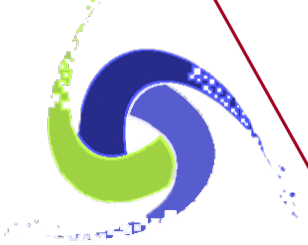


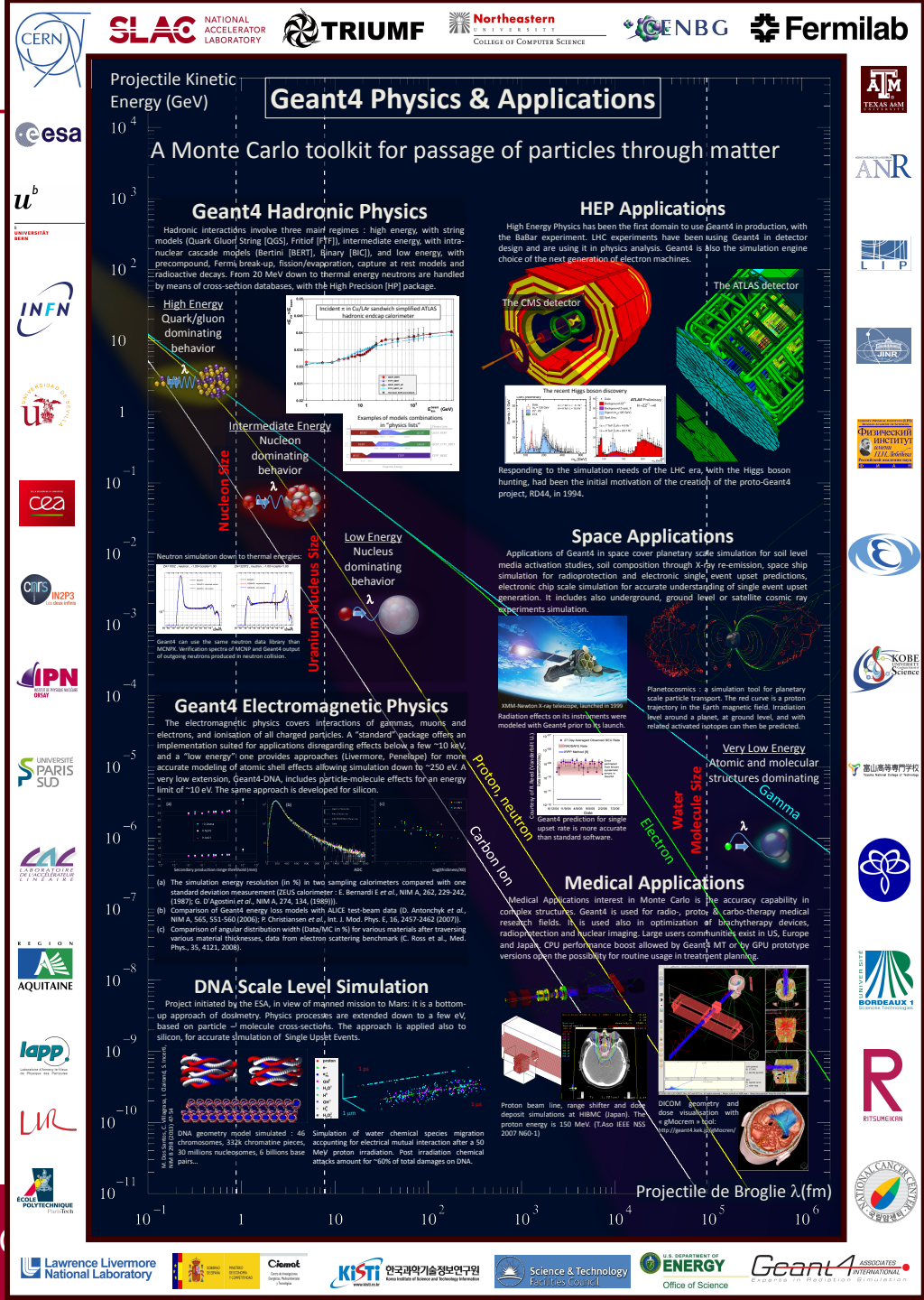
GEANT4
A SIMULATION TOOLKIT

Geant4 Kernel
- part 2

Makoto Asai
SLAC National Accelerator Laboratory
June 17, 2021

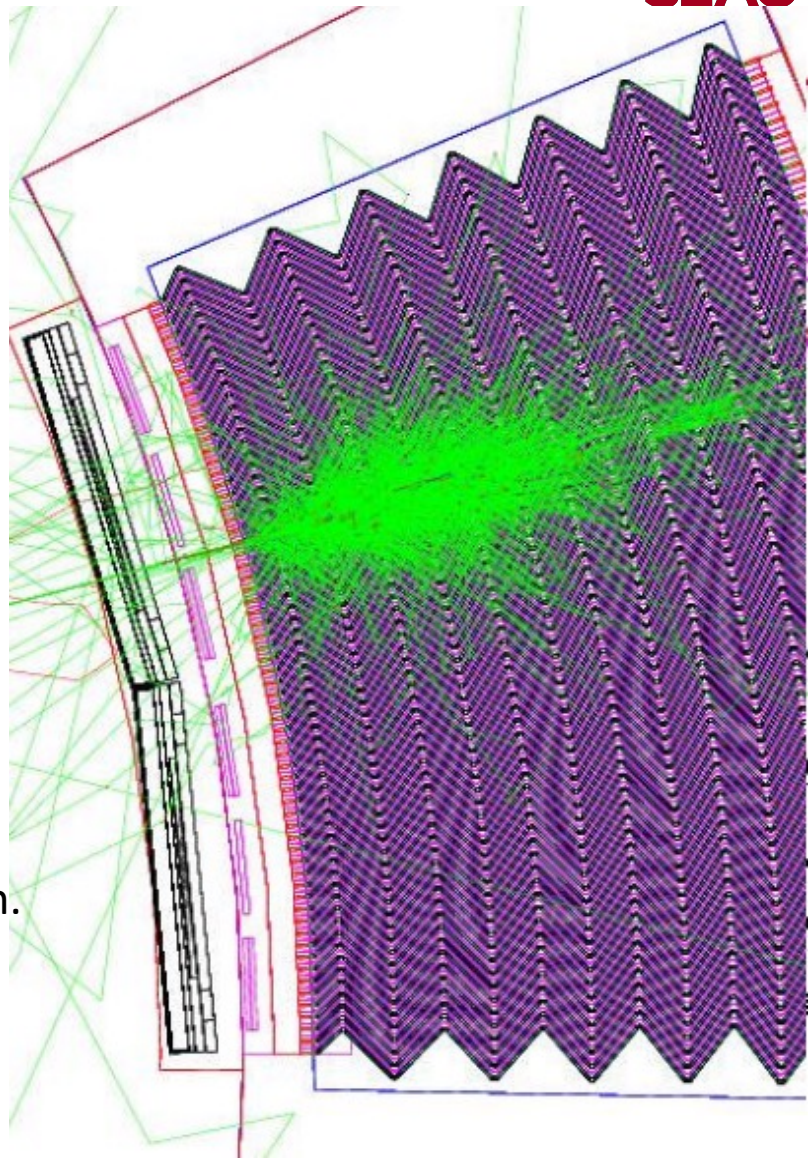
Contents

- Selecting physics models
- Primary particle generator
- Scoring and sensitive detector



Physics models in Geant4

- Geant4 offers
  - Electromagnetic processes
  - Hadronic and nuclear processes
  - Photon/lepton-hadron processes
  - Optical photon processes
  - Decay processes
  - Shower parameterization
  - Event biasing techniques
  - And you can plug-in more
- Geant4 provides sets of alternative physics models so that the user can freely choose appropriate models according to the type of his/her application.
  - For example, some models are more accurate than others at a sacrifice of speed.



