

Interaction with the Geant4 kernel – part 1

... User classes (cont'ed)

At initialization

G4VUserDetectorConstruction

G4VUserPhysicsList

G4VUserActionInitialization

Global: **only one instance** exists in memory, shared by all threads.

At execution

G4VUserPrimaryGeneratorAction

G4UserRunAction*

G4UserEventAction

G4UserStackingAction

G4UserTrackingAction

G4UserSteppingAction

Local: an **instance** of each action class exists for each thread.
(*) Two RunAction's allowed: one for master and one for threads



Outlook

- Run, Event, Track, ...
 - a word about multi-threading
- Optional user action classes
- Command-based scoring
- Accumulables
- Analysis tools (*detached slides*)



Part I: The main ingredients



Geant4 terminology: an overview

- The following **keywords** are often used in Geant4
 - **Run, Event, Track, Step**
 - **Processes**: At Rest, Along Step, Post Step
 - **Cut** (or production threshold)
 - **Worker/master** thread (for MT)



Run, Event and Tracks

Run

Event 0

track 1

track 2

track 3

track 4

Event 1

track 1

track 2

track 3

Event 2

track 1

Event 3

track 1

track 2

track 3

track 4



The Event (G4Event)

- An Event is the **basic unit** of simulation in Geant4
- At the beginning of processing, **primary tracks** are **generated** and they are pushed into a stack
- A track is 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)
- **G4EventManager** class manages the event
- **G4UserEventAction** is the optional User hook



The Run (G4Run)

- As an **analogy** with a **real experiment**, a run of Geant4 starts with '**Beam On**'
- 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
- At the beginning of a Run, **geometry** is **optimised** for **navigation** and **cross section tables** are (re)calculated
- The **G4 (MT) RunManager** class **manages** the **processing** of each Run, represented by:
 - **G4Run** class
 - **G4UserRunAction** for an optional User hook



The Track (G4Track)

- 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



G4TrackStatus

- After each step the track can **change its state**
- The **status** can be (red can only be set by the User)

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)

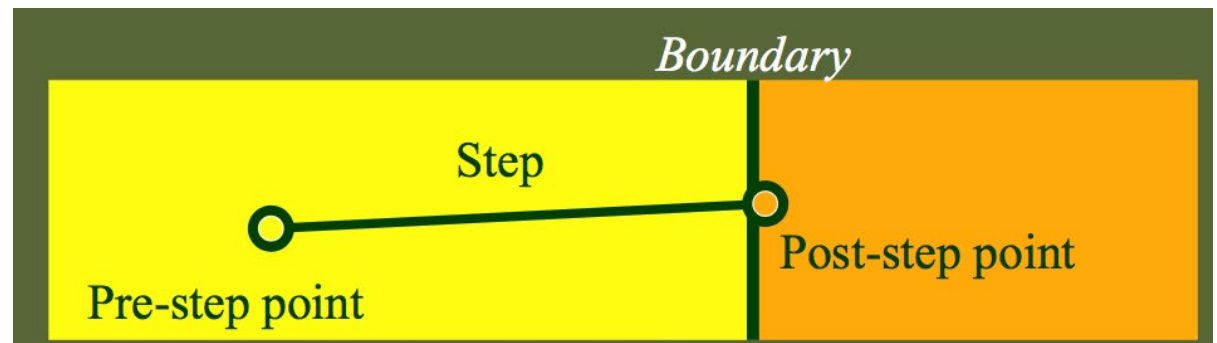


The Step (G4Step)

- **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 (*see later*)
 - **UserSteppingAction()** of your step action class file (*see later*)

