Scoring I

Makoto Asai (SLAC)
Geant4 Tutorial Course
Contents

• Retrieving information from Geant4
• Command-based scoring
• Add a new scorer/filter to command-based scoring
• Define scorers in the tracking volume
• Accumulate scores for a run
Retrieving information from Geant4
Extract useful information

- Given geometry, physics and primary track generation, Geant4 does proper physics simulation “silently”.
  - You have to add a bit of code to extract information useful to you.

- There are three ways:
  - Built-in scoring commands
    - Most commonly-used physics quantities are available.
  - Use scorers in the tracking volume
    - Create scores for each event
    - Create own Run class to accumulate scores
  - Assign `G4VSensitiveDetector` to a volume to generate “hit”.
    - Use user hooks (G4UserEventAction, G4UserRunAction) to get event / run summary
  - You may also use user hooks (G4UserTrackingAction, G4UserSteppingAction, etc.)
    - You have full access to almost all information
    - Straight-forward, but do-it-yourself

This talk
Command-based scoring
Command-based scoring

- Command-based scoring functionality offers the built-in scoring mesh and various scorers for commonly-used physics quantities such as dose, flux, etc.
  - Due to small performance overhead, it does not come by default.
- To use this functionality, access to the G4ScoringManager pointer after the instantiation of G4RunManager in your `main()`.

```cpp
#include "G4ScoringManager.hh"
int main()
{
  G4RunManager* runManager = new G4RunManager;
  G4ScoringManager* scoringManager =
    G4ScoringManager::GetScoringManager();
  ...
```

- All of the UI commands of this functionality are in `/score/` directory.
- `/examples/extended/runAndEvent/RE03`
Command-based scorers
Define a scoring mesh

- To define a scoring mesh, the user has to specify the followings.
  1. **Shape and name** of the 3D scoring mesh.
     - Currently, box and cylinder are available.
  2. **Size of the scoring mesh**.
     - Mesh size must be specified as "half width" similar to the arguments of G4Box / G4Tubs.
  3. **Number of bins** for each axes.
     - Note that too many bins causes immense memory consumption.
  4. **Optionally, position and rotation of the mesh**.
     - If not specified, the mesh is positioned at the center of the world volume without rotation.

```yaml
# define scoring mesh
/score/create/boxMesh boxMesh_1
/score/mesh/boxSize 100. 100. 100. cm
/score/mesh/nBin 30 30 30
```

- The mesh geometry can be completely independent to the real material geometry.
Scoring quantities

- A mesh may have arbitrary number of scorers. Each scorer scores one physics quantity.
  - energyDeposit * Energy deposit scorer.
  - cellCharge * Cell charge scorer.
  - cellFlux * Cell flux scorer.
  - passageCellFlux * Passage cell flux scorer.
  - doseDeposit * Dose deposit scorer.
  - nOfStep * Number of step scorer.
  - nOfSecondary * Number of secondary scorer.
  - trackLength * Track length scorer.
  - passageCellCurrent * Passage cell current scorer.
  - passageTrackLength * Passage track length scorer.
  - flatSurfaceCurrent * Flat surface current Scorer.
  - flatSurfaceFlux * Flat surface flux scorer.
  - nOfCollision * Number of collision scorer.
  - population * Population scorer.
  - nOfTrack * Number of track scorer.
  - nOfTerminatedTrack * Number of terminated tracks scorer.

/score/quantitly/xxxxx  <scorer_name>  <unit>
List of provided primitive scorers

- Concrete Primitive Scorers (See Application Developers Guide 4.4.6)
  - Track length
    - G4PSTrackLength, G4PSPassageTrackLength
  - Deposited energy
    - G4PSEnergyDeposit, G4PSDoseDeposit, G4PSChargeDeposit
  - Current/Flux
    - G4PSFlatSurfaceCurrent, G4PSSphereSurfaceCurrent, G4PSPassageCurrent, G4PSFlatSurfaceFlux, G4PSCellFlux, G4PSPassageCellFlux
  - Others
    - G4PSMinKinEAtGeneration, G4PSNofSecondary, G4PSNofStep

SurfaceCurrent: Count number of injecting particles at defined surface.