User classes

- main()
  - Geant4 does not provide *main().*

  Note : classes written in red are mandatory.
- Initialization classes
  - Use G4RunManager::SetUserInitialization() to define.
  - Invoked at the initialization
    - G4VUserDetectorConstruction
    - G4VUserPhysicsList 
    - G4VUserActionInitialization
- Action classes
  - Instantiated in G4VUserActionInitialization.
  - Invoked during an event loop
    - G4VUserPrimaryGeneratorAction
    - G4UserRunAction
    - G4UserEventAction
    - G4UserStackingAction
    - G4UserTrackingAction
    - G4UserSteppingAction

Select physics processes

- Geant4 does not have any default particles or processes.
  - Even for the particle transportation, you have to define it explicitly.
- Derive your own concrete class from G4VUserPhysicsList abstract base class.
  - Define all necessary particles
  - Define all necessary processes and assign them to proper particles
  - Define cut-off ranges applied to the world (and each region)
- Primarily, the user's task is choosing a "pre-packaged" physics list, that combines physics processes and models that are relevant to a typical application use-cases.
  - If "pre-packaged" physics lists do not meet your needs, you may add or alternate some processes/models.
  - If you are brave enough, you may implement your physics list.

https://geant4-userdoc.web.cern.ch/UsersGuides/PhysicsReferenceManual/html/index.html

Geant4 Homepage

Physics Reference Manual

10.7 (doc Rev5.0)

Search docs

CONTENTS

General Information

Particle Decay

Electromagnetic Interactions

Hadronic Physics in Geant4

Gamma- and Lepto-Nuclear Interactions

Solid State Physics

Docs » Physics Reference Manual

Physics Reference Manual

Scope of this Manual

The Physics Reference Manual provides detailed explanations of the physics implemented in the Geant4 toolkit.

The manual's purpose is threefold:

- to present the theoretical formulation, model, or parameterization of the physics interactions included in Geant4,
- to describe the probability of the occurrence of an interaction and the sampling mechanisms required to simulate it, and
- to serve as a reference for toolkit users and developers who wish to consult the underlying physics of an interaction.

This manual does not discuss code implementation or how to use the implemented physics interactions in a simulation. These topics are discussed in the *User's Guide for Application Developers*. Details of the object-oriented design and functionality of the Geant4 toolkit are given in the *User's Guide for Toolkit Developers*. The *Installation Guide for Setting up |Geant4| in Your Computing Environment* describes how to get the Geant4 code, install it, and run it.

Contents

https://geant4-userdoc.web.cern.ch/UsersGuides/PhysicsListGuide/html/index.html

Geant4 Homepage

PhysicsListGuide

10.7 (doc Rev5.0)

Search docs

CONTENTS:

Physics List Guide

Reference Physics Lists

Electromagnetic physics constructors

Hadronic Physics

Docs » Guide for Physics Lists

Guide for Physics Lists

Scope of this Manual

This guide is a description of the physics lists class which is one of the mandatory user classes for a GEANT4 application. For the most part the "reference" physic lists included in the source distribution are described here as well the modularity and electronic options. Some use cases and areas of application are also described.

Contents:

Most-recommended physics lists

- FTFP_BERT
  - Recommended for most of the use-cases
  - “Reference” to be used as the starting point
- QBBC
  - Recommended for medical and space engineering use-cases
- Shielding
  - Recommended for radiation shielding and deep-underground experiments

```
#include ”QBBC.hh”
int main(int argc,char** argv)
{
  auto* runManager =
    G4RunManagerFactory::CreateRunManager(G4RunManagerType::Default);

  runManager->SetUserInitialization(new B1DetectorConstruction());

  runManager->SetUserInitialization(new QBBC);
```

EM physics options

- EM Option 0 is by default used by all physics lists recommended in the previous slide.
- EM Option 1 (_EMV)
  - Faster than Option 0 with relatively low accuracy
- EM Option 4 (_EMZ)
  - Most accurate, with additional CPU cost
- EM GS (_GS)
  - Same as Option 0 except Goutsmit-Sounderson multiple-scattering mode (more accurate)
- EM SS (_SS)
  - No multiple-scattering, only single-scattering is used.
  - Only for low-energy applications
- EM Liv (_LIV)
  - Livermore EM physics models are used where applicable
- EM DNA (_DNA)
  - For DNA physics/chemistry

Applying EM physics options

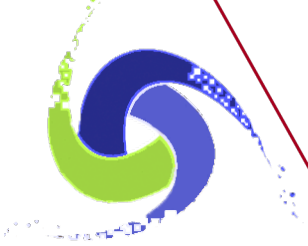
- Use G4PhysListFactory and specify the EM option name appended to your choice of the reference physics list.

```
#include ”G4PhysListFactory.hh”
int main(int argc,char** argv)
{
  auto* runManager = new G4MTRunManager();

  runManager->SetUserInitialization(new B1DetectorConstruction());

  G4PhysListFactory factory;
  auto* physList = factory.GetReferencePhysicsList(”FTFP_BERT_EMZ”);
  runManager->SetUserInitialization(physList);
```

Adding further extensions

- Further extensions can be added to the physics list/
  - Optical photon generation (e.g. Cherenkov)
  - Optical photon transport processes
  - Artificial track killer (e.g. neutron killer)
  - Event biasing

```
#include ”G4PhysListFactory.hh”
#include ”G4NeutronTrackingCut.hh”
int main(int argc,char** argv)
{
  auto* runManager = new G4MTRunManager();
  runManager->SetUserInitialization(new B1DetectorConstruction());

  G4PhysListFactory factory;
  auto* physList = factory.GetReferencePhysicsList(”FTFP_BERT_EMZ”);
  auto* neutronKiller = new G4NeutronTrackingCut();
  neutronKiller->SetTimeLimit(100.*CLHEP::s);
  physList->RegisterPhysics(neutronKiller);
  runManager->SetUserInitialization(physList);
```



GEANT4
A SIMULATION TOOLKIT

Primary vertex and Primary particle

User classes

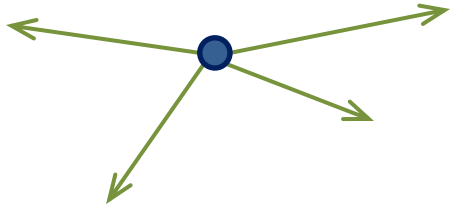
- main()
  - Geant4 does not provide *main().*

  Note : classes written in red are mandatory.
- Initialization classes
  - Use G4RunManager::SetUserInitialization() to define.
  - Invoked at the initialization
    - G4VUserDetectorConstruction
    - G4VUserPhysicsList
    - G4VUserActionInitialization
- Action classes
  - Instantiated in G4VUserActionInitialization.
  - Invoked during an event loop
    - G4VUserPrimaryGeneratorAction 
    - G4UserRunAction
    - G4UserEventAction
    - G4UserStackingAction
    - G4UserTrackingAction
    - G4UserSteppingAction

Primary vertex and primary particle

- Primary particle means particle with which you start an event.
  - E.g. particles made by the primary p-p collision, an alpha particle emitted from radioactive material, a gamma-ray from treatment head, etc.
  - Then Geant4 tracks these primary particles in your geometry with physics interactions and generates secondaries, detector responses and/or scores.
- Primary vertex has position and time. Primary particle has a particle ID, momentum and optionally polarization. One or more primary particles may be associated with a primary vertex. One event may have one or more primary vertices.

G4PrimaryVertex objects = {position, time}



G4PrimaryParticle objects = {PDG, momentum, polarization…}

- Generation of primary vertex/particle is one of the user-mandatory tasks. G4VUserPrimaryGeneratorAction is the abstract base class to control the generation.
  - Actual generation should be delegated to G4VPrimaryGenerator class. Several concrete implementations, e.g. G4ParticleGun, G4GeneralParticleSource, are provided.

