GEANT4
A SIMULATION TOOLKIT

User Actions
aka ‘Extracting Useful Information’

Mihaly Novak (CERN)

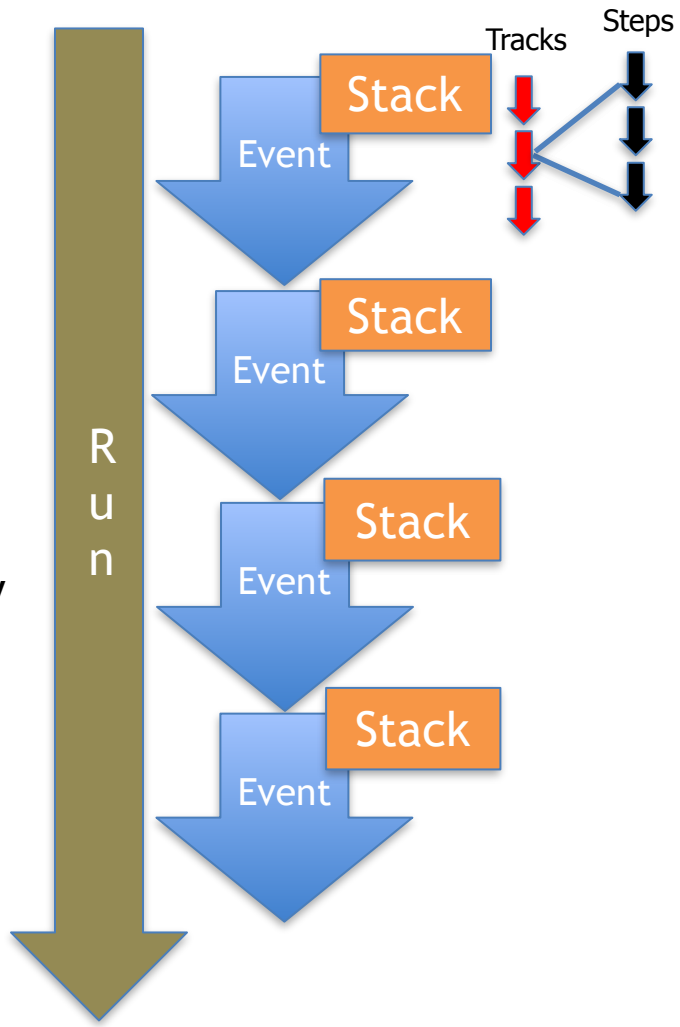
Geant4 Beginners Course

25-31 May 2021 CERN

Some of this material was prepared by Makoto Asai (SLAC).

User Actions - Overview

- mandatory Users actions classes
  - G4VUserActionInitialization
  - G4VUserPrimaryGeneratorAction
- optional Geant4 User Action classes
  - G4UserRunAction
  - G4UserEventAction
  - G4UserStackingAction (not included today)
  - G4UserTrackingAction
  - G4UserSteppingAction
- fully customizable (empty by default)
- user action classes are used to setup and/or modify the simulation or collect information about the run
  - allow to take actions specific for the given simulation
    - simulate only relevant particles
    - save specific information, fill histograms
    - speed-up simulation by applying different limits
- G4Run also plays an important role



G4VUserActionInitialization

G4VUserActionInitialization (1/2)

- **virtual void G4VUserActionInitialization::Build() const = 0;**
  - **Pure virtual** method(mandatory) to be implemented by the user to instantiate **all** (mandatory and optional) **user action** objects
  - **At least**, the mandatory UserPrimaryGeneratorAction needs to be instantiated here
  - This method is *invoked by **each worker***
- **virtual void G4VUserActionInitialization::BuildForMaster()**
  - This method is *invoked only by the **master***
  - Note, that it's not pure virtual (invoked only in MT)
  - For instantiating **only UserRunAction**
  - This will be the ***master's*** UserRunAction
  - This UserRunAction might or might not be the same as for *workers* (its IsMaster() method will return true)

G4VUserActionInitialization (2/2)

