackers)
  - large, undivided volumes (e.g. calorimeters)
  - inert materials
- The same cut may not be appropriate for all of these
- User can define regions (independent of geometry hierarchy tree) and assign different cuts for each region
  - A region can contain a subset of the logical volumes

Run, Event and Track

Event

- An Event is the basic unit of simulation
- At the beginning of event, primary tracks are generated and they are pushed into a stack
- Tracks are popped up from the stack one-by-one and 'tracked'
  - Secondary tracks are also pushed into the stack
  - When the stack gets empty, the processing of the event is completed
- G4Event class represents an event. At the end of a successful event it has:
  - List of primary vertices and particles (as input)
  - Hits and Trajectory collections (as outputs)

Run

- As an analogy with a real experiment, a run of Geant4 starts with 'Beam On' (LHC)
- Within a run, the user cannot change
  - The detector setup
  - The physics setting (processes, models)
- A run is a collection of events with the same detector and physics conditions
- The G4(MT)RunManager class manages the processing of each run, represented by:
  - G4Run class
  - G4UserRunAction for an optional user hook

Track

- The Track is a snapshot of a particle and it is represented by the G4Track class
  - It keeps 'current' information of the particle (i.e. energy, momentum, position, polarization, ..)
  - It is updated after every step
- The track object is deleted when:
  - It goes outside the world volume
  - It disappears in an interaction (decay, inelastic scattering)
  - It is slowed down to zero kinetic energy and there are no 'AtRest' processes
  - It is manually killed by the user
- No track object persists at the end of the event
- G4TrackingManager class manages the tracking
- G4UserTrackingAction is the optional User hook

G4Track status

- After each step the track can change its state
- The status can be:

Track Status	Description
fAlive	The particle is continued to be tracked
fStopButAlive	Kin. Energy = 0, but AtRest process will occur
fStopAndKill	Track has lost identity (has reached world boundary, decayed, ...), Secondaries will be tracked
fKillTrackAndSecondaries	Track and its secondary tracks are killed
fSuspend	Track and its secondary tracks are suspended (pushed to stack)
fPostponeToNextEvent	Track but NOT secondary tracks are postponed to the next event (secondaries are tracked in current event)

* The ones in red can only be set by the User

Step

- G4Step represents a step in the particle propagation
- A G4Step object stores transient information of the step
  - In the tracking algorithm, G4Step is updated each time a process is invoked (e.g. multiple scattering)
- You can extract information from a step after the step is completed, e.g.
  - in ProcessHits() method of your sensitive detector
  - in UserSteppingAction() of your step action class
- The G4Step has the information about the two points (pre-step and post-step) and the 'delta' information of a particle (energy loss on the step, ...)
- Each point knows the volume (and the 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

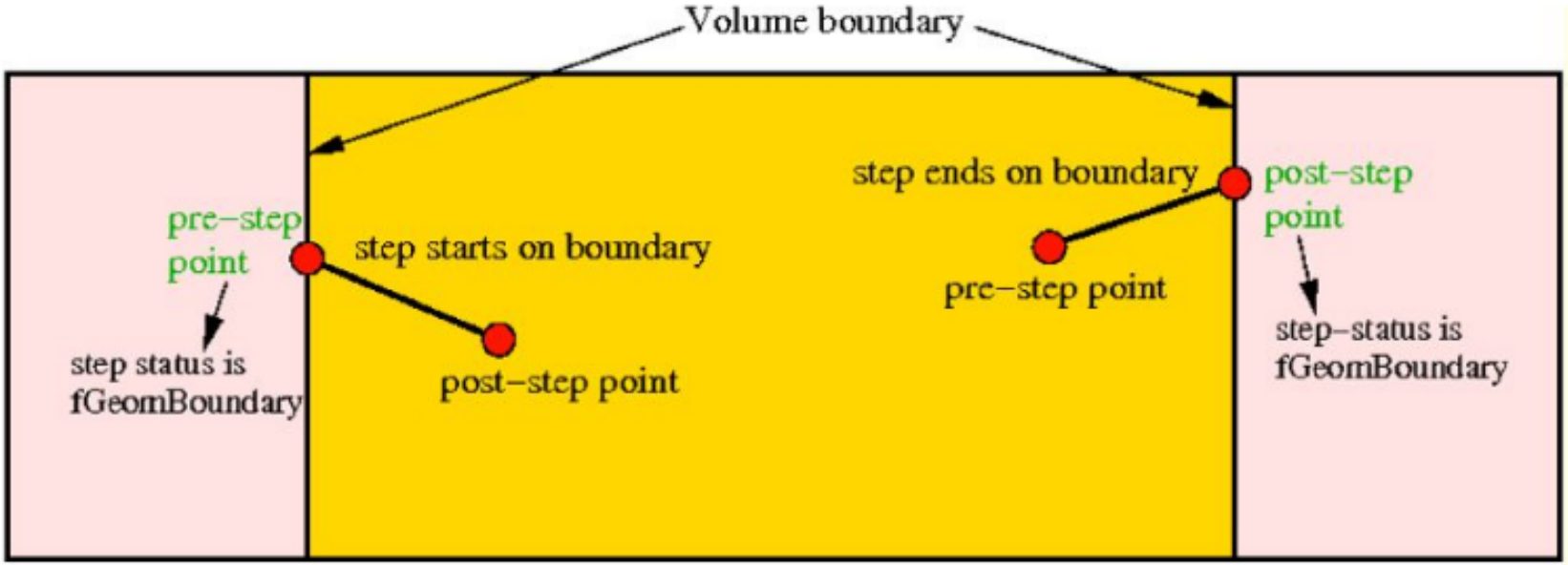
G4Step object

- A G4Step object contains
  - The two endpoints (pre and post step) so one has access to the volumes containing these endpoints
  - Changes in particle properties between the points
    - Difference of particle energy, momentum, ...
    - Energy deposition on step, step length, time-of-flight, ...
  - A pointer to the associated G4Track object
  - Volume hiearchy information
- G4Step provides many Getter... methods to access this information or objects
  - G4StepPoint* GetPreStepPoint(), ...

