Building Complete GEANT application

-- Basic Structure of Geant4 Code

-- Where to write what

Things to be discussed

Geant4 Analogy of real experiment

Basic structure of the simulation code.

Writing a basic simulation code

Mandatory classes for your simulation code.

- -- Implementation of these mandatory classes

Getting the required information out of you simulation

- -- Optional classes
  - -- Implementation of these optional classes

Geant4 Analogy of the real experiment setup

Beam On : As in real experiment the Geant4 run starts with “Beam On”

A run is basically a collection of event.

As in experiment once the run start, user cannot change anything
--> Geometry Setup
--> Physics processes to study

Before starting the run, following things need to be initialized
--> Detector setup (geometry is optimized)
--> Physics List (cross-section tables are calculated, depending upon the materials used in the geometry creation)

Important user classes : Geant4 Program structure

Define your entry point : main() : There is no starting point provided by Geant4.

It is the place where you actually registers different component of you application.

Initialization classes : Classes whose objects needs to initiated before you simulation starts.

Detector : **G4VUserDetectorConstruction**
Physics : **G4VUserPhysicsList / Existing or Implemented**
UserActions : **G4VUserActionInitialization**

Action classes :

instantiated in the **G4VUserActionInitialization**

The action classes are invoked during the event loop : ie. When you simulation is running.

G4VUserPrimaryGeneratorAction
G4UserRunAction
G4UserEventAction
G4UserStackingAction
G4UserTrackingAction
G4UserSteppingAction

The classes starting with **G4V** are abstract classes.
Their objects **can't** be created.
They are there to provide a skeleton required by Geant4
User needs to **inherit these classes**, and to implement few functions which are mandatory.

Creation of your DetectorConstruction : G4VUserDetectorConstruction

```
class G4VUserDetectorConstruction
{
 public:
  G4VUserDetectorConstruction();
  virtual ~G4VUserDetectorConstruction();

  virtual G4VPhysicalVolume* Construct() = 0;
};          (Pure virtual function)
```

The **Construct** method should return the pointer to the world physical volume, which represents your entire geometry setup.

```
class Sim01_DetectorConstruction : public
G4VUserDetectorConstruction
{
public:
        Sim01_DetectorConstruction(){}
        ~Sim01_DetectorConstruction(){}
        G4VPhysicalVolume* Construct(){
                //Write your stuff here
                //construct all your materials
                //construct all your volumes
                //declare you volume as sensitive
        }
};
```

Define your Physics

There is no default particles and physics process that comes automatically in your simulation code.

Not even particle transport.

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 all the required cut-off ranges

OR use the various physics lists that are already available in GEANT4.
FPFP_BERT **(add few more list)**

Primary Generator : G4VUserPrimaryGeneratorAction

The second mandatory user class : Controls the generation of primary particles.

--> This is again a abstract class

--> You cannot instantiate it : Will not do anything on its own

```
class G4VUserPrimaryGeneratorAction
{
 G4VUserPrimaryGeneratorAction();
  virtual ~G4VUserPrimaryGeneratorAction();
  virtual void GeneratePrimaries(G4Event* anEvent) = 0;
};
```

```
class Sim01_PrimaryGeneratorAction : public G4VUserPrimaryGeneratorAction
{
  G4ParticleGun *fParticleGun;
  Sim01_PrimaryGeneratorAction(){}
  ~Sim01_PrimaryGeneratorAction(){}

      void GeneratePrimaries(G4Event*){
         fParticleGun->GeneratePrimaryVertex();
      }

};
```

The generate primaries method is called at the beginning of every event.
Your primary generator will not generate any primary particle, until you call **GeneratePrimaryVertex()** function

Called only once

```
Sim01_PrimaryGeneratorAction::Sim01_PrimaryGeneratorAction() {

        int numOfParticle = 1;
        fParticleGun = new G4ParticleGun(numOfParticle);
        G4ParticleTable *particleTable = G4ParticleTable::GetParticleTable();
        G4ParticleDefinition *particle = particleTable->FindParticle("mu-");
        fParticleGun->SetParticleDefinition(particle);
        fParticleGun->SetParticleMomentumDirection(G4ThreeVector(0.,0.,-1.));
        fParticleGun->SetParticleEnergy(3.*GeV);
        fParticleGun->SetParticlePosition(G4ThreeVector(0.,0.,30.*cm));
}
```

Called in the beginning of every event

```
void Sim01_PrimaryGeneratorAction::GeneratePrimaries(G4Event
*event) {
        fParticleGun->SetParticleMomentumDirection(G4RandomDirection());
        fParticleGun->GeneratePrimaryVertex(event);
}
```