- G4VUserActionInitialization base class provides the following methods to ***set*** user actions, that should be used inside the Build() interface method to set the user actions after their construction:
  - **void SetUserAction(G4VUserPrimaryGeneratorAction*) const;**
  - **void SetUserAction(G4UserRunAction*) const;**
  - **void SetUserAction(G4UserEventAction*) const;**
  - **void SetUserAction(G4UserStackingAction*) const;**
  - **void SetUserAction(G4UserTrackingAction*) const;**
  - **void SetUserAction(G4UserSteppingAction*) const;**

```
// class YourActionInitialization : public G4VUserActionInitialization
void YourActionInitialization::Build() const
{
  // Set UserPrimaryGeneratorAction
  YourPrimaryGeneratorAction* primaryAction = new YourPrimaryGeneratorAction(fYourDetector);
  SetUserAction(primaryAction);
  // Set UserRunAction
  YourRunAction* runAction = new YourRunAction(fYourDetector, primaryAction);
  SetUserAction(runAction);
  // Set UserEventAction
  YourEventAction* eventAction = new YourEventAction();
  SetUserAction(eventAction);
  // Set UserSteppingAction
  SetUserAction( new YourSteppingAction(fYourDetector, eventAction) );
}
```

G4VUserActionInitialization (2/2)

-

```
// class YourActionInitialization : public G4VUserActionInitialization
void YourActionInitialization::BuildForMaster() const
{
   // we will use the same for the Master as for the Workers (IsMaster() can be
   // used to see if a given RunAction object is the one that belongs to the
   // Master thread or not: see for example YourRunAction::EndOfRunAction
   // method for an example of doing something only for the Master RunAction
   // object but not for the Worker RunAction objects)
   //
   // the primary-generator not used in the master RunAction only for the workers
   SetUserAction(new YourRunAction(fYourDetector));
}
```

```
// class YourActionInitialization : public G4VUserActionInitialization
void YourActionInitialization::Build() const
{
  // Set UserPrimaryGeneratorAction
  YourPrimaryGeneratorAction* primaryAction = new YourPrimaryGeneratorAction(fYourDetector);
  SetUserAction(primaryAction);
  // Set UserRunAction
  YourRunAction* runAction = new YourRunAction(fYourDetector, primaryAction);
  SetUserAction(runAction);
  // Set UserEventAction
  YourEventAction* eventAction = new YourEventAction();
  SetUserAction(eventAction);
  // Set UserSteppingAction
  SetUserAction( new YourSteppingAction(fYourDetector, eventAction) );
}
```

G4UserRunAction and G4Run

G4UserRunAction (1/2)

- **virtual G4Run* G4UserRunAction::GenerateRun()**
  - This method is invoked at the beginning of BeamOn.
  - User hook to provide derived G4Run and create his/her own concrete class to store some information about the run
  - It is invoked before the calculation of the physics table

G4UserRunAction (1/2) - G4Run

- **virtual G4Run* G4UserRunAction::GenerateRun()**
  - This method is invoked at the beginning of BeamOn.
  - User hook to provide derived G4Run and create his/her own concrete class to store some information about the run
  - It is invoked before the calculation of the physics table.

- But what is this **G4Run**? Or more exactly, *YourRun* derived from G4Run
  - Think about it as a (thread local) data with a merge functionality
  - An instance of *YourRun* is automatically generated for each thread (both workers and master) by calling the above GenerateRun() method of *YourRunAcion* derived from G4UserRunAction

```
34  // class YourRunAction : public G4UserRunAction
35  G4Run* YourRunAction::GenerateRun()
36  {
37    // class YourRun : public G4Run
38    fYourRun = new YourRun(fYourDetector, fYourPrimary);
39    return fYourRun;
40  }
```

