Book For Application Developers

Release 10.3

Geant4 Collaboration

Nov 08, 2017

CONTENTS:

1 Introduction **1**
- 1.1 Scope of this manual . . . 1
- 1.2 How to use this manual . . . 1

2 Getting Started with Geant4 - Running a Simple Example **3**
- 2.1 How to Define the main() Program . . . 3
  - 2.1.1 A Sample `main()` Method . . . 3
  - 2.1.2 `G4RunManager` . . . 4
  - 2.1.3 User Initialization and Action Classes . . . 5
  - 2.1.4 `G4UImanager` and UI CommandSubmission . . . 6
  - 2.1.5 `G4cout` and `G4cerr` . . . 7
- 2.2 How to Define a Detector Geometry . . . 7
  - 2.2.1 Basic Concepts . . . 7
  - 2.2.2 Create a Simple Volume . . . 8
  - 2.2.3 Choose a Solid . . . 8
  - 2.2.4 Create a Logical Volume . . . 8
  - 2.2.5 Place a Volume . . . 9
  - 2.2.6 Create a Physical Volume . . . 9
  - 2.2.7 Coordinate Systems and Rotations . . . 9
- 2.3 How to Specify Materials in the Detector . . . 10
  - 2.3.1 General Considerations . . . 10
  - 2.3.2 Define a Simple Material . . . 10
  - 2.3.3 Define a Molecule . . . 10
  - 2.3.4 Define a Mixture by Fractional Mass . . . 11
  - 2.3.5 Define a Material from the Geant4 Material Database . . . 11
  - 2.3.6 Define a Material from the Base Material . . . 11
  - 2.3.7 Print Material Information . . . 12
  - 2.3.8 Access to Geant4 material database . . . 12
- 2.4 How to Specify Particles . . . 12
  - 2.4.1 Particle Definition . . . 12
  - 2.4.2 Range Cuts . . . 14
- 2.5 How to Specify Physics Processes . . . 15
  - 2.5.1 Physics Processes . . . 15
  - 2.5.2 Managing Processes . . . 16
  - 2.5.3 Specifying Physics Processes . . . 16
- 2.6 How to Generate a Primary Event . . . 17
  - 2.6.1 Generating Primary Events . . . 17
  - 2.6.2 G4VPrimaryGenerator . . . 19
- 2.7 Geant4 General Particle Source . . . 20
  - 2.7.1 Introduction . . . 20

2.7.2 Configuration . . . 20
2.7.3 Macro Commands . . . 24
2.7.4 Example Macro File . . . 29
2.8 How to Make an Executable Program . . . 31
2.8.1 Using CMake to build Applications: Geant4Config.cmake . . . 31
2.8.2 Using Geant4Make to build Applications: binmake.gmk . . . 38
2.9 How to Set Up an Interactive Session . . . 40
2.9.1 Introduction . . . 40
2.9.2 A Short Description of Available Interfaces . . . 41
2.9.3 How to Select Interface in Your Applications . . . 42
2.10 How to Execute a Program . . . 43
2.10.1 Introduction . . . 43
2.10.2 'Hard-coded' Batch Mode . . . 44
2.10.3 Batch Mode with Macro File . . . 44
2.10.4 Interactive Mode Driven by Command Lines . . . 45
2.10.5 General Case . . . 47
2.11 How to Visualize the Detector and Events . . . 49
2.11.1 Introduction . . . 49
2.11.2 Visualization Drivers . . . 49
2.11.3 How to Incorporate Visualization Drivers into an Executable . . . 49
2.11.4 Writing the `main()` Method to Include Visualization . . . 50
2.11.5 Sample Visualization Sessions . . . 51
2.11.6 For More Information on Geant4 Visualization . . . 51

3 Toolkit Fundamentals 53
3.1 Class Categories and Domains . . . 53
3.1.1 What is a class category? . . . 53
3.1.2 Class categories in Geant4 . . . 53
3.2 Global Usage Classes . . . 55
3.2.1 Signature of Geant4 classes . . . 55
3.2.2 The *HEPRandom* module in CLHEP . . . 56
3.2.3 The *HEPNumerics* module . . . 59
3.2.4 General management classes . . . 59
3.3 System of units . . . 61
3.3.1 Basic units . . . 61
3.3.2 Input your data . . . 61
3.3.3 Output your data . . . 62
3.3.4 Introduce new units . . . 62
3.3.5 Print the list of units . . . 62
3.4 Run . . . 63
3.4.1 Basic concept of *Run* . . . 63
3.4.2 Geant4 as a state machine . . . 65
3.4.3 User's hook for state change . . . 65
3.4.4 Customizing the Run Manager . . . 66
3.4.5 Managing worker thread . . . 68
3.5 Event . . . 68
3.5.1 Representation of an event . . . 68
3.5.2 Structure of an event . . . 68
3.5.3 Mandates of `G4EventManager` . . . 68
3.5.4 Stacking mechanism . . . 69
3.6 Event Generator Interface . . . 69
3.6.1 Structure of a primary event . . . 69
3.6.2 Interface to a primary generator . . . 70
3.6.3 Event overlap using multiple generators . . . 72

3.7 Event Biasing Techniques . . . 72
3.7.1 Scoring, Geometrical Importance Sampling and Weight Roulette . . . 72
3.7.2 Physics Based Biasing . . . 79
3.7.3 Adjoint/Reverse Monte Carlo . . . 83
3.7.4 Generic Biasing . . . 88

4 Detector Definition and Response **95**
4.1 Geometry . . . 95
4.1.1 Introduction . . . 95
4.1.2 Solids . . . 95
4.1.3 Logical Volumes . . . 117
4.1.4 Physical Volumes . . . 119
4.1.5 Touchables: Uniquely Identifying a Volume . . . 129
4.1.6 Creating an Assembly of Volumes . . . 131
4.1.7 Reflecting Hierarchies of Volumes . . . 133
4.1.8 The Geometry Navigator . . . 135
4.1.9 A Simple Geometry Editor . . . 141
4.1.10 Converting Geometries from Geant3.21 . . . 143
4.1.11 Detecting Overlapping Volumes . . . 144
4.1.12 Dynamic Geometry Setups . . . 147
4.1.13 Importing XML Models Using GDML . . . 148
4.1.14 Importing ASCII Text Models . . . 148
4.1.15 Saving geometry tree objects in binary format . . . 149
4.2 Material . . . 149
4.2.1 General considerations . . . 149
4.2.2 Introduction to the Classes . . . 149
4.2.3 Recipes for Building Elements and Materials . . . 151
4.2.4 The Tables . . . 154
4.3 Electromagnetic Field . . . 154
4.3.1 An Overview of Propagation in a Field . . . 154
4.3.2 Practical Aspects . . . 155
4.3.3 Spin Tracking . . . 161
4.4 Hits . . . 162
4.4.1 Hit . . . 162
4.4.2 Sensitive detector . . . 164
4.4.3 G4SDManager . . . 164
4.4.4 `G4MultiFunctionalDetector` and `G4VPrimitiveScorer` . . . 165
4.4.5 Concrete classes of `G4VPrimitiveScorer` . . . 168
4.4.6 `G4VSDFilter` and its derived classes . . . 169
4.4.7 Muiltiple sensitive detectors associated to a single logical-volume . . . 170
4.5 Digitization . . . 170
4.5.1 Digi . . . 170
4.5.2 Digitizer module . . . 171
4.6 Object Persistency . . . 172
4.6.1 Persistency in Geant4 . . . 172
4.6.2 Using Root-I/O for persistency of Geant4 objects . . . 172
4.7 Parallel Geometries . . . 173
4.7.1 A parallel world . . . 173
4.7.2 Defining a parallel world . . . 173
4.7.3 Layered mass geometry . . . 175
4.8 Command-based scoring . . . 176
4.8.1 Introduction . . . 176
4.8.2 Defining a scoring mesh . . . 176
4.8.3 Drawing scores . . . 177

4.8.4 Writing scores to a file . . . 177

5 Tracking and Physics **179**
5.1 Tracking . . . 179
5.1.1 Basic Concepts . . . 179
5.1.2 Access to Track and Step Information . . . 180
5.1.3 Handling of Secondary Particles . . . 182
5.1.4 User Actions . . . 182
5.1.5 Verbose Outputs . . . 182
5.1.6 Trajectory and Trajectory Point . . . 182
5.2 Physics Processes . . . 184
5.2.1 Electromagnetic Interactions . . . 186
5.2.2 Hadronic Interactions . . . 199
5.2.3 Particle Decay Process . . . 210
5.2.4 Gamma-nuclear and Lepto-nuclear Processes . . . 212
5.2.5 Optical Photon Processes . . . 212
5.2.6 Parameterization . . . 221
5.2.7 Transportation Process . . . 227
5.3 Particles . . . 227
5.3.1 Basic concepts . . . 227
5.3.2 Definition of a particle . . . 228
5.3.3 Dynamic particle . . . 231
5.4 Production Threshold versus Tracking Cut . . . 231
5.4.1 General considerations . . . 231
5.4.2 Set production threshold (`SetCut` methods) . . . 232
5.4.3 Apply cut . . . 232
5.4.4 Why produce secondaries below threshold in some processes? . . . 232
5.4.5 Cuts in stopping range or in energy? . . . 233
5.4.6 Summary . . . 233
5.4.7 Special tracking cuts . . . 233
5.5 Cuts per Region . . . 234
5.5.1 General Concepts . . . 234
5.5.2 Default Region . . . 235
5.5.3 Assigning Production Cuts to a Region . . . 235
5.6 Physics Table . . . 235
5.6.1 General Concepts . . . 235
5.6.2 Material-Cuts Couple . . . 236
5.6.3 File I/O for the Physics Table . . . 236
5.6.4 Building the Physics Table . . . 236
5.7 User Limits . . . 237
5.7.1 General Concepts . . . 237
5.7.2 Processes co-working with G4UserLimits . . . 237
5.8 Track Error Propagation . . . 238
5.8.1 Physics . . . 238
5.8.2 Trajectory state . . . 238
5.8.3 Trajectory state error . . . 240
5.8.4 Targets . . . 240
5.8.5 Managing the track propagation . . . 241
5.8.6 Limiting the step . . . 242
5.9 Exotic Physics . . . 243
5.9.1 Physics . . . 243
5.9.2 Material . . . 243
5.9.3 Geometry . . . 243

6 User Actions **245**
6.1 User Actions . . . 245
6.2 Mandatory User Actions and Initializations . . . 245
6.2.1 G4VUserDetectorConstruction . . . 245
6.2.2 Physics Lists . . . 246
6.2.3 User Action Initialization . . . 249
6.3 Optional User Actions . . . 250
6.3.1 Usage of User Actions . . . 250
6.3.2 Killing Tracks in User Actions and Energy Conservation . . . 254
6.4 User Information Classes . . . 254
6.4.1 G4VUserEventInformation . . . 254
6.4.2 G4VUserTrackInformation . . . 254
6.4.3 G4VUserPrimaryVertexInformation and G4VUserPrimaryTrackInformation . . . 255
6.4.4 G4VUserRegionInformation . . . 255
6.5 Multiple User Actions . . . 256
6.5.1 Exceptions . . . 257

7 Control **259**
7.1 Built-in Commands . . . 259
7.2 User Interface - Defining New Commands . . . 259
7.2.1 G4UImessenger . . . 259
7.2.2 G4UIcommand and its derived classes . . . 260
7.2.3 An example messenger . . . 263
7.2.4 How to control the output of G4cout/G4cerr . . . 266

8 Visualization **269**
8.1 Introduction to Visualization . . . 269
8.1.1 What Can be Visualized . . . 269
8.1.2 You have a Choice of Visualization Drivers . . . 269
8.1.3 Choose the Driver that Meets Your Needs . . . 271
8.1.4 Controlling Visualization . . . 272
8.1.5 Visualization Details . . . 272
8.2 Adding Visualization to Your Executable . . . 273
8.2.1 Installing Visualization Drivers . . . 273
8.2.2 How to Realize Visualization Drivers in an Executable . . . 273
8.2.3 Visualization Manager . . . 274
8.2.4 How to Write the main() Function . . . 274
8.3 The Visualization Drivers . . . 276
8.3.1 Availability of drivers on the supported systems . . . 276
8.3.2 OpenGL . . . 277
8.3.3 Qt . . . 278
8.3.4 OpenInventor . . . 278
8.3.5 OpenInventor Extended Viewer . . . 279
8.3.6 HepRepFile . . . 280
8.3.7 HepRepXML . . . 281
8.3.8 DAWN . . . 283
8.3.9 Remote Visualization with the DAWN-Network Driver . . . 283
8.3.10 VRML . . . 284
8.3.11 RayTracer . . . 286
8.3.12 gMocren . . . 286
8.3.13 Wt . . . 287
8.3.14 Visualization of detector geometry tree . . . 290
8.3.15 GAG Tree . . . 291
8.3.16 XML Tree . . . 291

8.4 Controlling Visualization from Commands . . . 293
8.4.1 Scene, scene handler, and viewer . . . 293
8.4.2 Create a scene handler and a viewer: /vis/open command . . . 293
8.4.3 Create an empty scene: /vis/scene/create command . . . 294
8.4.4 Visualization of a physical volume: /vis/drawVolume command . . . 294
8.4.5 Visualization of a logical volume: /vis/specify command . . . 295
8.4.6 Visualization of trajectories: /vis/scene/add/trajectories command . . . 295
8.4.7 Visualization of hits: /vis/scene/add/hits command . . . 296
8.4.8 Visualization of Scored Data . . . 296
8.4.9 HepRep Attributes for Hits . . . 296
8.4.10 Basic camera workings: /vis/viewer/ commands . . . 297
8.4.11 Declare the end of visualization for flushing: /vis/viewer/flush command . . . 299
8.4.12 End of Event Action and End of Run Action: /vis/viewer/endOfEventAction and /vis/viewer/endOfRunAction commands . . . 299
8.4.13 HepRep Attributes for Trajectories . . . 299
8.4.14 How to save a view. . . . 300
8.4.15 How to save a view to an image file . . . 300
8.4.16 Culling . . . 301
8.4.17 Cut view . . . 302
8.4.18 Multithreading commands . . . 303
8.5 Controlling Visualization from Compiled Code . . . 303
8.5.1 G4VVisManager . . . 303
8.5.2 Visualization of detector components . . . 304
8.5.3 Visualization of trajectories . . . 304
8.5.4 Enhanced trajectory drawing . . . 305
8.5.5 HepRep Attributes for Trajectories . . . 305
8.5.6 Visualization of hits . . . 305
8.5.7 HepRep Attributes for Hits . . . 308
8.5.8 Visualization of text . . . 308
8.5.9 Visualization of polylines and tracking steps . . . 308
8.5.10 Visualization User Action . . . 310
8.5.11 Standalone Visualization . . . 311
8.6 Visualization Attributes . . . 311
8.6.1 Visibility . . . 311
8.6.2 Colour . . . 312
8.6.3 Forcing attributes . . . 314
8.6.4 Other attributes . . . 315
8.6.5 Constructors of G4VisAttributes . . . 315
8.6.6 How to assign G4VisAttributes to a logical volume . . . 316
8.6.7 Additional User-Defined Attributes . . . 316
8.7 Enhanced Trajectory Drawing . . . 317
8.7.1 Default Configuration . . . 317
8.7.2 Trajectory Drawing Models . . . 318
8.7.3 Controlling from Commands . . . 319
8.7.4 Controlling from Compiled Code . . . 321
8.7.5 Drawing by time . . . 321
8.8 Trajectory Filtering . . . 322
8.8.1 Controlling from Commands . . . 323
8.8.2 Example commands . . . 323
8.8.3 Hit and Digi Filtering . . . 324
8.9 Polylines, Markers and Text . . . 324
8.9.1 Polylines . . . 324
8.9.2 Markers . . . 325
8.9.3 Text . . . 326

8.10 Making a Movie . . . 327
8.10.1 Using "/vis/viewer/interpolate". . . . 328
8.10.2 OGLX . . . 328
8.10.3 Qt . . . 329
8.10.4 DAWNFILE . . . 329
8.10.5 RayTracerX . . . 330

9 Analysis **331**
9.1 Introduction . . . 331
9.2 Analysis Manager Classes . . . 331
9.2.1 Analysis Manager . . . 332
9.2.2 Files handling . . . 332
9.2.3 Histograms . . . 333
9.2.4 Profiles . . . 339
9.2.5 Plotting . . . 342
9.2.6 Ntuples . . . 342
9.2.7 Parallel Processing . . . 344
9.2.8 Coexistence of Several Managers . . . 345
9.2.9 Supported Features and Limitations . . . 345
9.3 Analysis Reader Classes . . . 346
9.3.1 Analysis Reader . . . 346
9.3.2 Files handling . . . 347
9.3.3 Histograms and Profiles . . . 347
9.3.4 Ntuples . . . 348
9.4 Accumulables . . . 349
9.4.1 G4Accumulable<T> . . . 349
9.4.2 User defined accumulables . . . 350
9.5 g4tools . . . 351
9.5.1 g4tools package . . . 352
9.5.2 User API . . . 352

10 Examples **355**
10.1 Introduction . . . 355
10.2 Basic Examples . . . 355
10.2.1 Basic Examples Summary . . . 355
10.2.2 Basic Examples Macros . . . 358
10.2.3 Multi-threading . . . 359
10.2.4 Example B1 . . . 359
10.2.5 Example B2 . . . 360
10.2.6 Example B3 . . . 361
10.2.7 Example B4 . . . 362
10.2.8 Example B5 . . . 363
10.3 Extended Examples . . . 365
10.3.1 Extended Example Summary . . . 365
10.4 Advanced Examples . . . 370
10.5 Novice Examples . . . 371

11 Appendix **373**
11.1 CLHEP Foundation Library . . . 373
11.2 Geant4Config.cmake CMake Config File . . . 373
11.2.1 Usage of Geant4Config.cmake . . . 373
11.2.2 Going further with CMake . . . 377
11.2.3 Building an Application against a Build of Geant4 . . . 377
11.3 GNUMake System: Makefiles and Environment Variables . . . 378

11.3.1 Geant4Make System . . . 378
11.3.2 Environment variables . . . 379
11.3.3 Linking External Libraries with Geant4 . . . 382
11.4 Development and Debug Tools . . . 383
11.4.1 Unix/Linux . . . 383
11.4.2 Windows . . . 384
11.5 Python Interface . . . 384
11.5.1 Installation . . . 384
11.5.2 Using Geant4Py . . . 384
11.5.3 Site-modules . . . 386
11.5.4 Examples . . . 387
11.6 Geant4 Material Database . . . 387
11.6.1 Simple Materials (Elements) . . . 387
11.6.2 NIST Compounds . . . 389
11.6.3 HEP and Nuclear Materials . . . 401
11.6.4 Space (ISS) Materials . . . 401
11.6.5 Bio-Chemical Materials . . . 402

Bibliography **405**

CHAPTER

ONE

INTRODUCTION

1.1 Scope of this manual

The User's Guide for Application Developers is the first manual the reader should consult when learning about Geant4 or developing a Geant4-based detector simulation program. This manual is designed to:

- introduce the first-time user to the Geant4 object-oriented detector simulation toolkit,
- provide a description of the available tools and how to use them, and
- supply the practical information required to develop and run simulation applications which may be used in real experiments.

This manual is intended to be an overview of the toolkit, rather than an exhaustive treatment of it. Related physics discussions are not included unless required for the description of a particular tool. Detailed discussions of the physics included in Geant4 can be found in the Physics Reference Manual. Details of the design and functionality of the Geant4 classes can be found in the User's Guide for Toolkit Developers.

Geant4 is a completely new detector simulation toolkit written in the C++ language. The reader is assumed to have a basic knowledge of object-oriented programming using C++. No knowledge of earlier FORTRAN versions of Geant is required. Although Geant4 is a fairly complicated software system, only a relatively small part of it needs to be understood in order to begin developing detector simulation applications.

1.2 How to use this manual

A very basic introduction to Geant4 is presented in **Chapter 2, "Getting Started with Geant4 - Running a Simple Example"**. It is a recipe for writing and running a simple Geant4 application program. New users of Geant4 should read this chapter first. It is strongly recommended that this chapter be read in conjunction with a Geant4 system installed and running on your computer. It is helpful to run the provided examples as they are discussed in the manual. To install the Geant4 system on your computer, please refer to the Installation Guide for Setting up Geant4 in Your Computing Environment.

Chapter 3, "Toolkit Fundamentals" discusses generalGeant4 issues such as class categories and the physical units system. It goes on to discuss runs and events, which are the basic units of a simulation.

Chapter 4, "Detector Definition and Response" describes how to construct a detector from customized materials and geometric shapes, and embed it in electromagnetic fields. It also describes how to make the detector sensitive to particles passing through it and how to store this information.

How particles are propagated through a material is treated in **Chapter 5, "Tracking and Physics"**. The Geant4 "philosophy" of particle tracking is presented along with summaries of the physics processes provided by the toolkit. The definition and implementation of Geant4 particles is discussed and a list of particle properties is provided.

Chapter 6, "User Actions" is a description of the "user hooks" by which the simulation code may be customized to perform special tasks.

Chapter 7, "Communication and Control" provides a summary of the commands available to the user to control the execution of the simulation. After Chapter 2, Chapters 6 and 7 are of formeost importance to the new application developer.

The display of detector geometry, tracks and events may be incorporated into a simulation application by using the tools described in **Chapter 8, "Visualization"**.

Chapter 9, "Examples" provides a set of basic, novice, extended and advanced simulation codes which may be compiled and run "as is" from the Geant4 source code. These examples may be used as educational tools or as base code from which more complex applications are developed.

CHAPTER

TWO

GETTING STARTED WITH GEANT4 - RUNNING A SIMPLE EXAMPLE

2.1 How to Define the main() Program

2.1.1 A Sample `main()` Method

The contents of `main()` will vary according to the needs of a given simulation application and therefore must be supplied by the user. The Geant4 toolkit does not provide a `main()` method, but a sample is provided here as a guide to the beginning user. *This example* is the simplest example of `main()` required to build a simulation program.

Listing 2.1: Simplest example of main()

```
#include "G4RunManager.hh"
#include "G4UImanager.hh"

#include "ExG4DetectorConstruction01.hh"
#include "ExG4PhysicsList00.hh"
#include "ExG4ActionInitialization01.hh"

int main()
{
  // construct the default run manager
  G4RunManager* runManager = new G4RunManager;

  // set mandatory initialization classes
  runManager->SetUserInitialization(new ExG4DetectorConstruction01);
  runManager->SetUserInitialization(new ExG4PhysicsList00);
  runManager->SetUserInitialization(new ExG4ActionInitialization01);

  // initialize G4 kernel
  runManager->Initialize();

  // get the pointer to the UI manager and set verbosities
  G4UImanager* UI = G4UImanager::GetUIpointer();
  UI->ApplyCommand("/run/verbose 1");
  UI->ApplyCommand("/event/verbose 1");
  UI->ApplyCommand("/tracking/verbose 1");

  // start a run
  int numberOfEvent = 3;
  runManager->BeamOn(numberOfEvent);

  // job termination
  delete runManager;
  return 0;
}
```

The `main()` method is implemented by two toolkit classes, `G4RunManager` and `G4UImanager`, and three classes, `ExG4DetectorConstruction01`, `ExG4PhysicsList00` and `ExG4ActionInitialization01`, which

are derived from toolkit classes. Each of these are explained in the following sections.

2.1.2 G4RunManager

The first thing `main()` must do is create an instance of the `G4RunManager` class. This is the only manager class in the Geant4 kernel which should be explicitly constructed in the user's `main()`. It controls the flow of the program and manages the event loop(s) within a run. If the user wants to make the simulation code multi-threaded, `G4MTRunManager` should be instantiated instead of G4RunManager.

When `G4RunManager` is created, the other major manager classes are also created. They are deleted automatically when `G4RunManager` is deleted. The run manager is also responsible for managing initialization procedures, including methods in the user initialization classes. Through these the run manager must be given all the information necessary to build and run the simulation, including

1. how the detector should be constructed,
2. all the particles and all the physics processes to be simulated,
3. how the primary particle(s) in an event should be produced, and
4. any additional requirements of the simulation.

In the sample `main()` the lines

```
runManager->SetUserInitialization(new ExG4DetectorConstruction01);
runManager->SetUserInitialization(new ExG4PhysicsList00);
runManager->SetUserInitialization(new ExG4ActionInitialization01);
```

create objects which specify the detector geometry, physics processes and primary particle, respectively, and pass their pointers to the run manager. `ExG4DetectorConstruction01` is an example of a user initialization class which is derived from `G4VUserDetectorConstruction`. This is where the user describes the entire detector setup, including

- its geometry,
- the materials used in its construction,
- a definition of its sensitive regions and
- the readout schemes of the sensitive regions.

Similarly `ExG4PhysicsList01` is derived from `G4VUserPhysicsList` and requires the user to define

- the particles to be used in the simulation,
- all the physics processes to be simulated.

User can also override the default implementation for

- the range cuts for these particles and

Also `ExG4ActionInitialization01` is derived from `G4VUserActionInitialization` and requires the user to define

- so-called user action classes (see next section) that are invoked during the simulation,
- which includes one mandatory user action to define the primary particles.

The next instruction

```
runManager->Initialize();
```

performs the detector construction, creates the physics processes, calculates cross sections and otherwise sets up the run. The final run manager method in `main()`

```
int numberOfEvent = 3;
runManager->beamOn(numberOfEvent);
```

begins a run of three sequentially processed events. The `beamOn()` method may be invoked any number of times within `main()` with each invocation representing a separate run. Once a run has begun neither the detector setup nor the physics processes may be changed. They may be changed between runs, however, as described in *Customizing the Run Manager*. More information on `G4RunManager` in general is found in *Run*.

As mentioned above, other manager classes are created when the run manager is created. One of these is the user interface manager, `G4UImanager`. In `main()` a pointer to the interface manager must be obtained

```
G4UImanager* UI = G4UImanager::getUIpointer();
```

in order for the user to issue commands to the program. In the present example the `applyCommand()` method is called three times to direct the program to print out information at the run, event and tracking levels of simulation. A wide range of commands is available which allows the user detailed control of the simulation. A list of these commands can be found in *Built-in Commands*.

2.1.3 User Initialization and Action Classes

User Classes

There are two kinds of user classes, user initialization classes and user action classes. User initialization classes are used during the initialization phase, while user action classes are used during the run. User initialization classes should be directly set to `G4RunManager` through `SetUserInitialization()` method, while user action classes should de defined in `G4VUserActionInitialization` class.

User Initialization Classes

All three user initialization classes are mandatory. They must be derived from the abstract base classes provided by Geant4:

- `G4VUserDetectorConstruction`
- `G4VUserPhysicsList`
- `G4VUserActionInitialization`

Geant4 does not provide default behavior for these classes. `G4RunManager` checks for the existence of these mandatory classes when the `Initialize()` and `BeamOn()` methods are invoked.

As mentioned in the previous section, `G4VUserDetectorConstruction` requires the user to define the detector and `G4VUserPhysicsList` requires the user to define the physics. Detector definition will be discussed in Sections *How to Define a Detector Geometry* and *How to Specify Materials in the Detector*. Physics definition will be discussed in *How to Specify Particles* and *How to Specify Physics Processes*. The user action `G4VUserPrimaryGeneratorAction` requires that the initial event state be defined. Primary event generation will be discussed in *How to Make an Executable Program*.

`G4VUserActionInitialization` should include at least one mandatory user action class `G4VUserPrimaryGeneratorAction`. All user action classes are descrived in the next section.

Listing 2.2: Simplest example of ExG4ActionInitialization01

```
#include "ExG4ActionInitialization01.hh"
#include "ExG4PrimaryGeneratorAction01.hh"

void ExG4ActionInitialization01::Build() const
{
  SetUserAction(new ExG4PrimaryGeneratorAction01);
}
```

User Action Classes

`G4VUserPrimaryGeneratorAction` is a mandatory class the user has to provide. It creates an instance of a primary particle generator. `ExG4PrimaryGeneratorAction01` is an example of a user action class which is derived from `G4VUserPrimaryGeneratorAction`. In this class the user must describe the initial state of the primary event. This class has a public virtual method named `GeneratePrimaries()` which will be invoked at the beginning of each event. Details will be given in *How to Generate a Primary Event*. Note that Geant4 does not provide any default behavior for generating a primary event.

Geant4 provides additional five user hook classes:

- `G4UserRunAction`
- `G4UserEventAction`
- `G4UserStackingAction`
- `G4UserTrackingAction`
- `G4UserSteppingAction`

These optional user action classes have several virtual methods which allow the specification of additional procedures at all levels of the simulation application. Details of the user initialization and action classes are provided in *User Actions*.

2.1.4 `G4UImanager` and UI CommandSubmission

Geant4 provides a category named **intercoms**. `G4UImanager` is the manager class of this category. Using the functionalities of this category, you can invoke **set** methods of class objects of which you do not know the pointer. In *This example*, the verbosities of various Geant4 manager classes are set. Detailed mechanism description and usage of **intercoms** will be given in the next chapter, with a list of available commands. Command submission can be done all through the application.

Listing 2.3: An example of main() using interactive terminal and visualization. Code modified from the previous example shown in blue.

```
#include "G4RunManager.hh"
#include "G4UImanager.hh"

#ifdef G4UI_USE
 #include "G4VisExecutive.hh"
#endif

#include "ExG4DetectorConstruction01.hh"
#include "ExG4PhysicsList00.hh"
#include "ExG4PrimaryGeneratorAction01.hh"

int main()
{
  // construct the default run manager
  G4RunManager* runManager = new G4RunManager;

  // set mandatory initialization classes
  runManager->SetUserInitialization(new ExG4DetectorConstruction01);
  runManager->SetUserInitialization(new ExG4PhysicsList00);

  // set mandatory user action class
  runManager->SetUserAction(new ExG4PrimaryGeneratorAction01);

  // initialize G4 kernel
  runManager->Initialize();

 // Get the pointer to the User Interface manager
 G4UImanager* UImanager = G4UImanager::GetUIpointer();
```

```
 if ( argc == 1 ) {
   // interactive mode : define UI session
#ifdef G4UI_USE
    G4UIExecutive* ui = new G4UIExecutive(argc, argv);
    UImanager->ApplyCommand("/control/execute init.mac");
    ui->SessionStart();
    delete ui;
#endif
 }
 else {
   // batch mode
   G4String command = "/control/execute ";
   G4String fileName = argv[1];
   UImanager->ApplyCommand(command+fileName);
 }

  // job termination
  delete runManager;
  return 0;
}
```

2.1.5 G4cout and G4cerr

Although not yet included in the above examples, output streams will be needed. G4cout and G4cerr are **iostream** objects defined by Geant4. The usage of these objects is exactly the same as the ordinary *cout* and *cerr*, except that the output streams will be handled by G4UImanager. Thus, output strings may be displayed on another window or stored in a file. Manipulation of these output streams will be described in *How to control the output of G4cout/G4cerr*. These objects should be used instead of the ordinary *cout* and *cerr*.

2.2 How to Define a Detector Geometry

2.2.1 Basic Concepts

A detector geometry in Geant4 is made of a number of volumes. The largest volume is called the **World** volume. It must contain, with some margin, all other volumes in the detector geometry. The other volumes are created and placed inside previous volumes, included in the World volume. The most simple (and efficient) shape to describe the World is a box.

Each volume is created by describing its shape and its physical characteristics, and then placing it inside a containing volume.

When a volume is placed within another volume, we call the former volume the daughter volume and the latter the mother volume. The coordinate system used to specify where the daughter volume is placed, is the coordinate system of the mother volume.

To describe a volume's shape, we use the concept of a solid. A solid is a geometrical object that has a shape and specific values for each of that shape's dimensions. A cube with a side of 10 centimeters and a cylinder of radius 30 cm and length 75 cm are examples of solids.

To describe a volume's full properties, we use a logical volume. It includes the geometrical properties of the solid, and adds physical characteristics: the material of the volume; whether it contains any sensitive detector elements; the magnetic field; etc.

We have yet to describe how to position the volume. To do this you create a physical volume, which places a copy of the logical volume inside a larger, containing, volume.

2.2.2 Create a Simple Volume

What do you need to do to create a volume?

- Create a solid.
- Create a logical volume, using this solid, and adding other attributes.

Each of the volume types (solid, logical, and physical) has an associated registry (*VolumeStore*) which contains a list of all the objects of that type constructed so far. The registries will automatically delete those objects when requested; users should not deleted geometry objects manually.

2.2.3 Choose a Solid

To create a simple box, you only need to define its name and its extent along each of the Cartesian axes.

Listing 2.4: Creating a box.

```
  G4double world_hx = 3.0*m;
  G4double world_hy = 1.0*m;
  G4double world_hz = 1.0*m;

  G4Box* worldBox
     = new G4Box("World", world_hx, world_hy, world_hz);
```

This creates a box named "World" with the extent from -3.0 meters to +3.0 meters along the X axis, from -1.0 to 1.0 meters in Y, and from -1.0 to 1.0 meters in Z. Note that the G4Box constructor takes as arguments the halfs of the total box size.