G4VUserPrimaryGeneratorAction

- This class is one of mandatory user classes to control the generation of primaries.
  - This class itself should NOT generate primaries but invoke **GeneratePrimaryVertex()** method of primary generator(s) to make primaries.
- Constructor
  - Instantiate primary generator(s)
  - Set default values to it(them)
- GeneratePrimaries() method
  - Invoked at the beginning of each event.
  - Randomize particle-by-particle value(s)
  - Set these values to primary generator(s)
    - Never use hard-coded UI commands
  - Invoke **GeneratePrimaryVertex()** method of primary generator(s)
- Your concrete class of G4VUserPrimaryGeneratorAction must be instantiated in the Build() method of your G4VUserActionInitialization

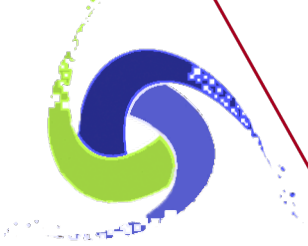
G4VUserPrimaryGeneratorAction

Constructor : Invoked only once

```
MyPrimaryGeneratorAction::MyPrimaryGeneratorAction()
{
    G4int n_particle = 1;
    fparticleGun  = new G4ParticleGun(n_particle);

    // default particle kinematic
    G4ParticleTable* particleTable = G4ParticleTable::GetParticleTable();
    G4ParticleDefinition* particle = particleTable->FindParticle("gamma");
    fparticleGun->SetParticleDefinition(particle);
    fparticleGun->SetParticleMomentumDirection(G4ThreeVector(0.,0.,1.));
    fparticleGun->SetParticleEnergy(100.*MeV);
    fparticleGun->SetParticlePosition(G4ThreeVector(0.,0.,-50*cm));
}
```

Invoked once per each event

```
void MyPrimaryGeneratorAction::GeneratePrimaries(G4Event* anEvent)
{
    fparticleGun->SetParticleMomentum(G4RandomDirection());
    fparticleGun->GeneratePrimaryVertex(anEvent);
}
```



GEANT4
A SIMULATION TOOLKIT

Built-in primary particle generators

G4ParticleGun

- Concrete implementations of G4VPrimaryGenerator
  - A good example for experiment-specific primary generator implementation
- It shoots one primary particle of a certain energy from a certain point at a certain time to a certain direction.
  - Various set methods are available
  - Intercoms commands are also available for setting initial values
- One of most frequently asked questions is :

  I want “particle shotgun”, “particle machinegun”, etc.
- Instead of implementing such a fancy weapon, in your implementation of UserPrimaryGeneratorAction, you can
  - Shoot random numbers in arbitrary distribution
  - Use set methods of G4ParticleGun
  - Use G4ParticleGun as many times as you want
  - Use any other primary generators as many times as you want to make overlapping events

What to do and where to do

- In the constructor of your UserPrimaryGeneratorAction
  - Instantiate G4ParticleGun
  - Set default values by set methods of G4ParticleGun
    - Particle type, kinetic energy, position and direction
- In your macro file or from your interactive terminal session
  - Set values for a run
    - Particle type, kinetic energy, position and direction
- In the GeneratePrimaries() method of your UserPrimaryGeneratorAction
  - Shoot random number(s) and prepare track-by-track or event-by-event values
    - Kinetic energy, position and direction
  - Use set methods of G4ParticleGun to set such values
  - Then invoke GeneratePrimaryVertex() method of G4ParticleGun
  - If you need more than one primary tracks per event, loop over randomization and GeneratePrimaryVertex().

- examples/basic/B5/src/B5PrimaryGeneratorAction.cc is a good example to start with.

G4VUserPrimaryGeneratorAction

```
void T01PrimaryGeneratorAction::
          GeneratePrimaries(G4Event* anEvent)
{ G4ParticleDefinition* particle;
  G4int i = (int)(5.*G4UniformRand());
  switch(i)
  { case 0: particle = positron; break; ... }
  particleGun->SetParticleDefinition(particle);
  G4double pp =
    momentum+(G4UniformRand()-0.5)*sigmaMomentum;
  G4double mass = particle->GetPDGMass();
  G4double Ekin = sqrt(pp*pp+mass*mass)-mass;
  particleGun->SetParticleEnergy(Ekin);
  G4double angle = (G4UniformRand()-0.5)*sigmaAngle;
  particleGun->SetParticleMomentumDirection
            (G4ThreeVector(sin(angle),0.,cos(angle)));
  particleGun->GeneratePrimaryVertex(anEvent);
}
```

- You can repeat this for generating more than one primary particles.

Interfaces to HEPEvt and HepMC

- Concrete implementations of G4VPrimaryGenerator
  - A good example for experiment-specific primary generator implementation
- G4HEPEvtInterface
  - Suitable to /HEPEVT/ common block, which many of (FORTRAN) HEP physics generators are compliant to.
  - ASCII file input
- G4HepMCInterface
  - An interface to HepMC class, which a few new (C++) HEP physics generators are compliant to.
  - ASCII file input or direct linking to a generator through HepMC.

G4GeneralParticleSource

- A concrete implementation of G4VPrimaryGenerator
  - Suitable especially to space applications

```
MyPrimaryGeneratorAction::
          MyPrimaryGeneratorAction()
{ generator = new G4GeneralParticleSource; }
void MyPrimaryGeneratorAction::
          GeneratePrimaries(G4Event* anEvent)
{ generator->GeneratePrimaryVertex(anEvent); }
```

- Detailed description

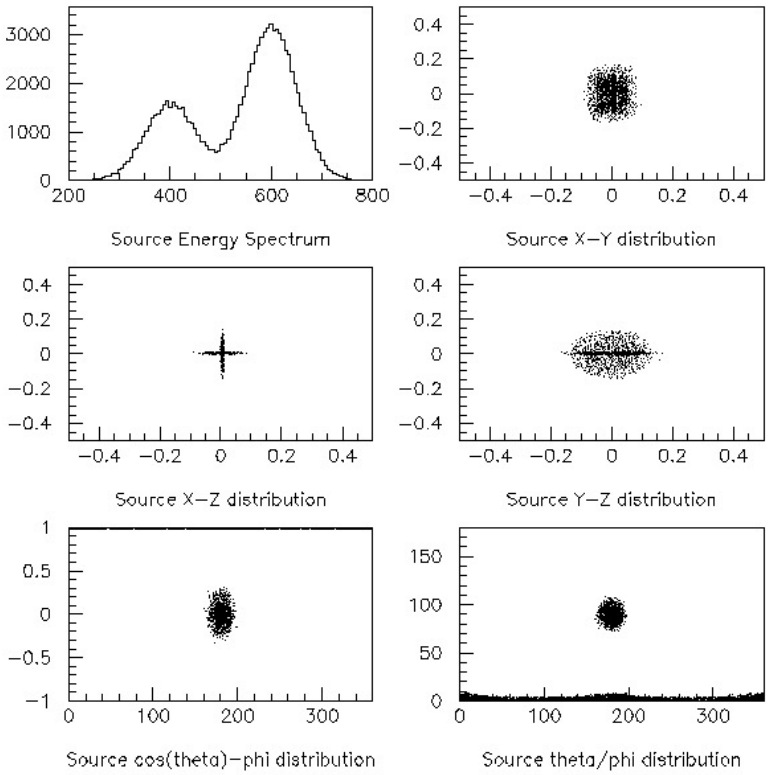
  Section 2.7 of Application Developer's Guide