The Geometry Boundary

- To check, if a step ends on a boundary, one may compare if the physical volume of pre and post-step points are equal
- One can also use the step status
  - Step Status provides information about the process that restricted the step length
  - It is attached to the step points: the pre has the status of the previous step, the post of the current step
  - If the status of POST is fGeometryBoundary, the step ends on a volume boundary (does not apply to world volume)
  - To check if a step starts on a volume boundary you can also use the step status of the PRE-step point

Step Concept and Boundaries



Boundaries Example

```
G4StepPoint* preStepPoint = step -> GetPreStepPoint();
G4StepPoint* postStepPoint = step -> GetPostStepPoint();


// Use the GetStepStatus() method of G4StepPoint to get the status of the
// current step (contained in post-step point) or the previous step
// (contained in pre-step point):
if(preStepPoint -> GetStepStatus() == fGeomBoundary) {
    G4cout << "Step starts on geometry boundary" << G4endl;
}
if(postStepPoint -> GetStepStatus() == fGeomBoundary) {
    G4cout << "Step ends on geometry boundary" << G4endl;
}

// You can retrieve the material of the next volume through the
// post-step point:
G4Material* nextMaterial = step->GetPostStepPoint()->GetMaterial();
```

Optional User Action classes

- Five base classes with virtual methods the user may override to step during the execution of the application
  - G4UserRunAction
  - G4UserEventAction
  - G4UserTrackingAction
  - G4UserStackingAction
  - G4UserSteppingAction
- Default implementation (not purely virtual): Do nothing

G4UserRunAction

This class has three virtual methods which are invoked by G4RunManager for each run:

- G4Run* GenerateRun()
  This method is invoked at the beginning of BeamOn. Because the user can inherit the class G4Run and create his/her own concrete class to store some information about the run, the GenerateRun() method is the place to instantiate such an object
- void BeginOfRunAction(const G4Run*)
  This method is invoked before entering the event loop. This method is invoked after the calculation of the physics tables.
- void EndOfRunAction(const G4Run*)
  This method is invoked at the very end of the run processing. It is typically used for a simple analysis of the processed run.

G4UserEventAction

This class has two virtual methods which are invoked by G4EventManager for each event:

- void BeginOfEventAction(const G4Event*)
  This method is invoked before converting the primary particles to G4Track objects. A typical use of this method would be to initialize and/or book histograms for a particular event.
- void EndOfEventAction(const G4Event*)
  This method is invoked at the very end of event processing. It is typically used for a simple analysis of the processed event.

G4UserStackingAction

This class has three virtual methods, ClassifyNewTrack, NewStage and PrepareNewEvent which the user may override in order to control the various track stacking mechanisms.

- G4ClassificationOfNewTrack ClassifyNewTrack(const G4Track*)

  It's invoked by G4StackManager whenever a new G4Track object is "pushed" onto a stack by G4EventManager.

  G4ClassificationOfNewTrack has four possible values:
  - fUrgent - track is placed in the urgent stack
  - fWaiting - track is placed in the waiting stack, and will not be simulated until the urgent stack is empty
  - fPostpone - track is postponed to the next event
  - fKill - the track is deleted immediately and not stored in any stack

These assignments may
be made based on the origin of the track which is obtained as follows:

G4int parent_ID = aTrack->get_parentID();

where:

- parent_ID = 0 indicates a primary particle
- parent_ID > 0 indicates a secondary particle
- parent_ID < 0 indicates postponed particle from previous event.

G4UserStackingAction

- void NewStage()
  It's invoked when the urgent stack is empty and the waiting stack contains at least one G4Track object.
- void PrepareNewEvent()
  It's invoked at the beginning of each event. At this point no primary particles have been converted to tracks, so the urgent and waiting stacks are empty.

G4UserSteppingAction

- void UserSteppingAction(const g4Step*)
  Get information about particles; kills tracks under specific circumstances

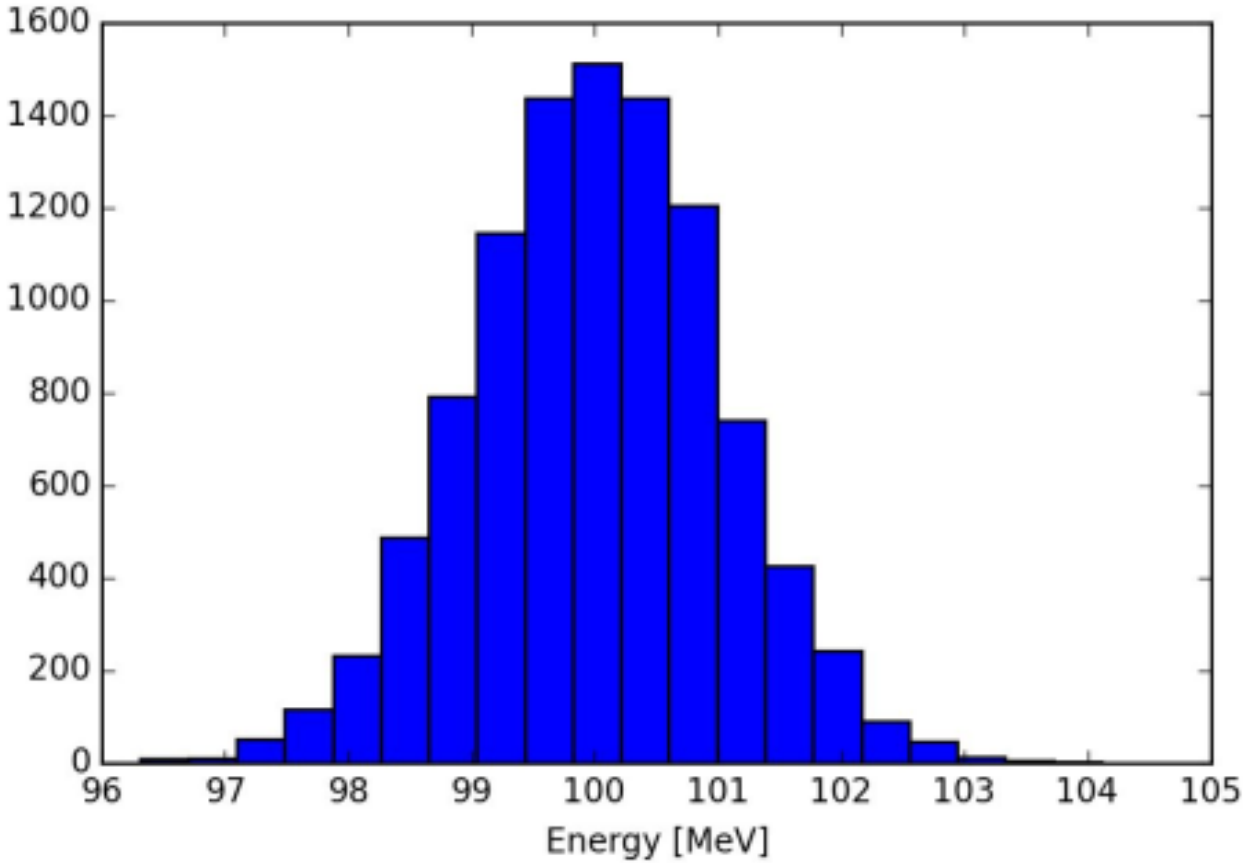
Geant4 Analysis Classes

- A basic analysis interface is available in Geant4 for histograms (1D and 2D) and ntuples
- Unified interface to support different output formats
  - ROOT, CSV, AIDA XML, and HBOOK
  - Code is the same, just change one line to switch from one to an other
- Everything is done using G4AnalysisManager