The Step in Geant4

- 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**



- **G4UserSteppingAction** is the optional User hook



The G4Step object

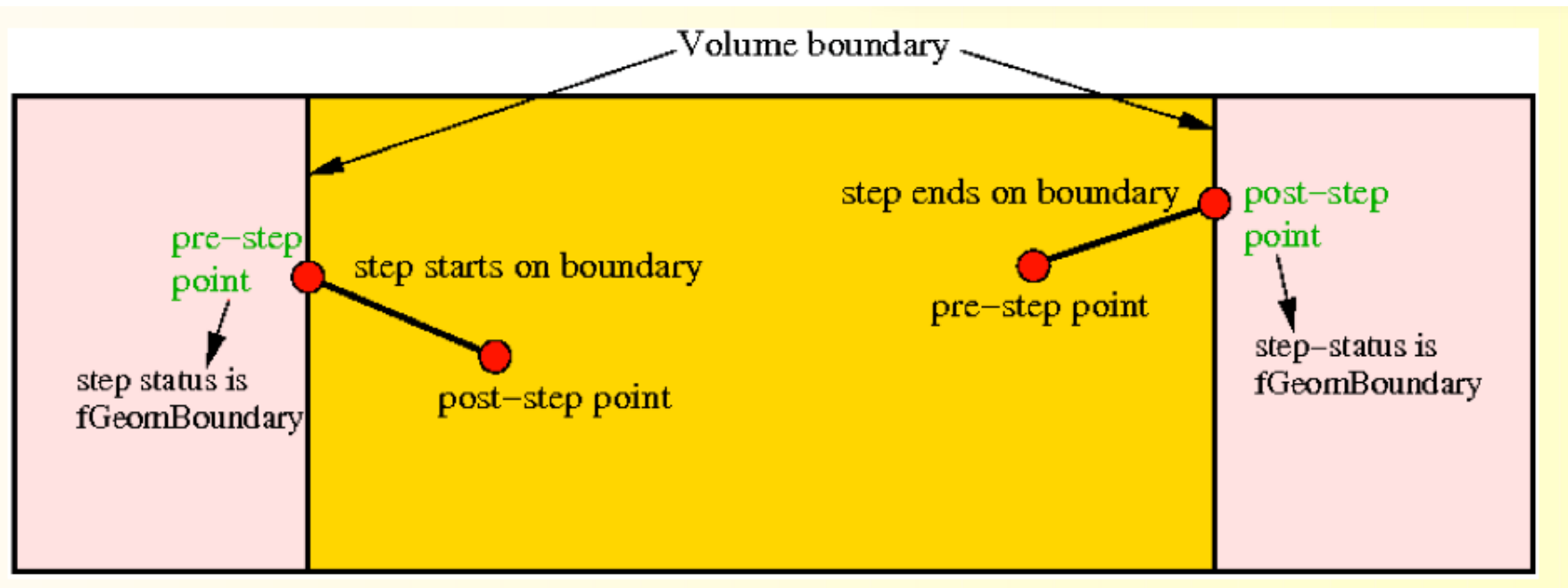
- A **G4Step** object contains
 - The **two endpoints** (pre and post step) so one has access to the **volumes** containing these endpoints
 - **Changes** in **particle properties** between the points
 - Difference of particle energy, momentum,
 - Energy deposition on step, step length, time-of-flight,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



Example: parent track and process

```
if (track->GetTrackID() != 1)
{
    G4cout << "Particle is a secondary" << G4endl;

    if (track->GetParentID() == 1)
    {
        G4cout << "But parent was a primary" << G4endl;
    }

    // Get process information
    G4VProcess* creatorProcess = track->GetCreatorProcess();
    G4String processName = creatorProcess->GetProcessName();
    G4cout << "Particle was created by " << processName << G4endl;
}
}
```




Example: boundaries

```
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();
```



Example: step "deltas"

```
MySensitiveDetector::ProcessHits(G4Step* step, G4TouchableHistory*) {  
    // Total energy deposition on the step (= energy deposited by energy loss  
    // process and energy of secondaries that were not created since their  
    // 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();  
}
```



Example: particle info

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

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

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

// The dynamic particle class contains e.g. the kinetic energy after the step:
G4double kinEnergy = dynParticle -> GetKineticEnergy();

// From the particle definition class you can retrieve static
// information like the particle name:
G4String particleName = particle -> GetParticleName();