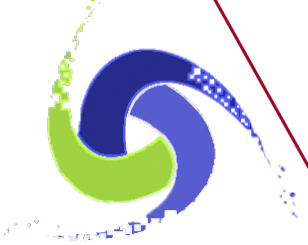
Example commands of General Particle Source

```
# two beams in a generator
#
# beam #1
# default intensity is 1 now change to 5.
/gps/source/intensity 5.
#
/gps/particle proton
/gps/pos/type Beam
#
# the incident surface is in the y-z plane
/gps/pos/rot1 0 1 0
/gps/pos/rot2 0 0 1
#
# the beam spot is centered at the origin and is of
# 1d gaussian shape with a 1 mm central plateau
/gps/pos/shape Circle
/gps/pos/centre  0. 0. 0. mm
/gps/pos/radius 1. mm
/gps/pos/sigma_r .2 mm
#
# the beam is travelling along the X_axis with
# 5 degrees dispersion
/gps/ang/rot1 0 0 1
/gps/ang/rot2 0 1 0
/gps/ang/type beam1d
/gps/ang/sigma_r 5. deg
#
# the beam energy is in gaussian profile
# centered at 400 MeV
/gps/ene/type Gauss
/gps/ene/mono 400 MeV
/gps/ene/sigma 50. MeV
```

(macro continuation…)

```
# beam #2
# 2x the instensity of beam #1
/gps/source/add 10.
#
# this is a electron beam
/gps/particle e-
/gps/pos/type Beam
# it beam spot is of 2d gaussian profile
# with a 1x2 mm2 central plateau
# it is in the x-y plane centred at the orgin
/gps/pos/centre  0. 0. 0. mm
/gps/pos/halfx 0.5 mm
/gps/pos/halfy 1. mm
/gps/pos/sigma_x 0.1 mm
# the spread in y direction is stronger
/gps/pos/sigma_y 0.2 mm
#
#the beam is travelling along -Z_axis
/gps/ang/type beam2d
/gps/ang/sigma_x 2. deg
/gps/ang/sigma_y 1. deg
# gaussian energy profile
/gps/ene/type Gauss
/gps/ene/mono 600 MeV
/gps/ene/sigma 50. MeV
```



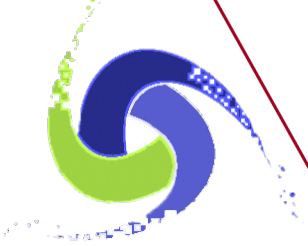


GEANT4
A SIMULATION TOOLKIT

Retrieving information from Geant4

Extract useful information

- Given geometry, physics and primary track generation, Geant4 does proper physics simulation “silently”.
  - You have to do something to extract information useful to you.
- There are three ways:
  - Built-in scoring commands
    - Most commonly-used physics quantities are available.
  - Use scorers in the tracking volume
    - Create scores for each event
    - Create own Run class to accumulate scores
  - Assign G4VSensitiveDetector to a volume to generate “hit”.
    - Use user hooks (G4UserEventAction, G4UserRunAction) to get event / run summary
- You may also use user hooks (G4UserTrackingAction, G4UserSteppingAction, etc.)
  - You have full access to almost all information
  - Straight-forward, but do-it-yourself



GEANT4
A SIMULATION TOOLKIT

Command-based scoring

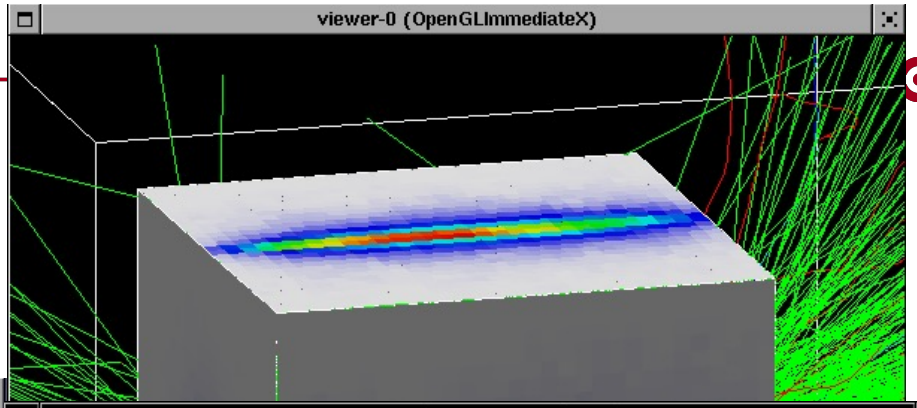
Command-based scoring

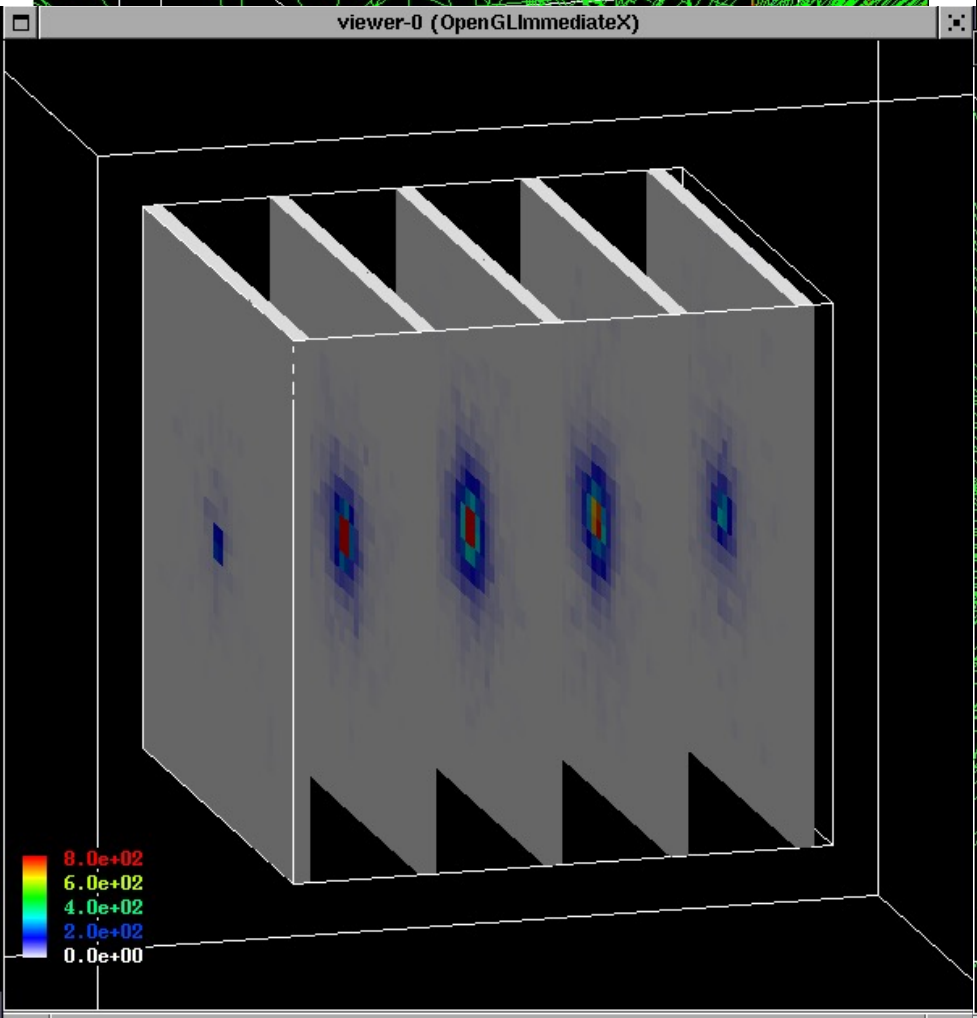
- Command-based scoring functionality offers the built-in scoring mesh and various scorers for commonly-used physics quantities such as dose, flux, etc.
  - Due to small performance overhead, it does not come by default.
- To use this functionality, access to the G4ScoringManager pointer after the instantiation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