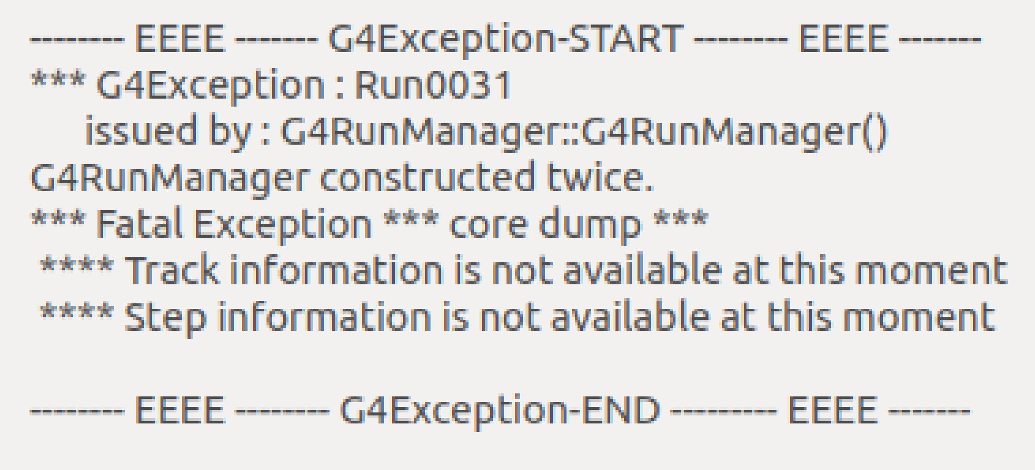
Run Manager : G4RunManager

One of the manager class in Geant4 .

Helps in linking various objects and modules required during the initialization and run.

The program cannot run without the Run Manager.

User can inherit in their derived class to customize the behaviour



```
-------- EEEE ------- G4Exception-START -------- EEEE -------
*** G4Exception : Run0031
      issued by : G4RunManager::G4RunManager()
G4RunManager constructed twice.
*** Fatal Exception *** core dump ***
 **** Track information is not available at this moment
 **** Step information is not available at this moment

-------- EEEE -------- G4Exception-END --------- EEEE -------
```

G4RunManager or its Derived class must be singleton
--> Only one object should exist in the program's memory.

Singleton instance helps in accessing the same RunManager object in different locations in the code.

Action Initialization : G4VUserActionInitialization

Basically used to instantiate various classes required during event loop

```
class G4VUserActionInitialization
{
    G4VUserActionInitialization();
    virtual ~G4VUserActionInitialization();

    virtual void Build() const = 0;
}
```

```
class Sim01_ActionInitialization : public
G4VUserActionInitialization
{
  public:
    Sim01_ActionInitialization(){}
    virtual ~Sim01_ActionInitialization(){}

    virtual void BuildForMaster() const{}
    virtual void Build() const{
        // Link the objects of classes invoked
           during the event loop
           // EventAction, SteppingAction
    }
};
```

Revisit : Geant4 Program structure

Define your entry point : main() : There is no starting point provided by Geant4.

It is the place where you actually registers different component of you application.

Initialization classes : Classes whose objects needs to initiated before you simulation starts.

Detector : **G4VUserDetectorConstruction**
Physics : **G4VUserPhysicsList / Existing or Implemented**
UserActions : **G4VUserActionInitialization**

Action classes :

instantiated in the **G4VUserActionInitialization**

The action classes are invoked during the event loop : ie. When you simulation is running.

G4VUserPrimaryGeneratorAction
G4UserRunAction
G4UserEventAction
G4UserStackingAction
G4UserTrackingAction
G4UserSteppingAction

The classes starting with **G4V** are abstract classes.
Their objects **can’t** be created.
They are there to provide a skeleton required by Geant4
User needs to **inherit these classes**, and to implement few functions which are mandatory.

Structure of main() function

Define your entry point : main() :The place where you actually registers different components of your application.

