Retrieving information from Geant 4 kernel

Acknowledgements:

A. Lechner,

J. Apostolakis, M. Asai, G. Cosmo, A. Howard

Review of
User Action and Initialisation classes

Mandatory user classes

- **DetectorConstruction**
  - Derived from *G4VUserDetectorConstruction*
  - Geometry and materials of the experimental setup
- **PhysicsList**
  - Derived from *G4VUserPhysicsList*
  - Selection of physics processes/models associated with each particle
  - Secondary production thresholds
- **PrimaryGeneration**
  - Derived from *G4VUserPrimaryGeneratorAction*
  - Generation of primary particles and vertices

Main

- **main()** function of your application
- Create instances of the mandatory classes
- Register the initialization classes, *i.e. the concrete subclasses of G4VUserDetectorConstruction and G4VUserPhysicsList,* to the RunManager
  - using the **SetUserInitialization()** function
- Register the user action class, i.e. the concrete subclass of *G4VUserPrimaryGeneratorAction*, to the RunManager
  - using the **SetUserAction()** function

Example of main function

```
// main()

#include “G4RunManager.hh“
#include “MyDetectorConstruction.hh“
#include “MyPhysicsList.hh“
#include ”MPrimaryGenerator.hh“

main() {
  G4RunManager* runManager = new G4RunManager(); // instantiate G4RunManager

  // mandatory initialization classes
  G4VUserDetectorConstruction* detector = new MyDetectorConstruction();
  runManager ->SetUserInitialization(detector);
  G4VUserPhysicsList* physicsList = new MyPhysicsList();
  runManager ->SetUserInitialization(physicsList);

  // mandatory user action class
  G4VUserPrimaryGeneratorAction* primaryGenerator = new MyPrimaryGenerator();
  runManager ->SetUserAction(primaryGenerator);
...
}
```

Optional UserAction classes

- Define your actions by deriving concrete classes from:
  - **G4UserRunAction**
  - **G4UserEventAction**
  - **G4UserStackingAction**
  - **G4UserTrackingAction**
  - **G4UserSteppingAction**
- As for the mandatory classes, you also should instantiate them in the main() function of your application and notify them to the **RunManager** by using its **SetUserAction()** member function

Optional UserAction classes

- **G4UserRunAction**
  - The **BeginOfRunAction** and **EndOfRunAction** methods are invoked at the beginning and end of a run
  - Can be used e.g. to book or store histograms, ...
- **G4UserEventAction**
  - The **BeginOfEventAction** and **EndOfEventAction** methods are invoked at the beginning and end of an event respectively
  - e.g. one can apply an event selection at the beginning of an event, process information of hits at the end of an event, ...
- **G4UserStackingAction**
  - Classify priorities of tracks
- **G4UserTrackingAction**
  - The **PreUserTrackingAction** and **PostUserTracking** methods can be overloaded
  - Can be used e.g. define trajectories, decide if a trajectory should be stored, ...
- **G4UserSteppingAction**
  - The **UserSteppingAction** method is invoked at the end of an event
  - e.g. you may change the track status in this method, ...

Relevant objects for information retrieval

Run	G4Run
Event	G4Event
Track	G4Track
Step	G4Step
Step point	G4StepPoint

- Bi-directional association between G4Track and G4Step
- G4Step has two G4StepPoint objects
  - Pre-Step Point
  - Post-Step Point

Endpoint of a step

- The G4StepPoint class represents the endpoint of a particle step

It contains (among other things):

- **Geometrical/Material** information
  - **Coordinates** of the particle position
  - **Pointer** to the **physical volume** that contains the position
  - **Pointer** to the **material** associated with this volume
- **Step status**
- **Physics process** information
  - **Pointer** to the **physics process** that defined the length of current and previous step
- And more…
  - Used to update the G4Track object by G4Step, which contains the endpoint

G4Step

- **G4Step** represents a step by which a particle is propagated in the simulation
- A G4Step object stores *transient* information of the step

- In the tracking algorithm, G4Step is updated each time a physics process was invoked
  - for AlongStep, PostStep and AtRest actions of a process
  - You can extract information from a step after the step was completed

- Both the **ProcessHits()** function of your sensitive detector and **UserSteppingAction()** of your step action class receive a pointer to the G4Step object

- Typically, you may retrieve step information in these functions
  - e.g. to fill hit objects in ProcessHits(), ...