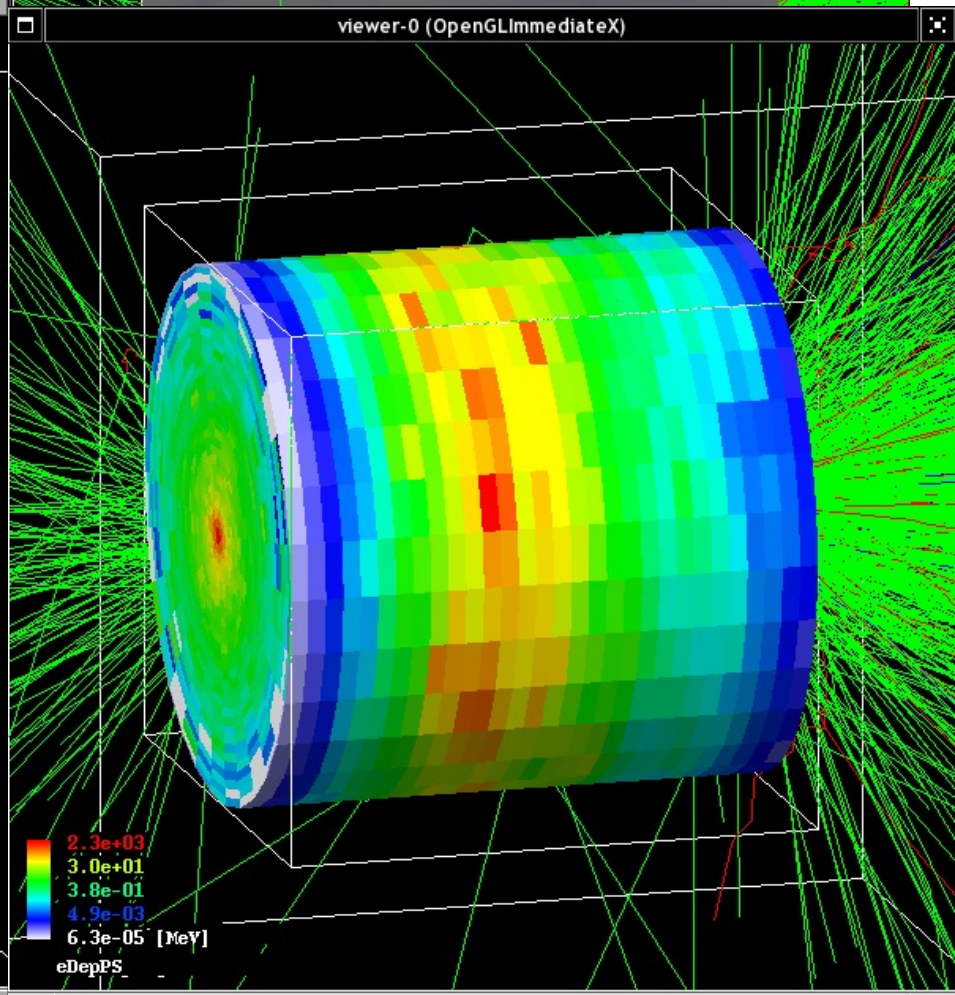
```
#include "G4ScoringManager.hh"
int main()
{
  auto* runManager = new G4MTRunManager;
  auto* scoringManager = G4ScoringManager::GetScoringManager();
   …
```

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

Command-based scorers







Define a scoring mesh

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

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

- The mesh geometry can be completely independent to the real material geometry.

Scoring quantities

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

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

Filter

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

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

Same primitive scorers with different filters may be defined.

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

Drawing a score

- Projection

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

- Slice

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

- Color map
  - By default, linear and log-scale color maps are available.
  - Minimum and maximum values can be defined by /score/colorMap/setMinMax command. Otherwise, min and max values are taken from the current score.

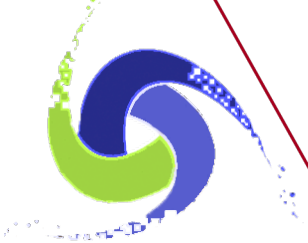
Write scores to a file

- Single score

  /score/dumpQuantityToFile <mesh_name> <scorer_name> <file_name>
- All scores

  /score/dumpAllQuantitiesToFile <mesh_name> <file_name>

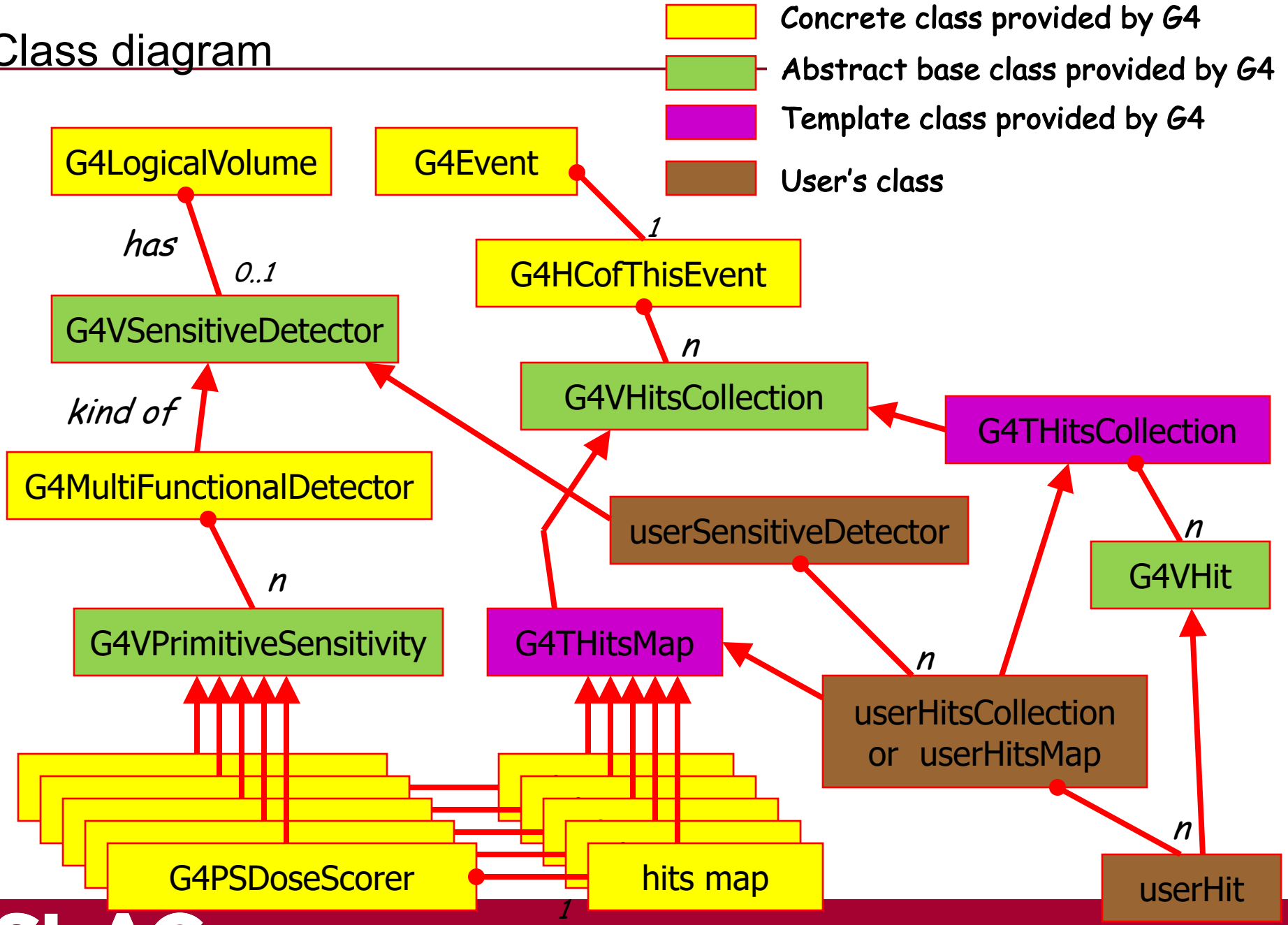
- By default, values are written in CSV.
- By creating a concrete class derived from G4VScoreWriter base class, the user can define his own file format.
  - Example in /examples/extended/runAndEvent/RE03
  - User's score writer class should be registered to G4ScoringManager.