G4cout << particleName << ": kinetic energy of "
        << (kinEnergy / MeV) << " MeV" << G4endl;
```



Part II: Optional User Action classes

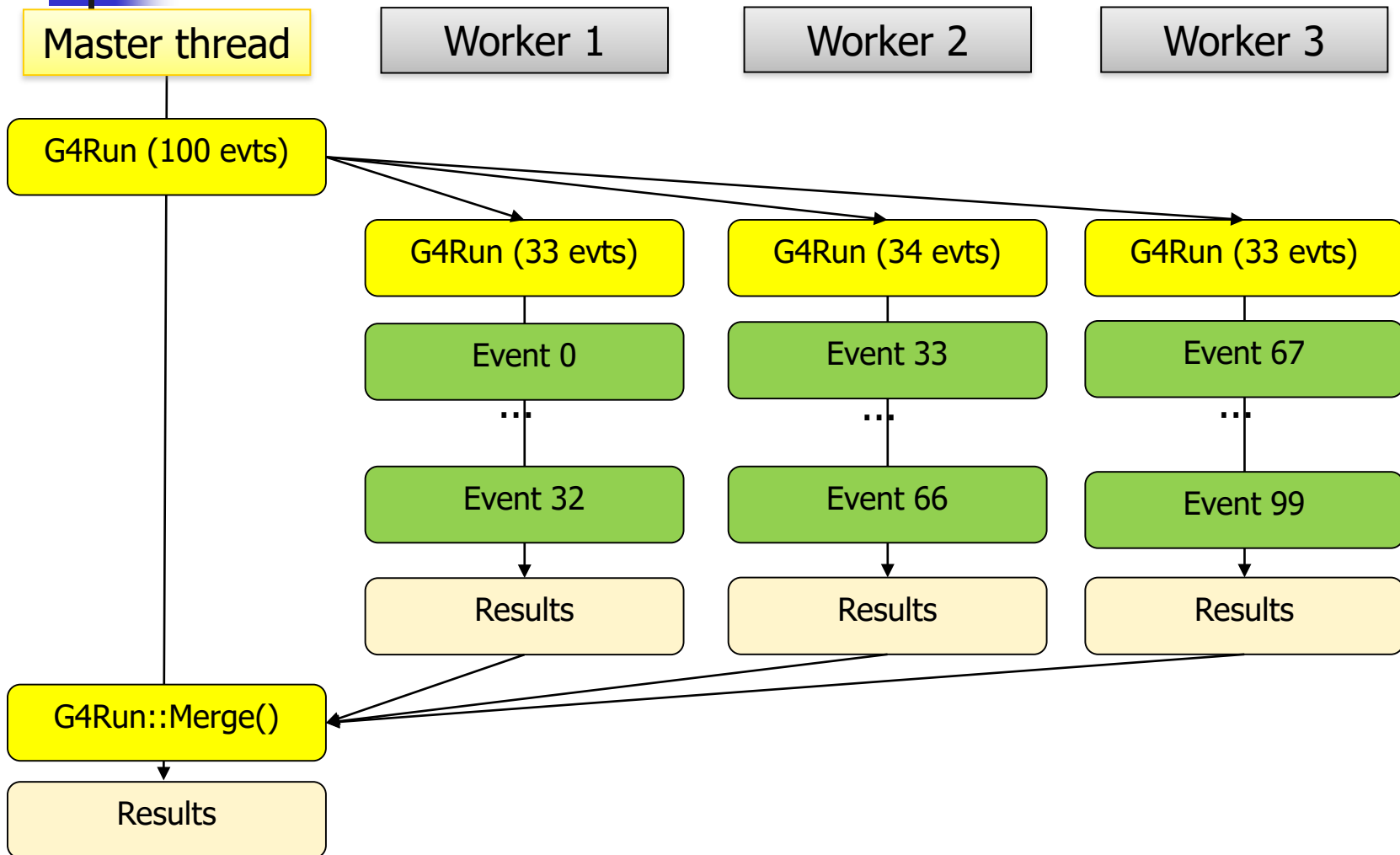


Optional user classes

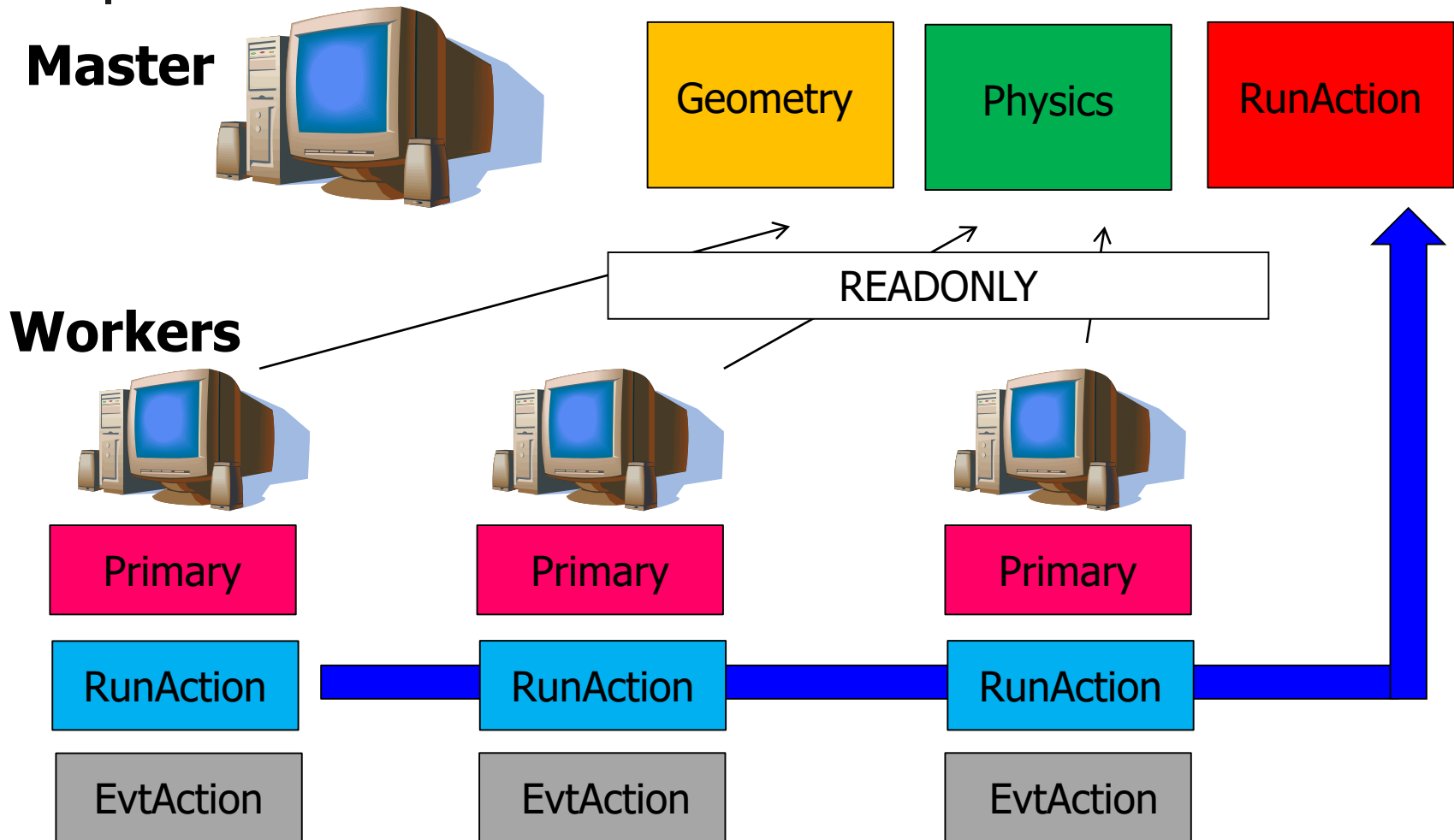
- Five **base classes** with **virtual methods** the user may override to step during the execution of the application ("user hooks")
 - G4User**Run**Action
 - G4User**Event**Action
 - G4User**Tracking**Action
 - G4User**Stacking**Action
 - G4User**Stepping**Action

e.g. actions to be done at the **beginning** and **end** of each event
- Default implementation (**not** purely virtual): **do nothing**
- Therefore, **override** only the methods you need.

Multi-threaded processing of events



User actions in MT mode





G4UserRunAction