G4UserRunAction (1/2) - G4Run

- **virtual G4Run* G4UserRunAction::GenerateRun()**
  - This method is invoked at the beginning of BeamOn.
  - User hook to provide derived G4Run and create his/her own concrete class to store some information about the run
  - It is invoked before the calculation of the physics table.
- But what is this **G4Run**? Or more exactly, *YourRun* derived from G4Run
  - Think about it as a (thread local) data with a merge functionality
  - An instance of *YourRun* is automatically generated for each thread (both workers and master) by calling the above GenerateRun() method of *YourRunAcion* derived from G4UserRunAction
  - The base **virtual void G4Run::Merge(const G4Run*)** method
    - needs to be implemented by your derived *YourRun*
    - how to merge your local (*worker*) YourRun data to the global (*master*) YourRun data
    - it is invoked by the end of the run to merge local data collected by the individual workers during the run

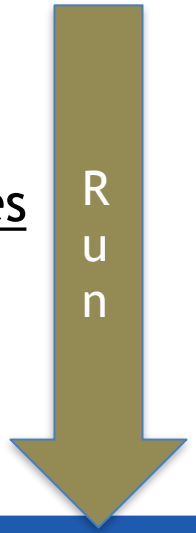
G4UserRunAction (1/2) - G4Run

- **virtual G4Run* G4UserRunAction::GenerateRun()**
  - This method is invoked at the beginning of BeamOn.
  - User hook to provide derived G4Run and create his/her own concrete class to store some information about the run
  - It is invoked before the calculation of the physics table.
- **Does it make G4VUserActionInitialization::Build() and BuildForMaster() clearer?**
  - The base **virtual void G4Run::Merge(const G4Run*)** method
    - needs to be implemented by your derived *YourRun*
    - how to merge your local (*worker*) YourRun data to the global (*master*) YourRun data
    - it is invoked by the end of the run to merge local data collected by the individual workers during the run

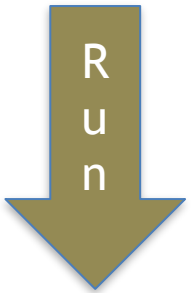
G4UserRunAction (1/2)

- **virtual G4Run* G4UserRunAction::GenerateRun()**
  - This method is invoked at the beginning of BeamOn.
  - User hook to provide derived G4Run and create his/her own concrete class to store some information about the run
  - It is invoked before the calculation of the physics table.

- **virtual void G4UserRunAction::BeginOfRunAction(const G4Run*)**
  - Invoked before entering the event loop
  - Typical use of this method would be to initialize and/or book histograms for a particular run
  - This method is invoked after initialisation of the physics tables
  - Note, you will access here your (thread local) *YourRun* object constructed when the above GenerateRun() method was invoked!



G4UserRunAction (2/2)



- **virtual void G4UserRunAction::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 and writing the results.
- **G4bool G4UserRunAction::IsMaster()** is a useful base class method
  - Commonly, a MT simulation will have several *YourRunAction* instances:
    - a single *master-thread* instance that is constructed in the G4VUserActionInitialization::BuildForMaster() method
    - and several *worker-thread* instances that are constructed in the G4VUserActionInitialization::Build() method
  - provides the ability to identify the single *master-thread* instance

G4UserRunAction (2/2)

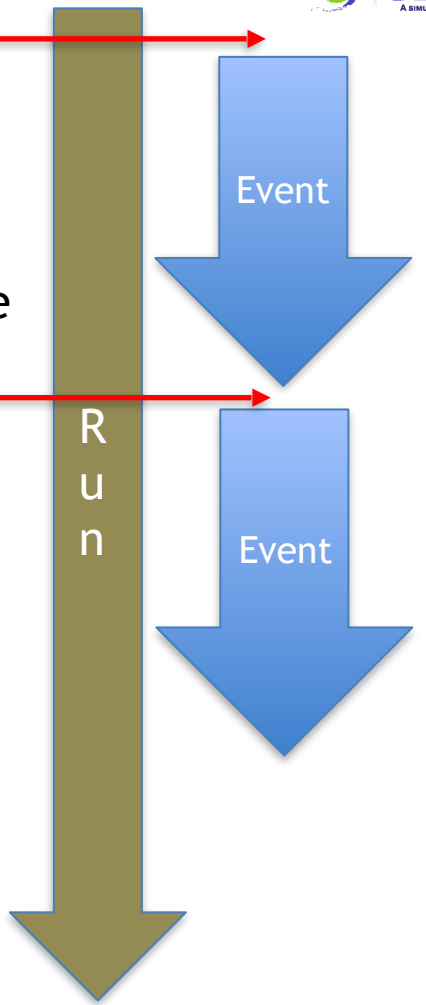
```
70  void YourRunAction::EndOfRunAction(const G4Run*)
71  {
72      // The worker-thread local `YourRun`-s are already merged to the global
73      // master's `YourRun` object at the end of the run. So write out the
74      // complete run summary: this is done only for the master-thread run action!
75      if ( IsMaster() )
76      {
77        fYourRun->EndOfRunSummary();
78      }
79  }
```