Selection of output format is performed by including a proper header file:

```
#ifndef MyAnalysis_h
#define MyAnalysis_h 1

#include "g4root.hh"
//#include "g4xml.hh"
//#include "g4csv.hh"  // can be used only with ntuples

#endif
```

Histograms


1600
1400
1200
1000
800
600
400
200
0
96
97
98
99
100
101
102
103
104
105
Energy [MeV]

Open file and book histograms

```
#include "MyAnalysis.hh"

void MyRunAction::BeginOfRunAction(const G4Run* run)
{
  // Get analysis manager
  G4AnalysisManager* man = G4AnalysisManager::Instance();
  man->SetVerboseLevel(1);
  man->SetFirstHistoId(1);

  // Creating histograms
  man->CreateH1("h", "Title", 100, 0., 800*MeV);
  man->CreateH1("hh", "Title", 100, 0., 10*MeV);

 // Open an output file
  man->OpenFile("myoutput");
}
```

Start numbering of histograms from ID=1

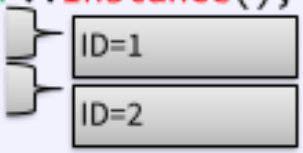
ID=1

ID=2

Open output file

Fill histograms and write the file

```
#include "MyAnalysis.hh"

void MyEventAction::EndOfEventAction(const G4Run* aRun)
{
  auto man = G4AnalysisManager::Instance();
  man->FillH1(1, fEnergyAbs);
  man->FillH1(2, fEnergyGap);
}

MyRunAction::~MyRunAction()
{
  auto man = G4AnalysisManager::Instance();
  man->Write();
}

int main()
{
  ...
  auto man = G4AnalysisManager::Instance();
  man->CloseFile();
}
```



Ntuples

ParticleID	Energy	x	y
0	99.5161753	-0.739157031	-0.014213165
1	98.0020355	1.852812521	1.128640204
2	100.0734469	0.863203688	-0.277949199
3	99.3508677	-2.063452685	-0.898594988
4	101.2505954	1.030581054	0.736468229
5	98.9849841	-1.464509417	-1.065372115
6	101.1547644	1.121931704	-0.203319254
7	100.8876748	0.012068917	-1.283410959
8	100.3013861	1.852532119	-0.520615895
9	100.6295882	1.084122362	0.556967258
10	100.4887681	-1.021971662	1.317380892
11	101.6716567	0.614222096	-0.483530242
12	99.1083093	-0.776034456	0.203524549
13	97.3595776	0.814378204	-0.690615126
14	100.7264612	-0.408732803	-1.278746667

Ntuple Support

- g4tools support ntuples
  - any number of ntuples
  - any number of columns
  - supported types: int/float/double
- For more complex tasks (other functionality of ROOT TTrees) have to link ROOT directly

Book Ntuples

```
#include "MyAnalysis.hh"

void MyRunAction::BeginOfRunAction(const G4Run* run)
{
 // Get analysis manager
  G4AnalysisManager* man = G4AnalysisManager::Instance();
  man-> SetFirstNtupleId(1);

 // Creating ntuples
  man->CreateNtuple("name", "Title");
  man->CreateNtupleDColumn("Eabs");
  man->CreateNtupleDColumn("Egap");
  man->FinishNtuple();

  man->CreateNtuple("name2","title2")
  man->CreateNtupleIColumn("ID");
  man->FinishNtuple();
}
```

Start numbering of ntuples from ID=1

ID=1

ID=2

Fill Ntuples

```
#include "MyAnalysis.hh"

void MyEventAction::EndOfEventAction(const G4Run* aRun)
{
  G4AnalysisManager* man = G4AnalysisManager::Instance();
  man->FillNtupleDColumn(1, 0, fEnergyAbs);
  man->FillNtupleDColumn(1, 1, fEnergyGap);
  man->AddNtupleRow(1);

  man->FillNtupleIColumn(2, 0, fID);
  man->AddNtupleRow(2);

}
```

ID=1, columns 0, 1

ID=2, column 0

Sensitive Detectors (SD)

- A logical volume becomes sensitive if it has a pointer to a sensitive detector (G4VSensitiveDetector)
  - A sensitive detector can be instantiated several times, where the instances are assigned to different logical volumes
- Two possibilities to make use of the SD functionality:
  - Create your own sensitive detector (using class inheritance)
  - Use Geant4 built-in tools: Primitive scorers

Adding sensitivity to a logical volume

- Create an instance of a sensitive detector and register it to the SensitiveDetector Manager
- Assign the pointer of your SD to the logical volume of your detector geometry
- Must be done in ConstructSDandField() of the user geometry class

```
G4VSensitiveDetector* mySensitive
   = new MySensitiveDetector(SDname="/MyDetector");
```
create instance

```
G4SDManager* sdMan =G4SDManager::GetSDMpointer();
sdMan->AddNewDetector(mySensitive);
```
Register to the SD manager

```
SetSensitiveDetector("LVname",mySensitive);
```
assign to logical volume

Name of the logical volume

Extract useful information

- Geant4 provides a number of primitive scorers, each one accumulating one physics quantity (e.g. total dose) for an event
- This is alternative to the customized sensitive detectors, which can be used with full flexibility to gain complete control
- It is convenient to use primitive scorers instead of user-defined sensitive detectors when:
  - you are not interested in recording each individual step, but accumulating physical quantities for an event or a run
  - you have not too many scorers

G4MultiFunctionalDetector

- G4MultiFunctionalDetector is a concrete class derived from G4VSensitiveDetector
- It should be assigned to a logical volume as a kind of (ready-for-the-use) sensitive detector
- It takes an arbitrary number of G4VPrimitiveScorer classes, to define the scoring quantities that you need
  - Each G4VPrimitiveScorer accumulates one physics quantity for each physical volume
  - E.g. G4PSDoseScorer (a concrete class of G4VPrimitiveScorer provided by Geant4) accumulates dose for each cell
- By using this approach, there is no need to implement sensitive detectors and hit classes!