It is also very simple to create a cylinder. To do this, you can use the `G4Tubs` class.

Listing 2.5: Creating a cylinder.

```
  G4double innerRadius = 0.*cm;
  G4double outerRadius = 60.*cm;
  G4double hz = 25.*cm;
  G4double startAngle = 0.*deg;
  G4double spanningAngle = 360.*deg;

  G4Tubs* trackerTube
    = new G4Tubs("Tracker",
                 innerRadius,
                 outerRadius,
                 hz,
                 startAngle,
                 spanningAngle);
```

This creates a full cylinder, named "Tracker", of radius 60 centimeters and length 50 cm (the hz parameter represents the half length in Z).

2.2.4 Create a Logical Volume

To create a logical volume, you must start with a solid and a material. So, using the box created above, you can create a simple logical volume filled with argon gas (see *How to Specify Materials in the Detector*) by entering:

```
G4LogicalVolume* worldLog
   = new G4LogicalVolume(worldBox, Ar, "World");
```

This logical volume is named "World".

Similarly we create a logical volume with the cylindrical solid filled with aluminium

```
G4LogicalVolume* trackerLog
  = new G4LogicalVolume(trackerTube, Al, "Tracker");
```

and named “Tracker”.

2.2.5 Place a Volume

How do you place a volume? You start with a logical volume, and then you decide the already existing volume inside of which to place it. Then you decide where to place its center within that volume, and how to rotate it. Once you have made these decisions, you can create a physical volume, which is the placed instance of the volume, and embodies all of these attributes.

2.2.6 Create a Physical Volume

You create a physical volume starting with your logical volume. A physical volume is simply a placed instance of the logical volume. This instance must be placed inside a mother logical volume. For simplicity it is unrotated:

Listing 2.6: A simple physical volume.

```
G4double pos_x = -1.0*meter;
G4double pos_y =  0.0*meter;
G4double pos_z =  0.0*meter;

G4VPhysicalVolume* trackerPhys
  = new G4PVPlacement(0,                        // no rotation
                      G4ThreeVector(pos_x, pos_y, pos_z),
                                                // translation position
                      trackerLog,               // its logical volume
                      "Tracker",                // its name
                      worldLog,                 // its mother (logical) volume
                      false,                    // no boolean operations
                      0);                       // its copy number
```

This places the logical volume `trackerLog` at the origin of the mother volume `worldLog`, shifted by one meter along X and unrotated. The resulting physical volume is named “Tracker” and has a copy number of 0.

An exception exists to the rule that a physical volume must be placed inside a mother volume. That exception is for the World volume, which is the largest volume created, and which contains all other volumes. This volume obviously cannot be contained in any other. Instead, it must be created as a `G4PVPlacement` with a null mother pointer. It also must be unrotated, and it must be placed at the origin of the global coordinate system.

Generally, it is best to choose a simple solid as the World volume, the G4Box solid type is used in all basic examples.

2.2.7 Coordinate Systems and Rotations

In Geant4, the rotation matrix associated to a placed physical volume represents the rotation of the reference system of this volume with respect to its mother.

A rotation matrix is normally constructed as in CLHEP, by instantiating the identity matrix and then applying a rotation to it. This is also demonstrated in Example B3.

2.3 How to Specify Materials in the Detector

2.3.1 General Considerations

In nature, general materials (chemical compounds, mixtures) are made of elements, and elements are made of isotopes. Therefore, these are the three main classes designed in Geant4. Each of these classes has a table as a static data member, which is for keeping track of the instances created of the respective classes. All three objects automatically register themselves into the corresponding table on construction, and should never be deleted in user code.

The `G4Element` class describes the properties of the atoms:

- atomic number,
- number of nucleons,
- atomic mass,
- shell energy,
- as well as quantities such as cross sections per atom, etc.

The `G4Material` class describes the macroscopic properties of matter:

- density,
- state,
- temperature,
- pressure,
- as well as macroscopic quantities like radiation length, mean free path, dE/dx, etc.

The `G4Material` class is the one which is visible to the rest of the toolkit, and is used by the tracking, the geometry, and the physics. It contains all the information relative to the eventual elements and isotopes of which it is made, at the same time hiding the implementation details.

2.3.2 Define a Simple Material

In the example below, liquid argon is created, by specifying its name, density, mass per mole, and atomic number.

Listing 2.7: Creating liquid argon.

```
G4double z, a, density;
density = 1.390*g/cm3;
a = 39.95*g/mole;

G4Material* lAr = new G4Material(name="liquidArgon", z=18., a, density);
```

The pointer to the material, *lAr*, will be used to specify the matter of which a given logical volume is made:

```
G4LogicalVolume* myLbox = new G4LogicalVolume(aBox,lAr,"Lbox",0,0,0);
```

2.3.3 Define a Molecule

In the example below, the water, *H2O*, is built from its components, by specifying the number of atoms in the molecule.

Listing 2.8: Creating water by defining its molecular components.

```
G4double z, a, density;
G4String name, symbol;
G4int ncomponents, natoms;

a = 1.01*g/mole;
G4Element* elH  = new G4Element(name="Hydrogen",symbol="H" , z= 1., a);
```

```
a = 16.00*g/mole;
G4Element* elO  = new G4Element(name="Oxygen"  ,symbol="O" , z= 8., a);

density = 1.000*g/cm3;
G4Material* H2O = new G4Material(name="Water",density,ncomponents=2);
H2O->AddElement(elH, natoms=2);
H2O->AddElement(elO, natoms=1);
```

2.3.4 Define a Mixture by Fractional Mass

In the example below, air is built from nitrogen and oxygen, by giving the fractional mass of each component.

Listing 2.9: Creating air by defining the fractional mass of its components.

```
G4double z, a, fractionmass, density;
G4String name, symbol;
G4int ncomponents;

a = 14.01*g/mole;
G4Element* elN  = new G4Element(name="Nitrogen",symbol="N" , z= 7., a);

a = 16.00*g/mole;
G4Element* elO  = new G4Element(name="Oxygen"  ,symbol="O" , z= 8., a);

density = 1.290*mg/cm3;
G4Material* Air = new G4Material(name="Air  ",density,ncomponents=2);
Air->AddElement(elN, fractionmass=70*perCent);
Air->AddElement(elO, fractionmass=30*perCent);
```

2.3.5 Define a Material from the Geant4 Material Database

In the example below, air and water are accessed via the Geant4 material database.

Listing 2.10: Defining air and water from the internal Geant4 database.

```
G4NistManager* man = G4NistManager::Instance();

G4Material* H2O  = man->FindOrBuildMaterial("G4_WATER");
G4Material* Air  = man->FindOrBuildMaterial("G4_AIR");
```

2.3.6 Define a Material from the Base Material

It is possible to build new material on base of an existing "base" material. This feature is useful for electromagnetic physics allowing to peak up for the derived material all correction data and precomputed tables of stopping powers and cross sections of the base material. In the example below, two methods how to create water with unusual density are shown.

Listing 2.11: Defining water with user defined density on base of G4_WATER.

```
G4double density;

density = 1.05*mg/cm3;
G4Material* water1 = new G4Material("Water_1.05",density,"G4_WATER");

density = 1.03*mg/cm3;
G4NistManager* man = G4NistManager::Instance();
G4Material* water2 = man->BuildMaterialWithNewDensity("Water_1.03","G4_WATER",density);
```

2.3.7 Print Material Information

Listing 2.12: Printing information about materials.

```
G4cout << H2O;                                \\ print a given material
G4cout << *(G4Material::GetMaterialTable());  \\ print the list of materials
```

In Geant4 examples you all possible ways to build a material.

2.3.8 Access to Geant4 material database

Listing 2.13: Geant4 material database may be accessed via UI commands.

```
/material/nist/printElement  Fe      \\ print element by name
/material/nist/printElementZ 13      \\ print element by atomic number
/material/nist/listMaterials type    \\ print materials type = [simple | compound | hep | all]
/material/g4/printElement    elmName \\ print instantiated element by name
/material/g4/printMaterial   matName \\ print instantiated material by name
```

In Geant4 examples you with find all possible ways to build a material.

2.4 How to Specify Particles

`G4VUserPhysicsList` is one of the mandatory user base classes described in *How to Define the main() Program*. Within this class all particles and physics processes to be used in your simulation must be defined. The range cut-off parameter should also be defined in this class.

The user must create a class derived from `G4VuserPhysicsList` and implement the following pure virtual methods:

```
ConstructParticle();      // construction of particles
ConstructProcess();       // construct processes and register them to particles
```

The user may also want to override the default implementation of the following virtual method:

```
SetCuts();                // setting a range cut value for all particles
```

This section provides some simple examples of the `ConstructParticle()` and `SetCuts()` methods. For information on `ConstructProcess()` methods, please see *How to Specify Physics Processes*.

2.4.1 Particle Definition

Geant4 provides various types of particles for use in simulations:

- ordinary particles, such as electrons, protons, and gammas
- resonant particles with very short lifetimes, such as vector mesons and delta baryons
- nuclei, such as deuteron, alpha, and heavy ions (including hyper-nuclei)
- quarks, di-quarks, and gluon

Each particle is represented by its own class, which is derived from `G4ParticleDefinition`. (Exception: G4Ions represents all heavy nuclei. Please see *Particles*.) Particles are organized into six major categories:

- lepton,
- meson,
- baryon,

- boson,
- shortlived and
- ion,

each of which is defined in a corresponding sub-directory under `geant4/source/particles`. There is also a corresponding granular library for each particle category.

The `G4ParticleDefinition` Class

`G4ParticleDefinition` has properties which characterize individual particles, such as, name, mass, charge, spin, and so on. Most of these properties are "read-only" and can not be changed directly. `G4ParticlePropertyTable` is used to retrieve (load) particle property of `G4ParticleDefinition` into (from) `G4ParticlePropertyData`.

How to Access a Particle

Each particle class type represents an individual particle type, and each class has a single object. This object can be accessed by using the static method of each class. There are some exceptions to this rule; please see *Particles* for details.

For example, the class `G4Electron` represents the electron and the member `G4Electron::theInstance` points its only object. The pointer to this object is available through the static methods `G4Electron::ElectronDefinition()`. `G4Electron::Definition()`.

More than 100 types of particles are provided by default, to be used in various physics processes. In normal applications, users will not need to define their own particles.

The unique object for each particle class is created when its static method to get the pointer is called at the first time. Because particles are dynamic objects and should be instantiated before initialization of physics processes, you must explicitly invoke static methods of all particle classes required by your program at the initialization step. (NOTE: The particle object was static and created automatically before 8.0 release)

Dictionary of Particles

The `G4ParticleTable` class is provided as a dictionary of particles. Various utility methods are provided, such as:

```
FindParticle(G4String name);         // find the particle by name
FindParticle(G4int PDGencoding)      // find the particle by PDG encoding .
```

`G4ParticleTable` is defined as a singleton object, and the static method `G4ParticleTable::GetParticleTable()` provides its pointer.

As for heavy ions (including hyper-nuclei), objects are created dynamically by requests from users and processes. The `G4ParticleTable` class provides methods to create ions, such as:

```
G4ParticleDefinition* GetIon(  G4int    atomicNumber,
                               G4int    atomicMass,
                               G4double   excitationEnergy);
```

Particles are registered automatically during construction. The user has no control over particle registration.

Constructing Particles

`ConstructParticle()` is a pure virtual method, in which the static member functions for all the particles you require should be called. This ensures that objects of these particles are created.

Warning: You must define "All PARTICLE TYPES" which are used in your application, except for heavy ions. "All PARTICLE TYPES" means not only primary particles, but also all other particles which may appear as secondaries generated by physics processes you use. Beginning with Geant4 version 8.0, you should keep this rule strictly because all particle definitions are revised to "non-static" objects.

For example, suppose you need a proton and a geantino, which is a virtual particle used for simulation and which does not interact with materials. The `ConstructParticle()` method is implemented as below:

Listing 2.14: Construct a proton and a geantino.

```
void MyPhysicsList::ConstructParticle()
{
  G4Proton::ProtonDefinition();
  G4Geantino::GeantinoDefinition();
}
```

Due to the large number of pre-defined particles in Geant4, it is cumbersome to list all the particles by this method. If you want all the particles in a Geant4 particle category, there are six utility classes, corresponding to each of the particle categories, which perform this function:

- `G4BosonConstructor`
- `G4LeptonConstructor`
- `G4MesonConstructor`
- `G4BarionConstructor`
- `G4IonConstructor`
- `G4ShortlivedConstructor`.

An example of this is shown in `ExN05PhysicsList`, listed below.

Listing 2.15: Construct all leptons.

```
void ExN05PhysicsList::ConstructLeptons()
{
  // Construct all leptons
  G4LeptonConstructor pConstructor;
  pConstructor.ConstructParticle();
}
```

2.4.2 Range Cuts

To avoid infrared divergence, some electromagnetic processes require a threshold below which no secondary will be generated. Because of this requirement, gammas, electrons and positrons require production threshold. This threshold should be defined as a distance, or range cut-off, which is internally converted to an energy for individual materials. The range threshold should be defined in the initialization phase using the `SetCuts()` method of `G4VUserPhysicsList`. *Cuts per Region* discusses threshold and tracking cuts in detail.

Setting the cuts

Production threshold values should be defined in `SetCuts()` which is a virtual method of the `G4VUserPhysicsList`. Construction of particles, materials, and processes should precede the invocation of `SetCuts()`. `G4RunManager` takes care of this sequence in usual applications.

This range cut value is converted threshold energies for each material and for each particle type (i.e. electron, positron and gamma) so that the particle with threshold energy stops (or is absorbed) after traveling the range cut distance. In addition, from the 9.3 release ,this range cut value is applied to the proton as production thresholds of nuclei for

hadron elastic processes. In this case, the range cut value does not means the distance of traveling. Threshold energies are calculated by a simple formula from the cut in range.

Note that the upper limit of the threshold energy is defined as 10 GeV. If you want to set higher threshold energy, you can change the limit by using "/cuts/setMaxCutEnergy" command before setting the range cut.

The idea of a "unique cut value in range" is one of the important features of Geant4 and is used to handle cut values in a coherent manner. For most applications, users need to determine only one cut value in range, and apply this value to gammas, electrons and positrons alike. (and proton too)

The default implemetation of `SetCuts()` method provides a `defaultCutValue` member as the unique range cut-off value for all particle types. The `defaultCutValue` is set to 1.0 mm by default. User can change this value by `SetDefaultCutValue()` The "/run/setCut" command may be used to change the default cut value interactively.

Warning: DO NOT change cut values inside the event loop. Cut values may however be changed between runs.

It is possible to set different range cut values for gammas, electrons and positrons by using `SetCutValue()` methods (or using "/run/setCutForAGivenParticle" command). However, user must be careful with physics outputs because Geant4 processes (especially energy loss) are designed to conform to the "unique cut value in range" scheme.

Beginning with Geant4 version 5.1, it is now possible to set production thresholds for each geometrical region. This new functionality is described in *Cuts per Region*.

2.5 How to Specify Physics Processes

2.5.1 Physics Processes

Physics processes describe how particles interact with materials. Geant4 provides seven major categories of processes:

- electromagnetic,
- hadronic,
- transportation,
- decay,
- optical,
- photolepton_hadron, and
- parameterisation.

All physics processes are derived from the `G4VProcess` base class. Its virtual methods

- `AtRestDoIt`,
- `AlongStepDoIt`, and
- `PostStepDoIt`

and the corresponding methods

- `AtRestGetPhysicalInteractionLength`,
- `AlongStepGetPhysicalInteractionLength`, and
- `PostStepGetPhysicalInteractionLength`

describe the behavior of a physics process when they are implemented in a derived class. The details of these methods are described in *Physics Processes*.

The following are specialized base classes to be used for simple processes:

`G4VAtRestProcess` Processes with only `AtRestDoIt`
`G4VContinuousProcess` Processes with only `AlongStepDoIt`
`G4VDiscreteProcess` processes with only `PostStepDoIt`

Another 4 virtual classes, such as G4VContinuousDiscreteProcess, are provided for complex processes.

2.5.2 Managing Processes

The G4ProcessManager class contains a list of processes that a particle can undertake. It has information on the order of invocation of the processes, as well as which kind of DoIt method is valid for each process in the list. A G4ProcessManager object corresponds to each particle and is attached to the G4ParticleDefiniton class.

In order to validate processes, they should be registered with the particle's G4ProcessManager. Process ordering information is included by using the AddProcess() and SetProcessOrdering() methods. For registration of simple processes, the AddAtRestProcess(), AddContinuousProcess() and AddDiscreteProcess() methods may be used.

G4ProcessManager is able to turn some processes on or off during a run by using the ActivateProcess() and InActivateProcess() methods. These methods are valid only after process registration is complete, so they must not be used in the *PreInit* phase.

The G4VUserPhysicsList class creates and attaches G4ProcessManager objects to all particle classes defined in the ConstructParticle() method.

2.5.3 Specifying Physics Processes

G4VUserPhysicsList is the base class for a "mandatory user class" (see *How to Define the main() Program*), in which all physics processes and all particles required in a simulation must be registered. The user must create a class derived from G4VUserPhysicsList and implement the pure virtual method ConstructProcess().

For example, if just the G4Geantino particle class is required, only the transportation process need be registered. The ConstructProcess() method would then be implemented as follows:

Listing 2.16: Register processes for a geantino.

```
void MyPhysicsList::ConstructProcess()
{
  // Define transportation process
  AddTransportation();
}
```

Here, the AddTransportation() method is provided in the G4VUserPhysicsList class to register the G4Transportation class with all particle classes. The G4Transportation class (and/or related classes) describes the particle motion in space and time. It is the mandatory process for tracking particles.

In the ConstructProcess() method, physics processes should be created and registered with each particle's instance of G4ProcessManager.

An example of process registration is given in the G4VUserPhysicsList::AddTransportation() method.

Registration in G4ProcessManager is a complex procedure for other processes and particles because the relations between processes are crucial for some processes. In order to ease registration procedures, G4PhysicsListHelper is provided. Users do not care about type of processes (ie. AtRest and/or Discrete and/or Continuous) or ordering parameters.

An example of electromagnetic process registration for the gamma is shown below

Listing 2.17: Register processes for a gamma.

```
void MyPhysicsList::ConstructProcess()
{
  // Define transportation process
  AddTransportation();
```

```
  // electromagnetic processes
  ConstructEM();
}

void MyPhysicsList::ConstructEM()
{
  // Get pointer to G4PhysicsListHelper
  G4PhysicsListHelper* ph = G4PhysicsListHelper::GetPhysicsListHelper();

  //  Get pointer to gamma
  G4ParticleDefinition* particle = G4Gamma::GammaDefinition();

  // Construct and register processes for gamma
  ph->RegisterProcess(new G4PhotoElectricEffect(), particle);
  ph->RegisterProcess(new G4ComptonScattering(), particle);
  ph->RegisterProcess(new G4GammaConversion(), particle);
  ph->RegisterProcess(new G4RayleighScattering(), particle);
}
```

2.6 How to Generate a Primary Event

2.6.1 Generating Primary Events

`G4VuserPrimaryGeneratorAction` is one of the mandatory classes available for deriving your own concrete class. In your concrete class, you have to specify how a primary event should be generated. Actual generation of primary particles will be done by concrete classes of `G4VPrimaryGenerator`, explained in the following subsection. Your `G4VUserPrimaryGeneratorAction` concrete class just arranges the way primary particles are generated.

Listing 2.18: ExG4PrimaryGeneratorAction01: An example of a G4VUserPrimaryGeneratorAction concrete class using G4ParticleGun. For the usage of G4Particle Gun refer to the next subsection.

```
//////////////////////////////////
// ExG4PrimaryGeneratorAction01.hh
//////////////////////////////////

#ifndef ExG4PrimaryGeneratorAction01_h
#define ExG4PrimaryGeneratorAction01_h 1

#include "G4VUserPrimaryGeneratorAction.hh"
#include "G4ThreeVector.hh"
#include "globals.hh"

class G4ParticleGun;
class G4Event;

class ExG4PrimaryGeneratorAction01 : public G4VUserPrimaryGeneratorAction
{
  public:
    ExG4PrimaryGeneratorAction01(
      const G4String& particleName = "geantino",
      G4double energy = 1.*MeV,
      G4ThreeVector position= G4ThreeVector(0,0,0),
      G4ThreeVector momentumDirection = G4ThreeVector(0,0,1));
    ~ExG4PrimaryGeneratorAction01();

    // methods
    virtual void GeneratePrimaries(G4Event*);

  private:
    // data members
```

```
    G4ParticleGun*  fParticleGun; //pointer a to G4 service class
};

#endif

//////////////////////////////////
// ExG4PrimaryGeneratorAction01.cc
//////////////////////////////////

#include "ExG4PrimaryGeneratorAction01.hh"

#include "G4Event.hh"
#include "G4ParticleGun.hh"
#include "G4ParticleTable.hh"
#include "G4ParticleDefinition.hh"

//....oooOOOOOooo........oooOOOOOooo........oooOOOOOooo........oooOOOOOooo......

ExG4PrimaryGeneratorAction01::ExG4PrimaryGeneratorAction01(
                              const G4String& particleName,
                              G4double energy,
                              G4ThreeVector position,
                              G4ThreeVector momentumDirection)
  : G4VUserPrimaryGeneratorAction(),
    fParticleGun(0)
{
  G4int nofParticles = 1;
  fParticleGun  = new G4ParticleGun(nofParticles);

  // default particle kinematic
  G4ParticleTable* particleTable = G4ParticleTable::GetParticleTable();
  G4ParticleDefinition* particle
    = particleTable->FindParticle(particleName);
  fParticleGun->SetParticleDefinition(particle);
  fParticleGun->SetParticleEnergy(energy);
  fParticleGun->SetParticlePosition(position);
  fParticleGun->SetParticleMomentumDirection(momentumDirection);
}

//....oooOOOOOooo........oooOOOOOooo........oooOOOOOooo........oooOOOOOooo......

ExG4PrimaryGeneratorAction01::~ExG4PrimaryGeneratorAction01()
{
  delete fParticleGun;
}

//....oooOOOOOooo........oooOOOOOooo........oooOOOOOooo........oooOOOOOooo......

void ExG4PrimaryGeneratorAction01::GeneratePrimaries(G4Event* anEvent)
{
  // this function is called at the begining of event

  fParticleGun->GeneratePrimaryVertex(anEvent);
}

//....oooOOOOOooo........oooOOOOOooo........oooOOOOOooo........oooOOOOOooo......
```

Selection of the generator

In the constructor of your `G4VUserPrimaryGeneratorAction`, you should instantiate the primary generator(s). If necessary, you need to set some initial conditions for the generator(s).

In *ExG4PrimaryGeneratorAction01*, `G4ParticleGun` is constructed to use as the actual primary particle generator. Methods of `G4ParticleGun` are described in the following section. Please note that the primary generator object(s)

you construct in your G4VUserPrimaryGeneratorAction concrete class must be deleted in your destructor.

Generation of an event

G4VUserPrimaryGeneratorAction has a pure virtual method named generatePrimaries(). This method is invoked at the beginning of each event. In this method, you have to invoke the G4VPrimaryGenerator concrete class you instantiated via the generatePrimaryVertex() method.

You can invoke more than one generator and/or invoke one generator more than once. Mixing up several generators can produce a more complicated primary event.

2.6.2 G4VPrimaryGenerator

Geant4 provides three G4VPrimaryGenerator concrete classes. Among these G4ParticleGun and G4GeneralParticleSource will be discussed here. The third one is G4HEPEvtInterface, which will be discussed in *Event Generator Interface*.

G4ParticleGun

G4ParticleGun is a generator provided by Geant4. This class generates primary particle(s) with a given momentum and position. It does not provide any sort of randomizing. The constructor of G4ParticleGun takes an integer which causes the generation of one or more primaries of exactly same kinematics. It is a rather frequent user requirement to generate a primary with randomized energy, momentum, and/or position. Such randomization can be achieved by invoking various set methods provided by G4ParticleGun. The invocation of these methods should be implemented in the generatePrimaries() method of your concrete G4VUserPrimaryGeneratorAction class before invoking generatePrimaryVertex() of G4ParticleGun. Geant4 provides various random number generation methods with various distributions (see *Global Usage Classes*).

Public methods of G4ParticleGun

The following methods are provided by G4ParticleGun, and all of them can be invoked from the generatePrimaries() method in your concrete G4VUserPrimaryGeneratorAction class.

- void SetParticleDefinition(G4ParticleDefinition*)
- void SetParticleMomentum(G4ParticleMomentum)
- void SetParticleMomentumDirection(G4ThreeVector)
- void SetParticleEnergy(G4double)
- void SetParticleTime(G4double)
- void SetParticlePosition(G4ThreeVector)
- void SetParticlePolarization(G4ThreeVector)
- void SetNumberOfParticles(G4int)

G4GeneralParticleSource

For many applications G4ParticleGun is a suitable particle generator. However if you want to generate primary particles in more sophisticated manner, you can utilize G4GeneralParticleSource, the Geant4 General Particle Source module (GPS), discussed in the next section (*Geant4 General Particle Source*).

2.7 Geant4 General Particle Source

2.7.1 Introduction

The G4GeneralParticleSource (GPS) is part of the Geant4 toolkit for Monte-Carlo, high-energy particle transport. Specifically, it allows the specifications of the spectral, spatial and angular distribution of the primary source particles. An overview of the GPS class structure is presented here. *Configuration* covers the configuration of GPS for a user application, and *Macro Commands* describes the macro command interface. *Example Macro File* gives an example input file to guide the first time user.

G4GeneralParticleSource is used exactly the same way as G4ParticleGun in a Geant4 application. In existing applications one can simply change your PrimaryGeneratorAction by globally replacing G4ParticleGun with G4GeneralParticleSource. GPS may be configured via command line, or macro based, input. The experienced user may also hard-code distributions using the methods and classes of the GPS that are described in more detail in a technical note[1].

The class diagram of GPS is shown in *Class diagram*. As of version 10.01, a split-class mechanism was introduced to reduce memory usage in multithreaded mode. The G4GeneralParticleSourceData class is a thread-safe singleton which provides access to the source information for the G4GeneralParticleSource class. The G4GeneralParticleSourceData class can have multiple instantiations of the G4SingleParticleSource class, each with independent positional, angular and energy distributions as well as incident particle types. To the user, this change should be transparent.

2.7.2 Configuration

GPS allows the user to control the following characteristics of primary particles:

- Spatial sampling: on simple 2D or 3D surfaces such as discs, spheres, and boxes.
- Angular distribution: unidirectional, isotropic, cosine-law, beam or arbitrary (user defined).
- Spectrum: linear, exponential, power-law, Gaussian, blackbody, or piece-wise fits to data.
- Multiple sources: multiple independent sources can be used in the same run.

As noted above, G4GeneralParticleSource is used exactly the same way as G4ParticleGun in a Geant4 application, and may be substituted for the latter by "global search and replace" in existing application source code.

Position Distribution

The position distribution can be defined by using several basic shapes to contain the starting positions of the particles. The easiest source distribution to define is a point source. One could also define planar sources, where the particles emanate from circles, annuli, ellipses, squares or rectangles. There are also methods for defining 1D or 2D accelerator beam spots. The five planes are oriented in the x-y plane. To define a circle one gives the radius, for an annulus one gives the inner and outer radii, and for an ellipse, a square or a rectangle one gives the half-lengths in x and y.

More complicated still, one can define surface or volume sources where the input particles can be confined to either the surface of a three dimensional shape or to within its entire volume. The four 3D shapes used within G4GeneralParticleSource are sphere, ellipsoid, cylinder and parallelepiped.A sphere can be defined simply by specifying the radius. Ellipsoids are defined by giving their half-lengths in x, y and z. Cylinders are defined such that the axis is parallel to the z-axis, the user is therefore required to give the radius and the z half-length. Parallelepipeds are defined by giving x, y and z half-lengths, plus the angles α, θ, and ϕ (*Parallelpiped angles*).

To allow easy definition of the sources, the planes and shapes are assumed to be orientated in a particular direction to the coordinate axes, as described above. For more general applications, the user may supply two vectors (x' and a vector in the plane x'-y') to rotate the co-ordinate axes of the shape with respect to the overall co-ordinate system

[1] General purpose Source Particle Module for Geant4/SPARSET: Technical Note, UoS-GSPM-Tech, Issue 1.1, C Ferguson, February 2000.

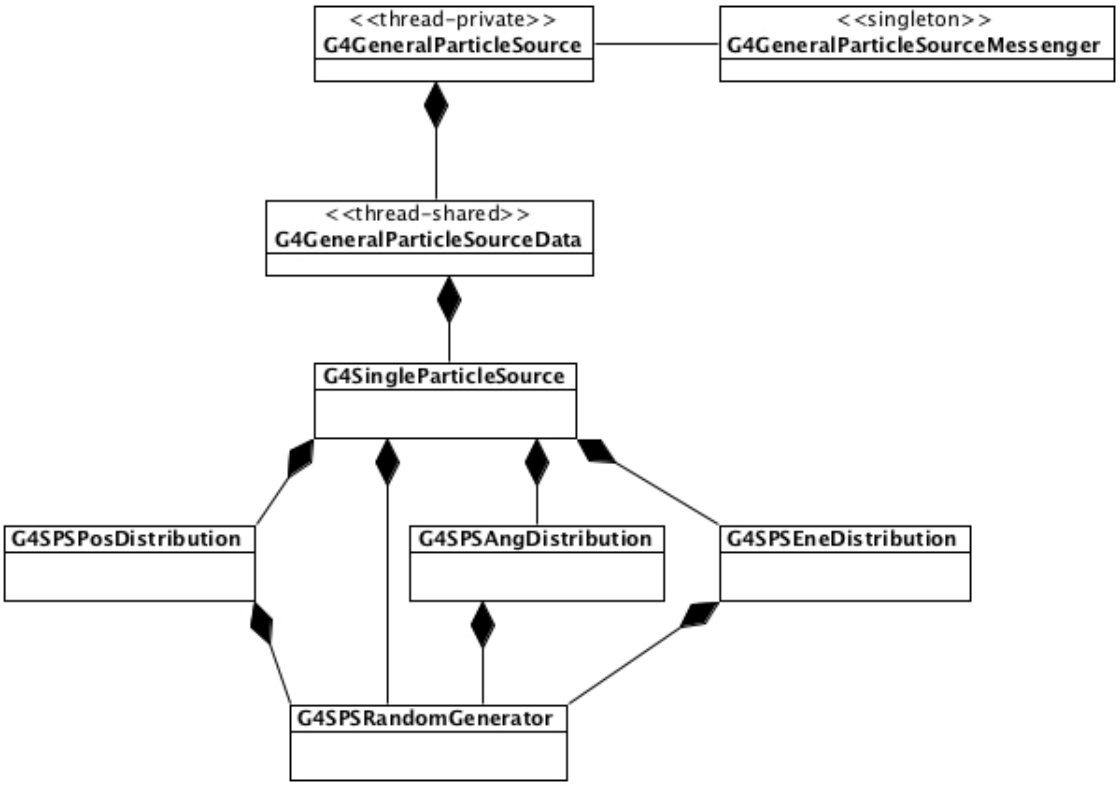


Fig. 2.1: The class diagram of `G4GeneralParticleSource`.

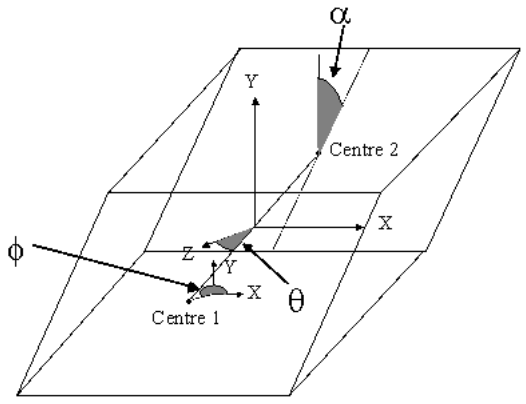


Fig. 2.2: The angles used in the definition of a Parallelepiped.

(*Rotation matrices*). The rotation matrix is automatically calculated within G4GeneralParticleSource. The starting points of particles are always distributed homogeneously over the 2D or 3D surfaces, although biasing can change this.

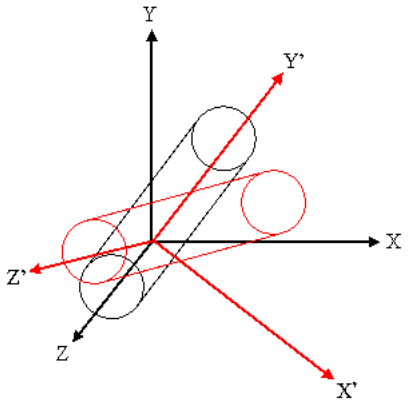


Fig. 2.3: An illustration of the use of rotation matrices. A cylinder is defined with its axis parallel to the z-axis (black lines), but the definition of 2 vectors can rotate it into the frame given by x', y', z' (red lines).