- G4bool **G4UserRunAction::IsMaster()** is a useful base class method
  - Commonly, a MT simulation will have several *YourRunAction* instances:
    - a single *master-thread* instance that is constructed in the G4VUserActionInitialization::BuildForMaster() method
    - and several *worker-thread* instances that are constructed in the G4VUserActionInitialization::Build() method
  - provides the ability to identify the single *master-thread* instance

G4UserEventAction

G4UserEventAction

- **virtual void BeginOfEventAction(const G4Event*)**
  - This method is invoked before converting the primary particles to G4Track objects, i.e. before a new event processing
  - A typical use of this method would be to (re)-initialize and/or book histograms for a particular event
- **virtual void EndOfEventAction(const G4Event*)**
  - This method is invoked at the very end of the event processing
  - Typically used for a simple analysis of the processed event or to fill/propagate the event related information/data (collected during the event processing) to the Run (*YourRun*)



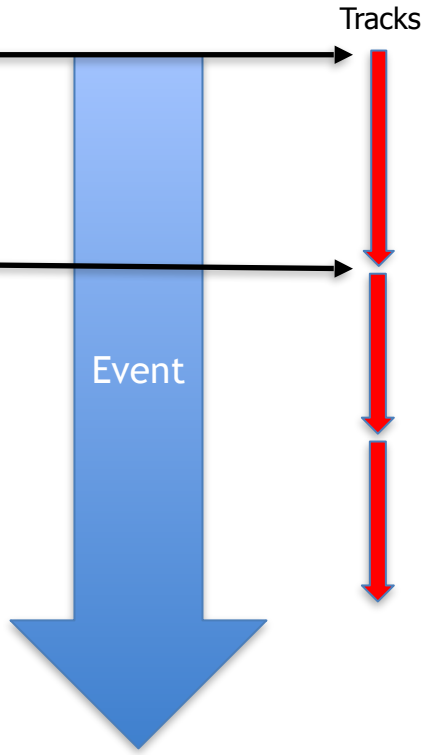
G4UserTrackingAction and `G4Track`

G4Track

- **`G4Track`:**
  - a **`G4Track`** object represents/describes **the state of a particle** that is under simulation in a given instant of the time (i.e. **a** given **time point**)
  - a snapshot of a particle **without** keeping any **information regarding the past**
  - its **`G4ParticleDefinition`** stores **static** particle **properties** (charge, mass, etc.) as it describes a particle type (e.g. **`G4Electron`**)
  - its **`G4DynamicParticle`** stores **dynamic** particle **properties** (energy, momentum, etc.)
  - while all **G4Track**-s, describing the same particle type, share the same, unique **`G4ParticleDefinition`** object of the given type (e.g. **`G4Electron`**) while each individual track has its own **`G4DynamicParticle`** object
  - the **`G4Track`** object is propagated in a step-by-step way during the simulation and the dynamic properties are continuously updated to reflect the current state
  - manager: **`G4TrackingManager`**; optional user hook: **`G4UserTrackingAction`**
  - step-by-step? what about the difference between two such states within a step?