```
void BeginOfRunAction (const G4Run*)  
void EndOfRunAction (const G4Run*)  
G4Run* GenerateRun ()
```

Uses:

- Book/output histograms and other analysis tools
- Custom G4Run with additional information
- Define parameters





G4UserEventAction

```
void BeginOfEventAction(const G4Event*)  
void EndOfEventAction(const G4Event*)
```

Uses:

- Hit collection and event analysis
- Event selection
- Logging (e.g. output event number)



G4UserStackingAction

G4ClassificationOfNewTrack

ClassifyNewTrack(const **G4Track***)

void **NewStage**()

void **PrepareNewEvent**()

Uses:

- Pre-selection of tracks (~manual cuts)
- Optimization of the order of track execution



G4UserTrackingAction

```
void PreUserTrackingAction (const  
    G4Track*)
```

```
void PostUserTrackingAction (const  
    G4Track*)
```

Uses:

- Track pre-selection
- Store trajectories



G4UserSteppingAction

```
void UserSteppingAction (const G4Step*)
```

Uses:

- Get information about particles
- Kill tracks under specific circumstances



Registration of user actions

- In multi-threading mode (and sequential), objects of user action classes must be **registered** to the **G4 (MT) RunManager** via a user-defined action initialization class

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

- In sequential mode, the actions can *also* be registered to the run manager directly (**not recommended**)

```
runManager->SetUserAction(new MyRunAction);
```

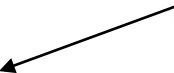


MyActionInitialization

- Register **thread-local** user actions

```
void MyActionInitialization::Build() const
{
    //Set mandatory classes
    SetUserAction(new MyPrimaryGeneratorAction());
    // Set optional user action classes
    SetUserAction(new MyEventAction());
    SetUserAction(new MyRunAction());
}
```

Also the primary
generator





Multiple user actions

- `G4MultiRunAction`
- `G4MultiEventAction`
- `G4MultiTrackingAction`
- `G4MultiSteppingAction`
- **no** `G4MultiStackingAction`