GEANT4
A SIMULATION TOOLKIT

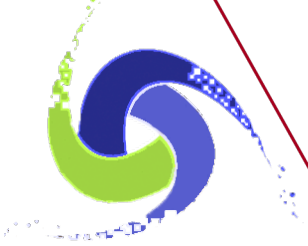
Define scorers to the tracking volume

Class diagram



example…

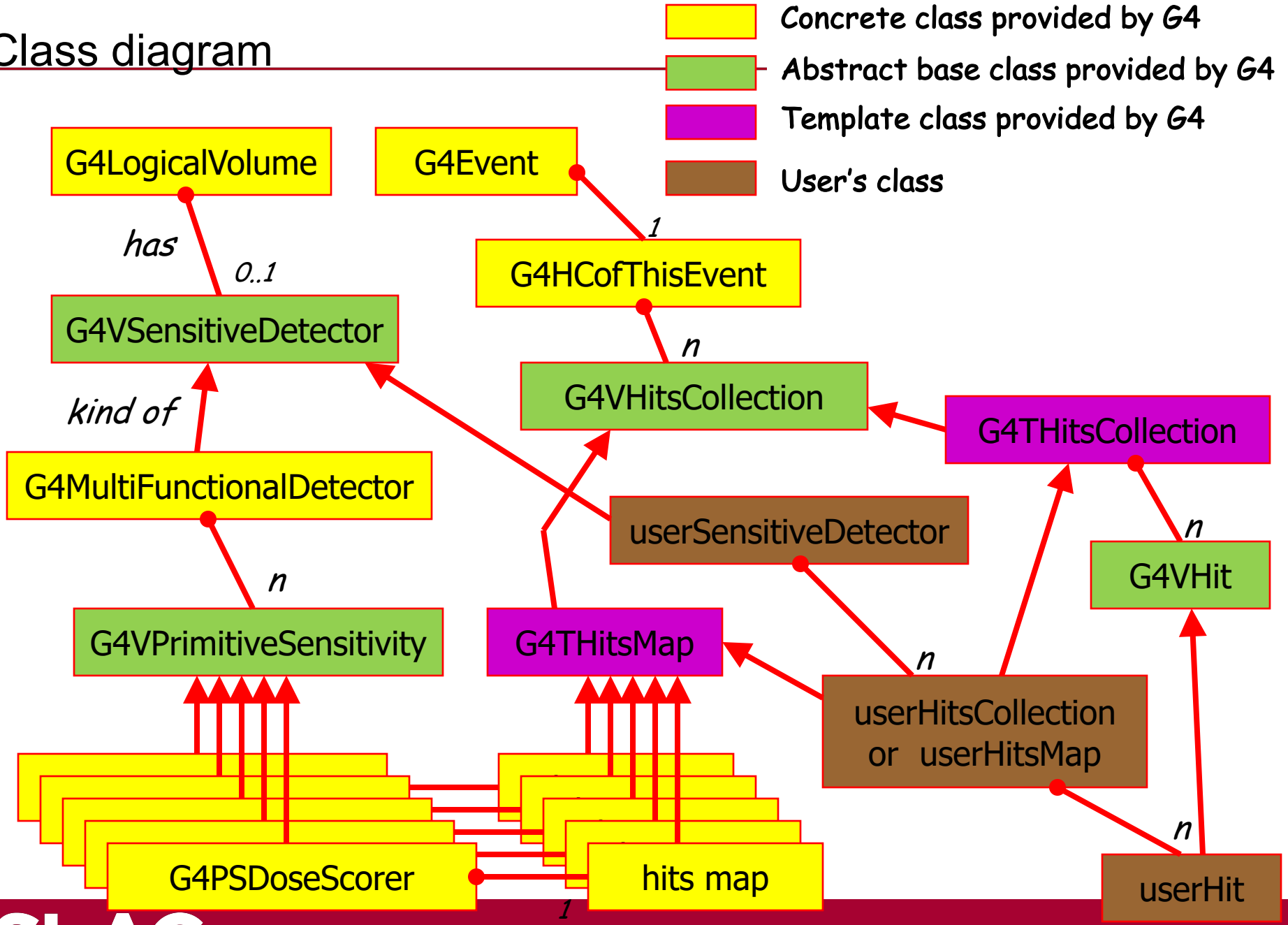
```
MyDetectorConstruction::ConstructSDandField()
{
    G4MultiFunctionalDetector* myScorer = new G4MultiFunctionalDetector("myCellScorer");
    G4VPrimitiveSensitivity* totalSurfFlux = new G4PSFlatSurfaceFlux("TotalSurfFlux");
    myScorer->Register(totalSurfFlux);
    G4VPrimitiveSensitivity* protonSufFlux = new G4PSFlatSurfaceFlux("ProtonSurfFlux");
    G4VSDFilter* protonFilter = new G4SDParticleFilter("protonFilter");
    protonFilter->Add("proton");
    protonSurfFlux->SetFilter(protonFilter);
    myScorer->Register(protonSurfFlux);

    G4SDManager::GetSDMpointer()->AddNewDetector(myScorer);
    SetSensitiveDetector("myLogVol",myScorer);
}
```



GEANT4
A SIMULATION TOOLKIT

Accumulate scores for a run

Class diagram



Score == G4THitsMap<G4double>

- At the end of successful event, G4Event has a vector of G4THitsMap as the scores.
- Create your own Run class derived from G4Run, and implement two methods.
- RecordEvent(const G4Event*) method is invoked in the worker thread at the end of each event. You can get all output of the event so that you can accumulate the sum of an event to a variable for entire run.
- Merge(const G4Run*) method of the run object in the master thread is invoked with the pointer to the thread-local run object when an event loop of that thread is over. You should merge thread-local scores to global scores.
- Your run class object should be instantiated in GenerateRun() method of your *UserRunAction*.
  - This UserRunAction must be instantiated both for master and worker threads.

Customized run class

```
#include “G4Run.hh”
#include “G4Event.hh”
#include “G4THitsMap.hh”
Class MyRun : public G4Run
{
  public:
    MyRun();
    virtual ~MyRun();
    virtual void RecordEvent(const G4Event*);
    virtual void Merge(const G4Run*);
  private:
    G4int nEvent;
    G4int  totalSurfFluxID,  protonSurfFluxID,  totalDoseID;
    G4THitsMap<G4double> totalSurfFlux;
    G4THitsMap<G4double> protonSurfFlux;
    G4THitsMap<G4double> totalDose;
  public:
    … access methods …
};
```

Implement how you accumulate event data

Implement how you merge thread-local scores

Customized run class

```
MyRun::MyRun()
{
  G4SDManager* SDM = G4SDManager::GetSDMpointer();
  totalSurfFluxID = SDM->GetCollectionID("myCellScorer/TotalSurfFlux");
  protonSurfFluxID = SDM->GetCollectionID("myCellScorer/ProtonSurfFlux");
  totalDoseID = SDM->GetCollectionID("myCellScorer/TotalDose");
}
```

name of G4MultiFunctionalDetector object

name of G4VPrimitiveSensitivity object

Customized run class

```
void MyRun::RecordEvent(const G4Event* evt)
{
  G4HCofThisEvent* HCE = evt->GetHCofThisEvent();
  G4THitsMap<G4double>* eventTotalSurfFlux
           = (G4THitsMap<G4double>*)(HCE->GetHC(totalSurfFluxID));
  G4THitsMap<G4double>* eventProtonSurfFlux
           = (G4THitsMap<G4double>*)(HCE->GetHC(protonSurfFluxID));
  G4THitsMap<G4double>* eventTotalDose
           = (G4THitsMap<G4double>*)(HCE->GetHC(totalDoseID));
  totalSurfFlux += *eventTotalSurfFlux;
  protonSurfFlux += *eventProtonSurfFlux;
  totalDose += *eventTotalDose;

  G4Run::RecordEvent(evt);
}
```

No need of loops.
+= operator is provided !

Don't forget to invoke base class method!