SurfaceFlux: Sum up $1/\cos(\text{angle})$ of injecting particles at defined surface.

CellFlux: Sum of $L/V$ of injecting particles in the geometrical cell.

L: Total step length in the cell.

V: Volume
Filter

- Each scorer may take a filter.
  - charged * Charged particle filter.
  - neutral * Neutral particle filter.
  - kineticEnergy * Kinetic energy filter.
    `/score/filter/kineticEnergy <fname> <eLow> <eHigh> <unit>`
  - particle * Particle filter.
    `/score/filter/particle <fname> <p1> … <pn>`
  - particleWithKineticEnergy * Particle with kinetic energy filter.
    `/score/quantity/energyDeposit  eDep  MeV`
    `/score/quantity/nOfStep  nOfStepGamma`
    `/score/filter/particle  gammaFilter  gamma`
    `/score/quantity/nOfStep  nOfStepEMinus`
    `/score/filter/particle  eMinusFilter  e-`
    `/score/quantity/nOfStep  nOfStepEPlus`
    `/score/filter/particle  ePlusFilter  e+`

Close the mesh when defining scorers is done.

Same primitive scorers with different filters may be defined.
Drawing a score

- Projection
  
  `/score/drawProjection <mesh_name> <scorer_name> <color_map>`

- Slice
  
  `/score/drawColumn <mesh_name> <scorer_name> <plane> <column> <color_map>`

- Color map
  
  - By default, linear and log-scale color maps are available.
  - Minimum and maximum values can be defined by `/score/colorMap/setMinMax` command. Otherwise, min and max values are taken from the current score.
Write scores to a file

- Single score
  /score/dumpQuantityToFile <mesh_name> <scorer_name> <file_name>

- All scores
  /score/dumpAllQuantitiesToFile <mesh_name> <file_name>

- By default, values are written in CSV.
- By creating a concrete class derived from G4VScoreWriter base class, the user can define his own file format.
  - Example in /examples/extended/runAndEvent/RE03
  - User’s score writer class should be registered to G4ScoringManager.
More than one scoring meshes

- You may define more than one scoring mesh.
  - And, you may define arbitrary number of primitive scorers to each scoring mesh.
- Mesh volumes may overlap with other meshes and/or with mass geometry.
- A step is limited on any boundary.
- Please be cautious of too many meshes, too granular meshes and/or too many primitive scorers.
  - Memory consumption
  - Computing speed
Add a new scorer/filter to command-based scorers
Scorer base class

- G4VPrimitiveScorer is the abstract base of all scorer classes.
- To make your own scorer you have to implement at least:
  - Constructor
  - Initialize()
    - Initialize G4THitsMap&lt;G4double&gt; map object
  - ProcessHits()
    - Get the physics quantity you want from G4Step, etc. and fill the map
  - Clear()
  - GetIndex()
    - Convert three copy numbers into an index of the map
- G4PSEnergyDeposit3D could be a good example.
- Create your own messenger class to define /score/quantity/&lt;your_quantity&gt; command.
  - Refer to G4ScorerQuantityMessengerQCmd class.
Filter class

- G4VSDFilter
  - Abstract base class which you can use to make your own filter class G4VSDFilter
    
    ```
    public:
        G4VSDFilter(G4String name);
        virtual ~G4VSDFilter();
    public:
        virtual G4bool Accept(const G4Step*) const = 0;
    ...
    ```

- Create your own messenger class to define /score/filter/<your_filter> command.
  - Refer to G4ScorerQuantityMessenger class.
Define scorers to the tracking volume
example