```
auto multiAction = new G4MultiEventAction{ new MyEventAction1,  
                                             new MyEventAction2 };  
//...  
multiAction->push_back(new MyEventAction3);  
SetUserAction(multiAction);
```

Containers enabling to have **multiple user actions** of the same “kind”, implemented as customized `std::vector`’s.

Part III: Command-based scoring



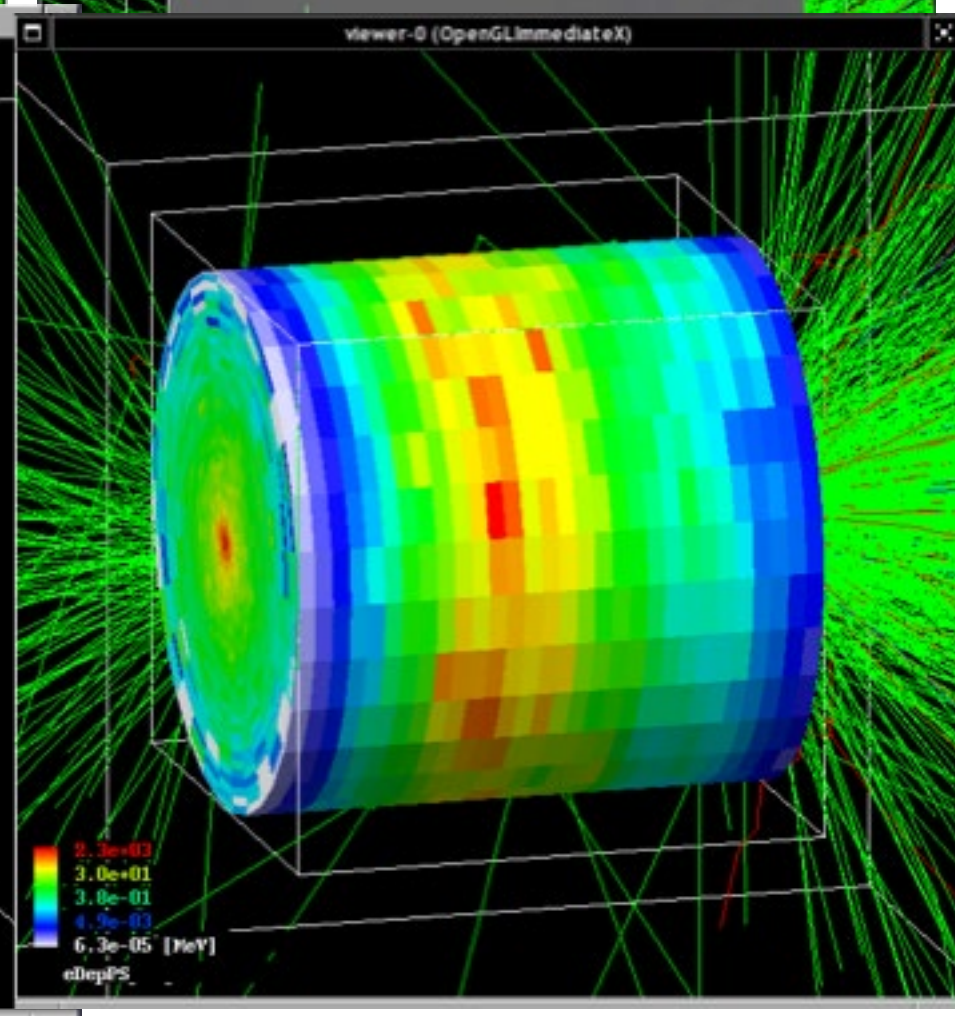
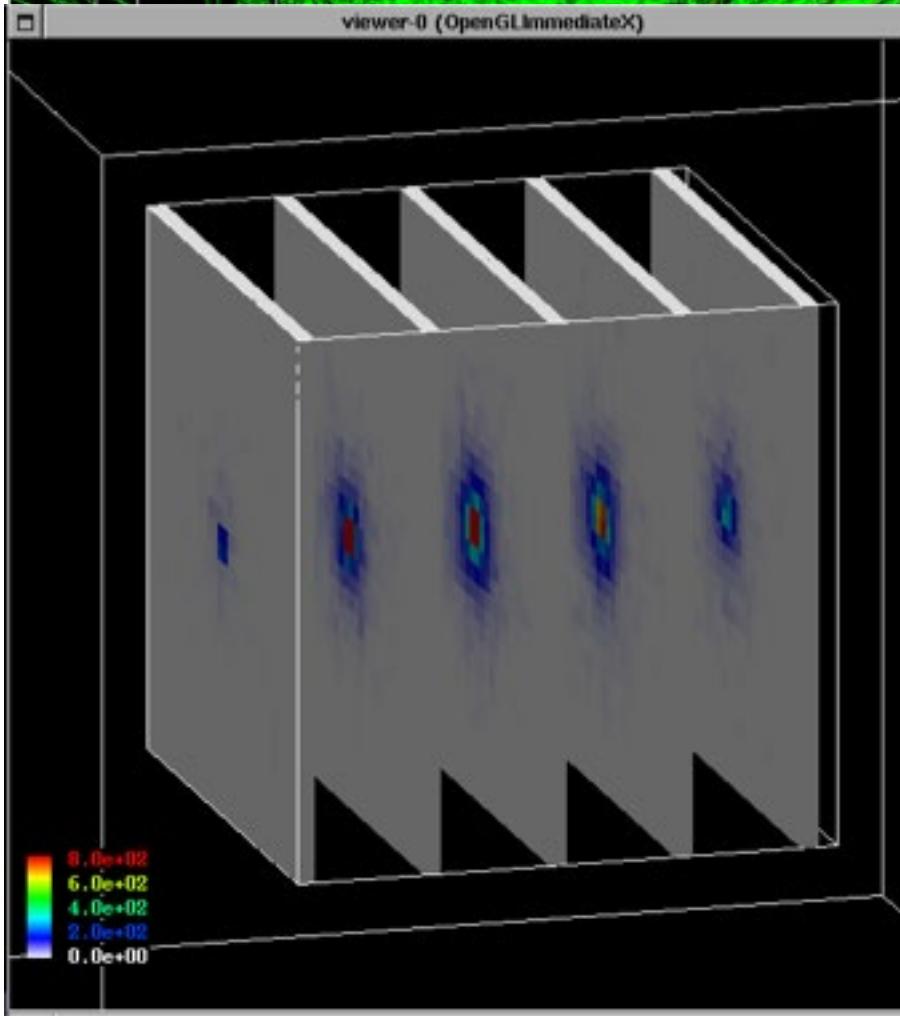
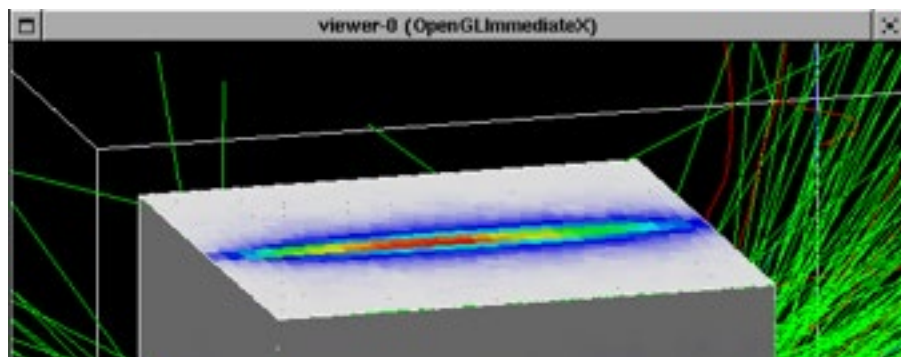
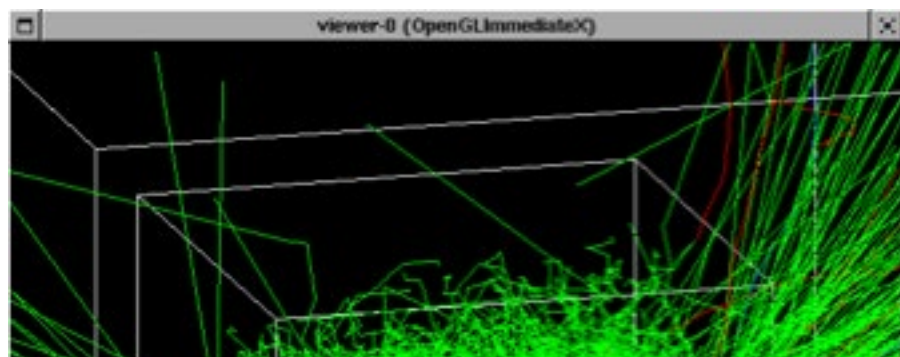


Command-based scoring

UI **commands** for scoring
→ **no C++ required**,
apart from instantiating
G4ScoringManager in
`main()`