G4UserTrackingAction

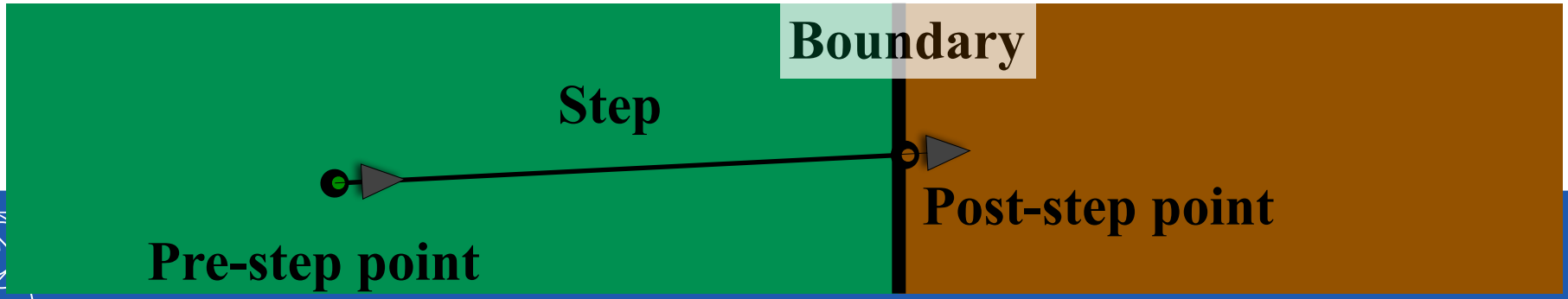
- Provides user hooks to access a particle track at the beginning and end of the corresponding particle's lifetime
- **virtual void BeginOfTrackingAction(const G4Track*)**
  - Invoked at the beginning of a particle lifetime (before start tracking)
- **virtual void EndOfTrackingAction(const G4Track*)**
  - Invoked at the end of a particle lifetime (at the end of tracking) that can be due to:
    - leaves the outermost (World) volume i.e. goes out of the simulation universe
    - participates in a destructive interaction (e.g. decay or photoelectric absorption, ect.)
    - its kinetic energy becomes zero and doesn't have ("at-rest") interaction that could happen
    - the user decided to (artificially) stop tracking this particle and kill (e.g. in the User Stepping Action)



G4UserSteppingAction and `G4Step`

G4Step

- **`G4Step`:**
  - a **`G4Step`** object can provide the information regarding the **change in** the **state of the particle (**that is under tracking) **within a** simulation **step** (i.e. **delta)**
  - has two **`G4StepPoint`**-s, pre- and post-step points, that stores information (position, direction, energy, material, volume, etc…) that belong to the corresponding point (space/time/step)
  - these are updated in a step-by-step way: the post-step point of the previous step becomes the pre-step point of the next step (when the next step starts)
  - **(important)** if a step is limited by the geometry (i.e. by a volume boundary), the post-step point:
    - **physically** stands **on the boundary** (the step status of the post step point i.e. `G4Step::GetPostStepPoint()->GetStepStatus()` is `fGeomBoundary`)
    - **logically** belongs **to the next volume**
    - since these "*boundary*" **`G4Step`**-s have information both regarding the previous and the next volumes/materials, boundary processes (e.g. reflection, refractions and transition radiation) can be simulated

