Geant4

Lecture 4: Getting at the numbers

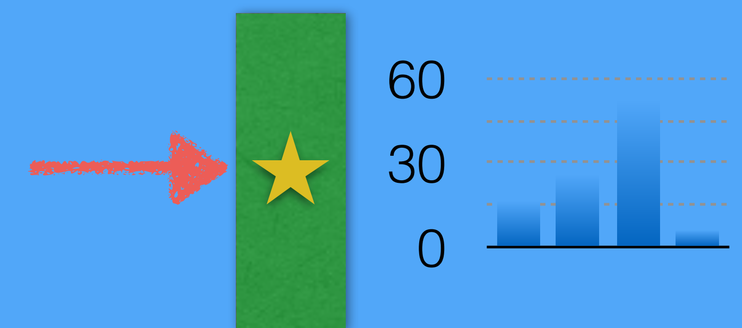
So far we've drawn pictures. Which can be enlightening. But for real results we need tables and graphs

Hierarchy

- **The Run**. A sequence of events with the same geometry, source, etc. Probably one run per job.
- **The Event.** One call to the particle gun, and everything that follows.
- **The Track**. A particle from start to finish. Less important - more later
- **The Step.** One particle moves a small distance and stuff happens. A track is a sequence of steps.

Corresponding objects `RunAction, EventAction, TrackingAction, SteppingAction`. Optional - you provide them. Run, Event, Track have function call at start and end

Example: fire particles at a block. Plot the energies deposited.



- SteppingAction: score the actual energy deposits
- EventAction: keep track of which deposit is in which event
- RunAction: Sort stuff at the beginning. Print/plot/tidy up at the end.

SteppingAction. Taken from exampleB1 with some extra print statements.

.hh file

3 functions
2 data members

This is my particular G4UserSteppingAction

Constructor

Destructor

Action

```
class B1SteppingAction : public G4UserSteppingAction
{
  public:
    B1SteppingAction(B1EventAction* eventAction);
    virtual ~B1SteppingAction();

    // method from the base class
    virtual void UserSteppingAction(const G4Step*);

  private:
    B1EventAction*  fEventAction;
    G4LogicalVolume* fScoringVolume;
};
```

It knows about the eventAction

fScoringVolume is one way to control where energies etc are taken notice of.
Set in DetectorConstruction
Specific to the example - not very general

.cc file

```
B1SteppingAction::B1SteppingAction(B1EventAction* eventAction)
: G4UserSteppingAction(),
  fEventAction(eventAction),
  fScoringVolume(0)
{G4cout<<"B1SteppingAction created\n";}

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......

B1SteppingAction::~B1SteppingAction()
{G4cout<<"B1SteppingAction deleted\n";}

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......

void B1SteppingAction::UserSteppingAction(const G4Step* step)
{
G4cout<<"B1SteppingAction called\n";
  if (!fScoringVolume) {
    const B1DetectorConstruction* detectorConstruction
      = static_cast<const B1DetectorConstruction*>
        (G4RunManager::GetRunManager()->GetUserDetectorConstruction());
    fScoringVolume = detectorConstruction->GetScoringVolume();
  }

  // get volume of the current step
  G4LogicalVolume* volume
    = step->GetPreStepPoint()->GetTouchableHandle()
      ->GetVolume()->GetLogicalVolume();

  // check if we are in scoring volume
  if (volume != fScoringVolume) return;

  // collect energy deposited in this step
  G4double edepStep = step->GetTotalEnergyDeposit();
  fEventAction->AddEdep(edepStep);
}

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......
```

G4Step accessible inside steppingaction

```
GetStepLength()
GetTotalEnergyDeposit()
GetNonIonizingEnergyDeposit()
GetNumberOfSecondariesInCurrentStep()
GetSecondaryInCurrentStep()

GetTrack()
    GetTrackID()
    GetParentID()


GetPreStepPoint() (also GetPostStepPoint)
    GetPosition()
    GetMomentumDirection()
    GetGlobalTime()
    GetLocalTime()
    GetMomentum()
    GetKineticEnergy()
    GetTouchableHandle()
        GetVolume()
            GetLogicalVolume
```

What might you do?