Step

A G4Step object contains:

- The **two endpoints**, i.e. the prestep and poststep point
  - Hence, one has access e.g. to the volumes containing the step endpoints
- **Changes in particle properties** between the points
  - E.g. difference of particle energy and momentum, ...
- More **step-related information** like
  - Energy deposition on step, step length, time of flight, ...
- A **pointer** to the associated **G4Track** object

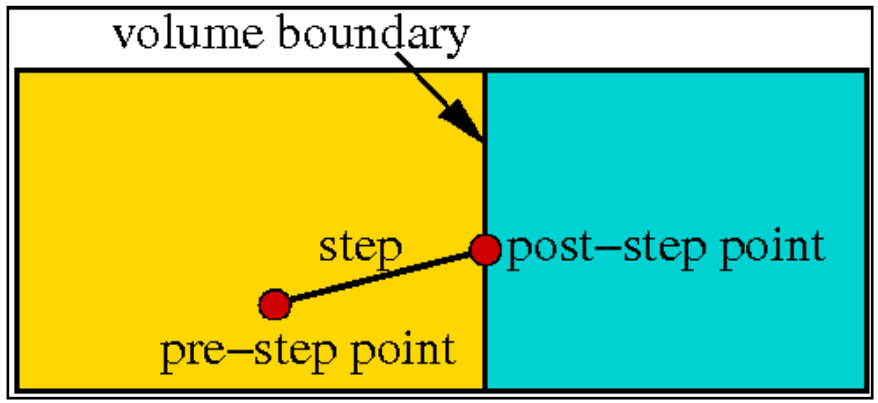
- G4Step provides various methods to access information
  - e.g. G4StepPoint* GetPreStepPoint()
  - G4double GetStepLength()
  - etc.

Example: Step information in SensitiveDetector

```
// in the implementation of your sensitive detector class:

MySensitiveDetector::ProcessHits(G4Step* step,G4TouchableHistory*) {

  // Total energy deposition in the step (= energy deposited by energy loss
  // process and energy of secondaries that were not created, since their
  // energy was < Cut):
  G4double energyDeposit = step ->GetTotalEnergyDeposit();

  // Difference of energy, position and momentum of particle between pre- and
  // post-step point
 G4double deltaEnergy = step ->GetDeltaEnergy();
 G4ThreeVector deltaPosition = step ->GetDeltaPosition();
 G4double deltaMomentum = step ->GetDeltaMomentum();

  // Step length
  G4double stepLength = step ->GetStepLength();
}
```

StepPoints and geometrical boundaries

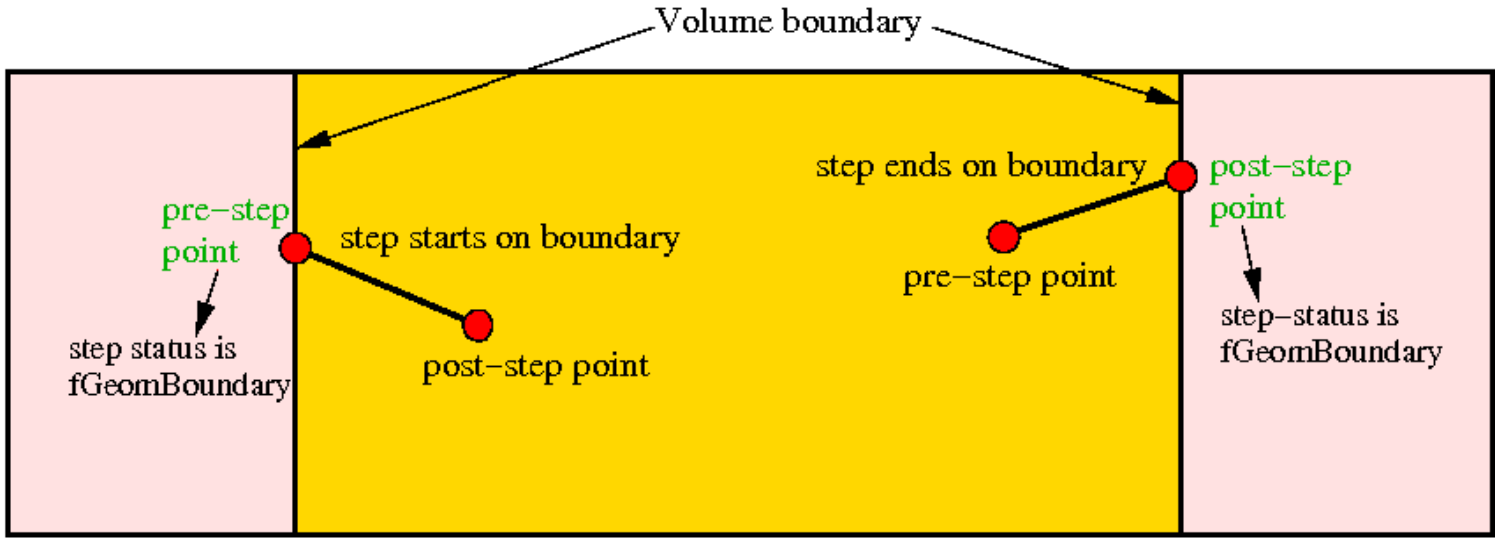
- If a step is limited by a boundary, the post-step point is physically *on* the boundary
- Note: the post-step point is then considered to be in the **next** volume
- This implies that the **post-step point** contains volume and material information of the **next** volume
- Together with the content of the pre-step point object, this allows one to keep track of boundary effects