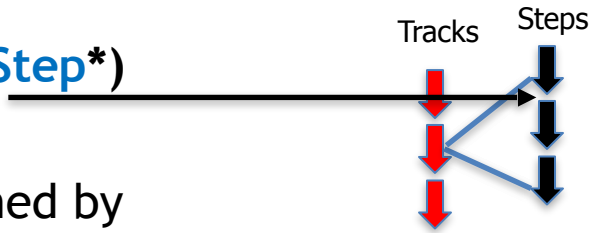


G4Step

- **`G4Step`:**
  - a **`G4Step`** object can provide the information regarding the **change in** the **state of the particle (**that is under tracking) **within a** simulation **step** (i.e. **delta)**
  - has two **`G4StepPoint`**-s, pre- and post-step points, that stores information (position, direction, energy, material, volume, etc…) that belong to the corresponding point (space/time/step)
  - these are updated in a step-by-step way: the post-step point of the previous step becomes the pre-step point of the next step (when the next step starts)
  - **(important)** if a step is limited by the geometry (i.e. by a volume boundary), the post-step point:
    - **physically** stands **on the boundary** (the step status of the post step point i.e. `G4Step::GetPostStepPoint()->GetStepStatus()` is `fGeomBoundary`)
    - **logically** belongs **to the next volume**
    - since these "*boundary*" **`G4Step`**-s have information both regarding the previous and the next volumes/materials, boundary processes (e.g. reflection, refractions and transition radiation) can be simulated
  - the **`G4Track`** object, that is under tracking i.e. generates information for the **`G4Step`** object, can be obtained from the step by the **`G4Step::GetTrack()`** method and the other way around **`G4Track::GetStep()`**
  - manager: **`G4SteppingManager`**; optional user hook: **`G4UserSteppingAction`**

G4UserSteppingAction

- Provides user hook to a particle step
- **virtual void UserSteppingAction(const G4Step*)**
  - Invoked after each simulation step
  - A step (its post-step point) can be defined by
    - A physical process (e.g. ionization, decay)
    - Transportation step: volume boundary
- Used for accessing any information regarding the given step
- The most frequently called user hook: **think about computing performance whatever you do here** (e.g. avoid things like string comparisons, etc.)!



G4UserSteppingAction

How to get information regarding the simulation when the `G4step* theStep` is given?

```
// get the pre-step point
G4StepPoint*         preStp = theStep->GetPreStepPoint();
// get the volume which the step was done
G4VPhysicalVolume* physVol = preStp->GetPhysicalVolume();
// get the energy deposit and length of the step
G4double            stpEdep = theStep->GetTotalEnergyDeposit();
G4double          stpLength = theStep->GetStepLength();
// get the track
G4Track*           theTrack = theStep->GetTrack();
// get the static and dynamic particle properties
const G4ParticleDefinition* partDef =  theTrack->GetParticleDefinition();
const G4DynamicParticle*    partDyn =  theTrack->GetDynamicParticle();
// get the charge of the particle that is under tracking in this step
G4double         partCharge = partDef->GetPDGCharge();
// get the post step point kinetic energy
G4double       postStpEkin = theStep->GetPostStepPoint()->GetKineticEnergy();
// or since the track is already updated to reflect the post-step point
// G4double       postStpEkin = partDyn->GetKineticEnergy();
// which is different in case of the pre-step point kinetic energy that can be
// obtained only from the pre-step point as
G4double         preStpEkin = preStp->GetKineticEnergy();
```

Recall again an example: the `Final` application in the git repo

G4VUserActionInitialization

```
// class YourActionInitialization : public G4VUserActionInitialization
void YourActionInitialization::BuildForMaster() const
{
   // we will use the same for the Master as for the Workers (IsMaster() can be
   // used to see if a given RunAction object is the one that belongs to the
   // Master thread or not: see for example YourRunAction::EndOfRunAction
   // method for an example of doing something only for the Master RunAction
   // object but not for the Worker RunAction objects)
   //
   // the primary-generator not used in the master RunAction only for the workers
   SetUserAction(new YourRunAction(fYourDetector));
}
```

```
// class YourActionInitialization : public G4VUserActionInitialization
void YourActionInitialization::Build() const
{
  // Set UserPrimaryGeneratorAction
  YourPrimaryGeneratorAction* primaryAction = new YourPrimaryGeneratorAction(fYourDetector);
  SetUserAction(primaryAction);
  // Set UserRunAction
  YourRunAction* runAction = new YourRunAction(fYourDetector, primaryAction);
  SetUserAction(runAction);
  // Set UserEventAction
  YourEventAction* eventAction = new YourEventAction();
  SetUserAction(eventAction);
  // Set UserSteppingAction
  SetUserAction( new YourSteppingAction(fYourDetector, eventAction) );
}
```