Angular Distribution

The angular distribution is used to control the directions in which the particles emanate from/incident upon the source point. In general there are three main choices, isotropic, cosine-law or user-defined. In addition there are options for specifying parallel beam as well as diversed accelerator beams. The isotropic distribution represents what would be seen from a uniform 4π flux. The cosine-law represents the distribution seen at a plane from a uniform 2π flux.

It is possible to bias (*Biasing*) both θ and ϕ for any of the predefined distributions, including setting lower and upper limits to θ and ϕ. User-defined distributions cannot be additionally biased (any bias should obviously be incorporated into the user definition).

Incident with zenith angle $\theta = 0$ means the particle is travelling along the -z axis. It is important to bear this in mind when specifying user-defined co-ordinates for angular distributions. The user must be careful to rotate the co-ordinate axes of the angular distribution if they have rotated the position distribution (*Rotation matrices*).

The user defined distribution requires the user to enter a histogram in either θ or ϕ or both. The user-defined distribution may be specified either with respect to the coordinate axes or with respect to the surface-normal of a shape or volume. For the surface-normal distribution, θ should only be defined between 0 and $\pi/2$, not the usual 0 to π range.

The top-level `/gps/direction` command uses direction cosines to specify the primary particle direction, as follows:

$$P_x = -\sin\theta\cos\phi \quad (2.1)$$
$$P_y = -\sin\theta\sin\phi \quad (2.2)$$
$$P_z = -\cos\theta \quad (2.3)$$

Energy Distribution

The energy of the input particles can be set to follow several built-in functions or a user-defined one, as shown in *GPS spectra*. The user can bias any of the pre-defined energy distributions in order to speed up the simulation (user-defined distributions are already biased, by construction).

Spectrum	Abbreviation	Functional Form	User Parameters
mono-energetic	Mono	$I \propto \delta(E - E_0)$	Energy E_0
linear	Lin	$I \propto I_0 + m \times E$	Intercept I_0, slope m
exponential	Exp	$I \propto \exp(-E/E_0)$	Energy scale-height E_0
power-law	Pow	$I \propto E^{\alpha}$	Spectral index α
Gaussian	Gauss	$I = (2\pi\sigma)^{-\frac{1}{2}} \exp[-(E/E_0)^2/\sigma^2]$	Mean energy E_0, standard deviation σ
bremsstrahlung	Brem	$I = \int 2E^2[h^2c^2(\exp(-E/kT) - 1)]^{-1}$	Temperature T
black body	Bbody	$I \propto (kT)^{\frac{1}{2}} E \exp(-E/kT)$	Temperature T (see text)
cosmic diffuse gamma ray	Cdg	$I \propto [(E/E_b)^{\alpha_1} + (E/E_b)^{\alpha_2}]^{-1}$	Energy range $E_{\min}$ to $E_{\max}$; energy E_b and indices α_1 and α_2 are fixed (see text)

There is also the option for the user to define a histogram in energy ("User") or energy per nucleon ("Epn") or to give an arbitrary point-wise spectrum ("Arb") that can be fit with various simple functions. The data for histograms or point spectra must be provided in ascending bin (abscissa) order. The point-wise spectrum may be differential (as with a binned histogram) or integral (a cumulative distribution function). If integral, the data must satsify $s(e1) \geq s(e2)$ for $e1 < e2$ when entered; this is not validated by the GPS code. The maximum energy of an integral spectrum is defined by the last-but-one data point, because GPS converts to a differential spectrum internally.

Unlike the other spectral distributions it has proved difficult to integrate indefinitely the black-body spectrum and this has lead to an alternative approach. Instead it has been decided to use the black-body formula to create a 10,000 bin histogram and then to produce random energies from this.

Similarly, the broken power-law for cosmic diffuse gamma rays makes generating an indefinite integral CDF problematic. Instead, the minimum and maximum energies specified by the user are used to construct a definite-integral CDF from which random energies are selected.

Biasing

The user can bias distributions by entering a histogram. It is the random numbers from which the quantities are picked that are biased and so one only needs a histogram from 0 to 1. Great care must be taken when using this option, as the way a quantity is calculated will affect how the biasing works, as discussed below. Bias histograms are entered in the same way as other user-defined histograms.

When creating biasing histograms it is important to bear in mind the way quantities are generated from those numbers. For example let us compare the biasing of a θ distribution with that of a ϕ distribution. Let us divide the θ and ϕ ranges up into 10 bins, and then decide we want to restrict the generated values to the first and last bins. This gives a new ϕ range of 0 to 0.628 and 5.655 to 6.283. Since ϕ is calculated using $\phi = 2\pi \times \mathrm{RNDM}$, this simple biasing will work correctly.

If we now look at θ, we expect to select values in the two ranges 0 to 0.314 (for $0 \leq \mathrm{RNDM} \leq 0.1$) and 2.827 to 3.142 (for $0 \leq \mathrm{RNDM} \leq 0.9$). However, the polar angle θ is calculated from the formula $\theta = \arccos(1 - 2 \times \mathrm{RNDM})$. From this, we see that 0.1 gives a θ of 0.644 and a RNDM of 0.9 gives a θ of 2.498. This means that the above will not bias the distribution as the user had wished. The user must therefore take into account the method used to generate random quantities when trying to apply a biasing scheme to them. Some quantities such as x, y, z and ϕ will be relatively easy to bias, but others may require more thought.

User-Defined Histograms

The user can define histograms for several reasons: angular distributions in either θ or ϕ; energy distributions; energy per nucleon distributions; or biasing of x, y, z, θ, ϕ, or energy. Even though the reasons may be different the approach is the same.

To choose a histogram the command `/gps/hist/type` is used (*Macro Commands*). If one wanted to enter an angular distribution one would type "theta" or "phi" as the argument. The histogram is loaded, one bin at a time, by using the `/gps/hist/point` command, followed by its two arguments the upper boundary of the bin and the weight (or area) of the bin. Histograms are therefore differential functions.

Currently histograms are limited to 1024 bins. The first value of each user input data pair is treated as the upper edge of the histogram bin and the second value is the bin content. The exception is the very first data pair the user input whose first value is the treated as the lower edge of the first bin of the histogram, and the second value is not used. This rule applies to all distribution histograms, as well as histograms for biasing.

The user has to be aware of the limitations of histograms. For example, in general θ is defined between 0 and π and ϕ is defined between 0 and 2π, so histograms defined outside of these limits may not give the user what they want (see also *Biasing*).

2.7.3 Macro Commands

`G4GeneralParticleSource` can be configured by typing commands from the `/gps` command directory tree, or including the `/gps` commands in a g4macro file.

`G4ParticleGun` equivalent commands

Command	Arguments	Description and restrictions
/gps/List		List available incident particles
/gps/particle	name	Defines the particle type [default *geantino*], using Geant4 naming convention.
/gps/direction	Px Py Pz	Set the momentum direction [default (1,0,0)] of generated particles using *direction cosines*
/gps/energy	E unit	Sets the energy [default 1 MeV] for mono-energetic sources. The units can be eV, keV, MeV, GeV, TeV or PeV. (NB: it is recommended to use /gps/ene/mono instead.)
/gps/position	X Y Z unit	Sets the centre co-ordinates (X,Y,Z) of the source [default (0,0,0) cm]. The units can be micron, mm, cm, m or km. (NB: it is reccomended to use /gps/pos/centre instead.)
/gps/ion	Z A Q E	After `/gps/particle ion`, sets the properties (atomic number Z, atomic mass A, ionic charge Q, excitation energy E in keV) of the ion.
/gps/ionLvl	Z A Q lvl	After `/gps/particle ion`, sets the properties (atomic number Z, atomic mass A, ionic charge Q, Number of metastable state excitation level (0-9) of the ion.
/gps/time	t0 unit	Sets the primary particle (event) time [default 0 ns]. The units can be ps, ns, us, ms, or s.
/gps/polarization	Px Py Pz	Sets the polarization vector of the source, which does not need to be a unit vector.
/gps/number	N	Sets the number of particles [default 1] to simulate on each event.
/gps/verbose	level	Control the amount of information printed out by the GPS code. Larger values produce more detailed output.

Multiple source specification

Command	Arguments	Description and restrictions
/gps/source/add	intensity	Add a new particle source with the specified intensity
/gps/source/list		List the particle sources defined.
/gps/source/clear		Remove all defined particle sources.
/gps/source/show		Display the current particle source
/gps/source/set	index	Select the specified particle source as the current one.
/gps/source/delete	index	Remove the specified particle source.
/gps/source/ multiplevertex	flag	Specify *true* for simulaneous generation of mutiple vertices, one from each specified source. False [default] generates a single vertex, choosing one source randomly.
/gps/source/ intensity	intensity	Reset the current source to the specified intensity
/gps/source/ flatsampling	flag	Set to True to allow biased sampling among the sources. Setting to True will ignore source intensities. The default is False.

Source position and structure

Command	Arguments	Description and restrictions
/gps/pos/type	dist	Sets the source positional distribution type: *Point* [default], *Plane, Beam*, *Surface, Volume*.
/gps/pos/shape	shape	Sets the source shape type, after `/gps/pos/type` has been used. For a Plane this can be *Circle, Annulus*, *Ellipse, Square, Rectangle*. For both Surface or Volume sources this can be *Sphere, Ellipsoid, Cylinder, Para* (parallelpiped).
/gps/pos/centre	X Y Z unit	Sets the centre co-ordinates (X,Y,Z) of the source [default (0,0,0) cm]. The units can only be micron, mm, cm, m or km.
/gps/pos/rot1	$R1_x$ $R1_y$ $R1_z$	Defines the first (x' direction) vector R1 [default (1,0,0)], which does not need to be a unit vector, and is used together with `/gps/pos/rot2` to create the rotation matrix of the shape defined with `/gps/shape`.
/gps/pos/rot2	$R2_x$ $R2_y$ $R2_z$	Defines the second vector R2 in the xy plane [default (0,1,0)], which does not need to be a unit vector, and is used tohgether with `/gps/pos/rot1` to create the rotation matrix of the shape defined with `/gps/shape`.
/gps/pos/halfx	len unit	Sets the half-length in x [default 0 cm] of the source. The units can only be micron, mm, cm, m or km.
/gps/pos/halfy	len unit	Sets the half-length in y [default 0 cm] of the source. The units can only be micron, mm, cm, m or km.
/gps/pos/halfz	len unit	Sets the half-length in z [default 0 cm] of the source. The units can only be micron, mm, cm, m or km.
/gps/pos/radius	len unit	Sets the radius [default 0 cm] of the source or the outer radius for annuli. The units can only be micron, mm, cm, m or km.
/gps/pos/inner_radius	len unit	Sets the inner radius [default 0 cm] for annuli. The units can only be micron, mm, cm, m or km.
/gps/pos/sigma_r	sigma unit	Sets the transverse (radial) standard deviation [default 0 cm] of beam position profile. The units can only be micron, mm, cm, m or km.
/gps/pos/sigma_x	sigma unit	Sets the standard deviation [default 0 cm] of beam position profile in x-direction. The units can only be micron, mm, cm, m or km.
/gps/pos/sigma_y	sigma unit	Sets the standard deviation [default 0 cm] of beam position profile in y-direction. The units can only be micron, mm, cm, m or km.
/gps/pos/paralp	alpha unit	Used with a Parallelepiped. The angle [default 0 rad] ALPHA formed by the y-axis and the plane joining the centre of the faces parallel to the zx plane at y and +y. The units can only be deg or rad.
/gps/pos/parthe	theta unit	Used with a Parallelepiped. Polar angle [default 0 rad] THETAS of the line connecting the centre of the face at z to the centre of the face at +z. The units can only be deg or rad.
/gps/pos/parphi	phi unit	Used with a Parallelepiped. The azimuth angle [default 0 rad] PHIS of the line connecting the centre of the face at z with the centre of the face at +z. The units can only be deg or rad.
/gps/pos/confine	name	Allows the user to confine the source to the physical volume *name* [default NULL].

Source direction and angular distribution

Command	Arguments	Description and restrictions
/gps/ang/type	AngDis	Sets the angular distribution type (*iso* [default], *cos, planar, beam1d, beam2d, focused, user*) to either isotropic, cosine-law or user-defined.
/gps/ang/rot1	$AR1_x$ $AR1_y$ $AR1_z$	Defines the first (x' direction) rotation vector AR1 [default (1,0,0)] for the angular distribution and is not necessarily a unit vector. Used with `/gps/ang/rot2` to compute the angular distribution rotation matrix.
/gps/ang/rot2	$AR2_x$ $AR2_y$ $AR2_z$	Defines the second rotation vector AR2 in the xy plane [default (0,1,0)] for the angular distribution, which does not necessarily have to be a unit vector. Used with `/gps/ang/rot2` to compute the angular distribution rotation matrix.
/gps/ang/mintheta	MinTheta unit	Sets a minimum value [default 0 rad] for the THETAS distribution. Units can be deg or rad.
/gps/ang/maxtheta	MaxTheta unit	Sets a maximum value [default PI rad] for the THETAS distribution. Units can be deg or rad.
/gps/ang/minphi	MinPhi unit	Sets a minimum value [default 0 rad] for the PHIS distribution. Units can be deg or rad.
/gps/ang/maxphi	MaxPhi unit	Sets a maximum value [default 2PI rad] for the PHIS distribution. Units can be deg or rad.
/gps/ang/sigma_r	sigma unit	Sets the standard deviation [default 0 rad] of beam directional profile in radial. The units can only be deg or rad.
/gps/ang/sigma_x	sigma unit	Sets the standard deviation [default 0 rad] of beam directional profile in x-direction. The units can only be deg or rad.
/gps/ang/sigma_y	sigma unit	Sets the standard deviation [default 0 rad] of beam directional profile in y-direction. The units can only be deg or rad.
/gps/ang/focuspoint	X Y Z unit	Set the focusing point (X,Y,Z) for the beam [default (0,0,0) cm]. The units can only be micron, mm, cm, m or km.
/gps/ang/user_coor	bool	Calculate the angular distribution with respect to the user definded co-ordinate system (*true*), or with respect to the global co-ordinate system (*false*, default).
/gps/ang/surfnorm	bool	Allows user to choose whether angular distributions are with respect to the co-ordinate system (*false*, default) or surface normals (*true*) for user-defined distributions.

Energy spectra

Command	Arguments	Description and restrictions
/gps/ene/type	EnergyDis	Sets the energy distribution type to one of (see Table *spectra*): *Mono* (mono-energetic, default), *Lin* (linear), *Pow* (power-law), `Exp` (exponential), *Gauss* (Gaussian), *Brem* (bremsstrahlung), *Bbody* (blackbody), *Cdg* (cosmic diffuse gamma-ray), *User* (user-defined histogram), *Arb* (point-wise spectrum), *Epn* (energy-per-nucleon histogram)
/gps/ene/min	Emin unit	Sets the minimum [default 0 keV] for the energy distribution. The units can be eV, keV, MeV, GeV, TeV or PeV.
/gps/ene/max	Emax unit	Sets the maximum [default 0 keV] for the energy distribution. The units can be eV, keV, MeV, GeV, TeV or PeV.
/gps/ene/mono	E unit	Sets the energy [default 1 MeV] for mono-energetic sources. The units can be eV, keV, MeV, GeV, TeV or PeV.
/gps/ene/sigma	sigma unit	Sets the standard deviation [default 0 keV] in energy for Gaussian or Mono energy distributions. The units can be eV, keV, MeV, GeV, TeV or PeV.
/gps/ene/alpha	alpha	Sets the exponent *ALPHA* [default 0] for a power-law distribution.
/gps/ene/temp	T	Sets the temperature in kelvins [default 0] for black body and bremsstrahlung spectra.
/gps/ene/ezero	E0	Sets scale E_0 [default 0] for exponential distributions.
/gps/ene/gradient	gradient	Sets the gradient (slope) [default 0] for linear distributions.
/gps/ene/intercept	intercept	Sets the Y-intercept [default 0] for the linear distributions.
/gps/ene/biasAlpha	alpha	Sets the exponent *ALPHA* [default 0] for a biased power-law distribution. Bias weight is determined from the power-law probability distribution.
/gps/ene/calculate		Prepares integral PDFs for the interally-binned cosmic diffuse gamma ray (*Cdg*) and black body (*Bbody*) distributions.
/gps/ene/emspec	bool	Allows user to specify distributions are in momentum (*false*) or energy (*true*, default). Only valid for *User* and *Arb* distributions.
/gps/ene/diffspec	bool	Allows user to specify whether a point-wise spectrum is integral (*false*) or differential (*true*, default). The integral spectrum is only usable for *Arb* distributions.

User-defined histograms and interpolated functions

Command	Arguments	Description and restrictions
/gps/hist/type	type	Set the histogram type: predefined *biasx* [default], *biasy, biasz, biast* (angle THETAS), *biasp* (angle PHIS), *biaspt* (position THETAS), *biaspp* (position PHIS), *biase*; user-defined histograms *theta, phi, energy, arb* (point-wise), *epn* (energy per nucleon).
/gps/hist/reset	type	Re-set the specified histogram: *biasx* [default], , *biasy, biasz, biast*, *biasp, biaspt, biaspp, biase, theta*, *phi, energy, arb, epn*.
/gps/hist/point	E_{hi} Weight	Specify one entry (with contents *Weight*) in a histogram (where *E:sub:'hi'* is the bin upper edge) or point-wise distribution (where *E:sub:'hi'* is the abscissa). The abscissa E_{hi} must be in Geant4 default units (MeV for energy, rad for angle).
/gps/hist/file	HistFile	Import an arbitary energy histogram in an ASCII file. The format should be one *E:sub:'hi'* *Weight* pair per line of the file, following the detailed instructions in *User-defined histograms and interpolated functions* For histograms, *E:sub:'hi'* is the bin upper edge, for point-wise distributions *E:sub:'hi'* is the abscissa. The abscissa E_{hi} must be in Geant4 default units (MeV for energy, rad for angle).
/gps/hist/inter	type	Sets the interpolation type (*Lin* linear, *Log* logarithmic, `Exp` exponential, *Spline* cubic spline) for point-wise spectra. This command **must** be issued immediately after the last data point.

2.7.4 Example Macro File

```
# Macro test2.g4mac
/control/verbose 0
/tracking/verbose 0
/event/verbose 0
/gps/verbose 2
/gps/particle gamma
/gps/pos/type Plane
/gps/pos/shape Square
/gps/pos/centre 1 2 1 cm
/gps/pos/halfx 2 cm
/gps/pos/halfy 2 cm
/gps/ang/type cos
/gps/ene/type Lin
/gps/ene/min 2 MeV
/gps/ene/max 10 MeV
/gps/ene/gradient 1
/gps/ene/intercept 1
/run/beamOn 10000
```

The above macro defines a planar source, square in shape, 4 cm by 4 cm and centred at (1,2,1) cm. By default the normal of this plane is the z-axis. The angular distribution is to follow the cosine-law. The energy spectrum is linear, with gradient and intercept equal to 1, and extends from 2 to 10 MeV. 10,000 primaries are to be generated.

The standard Geant4 output should show that the primary particles start from between 1, 0, 1 and 3, 4, 1 (in cm) and have energies between 2 and 10 MeV, as shown in *GPS example*, in which we plotted the actual energy, position and angular distributions of the primary particles generated by the above macro file.

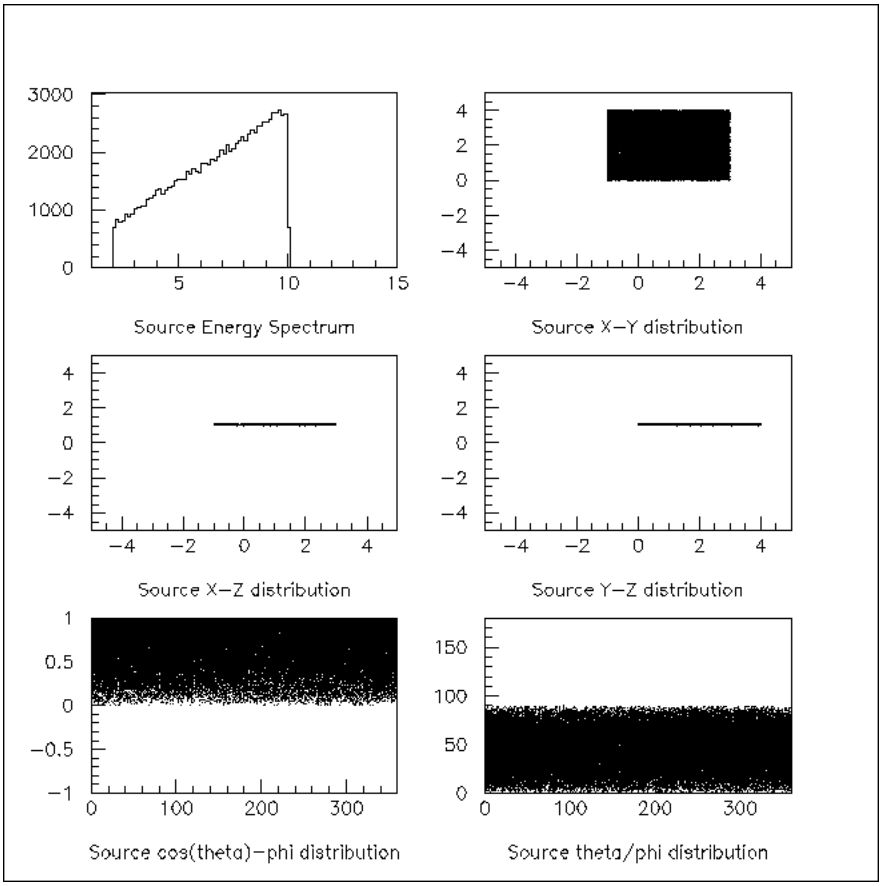


Fig. 2.4: Figure 4. Energy, position and angular distributions of the primary particles as generated by the macro file shown above.

2.8 How to Make an Executable Program

The code for the user examples in Geant4 is placed in the subdirectory `examples` of the main Geant4 source package. This directory is installed to the `share/Geant4-X.Y.Z/examples` (where `X.Y.Z` is the Geant4 version number) subdirectory under the installation prefix. In the following sections, a quick overview will be given on how to build a concrete example, "ExampleB1", which is part of the Geant4 distribution, using CMake and the older, and now deprecated, Geant4Make system.

2.8.1 Using CMake to build Applications: Geant4Config.cmake

Geant4 installs a file named `Geant4Config.cmake` located in:

```
+- CMAKE_INSTALL_PREFIX
    +- lib/
       +- Geant4-G4VERSION/
          +- Geant4Config.cmake
```

which is designed for use with the CMake scripting language `find_package` command. Building a Geant4 application using CMake therefore involves writing a CMake script `CMakeLists.txt` using this and other CMake commands to locate Geant4 and describe the build of your application against it. Whilst it requires a bit of effort to write the script, CMake provides a very powerful and flexible tool, especially if you are working on multiple platforms. It is therefore the method we recommend for building Geant4 applications.

We'll use Basic Example B1, which you may find in the Geant4 source directory under `examples/basic/B1`, to demonstrate the use of CMake to build a Geant4 application. You'll find links to the latest CMake documentation for the commands used throughout, so please follow these for further information. The application sources and scripts are arranged in the following directory structure:

```
+- B1/
   +- CMakeLists.txt
   +- exampleB1.cc
   +- include/
   |  ... headers.hh ...
   +- src/
      ... sources.cc ...
```

Here, `exampleB1.cc` contains `main()` for the application, with `include/` and `src/` containing the implementation class headers and sources respectively. This arrangement of source files is not mandatory when building with CMake, apart from the location of the `CMakeLists.txt` file in the root directory of the application.

The text file `CMakeLists.txt` is the CMake script containing commands which describe how to build the exampleB1 application:

```
# (1)
cmake_minimum_required(VERSION 2.6 FATAL_ERROR)
project(B1)

# (2)
option(WITH_GEANT4_UIVIS "Build example with Geant4 UI and Vis drivers" ON)
if(WITH_GEANT4_UIVIS)
  find_package(Geant4 REQUIRED ui_all vis_all)
else()
  find_package(Geant4 REQUIRED)
endif()

# (3)
include(${Geant4_USE_FILE})
include_directories(${PROJECT_SOURCE_DIR}/include)

# (4)
```

```
file(GLOB sources ${PROJECT_SOURCE_DIR}/src/*.cc)
file(GLOB headers ${PROJECT_SOURCE_DIR}/include/*.hh)

# (5)
add_executable(exampleB1 exampleB1.cc ${sources} ${headers})
target_link_libraries(exampleB1 ${Geant4_LIBRARIES})

# (6)
set(EXAMPLEB1_SCRIPTS
  exampleB1.in
  exampleB1.out
  init_vis.mac
  run1.mac
  run2.mac
  vis.mac
  )

foreach(_script ${EXAMPLEB1_SCRIPTS})
  configure_file(
    ${PROJECT_SOURCE_DIR}/${_script}
    ${PROJECT_BINARY_DIR}/${_script}
    COPYONLY
    )
endforeach()

# (7)
install(TARGETS exampleB1 DESTINATION bin)
```

For clarity, the above listing has stripped out the main comments (CMake comments begin with a "#") you'll find in the actual file to highlight each distinct task:

1. Basic Configuration

   The `cmake_minimum_required` command simply ensures we're using a suitable version of CMake. Though the build of Geant4 itself requires CMake 3.3 and we recommend this version for your own projects, `Geant4Config.cmake` can support earlier versions from 2.6.4 and the 2.8.X series. The `project` command sets the name of the project and enables and configures C and C++ compilers.
2. Find and Configure Geant4

   The aforementioned `find_package` command is used to locate and configure Geant4 (we'll see how to specify the location later when we run CMake), the `REQUIRED` argument being supplied so that CMake will fail with an error if it cannot find Geant4. The `option` command specifies a boolean variable which defaults to `ON`, and which can be set when running CMake via a `-D` command line argument, or toggled in the CMake GUI interfaces. We wrap the calls to `find_package` in a conditional block on the option value. This allows us to configure the use of Geant4 UI and Visualization drivers by exampleB1 via the `ui_all vis_all` "component" arguments to `find_package`. These components and their usage is described later.
3. Configure the Project to Use Geant4 and B1 Headers

   To automatically configure the header path, and force setting of compiler flags and compiler definitions needed for compiling against Geant4, we use the `include` command to load a CMake script supplied by Geant4. The CMake variable named `Geant4_USE_FILE` is set to the path to this module when Geant4 is located by `find_package`. We use the `include_directories` command to add the B1 header directory to the compiler's header search path. The CMake variable `PROJECT_SOURCE_DIR` points to the top level directory of the project and is set by the earlier call to the `project` command.
4. List the Sources to Build the Application

   Use the globbing functionality of the `file` command to prepare lists of the B1 source and header files.

   Note however that CMake globbing *is only used here as a convenience*. The expansion of the glob only happens when CMake is run, so if you later add or remove files, the generated build scripts

will not know a change has taken place. Kitware strongly recommend listing sources explicitly as CMake automatically makes the build depend on the `CMakeLists.txt` file. This means that if you explicitly list the sources in `CMakeLists.txt`, any changes you make will be automatically picked when you rebuild. This is most useful when you are working on a project with sources under version control and multiple contributors.

5. Define and Link the Executable

   The `add_executable` command defines the build of an application, outputting an executable named by its first argument, with the sources following. Note that we add the headers to the list of sources so that they will appear in IDEs like Xcode.

   After adding the executable, we use the `target_link_libraries` command to link it with the Geant4 libraries. The `Geant4_LIBRARIES` variable is set by `find_package` when Geant4 is located, and is a list of all the libraries needed to link against to use Geant4.

6. Copy any Runtime Scripts to the Build Directory

   Because we want to support out of source builds so that we won't mix CMake generated files with our actual sources, we copy any scripts used by the B1 application to the build directory. We use `foreach` to loop over the list of scripts we constructed, and `configure_file` to perform the actual copy.

   Here, the CMake variable `PROJECT_BINARY_DIR` is set by the earlier call to the `project` command and points to the directory where we run CMake to configure the build.

7. If Required, Install the Executable

   Use the `install` command to create an install target that will install the executable to a `bin` directory under `CMAKE_INSTALL_PREFIX`.

   If you don't intend your application to be installable, i.e. you only want to use it locally when built, you can leave this out.

This sequence of commands is the most basic needed to compile and link an application with Geant4, and is easily extendable to more involved use cases such as platform specific configuration or using other third party packages (via `find_package`).

With the CMake script in place, using it to build an application is a two step process. First CMake is run to generate buildscripts to describe the build. By default, these will be Makefiles on Unix platforms, and Visual Studio solutions on Windows, but you can generate scripts for other tools like Xcode and Eclipse if you wish. Second, the buildscripts are run by the chosen build tool to compile and link the application.

A key concept with CMake is that we generate the buildscripts and run the build in a separate directory, the so-called *build directory*, from the directory in which the sources reside, the so-called *source directory*. This is the exact same technique we used when building Geant4 itself. Whilst this may seem awkward to begin with, it is a very useful technique to employ. It prevents mixing of CMake generated files with those of your application, and allows you to have multiple builds against a single source without having to clean up, reconfigure and rebuild.

We'll illustrate this configure and build process on Linux/OS X using Makefiles, and on Windows using Visual Studio. The example script and Geant4's `Geant4Config.cmake` script are vanilla CMake, so you should be able to use other Generators (such as Xcode and Eclipse) without issue.

Building ExampleB1 with CMake on Unix with Makefiles

We'll assume, *for illustration only*, that you've copied the exampleB1 sources into a directory under your home area so that we have

```
+- /home/you/B1/
   +- CMakeLists.txt
   +- exampleB1.cc
   +- include/
   +- src/
   +- ...
```

Here, our *source directory* is `/home/you/B1`, in other words the directory holding the `CMakeLists.txt` file.

Let's also assume that you have already installed Geant4 in your home area under, *for illustration only*, `/home/you/geant4-install`.

Our first step is to create a *build directory* in which build the example. We will create this alongside our B1 *source directory* as follows:

```
$ cd $HOME
$ mkdir B1-build
```

We now change to this *build directory* and run CMake to generate the Makefiles needed to build the B1 application. We pass CMake two arguments

```
$ cd $HOME/B1-build
$ cmake -DGeant4_DIR=/home/you/geant4-install/lib64/Geant4-G4VERSION $HOME/B1
```

Here, the first argument points CMake to our install of Geant4. Specifically, it is the directory holding the `Geant4Config.cmake` file that Geant4 installs to help CMake find and use Geant4. You should of course adapt the value of this variable to the location of your actual Geant4 install. This provides the most specific way to point CMake to the Geant4 install you want to use. You may also use the `CMAKE_PREFIX_PATH` variable, e.g.

```
$ cd $HOME/B1-build
$ cmake -DCMAKE_PREFIX_PATH=/home/you/geant4-install $HOME/B1
```

This is most useful for system integrators as it may be extended with paths to the install prefixes of additional required software packages and also may be set as an environment variable that CMake will use at configuration time.

The second argument to CMake is the path to the *source directory* of the application we want to build. Here it's just the B1 directory as discussed earlier. You should of course adapt the value of that variable to where you copied the B1 source directory.

CMake will now run to configure the build and generate Makefiles. On Linux, you will see the output

```
$ cmake -DGeant4_DIR=/home/you/geant4-install/lib64/Geant4-G4VERSION $HOME/B1
-- The C compiler identification is GNU 4.9.2
-- The CXX compiler identification is GNU 4.9.2
-- Check for working C compiler: /usr/bin/gcc-4.9
-- Check for working C compiler: /usr/bin/gcc-4.9 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/g++-4.9
-- Check for working CXX compiler: /usr/bin/g++-4.9 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/you/B1-build
```

On OS X, you will see slightly different output, but the last three lines should be identical.