StepPoints and geometrical boundaries

- To check, if a step **ends** on a boundary, one may compare if the physical volumes of pre- and post-step points are equal, or one makes use of the **step status**
  - The **step status** provides information about the process that restricted the step length (see Appl. Developers Manual for details)
  - It is attached to step points: the pre-step point has the status of the *previous* step, and the post-step point of the *current* step
  - *If the status of the post-step point is **"fGeomBoundary"**, 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**
  - *If the status of the PREstep point is **"fGeomBoundary"**, the step starts on a volume boundary (does not apply to world volume)*

Steps starting or ending on boundaries



Example of using StepPoint

```
// in the implementation of your user step action class
#include “G4Step.hh”

MySteppingAction::UserSteppingAction(const G4Step* step) {
  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) {
     std::cout << “Step starts on geometry boundary“ << std::endl;
     }
  if (postStepPoint ->GetStepStatus() == fGeomBoundary) {
     std::cout << “Step ends on geometry boundary“ << std::endl;

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

Track

- A Geant4 track, represented by **G4Track**, is a “snapshot” of the status of a particle **after a step was completed**
  - i.e. it has information corresponding the post-step point of the step
  - e.g. kinetic energy of particle, momentum direction, time since event and track was created, *track status (see later)*,...
  - It also holds a pointer to a DynamicParticle object
  - It does not record information of previous steps
  - *It is NOT a collection of G4Step objects!*
- It has also some information that is not subject to change during stepping
  - Track ID, information about primary vertex,...
  - Primaries have track ID=1, secondaries have larger track ID

G4Trajectory

- G4Track and G4Step have **no memory** of previous steps, and no G4Track object is available at the end of an event
- However, if you activate the use of **G4Trajectory** objects, some track information becomes available
  - G4Trajectory objects will be available in G4Event at the end of an event
- G4Trajectory has a collection of **G4TrajectorPoint** objects
- ***G4Trajectory*** *stores specific information of* ***G4Tracks***
- ***G4TrajectoryPoints*** *store specific information of* ***G4Steps***
- Do not store many trajectories
  - it consumes lots of memory!

Track deletion

A track object is deleted if:

- the particle **leaves the world volume**
- it **disappears due to a physical process** (e.g. decay)
  - Note that in some hadronic interactions, the particle “looses” its identity: It is treated as a secondary particle due to fact that the interaction partners cannot be distinguished; in this case the primary track is deleted
- its **kinetic energy falls to 0** (and no “AtRest” action is required)
- the **user kills** the track

Track status

- After each step the track can change its state
- The status can be one of the following:
  - states in yellow can only be set by the user, but will not be set by the kernel

Track Status	Description
fAlive	The particle continues to be tracked
fStopButAlive	Kinetic 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)

Retrieving information from tracks

```
// retrieving information from tracks (given the G4Track object "track")

if (track ->GetTrackID() != 1) {
  std::cout << "Particle is a secondary" << std::endl;

   // Note in this context that primary hadrons might loose their identity
  if (track -> GetParentID() == 1)
              std::cout << "But parent was a primary" << std::endl;

  G4VProcess* creatorProcess = track ->GetCreatorProcess();
  if (creatorProcess ->GetProcessName() == "LowEnergyIoni") {
     std::cout << "Particle was created by the LowEnergy ionization process"
               << std::endl;
     }
  }
```

Example of UserTrackingAction

```
// in the implementation of your user tracking action class

#include “G4TrackingManager.hh“
#include “G4Electron.hh“

UserTrackingAction::PostUserTrackingAction(const G4Track* track) {
// The user tracking action class holds the pointer to the tracking manager,
// fpTrackingManager
// From the tracking manager we can retrieve the secondary track vector,
// which is a container class for tracks
G4TrackVector* secondaryTracks = fpTrackingManager ->GimmeSecondaries();

// You can use the secondaryTracks vector to retrieve the number of secondaries, //
the initial kinetic energies, the particle type, ...
if (secondaryTracks) { ... }