```
int main() {  
    ...  
    G4ScoringManager::GetScoringManager();  
    ...  
}
```

- Define a scoring mesh
 - `/score/create/boxMesh <mesh_name>`
 - `/score/open, /score/close`
- Define mesh parameters
 - `/score/mesh/boxsize <dx> <dy> <dz>`
 - `/score/mesh/nbin <nx> <ny> <nz>`
 - `/score/mesh/translate,`
- Define primitive scorers
 - `/score/quantity/eDep <scorer_name>`
 - `/score/quantity/cellFlux <scorer_name>`
 - currently **20 scorers** are available
- Define filters
 - `/score/filter/particle <filter_name>`
`<particle_list>`
 - `/score/filter/kinE <filter_name>`
`<Emin> <Emax> <unit>`
 - currently **5 filters** are available
- Output
 - `/score/draw <mesh_name>`
`<scorer_name>`
 - `/score/dump,`
 - `/score/list`





G4analysis tools

(detached session)



Geant4 analysis classes

- A **basic analysis interface** is available in Geant4 for **histograms** (1D and 2D) and **ntuples**
 - Make life easier because they are **thread-safe**
 - **ROOT is not! Manual text output usually not!**
 - No need to worry about the interference of threads
- **Unique interface** to support different output formats
 - ROOT, AIDA XML, CSV and HBOOK
 - **Code** is the same, just change one line to switch from one to another
- Everything done via **G4AnalysisManager**
 - **Singleton** class → use Instance()
 - **UI commands** available



g4analysis

- Selection of output format is performed by including a proper **header file**
- **All** the rest of the code **unchanged**
 - Unique interface