If you now list the contents of you build directory, you can see the files generated:

```
$ ls
CMakeCache.txt       exampleB1.in   Makefile      vis.mac
CMakeFiles           exampleB1.out  run1.mac
cmake_install.cmake  init_vis.mac   run2.mac
```

Note the `Makefile` and that all the scripts for running the exampleB1 application we're about to build have been copied across. With the Makefile available, we can now build by simply running make:

```
$ make -jN
```

CMake generated Makefiles support parallel builds, so can set `N` suitable for the number of cores on your machine (e.g. on a dual core processor, you could set N to 2). When make runs, you should see the output

```
$ make
Scanning dependencies of target exampleB1
[ 16%] Building CXX object CMakeFiles/exampleB1.dir/exampleB1.cc.o
[ 33%] Building CXX object CMakeFiles/exampleB1.dir/src/B1PrimaryGeneratorAction.cc.o
[ 50%] Building CXX object CMakeFiles/exampleB1.dir/src/B1EventAction.cc.o
[ 66%] Building CXX object CMakeFiles/exampleB1.dir/src/B1RunAction.cc.o
[ 83%] Building CXX object CMakeFiles/exampleB1.dir/src/B1DetectorConstruction.cc.o
[100%] Building CXX object CMakeFiles/exampleB1.dir/src/B1SteppingAction.cc.o
Linking CXX executable exampleB1
[100%] Built target exampleB1
```

CMake Unix Makefiles are quite terse, but you can make them more verbose by adding the `VERBOSE` argument to make:

```
$ make VERBOSE=1
```

If you now list the contents of your *build directory* you will see the exampleB1 application executable has been created:

```
$ ls
CMakeCache.txt       exampleB1      init_vis.mac      run2.mac
CMakeFiles           exampleB1.in   Makefile          vis.mac
cmake_install.cmake  exampleB1.out  run1.mac
```

You can now run the application in place:

```
$ ./exampleB1
Available UI session types: [ GAG, tcsh, csh ]

*************************************************************
 Geant4 version Name: geant4-10-03 [MT]   (2-December-2016)
  << in Multi-threaded mode >>
                      Copyright : Geant4 Collaboration
                      Reference : NIM A 506 (2003), 250-303
                            WWW : http://cern.ch/geant4
*************************************************************

<<< Reference Physics List QBBC
Visualization Manager instantiating with verbosity "warnings (3)"...
Visualization Manager initialising...
Registering graphics systems...
```

Note that the exact output shown will depend on how both Geant4 and your application were configured. Further output and behaviour beyond the `Registering graphics systems...` line will depend on what UI and Visualization drivers your Geant4 install supports. If you recall the use of the `ui_all vis_all` in the `find_package` command, this results in all available UI and Visualization drivers being activated in your application. If you didn't want any UI or Visualization, you could rerun CMake as:

```
$ cmake -DWITH_GEANT4_UIVIS=OFF -DGeant4_DIR=/home/you/geant4-install/lib64/Geant4-G4VERSION
↪$HOME/B1
```

This would switch the `option` we set up to false, and result in `find_package` not activating any UI or Visualization for the application. You can easily adapt this pattern to provide options for your application such as additional components or features.

Once the build is configured, you can edit code for the application in its *source directory*. You only need to rerun `make` in the corresponding *build directory* to pick up and compile the changes. However, note that due to the use

of CMake globbing to create the source file list, if you add or remove files, you need to rerun CMake to pick up the changes! This is another reason why Kitware recommend listing the sources explicitly.

Building ExampleB1 with CMake on Windows with Visual Studio

As with building Geant4 itself, the simplest system to use for building applications on Windows is a Visual Studio Developer Command Prompt, which can be started from *Start → All Programs → Visual Studio 2015 → Visual Studio Tools → Developer Command Prompt for VS2015*.

We'll assume, *for illustration only*, that you've copied the exampleB1 sources into a directory `C:\Users\YourUsername\Geant4\B1` so that we have:

```
+- C:\Users\YourUsername\Geant4\B1
   +- CMakeLists.txt
   +- exampleB1.cc
   +- include\
   +- src\
   +- ...
```

Here, our *source directory* is `C:\Users\YourUsername\Geant4\B1`, in other words the directory holding the `CMakeLists.txt` file.

Let's also assume that you have already installed Geant4 in your home area under, *for illustration only*, `C:\Users\YourUsername\Geant4\geant4G4ZIPNAME-install`.

Our first step is to create a *build directory* in which build the example. We will create this alongside our B1 *source directory* as follows, working from the Visual Studio Developer Command Prompt:

```
> cd %HOMEPATH%\Geant4
> mkdir B1-build
```

We now change to this *build directory* and run CMake to generate the Visual Studio solution needed to build the B1 application. We pass CMake two arguments:

```
> cd %HOMEPATH%\Geant4\B1-build
> cmake -DGeant4_DIR=%HOMEPATH%\geant4G4ZIPNAME-install\lib\Geant4-G4VERSION %HOMEPATH%\Geant4\B1
```

Here, the first argument points CMake to our install of Geant4. Specifically, it is the directory holding the `Geant4Config.cmake` file that Geant4 installs to help CMake find and use Geant4. You should of course adapt the value of this variable to the location of your actual Geant4 install. As with the examples above, you can also use the `CMAKE_PREFIX_PATH` variable.

The second argument is the path to the *source directory* of the application we want to build. Here it's just the B1 directory as discussed earlier. You should of course adapt the value of that variable to where you copied the B1 source directory.

CMake will now run to configure the build and generate Visual Studio solutions and you will see the output

```
> cmake -DGeant4_DIR=%HOMEPATH%\geant4G4ZIPNAME-install\lib\Geant4-G4VERSION %HOMEPATH%\Geant4\B1
-- Building for: Visual Studio 14 2015
-- The C compiler identification is MSVC 19.0.23026.0
-- The CXX compiler identification is MSVC 19.0.23026.0
-- Check for working C compiler using: Visual Studio 14 2015
-- Check for working C compiler using: Visual Studio 14 2015 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler using: Visual Studio 14 2015
-- Check for working CXX compiler using: Visual Studio 14 2015 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
```

```
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: C:/Users/YourUsername/Geant4/B1-build
```

If you now list the contents of you build directory, you can see the files generated:

```
> dir /B
ALL_BUILD.vcxproj
ALL_BUILD.vcxproj.filters
B1.sln
B1.vcxproj
B1.vcxproj.filters
CMakeCache.txt
CMakeFiles
cmake_install.cmake
exampleB1.in
exampleB1.out
exampleB1.vcxproj
exampleB1.vcxproj.filters
init_vis.mac
INSTALL.vcxproj
INSTALL.vcxproj.filters
run1.mac
run2.mac
vis.mac
ZERO_CHECK.vcxproj
ZERO_CHECK.vcxproj.filters
```

Note the `B1.sln` solution file and that all the scripts for running the exampleB1 application we're about to build have been copied across. With the solution available, we can now build by running cmake to drive MSBuild:

```
> cmake --build . --config Release
```

Solution based builds are quite verbose, but you should not see any errors at the end. In the above, we have built the B1 program in `Release` mode, meaning that it is optimized and has no debugging symbols. As with building Geant4 itself, this is chosen to provide optimum performance. If you require debugging information for your application, simply change the argument to `RelWithDebInfo`. Note that in both cases you must match the configuration of your application with that of the Geant4 install, i.e. if you are building the application in `Release` mode, then ensure it uses a `Release` build of Geant4. Link and/or runtime errors may result if mixed configurations are used.

After running the build, if we list the contents of the build directory again we see

```
> dir /B
ALL_BUILD.vcxproj
ALL_BUILD.vcxproj.filters
B1.sln
B1.vcxproj
B1.vcxproj.filters
CMakeCache.txt
CMakeFiles
cmake_install.cmake
exampleB1.dir
exampleB1.in
exampleB1.out
exampleB1.vcxproj
exampleB1.vcxproj.filters
init_vis.mac
INSTALL.vcxproj
INSTALL.vcxproj.filters
Release
run1.mac
run2.mac
vis.mac
```

```
Win32
ZERO_CHECK.vcxproj
ZERO_CHECK.vcxproj.filters

> dir /B Release
exampleB1.exe
...
```

Here, the `Release` subdirectory contains the executable, and the main build directory contains all the `.mac` scripts for running the program. If you build in different modes, the executable for that mode will be in a directory named for that mode, e.g. `RelWithDebInfo/exampleB1.exe`. You can now run the application in place:

```
> .\Release\exampleB1.exe
Available UI session types: [ Win32, GAG, csh ]

*************************************************************
 Geant4 version Name: geant4-10-03   (2-December-2016)
                      Copyright : Geant4 Collaboration
                      Reference : NIM A 506 (2003), 250-303
                            WWW : http://cern.ch/geant4
*************************************************************

<<< Reference Physics List QBBC
Visualization Manager instantiating with verbosity "warnings (3)"...
Visualization Manager initialising...
Registering graphics systems...
```

Note that the exact output shown will depend on how both Geant4 and your application were configured. Further output and behaviour beyond the `Registering graphics systems...` line will depend on what UI and Visualization drivers your Geant4 install supports.

Whilst the Visual Studio Developer Command prompt provides the simplest way to build an application, the generated Visual Studio Solution file (`B1.sln` in the above example) may also be opened directly in the Visual Studio IDE. This provides a more comprehensive development and debugging environment, and you should consult its documentation if you wish to use this.

One key CMake related item to note goes back to our listing of the *headers* for the application in the call to `add_executable`. Whilst CMake will naturally ignore these for configuring compilation of the application, it will add them to the Visual Studio Solution. If you do not list them, they will not be editable in the Solution in the Visual Studio IDE.

2.8.2 Using Geant4Make to build Applications: binmake.gmk

Geant4Make is the Geant4 GNU Make toolchain formerly used to build the toolkit and applications. It is installed on UNIX systems (except for Cygwin) for backwards compatibility with the Geant4 Examples and your existing applications which use a GNUmakefile and the Geant4Make `binmake.gmk` file. *However, please note that the system is now deprecated, meaning that it is no longer supported and may be removed in future releases without warning. You should migrate your application to be built using CMake via the ''Geant4Config.cmake'' script, or any other build tool of your choice, using the ''geant4-config'' program to query the relevant compiler/linker flags.*

The files for Geant4Make are installed under:

```
+- CMAKE_INSTALL_PREFIX/
   +- share/
      +- geant4make/
         +- geant4make.sh
         +- geant4make.csh
         +- config/
            +- binmake.gmk
            +- ...
```

The system is designed to form a self-contained GNUMake system which is configured primarily by environment variables (though you may manually replace these with Make variables if you prefer). Building a Geant4 application using Geant4Make therefore involves configuring your environment followed by writing a GNUmakefile using the Geant4Make variables and GNUMake modules.

To configure your environment, simply source the relevant configuration script `CMAKE_INSTALL_PREFIX/share/Geant4-G4VERSION/geant4make/geant4make.(c)sh` for your shell. Whilst both scripts can be sourced interactively, if you are using the C shell and need to source the script inside another script, you must use the commands:

```
cd CMAKE_INSTALL_PREFIX/share/Geant4-G4VERSION/geant4make
source geant4make.csh
```

or alternatively

```
source CMAKE_INSTALL_PREFIX/share/Geant4-G4VERSION/geant4make/geant4make.csh \\
  CMAKE_INSTALL_PREFIX/share/Geant4-G4VERSION/geant4make
```

In both cases, you should replace `CMAKE_INSTALL_PREFIX` with the actual prefix you installed Geant4 under. Both of these commands work around a limitation in the C shell which prevents the script locating itself.

Please also note that due to limitations of Geant4Make, you *should not* rely on the environment variables it sets for paths into Geant4 itself. In particular, note that the `G4INSTALL` variable *is not equivalent to CMAKE_INSTALL_PREFIX*.

Once you have configured your environment, you can start building your application. Geant4Make enforces a specific organization and naming of your sources in order to simplify the build. We'll use Basic Example B1, which you may find in the Geant4 source directory under `examples/basic/B1`, as the canonical example again. Here, the sources are arranged as follows

```
+- B1/
    +- GNUmakefile
    +- exampleB1.cc
    +- include/
    |  ... headers.hh ...
    +- src/
       ... sources.cc ...
```

As before, `exampleB1.cc` contains `main()` for the application, with `include/` and `src/` containing the implementation class headers and sources respectively. You `must` organise your sources in this structure with these filename extensions to use Geant4Make as it will expect this structure when it tries to build the application.

With this structure in place, the GNUmakefile for exampleB1 is very simple:

```
name := exampleB1
G4TARGET := $(name)
G4EXLIB := true

.PHONY: all
all: lib bin

include $(G4INSTALL)/config/binmake.gmk
```

Here, `name` is set to the application to be built, and it must match the name of the file containing the `main()` program without the `.cc` extension. The rest of the variables are structural to prepare the build, and finally the core Geant4Make module is included. The `G4INSTALL` variable is set in the environment by the `geant4make` script to point to the root of the Geant4Make directory structure.

With this structure in place, simply run `make` to build your application:

```
$ make
```

If you need extra detail on the build, you append `CPPVERBOSE=1` to the make command to see a detailed log of the command executed.

The application executable will be output to `$(G4WORKDIR)/bin/$(G4SYSTEM)/exampleB1`, where `$(G4SYSTEM)` is the system and compiler combination you are running on, e.g. Linux-g++. By default, `$(G4WORKDIR)` is set by the `geant4make` scripts to `$(HOME)/geant4_workdir`, and also prepends this directory to your `PATH`. You can therefore run the application directly once it's built:

```
$ exampleB1
```

If you prefer to keep your application builds separate, then you can set `G4WORKDIR` in the `GNUmakefile` before including `binmake.gmk`. In this case you would have to run the executable by supplying the full path.

Further documentation of the usage of Geant4Make and syntax and extensions for the `GNUMakefile` is described in the FAQ and Appendices of the Geant4 User's Guide for Application Developers.

Please note that the Geant4Make toolchain is provided purely for conveniance and backwards compatibility. We encourage you to use and migrate your applications to the new CMake and `geant4-config` tools. Geant4Make is deprecated in Geant4 10.0 and later.

2.9 How to Set Up an Interactive Session

2.9.1 Introduction

Roles of the "intercoms" category

The "intercoms" category provides an expandable command interpreter. It is the key mechanism of Geant4 to realize secure user interactions across categories without being annoyed by dependencies among categories. Geant4 commands can be used in an interactive session, a batch mode with a macro file, or a direct C++ call.

User Interfaces to drive the simulation

Geant4 can be controllled by a seriese of Geant4 UI commands. The "intercoms" category provides the abstract class `G4UIsession` that processes interactive commands. The concrete implementations of (graphical) user interface are provided in the "interfaces" category. The strategy realize to adopt various user interface tools, and allows Geant4 to utilize the state-of-the-art GUI tools such as Motif, Qt, and Java etc. The following interfaces is currently available;

1. Command-line terminal (dumb terminal and tcsh-like terminal)
2. Xm, Qt, Win32, variations of the above terminal by using a Motif, Qt, Windows widgets
3. GAG, a fully graphical user interface and its network extension GainServer of the client/server type.

Implementation of the user sesssions (1 and 2) is included in the `source/interfaces/basic` directory. As for GAG, the front-end class is included in the `source/interfaces/GAG` directory, while its partner GUI package MOMO.jar is available under the `environments/MOMO` directory. MOMO.jar, Java archive file, contains not only GAG, but also GGE and other helper packages. Supplementary information is available from the author's web page (see URL below).

GAG, GainServer's client GUI Gain: http://www-geant4.kek.jp/~yoshidah/

2.9.2 A Short Description of Available Interfaces

G4UIterminal

This interface opens a session on a command-line terminal. G4UIterminal runs on all supported platforms. There are two kinds of shells, G4UIcsh and G4UItcsh. G4UItcsh supports tcsh-like readline features (cursor and command completion) and works on Linux on Mac, while G4UIcsh is a plain standard input (cin) shell that works on all platforms. The following built-in commands are available in G4UIterminal;

cd, pwd change, display the current command directory.
ls, lc list commands and subdirectories in the current directory.
history show previous commands.
!historyID reissue previous command.
?command show current parameter values of the command.
help command show command help.
exit terminate the session.

G4UItcsh supports user-friendly key bindings a-la-tcsh. G4UItcsh runs on Linux and Mac. The following keybindings are supported;

^A move cursor to the top
^B backward cursor ([LEFT] cursor)
^C (except Windows terminal) abort a run (soft abort) during event processing. A program will be terminated while accepting a user command.
^D delete/exit/show matched list
^E move cursor to the end
^F forward cursor ([RIGHT] cursor)
^K clear after the cursor
^N next command ([DOWN] cursor)
^P previous command ([UP] cursor)
TAB command completion
DEL backspace
BS backspace

The example below shows how to set a user's prompt.

```
G4UItcsh* tcsh = new G4UItcsh();
tcsh-> SetPrompt("%s>");
```

The following strings are supported as substitutions in a prompt string.

%s current application status
%/ current working directory
%h history number

Command history in a user's session is saved in a file $(HOME)/.g4_hist that is automatically read at the next session, so that command history is available across sessions.

G4UIXm, G4UIQt and G4UIWin32 classes

These interfaces are versions of G4UIterminal implemented over libraries Motif, Qt and WIN32 respectively. G4UIXm uses the Motif XmCommand widget, G4UIQt the Qt dialog widget, and G4UIWin32 the Windows "edit" component to do the command capturing. These interfaces are useful if working in conjunction with visualization drivers that use the Xt library, Qt library or the WIN32 one.

A command box is at disposal for entering or recalling Geant4 commands. Command completion by typing "TAB" key is available in the command box. The shell commands "exit, cont, help, ls, cd..." are also supported. A menu bar

can be customized through the *AddMenu* and *AddButton* method. Ex:

/gui/addMenu test Test
/gui/addButton test Init /run/initialize
/gui/addButton test "Set gun" "/control/execute gun.g4m"
/gui/addButton test "Run one event" "/run/beamOn 1"

`G4UIXm` runs on Unix/Linux with Motif. `G4UIQt` run everywhere with Qt. `G4UIWin32` runs on Windows.

`G4UIGAG` and `G4UIGainServer` classes

They are front-end classes of Geant4 which make connections with their respective graphical user interfaces, GAG (Geant4 Adaptive GUI) via pipe, and Gain (Geant4 adaptive interface for network) via sockets. While GAG must run on the same system (Windows or Unixen) as a Geant4 application, Gain can run on a remote system (Windows, Linux, etc.) in which JRE (Java Runtime Environment) is installed. A Geant4 application is invoked on a Unix (Linux) system and behaves as a network server. It opens a port, waiting the connection from the Gain. Gain has capability to connect to multiple Geant4 "servers" on Unixen systems at different hosts.

Client GUIs, GAG and Gain have almost similar look-and-feel. So, GAG's functionalities are briefly explained here. Please refer to the URL previously mentioned for details.

Using GAG, user can select a command, set its parameters and execute it. It is adaptive, in the sense that it reflects the internal states of Geant4 that is a state machine. So, GAG always provides users with the Geant4 commands which may be added, deleted, enabled or disabled during a session. GAG does nothing by itself but to play an intermediate between user and an executable simulation program via pipes. Geant4's front-end class `G4UIGAG` must be instantiated to communicate with GAG. GAG runs on Linux and Windows. MOMO.jar is supplied in the Geant4 source distribution and can be run by a command:

```
%java -jar  /path/to/geant4.10.00/environments/MOMO/MOMO.jar
```

GAG has following functions.

GAG Menu: The menus are to choose and run a Geant4 executable file, to kill or exit a Geant4 process and to exit GAG. Upon the normal exit or an unexpected death of the Geant4 process, GAG window are automatically reset to run another Geant4 executable.

Geant4 Command tree: Upon the establishment of the pipe connection with the Geant4 process, GAG displays the command menu, using expandable tree browser whose look and feel is similar to a file browser. Disabled commands are shown in opaque. GAG doesn't display commands that are just below the root of the command hierarchy. Direct type-in field is available for such input. Guidance of command categories and commands are displayed upon focusing. GAG has a command history function. User can re-execute a command with old parameters, edit the history, or save the history to create a macro file.

Command Parameter panel: GAG's parameter panel is the user-friendliest part. It displays parameter name, its guidance, its type(s) (integer, double, Boolean or string), omittable, default value(s), expression(s) of its range and candidate list(s) (for example, of units). Range check is done by intercoms and the error message from it is shown in the pop-up dialog box. When a parameter component has a candidate list, a list box is automatically displayed . When a file is requested by a command, the file chooser is available.

Logging: Log can be redirected to the terminal (xterm or cygwin window) from which GAG is invoked. It can be interrupted as will, in the middle of a long session of execution. Log can be saved to a file independent of the above redirection . GAG displays warning or error messages from Geant4 in a pop-up warning widget.

2.9.3 How to Select Interface in Your Applications

To choose an interface (`G4UIxxx` where `xxx = terminal,Xm, Win32, Qt, GAG, GainServer`) in your programs, there are two ways.

- Calling G4UIxxx directly:

```
#include "G4Uixxx.hh"

G4UIsession* session = new G4UIxxx;
session-> SessionStart();

delete session;
```

Note: For using a tcsh session, `G4UIterminal` is instantiated like:

```
G4UIsession* session = new G4UIterminal(new G4UItcsh);
```

If the user wants to deactivate the default signal handler (soft abort) raised by "Ctr-C", the false flag can be set in the second argument of the `G4UIterminal` constructor like

```
G4UIsession* session = new G4UIterminal(new G4UItcsh, false).
```

- Using `G4UIExecutive` This is more convenient way for choosing a session type, that can select a session at run-time according to a rule described below.

```
#include "G4UIExecutive.hh"

G4UIExecutive* ui = new G4UIExecutive(argc, argv);
ui->SessionStart();

delete ui;
```

`G4UIExecutive` has several ways to choose a session type. A session is selected in the following rule. Note that session types are identified by a case-insensitive characters ("qt", "xm", "win32", "gag", "tcsh", "csh").

1. Check the argument of the constructor of G4UIExecutive. You can specify a session like `new G4UIExecutive(argc, argv, "qt");`
2. Check environment variables, `G4UI_USE_XX (XX= QT, XM, WIN32, GAG, TCSH)`. Select a session if the corresponding environment variable is defined. Variables are checked in the order of QT, XM, WIN32, GAG, TCSH if multiple variables are set.
3. Check `~/.g4session` . You can specify the default session type and a session type by each application in that file. The below shows a sample of `.g4session`.

```
tcsh  # default session
exampleN03 Qt # (application name / session type)
myapp  tcsh
hoge csh
```

4. Guess the best session type according to build session libraries. The order of the selection is Qt, tcsh, Xm.

In any cases, `G4UIExecutive` checks if a specified session is build or not. If not, it goes the next step. A terminal session with csh is the fallback session. If none of specified session is available, then it will be selected.

2.10 How to Execute a Program

2.10.1 Introduction

A Geant4 application can be run either in

- 'purely hard-coded' batch mode
- batch mode, but reading a macro of commands
- interactive mode, driven by command lines

- interactive mode via a Graphical User Interface

The last mode will be covered in *How to Set Up an Interactive Session*. The first three modes are explained here.

2.10.2 'Hard-coded' Batch Mode

Below is a modified main program of the basic example B1 to represent an application which will run in batch mode.

Listing 2.19: An example of the `main()` routine for an application which will run in batch mode.

```
int main()
{
  // Construct the default run manager
  G4RunManager* runManager = new G4RunManager;

  // Set mandatory initialization classes
  runManager->SetUserInitialization(new B1DetectorConstruction);
  runManager->SetUserInitialization(new QGSP_BIC_EMY);
  runManager->SetUserAction(new B1PrimaryGeneratorAction);

  // Set user action classes
  runManager->SetUserAction(new B1SteppingAction());
  runManager->SetUserAction(new B1EventAction());
  runManager->SetUserAction(new B1RunAction());

  // Initialize G4 kernel
  runManager->Initialize();

  // start a run
  int numberOfEvent = 1000;
  runManager->BeamOn(numberOfEvent);

  // job termination
  delete runManager;
  return 0;
}
```

Even the number of events in the run is 'frozen'. To change this number you must at least recompile `main()`.

2.10.3 Batch Mode with Macro File

Below is a modified main program of the basic example B1 to represent an application which will run in batch mode, but reading a file of commands.

Listing 2.20: An example of the `main()` routine for an application which will run in batch mode, but reading a file of commands.

```
int main(int argc,char** argv)
{
  // Construct the default run manager
  G4RunManager* runManager = new G4RunManager;

  // Set mandatory initialization classes
  runManager->SetUserInitialization(new B1DetectorConstruction);
  runManager->SetUserInitialization(new QGSP_BIC_EMY);
  runManager->SetUserAction(new B1PrimaryGeneratorAction);

  // Set user action classes
  runManager->SetUserAction(new B1SteppingAction());
  runManager->SetUserAction(new B1EventAction());
  runManager->SetUserAction(new B1RunAction());
```

```
  // Initialize G4 kernel
  runManager->Initialize();

  //read a macro file of commands
  G4UImanager* UI = G4UImanager::GetUIpointer();
  G4String command = "/control/execute ";
  G4String fileName = argv[1];
  UI->ApplyCommand(command+fileName);

  // job termination
  delete runManager;
  return 0;
}
```

This example will be executed with the command:

```
> exampleB1  run1.mac
```

where `exampleB1` is the name of the executable and `run1.mac` is a macro of commands located in the current directory, which could look like:

Listing 2.21: A typical command macro.

```
#
# Macro file for myProgram
#
# set verbose level for this run
#
/run/verbose      2
/event/verbose    0
/tracking/verbose 1
#
# Set the initial kinematic and run 100 events
# electron 1 GeV to the direction (1.,0.,0.)
#
/gun/particle e-
/gun/energy 1 GeV
/run/beamOn 100
```

Indeed, you can re-execute your program with different run conditions without recompiling anything.

Note: many G4 category of classes have a verbose flag which controls the level of 'verbosity'.

Usually `verbose=0` means silent. For instance

- `/run/verbose` is for the `RunManager`
- `/event/verbose` is for the `EventManager`
- `/tracking/verbose` is for the `TrackingManager`
- ...etc...

2.10.4 Interactive Mode Driven by Command Lines

Below is an example of the main program for an application which will run interactively, waiting for command lines entered from the keyboard.

Listing 2.22: An example of the `main()` routine for an application which will run interactively, waiting for commands from the keyboard.

```
int main(int argc,char** argv)
{
  // Construct the default run manager
  G4RunManager* runManager = new G4RunManager;

  // Set mandatory initialization classes
  runManager->SetUserInitialization(new B1DetectorConstruction);
  runManager->SetUserInitialization(new QGSP_BIC_EMY);
  runManager->SetUserAction(new B1PrimaryGeneratorAction);

  // Set user action classes
  runManager->SetUserAction(new B1SteppingAction());
  runManager->SetUserAction(new B1EventAction());
  runManager->SetUserAction(new B1RunAction());

  // Initialize G4 kernel
  runManager->Initialize();

  // Define UI terminal for interactive mode
  G4UIsession * session = new G4UIterminal;
  session->SessionStart();
  delete session;

  // job termination
  delete runManager;
  return 0;
}
```

This example will be executed with the command:

```
> exampleB1
```

where `exampleB1` is the name of the executable.

The G4 kernel will prompt:

```
Idle>
```

and you can start your session. An example session could be:

Run 5 events:

```
Idle> /run/beamOn 5
```

Switch on tracking/verbose and run one more event:

```
Idle> /tracking/verbose 1
Idle> /run/beamOn 1
```

Change primary particle type an run more events:

```
Idle> /gun/particle mu+
Idle> /gun/energy 10 GeV
Idle> /run/beamOn 1
Idle> /gun/particle proton
Idle> /gun/energy 100 MeV
Idle> /run/beamOn 3
Idle> exit
```

For the meaning of the machine state `Idle`, see *Geant4 as a state machine*.

This mode is useful for running a few events in debug mode and visualizing them. How to include visualization will be shown in the next, general case, example.

2.10.5 General Case

All basic examples in the `examples/basic` subdirectory of the Geant4 source distribution have the following `main()` structure. The application can be run either in batch or interactive mode.

Listing 2.23: The typical `main()` routine from the examples directory.

```
int main(int argc,char** argv)
{
  // Construct the default run manager
  G4RunManager* runManager = new G4RunManager;

  // Set mandatory initialization classes
  runManager->SetUserInitialization(new B1DetectorConstruction);
  runManager->SetUserInitialization(new QGSP_BIC_EMY);
  runManager->SetUserAction(new B1PrimaryGeneratorAction);

  // Set user action classes
  runManager->SetUserAction(new B1SteppingAction());
  runManager->SetUserAction(new B1EventAction());
  runManager->SetUserAction(new B1RunAction());

  // Initialize G4 kernel
  runManager->Initialize();

#ifdef G4VIS_USE
  // Initialize visualization
  G4VisManager* visManager = new G4VisExecutive;
  // G4VisExecutive can take a verbosity argument - see /vis/verbose guidance.
  // G4VisManager* visManager = new G4VisExecutive("Quiet");
  visManager->Initialize();
#endif

  // Get the pointer to the User Interface manager
  G4UImanager* UImanager = G4UImanager::GetUIpointer();

  if (argc!=1) {
    // batch mode
    G4String command = "/control/execute ";
    G4String fileName = argv[1];
    UImanager->ApplyCommand(command+fileName);
  }
  else {
    // interactive mode : define UI session
#ifdef G4UI_USE
    G4UIExecutive* ui = new G4UIExecutive(argc, argv);
#ifdef G4VIS_USE
    UImanager->ApplyCommand("/control/execute init_vis.mac");
#else
    UImanager->ApplyCommand("/control/execute init.mac");
#endif
    ui->SessionStart();
    delete ui;
#endif
  }

  // Job termination
  // Free the store: user actions, physics_list and detector_description are
  // owned and deleted by the run manager, so they should not be deleted
  // in the main() program !

#ifdef G4VIS_USE
```

```
  delete visManager;
#endif
  delete runManager;
}
```

Notice that both user interface and visualization systems are under the control of the compiler preprocessor symbols G4UI_USE and G4VIS_USE. Geant4's CMake support script automatically adds definitions for these symbols to the compiler flags, unless you set the CMake variables G4UI_NONE and G4VIS_NONE before calling find_package(Geant4). This provides you with a simple system to control the enabling of the user interface and visualization systems, though you are free to use your own names for the preprocessor symbols if your use case requires (though you must then add them to the compiler flags yourself). Notice also that, in interactive mode, few intializations have been put in the macros init_vis.mac, or init_vis.mac, which is executed before the session start.

Listing 2.24: The init.mac macro

```
# Macro file for the initialization phase of example B1
# when running in interactive mode without visualization
#
# Set some default verbose
/control/verbose 2
/control/saveHistory
/run/verbose 2
```

The init_vis.mac macro has just added a line with a call to vis.mac:

```
# Macro file for the initialization phase of example B1
# when running in interactive mode with visualization
#
# Set some default verbose
#
/control/verbose 2
/control/saveHistory
/run/verbose 2
#
# Visualization setting
/control/execute vis.mac
```

The vis.mac macro defines a minimal setting for drawing volumes and trajectories accumulated for all events of a given run:

```
# Macro file for the visualization setting in the initialization phase
# of the B1 example when running in interactive mode
#
#
# Use this open statement to create an OpenGL view:
/vis/open OGL 600x600-0+0
#
# Draw geometry:
/vis/drawVolume
#
# Specify view angle:
/vis/viewer/set/viewpointThetaPhi 90. 180.
#
# Draw smooth trajectories at end of event, showing trajectory points
# as markers 2 pixels wide:
/vis/scene/add/trajectories smooth
#
# To superimpose all of the events from a given run:
/vis/scene/endOfEventAction accumulate
#
# Re-establish auto refreshing and verbosity:
/vis/viewer/set/autoRefresh true
```

```
/vis/verbose warnings
#
# For file-based drivers, use this to create an empty detector view:
#/vis/viewer/flush
```

Also, this example demonstrates that you can read and execute a macro from another macro or interactively:

```
Idle> /control/execute  mySubMacro.mac
```

2.11 How to Visualize the Detector and Events

2.11.1 Introduction

This section briefly explains how to perform Geant4 Visualization. The description here is based on the sample program `examples/basic/B1`. More details are given in *Visualization*.

2.11.2 Visualization Drivers

The Geant4 visualization system was developed in response to a diverse set of requirements:

1. Quick response to study geometries, trajectories and hits
2. High-quality output for publications
3. Flexible camera control to debug complex geometries
4. Tools to show volume overlap errors in detector geometries
5. Interactive picking to get more information on visualized objects

No one graphics system is ideal for all of these requirements, and many of the large software frameworks into which Geant4 has been incorporated already have their own visualization systems, so Geant4 visualization was designed around an abstract interface that supports a diverse family of graphics systems. Some of these graphics systems use a graphics library compiled with Geant4, such as OpenGL, Qt or OpenInventor, while others involve a separate application, such as HepRApp or DAWN.

You need not use all visualization drivers. You can select those suitable to your purposes. In the following, for simplicity, we assume that the Geant4 libraries are built with the Qt driver.

If you build Geant4 using the standard `CMake` procedure, you include Qt by setting GEANT4_USE_QT to ON.

In order to use the the Qt driver, you need the OpenGL library, which is installed in many platforms by default and `CMake` will find it. (If you wish to "do-it-yourself", see *Installing Visualization Drivers*.) The makefiles then set appropriate C-pre-processor flags to select appropriate code at compilation time.

If you are using multithreaded mode, from Geant4 version 10.2 event drawing is performed by a separate thread and you may need to optimise this with special `/vis/multithreading` commands - see *Multithreading commands*.

2.11.3 How to Incorporate Visualization Drivers into an Executable

Most Geant4 examples already incorporate visualization drivers. If you want to include visualization in your own Geant4 application, you need to instantiate and initialize a subclass of `G4VisManager` that implements the pure virtual function `RegisterGraphicsSystems()`.

The provided class `G4VisExecutive` can handle all of this work for you. `G4VisExecutive` is sensitive to the `G4VIS_...` variables (that you either set by hand or that are set for you by GNUMake or CMake configuration). See any of the Geant4 examples for how to use `G4VisExecutive`.

If you really want to write your own subclass, rather than use `G4VisExecutive`, you may do so. You will see how to do this by looking at `G4VisExecutive.icc`. This subclass must be compiled in the user's domain to force the loading of appropriate libraries in the right order. A typical extract is:

```
...
  RegisterGraphicsSystem (new G4DAWNFILE);
...
#ifdef G4VIS_USE_OPENGLX
  RegisterGraphicsSystem (new G4OpenGLImmediateX);
  RegisterGraphicsSystem (new G4OpenGLStoredX);
#endif
...
```

The `G4VisExecutive` takes ownership of all registered graphics systems, and will delete them when it is deleted at the end of the user's job (see below).