Things TODO:
1) Instantiate your RunManager
2) Instantiate your DetectorConstruction
3) Instantiate your PhysicsList
4) Instantiate your ActionInitialization
5) Run your code
Optional
6) Instantiate your Visualization Manager

```
Int main(){
G4RunManager *runManager = new
G4RunManager;
DetectorConstruction *det = new
DetectorConstruction();
G4VModularPhysicsList *physicsList = new
FTFP_BERT;
ActionInitialization *actIni = new
ActionInitialization();
runManager->SetUserInitialization(det);
runManager->SetUserInitialization(physicsList);
runManager->SetUserInitialization(actIni);
G4UImanager *UImanager =
G4UImanager::GetUIpointer();
Uimanager->ApplyCommand(“/control/execute
Run.mac”);

}
```

Run.mac

```
/run/initialize
/run/beamOn 100
```

Structure of main() function

Define your entry point : main() :The place where you actually registers different components of your application.

```
Int main(){

G4RunManager *runManager = new G4RunManager;
DetectorConstruction *det = new DetectorConstruction();
G4VModularPhysicsList *physicsList = new FTFP_BERT;
ActionInitialization *actIni = new ActionInitialization();

runManager->SetUserInitialization(det);
runManager->SetUserInitialization(physicsList);
runManager->SetUserInitialization(actIni);

G4UImanager *UImanager = G4UImanager::GetUIpointer();
Uimanager->ApplyCommand(“/control/execute Run.mac”);

}
```

Run.mac

```
/run/initialize
/run/beamOn 100
```

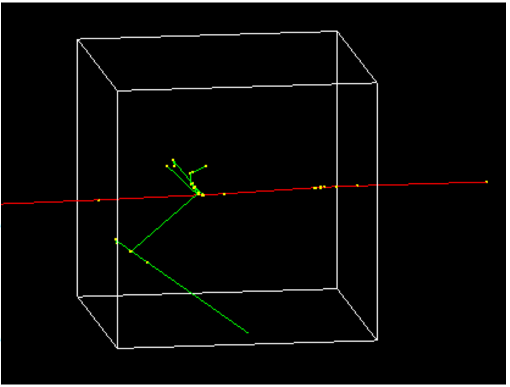
Our program is running : Where is the output ??

Geant4 runs the full simulation silently.

The required information needs to extracted.

Just to see what going on :
--> **use UI commands : *|tracking/verbose 1***

This will basically start printing all the tracking information.
--> Particle information (location, direction etc.)
--> Step information
--> Energy loss
--> Associated volume
--> TrackId



```
*********************************************************************************
****
* G4Track Information:   Particle = mu-,   Track ID = 1,   Parent ID = 0
*********************************************************************************
****
Step#     X(mm)     Y(mm)     Z(mm) KinE(MeV)  dE(MeV) StepLeng TrackLeng  NextVolume ProcName
0         0         0       100     2e+03        0        0          0       World initStep
1         0         0        50     2e+03 1.49e-24       50         50 Physical_Lead_Block Transportation
2         0         0        42  1.99e+03     9.49     7.96         58 Physical_Lead_Block muIoni
3 -0.00409    -0.105      37.3  1.98e+03     5.58      4.7       62.7 Physical_Lead_Block CoulombScat
4  -0.0242    -0.159      35.8  1.98e+03     1.63     1.52       64.2 Physical_Lead_Block muIoni
5  -0.0527    -0.219      33.9  1.98e+03     2.27     1.95       66.1 Physical_Lead_Block muIoni
6   -0.517     -1.38      -1.2  1.93e+03     40.6     35.1        101 Physical_Lead_Block muIoni
7   -0.746     -1.51     -9.38  1.91e+03     8.89     8.18        109 Physical_Lead_Block muIoni
8    -1.37     -2.15       -50  1.86e+03     49.5     40.6        150       World Transportation
9     32.3       -19    -1e+03  1.86e+03 2.82e-23      951    1.1e+03  OutOfWorld Transportation
*****************************************************************************************************
```

Geant4 Classes to get the information from the simulations

Information can be fetched at different levels, depending upon the requirements.

--> Run level information (G4UserRunAction)

--> Event level information (G4UserEventAction)

--> Step level information (G4UserSteppingAction)

--> Few more are also there.

Getting information from RunAction

```
class G4UserRunAction
{
  public:
    G4UserRunAction();
    virtual ~G4UserRunAction();

  public:
    virtual G4Run* GenerateRun();
    virtual void BeginOfRunAction(const G4Run*
aRun);
    virtual void EndOfRunAction(const G4Run*
aRun);
}
```

```
class Sim01_RunAction : public  G4UserRunAction{
public:
    Sim01_RunAction();
    ~Sim01_RunAction();
public:
    void BeginOfRunAction(const G4Run*){
        //Write your stuff here
        //Open some file for writing
        //Initialize your required datastructure
        //ROOT Tree, histogram
    }
    void EndOfRunAction(const G4Run*){
        //Write your stuff here
        //Print summary of Run
        //Close all the open resources
    }
};
```

Now just register the object of your **RunAction** in the **Build** function of your **ActionInitialization**

SetUserAction(new Sim01_RunAction);