G4UserRunAction

```
34  // class YourRunAction : public G4UserRunAction
35  G4Run* YourRunAction::GenerateRun()
36  {
37    // class YourRun : public G4Run
38    fYourRun = new YourRun(fYourDetector, fYourPrimary);
39    return fYourRun;
40  }
```

```
3  void YourRunAction::BeginOfRunAction(const G4Run* /*run*/)
4  {
5    if ( fIsEdepHistogramUICmdInvoked )
6    {
7      // user defined the properties of the Edep-histo by invoking the UI comman
8      fYourRun->SetEdepHisto(fEdepHistFileName, fEdepHistMinEnergy,
9                             fEdepHistMaxEnergy, fEdepHistNumBins);
0    }
1  }
```

```
70  void YourRunAction::EndOfRunAction(const G4Run*)
71  {
72      // The worker-thread local `YourRun`-s are already merged to the global
73      // master's `YourRun` object at the end of the run. So write out the
74      // complete run summary: this is done only for the master-thread run action!
75      if ( IsMaster() )
76      {
77        fYourRun->EndOfRunSummary();
78      }
```

G4UserEventAction

```
// Before each event: reset per-event variables
void YourEventAction::BeginOfEventAction(const G4Event* /*anEvent*/)
{
  fEdepPerEvt           = 0.0;
  fChTrackLengthPerEvt  = 0.0;
}
```

```
// After each event:
// fill the data collected for this event into the Run global (thread local)
// data Run data object (i.e. into YourRun)
void YourEventAction::EndOfEventAction(const G4Event* /*anEvent*/)
{
    // get the current Run object and cast it to YourRun (because for sure this is its type)
    YourRun* currentRun = static_cast< YourRun* > ( G4RunManager::GetRunManager()->GetNonConstCurrentRun() );
    // add the quantities to the (thread local) run global YourRun object
    currentRun->AddEnergyDepositPerEvent( fEdepPerEvt );
    currentRun->AddChTrackLengthPerEvent( fChTrackLengthPerEvt );
    currentRun->FillEdepHistogram( fEdepPerEvt );
}
```

G4UserEventAction

```
// Before each event: reset per-event variables
void YourEventAction::BeginOfEventAction(const G4Event* /*anEvent*/)
{
  fEdepPerEvt           = 0.0;
  fChTrackLengthPerEvt  = 0.0;
}
```

G4SteppingAction::UserSteppingAction() populates per event data !

```
// After each event:
// fill the data collected for this event into the Run global (thread local)
// data Run data object (i.e. into YourRun)
void YourEventAction::EndOfEventAction(const G4Event* /*anEvent*/)
{
    // get the current Run object and cast it to YourRun (because for sure this is its type)
    YourRun* currentRun = static_cast< YourRun* > ( G4RunManager::GetRunManager()->GetNonConstCurrentRun() );
    // add the quantities to the (thread local) run global YourRun object
    currentRun->AddEnergyDepositPerEvent( fEdepPerEvt );
    currentRun->AddChTrackLengthPerEvent( fChTrackLengthPerEvt );
    currentRun->FillEdepHistogram( fEdepPerEvt );
}
```

G4UserSteppingAction

```
//
// Score only if the step was done in the Target:
//   - cllect energy deposit for the mean (per-event) energy deposit computation
//   - same for the charged particle track length
//
void YourSteppingAction::UserSteppingAction(const G4Step* theStep)
{
    // Score steps done only in the target: i.e. pre-step point was in target
    if (theStep->GetPreStepPoint()->GetTouchableHandle()->GetVolume()
        != fYourDetector->GetTargetPhysicalVolume() )  return;
    // Step was done inside the Target so do scoring:
    //
    // Get the energy deposit and the step length
    const G4double eDep   = theStep->GetTotalEnergyDeposit();
    const G4double trackL = theStep->GetStepLength();
    // Get the G4Track and the ParticleDefinition to see if the particle is charged
    const G4ParticleDefinition* pDef = theStep->GetTrack()->GetParticleDefinition();
    if ( pDef->GetPDGCharge() != 0.0 )
    {
      // add current step length to the charged particle track length per-event
      fYourEventAction->AddChargedTrackLengthPerStep( trackL );
    }
    // add current energy deposit to the charged particle track length per-event
    fYourEventAction->AddEnergyDepositPerStep( eDep );
}
```

Questions?