G4VPrimitiveScorer

- Primitive scorers (classes derived from G4VPrimitiveScorer) have to be registered to the G4MultiFunctionalDetector
  - RegisterPrimitive()
  - RemovePrimitive()
- They are designed to score one kind of quantity (surface flux, total dose) and to generate one hit collection per event
  - automatically named as ⟨MultiFunctionalDetectorName⟩/⟨PrimitiveScorerName⟩
  - hit collections can be retrieved in the EventAction or RunAction (as those generated by sensitive detectors)
  - do not share the same primitive scorer object among multiple G4MultiFunctionalDetector objects (results may mix up!)

For example

```
MyDetectorConstruction::ConstructSDandField()
{
   G4MultiFunctionalDetector* myScorer = new
   G4MultiFunctionalDetector("myCellScorer");

    myCellLog->SetSensitiveDetector(myScorer);

    G4VPrimitiveScorer* totalSurfFlux = new
      G4PSFlatSurfaceFlux("TotalSurfFlux");
    myScorer->RegisterPrimitive(totalSurfFlux);
    G4VPrimitiveScorer* totalDose =   new
      G4PSDoseDeposit("TotalDose");
    myScorer->RegisterPrimitive(totalDose);
}
```

instantiate multi-functional detector

attach to volume

create a primitive scorer (surface flux) and register it

create a primitive scorer (total dose) and register it

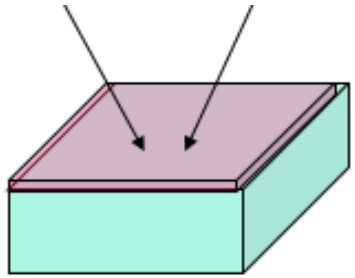
Some primitive scorers that you may find useful

Primitive Scorers:

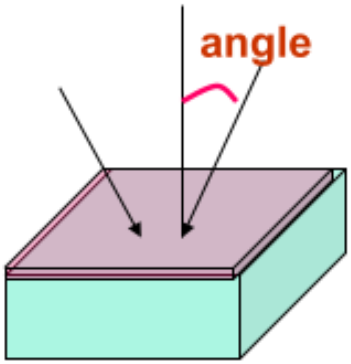
- Track length
  - G4PSTrackLength, G4PSPassageTrackLength
- Deposited energy
  - G4PSEnergyDepsit, G4PSDoseDeposit
- Current/Flux
  - G4PSFlatSurfaceCurrent,
  - G4PSSphereSurfaceCurrent,G4PSPassageCurrent,
  - G4PSFlatSurfaceFlux, G4PSCellFlux, G4PSPassageCellFlux
- Others
  - G4PSMinKinEAtGeneration, G4PSNofSecondary, G4PSNofStep,
  - G4PSCellCharge

A closer look at some scorers

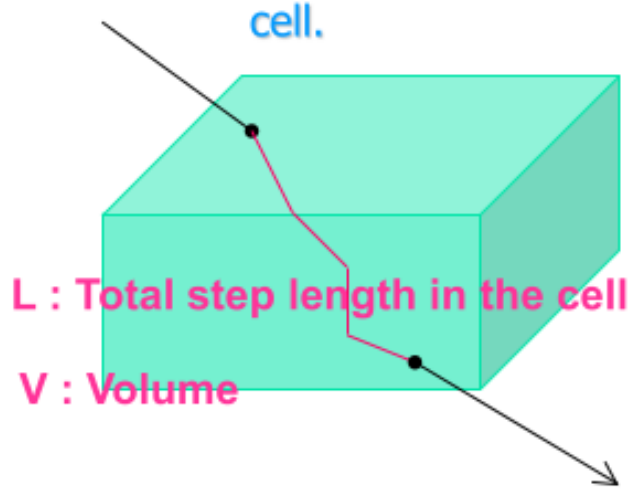
SurfaceCurrent :
Count number of injecting particles at defined surface.



CellFlux :
Sum of L / V of injecting particles in the geometrical cell.



SurfaceFlux :
Sum up 1/cos(angle) of injecting particles at defined surface



V : Volume

G4VSDFilter

- A G4VSDFilter can be attached to G4VPrimitiveScorer to define which kind of tracks have to be scored (e.g. one wants to know surface flux of protons only)
  - G4SDChargeFilter (accepts only charged particles)
  - G4SDNeutralFilter (accepts only neutral particles)
  - G4SDKineticEnergyFilter (accepts tracks in a defined range of kinetic energy)
  - G4SDParticleFilter (accepts tracks of a given particle type)
  - G4VSDFilter (base class to create user-customized filters)

Example

```
MyDetectorConstruction::ConstructSDandField()
{
   G4VPrimitiveScorer* protonSurfFlux
   = new G4PSFlatSurfaceFlux("pSurfFlux");
    G4VSDFilter* protonFilter = new
       G4SDParticleFilter("protonFilter");
    protonFilter->Add("proton");

    protonSurfFlux->SetFilter(protonFilter);

    myScorer->RegisterPrimitive(protonSurfFlux);
}
```