Just stuff - eg for shielding calculations. Direct from SteppingAction

- Find the total energy deposited (dosimetry)
- Find the total flux (track length)
- Count number of particles (of some type) crossing boundary

Detector simulation - SteppingAction processed by EventAction

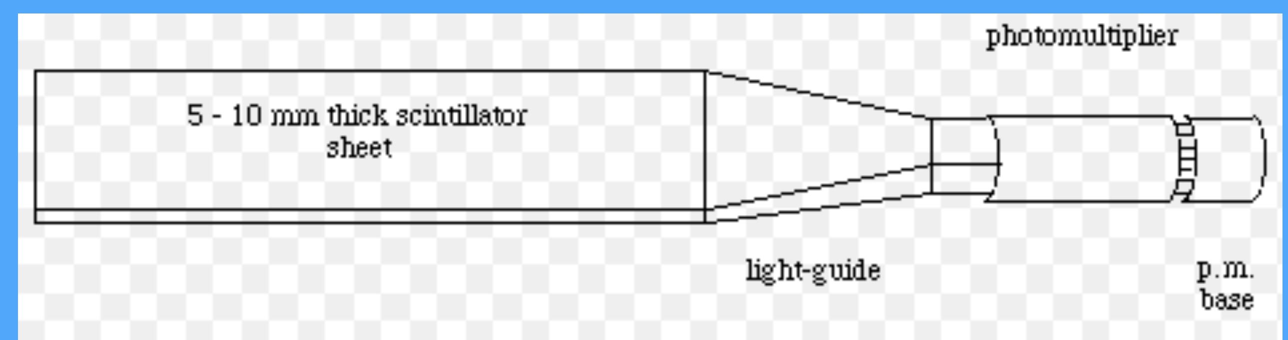
- Find the time a track enters the detector
- Find the position that a hodoscope or tracking chamber would report
- Find energy deposited in this volume in this event

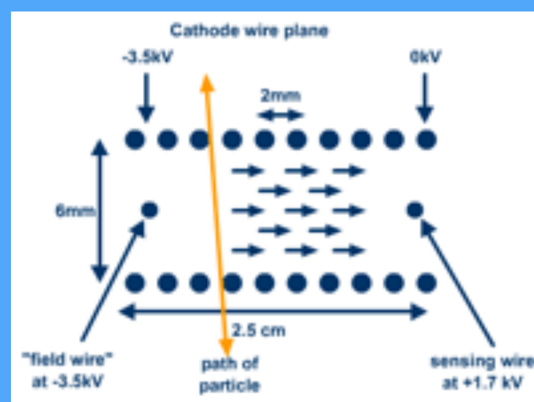
Hits and Digits

Remember G4 was written for simulating detectors

Particles interact in detector ('hit') producing a signal which is amplified and read out by the electronics ('digits')

Example: Track goes through scintillator. Light pulse amplified by phototube and sets voltage level when threshold exceeded. Good timing accuracy but poor space and energy resolution





Example: Track goes through gas creating electrons which drift towards a wire (known velocity) where they multiply and give a signal. Good position but poor energy resolution: timing needed.

SteppingAction needs to talk to EventAction

```
void B1ActionInitialization::Build() const
{
  SetUserAction(new B1PrimaryGeneratorAction);

  B1RunAction* runAction = new B1RunAction;
  SetUserAction(runAction);

  B1EventAction* eventAction = new B1EventAction(runAction);
  SetUserAction(eventAction);

  SetUserAction(new B1SteppingAction(eventAction));
}
```

Called as part of initialisation

Four UserActions are set so the Run Manager knows about them and will invoke them at appropriate times.