MyDetectorConstruction::Construct()
{
    G4LogicalVolume* myCellLog = new G4LogicalVolume(...);
    G4VPhysicalVolume* myCellPhys = new G4PVParametrised(...);
    G4MultiFunctionalDetector* myScorer =
        new G4MultiFunctionalDetector("myCellScorer");
    G4SDManager::GetSDMpointer()->AddNewDetector(myScorer);
    myCellLog->SetSensitiveDetector(myScorer);
    G4VPrimitiveSensitivity* totalSurfFlux = new
        G4PSFlatSurfaceFlux("TotalSurfFlux", fCurrent_In, "percm2");
    myScorer->Register(totalSurfFlux);
    G4VPrimitiveSensitivity* totalDose = new G4PSDoseDeposit("TotalDose");
    myScorer->Register(totalDose);
}

You may register arbitrary number of primitive scorers.
Keys of G4THitsMap

- All provided primitive scorer classes use `G4THitsMap<G4double>`.
- By default, the copy number is taken from the physical volume to which `G4MultiFunctionalDetector` is assigned.
  - If the physical volume is placed only once, but its (grand-)mother volume is replicated, use the second argument of the constructor of the primitive scorer to indicate the level where the copy number should be taken.
    - e.g. `G4PSCellFlux(G4Steing name, G4String& unit, G4int depth=0)`
  - If your indexing scheme is more complicated (e.g. utilizing copy numbers of more than one hierarchies), you can override the virtual method `GetIndex()` provided for all the primitive scorers.

See exampleN07

---

**Key should be taken from upper geometry hierarchy**

- Copy No 0
- Copy No 0
- Copy No 0
- Copy No 0
- Copy No 0
- Copy No 0

Scorer A

Scorer B

- If the physical volume is placed only once, but its (grand-)mother volume is replicated, use the second argument of the constructor of the primitive scorer to indicate the level where the copy number should be taken.
  - e.g. `G4PSCellFlux(G4Steing name, G4String& unit, G4int depth=0)`

See exampleN07
Creating your own scorer

- Though we provide most commonly-used scorers, you may want to create your own.
  - If you believe your requirement is quite common, just let us know, so that we will add a new scorer.
- G4VPrimitiveScorer is the abstract base class.

```cpp
class G4VPrimitiveScorer
{
public:
  G4VPrimitiveScorer(G4String name, G4int depth=0);
  virtual ~G4VPrimitiveScorer();
protected:
  virtual G4bool ProcessHits(G4Step*,
                              G4TouchableHistory*) = 0;
  virtual G4int GetIndex(G4Step*);
public:
  virtual void Initialize(G4HCofThisEvent*);
  virtual void EndOfEvent(G4HCofThisEvent*);
  virtual void clear();
  ...
};
```

- GetIndex() has already been introduced. Other four methods written in red will be discussed at “Scoring 2” talk.
G4VSDFilter

- **G4VSDFilter** can be attached to **G4VSensitiveDetector** and/or **G4VPrimitiveSensitivity** to define which kinds of tracks are to be scored.
  
  - E.g., surface flux of protons can be scored by **G4PSFlatSurfaceFlux** with a filter that accepts protons only.
MyDetectorConstruction::Construct()
{
    G4LogicalVolume* myCellLog = new G4LogicalVolume(...);
    G4PVPhysicalVolume* myCellPhys = new G4PVParametrised(...);
    G4MultiFunctionalDetector* myScorer = new G4MultiFunctionalDetector("myCellScorer");
    G4SDManager::GetSDMpointer()->AddNewDetector(myScorer);
    myCellLog->SetSensitiveDetector(myScorer);
    G4VPrimitiveSensitivity* totalSurfFlux = new G4PSFlatSurfaceFlux("TotalSurfFlux");
    myScorer->Register(totalSurfFlux);
    G4VPrimitiveSensitivity* protonSufFlux = new G4PSFlatSurfaceFlux("ProtonSurfFlux");
    G4VSDFilter* protonFilter = new G4SDParticleFilter("protonFilter");
    protonFilter->Add("proton");
    protonSurfFlux->SetFilter(protonFilter);
    myScorer->Register(protonSurfFlux);
}
Accumulate scores for a run
Class diagram