If you wish to use `G4VisExecutive` but register an additional graphics system, `XXX` say, you may do so either before or after initializing:

```
visManager->RegisterGraphicsSytem(new XXX);
visManager->Initialize();
```

An example of a typical `main()` function is given below.

2.11.4 Writing the `main()` Method to Include Visualization

Now we explain how to write a visualization manager and the `main()` function for Geant4 visualization. In order that your Geant4 executable is able to perform visualization, you must instantiate and initialize *your* Visualization Manager in the `main()` function. The typical `main()` function available for visualization is written in the following style:

Listing 2.25: The typical `main()` routine available for visualization.

```
//----- C++ source codes: main() function for visualization
#ifdef G4VIS_USE
#include "G4VisExecutive.hh"
#endif

.....

int main(int argc,char** argv) {

.....

  // Instantiation and initialization of the Visualization Manager
#ifdef G4VIS_USE
  // visualization manager
  G4VisManager* visManager = new G4VisExecutive;
  // G4VisExecutive can take a verbosity argument - see /vis/verbose guidance.
  // G4VisManager* visManager = new G4VisExecutive("Quiet");
  visManager->Initialize();
#endif

.....

  // Job termination
#ifdef G4VIS_USE
  delete visManager;
#endif

.....

  return 0;
```

```
}

//----- end of C++
```

In the instantiation, initialization, and deletion of the Visualization Manager, the use of the macro `G4VIS_USE` is recommended as it is set automatically by the CMake and GNUmake support scripts. This allows one easily to build an executable without visualization, if required, without changing the code (but remember you have to force recompilation whenever you change the environment). Note that it is your responsibility to delete the instantiated Visualization Manager by yourself. A complete description of a sample `main()` function is described in `examples/basic/B1/exampleB1.cc`.

2.11.5 Sample Visualization Sessions

Most Geant4 examples include a vis.mac. Run that macro to see a typical visualization. Read the comments in the macro to learn a little bit about some visualization commands. The vis.mac also includes commented-out optional visualization commands. By uncommenting some of these, you can see additional visualization features.

2.11.6 For More Information on Geant4 Visualization

See the *Visualization* part of this user guide.

CHAPTER

THREE

TOOLKIT FUNDAMENTALS

3.1 Class Categories and Domains

3.1.1 What is a class category?

In the design of a large software system such as Geant4, it is essential to partition it into smaller logical units. This makes the design well organized and easier to develop. Once the logical units are defined independent to each other as much as possible, they can be developed in parallel without serious interference.

In object-oriented analysis and design methodology by Grady Booch *[Booch1994]*, class categories are used to create logical units. They are defined as "clusters of classes that are themselves cohesive, but are loosely coupled relative to other clusters." This means that a class category contains classes which have a close relationship (for example, the "has-a" relation). However, relationships between classes which belong to different class categories are weak, i.e., only limited classes of these have "uses" relations. The class categories and their relations are presented by a class category diagram. The class category diagram designed for Geant4 is shown in the figure below (*Class categories*). Each box in the figure represents a class category, and a "uses" relation by a straight line. The circle at an end of a straight line means the class category which has this circle uses the other category.

The file organization of the Geant4 codes follows basically the structure of this class cateogory. This *User's Manual* is also organized according to class categories.

In the development and maintenance of Geant4, one software team will be assigned to a class category. This team will have a responsibility to develop and maintain all classes belonging to the class category.

3.1.2 Class categories in Geant4

The following is a brief summary of the role of each class category in Geant4.

1. **Run and Event**
   These are categories related to the generation of events, interfaces to event generators, and any secondary particles produced. Their roles are principally to provide particles to be tracked to the Tracking Management.
2. **Tracking and Track**
   These are categories related to propagating a particle by analyzing the factors limiting the step and applying the relevant physics processes. The important aspect of the design was that a generalized Geant4 physics process (or interaction) could perform actions, along a tracking step, either localized in space, or in time, or distributed in space and time (and all the possible combinations that could be built from these cases).
3. **Geometry and Magnetic Field**
   These categories manage the geometrical definition of a detector (solid modeling) and the computation of distances to solids (also in a magnetic field). The Geant4 geometry solid modeler is based on the ISO STEP standard and it is fully compliant with it. A key feature of the Geant4 geometry is that the volume definitions are independent of the solid representation. By this abstract interface for the G4 solids, the tracking component works identically for various representations. The treatment of the propagation in the presence of fields

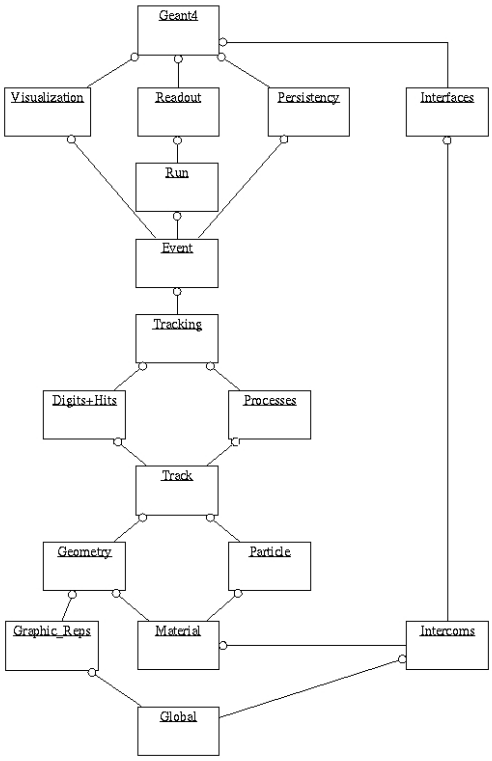


Fig. 3.1: Class categories in Geant4.

has been provided within specified accuracy. An OO design allows to exchange different numerical algorithms and/or different fields (not only B-field), without affecting any other component of the toolkit.

4. **Particle Definition and Matter**
   These two categories manage the the definition of materials and particles.
5. **Physics**
   This category manages all physics processes participating in the interactions of particles in matter. The abstract interface of physics processes allows multiple implementations of physics models per interaction or per channel. Models can be selected by energy range, particle type, material, etc. Data encapsulation and polymorphism make it possible to give transparent access to the cross sections (independently of the choice of reading from an ascii file, or of interpolating from a tabulated set, or of computing analytically from a formula). Electromagnetic and hadronic physics were handled in a uniform way in such a design, opening up the physics to the users.
6. **Hits and Digitization**
   These two categories manage the creation of hits and their use for the digitization phase. The basic design and implementation of the Hits and Digi had been realized, and also several prototypes, test cases and scenarios had been developed before the alpha-release. Volumes (not necessarily the ones used by the tracking) are aggregated in sensitive detectors, while hits collections represent the logical read out of the detector. Different ways of creating and managing hits collections had been delivered and tested, notably for both single hits and calorimetry hits types. In all cases, hits collections had been successfully stored into and retrieved from an Object Data Base Management System.
7. **Visualization**
   This manages the visualization of solids, trajectories and hits, and interacts with underlying graphical libraries (the Visualization class category). The basic and most frequently used graphics functionality had been implemented already by the alpha-release. The OO design of the visualization component allowed us to develop several drivers independently, such as for OpenGL, Qt and OpenInventor (for X11 and Windows), DAWN, Postscript (via DAWN) and VRML.

8. **Interfaces**
   This category handles the production of the graphical user interface (GUI) and the interactions with external software (OODBMS, reconstruction etc.).

3.2 Global Usage Classes

The "global" category in Geant4 collects all classes, types, structures and constants which are considered of general use within the Geant4 toolkit. This category also defines the interface with third-party software libraries (CLHEP, STL, etc.) and system-related types, by defining, where appropriate, `typedefs` according to the Geant4 code conventions.

3.2.1 Signature of Geant4 classes

In order to keep an homogeneous naming style, and according to the Geant4 coding style conventions, each class part of the Geant4 kernel has its name beginning with the prefix `G4`, e.g., `G4VHit, G4GeometryManager, G4ProcessVector`, etc. Instead of the raw C types, `G4` types are used within the Geant4 code. For the basic numeric types (`int, float, double`, etc.), different compilers and different platforms provide different value ranges. In order to assure portability, the use of `G4int, G4float, G4double, G4bool`, globally defined, is preferable. `G4` types implement the right generic type for a given architecture.

Basic types

The basic types in Geant4 are considered to be the following:

- `G4int`,
- `G4long`,
- `G4float`,
- `G4double`,
- `G4bool`,
- `G4complex`,
- `G4String`.

which currently consist of simple `typedefs` to respective types defined in the **CLHEP**, **STL** or system libraries. Most definitions of these basic types come with the inclusion of a single header file, `globals.hh`. This file also provides inclusion of required system headers, as well as some global utility functions needed and used within the Geant4 kernel.

Typedefs to CLHEP classes and their usage

The following classes are `typedefs` to the corresponding classes of the **CLHEP** (**Computing Library for High Energy Physics**) distribution. For more detailed documentation please refer to the CLHEP reference guide and the CLHEP user manual.

- `G4ThreeVector, G4RotationMatrix, G4LorentzVector` and `G4LorentzRotation`:
  Vector classes: defining 3-component (x,y,z) vector entities, rotation of such objects as 3x3 matrices, 4-component (x,y,z,t) vector entities and their rotation as 4x4 matrices.
- `G4Plane3D, G4Transform3D, G4Normal3D, G4Point3D, G4Scale3D`, and `G4Vector3D`:
  Geometrical classes: defining geometrical entities and transformations in 3D space.

3.2.2 The *HEPRandom* module in CLHEP

The *HEPRandom* module, originally part of the Geant4 kernel, and now distributed as a module of **CLHEP**, has been designed and developed starting from the *Random* class of MC++, the original **CLHEP**'s *HepRandom* module and the **Rogue Wave** approach in the **Math.h++** package. For detailed documentation on the *HEPRandom* classes see the CLHEP reference guide and the CLHEP user manual.

Information written in this manual is extracted from the original manifesto distributed with the *HEPRandom* package.

The *HEPRandom* module consists of classes implementing different random `engines` and different random `distributions`. A distribution associated to an engine constitutes a random `generator`. A distribution class can collect different algorithms and different calling sequences for each method to define distribution parameters or range-intervals. An engine implements the basic algorithm for pseudo-random numbers generation.

There are 3 different ways of shooting random values:

1. Using the static generator defined in the *HepRandom* class: random values are shot using static methods `shoot()` defined for each distribution class. The static generator will use, as default engine, a *HepJamesRandom* object, and the user can set its properties or change it with a new instantiated engine object by using the static methods defined in the *HepRandom* class.
2. Skipping the static generator and specifying an engine object: random values are shot using static methods `shoot(*HepRandomEngine)` defined for each distribution class. The user must instantiate an engine object and give it as argument to the shoot method. The generator mechanism will then be by-passed by using the basic `flat()` method of the specified engine. The user must take care of the engine objects he/she instantiates.
3. Skipping the static generator and instantiating a distribution object: random values are shot using `fire()` methods (NOT static) defined for each distribution class. The user must instantiate a distribution object giving as argument to the constructor an engine by pointer or by reference. By doing so, the engine will be associated to the distribution object and the generator mechanism will be by-passed by using the basic `flat()` method of that engine.

In this guide, we'll only focus on the static generator (point 1.), since the static interface of *HEPRandom* is the only one used within the Geant4 toolkit.

HEPRandom engines

The class *HepRandomEngine* is the abstract class defining the interface for each random engine. It implements the `getSeed()` and `getSeeds()` methods which return the `initial seed` value and the initial array of seeds (if any) respectively. Many concrete random engines can be defined and added to the structure, simply making them inheriting from *HepRandomEngine*. Several different engines are currently implemented in *HepRandom*, we describe here five of them:

- *HepJamesRandom*
  It implements the algorithm described in *F.James, Comp. Phys. Comm. 60 (1990) 329* for pseudo-random number generation. This is the default random engine for the static generator; it will be invoked by each distribution class unless the user sets a different one.
- *DRand48Engine*
  Random engine using the `drand48()` and `srand48()` system functions from C standard library to implement the `flat()` basic distribution and for setting seeds respectively. *DRand48Engine* uses the `seed48()` function from C standard library to retrieve the current internal status of the generator, which is represented by 3 short values. *DRand48Engine* is the only engine defined in *HEPRandom* which intrinsically works in 32 bits precision. Copies of an object of this kind are not allowed.
- *MixMaxRng*
  Random number engine implementing the MixMax Matrix Generator of Pseudorandom Numbers generator proposed by *N.Z.Akopov, G.K.Saviddy and N.G.Ter-Arutyunian, J.Compt.Phy. 97, (1991) 573* and *G.Savvidy and N.Savvidy, J.Comput.Phys. 97 (1991) 566*.

- *RanluxEngine*
  The algorithm for *RanluxEngine* has been taken from the original implementation in FORTRAN77 by Fred James, part of the **MATHLIB HEP** library. The initialisation is carried out using a Multiplicative Congruential generator using formula constants of L'Ecuyer as described in *F.James, Comp. Phys. Comm. 60 (1990) 329-344*. The engine provides five different luxury levels for quality of random generation. When instantiating a *RanluxEngine*, the user can specify the luxury level to the constructor (if not, the default value 3 is taken). For example:

```
RanluxEngine theRanluxEngine(seed,4);
// instantiates an engine with `seed' and the best luxury-level
... or
RanluxEngine theRanluxEngine;
// instantiates an engine with default seed value and luxury-level
...
```

  The class provides a `getLuxury()` method to get the engine luxury level.
  The `SetSeed()` and `SetSeeds()` methods to set the initial seeds for the engine, can be invoked specifying the luxury level. For example:

```
// static interface
HepRandom::setTheSeed(seed,4);  // sets the seed to `seed' and luxury to 4
HepRandom::setTheSeed(seed);    // sets the seed to `seed' keeping
                                // the current luxury level
```

- *RanecuEngine*
  The algorithm for *RanecuEngine* is taken from the one originally written in FORTRAN77 as part of the **MATHLIB HEP** library. The initialisation is carried out using a Multiplicative Congruential generator using formula constants of L'Ecuyer as described in *F.James, Comp. Phys. Comm. 60 (1990) 329-344*. Handling of seeds for this engine is slightly different than the other engines in *HEPRandom*. Seeds are taken from a seed table given an index, the `getSeed()` method returns the current index of seed table. The `setSeeds()` method will set seeds in the local `SeedTable` at a given position index (if the index number specified exceeds the table's size, `[index%size]` is taken). For example:

```
// static interface
const G4long* table_entry;
table_entry = HepRandom::getTheSeeds();
// it returns a pointer `table_entry' to the local SeedTable
// at the current `index' position. The couple of seeds
// accessed represents the current `status' of the engine itself !
...
G4int index=n;
G4long seeds[2];
HepRandom::setTheSeeds(seeds,index);
// sets the new `index' for seeds and modify the values inside
// the local SeedTable at the `index' position. If the index
// is not specified, the current index in the table is considered.
...
```

  The `setSeed()` method resets the current 'status' of the engine to the original seeds stored in the static table of seeds in *HepRandom*, at the specified index.

Except for the *RanecuEngine*, for which the internal status is represented by just a couple of longs, all the other engines have a much more complex representation of their internal status, which currently can be obtained only through the methods `saveStatus()`, `restoreStatus()` and `showStatus()`, which can also be statically called from *HepRandom*. The status of the generator is needed for example to be able to reproduce a run or an event in a run at a given stage of the simulation.

RanecuEngine is probably the most suitable engine for this kind of operation, since its internal status can be fetched/reset by simply using `getSeeds()`/`setSeeds()` (`getTheSeeds()`/`setTheSeeds()` for the static interface in *HepRandom*).

The static interface in the *HepRandom* class

HepRandom a singleton class and using a *HepJamesRandom* engine as default algorithm for pseudo-random number generation. *HepRandom* defines a static private data member, `theGenerator`, and a set of static methods to manipulate it. By means of `theGenerator`, the user can change the underlying engine algorithm, get and set the seeds, and use any kind of defined random distribution. The static methods `setTheSeed()` and `getTheSeed()` will set and get respectively the `initial` seed to the main engine used by the static generator. For example:

```
HepRandom::setTheSeed(seed);  // to change the current seed to 'seed'
int startSeed = HepRandom::getTheSeed();  // to get the current initial seed
HepRandom::saveEngineStatus();    // to save the current engine status on file
HepRandom::restoreEngineStatus(); // to restore the current engine to a previous
                                  // saved configuration
HepRandom::showEngineStatus();    // to display the current engine status to stdout
...
int index=n;
long seeds[2];
HepRandom::getTheTableSeeds(seeds,index);
  // fills `seeds' with the values stored in the global
  // seedTable at position `index'
```

Only one random engine can be active at a time, the user can decide at any time to change it, define a new one (if not done already) and set it. For example:

```
RanecuEngine theNewEngine;
HepRandom::setTheEngine(&theNewEngine);
 ...
```

or simply setting it to an old instantiated engine (the old engine status is kept and the new random sequence will start exactly from the last one previously interrupted). For example:

```
HepRandom::setTheEngine(&myOldEngine);
```

Other static methods defined in this class are:

- `void setTheSeeds(const G4long* seeds, G4int)`
- `const G4long* getTheSeeds()`
  To set/get an array of seeds for the generator, in the case of a *RanecuEngine* this corresponds also to set/get the current status of the engine.
- `HepRandomEngine* getTheEngine()`
  To get a pointer to the current engine used by the static generator.

HEPRandom distributions

A distribution-class can collect different algorithms and different calling sequences for each method to define distribution parameters or range-intervals; it also collects methods to fill arrays, of specified size, of random values, according to the distribution. This class collects either static and not static methods. A set of distribution classes are defined in *HEPRandom*. Here is the description of some of them:

- *RandFlat* Class to shoot flat random values (integers or double) within a specified interval. The class provides also methods to shoot just random bits.
- *RandExponential* Class to shoot exponential distributed random values, given a mean (default mean = 1)
- *RandGauss* Class to shoot Gaussian distributed random values, given a mean (default = 0) or specifying also a deviation (default = 1). Gaussian random numbers are generated two at the time, so every other time a number is shot, the number returned is the one generated the time before.
- *RandBreitWigner* Class to shoot numbers according to the Breit-Wigner distribution algorithms (plain or mean^2).

- *RandPoisson* Class to shoot numbers according to the Poisson distribution, given a mean (default = 1) (Algorithm taken from *W.H.Press et al., Numerical Recipes in C, Second Edition*).

3.2.3 The *HEPNumerics* module

A set of classes implementing numerical algorithms has been developed in Geant4. Most of the algorithms and methods have been implemented mainly based on recommendations given in the books:

- B.H. Flowers, `An introduction to Numerical Methods In C++`, Claredon Press, Oxford 1995.
- M. Abramowitz, I. Stegun, `Handbook of mathematical functions`, DOVER Publications INC, New York 1965 ; chapters 9, 10, and 22.

This set of classes includes:

- `G4ChebyshevApproximation` Class creating the Chebyshev approximation for a function pointed by fFunction data member. The Chebyshev polynomial approximation provides an efficient evaluation of the minimax polynomial, which (among all polynomials of the same degree) has the smallest maximum deviation from the true function.
- `G4DataInterpolation` Class providing methods for data interpolations and extrapolations: Polynomial, Cubic Spline, ...
- `G4GaussChebyshevQ`
- `G4GaussHermiteQ`
- `G4GaussJacobiQ`
- `G4GaussLaguerreQ` Classes implementing the Gauss-Chebyshev, Gauss-Hermite, Gauss-Jacobi, Gauss-Laguerre and Gauss-Legendre quadrature methods. Roots of orthogonal polynomials and corresponding weights are calculated based on iteration method (by bisection Newton algorithm).
- `G4Integrator` Template class collecting integrator methods for generic functions (Legendre, Simpson, Adaptive Gauss, Laguerre, Hermite, Jacobi).
- `G4SimpleIntegration` Class implementing simple numerical methods (Trapezoidal, MidPoint, Gauss, Simpson, Adaptive Gauss, for integration of functions with signature: double f(double).

3.2.4 General management classes

The `global` category defines also a set of `utility` classes generally used within the kernel of Geant4. These classes include:

- `G4Allocator`
  A class for fast allocation of objects to the heap through paging mechanism. It's meant to be used by associating it to the object to be allocated and defining for it `new` and `delete` operators via `MallocSingle()` and `FreeSingle()` methods of `G4Allocator`.

  Note: `G4Allocator` assumes that objects being allocated have all the same size for the type they represent. For this reason, classes which are handled by `G4Allocator` should *avoid* to be used as base-classes for others. Similarly, base-classes of sub-classes handled through `G4Allocator` should not define their (eventually empty) virtual destructors inlined; such measure is necessary in order also to prevent bad aliasing optimisations by compilers which may potentially lead to crashes in the attempt to free allocated chunks of memory when using the base-class pointer or not.

  The list of allocators implicitly defined and used in Geant4 is reported here:
  - events (`G4Event`): anEventAllocator
  - tracks (`G4Track`): aTrackAllocator
  - stacked tracks (`G4StackedTrack`): aStackedTrackAllocator
  - primary particles (`G4PrimaryParticle`): aPrimaryParticleAllocator
  - primary vertices (`G4PrimaryVertex`): aPrimaryVertexAllocator

  - decay products (G4DecayProducts): aDecayProductsAllocator
  - digits collections of an event (G4DCofThisEvent): anDCoTHAllocator
  - digits collections (G4DigiCollection): aDCAllocator
  - hits collections of an event (G4HCofThisEvent): anHCoTHAllocator
  - hits collections (G4HitsCollection): anHCAllocator
  - touchable histories (G4TouchableHistory): aTouchableHistoryAllocator
  - trajectories (G4Trajectory): aTrajectoryAllocator
  - trajectory points (G4TrajectoryPoint): aTrajectoryPointAllocator
  - trajectory containers (G4TrajectoryContainer): aTrajectoryContainerAllocator
  - navigation levels (G4NavigationLevel): aNavigationLevelAllocator
  - navigation level nodes (G4NavigationLevelRep): aNavigLevelRepAllocator
  - reference-counted handles (G4ReferenceCountedHandle<X>): aRCHAllocator
  - counted objects (G4CountedObject<X>): aCountedObjectAllocator
  - HEPEvt primary particles (G4HEPEvtParticle): aHEPEvtParticleAllocator
  - electron occupancy objects(G4ElectronOccupancy): aElectronOccupancyAllocator
  - "rich" trajectories (G4RichTrajectory): aRichTrajectoryAllocator
  - "rich" trajectory points (G4RichTrajectoryPoint): aRichTrajectoryPointAllocator
  - "smooth" trajectories (G4SmoothTrajectory): aSmoothTrajectoryAllocator
  - "smooth" trajectory points (G4SmoothTrajectoryPoint): aSmoothTrajectoryPointAllocator
  - "ray" trajectories (G4RayTrajectory): G4RayTrajectoryAllocator
  - "ray" trajectory points (G4RayTrajectoryPoint): G4RayTrajectoryPointAllocator

  For each of these allocators, accessible from the global namespace, it is possible to monitor the allocation in their memory pools or force them to release the allocated memory (for example at the end of a run):

```
// Return the size of the total memory allocated for tracks
//
aTrackAllocator.GetAllocatedSize();

// Return allocated storage for tracks to the free store
//
aTrackAllocator.ResetStorage();
```

- G4ReferenceCountedHandle
  Template class acting as a smart pointer and wrapping the type to be counted. It performs the reference counting during the life-time of the counted object.
- G4FastVector
  Template class defining a vector of pointers, not performing boundary checking.
- G4PhysicsVector
  Defines a physics vector which has values of energy-loss, cross-section, and other physics values of a particle in matter in a given range of the energy, momentum, etc. This class serves as the base class for a vector having various energy scale, for example like 'log' (G4PhysicsLogVector) 'linear' (G4PhysicsLinearVector), 'free' (G4PhysicsFreeVector), etc.
- G4LPhysicsFreeVector
  Implements a free vector for low energy physics cross-section data. A subdivision method is used to find the energy|momentum bin.
- G4PhysicsOrderedFreeVector
  A physics ordered free vector inherits from G4PhysicsVector. It provides, in addition, a method for the user to insert energy/value pairs in sequence. Methods to retrieve the max and min energies and values from the vector are also provided.
- G4Timer
  Utility class providing methods to measure elapsed user/system process time. Uses <sys/times.h> and <unistd.h> - POSIX.1.
- G4UserLimits
  Class collecting methods for get and set any kind of step limitation allowed in Geant4.
- G4UnitsTable
  Placeholder for the system of units in Geant4.

3.3 System of units

3.3.1 Basic units

Geant4 offers the user the possibility to choose and use the preferred units for any quantity. In fact, Geant4 takes care of the units. Internally a consistent set on units based on the `HepSystemOfUnits` is used:

```
millimeter              (mm)
nanosecond              (ns)
Mega electron Volt      (MeV)
positron charge         (eplus)
degree Kelvin           (kelvin)
the amount of substance (mole)
luminous intensity      (candela)
radian                  (radian)
steradian               (steradian)
```

All other units are defined from the basic ones.

For instance:

```
millimeter = mm = 1;
meter = m = 1000*mm;
...
m3 = m*m*m;
...
```

In the file `$CLHEP_BASE_DIR/include/CLHEP/Units/SystemOfUnits.h` from the CLHEP installation, one can find all untis definitions.

One can also change the system of units to be used by the kernel.

3.3.2 Input your data

Avoid 'hard coded' data

The user **must** give the units for the data to introduce:

```
G4double Size = 15*km, KineticEnergy = 90.3*GeV, density = 11*mg/cm3;
```

Geant4 assumes that these specifications for the units are respected, in order to assure independence from the units chosen in the client application.

If units are not specified in the client application, data are implicitly treated in internal Geant4 system units; this practice is however strongly discouraged.

If the data set comes from an array or from an external file, it is strongly recommended to set the units as soon as the data are read, before any treatment. For instance:

```
for (int j=0, j<jmax, j++) CrossSection[j] *= millibarn;
...
my calculations
...
```

Interactive commands

Some built-in commands from the User Interface (UI) also require units to be specified.

For instance:

```
/gun/energy 15.2 keV
/gun/position 3 2 -7 meter
```

If units are not specified, or are not valid, the command is refused.

3.3.3 Output your data

You can output your data with the wished units. To do so, it is sufficient to **divide** the data by the corresponding unit:

```
G4cout << KineticEnergy/keV << " keV";
G4cout << density/(g/cm3)   << " g/cm3";
```

Of course, `G4cout << KineticEnergy` will print the energy in the internal units system.

There is another way to output the data. Let Geant4 choose the most appropriate units for the actual numerical value of the data. It is sufficient to specify to which category the data belong to (Length, Time, Energy, etc.). For example:

```
G4cout << G4BestUnit(StepSize, "Length");
```

`StepSize` will be printed in km, m, mm, fermi, etc. depending of its actual value.

3.3.4 Introduce new units

If wished to introduce new units, there are two methods:

- You can extend the file `SystemOfUnits.h`

  ```
  #include "SystemOfUnits.h"

  static const G4double inch = 2.54*cm;
  ```

  Using this method, it is not easy to define composed units. It is better to do the following:
- Instantiate an object of the class `G4UnitDefinition`. These objects are owned by the global `G4UnitsTable` at construction, and must not be deleted by the user.

  ```
  new G4UnitDefinition ( name, symbol, category, value )
  ```

  For example: define a few units for speed

  ```
  new G4UnitDefinition ( "km/hour" , "km/h", "Speed", km/(3600*s) );
  new G4UnitDefinition ( "meter/ns", "m/ns", "Speed", m/ns );
  ```

  The category "Speed" does not exist by default in `G4UnitsTable`, but it will be created automatically. The class `G4UnitDefinition` is defined in `source/global/management/G4UnitsTable.hh`.

3.3.5 Print the list of units

You can print the list of units with the static function: `G4UnitDefinition::PrintUnitsTable();` or with the interactive command: `/units/list`

3.4 Run

3.4.1 Basic concept of *Run*

In Geant4, *Run* is the largest unit of simulation. A run consists of a sequence of events. Within a run, the detector geometry, the set up of sensitive detectors, and the physics processes used in the simulation should be kept unchanged. A run is represented by a `G4Run` class object. A run starts with `BeamOn()` method of `G4RunManager`.

Representation of a run

`G4Run` represents a run. It has a run identification number, which should be set by the user, and the number of events simulated during the run. Please note that the run identification number is not used by the Geant4 kernel, and thus can be arbitrarily assigned at the user's convenience.

`G4Run` has pointers to the tables `G4VHitsCollection` and `G4VDigiCollection`. These tables are associated in case *sensitive detectors* and *digitizer modules* are simulated, respectively. The usage of these tables will be mentioned in *Hits* and *Digitization*.

`G4Run` has two virtual methods, and thus you can extend `G4Run` class. In particular if you use Geant4 in multi-threaded mode and need to accumulate values, these two virtual method must be overwritten to specify how such values should be collected firstly for a worker thread, and then for the entire run. These virtual methods are the following.

`virtual void RecordEvent(const G4Event*)` Method to be overwritten by the user for recording events in this (thread-local) run. At the end of the implementation, G4Run base-class method for must be invoked for recording data members in the base class.

`void Merge(const G4Run*)` Method to be overwritten by the user for merging local Run object to the global Run object. At the end of the implementation, G4Run base-class method for must be invoked for merging data members in the base class.

Manage the run procedures

`G4RunManager` manages the procedures of a run. In the constructor of `G4RunManager`, all of the manager classes in Geant4 kernel, except for some static managers, are constructed. These managers are deleted in the destructor of `G4RunManager`. `G4RunManager` must be a singleton created in the user's `main()` program; the pointer to this singleton object can be obtained by other code using the `GetRunManager()` static method.

As already mentioned in *How to Define the main() Program*, all of the *user initialization* classes defined by the user should be assigned to `G4RunManager` before starting initialization of the Geant4 kernel. The assignments of these user classes are done by `SetUserInitialization()` methods. All user classes defined by the Geant4 kernel will be summarized in *User Actions*.

`G4RunManager` has several public methods, which are listed below.

`Initialize()` All initializations required by the Geant4 kernel are triggered by this method. Initializations are:

- construction of the detector geometry and set up of sensitive detectors and/or digitizer modules,
- construction of particles and physics processes,
- calculation of cross-section tables.

This method is thus mandatory before proceeding to the first run. This method will be invoked automatically for the second and later runs in case some of the initialized quantities need to be updated.

`BeamOn(G4int numberOfEvent)` This method triggers the actual simulation of a run, that is, an event loop. It takes an integer argument which represents the number of events to be simulated.

`GetRunManager()` This static method returns the pointer to the G4RunManager singleton object.

GetCurrentEvent() This method returns the pointer to the G4Event object which is currently being simulated. This method is available only when an event is being processed. At this moment, the application state of Geant4, which is explained in the following sub-section, is "EventProc". When Geant4 is in a state other than "EventProc", this method returns null. Please note that the return value of this method is const G4Event * and thus you cannot modify the contents of the object.

SetNumberOfEventsToBeStored(G4int nPrevious) When simulating the "pile up" of more than one event, it is essential to access more than one event at the same moment. By invoking this method, G4RunManager keeps nPrevious G4Event objects. This method must be invoked before proceeding to BeamOn().

GetPreviousEvent(G4int i_thPrevious) The pointer to the i_thPrevious G4Event object can be obtained through this method. A pointer to a const object is returned. It is inevitable that i_thPrevious events must have already been simulated in the same run for getting the i_thPrevious event. Otherwise, this method returns null.

AbortRun() This method should be invoked whenever the processing of a run must be stopped. It is valid for GeomClosed and EventProc states. Run processing will be safely aborted even in the midst of processing an event. However, the last event of the aborted run will be incomplete and should not be used for further analysis.

Run manager classes for multi-threading mode

`G4MTRunManager` is the replacement of `G4RunManager` for multi-threading mode. At the very end of `Initialize()` method, `G4MTRunManager` creates and starts worker threads. The event each thread is tasked is in first-come-first-served basis, so that event numbers each thread has are not sequential.

`G4WorkerRunManager` is the local RunManager automatically instantiated by `G4MTRunManager` to take care of initialization and event handling of a thread. Both `G4MTRunManager` and `G4WorkerRunManager` are derived classes of `G4RunManager` base class.

The static method `G4RunManager::GetRunManager()` returns the following pointer.

- It returns the pointer to the `G4WorkerRunManager` of the local thread when it is invoked from thread-local object.
- It returns the pointer to the `G4MTRunManager` when it is invoked from shared object.
- It returns the pointer to the base `G4RunManager` if it is used in the sequential mode.