Query: why two different styles? 2 & 3 look different from 1 & 4?

Answer: SteppingAction contains a pointer to EventAction which has to be set in the constructor. So it must be available. Likewise EventAction need RunAction

```
class B1SteppingAction : public G4UserSteppingAction
{
  public:
    B1SteppingAction(B1EventAction* eventAction);
    virtual ~B1SteppingAction();

    // method from the base class
    virtual void UserSteppingAction(const G4Step*);

  private:
    B1EventAction*  fEventAction;
    G4LogicalVolume* fScoringVolume;
};
```

```
B1SteppingAction::B1SteppingAction(B1EventAction* eventAction)
: G4UserSteppingAction(),
  fEventAction(eventAction),
  fScoringVolume(0)
{G4cout<<"B1SteppingAction created\n";}
```

SteppingAction and EventAction

Want to know the energy deposited in each event.
Stepping action obtains the energy deposited in this step
Variable needed to store running total for this event
SteppingAction increments it
EventAction zeros it at the start, processes it at the end

```
class B1EventAction : public G4UserEventAction
{
  public:
    B1EventAction(B1RunAction* runAction);
    virtual ~B1EventAction();

    virtual void BeginOfEventAction(const G4Event* event);
    virtual void EndOfEventAction(const G4Event* event);

    void AddEdep(G4double edep) { fEdep += edep; }

  private:
    B1RunAction* fRunAction;
    G4double     fEdep;
};
```

```
// collect energy deposited in this step
G4double edepStep = step->GetTotalEnergyDeposit();
fEventAction->AddEdep(edepStep);
```

```
//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo.

void B1EventAction::BeginOfEventAction(const G4Event*)
{
  fEdep = 0.;
}

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo.

void B1EventAction::EndOfEventAction(const G4Event*)
{
  // accumulate statistics in run action
  fRunAction->AddEdep(fEdep);
}
```

From B1SteppingAction.cc

From B1EventAction.hh

From B1EventAction.cc

Alternative design?

Make fEdep member data in SteppingAction?
Then accumulation would be easier.
```
fEdep += step->GetTotalEnergyDeposit();
```
Event Start and End actions more complicated as EventAction would need to know about SteppingAction, but not vice versa

Make fEdep global?
Then accumulation would be easier.
```
fEdep += step->GetTotalEnergyDeposit();
```
Event Start and End actions also simple. No need for EventAction or SteppingAction to know about each other

Global Data

Defined once, outside any function or class definitions. Typically in file continuing main program