```
#ifndef MyAnalysis_h
#define MyAnalysis_h 1

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

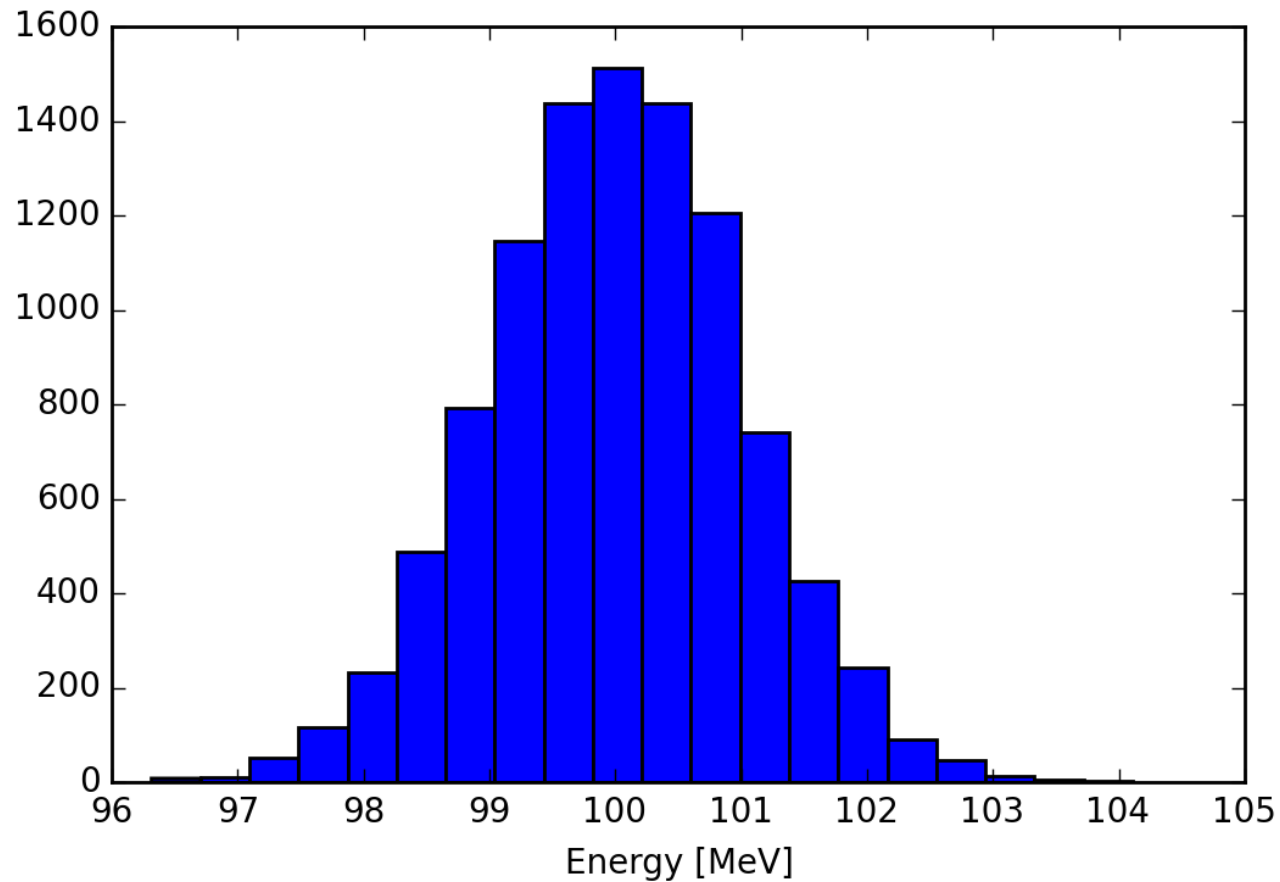
#endif
```



Advanced topic: It is possible to use more formats at the same time. See documentation.



Histograms





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);
```

} Start numbering of
histograms from ID=1

```
    // Creating histograms
```

```
    man->CreateH1("h","Title", 100, 0., 800);
```

```
    man->CreateH1("hh","Title",100,0.,10);
```

} ID=1
ID=2

```
    // Open an output file
```

```
    man->OpenFile("myoutput");
```

} Open output file

```
}
```

Fill histograms and write on file

```
#include "MyAnalysis.hh"

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

void MyRunAction::EndOfRunAction(const G4Run* aRun)
{
    G4AnalysisManager::Instance()->Write();
}

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




Ntuples

EventID	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



Ntuples support

- g4tool supports **ntuples**
 - Any number of ntuples
 - Any number of columns per ntuple
 - Supported types are **int/float/double**
- For more complex tasks (e.g. full functionality of ROOT TTrees) have to link ROOT directly
 - And take care of thread-safety



Book ntuples

```
#include "MyAnalysis.hh"

void MyRunAction::BeginOfRunAction(const G4Run* run)
{
    // Get analysis manager
    auto man = G4AnalysisManager::Instance();
    man->SetFirstNtupleId(1); } Start numbering of
                               ntuples from ID=1

    // Creating ntuple
    man->CreateNtuple("name", "Title");
    man->CreateNtupleDColumn("Eabs");
    man->CreateNtupleDColumn("Egap");
    man->FinishNtuple(); } ID=1

    man->CreateNtuple("name2", "title2");
    man->CreateNtupleIColumn("ID");
    man->FinishNtuple(); } ID=2
}
```



Fill ntuples

- File handling and general clean-up as shown for histograms

```
#include "MyAnalysis.hh"
void MyEventAction::EndOfEventAction(const G4Run* aRun)
{
    auto man = G4AnalysisManager::Instance();
    man->FillNtupleDColumn(1, 0, fEnergyAbs);
    man->FillNtupleDColumn(1, 1, fEnergyGap);
    man->AddNtupleRow(1);
    } ID=1,
    columns 0, 1

    man->FillNtupleIColumn(2, 0, fID);
    man->AddNtupleRow(2);
    } ID=2,
    column 0

}
```



G4Accumulable<T>

- Templated class to **collect** simple information
 - Thread-safe
 - Accumulable during Run
 - Value **merge** at the end (explicit) MT
 - Scalar variables only (otherwise, expert)
- Alternative to ntuples/histograms (*later*)
- Managed by **G4AccumulableManager**
- See example **B1** and **B3a**



Geant4 10.2: Previously named **G4Parameter**!



G4Accumulable – C++ (1)

1) **Declare** (instance) variables (of RunAction)

```
G4Accumulable<G4int>      fNElectrons;  
G4Accumulable<G4double> fAverageElectronEnergy;
```

2) **Register** to accumulable manager (in RunAction constructor)