`G4RunManager` has a method `GetRunManagerType()` that returns an enum named `RMType` to indicate what kind of `RunManager` it is. `RMType` is defined as `{ sequentialRM, masterRM, workerRM }`. From the thread-local object, a static method `G4MTRunManager::GetMasterRunManager()` is available to access to `G4MTRunManager`. From a worker thread, the user may access to, for example, detector construction (it is a shared class) through this `GetMasterRunManager()` method.

G4UserRunAction

`G4UserRunAction` is one of the *user action* classes from which you can derive your own concrete class. This base class has three virtual methods as follows:

GenerateRun() This method is invoked at the beginning of the BeamOn() method but after confirmation of the conditions of the Geant4 kernel. This method should be used to instantiate a user-specific run class object.

BeginOfRunAction() This method is invoked at the beginning of the BeamOn() method but after confirmation of the conditions of the Geant4 kernel. Likely uses of this method include:

- setting a run identification number,
- booking histograms,
- setting run specific conditions of the sensitive detectors and/or digitizer modules (e.g., dead channels).

EndOfRunAction() This method is invoked at the very end of the BeamOn() method. Typical use cases of this method are

- store/print histograms,

- manipulate run summaries.

3.4.2 Geant4 as a state machine

Geant4 is designed as a state machine. Some methods in Geant4 are available for only a certain state(s). `G4RunManager` controls the state changes of the Geant4 application. States of Geant4 are represented by the enumeration `G4ApplicationState`. It has six states through the life cycle of a Geant4 application.

`G4State_PreInit` state A Geant4 application starts with this state. The application needs to be initialized when it is in this state. The application occasionally comes back to this state if geometry, physics processes, and/or cut-off have been changed after processing a run.

`G4State_Init` state The application is in this state while the Initialize() method of G4RunManager is being invoked. Methods defined in any user initialization classes are invoked during this state.

`G4State_Idle` state The application is ready for starting a run.

`G4State_GeomClosed` state When BeamOn() is invoked, the application proceeds to this state to process a run. Geometry, physics processes, and cut-off cannot be changed during run processing.

`G4State_EventProc` state A Geant4 application is in this state when a particular event is being processed. GetCurrentEvent() and GetPreviousEvent() methods of G4RunManager are available only at this state.

`G4State_Quit` state When the destructor of G4RunManager is invoked, the application comes to this "dead end" state. Managers of the Geant4 kernel are being deleted and thus the application cannot come back to any other state.

`G4State_Abort` state When a G4Exception occurs, the application comes to this "dead end" state and causes a core dump. The user still has a hook to do some "safe" opperations, e.g. storing histograms, by implementing a user concrete class of G4VStateDependent. The user also has a choice to suppress the occurence of G4Exception by a UI command /control/suppressAbortion. When abortion is suppressed, you will still get error messages issued by G4Exception, and there is NO guarantee of a correct result after the G4Exception error message.

`G4StateManager` belongs to the *intercoms* category.

3.4.3 User's hook for state change

In case the user wants to do something at the moment of state change of Geant4, the user can create a concrete class of the `G4VStateDependent` base class. For example, the user can store histograms when G4Exception occurs and Geant4 comes to the *Abort* state, but before the actual core dump.

The following is an example user code which stores histograms when Geant4 becomes to the *Abort* state. This class object should be mabe in, for example `main()`, by the user code. This object will be automatically registered to `G4StateManager` at its construction.

Listing 3.1: Header file of UserHookForAbortState

```
#ifndef UserHookForAbortState_H
#define UserHookForAbortState_H 1

#include "G4VStateDependent.hh"

class UserHookForAbortState : public G4VStateDependent
{
 public:
  UserHookForAbortState();   // constructor
  ~UserHookForAbortState();  // destructor

  virtual G4bool Notify(G4ApplicationState requiredState);
};
```

Listing 3.2: Source file of UserHookForAbortState

```
#include "UserHookForAbortState.hh"

UserHookForAbortState::UserHookForAbortState() {;}
UserHookForAbortState::~UserHookForAbortState() {;}

G4bool UserHookForAbortState::Notify(G4ApplicationState requiredState)
{
  if(requiredState!=Abort) return true;

  // Do book keeping here

  return true;
}
```

3.4.4 Customizing the Run Manager

Virtual Methods in the Run Manager

`G4RunManager` is a concrete class with a complete set of functionalities for managing the Geant4 kernel. It is the only manager class in the Geant4 kernel which must be constructed in the `main()` method of the user's application. Thus, instead of constructing the `G4RunManager` provided by Geant4, you are free to construct your own `RunManager`. It is recommended, however, that your `RunManager` inherit `G4RunManager`. For this purpose, `G4RunManager` has various virtual methods which provide all the functionalities required to handle the Geant4 kernel. Hence, your customized run manager need only override the methods particular to your needs; the remaining methods in `G4RunManager` base class can still be used. A summary of the available methods is presented here:

`public: virtual void Initialize();` main entry point of Geant4 kernel initialization
`protected: virtual void InitializeGeometry();` geometry construction
`protected: virtual void InitializePhysics();` physics processes construction
`public: virtual void BeamOn(G4int n_event);` main entry point of the event loop
`protected: virtual G4bool ConfirmBeamOnCondition();` check the kernel conditions for the event loop
`protected: virtual void RunInitialization();` prepare a run
`protected: virtual void DoEventLoop(G4int n_events);` manage an event loop
`protected: virtual G4Event* GenerateEvent(G4int i_event);` generation of G4Event object
`protected: virtual void AnalyzeEvent(G4Event* anEvent);` storage/analysis of an event
`protected: virtual void RunTermination();` terminate a run
`public: virtual void DefineWorldVolume(G4VPhysicalVolume * worldVol);` set the world volume to G4Navigator
`public: virtual void AbortRun();` abort the run

Customizing the Event Loop

In `G4RunManager` the event loop is handled by the virtual method `DoEventLoop()`. This method is implemented by a `for` loop consisting of the following steps:

1. construct a `G4Event` object and assign to it primary vertex(es) and primary particles. This is done by the virtual `GeneratePrimaryEvent()` method.
2. send the `G4Event` object to `G4EventManager` for the detector simulation. *Hits* and *trajectories* will be associated with the `G4Event` object as a consequence.
3. perform bookkeeping for the current `G4Event` object. This is done by the virtual `AnalyzeEvent()` method.

DoEventLoop() performs the entire simulation of an event. However, it is often useful to split the above three steps into isolated application programs. If, for example, you wish to examine the effects of changing discriminator thresholds, ADC gate widths and/or trigger conditions on simulated events, much time can be saved by performing steps 1 and 2 in one program and step 3 in another. The first program need only generate the hit/trajectory information once and store it, perhaps in a database. The second program could then retrieve the stored G4Event objects and perform the digitization (analysis) using the above threshold, gate and trigger settings. These settings could then be changed and the digitization program re-run without re-generating the G4Events.

Changing the Detector Geometry

The detector geometry defined in your G4VUserDetectorConstruction concrete class can be changed during a run break (between two runs). Two different cases are considered.

The first is the case in which you want to delete the entire structure of your old geometry and build up a completely new set of volumes. For this case, you need to delete them by yoursef, and let *RunManager* invokes Construct() and ConstructSDandField() methods of your detector construction once again when RunManager starts the next run.

```
G4RunManager* runManager = G4RunManager::GetRunManager();
runManager->ReinitializeGeometry();
```

If this ReinitializeGeometry() is invoked, GeometryHasBeenModified() (discussed next) is automatically invoked. Presumably this case is rather rare. The second case is more frequent for the user.

The second case is the following. Suppose you want to move and/or rotate a particular piece of your detector component. This case can easily happen for a beam test of your detector. It is obvious for this case that you need not change the world volume. Rather, it should be said that your world volume (experimental hall for your beam test) should be big enough for moving/rotating your test detector. For this case, you can still use all of your detector geometries, and just use a Set method of a particular physical volume to update the transformation vector as you want. Thus, you don't need to re-set your world volume pointer to *RunManager*.

If you want to change your geometry for every run, you can implement it in the BeginOfRunAction() method of G4UserRunAction class, which will be invoked at the beginning of each run, or, derive the RunInitialization() method. Please note that, for both of the above mentioned cases, you need to let *RunManager* know "the geometry needs to be closed again". Thus, you need to invoke

```
runManager->GeometryHasBeenModified();
```

before proceeding to the next run. An example of changing geometry is given in a Geant4 tutorial in Geant4 Training kit #2.

Switch physics processes

In the InitializePhysics() method, G4VUserPhysicsList::Construct is invoked in order to define particles and physics processes in your application. Basically, you can not add nor remove any particles during execution, because particles are static objects in Geant4 (see *How to Specify Particles* and *Particles* for details). In addition, it is very difficult to add and/or remove physics processes during execution, because registration procedures are very complex, except for experts (see *How to Specify Physics Processes* and *Physics Processes*). This is why the initializePhysics() method is assumed to be invoked at once in Geant4 kernel initialization.

However, you can switch on/off physics processes defined in your G4VUserPhysicsList concrete class and also change parameters in physics processes during the run break.

You can use ActivateProcess() and InActivateProcess() methods of G4ProcessManager anywhere outside the event loop to switch on/off some process. You should be very careful to switch on/off processes inside the event loop, though it is not prohibited to use these methods even in the *EventProc* state.

It is a likely case to change cut-off values in a run. You can change `defaultCutValue` in `G4VUserPhysicsList` during the *Idle* state. In this case, all cross section tables need to be recalculated before the event loop. You should use the `CutOffHasBeenModified()` method when you change cut-off values so that the `SetCuts` method of your *PhysicsList* concrete class will be invoked.

3.4.5 Managing worker thread

`G4UserWorkerInitialization` is an additional user initialization class to be used only for the multi-threaded mode. The object of this class can be set to `G4MTRunManager`, but not to `G4RunManager`. `G4UserWorkerInitialization` class has five virtual methods as the user hooks which are invoked at several occasions of the life cycle of each thread.

`virtual void WorkerInitialize() const` This method is called after the tread is created but before the G4WorkerRunManager is instantiated.

`virtual void WorkerStart() const` This method is called once at the beginning of simulation job when kernel classes and user action classes have already instantiated but geometry and physics have not been yet initialized. This situation is identical to " PreInit " state in the sequential mode.

`virtual void WorkerStartRun() const` This method is called before an event loop. Geometry and physics have already been set up for the thread. All threads are synchronized and ready to start the local event loop. This situation is identical to " Idle " state in the sequential mode.

`virtual void WorkerRunEnd() const` This method is called for each thread when the local event loop is done, but before the synchronization over all worker threads.

`virtual void WorkerStop() const` This method is called once at the end of simulation job.

3.5 Event

3.5.1 Representation of an event

`G4Event` represents an event. An object of this class contains all inputs and outputs of the simulated event. This class object is constructed in `G4RunManager` and sent to `G4EventManager`. The event currently being processed can be obtained via the `getCurrentEvent()` method of `G4RunManager`.

3.5.2 Structure of an event

A `G4Event` object has four major types of information. Get methods for this information are available in `G4Event`.

Primary vertexes and primary particles Details are given in *Event Generator Interface*.

Trajectories Trajectories are stored in G4TrajectoryContainer class objects and the pointer to this container is stored in `G4Event`. The contents of a trajectory are given in *Trajectory and Trajectory Point*.

Hits collections Collections of hits generated by *sensitive detectors* are kept in `G4HCofThisEvent` class object and the pointer to this container class object is stored in `G4Event`. See *Hits* for the details.

Digits collections Collections of digits generated by *digitizer modules* are kept in `G4DCofThisEvent` class object and the pointer to this container class object is stored in `G4Event`. See *Digitization* for the details.

3.5.3 Mandates of `G4EventManager`

`G4EventManager` is the manager class to take care of one event. It is responsible for:

- converting `G4PrimaryVertex` and `G4PrimaryParticle` objects associated with the current `G4Event` object to `G4Track` objects. All of `G4Track` objects representing the primary particles are sent to `G4StackManager`.

- Pop one G4Track object from G4StackManager and send it to G4TrackingManager. The current G4Track object is deleted by G4EventManager after the track is simulated by G4TrackingManager, if the track is marked as "killed".
- In case the primary track is "suspended" or "postponed to next event" by G4TrackingManager, it is sent back to the G4StackManager. Secondary G4Track objects returned by G4TrackingManager are also sent to G4StackManager.
- When G4StackManager returns NULL for the "pop" request, G4EventManager terminates the current processing event.
- invokes the user-defined methods beginOfEventAction() and endOfEventAction() from the G4UserEventAction class. See *User Information Classes* for details.

3.5.4 Stacking mechanism

G4StackManager has three stacks, named *urgent*, *waiting* and *postpone-to-next-event*, which are objects of the G4TrackStack class. By default, all G4Track objects are stored in the *urgent* stack and handled in a "last in first out" manner. In this case, the other two stacks are not used. However, tracks may be routed to the other two stacks by the user-defined G4UserStackingAction concrete class.

If the methods of G4UserStackingAction have been overridden by the user, the *postpone-to-next-event* and *waiting* stacks may contain tracks. At the beginning of an event, G4StackManager checks to see if any tracks left over from the previous event are stored in the *postpone-to-next-event stack*. If so, it attemps to move them to the *urgent* stack. But first the PrepareNewEvent() method of G4UserStackingAction is called. Here tracks may be re-classified by the user and sent to the *urgent* or *waiting* stacks, or deferred again to the *postpone-to-next-event* stack. As the event is processed G4StackManager pops tracks from the *urgent* stack until it is empty. At this point the NewStage() method of G4UserStackingAction is called. In this method tracks from the *waiting* stack may be sent to the *urgent* stack, retained in the *waiting* stack or postponed to the next event.

Details of the user-defined methods of G4UserStackingAction and how they affect track stack management are given in *User Information Classes*.

3.6 Event Generator Interface

3.6.1 Structure of a primary event

Primary vertex and primary particle

The G4Event class object should have a set of primary particles when it is sent to G4EventManager via processOneEvent() method. It is the mandate of your G4VUserPrimaryGeneratorAction concrete class to send primary particles to the G4Event object.

The G4PrimaryParticle class represents a primary particle with which Geant4 starts simulating an event. This class object has information on particle type and its three momenta. The positional and time information of primary particle(s) are stored in the G4PrimaryVertex class object and, thus, this class object can have one or more G4PrimaryParticle class objects which share the same vertex. Primary vertexes and primary particles are associated with the G4Event object by a form of linked list.

A concrete class of G4VPrimaryGenerator, the G4PrimaryParticle object is constructed with either a pointer to G4ParticleDefinition or an integer number which represents P.D.G. particle code. For the case of some artificial particles, e.g., geantino, optical photon, etc., or exotic nuclear fragments, which the P.D.G. particle code does not cover, the G4PrimaryParticle should be constructed by G4ParticleDefinition pointer. On the other hand, elementary particles with very short life time, e.g., weak bosons, or quarks/gluons, can be instantiated as G4PrimaryParticle objects using the P.D.G. particle code. It should be noted that, even though primary particles with such a very short life time are defined, Geant4 will simulate only the particles which are defined as

G4ParticleDefinition class objects. Other primary particles will be simply ignored by G4EventManager. But it may still be useful to construct such "intermediate" particles for recording the origin of the primary event.

Forced decay channel

The G4PrimaryParticle class object can have a list of its daughter particles. If the parent particle is an "intermediate" particle, which Geant4 does not have a corresponding G4ParticleDefinition, this parent particle is ignored and daughters are assumed to start from the vertex with which their parent is associated. For example, a Z boson is associated with a vertex and it has positive and negative muons as its daughters, these muons will start from that vertex.

There are some kinds of particles which should fly some reasonable distances and, thus, should be simulated by Geant4, but you still want to follow the decay channel generated by an event generator. A typical case of these particles is B meson. Even for the case of a primary particle which has a corresponding G4ParticleDefinition, it can have daughter primary particles. Geant4 will trace the parent particle until it comes to decay, obeying multiple scattering, ionization loss, rotation with the magnetic field, etc. according to its particle type. When the parent comes to decay, instead of randomly choosing its decay channel, it follows the "pre-assigned" decay channel. To conserve the energy and the momentum of the parent, daughters will be Lorentz transformed according to their parent's frame.

3.6.2 Interface to a primary generator

G4HEPEvtInterface

Unfortunately, almost all event generators presently in use, commonly are written in FORTRAN. For Geant4, it was decided to not link with any FORTRAN program or library, even though the C++ language syntax itself allows such a link. Linking to a FORTRAN package might be convenient in some cases, but we will lose many advantages of object-oriented features of C++, such as robustness. Instead, Geant4 provides an ASCII file interface for such event generators.

G4HEPEvtInterface is one of G4VPrimaryGenerator concrete class and thus it can be used in your G4VUserPrimaryGeneratorAction concrete class. G4HEPEvtInterface reads an ASCII file produced by an event generator and reproduces G4PrimaryParticle objects associated with a G4PrimaryVertex object. It reproduces a full production chain of the event generator, starting with primary quarks, etc. In other words, G4HEPEvtInterface converts information stored in the /HEPEVT/ common block to an object-oriented data structure. Because the /HEPEVT/ common block is commonly used by almost all event generators written in FORTRAN, G4HEPEvtInterface can interface to almost all event generators currently used in the HEP community. The constructor of G4HEPEvtInterface takes the file name. *Example* shows an example how to use G4HEPEvtInterface. Note that an event generator is not assumed to give a place of the primary particles, the interaction point must be set before invoking GeneratePrimaryVertex() method.

Listing 3.3: An example code for G4HEPEvtInterface

```
#ifndef ExN04PrimaryGeneratorAction_h
#define ExN04PrimaryGeneratorAction_h 1

#include "G4VUserPrimaryGeneratorAction.hh"
#include "globals.hh"

class G4VPrimaryGenerator;
class G4Event;

class ExN04PrimaryGeneratorAction : public G4VUserPrimaryGeneratorAction
{
  public:
    ExN04PrimaryGeneratorAction();
    ~ExN04PrimaryGeneratorAction();
```

```
  public:
    void GeneratePrimaries(G4Event* anEvent);

  private:
    G4VPrimaryGenerator* HEPEvt;
};

#endif


#include "ExN04PrimaryGeneratorAction.hh"

#include "G4Event.hh"
#include "G4HEPEvtInterface.hh"

ExN04PrimaryGeneratorAction::ExN04PrimaryGeneratorAction()
{
  HEPEvt = new G4HEPEvtInterface("pythia_event.data");
}

ExN04PrimaryGeneratorAction::~ExN04PrimaryGeneratorAction()
{
  delete HEPEvt;
}

void ExN04PrimaryGeneratorAction::GeneratePrimaries(G4Event* anEvent)
{
  HEPEvt->SetParticlePosition(G4ThreeVector(0.*cm,0.*cm,0.*cm));
  HEPEvt->GeneratePrimaryVertex(anEvent);
}
```

Format of the ASCII file

An ASCII file, which will be fed by `G4HEPEvtInterface` should have the following format.

- The first line of each primary event should be an integer which represents the number of the following lines of primary particles.
- Each line in an event corresponds to a particle in the `/HEPEVT/` common. Each line has `ISTHEP, IDHEP, JDAHEP(1), JDAHEP(2), PHEP(1), PHEP(2), PHEP(3), PHEP(5)`. Refer to the `/HEPEVT/` manual for the meanings of these variables.

FORTRAN listing shows an example FORTRAN code to generate an ASCII file.

Listing 3.4: A FORTRAN example using the `/HEPEVT/` common.

```
*************************************************************
      SUBROUTINE HEP2G4
*
* Convert /HEPEVT/ event structure to an ASCII file
* to be fed by G4HEPEvtInterface
*
*************************************************************
      PARAMETER (NMXHEP=2000)
      COMMON/HEPEVT/NEVHEP,NHEP,ISTHEP(NMXHEP),IDHEP(NMXHEP),
     >JMOHEP(2,NMXHEP),JDAHEP(2,NMXHEP),PHEP(5,NMXHEP),VHEP(4,NMXHEP)
      DOUBLE PRECISION PHEP,VHEP
*
      WRITE(6,*) NHEP
      DO IHEP=1,NHEP
       WRITE(6,10)
     >  ISTHEP(IHEP),IDHEP(IHEP),JDAHEP(1,IHEP),JDAHEP(2,IHEP),
     >  PHEP(1,IHEP),PHEP(2,IHEP),PHEP(3,IHEP),PHEP(5,IHEP)
10     FORMAT(4I10,4(1X,D15.8))
      ENDDO
```

```
*
      RETURN
      END
```

Future interface to the new generation generators

Several activities have already been started for developing object-oriented event generators. Such new generators can be easily linked and used with a Geant4 based simulation. Furthermore, we need not distinguish a primary generator from the physics processes used in Geant4. Future generators can be a kind of physics process plugged-in by inheriting `G4VProcess`.

3.6.3 Event overlap using multiple generators

Your `G4VUserPrimaryGeneratorAction` concrete class can have more than one `G4VPrimaryGenerator` concrete class. Each `G4VPrimaryGenerator` concrete class can be accessed more than once per event. Using these class objects, one event can have more than one primary event.

One possible use is the following. Within an event, a `G4HEPEvtInterface` class object instantiated with a minimum bias event file is accessed 20 times and another `G4HEPEvtInterface` class object instantiated with a signal event file is accessed once. Thus, this event represents a typical signal event of LHC overlapping 20 minimum bias events. It should be noted that a simulation of event overlapping can be done by merging hits and/or digits associated with several events, and these events can be simulated independently. Digitization over multiple events will be mentioned in *Digitization*.

3.7 Event Biasing Techniques

3.7.1 Scoring, Geometrical Importance Sampling and Weight Roulette

Geant4 provides event biasing techniques which may be used to save computing time in such applications as the simulation of radiation shielding. These are *geometrical splitting* and *Russian roulette* (also called geometrical importance sampling), and *weight roulette*. Scoring is carried out by `G4MultiFunctionalDetector` (see *G4MultiFunctionalDetector and G4VPrimitiveScorer* and *Concrete classes of G4VPrimitiveScorer*) using the standard Geant4 scoring technique. Biasing specific scorers have been implemented and are described within `G4MultiFunctionalDetector` documentation. In this chapter, it is assumed that the reader is familiar with both the usage of Geant4 and the concepts of importance sampling. More detailed documentation may be found in the documents 'Scoring, geometrical importance sampling and weight roulette'. A detailed description of different use-cases which employ the sampling and scoring techniques can be found in the document 'Use cases of importance sampling and scoring in Geant4'.

The purpose of importance sampling is to save computing time by sampling less often the particle histories entering "less important" geometry regions, and more often in more "important" regions. Given the same amount of computing time, an importance-sampled and an analogue-sampled simulation must show equal mean values, while the importance-sampled simulation will have a decreased variance.

The implementation of scoring is independent of the implementation of importance sampling. However both share common concepts. *Scoring and importance sampling apply to particle types chosen by the user*, which should be borne in mind when interpreting the output of any biased simulation.

Examples on how to use scoring and importance sampling may be found in `examples/extended/biasing`.

Geometries

The kind of scoring referred to in this note and the importance sampling apply to spatial cells provided by the user.

A **cell** is a physical volume (further specified by it's replica number, if the volume is a replica). Cells may be defined in two kinds of geometries:

1. **mass geometry**: the geometry setup of the experiment to be simulated. Physics processes apply to this geometry.
2. **parallel-geometry**: a geometry constructed to define the physical volumes according to which scoring and/or importance sampling is applied.

The user has the choice to score and/or sample by importance the particles of the chosen type, according to mass geometry or to parallel geometry. It is possible to utilize several parallel geometries in addition to the mass geometry. This provides the user with a lot of flexibility to define separate geometries for different particle types in order to apply scoring or/and importance sampling.

Note: Parallel geometries should be constructed using the implementation as described in *Parallel Geometries*. There are a few conditions for parallel geometries:

- The world volume for parallel and mass geometries must be identical copies.
- Scoring and importance cells must not share boundaries with the world volume.

Changing the Sampling

Samplers are higher level tools which perform the necessary changes of the Geant4 sampling in order to apply importance sampling and weight roulette.

Variance reduction (and scoring through the `G4MultiFunctionalDetector`) may be combined arbitrarily for chosen particle types and may be applied to the mass or to parallel geometries.

The `G4GeometrySampler` can be applied equally to mass or parallel geometries with an abstract interface supplied by `G4VSampler`. `G4VSampler` provides `Prepare...` methods and a `Configure` method:

```
class G4VSampler
{
  public:
   G4VSampler();
   virtual ~G4VSampler();
   virtual void PrepareImportanceSampling(G4VIStore *istore,
                                          const G4VImportanceAlgorithm
                                          *ialg = 0) = 0;
   virtual void PrepareWeightRoulett(G4double wsurvive = 0.5,
                                     G4double wlimit = 0.25,
                                     G4double isource = 1) = 0;
   virtual void PrepareWeightWindow(G4VWeightWindowStore *wwstore,
                                    G4VWeightWindowAlgorithm *wwAlg = 0,
                                    G4PlaceOfAction placeOfAction =
                                    onBoundary) = 0;
   virtual void Configure() = 0;
   virtual void ClearSampling() = 0;
   virtual G4bool IsConfigured() const = 0;
};
```

The methods for setting up the desired combination need specific information:

- Importance sampling: message `PrepareImportanceSampling` with a *G4VIStore* and optionally a `G4VImportanceAlgorithm`
- Weight window: message `PrepareWeightWindow` with the arguments:

  - **wwstore*: a `G4VWeightWindowStore` for retrieving the lower weight bounds for the energy-space cells
  - **wwAlg*: a `G4VWeightWindowAlgorithm` if a customized algorithm should be used
  - *placeOfAction*: a `G4PlaceOfAction` specifying where to perform the biasing
- Weight roulette: message `PrepareWeightRoulett` with the optional parameters:
  - *wsurvive*: survival weight
  - *wlimit*: minimal allowed value of weight * source importance / cell importance
  - *isource*: importance of the source cell

Each object of a sampler class is responsible for one particle type. The particle type is given to the constructor of the sampler classes via the particle type name, e.g. "neutron". Depending on the specific purpose, the `Configure()` of a sampler will set up specialized processes (derived from `G4VProcess`) for transportation in the parallel geometry, importance sampling and weight roulette for the given particle type. When `Configure()` is invoked the sampler places the processes in the correct order independent of the order in which user invoked the `Prepare...` methods.

Note:

- The `Prepare...()` functions may each only be invoked once.
- To configure the sampling the function `Configure()` must be called *after* the `G4RunManager` has been initialized and the PhysicsList has been instantiated.

The interface and framework are demonstrated in the `examples/extended/biasing` directory, with the main changes being to the G4GeometrySampler class and the fact that in the parallel case the WorldVolume is a copy of the Mass World. The parallel geometry now has to inherit from `G4VUserParallelWorld` which also has the `GetWorld()` method in order to retrieve a copy of the mass geometry WorldVolume.

```
class B02ImportanceDetectorConstruction : public G4VUserParallelWorld
ghostWorld = GetWorld();
```

The constructor for `G4GeometrySampler` takes a pointer to the physical world volume and the particle type name (e.g. "neutron") as arguments. In a single mass geometry the sampler is created as follows:

```
G4GeometrySampler mgs(detector->GetWorldVolume(),"neutron");
mgs.SetParallel(false);
```

Whilst the following lines of code are required in order to set up the sampler for the parallel geometry case:

```
G4VPhysicalVolume* ghostWorld = pdet->GetWorldVolume();

G4GeometrySampler pgs(ghostWorld,"neutron");

pgs.SetParallel(true);
```

Also note that the preparation and configuration of the samplers has to be carried out *after* the instantiation of the UserPhysicsList. With the modular reference PhysicsList the following set-up is required (first is for biasing, the second for scoring):

```
physicsList->RegisterPhysics(new G4ImportanceBiasing(&pgs,parallelName));
physicsList->RegisterPhysics(new G4ParallelWorldPhysics(parallelName));
```

If the a UserPhysicsList is being implemented, then the following should be used to give the pointer to the GeometrySampler to the PhysicsList:

```
physlist->AddBiasing(&pgs,parallelName);
```

Then to instantiate the biasing physics process the following should be included in the UserPhysicsList and called from `ConstructProcess()`:

```
AddBiasingProcess(){
  fGeomSampler->SetParallel(true); // parallelworld
  G4IStore* iStore = G4IStore::GetInstance(fBiasWorldName);
  fGeomSampler->SetWorld(iStore->GetParallelWorldVolumePointer());
  //  fGeomSampler->PrepareImportanceSampling(G4IStore::
  //                                GetInstance(fBiasWorldName), 0);
  static G4bool first = true;
  if(first) {
    fGeomSampler->PrepareImportanceSampling(iStore, 0);

    fGeomSampler->Configure();
    G4cout << " GeomSampler Configured!!! " << G4endl;
    first = false;
  }

#ifdef G4MULTITHREADED
    fGeomSampler->AddProcess();
#else
  G4cout << " Running in singlethreaded mode!!! " << G4endl;
#endif
```

```
pgs.PrepareImportanceSampling(G4IStore::GetInstance(pdet->GetName()), 0);
pgs.Configure();
```

Due to the fact that biasing is a process and has to be inserted after all the other processes have been created.

Importance Sampling

Importance sampling acts on particles crossing boundaries between "importance cells". The action taken depends on the importance values assigned to the cells. In general a particle history is either split or Russian roulette is played if the importance increases or decreases, respectively. A weight assigned to the history is changed according to the action taken.

The tools provided for importance sampling require the user to have a good understanding of the physics in the problem. This is because the user has to decide which particle types require importance sampled, define the cells, and assign importance values to the cells. If this is not done properly the results cannot be expected to describe a real experiment.

The assignment of importance values to a cell is done using an importance store described below.

An "importance store" with the interface `G4VIStore` is used to store importance values related to cells. In order to do importance sampling the user has to create an object (e.g. of class `G4IStore`) of type `G4VIStore`. The samplers may be given a `G4VIStore`. The user fills the store with cells and their importance values. The store is now a singleton class so should be created using a GetInstance method:

```
G4IStore *aIstore = G4IStore::GetInstance();
```

Or if a parallel world is used:

```
G4IStore *aIstore = G4IStore::GetInstance(pdet->GetName());
```

An importance store has to be constructed with a reference to the world volume of the geometry used for importance sampling. This may be the world volume of the mass or of a parallel geometry. Importance stores derive from the interface `G4VIStore`:

```
class  G4VIStore
{
  public:
    G4VIStore();
    virtual  ~G4VIStore();
```

```
    virtual G4double GetImportance(const G4GeometryCell &gCell) const = 0;
    virtual G4bool IsKnown(const G4GeometryCell &gCell) const = 0;
    virtual const G4VPhysicalVolume &GetWorldVolume() const = 0;
};
```

A concrete implementation of an importance store is provided by the class `G4VStore`. The *public* part of the class is:

```
class G4IStore : public G4VIStore
{
  public:
    explicit G4IStore(const G4VPhysicalVolume &worldvolume);
    virtual ~G4IStore();
    virtual G4double GetImportance(const G4GeometryCell &gCell) const;
    virtual G4bool IsKnown(const G4GeometryCell &gCell) const;
    virtual const G4VPhysicalVolume &GetWorldVolume() const;
    void AddImportanceGeometryCell(G4double importance,
                              const G4GeometryCell &gCell);
    void AddImportanceGeometryCell(G4double importance,
                              const G4VPhysicalVolume &,
                                    G4int aRepNum = 0);
    void ChangeImportance(G4double importance,
                          const G4GeometryCell &gCell);
    void ChangeImportance(G4double importance,
                          const G4VPhysicalVolume &,
                                G4int aRepNum = 0);
    G4double GetImportance(const G4VPhysicalVolume &,
                                G4int aRepNum = 0) const ;
  private: .....
};
```

The member function `AddImportanceGeometryCell()` enters a cell and an importance value into the importance store. The importance values may be returned either according to a physical volume and a replica number or according to a `G4GeometryCell`. The user must be aware of the interpretation of assigning importance values to a cell. If scoring is also implemented then this is attached to logical volumes, in which case the physical volume and replica number method should be used for assigning importance values. See `examples/extended/biasing B01` and `B02` for examples of this.

Note: An importance value must be assigned to every cell.

The different cases:

- *Cell is not in store*
  Not filling a certain cell in the store will cause an exception.
- *Importance value = zero*
  Tracks of the chosen particle type will be killed.
- *importance values > 0*
  Normal allowed values
- *Importance value smaller zero*
  Not allowed!

The Importance Sampling Algorithm

Importance sampling supports using a customized importance sampling algorithm. To this end, the sampler interface *Changing the Sampling* may be given a pointer to the interface `G4VImportanceAlgorithm`:

```
class G4VImportanceAlgorithm
{
  public:
    G4VImportanceAlgorithm();
```

```
    virtual ~G4VImportanceAlgorithm();
    virtual G4Nsplit_Weight Calculate(G4double ipre,
                                      G4double ipost,
                                      G4double init_w) const = 0;
};
```

The method `Calculate()` takes the arguments:

- *ipre*, *ipost* : importance of the previous cell and the importance of the current cell, respectively.
- *init_w*: the particle's weight

It returns the struct:

```
class G4Nsplit_Weight
{
  public:

  G4int fN;
  G4double fW;
};
```

- *fN*: the calculated number of particles to exit the importance sampling
- *fW*: the weight of the particles

The user may have a customized algorithm used by providing a class inheriting from `G4VImportanceAlgorithm`.