- **G4LogicalVolume**
- **G4Event**
  - has 0..1 **G4VSensitiveDetector**
  - kind of **G4MultiFunctionalDetector**
  - n **G4VPrimitiveSensitivity**
- **G4HCoFThisEvent**
  - n **G4VHitsCollection**
  - userSensitiveDetector
- **G4VHitsCollection**
  - n **G4THitsCollection**
  - userSensitiveDetector
  - n **G4THitsMap**
  - userHitsCollection or userHitsMap
  - n **userHit**
- **G4PSDoseScorer**

*Concrete class provided by G4*
*Abstract base class provided by G4*
*Template class provided by G4*
*User’s class*
Score == G4THitsMap<

- At the end of successful event, G4Event has a vector of G4THitsMap as the scores.
- Create your own Run class derived from G4Run, and implement RecordEvent(const G4Event*) virtual method. Here you can get all output of the event so that you can accumulate the sum of an event to a variable for entire run.
  - RecordEvent(const G4Event*) is automatically invoked by G4RunManager.
  - Your run class object should be instantiated in GenerateRun() method of your UserRunAction.
Customized run class

```cpp
#include "G4Run.hh"
#include "G4Event.hh"
#include "G4THitsMap.hh"

Class MyRun : public G4Run
{
public:
    MyRun();
    virtual ~MyRun();
    virtual void RecordEvent(const G4Event*);

private:
    G4int nEvent;
    G4int totalSurfFluxID, protonSurfFluxID, totalDoseID;
    G4THitsMap<G4double> totalSurfFlux;
    G4THitsMap<G4double> protonSurfFlux;
    G4THitsMap<G4double> totalDose;

public:
    … access methods …
};
```

Implement how you accumulate event data
Customized run class

MyRun::MyRun() : nEvent(0)
{
    G4SDManager* SDM = G4SDManager::GetSDMpointer();
    totalSurfFluxID = SDM->GetCollectionID("myCellScorer/TotalSurfFlux");
    protonSurfFluxID = SDM->GetCollectionID("myCellScorer/ProtonSurfFlux");
    totalDoseID = SDM->GetCollectionID("myCellScorer/TotalDose");
}

name of G4MultiFunctionalDetector object

name of G4VPrimitiveSensitivity object
Customized run class

```cpp
void MyRun::RecordEvent(const G4Event* evt)
{
    nEvent++;
    G4HCofThisEvent* HCE = evt->GetHCofThisEvent();
    G4THitsMap<G4double>* eventTotalSurfFlux = (G4THitsMap<G4double>*)(HCE->GetHC(totalSurfFluxID));
    G4THitsMap<G4double>* eventProtonSurfFlux = (G4THitsMap<G4double>*)(HCE->GetHC(protonSurfFluxID));
    G4THitsMap<G4double>* eventTotalDose = (G4THitsMap<G4double>*)(HCE->GetHC(totalDose));
    totalSurfFlux += *eventTotalSurfFlux;
    protonSurfFlux += *eventProtonSurfFlux;
    totalDose += *eventTotalDose;
}
```

No need of loops. `+=` operator is provided!
RunAction with customized run

G4Run* MyRunAction::GenerateRun()
{
    return (new MyRun());
}

void MyRunAction::EndOfRunAction(const G4Run* aRun)
{
    MyRun* theRun = (MyRun*)aRun;
    // … analyze / record / print-out your run summary
    // MyRun object has everything you need …
}

• As you have seen, to accumulate event data, you do NOT need
  – Event / tracking / stepping action classes
• All you need are your Run and RunAction classes.

Refer to exampleN07