create a primitive scorer (surface flux), as before

create a particle filter and add protons to it

register the filter to the primitive scorer

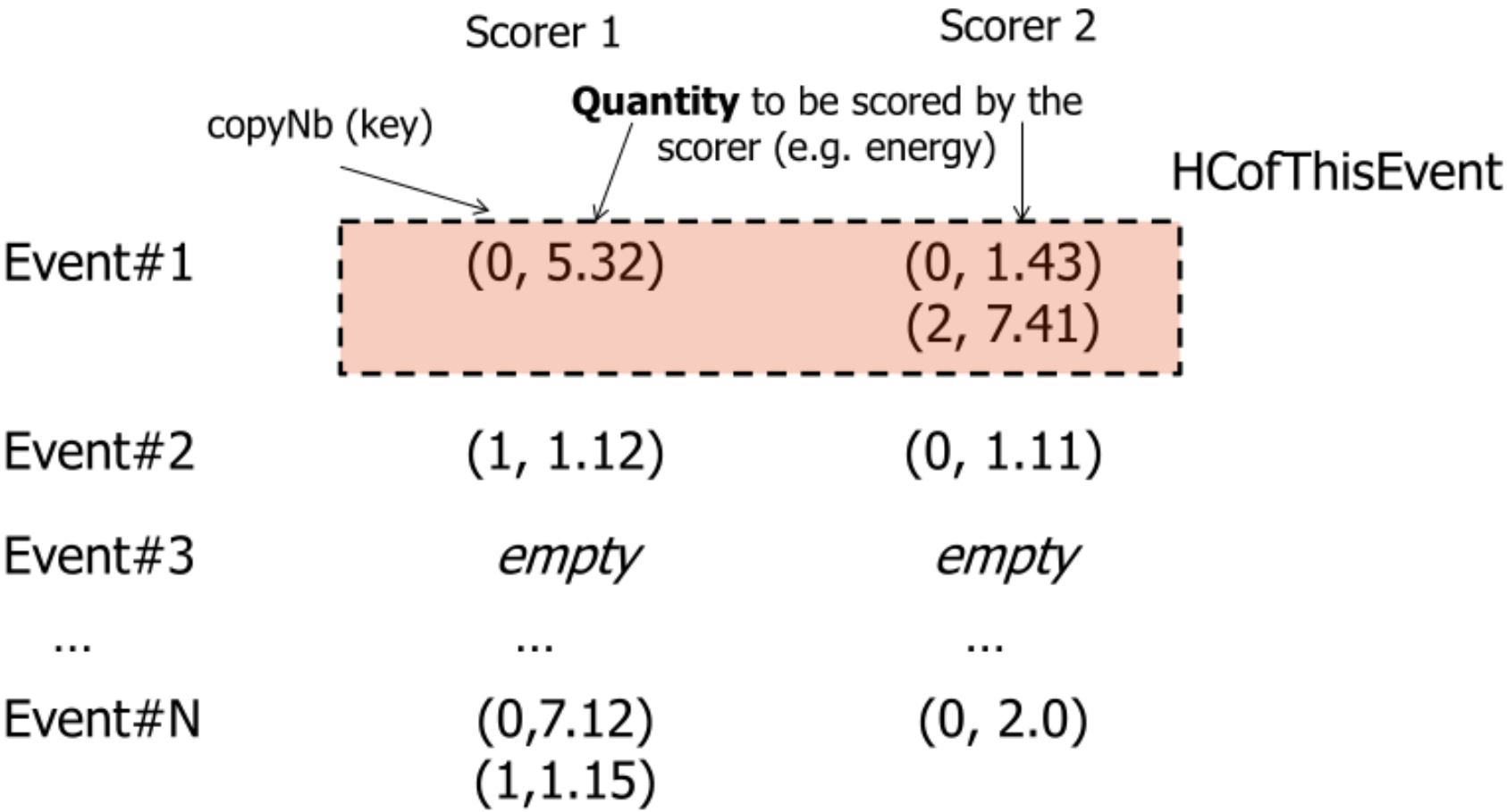
register the scorer to the multifunc detector (as shown before)

How to retrieve information

Each scorer creates a hit collection, which is attached to the G4Event object

- Can be retrieved and read at the end of the event, using an integer ID
- Hits collections mapped as G4THitsMap¡G4double¿* so can loop on the individual entries
- Operator += provided which automatically sums up all hits (no need to loop manually)

How to retrieve information



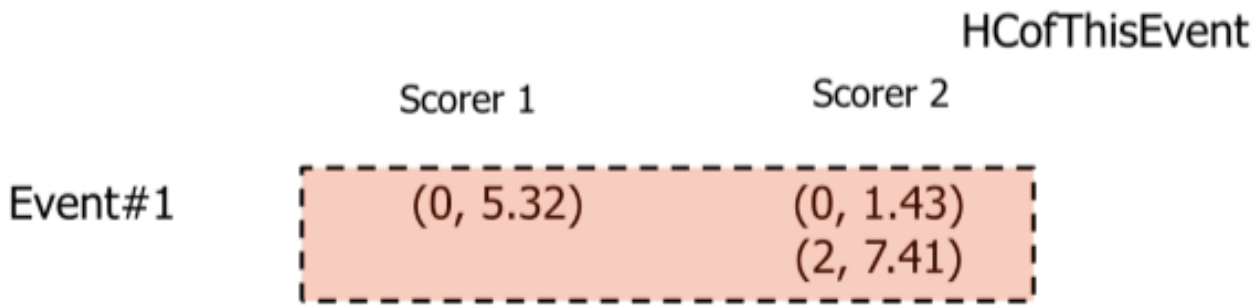
How to retrieve information

```
//needed only once
G4int collID = G4SDManager::GetSDMpointer()
  ->GetCollectionID("myCellScorer/TotalSurfFlux");
```

Get **ID** for the collection (given the name)

```
G4HCofThisEvent* HCE = event->GetHCofThisEvent();
```

Get **all HC** available in this event



How to retrieve information

```
//needed only once
G4int collID = G4SDManager::GetSDMpointer()
  ->GetCollectionID("myCellScorer/TotalSurfFlux");
```

Get **ID** for the collection (given the name)

```
G4HCofThisEvent* HCE = event->GetHCofThisEvent();
```

Get **all HC** available in this event

```
G4THitsMap<G4double>* evtMap =
    static_cast<G4THitsMap<G4double>*>
    (HCE->GetHC(collID));
```

Get the HC with the **given ID** (need a cast)

HCofThisEvent

	Scorer 1	Scorer 2
Event#1	(0, 5.32)	(0, 1.43) (2, 7.41)

How to retrieve information

```
//needed only once
G4int collID = G4SDManager::GetSDMpointer()
  ->GetCollectionID("myCellScorer/TotalSurfFlux");
```

Get **ID** for the collection (given the name)

```
G4HCofThisEvent* HCE = event->GetHCofThisEvent();
```