If no customized algorithm is given to the sampler the default importance sampling algorithm is used. This algorithm is implemented in `G4ImportanceAlgorithm`.

The Weight Window Technique

The weight window technique is a weight-based alternative to importance sampling:

- applies splitting and Russian roulette depending on space (cells) and energy
- user defines weight windows in contrast to defining importance values as in importance sampling

In contrast to importance sampling this technique is not weight blind. Instead the technique is applied according to the particle weight with respect to the current energy-space cell.

Therefore the technique is convenient to apply in combination with other variance reduction techniques such as cross-section biasing and implicit capture.

A weight window may be specified for every cell and for several energy regions: *space-energy cell*.

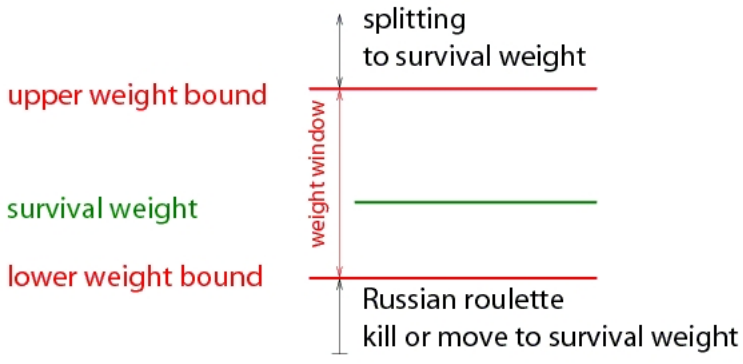


Fig. 3.2: Weight window concept

Weight window concept

The user specifies a *lower weight bound W_L* for every space-energy cell.

- The upper weight bound W_U and the survival weight W_S are calculated as:
  W_U = C_U *W_L* and
  W_S = C_S *W_L*.
- The user specifies C_S and C_U once for the whole problem.
- The user may give different sets of energy bounds for every cell or one set for all geometrical cells
- Special case: if C_S = C_U = 1 for all energies then weight window is equivalent to importance sampling
- The user can choose to apply the technique: at boundaries, on collisions or on boundaries and collisions

The energy-space cells are realized by `G4GeometryCell` as in importance sampling. The cells are stored in a weight window store defined by `G4VWeightWindowStore`:

```
class  G4VWeightWindowStore {
 public:
   G4VWeightWindowStore();
   virtual  ~G4VWeightWindowStore();
   virtual G4double GetLowerWeitgh(const G4GeometryCell &gCell,
                                   G4double partEnergy) const = 0;
   virtual G4bool IsKnown(const G4GeometryCell &gCell) const = 0;
   virtual const G4VPhysicalVolume &GetWorldVolume() const = 0;
};
```

A concrete implementation is provided:

```
class G4WeightWindowStore: public G4VWeightWindowStore {
 public:
   explicit G4WeightWindowStore(const G4VPhysicalVolume &worldvolume);
   virtual ~G4WeightWindowStore();
   virtual G4double GetLowerWeitgh(const G4GeometryCell &gCell,
                                   G4double partEnergy) const;
   virtual G4bool IsKnown(const G4GeometryCell &gCell) const;
   virtual const G4VPhysicalVolume &GetWorldVolume() const;
   void AddLowerWeights(const G4GeometryCell &gCell,
                        const std::vector<G4double> &lowerWeights);
   void AddUpperEboundLowerWeightPairs(const G4GeometryCell &gCell,
                                       const G4UpperEnergyToLowerWeightMap&
                                       enWeMap);
   void SetGeneralUpperEnergyBounds(const
       std::set<G4double, std::less<G4double> > & enBounds);

 private::
 ...
};
```

The user may choose equal energy bounds for all cells. In this case a set of upper energy bounds must be given to the store using the method `SetGeneralUpperEnergyBounds`. If a general set of energy bounds have been set `AddLowerWeights` can be used to add the cells.

Alternatively, the user may chose different energy regions for different cells. In this case the user must provide a mapping of upper energy bounds to lower weight bounds for every cell using the method `AddUpperEboundLowerWeightPairs`.

Weight window algorithms implementing the interface class `G4VWeightWindowAlgorithm` can be used to define a customized algorithm:

```
class G4VWeightWindowAlgorithm {
 public:
   G4VWeightWindowAlgorithm();
   virtual ~G4VWeightWindowAlgorithm();
   virtual G4Nsplit_Weight Calculate(G4double init_w,
                                     G4double lowerWeightBound) const = 0;
};
```

A concrete implementation is provided and used as a default:

```
class G4WeightWindowAlgorithm : public G4VWeightWindowAlgorithm {
 public:
   G4WeightWindowAlgorithm(G4double upperLimitFaktor = 5,
                           G4double survivalFaktor = 3,
                           G4int maxNumberOfSplits = 5);
   virtual ~G4WeightWindowAlgorithm();
   virtual G4Nsplit_Weight Calculate(G4double init_w,
                                     G4double lowerWeightBound) const;
 private:
  ...
};
```

The constructor takes three parameters which are used to: calculate the upper weight bound (upperLimitFaktor), calculate the survival weight (survivalFaktor), and introduce a maximal number (maxNumberOfSplits) of copies to be created in one go.

In addition, the inverse of the maxNumberOfSplits is used to specify the minimum survival probability in case of Russian roulette.

The Weight Roulette Technique

Weight roulette (also called weight cutoff) is usually applied if importance sampling and implicit capture are used together. Implicit capture is not described here but it is useful to note that this procedure reduces a particle weight in every collision instead of killing the particle with some probability.

Together with importance sampling the weight of a particle may become so low that it does not change any result significantly. Hence tracking a very low weight particle is a waste of computing time. Weight roulette is applied in order to solve this problem.

The weight roulette concept

Weight roulette takes into account the importance "Ic" of the current cell and the importance "Is" of the cell in which the source is located, by using the ratio "R=Is/Ic".

Weight roulette uses a relative minimal weight limit and a relative survival weight. When a particle falls below the weight limit Russian roulette is applied. If the particle survives, tracking will be continued and the particle weight will be set to the survival weight.

The weight roulette uses the following parameters with their default values:

- *wsurvival*: 0.5
- *wlimit*: 0.25
- *isource*: 1

The following algorithm is applied:

If a particle weight "w" is lower than R*wlimit:

- the weight of the particle will be changed to "ws = wsurvival*R"
- the probability for the particle to survive is "p = w/ws"

3.7.2 Physics Based Biasing

Geant4 supports physics based biasing through a number of general use, built in biasing techniques. A utility class, G4WrapperProcess, is also available to support user defined biasing.

Built in Biasing Options

Primary Particle Biasing

Primary particle biasing can be used to increase the number of primary particles generated in a particular phase space region of interest. The weight of the primary particle is modified as appropriate. A general implementation is provided through the `G4GeneralParticleSource` class. It is possible to bias position, angular and energy distributions.

`G4GeneralParticleSource` is a concrete implementation of `G4VPrimaryGenerator`. To use, instantiate `G4GeneralParticleSource` in the `G4VUserPrimaryGeneratorAction` class, as demonstrated below.

```
MyPrimaryGeneratorAction::MyPrimaryGeneratorAction() {
   generator = new G4GeneralParticleSource;
}

void
MyPrimaryGeneratorAction::GeneratePrimaries(G4Event*anEvent){
   generator->GeneratePrimaryVertex(anEvent);
}
```

The biasing can be configured through interactive commands, as desribed in *Geant4 General Particle Source*. Examples are also distributed with the Geant4 distribution in **examples/extended/eventgenerator/exgps**.

Hadronic Leading Particle Biasing

One hadronic leading particle biasing technique is implemented in the G4HadLeadBias utility. This method keeps only the most important part of the event, as well as representative tracks of each given particle type. So the track with the highest energy as well as one of each of Baryon, pi0, mesons and leptons. As usual, appropriate weights are assigned to the particles. Setting the **SwitchLeadBiasOn** environmental variable will activate this utility.

Hadronic Cross Section Biasing

Cross section biasing artificially enhances/reduces the cross section of a process. This may be useful for studying thin layer interactions or thick layer shielding. The built in hadronic cross section biasing applies to photon inelastic, electron nuclear and positron nuclear processes.

The biasing is controlled through the **BiasCrossSectionByFactor** method in G4HadronicProcess, as demonstrated below.

```
void MyPhysicsList::ConstructProcess()
{
   ...
   G4ElectroNuclearReaction * theElectroReaction =
      new G4ElectroNuclearReaction;

   G4ElectronNuclearProcess theElectronNuclearProcess;
   theElectronNuclearProcess.RegisterMe(theElectroReaction);
   theElectronNuclearProcess.BiasCrossSectionByFactor(100);

   pManager->AddDiscreteProcess(&theElectronNuclearProcess);
   ...
}
```

Radioactive Decay Biasing

The G4RadioactiveDecay (GRDM) class simulates the decay of radioactive nuclei and implements the following biasing options:

- Increase the sampling rate of radionuclides within observation times through a user defined probability distribution function
- Nuclear splitting, where the parent nuclide is split into a user defined number of nuclides
- Branching ratio biasing where branching ratios are sampled with equal probability

G4RadioactiveDecay is a process which must be registered with a process manager, as demonstrated below.

```
void MyPhysicsList::ConstructProcess()
{
   ...
   G4RadioactiveDecay* theRadioactiveDecay =
      new  G4RadioactiveDecay();

   G4ProcessManager* pmanager = ...
   pmanager ->AddProcess(theRadioactiveDecay);
   ...
}
```

Biasing can be controlled either in compiled code or through interactive commands. Radioactive decay biasing examples are also distributed with the Geant4 distribution in **examples/extended/radioactivedecay/exrdm**.

To select biasing as part of the process registration, use

```
theRadioactiveDecay->SetAnalogueMonteCarlo(false);
```

or the equivalent macro command:

```
/grdm/analogeMC [true|false]
```

In both cases, *true* specifies that the unbiased (analogue) simulation will be done, and *false* selects biasing.

Limited Radionuclides

Radioactive decay may be restricted to only specific nuclides, in order (for example) to avoid tracking extremely long-lived daughters in decay chains which are not of experimental interest. To limit the range of nuclides decayed as part of the process registration (above), use

```
G4NucleusLimits limits(aMin, aMax, zMin, zMax);
theRadioactiveDecay->SetNucleusLimits(limits);
```

or via the macro command

```
/grdm/nucleusLimits [aMin] [aMax] [zMin] [zMax]
```

Geometric Biasing

Radioactive decays may be generated throughout the user's detector model, in one or more specified volumes, or nowhere. The detector geometry must be defined before applying these geometric biases.

Volumes may be selected or deselected programmatically using

```
theRadioactiveDecay->SelectAllVolumes();
theRadioactiveDecay->DeselectAllVolumes();

G4LogicalVolume* aLogicalVolume;        // Acquired by the user
theRadioactiveDecay->SelectVolume(aLogicalVolume);
theRadioactiveDecay->DeselectVolume(aLogicalVolume);
```

or with the equivalent macro commands

```
/grdm/allVolumes
/grdm/noVolumes
/grdm/selectVolume [logicalVolume]
/grdm/deselectVolume [logicalVolume]
```

In macro commands, the volumes are specified by name, and found by searching the `G4LogicalVolumeStore`.

Decay Time Biasing

The decay time function (normally an exponential in the natural lifetime) of the primary particle may be replaced with a *time profile* F(t), as discussed in Section 40.6 of the *Physics Reference Manual*. The profile function is represented as a two-column ASCII text file with up to 100 time points (first column) with fractions (second column).

```
theRadioactiveDecay->SetSourceTimeProfile(fileName);
theRadioactiveDecay->SetDecayBias(fileName);
```

```
/grdm/sourceTimeProfile [fileName]
/grdm/decayBiasProfile [fileName]
```

Branching Fraction Biasing

Radionuclides with rare decay channels may be biased by forcing all channels to be selected uniformly (`BRBias` = *true* below), rather than according to their natural branching fractions (*false*).

```
theRadioactiveDecay->SetBRBias(true);
```

```
/grdm/BRbias [true|false]
```

Nuclear Splitting

The statistical efficiency of generated events may be increased by generating multiple "copies" of nuclei in an event, each of which is decayed independently, with an assigned weight of 1/Nsplit. Scoring the results of tracking the decay daughters, using their corresponding weights, can improve the statistical reach of a simulation while preserving the shape of the resulting distributions.

```
theRadioactiveDecay->SetSplitNuclei(Nsplit);
```

```
/grdm/splitNucleus [Nsplit]
```

G4WrapperProcess

G4WrapperProcess can be used to implement user defined event biasing. G4WrapperProcess, which is a process itself, wraps an existing process. By default, all function calls are forwared to the wrapped process. It is a non-invasive way to modify the behaviour of an existing process.

To use this utility, first create a derived class inheriting from G4WrapperProcess. Override the methods whose behaviour you would like to modify, for example, PostStepDoIt, and register the derived class in place of the process to be wrapped. Finally, register the wrapped process with G4WrapperProcess. The code snippets below demonstrate its use.

```
class MyWrapperProcess  : public G4WrapperProcess {
...
 G4VParticleChange* PostStepDoIt(const G4Track& track,
                                 const G4Step& step) {
    // Do something interesting
 }
};


void MyPhysicsList::ConstructProcess()
{
...
   G4eBremsstrahlung* bremProcess =
      new G4eBremsstrahlung();

   MyWrapperProcess* wrapper = new MyWrapperProcess();
   wrapper->RegisterProcess(bremProcess);

   processManager->AddProcess(wrapper, -1, -1, 3);
}
```

3.7.3 Adjoint/Reverse Monte Carlo

Another powerful biasing technique available in Geant4 is the Reverse Monte Carlo (RMC) method, also known as the Adjoint Monte Carlo method. In this method particles are generated on the external boundary of the sensitive part of the geometry and then are tracked backward in the geometry till they reach the external source surface, or exceed an energy threshold. By this way the computing time is focused only on particle tracks that are contributing to the tallies. The RMC method is much rapid than the Forward MC method when the sensitive part of the geometry is small compared to the rest of the geometry and to the external source, that has to be extensive and not beam like. At the moment the RMC method is implemented in Geant4 only for some electromagnetic processes (see *Reverse processes*). An example illustrating the use of the Reverse MC method in Geant4 is distributed within the Geant4 toolkit in **examples/extended/biasing/ReverseMC01**.

Treatment of the Reverse MC method in Geant4

Different G4Adjoint classes have been implemented into the Geant4 toolkit in order to run an adjoint/reverse simulation in a Geant4 application. This implementation is illustrated in *Schematic view*. An adjoint run is divided in a serie of alternative adjoint and forward tracking of adjoint and normal particles. One Geant4 event treats one of this tracking phase.

Adjoint tracking phase

Adjoint particles (adjoint_e-, adjoint_gamma,...) are generated one by one on the so called adjoint source with random position, energy (1/E distribution) and direction. The adjoint source is the external surface of a user defined volume or of a user defined sphere. The adjoint source should contain one or several sensitive volumes and should be small compared to the entire geometry. The user can set the minimum and maximum energy of the adjoint source. After its generation the adjoint primary particle is tracked backward in the geometry till a user defined external surface (spherical or boundary of a volume) or is killed before if it reaches a user defined upper energy limit that represents the maximum energy of the external source. During the reverse tracking, reverse processes take place where the adjoint particle being tracked can be either scattered or transformed in another type of adjoint particle. During the reverse

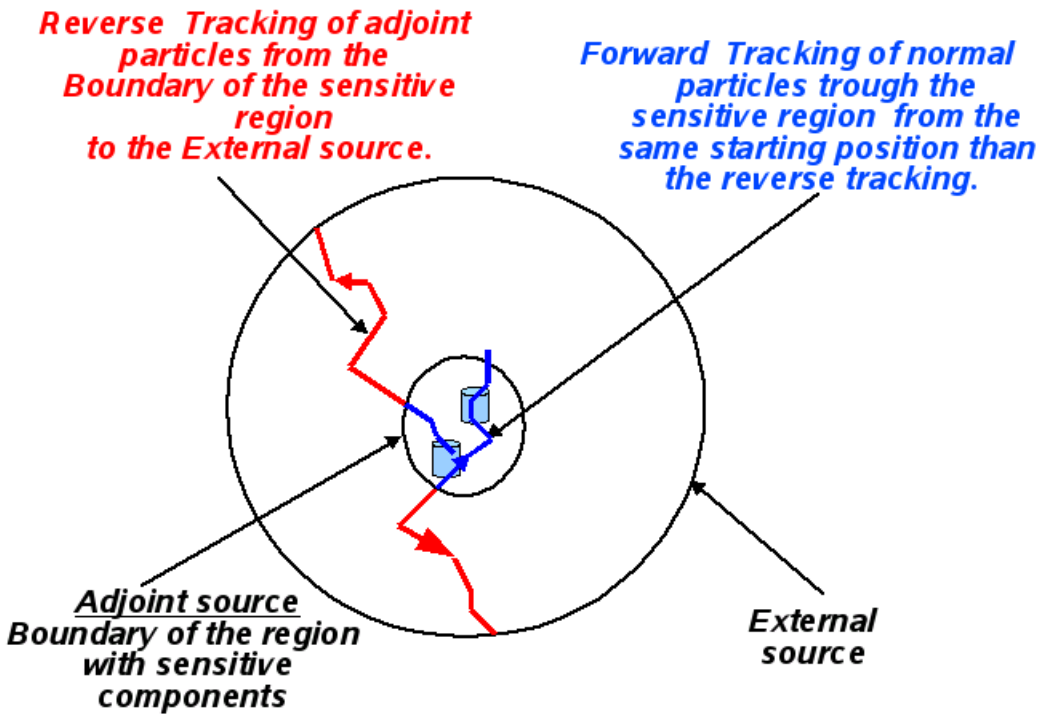


Fig. 3.3: Schematic view of an adjoint/reverse simulation in Geant4

tracking the G4AdjointSimulationManager replaces the user defined primary, run, stepping, ... actions, by its own actions. A reverse tracking phase corresponds to one Geant4 event.

Forward tracking phase

When an adjoint particle reaches the external surface its weight, type, position, and direction are registered and a normal primary particle, with a type equivalent to the last generated primary adjoint, is generated with the same energy, position but opposite direction and is tracked in the forward direction in the sensitive region as in a forward MC simulation. During this forward tracking phase the event, stacking, stepping, tracking actions defined by the user for his forward simulation are used. By this clear separation between adjoint and forward tracking phases, the code of the user developed for a forward simulation should be only slightly modified to adapt it for an adjoint simulation (see *How to update a G4 application to use the reverse Monte Carlo mode*). Indeed the computation of the signals is done by the same actions or classes that the one used in the forward simulation mode. A forward tracking phase corresponds to one G4 event.

Reverse processes

During the reverse tracking, reverse processes act on the adjoint particles. The reverse processes that are at the moment available in Geant4 are the:

- Reverse discrete ionization for e-, proton and ions
- Continuous gain of energy by ionization and bremsstrahlung for e- and by ionization for protons and ions
- Reverse discrete e- bremsstrahlung
- Reverse photo-electric effect
- Reverse Compton scattering

- Approximated multiple scattering (see comment in *Reverse multiple scattering*)

It is important to note that the electromagnetic reverse processes are cut dependent as their equivalent forward processes. The implementation of the reverse processes is based on the forward processes implemented in the G4 standard electromagnetic package.

Nb of adjoint particle types and nb of G4 events of an adjoint simulation

The list of type of adjoint and forward particles that are generated on the adjoint source and considered in the simulation is a function of the adjoint processes declared in the physics list. For example if only the e- and gamma electromagnetic processes are considered, only adjoint e- and adjoint gamma will be considered as primaries. In this case an adjoint event will be divided in four G4 event consisting in the reverse tracking of an adjoint e-, the forward tracking of its equivalent forward e-, the reverse tracking of an adjoint gamma, and the forward tracking of its equivalent forward gamma. In this case a run of 100 adjoint events will consist into 400 Geant4 events. If the proton ionization is also considered adjoint and forward protons are also generated as primaries and 600 Geant4 events are processed for 100 adjoint events.

How to update a G4 application to use the reverse Monte Carlo mode

Some modifications are needed to an existing Geant4 application in order to adapt it for the use of the reverse simulation mode (see also the G4 example **examples/extended/biasing/ReverseMC01**). It consists into the:

- Creation of the adjoint simulation manager in the main code
- Optional declaration of user actions that will be used during the adjoint tracking phase
- Use of a special physics lists that combine the adjoint and forward processes
- Modification of the user analysis part of the code

Creation of G4AdjointSimManager in the main

The class G4AdjointSimManager represents the manager of an adjoint simulation. This static class should be created somewhere in the main code. The way to do that is illustrated below

```
int main(int argc,char** argv) {
   ...
   G4AdjointSimManager* theAdjointSimManager = G4AdjointSimManager::GetInstance();
   ...
}
```

By doing this the G4 application can be run in the reverse MC mode as well as in the forward MC mode. It is important to note that G4AdjointSimManager is not a new G4RunManager and that the creation of G4RunManager in the main and the declaration of the geometry, physics list, and user actions to G4RunManager is still needed. The definition of the adjoint and external sources and the start of an adjoint simulation can be controlled by G4UI commands in the directory **/adjoint**.

Optional declaration of adjoint user actions

During an adjoint simulation the user stepping, tracking, stacking and event actions declared to G4RunManager are used only during the G4 events dedicated to the forward tracking of normal particles in the sensitive region, while during the events where adjoint particles are tracked backward the following happen concerning these actions:

- The user stepping action is replaced by G4AdjointSteppingAction that is reponsible to stop an adjoint track when it reaches the external source, exceed the maximum energy of the external source, or cross the adjoint source surface. If needed the user can declare its own stepping action that will be called by G4AdjointSteppingAction

after the check of stopping track conditions. This stepping action can be different that the stepping action used for the forward simulation. It is declared to G4AdjointSimManager by the following lines of code:

```
G4AdjointSimManager* theAdjointSimManager = G4AdjointSimManager::GetInstance();
theAdjointSimManager->SetAdjointSteppingAction(aUserDefinedSteppingAction);
```

- No stacking, tracking and event actions are considered by default. If needed the user can declare to G4AdjointSimManager stacking, tracking and event actions that will be used only during the adjoint tracking phase. The following lines of code show how to declare these adjoint actions to G4AdjointSimManager:

```
G4AdjointSimManager* theAdjointSimManager = G4AdjointSimManager::GetInstance();
theAdjointSimManager->SetAdjointEventAction(aUserDefinedEventAction);
theAdjointSimManager->SetAdjointStackingAction(aUserDefinedStackingAction);
theAdjointSimManager->SetAdjointTrackingAction(aUserDefinedTrackingAction);
```

By default no user run action is considered in an adjoint simulation but if needed such action can be declared to G4AdjointSimManager as such:

```
G4AdjointSimManager* theAdjointSimManager = G4AdjointSimManager::GetInstance();
theAdjointSimManager->SetAdjointRunAction(aUserDefinedRunAction);
```

Physics list for reverse and forward electromagnetic processes

To run an adjoint simulation a specific physics list should be used where existing G4 adjoint electromagnetic processes and their forward equivalent have to be declared. An example of such physics list is provided by the class G4AdjointPhysicsLits in the G4 example **extended/biasing/ReverseMC01**.

Modification in the analysis part of the code

The user code should be modified to normalize the signals computed during the forward tracking phase to the weight of the last adjoint particle that reaches the external surface. This weight represents the statistical weight that the last full adjoint tracks (from the adjoint source to the external source) would have in a forward simulation. If multiplied by a signal and registered in function of energy and/or direction the simulation results will give an answer matrix of this signal. To normalize it to a given spectrum it has to be furthermore multiplied by a directional differential flux corresponding to this spectrum The weight, direction, position , kinetic energy and type of the last adjoint particle that reaches the external source, and that would represents the primary of a forward simulation, can be gotten from G4AdjointSimManager by using for example the following line of codes

```
G4AdjointSimManager* theAdjointSimManager = G4AdjointSimManager::GetInstance();
G4String particle_name = theAdjointSimManager->GetFwdParticleNameAtEndOfLastAdjointTrack();
G4int PDGEncoding= theAdjointSimManager->GetFwdParticlePDGEncodingAtEndOfLastAdjointTrack();
G4double weight = theAdjointSimManager->GetWeightAtEndOfLastAdjointTrack();
G4double Ekin  = theAdjointSimManager->GetEkinAtEndOfLastAdjointTrack();
G4double Ekin_per_nuc=theAdjointSimManager->GetEkinNucAtEndOfLastAdjointTrack(); // for ions
G4ThreeVector dir =  theAdjointSimManager->GetDirectionAtEndOfLastAdjointTrack();
G4ThreeVector pos = theAdjointSimManager->GetPositionAtEndOfLastAdjointTrack();
```

In order to have a code working for both forward and adjoint simulation mode, the extra code needed in user actions or analysis manager for the adjoint simulation mode can be separated to the code needed only for the normal forward simulation by using the following public method of G4AdjointSimManager:

```
G4bool GetAdjointSimMode();
```

that returns true if an adjoint simulation is running and false if not.

The following code example shows how to normalize a detector signal and compute an answer matrix in the case of an adjoint simulation.

Listing 3.5: Normalization in the case of an adjoint simulation. The detector signal S computed during the forward tracking phase is normalized to a primary source of e- with a differential directional flux given by the function F. An answer matrix of the signal is also computed.

```
G4double S = ...; // signal in the sensitive volume computed during a forward tracking phase

//Normalization of the signal for an adjoint simulation
G4AdjointSimManager* theAdjSimManager = G4AdjointSimManager::GetInstance();
if (theAdjSimManager->GetAdjointSimMode()) {
  G4double normalized_S=0.;        //normalized to a given e- primary spectrum
  G4double S_for_answer_matrix=0.; //for e- answer matrix

  if (theAdjSimManager->GetFwdParticleNameAtEndOfLastAdjointTrack() == "e-") {
    G4double ekin_prim = theAdjSimManager->GetEkinAtEndOfLastAdjointTrack();
    G4ThreeVector dir_prim =  theAdjointSimManager->GetDirectionAtEndOfLastAdjointTrack();
    G4double weight_prim = theAdjSimManager->GetWeightAtEndOfLastAdjointTrack();
    S_for_answer_matrix = S*weight_prim;
    normalized_S = S_for_answer_matrix*F(ekin_prim,dir);
    // F(ekin_prim,dir_prim) gives the differential directional flux of primary e-
  }
  //follows the code where normalized_S and S_for_answer_matrix are registered or whatever
   ....
}

//analysis/normalization code for forward simulation
else {
  ....
}
....
```

Control of an adjoint simulation

The G4UI commands in the directory /adjoint. allow the user to :

- Define the adjoint source where adjoint primaries are generated
- Define the external source till which adjoint particles are tracked
- Start an adjoint simulation

Known issues in the Reverse MC mode

Occasional wrong high weight in the adjoint simulation

In rare cases an adjoint track may get a wrong high weight when reaching the external source. While this happens not often it may corrupt the simulation results significantly. This happens in some tracks where both reverse photo-electric and bremsstrahlung processes take place at low energy. We still need some investigations to remove this problem at the level of physical adjoint/reverse processes. However this problem can be solved at the level of event actions or analysis in the user code by adding a test on the normalized signal during an adjoint simulation. An example of such test has been implemented in the Geant4 example **extended/biasing/ReverseMC01**. In this implementation an event is rejected when the relative error of the computed normalized energy deposited increases during one event by more than 50% while the computed precision is already below 10%.

Reverse bremsstrahlung

A difference between the differential cross sections used in the adjoint and forward bremsstrahlung models is the source of a higher flux of >100 keV gamma in the reverse simulation compared to the forward simulation mode.

In principle the adjoint processes/models should make use of the direct differential cross section to sample the adjoint secondaries and compute the adjoint cross section. However due to the way the effective differential cross section is considered in the forward model G4eBremsstrahlungModel this was not possible to achieve for the reverse bremsstrahlung. Indeed the differential cross section used in G4AdjointeBremstrahlungModel is obtained by the numerical derivation over the cut energy of the direct cross section provided by G4eBremsstrahlungModel. This would be a correct procedure if the distribution of secondary in G4eBremsstrahlungModel would match this differential cross section. Unfortunately it is not the case as independent parameterization are used in G4eBremsstrahlungModel for both the cross sections and the sampling of secondaries. (It means that in the forward case if one would integrate the effective differential cross section considered in the simulation we would not find back the used cross section). In the future we plan to correct this problem by using an extra weight correction factor after the occurrence of a reverse bremsstrahlung. This weight factor should be the ratio between the differential CS used in the adjoint simulation and the one effectively used in the forward processes. As it is impossible to have a simple and direct access to the forward differential CS in G4eBremsstrahlungModel we are investigating the feasibility to use the differential CS considered in G4Penelope models.

Reverse multiple scattering

For the reverse multiple scattering the same model is used than in the forward case. This approximation makes that the discrepancy between the adjoint and forward simulation cases can get to a level of ~ 10-15% relative differences in the test cases that we have considered. In the future we plan to improve the adjoint multiple scattering models by forcing the computation of multiple scattering effect at the end of an adjoint step.

3.7.4 Generic Biasing

The generic biasing scheme provides facilities for:

- physics-based biasing, to alter the behavior of existing physics processes:
  - biasing of physics process interaction occurence,
  - biasing of physics process final state production;
- non-physics-based biasing, to introduce or remove particles in the simulation but without affecting the existing physics processes, with techniques like, but not limited to
  - splitting,
  - Russian roulette (killing).

Decisions on what techniques to apply are taken on a step by step and inter-step basis, hence providing a lot of flexibility.

The scheme has been introduced in 10.0, with new features and some non-backward compatible changes introduced in 10.1 and 10.2; these are documented in *Changes from 10.0 to 10.1* and *Changes from 10.1 to 10.2*. Parallel geometry capability has been introduced in 10.3.

Overview

The generic biasing scheme relies on two abstract classes, that are meant to model the biasing problems. You have to inherit from them to create your own concrete classes, or use some of the concrete instances provided (see *Existing biasing operations, operator and interaction laws*), if they respond to your case. A dedicated process provides the interface between these biasing classes and the tracking. In case of parallel geometry usage, an other process handles the navigation in these geometries.

The two abstract classes are:

- `G4VBiasingOperation`: which represents a simple, or "atomic" biasing operation, like changing a process interaction occurence probability, or changing its final state production, or making a splitting operation, etc. For the occurence biasing case, the biasing is handled with an other class, **"G4VBiasingInteractionLaw"**, which

holds the properties of the biased interaction law. An object of this class type must be provided by the occurence biasing operation returned.
- `G4VBiasingOperator`: which purpose is to make decisions on the above biasing operations to be applied. It is attached to a G4LogicalVolume and is the pilot of the biasing in this volume. An operator may decide to delegate to other operators. An operator acts only in the `G4LogicalVolume` it is attached to. In volumes with no biasing operator attached, the usual tracking is applied.

The process acting as interface between the biasing classes and the tracking is:

- `G4BiasingProcessInterface`: it is a concrete `G4VProcess` implementation. It interrogates the current biasing operator, if any, for biasing operations to be applied. The `G4BiasingProcessInterface` can either:
  - hold a physics process that it wraps and controls: in this case it asks the operator for physics-based biasing operations (only) to be applied to the wrapped process,
  - not hold a physics process: in this case it asks the operator for non-physics-based biasing operations (only): splitting, killing, etc.
- The `G4BiasingProcessInterface` class provides many information that can be used by the biasing operator. Each `G4BiasingProcessInterface` provides its identity to the biasing operator it calls, so that the operator has this information but also information of the underneath wrapped physics process, if it is the case.
  The `G4BiasingProcessInterface` can be asked for all other `G4BiasingProcessInterface` instances at play on the current track. In particular, this allows the operator to get all cross-sections at the current point (feature available since 10.1). The code is organized in such a way that these cross-sections are all available at the first call to the operator in the current step.
- To make `G4BiasingProcessInterface` instances wrapping physics processes, or to insert instances not holding a physics process, the physics list has to be modified -the generic biasing approach is hence invasive to the physics list-. The way to configure your physics list and related helper tools are described below.

The process handling parallel geometries is:

- `G4ParallelGeometriesLimiterProcess`, it is a concrete `G4VProcess` implementation, which takes care of limiting the step on the boundaries of parallel geometries.
- A single instance of `G4ParallelGeometriesLimiterProcess` handles all parallel geometries to be considered for a particle type. It collects these geometries by means of `myLimiterProcess->AddParallelWorld("myParallelGeometry")` calls.
  Given such a process is attached to a particle type, parallel geometries are hence specified per particle type.
- Attaching an instance of this process to a given particle type, and specifying the parallel geometries to be considered is eased by the helper tools as explained below.