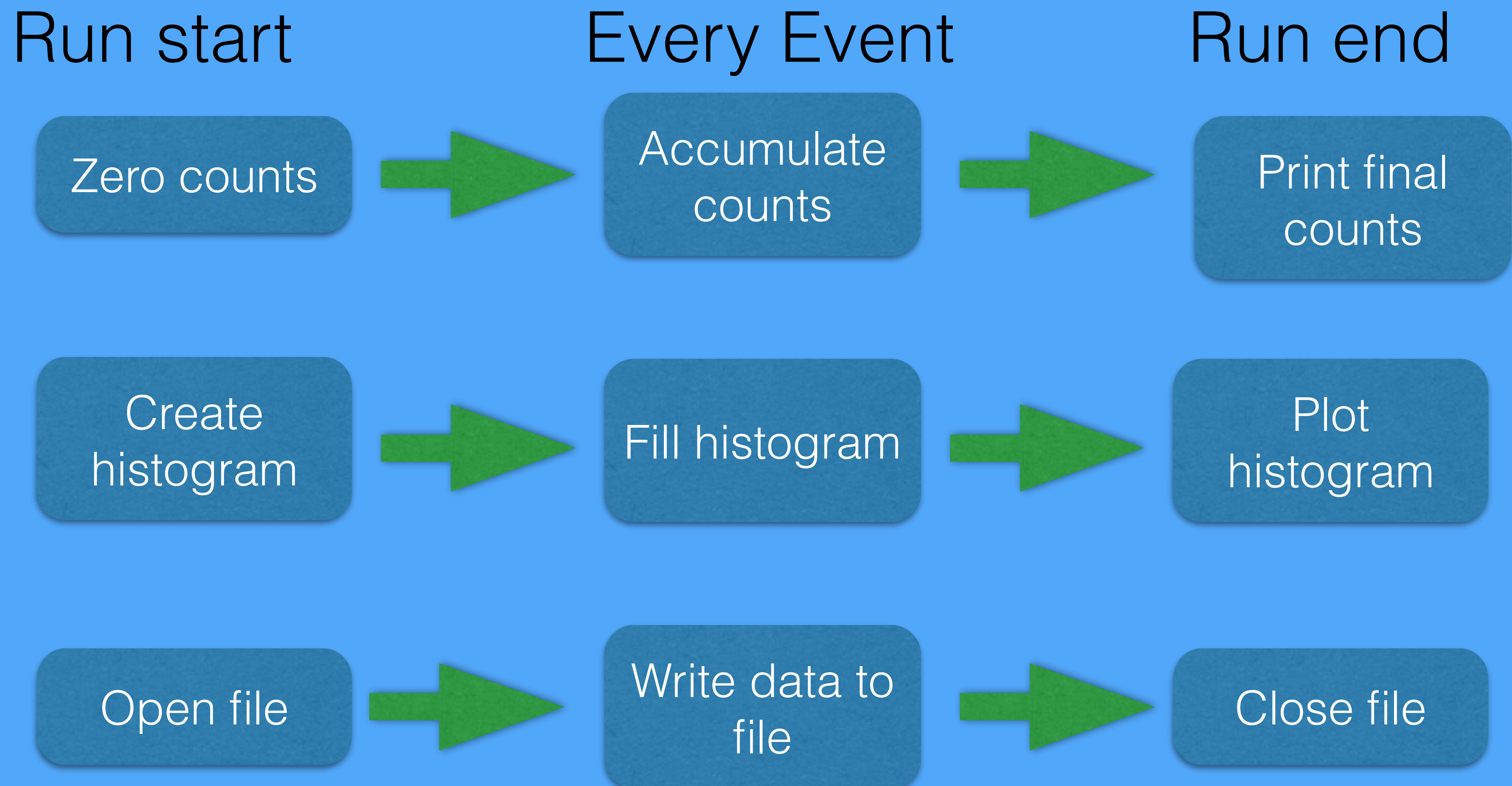
To use it in separate file, you declare it with extern

```
G4double fEdep; // in your main program file but before int main(){…

extern G4double fEdep;// in SteppingAction.cc or wherever
```

Global data is ***powerful but dangerous***. Breaks principle of data encapsulation, i.e. the programmer and the compiler know that variables can only be affected through calls to the objects that contain them. But it can save much hassle.

What about RunAction? Typical cases



EventAction needs access to the RunAction data (counts, histogram, file…)

Do you need the Run Manager?

Possibly not.

If there is one run per job, then the EventManager constructor and destructor are called at the beginning and end of the job, and you may be able to put initial and final stages there

Final stage must not rely on data in other classes which may have been destroyed first

Like many design choices, there are lots of right answers and no universal ‘best’ practice.