Getting information from EventAction

```
class G4UserEventAction
{

      G4UserEventAction();
      virtual ~G4UserEventAction();
      virtual void BeginOfEventAction(const
        G4Event* anEvent);
      virtual void EndOfEventAction(const
        G4Event* anEvent);
}
```

```
class Sim01_EventAction : public  G4UserEventAction{

    Sim01_EventAction();
    ~Sim01_EventAction();
     Doubel eDep;

    void BeginOfEventAction(const G4Event* anEvent){
        //Write your stuff here
        //Initialize all event related parameter
        eDep=0;
    }
    void EndOfEventAction(const G4Event* anEvent){
        //Write your stuff here
        //Print total energy deposited
        //Use G4RunManager::GetRunManager()
     }
}
;
```

Now just register the object of your **EventAction** in the **Build** function of your **ActionInitialization**

SetUserAction(new Sim01_EventAction);

Getting information from SteppingAction

```
class G4UserSteppingAction
{
   G4UserSteppingAction();
   virtual ~G4UserSteppingAction();

   virtual void UserSteppingAction(const
G4Step*){;}
};
```

```
class Sim01_SteppingAction : public
G4UserSteppingAction{

    Sim01_SteppingAction();
    ~Sim01_SteppingAction();

    void UserSteppingAction(const G4Step *step){
        //Write your stuff here like
        //Use G4RunManager::GetRunManager()

        std::cout << step->GetLength() << std::endl;
        std::cout << step->GetTotalEnergyDeposit() <<
std::endl;
    }
};
```

Now just register the object of your **SteppingAction** in the **Build** function of your **ActionInitialization**

SetUserAction(new Sim01_SteppingAction);

Efficient scoring : Making your detector sensitive

Stepping action class process every step, irrespective of the volume

But what if you want to analyze steps which belongs to particular volume

Can be done by check the volume name before doing the processing on the step

This introduce extra burden on the simulation.

Geant4 provides a concept of sensitive detector, where the required processing is
Done only if the volume is declared as sensitive

Lets have a look at the Sensitive detector class.

Sensitive Detector : G4VSensitiveDetector

```
class G4VSensitiveDetector
{
   //Constructors
   //Destructors

G4bool ProcessHits(
                G4Step*aStep,
                G4TouchableHistory*ROhist) = 0;

void Initialize(G4HCofThisEvent*);
void EndOfEvent(G4HCofThisEvent*);
}
```

```
class MySD : public G4VSensitiveDetector
{
        //constructors
        //destructors
        virtual G4bool ProcessHits(
                        G4Step *,
                        G4TouchableHistory *){

                        //Write your stuff here
        }

        void Initialize(G4HCofThisEvent*){
                //Initialize required data members
        }
        void EndOfEvent(G4HCofThisEvent*){
                //Things to do at the end of event
        }

};
```

Making a Logical Volume Sensitive

We have created a sensitive detector class, but not yet link it to our detector volume

Now we need and Sensitive Detector Manager class : **G4SDManager**

```
G4VPhysicalVolume* Construct(){
  G4LogicalVolume myVol;  //Logical volume that we want to make sensitive
  G4SDManager *sdman = G4SDManager::GetSDMpointer(); //pointer to SDManager
  MySD *mySD = new MySD("MySensitiveDetector");  //object of Sensitive Detector class
  sdman->AddNewDetector(mySD);   // registering the Sensitive Detector with manager
  myVol->SetSensitiveDetector(mySD); //finally making the logical volume sensitive
}
```

Thanks for your attention

Classes invoked during the event loop

G4RunManager / G4VUserActionInitialization

G4RunManager

```
void SetUserAction(G4UserRunAction* userAction);
void SetUserAction(G4VUserPrimaryGeneratorAction* userAction);
void SetUserAction(G4UserEventAction* userAction);
void SetUserAction(G4UserStackingAction* userAction);
void SetUserAction(G4UserTrackingAction* userAction);
void SetUserAction(G4UserSteppingAction* userAction);
```

G4VUserActionInitialization

```
void SetUserAction(G4VUserPrimaryGeneratorAction*) const;
void SetUserAction(G4UserRunAction*) const;
void SetUserAction(G4UserEventAction*) const;
void SetUserAction(G4UserStackingAction*) const;
void SetUserAction(G4UserTrackingAction*) const;
void SetUserAction(G4UserSteppingAction*) const;
```

Sample programs

The program discussed during the presentation is available at following link.

https://github.com/rsehgal/IUCCA_tutorials

Particularly Sim09, contains everything, and you can switch ON/OFF various classes at the compile time using the flags available in CMAKE

If you have **cmake-curses-gui** installed, then you can use
ccmake . (provided you had compiled the code in the current directory)
to see various flags.