Customized run class

```
void MyRun::Merge(const G4Run* run)
{
  const MyRun* localRun = static_cast<const MyRun*>(run);

  totalSurfFlux += *(localRun . totalSurfFlux);
  protonSurfFlux += *(localRun . protonSurfFlux);
  totalDose += *(localRun . totalDose);

  G4Run::Merge(run);
}
```

Cast !

No need of loops.
+= operator is provided !

Don't forget to invoke base class method!

RunAction with customized run

```
G4Run* MyRunAction::GenerateRun()
{ return (new MyRun()); }
void MyRunAction::EndOfRunAction(const G4Run* aRun)
{
  const MyRun* theRun = static_cast<const MyRun*>(aRun);

  if( IsMaster() )
  {
    // … analyze / record / print-out your run summary
    // MyRun object has everything you need …
   }
}
```

IsMaster() returns true for the RunAction object assigned to the master thread. (also returns true for sequential mode)

- As you have seen, to accumulate event data, you do NOT need
  - Event / tracking / stepping action classes
- All you need are your Run and RunAction classes.

Sensitive detector vs. primitive scorer

Sensitive detector

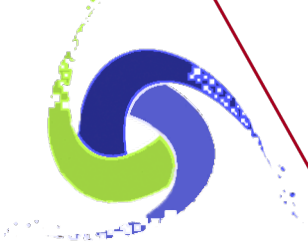
- You have to implement your own detector and hit classes.
- One hit class can contain many quantities. A hit can be made for each individual step, or accumulate quantities.
- Basically one hits collection is made per one detector.
- Hits collection is relatively compact.

Primitive scorer

- Many scorers are provided by Geant4. You can add your own.
- Each scorer accumulates one quantity for an event.
- G4MultiFunctionalDetector creates many collections (maps), i.e. one collection per one scorer.
- Keys of maps are redundant for scorers of same volume.

I would suggest to :

- Use primitive scorers
  - if you are not interested in recording each individual step but accumulating some physics quantities for an event or a run, and
  - if you do not have to have too many scorers.
- Otherwise, consider implementing your own sensitive detector.

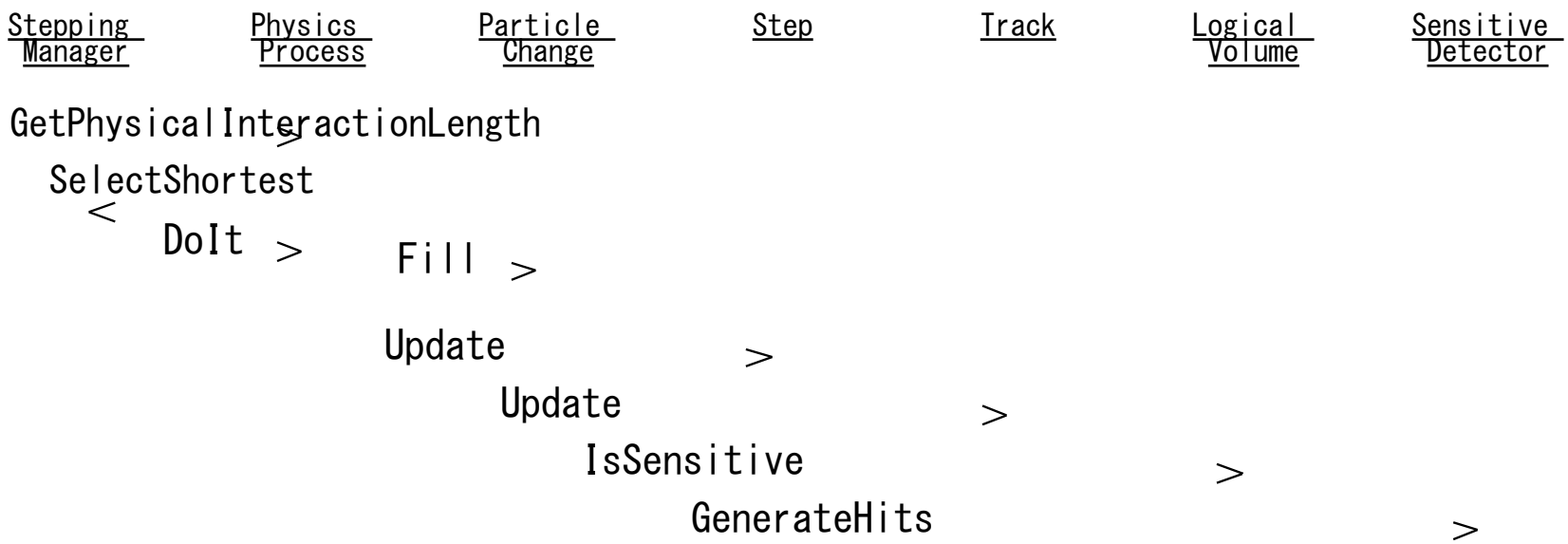


GEANT4
A SIMULATION TOOLKIT

Basic structure of detector sensitivity

Sensitive detector

- A G4VSensitiveDetector object can be assigned to G4LogicalVolume.
- In case a step takes place in a logical volume that has a G4VSensitiveDetector object, this G4VSensitiveDetector is invoked with the current G4Step object.
  - You can implement your own sensitive detector classes, or use scorer classes provided by Geant4.