```
G4AccumulableManager* accManager = G4AccumulableManager::Instance();  
accManager->RegisterAccumulable(fNElectrons);  
accManager->RegisterAccumulable(fAverageElectronEnergy);
```

3) **Reset** to zero values (in RunAction::BeginOfRunAction)

```
G4AccumulableManager* accManager = G4AccumulableManager::Instance();  
accManager->Reset();
```

4) **Update** during run (e.g. in Stacking action)

```
fNElectrons += 1;      // Normal arithmetics
```



G4Accumulable – C++ (2)

5) **Merge** after run (in `RunAction::EndOfRunAction`)

MT

```
G4AccumulableManager* accManager = G4AccumulableManager::Instance();  
accManager->Merge();
```

6) **Report** after run (in `RunAction::EndOfRunAction`)

```
G4AccumulableManager* accManager = G4AccumulableManager::Instance();  
if (IsMaster())  
{  
    if (fNElectrons.GetValue())  
    {  
        G4cout << " * Produced " << fNElectrons.GetValue();  
        G4cout << " secondary electrons/event. Average energy: ";  
        G4cout << fAverageElectronEnergy.GetValue()/keV/fNElectrons.GetValue();  
        G4cout << " keV" << G4endl;  
    }  
    else  
        G4cout << " * No secondary electrons produced" << G4endl;  
}
```



More slides...



Output stream (G4cout)

- **G4cout** is a **iostream** object defined by Geant4.
 - Used in the same way as standard **std::cout**
 - Output streams handled by **G4UImanager**
 - **G4endl** is the equivalent of **std::endl** to end a line
- MT-handling: will display also the **threadID**
WT1> I am here
WT5> I am here
- Output strings may be displayed in **another window** (Qt GUI) or **redirected** to a file



Example: output on screen

```
void SteppingAction::UserSteppingAction(const G4Step* aStep)
{
    // Collect data
    G4Track* theTrack = aStep->GetTrack();
    G4DynamicParticle* particle = theTrack->GetDynamicParticle();
    G4ParticleDefinition* parDef = particle->GetDefinition();

    G4double edep = aStep->GetTotalEnergyDeposit();
    G4double particleCharge = particle->GetCharge();
    G4double kineticEnergy = theTrack->GetKineticEnergy();

    // The output
    G4cout
        << "Energy deposited--->" << " " << edep << "
        << "Charge--->" << " " << particleCharge << " "
        << "Kinetic Energy --->" << " " << kineticEnergy << " " <<
    G4endl;
}
```



Output on screen: an example

Begin of Event: 0

Energy deposited---	9.85941e-22	Charge---	6	Kinetic energy---	160
Energy deposited---	8.36876	Charge---	6	Kinetic energy---	151.631
Energy deposited---	8.63368	Charge---	6	Kinetic energy---	142.998
Energy deposited---	5.98509	Charge---	6	Kinetic energy---	137.012
Energy deposited---	4.73055	Charge---	6	Kinetic energy---	132.282
Energy deposited---	0.0225575	Charge---	6	Kinetic energy---	132.254
Energy deposited---	1.47468	Charge---	6	Kinetic energy---	130.785
Energy deposited---	0.0218983	Charge---	6	Kinetic energy---	130.76
Energy deposited---	5.22223	Charge---	6	Kinetic energy---	125.541
Energy deposited---	7.10685	Charge---	6	Kinetic energy---	118.434
Energy deposited---	6.62999	Charge---	6	Kinetic energy---	111.804
Energy deposited---	6.50997	Charge---	6	Kinetic energy---	105.294
Energy deposited---	6.28403	Charge---	6	Kinetic energy---	99.0097
Energy deposited---	5.77231	Charge---	6	Kinetic energy---	93.2374
Energy deposited---	5.2333	Charge---	6	Kinetic energy---	88.0041
Energy deposited---	3.9153	Charge---	6	Kinetic energy---	84.0888
Energy deposited---	14.3767	Charge---	6	Kinetic energy---	69.7121
Energy deposited---	14.3352	Charge---	6	Kinetic energy---	55.3769

Example: output to an ASCII file

MT



```
#include <fstream>

class SteppingAction{
    // ...
    std::ofstream fout;
};

SteppingAction::SteppingAction() : fout("outfile.txt") { }

void SteppingAction::UserSteppingAction(const G4Step* aStep)
{
    G4Track* theTrack = aStep->GetTrack();
    G4double edep = aStep->GetTotalEnergyDeposit();
    G4double kineticEnergy = theTrack->GetKineticEnergy();

    // The output
    fout
        << "Energy deposited--->" << " " << edep << " "
        << "Kinetic Energy -->" << " " << kineticEnergy << G4endl;
}
```



Hands-on session

- Task4
 - Task4a: User Actions
 - Task4b: Command-based scoring
- `http://geant4.lngs.infn.it/belgrade2019/task4`

Two years open post-doc position (junior or senior) for working on Geant4 for simulation of a protontherapy beamline and connected issues

In case of interest write to
pablo.cirrone@lns.infn.it