Get **all HC** available in this event

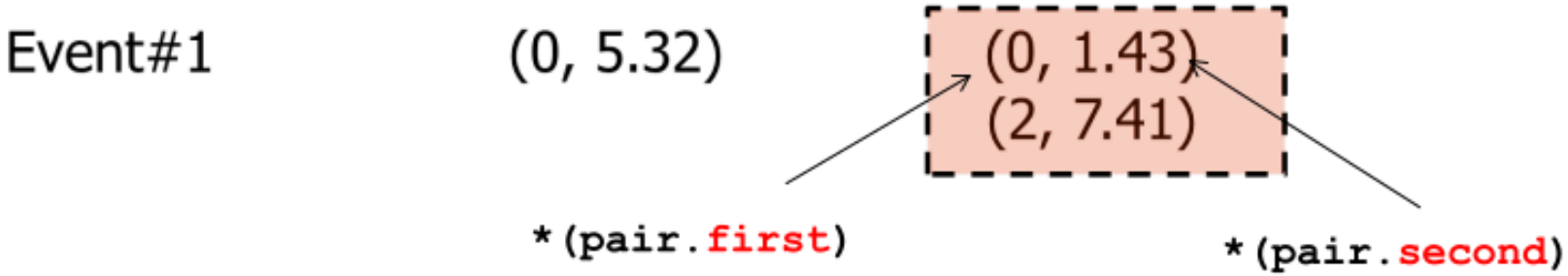
```
G4THitsMap<G4double>* evtMap =
    static_cast<G4THitsMap<G4double>*>
    (HCE->GetHC(collID));
```

Get the HC with the **given ID** (need a cast)

```
for (auto pair : *(evtMap->GetMap())) {
    G4double flux = *(pair.second);
    G4int copyNb  = *(pair.first);
}
```

Loop over the **individual entries** of the HC: the key of the map is the copyNb, the other field is the real content

How to retrieve information




```
for (auto pair : *(evtMap->GetMap())) {
    G4double flux = *(pair.second);
    G4int copyNb  = *(pair.first);
}
```

Loop1: copyNb = 0, value = 1.43
Loop2: copyNb = 2, value = 7.41

The ingredients of user SD

- A powerful and flexible way of extracting information from the physics simulation is to define your own SD
- Derive your own concrete classes from the base classes and customize them according to your needs

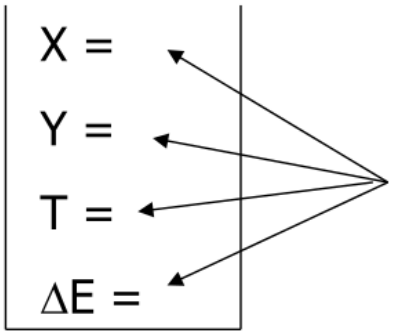
Hit: G4VHit
Hits Collection: G4THitsCollection
Sensitive Detector: G4VSensitiveDetector

Hit class

- Hit is a user-defined class which derives from the base class G4VHit. Two virtual methods
  - Draw()
  - Print()
- You can store various types of information by implementing your own concrete Hit class
- Typically, one may want to record information like
  - Position, time and ΔE of a step
  - Momentum, energy, position, volume, particle type of a given track
  - etc.

Hit class

A “Hit” is like a “container”, a empty box which will store the information retrieved step by step



The Hit **concrete class** (derived by `G4VHit`) must be written by the user: the user must decide **which variables** and/or information the hit should store and **when** store them

The Hit objects are created and filled by the SensitiveDetector class (invoked at each step in detectors defined as sensitive). Stored in the “HitCollection”, attached to the G4Event: can be retrieved at the end of the event

Hit class

Example

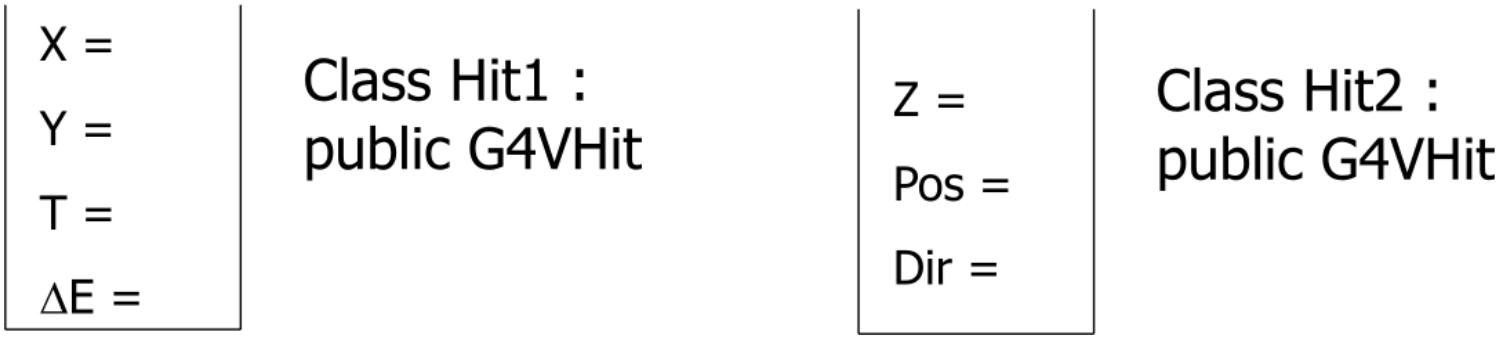
```
// header file: MyHit.hh

#include "G4VHit.hh"

class MyHit : public G4VHit {

 public:
   MyHit();
   virtual ~MyHit();
   ...

   inline void SetEnergyDeposit(G4double energy) { energyDeposit = energy; }

   inline G4double GetEnergyDeposit() { return energyDeposit;}

  ... // more get and set methods

private:
   G4double energyDeposit;
  ... // more data members
};
```

public methods to handle data member

data member (private)