Defining a sensitive detector

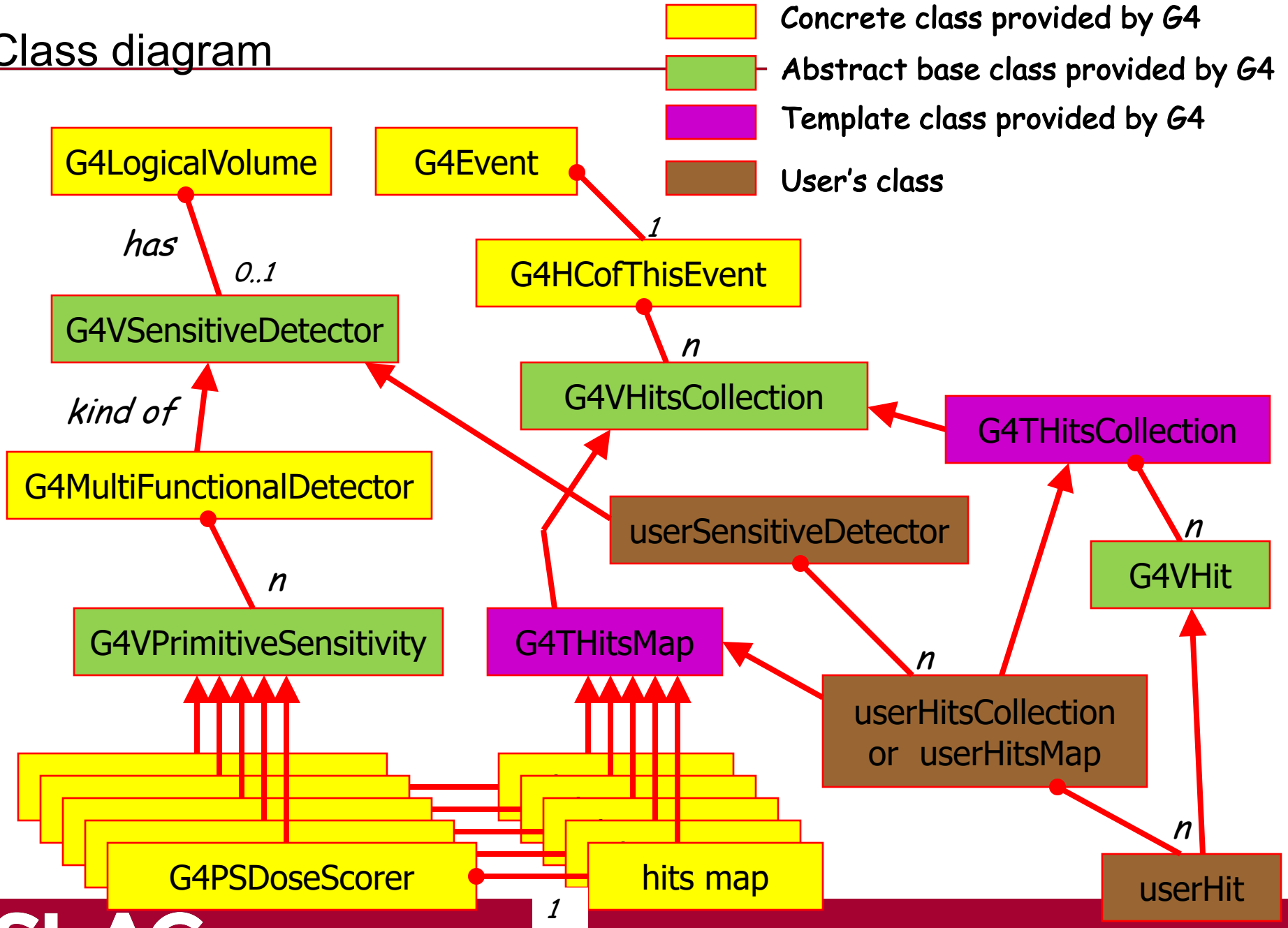
- Basic strategy

  In your ConstructSDandField() method

```
G4VSensetiveDetector* pSensetivePart
  = new MyDetector("/mydet");
G4SDManager::GetSDMpointer()
                ->AddNewDetector(pSensetivePart);
SetSensitiveDetector("myLogicalVolume",pSensetivePart);
```

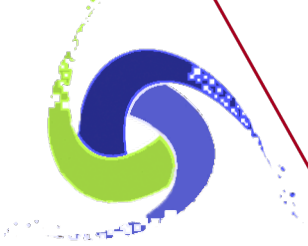
- Each detector object must have a unique name.
  - Some logical volumes can share one detector object.
  - More than one detector objects can be made from one detector class with different detector name.
  - One logical volume cannot have more than one detector objects. But, one detector object can generate more than one kinds of hits.
    - e.g. a double-sided silicon micro-strip detector can generate hits for each side separately.

Class diagram



Hits collection, hits map

- G4VHitsCollection is the common abstract base class of both G4THitsCollection and G4THitsMap.
- G4THitsCollection is a template vector class to store pointers of objects of one concrete hit class type.
  - A hit class (deliverable of G4VHit abstract base class) should have its own identifier (e.g. cell ID).
  - In other words, G4THitsCollection requires you to implement your hit class.
- G4THitsMap is a template map class so that it stores keys (typically cell ID, i.e. copy number of the volume) with pointers of objects of one type.
  - Objects may not be those of hit class.
    - All of currently provided scorer classes use G4THitsMap with simple double.
  - Since G4THitsMap is a template, it can be used by your sensitive detector class to store hits.



GEANT4
A SIMULATION TOOLKIT

Sensitive detector and hit

Sensitive detector and Hit

- Each Logical Volume can have a pointer to a sensitive detector.
  - Then this volume becomes sensitive.
- Hit is a snapshot of the physical interaction of a track or an accumulation of interactions of tracks in the sensitive region of your detector.
- A sensitive detector creates hit(s) using the information given in G4Step object. The user has to provide his/her own implementation of the detector response.
- Hit objects, which are still the user's class objects, are collected in a G4Event object at the end of an event.

Hit class

SLAC

- Hit is a user-defined class derived from G4VHit.
- You can store various types information by implementing your own concrete Hit class. For example:
  - Position and time of the step
  - Momentum and energy of the track
  - Energy deposition of the step
  - Geometrical information
  - or any combination of above
- Hit objects of a concrete hit class must be stored in a dedicated collection which is instantiated from G4THitsCollection template class.
- The collection will be associated to a G4Event object via G4HCofThisEvent.
- Hits collections are accessible
  - through G4Event at the end of event.
    - to be used for analyzing an event
  - through G4SDManager during processing an event.
    - to be used for event filtering.

Implementation of Hit class

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

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

G4Allocator

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

Use of G4Allocator

MyHit.hh

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

MyHit.cc

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

Sensitive Detector class

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

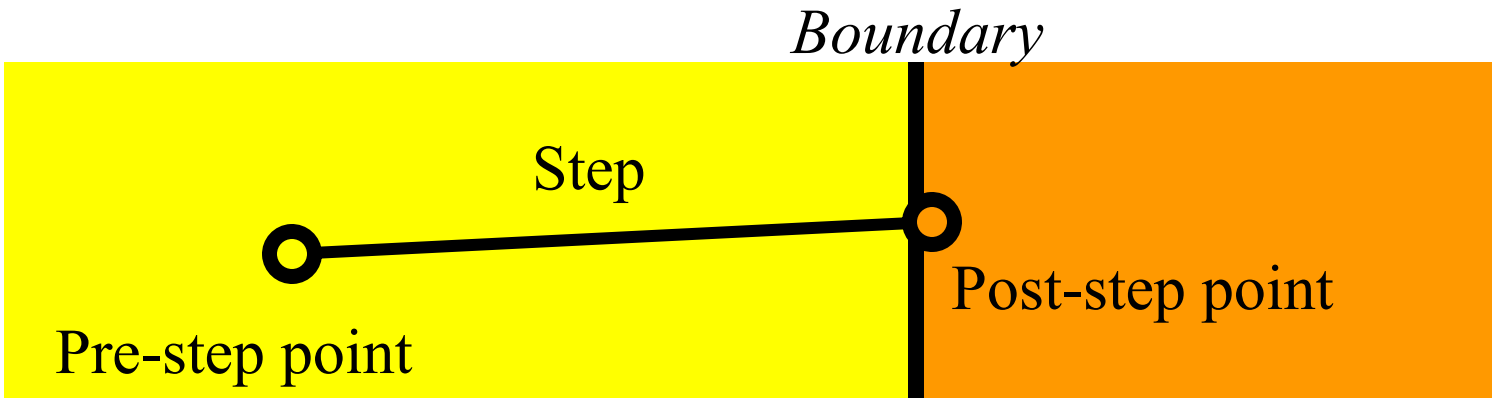
```
#include "G4VSensitiveDetector.hh"
#include "MyHit.hh"
class G4Step;
class G4HCofThisEvent;
class MyDetector : public G4VSensitiveDetector
{
  public:
      MyDetector(G4String name);
      virtual ~MyDetector();
      virtual void Initialize(G4HCofThisEvent*HCE);
      virtual G4bool ProcessHits(G4Step*aStep,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

Sensitive detector

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

Step

- Step has two points and also “delta” information of a particle (energy loss on the step, time-of-flight spent by the step, etc.).
- Each point knows the volume (and material). In case a step is limited by a volume boundary, the end point physically stands on the boundary, and it logically belongs to the next volume.
- Note that you must get the volume information from the “PreStepPoint”.



Implementation of Sensitive Detector - 1

```
MyDetector::MyDetector(G4String detector_name)
                  :G4VSensitiveDetector(detector_name),
                   collectionID(-1)
{
  collectionName.insert("collection_name");
}
```

- In the constructor, define the name of the hits collection which is handled by this sensitive detector
- In case your sensitive detector generates more than one kinds of hits (e.g. anode and cathode hits separately), define all collection names.

Implementation of Sensitive Detector - 2

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

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

Implementation of Sensitive Detector - 3


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

- This ProcessHits() method is invoked for every steps in the volume(s) where this sensitive detector is assigned.
- In this method, generate a hit corresponding to the current step (for tracking detector), or accumulate the energy deposition of the current step to the existing hit object where the current step belongs to (for calorimeter detector).
- Don't forget to collect geometry information (e.g. copy number) from "PreStepPoint".
- Currently, returning boolean value is not used.

Implementation of Sensitive Detector - 4

```
void MyDetector::EndOfEvent(G4HCofThisEvent*HCE)
{;}
```

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