Minimal File Output

- In `Eventaction.hh` add `#include <fstream>` and data member `std::ofstream resultsfile;`

- In `Eventaction.cc`

  1. add to constructor `resultsfile.open("yourfilename");`
     `if(!resultsfile.is_open()) {G4cout<<"file not opened"; exit(1);}`

  2. add to destructor `resultsfile.close();`

  3. **add to** `EndOfEventAction`
     `resultsfile<<whatever<<G4endl;`

Histograms

- You can use ROOT to book, fill and draw histograms. This adds another layer of complexity that we'll avoid.
- Dumping stuff to file and histogramming it later with R or MATLAB or … is the best way for many purposes
- If your files are enormous, binary files will save you a large factor.
- A simple histogrammer is easy to write if you know what you want - say 20 bins between -5 and 5. `int h[20]={0};` in class definition, `int i=(x-5)/0.5; if((i>=0) & (i<20)) h[i]++;` in EventAction, `for(int i=0;i<20;i++) G4cout<<h[i]<<" ";` in Destructor. Adapt to your own particular neeeds

Other Actions (1) Tracking

You can define a `TrackingAction`. Inherits from `G4UserTrackingAction`

Calls `PreUserTrackingAction(G4Track*)` and `PostUserTrackingAction(G4Track*)`

Called before and after a track - a sequence of steps within an event (an event can have many tracks)

Information in G4Track

GetTrackID() - a number to identify track
GetParentID() - likewise. Zero for primary
GetDynamicParticle() - kinematic variables
GetParticleDefinition() - book values of mass, width, charge name etc
and more

How do we find out all this stuff?

There are usually lots of ways of accessing the information you want.

Finding an example and copying it is not good, as the person you're copying code from is probably no more knowledgeable than you

Finding a tutorial is a bit better, but often the tutorial is illustrating a different problem.

For example, suppose you want to know the particle velocity. One example may tell you a way to find the energy. Another may tell you how to find the mass. Problem solved! But you can actually access the velocity directly.

The best way to see what data and methods are in a class is to look at the source code in the .hh files. (No need to look at the .cc files). They are in `/home/software/geant/geant4.10.03/source/<directory structure>/include` . The directory structure is messy. To find what's where I recommend `find /home/software/geant/geant4.10.03/source/ classname.hh | grep classname`

```
[roger@iiaa1 G4L4]$ find /home/software/geant/geant4.10.03/source G4ParticleGun.hh | grep G4ParticleGun
/home/software/geant/geant4.10.03/source/event/src/G4ParticleGun.cc
/home/software/geant/geant4.10.03/source/event/src/G4ParticleGunMessenger.cc
/home/software/geant/geant4.10.03/source/event/include/G4ParticleGunMessenger.hh
/home/software/geant/geant4.10.03/source/event/include/G4ParticleGun.hh
find: â: No such file or directory
```

If you have a nice IDE this makes life much easier

Other Actions (2) Stacking

There is another possibility called `StackingAction`
You can ignore it

Geant4 starts with a list of tracks from the generation. It processes one track at a time. In doing so more tracks may be generated and they are put on the stack, which is LIFO.

StackingAction gives the possibility to re-prioritise tracks so that ones deemed important are processed first.

Limited use, unless you're really tweaking performance.

Highlights point: these actions can not only be used for recording what's happenng, but also for driving the simulation (killing boring tracks). Lots of the stuff in the code is there for this purpose and we don't need it.

Where am I?

You want to know about steps in some volumes (detectors) but not others. Or differently in different volumes (scintillator + drift chamber).

Directly: can look at

```
step->GetPreStepPoint()->GetPhysicalVolume()->GetName()
```

This is the name you gave it in the DetectorConstruction code. So you can test it against "Shape2" (or whatever) and take appropriate action.

More efficiently: note physical or logical volume when created in constructor, store in some data member you've created, and test against

```
step->GetPreStepPoint()->GetPhysicalVolume()
```

or

```
step->GetPreStepPoint()->GetTouchableHandle()->GetVolume()->GetLogicalVolume();
```

Physical or logical? If one LV per PV then PV easier. If several LV per PV need to get it right

Assignment

Simulate 25 GeV protons incident on a 10 cm radius ball of iron (Aster), cobalt (Dario) and nickel (Mert).

Plot the energy spectrum of pions (i) produced, and (ii) emerging from the ball.

Plot the numbers of pions per event, produced and emerging. Is the distribution Poisson?