Geant4 Hits

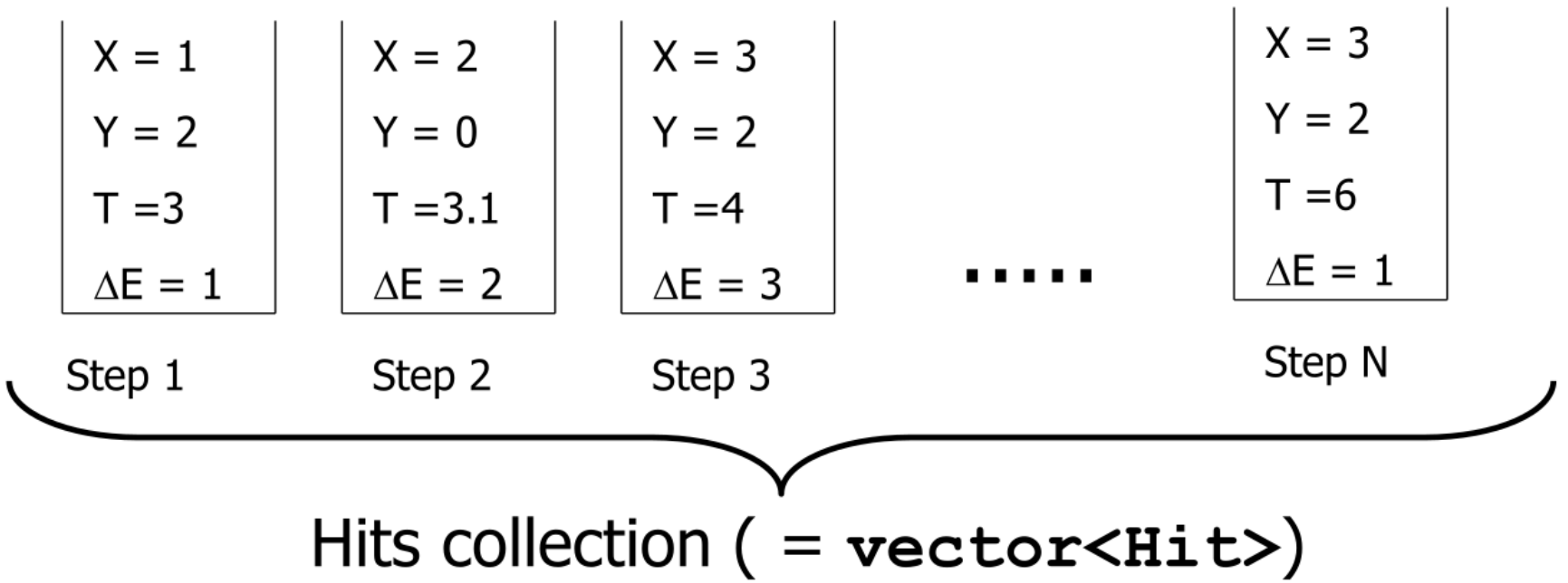
Since in the simulation one may have different sensitive detectors in the same setup (e.g. a calorimeter and a Si detector), it is possible to define many Hit classes (all derived by G4VHit) storing different information



More general than the simple (copyNb, quantity) structure used by the Geant4 native scorers

Hits Collection

At each step in a detector defined as sensitive, the method ProcessHit() of the user SensitiveDetector class is invoked: it must create, fill and store the Hit objects

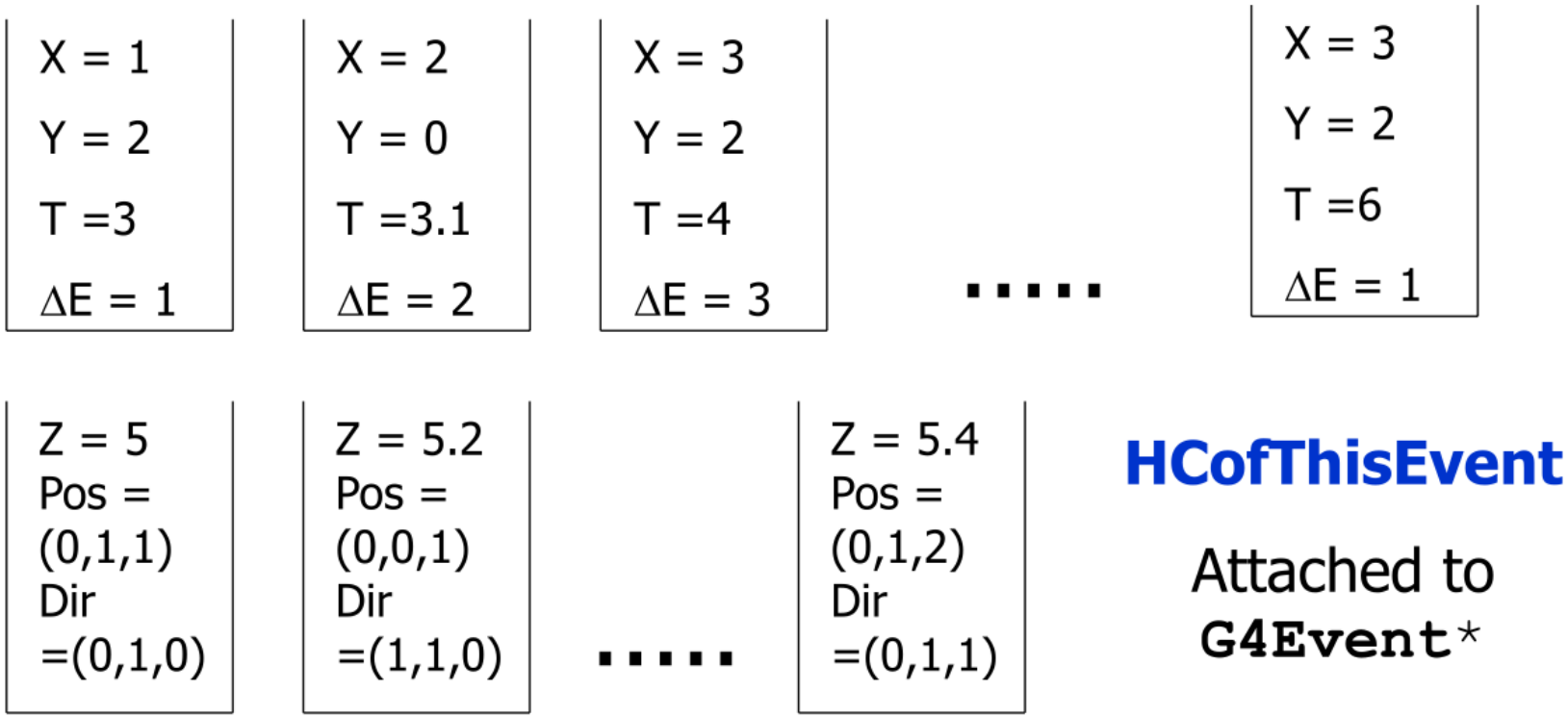


Hits Collection

- Once created in the sensitive detectors, objects of the concrete hit class must be stored in a dedicated collection
  - Template class G4THitsCollection¡MyHit¿, which is actually a vector of MyHit*
- The hits collections can be accesses in different phases of tracking
  - At the end of each event, through the G4Event (a-posteriori event analysis)
  - During event processing, through the Sensitive Detector Manager G4SDManager (event filtering)