// Note: The G4TrackVector is defined as:
// typedef std::vector<G4Track*> G4TrackVector;
// hence has all the functionality of the STL vector
}
```

Particle classes in Geant4

Class	What does it provide?
G4Track	Information relevant to tracking the particle, e.g. position, time, step etc.
G4DynamicParticle	Dynamic information, e.g. particle momentum, kinetic energy etc.
G4ParticleDefinition	Static information, e.g. particle mass, charge etc. Also relation to physics processes associated with the particle

G4Track holds a pointer to a **G4DynamicParticle** object
G4DynamicParticle holds a pointer to a **G4ParticleDefinition** object

Particles

- More than 100 types of particles are defined in Geant4
- Particles are categorized into: leptons, mesons, baryons, bosons, short-lived and ions

- Most particles are represented by their own class, that derives from **G4ParticleDefinition** (except ions)
  - e.g. G4Electron, G4Neutron, G4KaonPlus, ...
- For each particle class, only a **single static instance** may exist
  - All G4DynamicParticle objects have a pointer to the same particle definition class, if they deal with the same type of particle
  - The unique class instances are created (by the user) in the "initialization phase" (as a part of the Physics List setup)

- Once created, the user can retrieve information from the particle class instances using a range of accessors (Get methods)

Using ParticleDefinition

- The G4ParticleDefinition class provides
  - **Static information**: name, mass, charge, spin, lifetime etc.
  - **Pointer to G4ProcessManager**, which holds a list of physics processes associated with the particle

- The pointer to an instance of a particular particle class can be obtained through the static **Definition()** method
  - e.g. G4ParticleDefinition* electron = G4Electron::Definition();
- Alternatively, the **G4ParticleTable** class (singleton) provides methods to find particles according to a specific attribute
  - e.g. by particle name
  - G4ParticleDefinition* electron =
    G4ParticleTable::GetParticleTable() ->FindParticle("e");

Retrieving static particle information

```
#include “G4ParticleDefinition.hh”
#include “G4ParticleTable.hh”

G4ParticleDefinition* proton = G4Proton::Definition();

double protonPDGMass =          proton ->GetPDGMass();
double protonPDGCharge =        proton ->GetPDGCharge();
int protonPDGNumber =           proton ->GetPDGEncoding();
G4String protonPartType =       proton ->GetParticleType(); // “baryon”
G4String protonPartSubType =    proton ->GetParticleSubType(); // “nucleon”
int protonBaryonNumber =        proton ->GetBaryonNumber();

G4ParticleTable* particleTable = G4ParticleTable::GetParticleTable();
G4ParticleDefinition* pionPlus = particleTable ->FindParticle(“pi+”);

bool particleIsStable =         pionPlus ->GetPDGStable();
double pionPlusLifeTime =       pionPlus ->GetPDGLifeTime();
double pionPlusIsospin =        pionPlus ->GetPDGIsospin();
```

DynamicParticle

- A **G4DynamicalParticle** object represents an *individual* particle
  - whereas G4ParticleDefinition represents a particle type
- Each **G4Track** object has an unique instance of **G4DynamicParticle**, that exists as long as the track is not deleted
- A **G4DynamicParticle** object is responsible for
  - Dynamic information, i.e. physical properties that may change from step to step: kinetic energy, spin, polarisation, charge (ions), ...
  - It holds a **pointer to a ParticleDefinition** object

Access to information of a DynamicParticle

- *Once all PostStepDoIt() methods have been invoked (for a given step), the G4DynamicParticle instance is updated by the track to hold the particle properties resulting from the physics processes of the step*
- Various Get methods are defined in G4DynamicParticle to allow the retrieval of dynamic information
- Typically, you may want to retrieve dynamic information in the **ProcessHits()** function of your **SensitiveDetector**, or in **UserSteppingAction**() of your **SteppingAction** class
- The **GetDynamicParticle()** method of **G4Track** returns a pointer to the associated instance of G4DynamicParticle
- Use the **GetDefinition()** method of **G4DynamicParticle** to obtain the pointer to the G4ParticleDefinition object
- Proceed as previously shown to retrieve static information

Example with DynamicParticle

```
#include “G4ParticleDefinition.hh”
#include “G4DynamicParticle.hh”
#include “G4Step.hh”
#include “G4Track.hh”

// Retrieve from the current step the track (after PostStepDoIt of step is completed)
G4Track* track = step ->GetTrack();

// From the track you can obtain the pointer to the dynamic particle
const G4DynamicParticle* dynamicParticle = track ->GetDynamicParticle();

// From the dynamic particle, retrieve the particle definition
G4ParticleDefinition* particle = dynamicParticle ->GetDefinition();

// The dynamic particle class contains e.g. the kinetic energy after the step
double kinEnergy = dynamicParticle ->GetKineticEnergy();

// From the particle definition class you can retrieve static information like
// the particle name
G4String particleName = particle ->GetParticleName();
std::cout  << particleName << “: kinetic energy is ”
           << kinEnergy/MeV << “ MeV”
           << std::endl;
```