The HCofThisEvent

Remember that you may have many kinds of Hits (and Hits Collections)



Hits Collections of an event

- A G4Event object has a G4HCofThisEvent object at the end of the event processing (if it was successful)
  - The pointer to the G4HCofThisEvent object can be retrieved using the G4Event::GetHCofThisEvent() method
- The G4HCofThisEvent stores all hits collections created within the event
  - Hit collections are accessible and can be processes e.g. in the EndOfEventAction() method of the User Event Action class
  - Transient: information cleaned up at each new event

SD and Hits

- Using information from particle steps, a sensitive detector either
  - constructs, fills and stores one (or more) hit object
  - accumulates values to existing hits
- Hits objects can be filled with information in the ProcessHits() method of the SD concrete user class
  - This method has pointers to the current G4Step and to the G4TouchableHistory of the Parallel World (if defined)

Sensitive Detector (SD)

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

Sensitive Detector

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

```
// header file: MySensitiveDetector.hh
#include "G4VSensitiveDetector.hh"
...
class MySensitiveDetector : public G4VSensitiveDetector {

 public:
    MySensitiveDetector(G4String name);
    virtual ~MySensitiveDetector();

    virtual void Initialize(G4HCofThisEvent*HCE);
    virtual G4bool ProcessHits(G4Step* step,
                               G4TouchableHistory* ROhist);
    virtual void EndOfEvent(G4HCofThisEvent*HCE);

 private:
    MyHitsCollection * hitsCollection;
    G4int collectionID;
};
```

User concrete SD class

SD implementation: constructor

- Specify a hits collection (by its unique name) for each type of hits considered in the sensitive detector:
  - Insert the name(s) in the collectionName vector

```
MySensitiveDetector::MySensitiveDetector(G4String detectorUniqueName)
                                        : G4VSensitiveDetector(detectorUniquename),
                                          collectionID(-1) {

  collectionName.insert("collection_name");
}
```

Base class 

```
class G4VSensitiveDetector {
 ...
protected:
   G4CollectionNameVector collectionName;
   // This protected name vector must be filled in
   // the  constructor of the concrete class for
   // registering names of hits collections
 ...
};
```

SD implementation: Initialize()

- The Initialize() method is invoked at the beginning of each event
- Construct all hits collections and insert them in the G4HCofThisEvent object, which is passed as argument to Initialize()
  - The AddHitsCollection() method of G4HCofThisEvent requires the collection ID
- The unique collection ID can be obtained with GetCollectionID()
  - GetCollectionID() cannot be invoked in the constructor of this SD
  - Hence, we defined a private data member (collectionID), which is set at the first call of the Initialize() function

```
void MySensitiveDetector::Initialize(G4HCofThisEvent*HCE) {
  if(collectionID < 0)
        collectionID = GetCollectionID(0); // Argument : order of collect.
                                           // as stored in the collectionName
  hitsCollection = new MyHitsCollection
                   (SensitiveDetectorName, collectionName[0]);

  HCE -> AddHitsCollection(collectionID, hitsCollection);
}
```

SD implementation: ProcessHits()

- This ProcessHits() method is invoked for every step in the volume(s) which hold a pointer to this SD (= each volume defined as "sensitive")
- The main mandate of this method is to generate hit(s) or to accumulate data to existing hit objects, by using information from the current step

```
G4bool MySensitiveDetector::ProcessHits(G4Step* step,
                                        G4TouchableHistory*ROhist) {
  MyHit* hit = new MyHit();      // 1) create hit
...
  // some set methods, e.g. for a tracking detector:
  G4double energyDeposit = step -> GetTotalEnergyDeposit();   // 2) fill hit
  hit -> SetEnergyDeposit(energyDeposit); // See implement. of our Hit class
...
  hitsCollection -> insert(aHit);  // 3) insert in the collection
  return true;
}
```

Processing hit information

- Retrieve the pointer of a hits collection with the GetHC()method of G4HCofThisEvent collection using the collection index (a G4int number)
- Index numbers of a hit collection are unique and don't change for a run. The number can be obtained by G4SDManager::GetCollectionID("name");
- Notes:
  - if the collection(s) are not created, the pointers of the collection(s) are NULL: check before trying to access it
  - Need an explicit cast from G4VHitsCollection (see code)

Process hit: example

```
void MyEventAction::EndOfEventAction(const G4Event* event) {

 // index is a data member, representing the hits collection index of the
 // considered collection. It was initialized to -1 in the class constructor
 if(index < 0)   index =
   G4SDManager::GetSDMpointer() -> GetCollectionID("myDet/myColl");

 G4HCofThisEvent* HCE = event-> GetHCofThisEvent();

 MyHitsCollection* hitsColl = 0;
 if(HCE)  hitsColl = (MyHitsCollection*)(HCE->GetHC(index));
```

retrieve index

retrieve all hits collections

retrieve hits collection by index

Be sure that this is non-NULL

Processing hit information

- Loop through the entries of a hits collection to access individual hits
  - Since the HitsCollection is a vector, you can use the [] operator to get the hit object corresponding to a given index
- Retrieve the information contained in this hit (e.g. using the Get/Set methods of the concrete user Hit class) and process it
- Store the output in analysis objects

Process hit: example

```
void MyEventAction::EndOfEventAction(const G4Event* event) {

 // index is a data member, representing the hits collection index of the
 // considered collection. It was initialized to -1 in the class constructor
 if(index < 0)   index =
  G4SDManager::GetSDMpointer() -> GetCollectionID("myDet/myColl");

 G4HCofThisEvent* HCE = event-> GetHCofThisEvent();

 MyHitsCollection* hitsColl = 0;
 if(HCE)  hitsColl = (MyHitsCollection*)(HCE->GetHC(index));

 if(hitsColl)  {
   int numberHits = hitsColl->entries();

   for(int i1= 0; i1 < numberHits ; i1++) {
     MyHit* hit = (*hitsColl)[i1];
     // Retrieve information from hit object, e.g.
     G4double energy = hit -> GetEnergyDeposit;
     ... // Further process and store information
    }
 }
}
```

Be sure that this is non-NULL

cast

loop over individual hits, retrieve the data

The HCofThisEvent

Remember that you may have many kinds of Hits (and Hits Collections)