Getting Started

Examples

Six "Generic Biasing (GB)" examples are proposed (they have been introduced in 10.0, 10.1 and 10.3, two examples each time):

- `examples/extended/biasing/GB01`:
  - which shows how biasing of process cross-section can be done.
  - This example uses the physics-based biasing operation `G4BOptnChangeCrossSection` defined in `geant4/source/processes/biasing/generic`. This operation performs the actual process cross-section change. In the example a first `G4VBiasingOperator`, `GB01BOptrChangeCrossSection`, configures and selects this operation. This operator applies to only one particle type.
  - To allow several particle types to be biased, a second `G4VBiasingOperator`, `GB01BOptrMultiParticleChangeCrossSection`, is implemented, and which holds a

GB01BOptrChangeCrossSection operator for each particle type to be biased. This second operator then delegates to the first one the handling of the biasing operations.

- examples/extended/biasing/GB02:
  - which shows how a "force collision" scheme very close to the MNCP one can be activated.
  - This second example has a quite similar approach than the GB01 one, with a G4VBiasingOperator, QGB02BOptrMultiParticleForceCollision, that holds as many operators than particle types to be biased, this operators being of G4BOptrForceCollision type.
  - This G4BOptrForceCollision operator is defined in source/processes/biasing/generic. It combines several biasing operations to build-up the needed logic (see *Setting up the application*). It can be in particular looked at to see how it collects and makes use of physics process cross-sections.
- examples/extended/biasing/GB03:
  - which implements a kind of importance geometry biasing, using the generic biasing classes.
  - The example uses a simple sampling calorimeter. On the boundary of the absorber parts, it does splitting (killing) if the track is moving forward (backward). As the splitting can be too strong in some cases, falling into an over-splitting situation, even with a splitting by a factor 2, a technique is introduced to alleviate the problem : a probability to apply the splitting (killing) is introduced, and with proper tuning of this probability, the over-splitting can be avoided.
- examples/extended/biasing/GB04:
  - which implements a bremsstrahlung splitting. Bremsstrahlung splitting exists in the EM package. In the present example, it is shown how to implement a similar technique, using the generic biasing classes.
  - A biasing operator, GB04BOptrBremSplitting, sends a final state biasing operation, GB04BOptnBremSplitting, for the bremsstrahlung process. Splitting factor, and options to control the biasing are available through command line.
- examples/extended/biasing/GB05:
  - which illustrates a technique that uses physics cross-sections to determine the splitting[killing] rate in a shielding problem, it is applied to neutrons. This technique is supposed to be an invention, to illustrate a technique combining physics-based information with splitting/killing.
  - In the classical treatment of the shielding problem, the shield is divided in slices at the boundaries of which particles are splitted[killed] if moving forward[backward]. In the present technique, we collect the cross-sections of "absorbing/destroying" processes : decay, capture, inelastic. We then use the generic biasing facilities to create an equivalent of a splitting process, that has a "cross-section" which is the sum of the previous ones. This process is competing with other processes, as a regular one. When this process wins the competition, it splits the track, with a splitting factor 2. This splitting is hence occuring at the same rate than the absorption, resulting in an expected maintained (unweighted) flux.
  - GB05BOptrSplitAndKillByCrossSection and GB05BOptnSplitAndKillByCrossSection are respectively the biasing operator and operation. The operator collects the absorbing cross-sections at the beginning of the step, passes them to the operation, requests it to sample the distance to its next interaction, and returns this operation to the calling G4BiasingProcessInterface as the operation to be applied in the step.
  - The operation interaction distance is then proposed by the calling G4BiasingProcessInterface and, if being the shortest of the interaction distances, the operation final state generation (the splitting) is applied by the process.
- examples/extended/biasing/GB06:
  - which demonstrates the use of parallel geometries in generic biasing, on a classical shield problem, using geometry-based importance biasing.
  - The mass geometry consists of a single block of concrete; it is overlayed by a parallel geometry defining the slices used for splitting/killing.
  - The navigation capability in the parallel geometry is activated in the main program, by means of the physics list constructor.

Setting up the application

For making an existing G4VBiasingOperator used by your application, you have to do two things:

1. Attach the operator to the G4LogicalVolume where the biasing should take place: You have to make this attachment in your ConstructSDandField() method (to make your application both sequential and MT-compliant):

   Listing 3.6: Attachment of a G4BiasingOperator to a G4LogicalVolume. We assume such a volume has been created with name "volumeWithBiasing", and we assume that a biasing operator class MyBiasingOperator has been created, inheriting from G4VBiasingOperator:

   ```
   // Fetch the logical volume pointer by name (it is an example, not a mandatory way):
   G4LogicalVolume* biasingVolume = G4LogicalVolumeStore::GetInstance()->GetVolume(
   ↪"volumeWithBiasing");
   // Create the biasing operator:
   MyBiasingOperator* myBiasingOperator = new MyBiasingOperator("ExampleOperator");
   // Attach it to the volume:
   myBiasingOperator->AttachTo(biasingVolume);
   ```

2. Setup the physics list you use to properly include the needed G4BiasingProcessInterface instances. You have several options for this.
   - The easiest way is if you use a pre-packaged physics list (e.g. FTFP_BERT, QGSP...). As such a physics list is of G4VModularPhysicsList type, you can alter it with a G4VPhysicsConstructor. The constructor G4GenericBiasingPhysics is meant for this. It can be used, typically in your main program, as:

     Listing 3.7: Use of the G4GenericBiasingPhysics physics constructor to setup a pre-packaged physics list (of G4VModularPhysicsList type). Here we assume the FTFP_BERT physics list, and we assume that runManager is a pointer on a created G4RunManager or G4RMTunManager object.

     ```
     // Instantiate the physics list:
     FTFP_BERT* physicsList = new FTFP_BERT;
     // Create the physics constructor for biasing:
     G4GenericBiasingPhysics* biasingPhysics = new G4GenericBiasingPhysics();
     // Tell what particle types have to be biased:
     biasingPhysics->Bias("gamma");
     biasingPhysics->Bias("neutron");
     // Register the physics constructor to the physics list:
     physicsList->RegisterPhysics(biasingPhysics);
     // Set this physics list to the run manager:
     runManager->SetUserInitialization(physicsList);
     ```

     Doing so, all physics processes will be wrapped, and, for example, the gamma conversion process, "conv", will appear as "biasWrapper(conv)" when dumping the processes (/particle/process/dump). An additionnal "biasWrapper(0)" process, for non-physics-based biasing is also inserted.
     Other methods to specifically chose some physics processes to be biased or to insert only G4BiasingProcessInterface instances for non-physics-based biasing also exist.
   - The second way is useful if you write your own physics list, and if this one is not a modular physics list, but inherits directly from the lowest level abstract class G4VUserPhysicsList. In this case, the above solution with G4GenericBiasingPhysics does not apply. Instead you can use the G4BiasingHelper utility class (this one is indeed used by G4GenericBiasingPhysics).

     Listing 3.8: Use of the G4BiasingHelper utility class to setup a physics list for biasing in case this physics list is not of G4VModularPhysicsList type but inherits directly from G4VUserPhysicsList.

     ```
     // Get physics list helper:
     G4PhysicsListHelper* ph = G4PhysicsListHelper::GetPhysicsListHelper();
     ...
     ```

```
// Assume "particle" is a pointer on a G4ParticleDefinition object
G4String particleName = particle->GetParticleName();
if (particleName == "gamma")
{
ph->RegisterProcess(new G4PhotoElectricEffect , particle);
ph->RegisterProcess(new G4ComptonScattering ,   particle);
ph->RegisterProcess(new G4GammaConversion ,     particle);
G4ProcessManager* pmanager = particle->GetProcessManager();
G4BiasingHelper::ActivatePhysicsBiasing(pmanager, "phot");
G4BiasingHelper::ActivatePhysicsBiasing(pmanager, "compt");
G4BiasingHelper::ActivatePhysicsBiasing(pmanager, "conv");
G4BiasingHelper::ActivateNonPhysicsBiasing(pmanager);
}
```

- A last way to setup the physiscs list is by direct insertion of the `G4BiasingProcessInterface` instances, but this requires solid expertise in physics list creation.

In case you also use parallel geometries, you have to make the generic biasing sensitive to these. Assuming you have created three parallel geometries with names `"parallelWorld1"`, `"parallelWorld2"` and `"parallelWorld3"` that you want to be active for neutrons, the additionnal calls you have to make compared to example *EvtBias.GenericBiasing.Overview.UsePhysConstructor* above are simply:

Listing 3.9: Calls to activate parallel geometry navigation

```
// -- activate parallel geometries for neutrons:
biasingPhysics->AddParallelGeometry("neutron","parallelWorld1");
biasingPhysics->AddParallelGeometry("neutron","parallelWorld2");
biasingPhysics->AddParallelGeometry("neutron","parallelWorld3");
```

It is also possible, even though less convenient, to use the `G4BiasingHelper` utility class making calls to the static method `limiter = G4BiasingHelper::AddLimiterProcess(pmanager, "limiterProcessName")` in addition to the ones of example *EvtBias.GenericBiasing.Overview.UseBiasingHelper* above. This call returns a pointer `limiter` on the constructed `G4ParallelGeometriesLimiterProcess` process, setting its name as `"limiterProcessName"`, this pointer has then to be used to specify the parallel geometries to the process : `limiter->AddParallelWorld("parallelWorld1")...`

Existing biasing operations, operator and interaction laws

This is set of biasing operations and one operator available in 10.1, as well as a set of biasing interaction laws. These are defined in `source/processes/biasing/generic`. Please note that several examples (*Examples*) also implement dedicated operators and operations.

These classes have been tested for now with neutral particles.

- `G4VBiasingOperation` classes:
  - `G4BOptnCloning`: a non-physics-based biasing operation that clones the current track. Each of the two copies is given freely a weight.
  - `G4BOptnChangeCrossSection`: a physics-based biasing operation to change one process cross-section
  - `G4BOptnForceFreeFlight`: a physics-based biasing operation to force a flight with no interaction through the current volume. This operation is better said a "silent flight": the flight is conducted under a zero weight, and the track weight is restored at the end of the free flight, taking into account the cumulated weight change for the non-interaction flight. This special feature is because this class in used in the MCNP-like force collision scheme `G4BOptrForceCollision`.
  - `G4BOptnForceCommonTruncatedExp`: a physics-based biasing operation to force a collision inside the current volume. It is "common" as several processes may be forced together, driving the related interaction law by the sum of these processes cross-section. The relative natural occurence of processes is conserved. This operation makes use of a "truncated exponential" law, which is the exponential law limited to a segment [0,L], where L is the distance to exit the current volume.

- G4VBiasingOperator class:
  - G4BOptrForceCollision: a biasing operator that implements a force collision scheme quite close to the one provided by MCNP. It handles the scheme though the following sequence:
    1. The operator starts by using a G4BOptnCloning cloning operation, making a copy of the primary entering the volume. The primary is given a zero weight.
    2. The primary is then transported through to the volume, without interactions. This is done with the operator requesting forced free flight G4BOptnForceFreeFlight operations to all physics processes. The weight is zero to prevent the primary to contribute to scores. This flight purpose is to accumulate the probability to fly through the volume without interaction. When the primary reaches the volume boundary, the first free flight operation restores the primary weight to its initial weight and all operations multiply this weight by their weight for non-interaction flight. The operator then abandons here the primary track, letting it back to normal tracking.
    3. The copy of the primary track starts and the track is forced to interact in the volume, using the G4BOptnForceCommonTruncatedExp operation, itself using the total cross-section to compute the forced interaction law (exponential law limited to path lenght in the volume). One of the physics processes is randomly selected (on the basis of cross-section values) for the interaction.
    4. Other processes are receiving a forced free flight operation, from the operator.
    5. The copy of the primary is transported up to its interaction point. With these operations configured, the G4BiasingProcessInterface instances have all needed information to automatically compute the weight of the primary track and of its interaction products.

    As this operation starts on the volume boundary, a single force interaction occures: if the track survives the interaction (e.g Compton process), as it moved apart the boundary, the operator does not consider it further.
- G4VBiasingInteractionLaw classes. These classes describe the interaction law in term of a non-interaction probability over a segment of lenght l, and an "effective" cross-section for an interaction at distance l (see Physics Reference Manual, section generic biasing [to come]). An interaction law can also be sampled.
  - G4InteractionLawPhysical: the usual exponential law, driven by a cross-section constant over a step. The effective cross-section is the cross-section.
  - G4ILawForceFreeFlight: an "interaction" law for, precisely, a non-interacting track, with non-interaction probability always 1, and zero effective cross-section. It is a limit case of the modeling.
  - G4ILawTruncatedExp: an exponential interaction law limited to a segment [0,L]. The non-interaction probability and effective cross-section depend on l, the distance travelled, and become zero and infinite, respectively, at l=L.

Changes from 10.0 to 10.1

The G4VBiasingOperation class has been evoled to simplify the interface. The changes regard physics-based biasing (occurence biasing and final state biaising) and are:

- Suppression of the method virtual G4ForceCondition ProposeForceCondition(const G4ForceCondition wrappedProcessCondition)
  - The functionality has been kept, absorbing the ProposeForceCondition(...) method by the ProvideOccurenceBiasingInteractionLaw(...) one, which has now the signature:

```
``virtual const G4VBiasingInteractionLaw*
ProvideOccurenceBiasingInteractionLaw ( const
G4BiasingProcessInterface* callingProcess, G4ForceCondition&
proposeForceCondition) = 0;``
```

  - The value of proposeForceCondition passed to the method is the G4ForceCondition value of the wrapped process, as this was the case with deprecated method ProposeForceCondition(...) .
- Suppression of the virtual method "G4bool DenyProcessPostStepDoIt(const G4BiasingProcessInterface* callingProcess, const G4Track* track, const G4Step* step, G4double& proposedTrackWeight)":
  - This method was used to prevent the wrapped process hold by callingProcess to have its

PostStepDoIt(...) called, providing a weight for this non-call.
  - The method has been removed, but the functionality still exists, and has been merged and generalized with the change of the pure virtual ApplyFinalStateBiasing(...) described just after.
- Extra argument G4bool& forceBiasedFinalState added as last argument of "virtual G4VParticleChange* ApplyFinalStateBiasing(const G4BiasingProcessInterface* callingProcess, const G4Track* track, const G4Step* step, G4bool& forceBiasedFinalState) = 0"
  - This method is meant to return a final state interaction through the G4VParticleChange. The final state may be the analog wrapped process one, or a biased one, which comes with its weight correction for biaising the final state. If an occurence biasing is also at play in the same step, the weight correction for this biasing is applied to the final state before this one is returned to the stepping. This is the default behavior. This behavior can be controlled by forceBiasedFinalState:
    * If forceBiasedFinalState is left false, the above default behavior is applied.
    * If forceBiasedFinalState is set to true, the G4VParticleChange final state will be *returned as is* to the stepping, and that, *regardless* there is an occurence at play. Hence, when setting forceBiasedFinalState to true, the biasing operation *takes full responsibilty* for the total weight (occurence + final state) calculation.
- Deletion of G4ILawCommonTruncatedExp, which could be eliminated after better implementation of G4BOptnForceCommonTruncatedExp operation.

Changes from 10.1 to 10.2

Changes in 10.2 derive from the introduction of the track feature G4VAuxiliaryTrackInformation. They regard essentially the force collision operator G4BOptrForceCollision and related features. These changes are transparent to the user if using G4BOptrForceCollision and following examples/extended/biasing/GB02. The information below are provided for developers of biasing classes.

The G4VAuxiliaryTrackInformation functionnality allows to extend the G4Track attributes with an instance of a concrete class deriving from G4VAuxiliaryTrackInformation. Such an object is registered to the G4Track using an ID that has to be previously obtained from the G4PhysicsModelCatalog. The G4VBiasingOperator class defines two new virtual methods, Configure() and ConfigureForWorker(), to help with the creation of these ID's at the proper time (see G4BOptrForceCollision as an example).

Before 10.2, the G4BOptrForceCollision class was using state variables to make the bookkeeping of the tracks handled by the scheme. Now this bookkeeping is handled using a G4VAuxiliaryTrackInformation, G4BOptrForceCollisionTrackData.

To help with the bookkeeping, the base class G4VBiasingOperator was defining a set of methods (GetBirthOperation(..), RememberSecondaries(..), ForgetTrack(..)), these have been removed in 10.2 and are easy to overpass with a dedicated G4VAuxiliaryTrackInformation.

CHAPTER

FOUR

DETECTOR DEFINITION AND RESPONSE

4.1 Geometry

4.1.1 Introduction

The detector definition requires the representation of its geometrical elements, their materials and electronics properties, together with visualization attributes and user defined properties. The geometrical representation of detector elements focuses on the definition of solid models and their spatial position, as well as their logical relations to one another, such as in the case of containment.

Geant4 uses the concept of "Logical Volume" to manage the representation of detector element properties. The concept of "Physical Volume" is used to manage the representation of the spatial positioning of detector elements and their logical relations. The concept of "Solid" is used to manage the representation of the detector element solid modeling. Volumes and solids must be dynamically allocated using 'new' in the user program; they must not be declared as local objects. Volumes and solids are automatically registered on creation to dedicated stores; these stores will delete all objects at the end of the job.

4.1.2 Solids

The Geant4 geometry modeller implements Constructive Solid Geometry (CSG) representations for geometrical primitives. CSG representations are easy to use and normally give superior performance.

All solids must be allocated using 'new' in the user's program; they get registered to a `G4SolidStore` at construction, which will also take care to deallocate them at the end of the job, if not done already in the user's code.

All constructed solids can stream out their contents via appropriate methods and streaming operators.

For all solids it is possible to estimate the geometrical volume and the surface area by invoking the methods:

```
G4double GetCubicVolume()
G4double GetSurfaceArea()
```

which return an estimate of the solid volume and total area in internal units respectively. For elementary solids the functions compute the exact geometrical quantities, while for composite or complex solids an estimate is made using Monte Carlo techniques.

For all solids it is also possible to generate pseudo-random points lying on their surfaces, by invoking the method

```
G4ThreeVector GetPointOnSurface() const
```

which returns the generated point in local coordinates relative to the solid. To be noted that this function is not meant to provide a uniform distribution of points on the surfaces of the solids.

Since release 10.3, solids can be scaled in their dimensions along the Cartesian axes X, Y or Z, by providing a scale transformation associated to the original solid.

```
G4ScaledSolid( const G4String& pName,
                     G4VSolid* pSolid ,
               const G4Scale3D& pScale  )
```

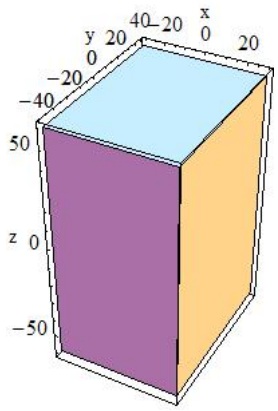
Constructed Solid Geometry (CSG) Solids

CSG solids are defined directly as three-dimensional primitives. They are described by a minimal set of parameters necessary to define the shape and size of the solid. CSG solids are Boxes, Tubes and their sections, Cones and their sections, Spheres, Wedges, and Toruses.

Box:

To create a **box** one can use the constructor:

```
G4Box(const G4String& pName,
            G4double  pX,
            G4double  pY,
            G4double  pZ)
```



In the picture:
pX = 30, pY = 40, pZ = 60

by giving the box a name and its half-lengths along the X, Y and Z axis:

pX	half length in X	pY	half length in Y	pZ	half length in Z

This will create a box that extends from −pX to +pX in X, from −pY to +pY in Y, and from −pZ to +pZ in Z.

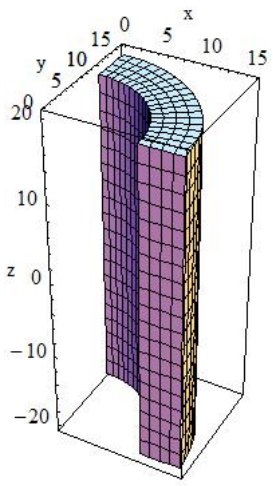
For example to create a box that is 2 by 6 by 10 centimeters in full length, and called BoxA one should use the following code:

```
G4Box* aBox = new G4Box("BoxA", 1.0*cm, 3.0*cm, 5.0*cm);
```

Cylindrical Section or Tube:

Similarly to create a **cylindrical section** or **tube**, one would use the constructor:

```
G4Tubs(const G4String& pName,
             G4double  pRMin,
             G4double  pRMax,
             G4double  pDz,
             G4double  pSPhi,
             G4double  pDPhi)
```



In the picture:
`pRMin = 10, pRMax = 15, pDz = 20`

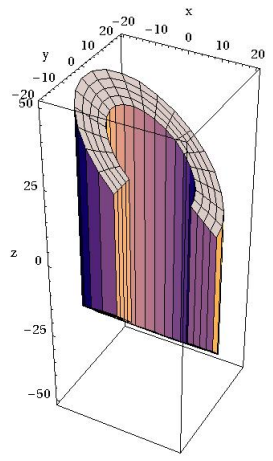
giving its name `pName` and its parameters which are:

`pRMin`	Inner radius	`pRMax`	Outer radius
`pDz`	Half length in z	`pSPhi`	Starting phi angle in radians
`pDPhi`	Angle of the segment in radians		

Cylindrical Cut Section or Cut Tube:

A cut in `Z` can be applied to a cylindrical section to obtain a **cut tube**. The following constructor should be used:

```
G4CutTubs( const G4String& pName,
                 G4double pRMin,
                 G4double pRMax,
                 G4double pDz,
                 G4double pSPhi,
                 G4double pDPhi,
                 G4ThreeVector pLowNorm,
                 G4ThreeVector pHighNorm)
```



In the picture:
`pRMin = 12, pRMax = 20, pDz = 30, pSPhi = 0, pDPhi = 1.5*pi, pLowNorm = (0,-0.7,-0.71), pHighNorm = (0.7,0,0.71)`

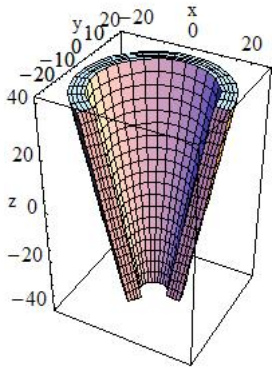
giving its name pName and its parameters which are:

pRMin	Inner radius	pRMax	Outer radius
pDz	Half length in z	pSPhi	Starting phi angle in radians
pDPhi	Angle of the segment in radians	pLowNorm	Outside Normal at -z
pHighNorm	Outside Normal at +z		

Cone or Conical section:

Similarly to create a **cone**, or **conical section**, one would use the constructor

```
G4Cons(const G4String& pName,
             G4double  pRmin1,
             G4double  pRmax1,
             G4double  pRmin2,
             G4double  pRmax2,
             G4double  pDz,
             G4double  pSPhi,
             G4double  pDPhi)
```



In the picture:
`pRmin1 = 5, pRmax1 = 10, pRmin2 = 20, pRmax2 = 25, pDz = 40, pSPhi = 0, pDPhi = 4/3*Pi`

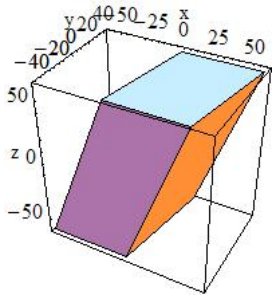
giving its name pName, and its parameters which are:

pRmin1	inside radius at −pDz	pRmax1	outside radius at −pDz
pRmin2	inside radius at +pDz	pRmax2	outside radius at +pDz
pDz	half length in z	pSPhi	starting angle of the segment in radians
pDPhi	the angle of the segment in radians		

Parallelepiped:

A **parallelepiped** is constructed using:

```
G4Para(const G4String& pName,
             G4double   dx,
             G4double   dy,
             G4double   dz,
             G4double   alpha,
             G4double   theta,
             G4double   phi)
```



In the picture:
`dx = 30, dy = 40, dz = 60`

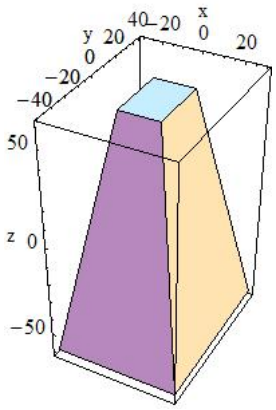
giving its name pName and its parameters which are:

dx,dy, dz	Half-length in x,y,z
alpha	Angle formed by the y axis and by the plane joining the centre of the faces *parallel* to the z-x plane at -dy and +dy
theta	Polar angle of the line joining the centres of the faces at -dz and +dz in z
phi	Azimuthal angle of the line joining the centres of the faces at -dz and +dz in z

Trapezoid:

To construct a **trapezoid** use:

```
G4Trd(const G4String& pName,
            G4double  dx1,
            G4double  dx2,
            G4double  dy1,
            G4double  dy2,
            G4double  dz)
```



In the picture: dx1 = 30, dx2 = 10, dy1 = 40, dy2 = 15, dz = 60

to obtain a solid with name pName and parameters

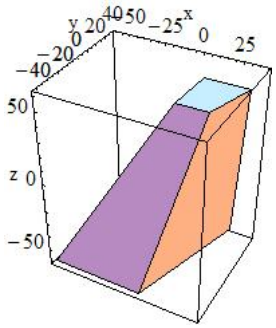
dx1	Half-length along x at the surface positioned at −dz
dx2	Half-length along x at the surface positioned at +dz
dy1	Half-length along y at the surface positioned at −dz
dy2	Half-length along y at the surface positioned at +dz
dz	Half-length along z axis

Generic Trapezoid:

To build a **generic trapezoid**, the G4Trap class is provided. Here are the two costructors for a Right Angular Wedge and for the general trapezoid for it:

```
G4Trap(const G4String& pName,
             G4double  pZ,
             G4double  pY,
             G4double  pX,
             G4double  pLTX)

G4Trap(const G4String& pName,
             G4double  pDz,    G4double pTheta,
             G4double  pPhi,   G4double pDy1,
             G4double  pDx1,   G4double pDx2,
             G4double  pAlp1,  G4double pDy2,
             G4double  pDx3,   G4double pDx4,
             G4double  pAlp2)
```



In the picture:
`pDx1 = 30, pDx2 = 40, pDy1 = 40, pDx3 = 10, pDx4 = 14, pDy2 = 16, pDz = 60, pTheta = 20*Degree, pPhi = 5*Degree, pAlp1 = pAlp2 = 10*Degree`

to obtain a Right Angular Wedge with name `pName` and parameters:

`pZ`	Length along z
`pY`	Length along y
`pX`	Length along x at the wider side
`pLTX`	Length along x at the narrower side (`plTX<=pX`)

or to obtain the general trapezoid:

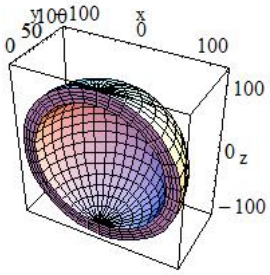
`pDx1`	Half x length of the side at y=-pDy1 of the face at -pDz
`pDx2`	Half x length of the side at y=+pDy1 of the face at -pDz
`pDz`	Half z length
`pTheta`	Polar angle of the line joining the centres of the faces at -/+pDz
`pPhi`	Azimuthal angle of the line joining the centre of the face at -pDz to the centre of the face at +pDz
`pDy1`	Half y length at -pDz
`pDy2`	Half y length at +pDz
`pDx3`	Half x length of the side at y=-pDy2 of the face at +pDz
`pDx4`	Half x length of the side at y=+pDy2 of the face at +pDz
`pAlp1`	Angle with respect to the y axis from the centre of the side (lower endcap)
`pAlp2`	Angle with respect to the y axis from the centre of the side (upper endcap)

Note: The angle `pAlph1` and `pAlph2` have to be the same due to the planarity condition.

Sphere or Spherical Shell Section:

To build a **sphere**, or a **spherical shell section**, use:

```
G4Sphere(const G4String& pName,
               G4double   pRmin,
               G4double   pRmax,
               G4double   pSPhi,
               G4double   pDPhi,
               G4double   pSTheta,
               G4double   pDTheta )
```



In the picture:
`pRmin = 100, pRmax = 120, pSPhi = 0*Degree, pDPhi = 180*Degree, pSTheta = 0 Degree, pDTheta = 180*Degree`

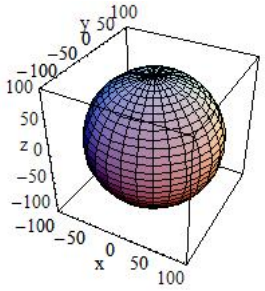
to obtain a solid with name `pName` and parameters:

`pRmin`	Inner radius
`pRmax`	Outer radius
`pSPhi`	Starting Phi angle of the segment in radians
`pDPhi`	Delta Phi angle of the segment in radians
`pSTheta`	Starting Theta angle of the segment in radians
`pDTheta`	Delta Theta angle of the segment in radians

Full Solid Sphere:

To build a **full solid sphere** use:

```
G4Orb(const G4String& pName,
            G4double  pRmax)
```



In the picture:
`pRmax = 100`

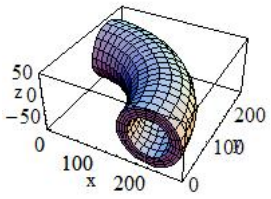
The Orb can be obtained from a Sphere with: `pRmin` = 0, `pSPhi` = 0, `pDPhi` = $2 * \pi$, `pSTheta` = 0, `pDTheta` = π

`pRmax`	Outer radius

Torus:

To build a **torus** use:

```
G4Torus(const G4String& pName,
              G4double   pRmin,
              G4double   pRmax,
              G4double   pRtor,
              G4double   pSPhi,
              G4double   pDPhi)
```



In the picture:
`pRmin = 40, pRmax = 60, pRtor = 200, pSPhi = 0, pDPhi = 90*degree`

to obtain a solid with name `pName` and parameters:

`pRmin`	Inside radius
`pRmax`	Outside radius
`pRtor`	Swept radius of torus
`pSPhi`	Starting Phi angle in radians (`fSPhi+fDPhi<=2PI`, `fSPhi>-2PI`)
`pDPhi`	Delta angle of the segment in radians

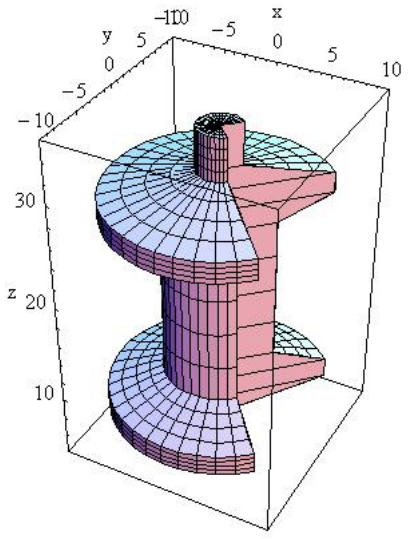
In addition, the Geant4 Design Documentation shows in the Solids Class Diagram the complete list of CSG classes.

Specific CSG Solids

Polycons:

Polycons (PCON) are implemented in Geant4 through the `G4Polycone` class:

```
G4Polycone(const G4String& pName,
                 G4double   phiStart,
                 G4double   phiTotal,
                 G4int      numZPlanes,
           const G4double   zPlane[],
           const G4double   rInner[],
           const G4double   rOuter[])
```



In the picture:
`phiStart = 1/4*Pi, phiTotal = 3/2*Pi, numZPlanes = 9, rInner = { 0, 0, 0, 0, 0, 0, 0, 0, 0}, rOuter = { 0, 10, 10, 5 , 5, 10 , 10 , 2, 2}, z = { 5, 7, 9, 11, 25, 27, 29, 31, 35 }`

where:

phiStart	Initial Phi starting angle
phiTotal	Total Phi angle
numZPlanes	Number of z planes
numRZ	Number of corners in r,z space
zPlane	Position of z planes, with z in increasing order
rInner	Tangent distance to inner surface
rOuter	Tangent distance to outer surface
r	r coordinate of corners
z	z coordinate of corners

A **Polycone** where Z planes position can also decrease is implemented through the G4GenericPolycone class:

```
G4GenericPolycone(const G4String& pName,
                        G4double   phiStart,
                        G4double   phiTotal,
                        G4int      numRZ,
                  const G4double  r[],
                  const G4double  z[])
```

where:

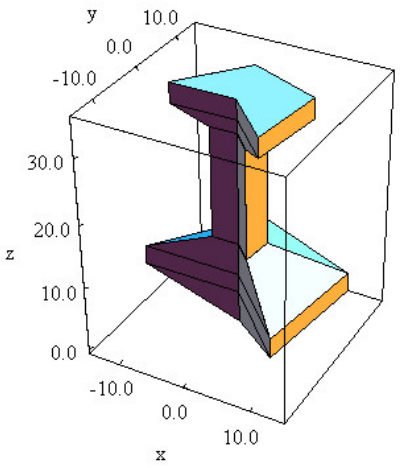
phiStart	Initial Phi starting angle
phiTotal	Total Phi angle
numRZ	Number of corners in r,z space
r	r coordinate of corners
z	z coordinate of corners

Polyhedra (PGON):

Polyhedra (PGON) are implemented through G4Polyhedra:

```
G4Polyhedra(const G4String& pName,
                  G4double  phiStart,
                  G4double  phiTotal,
                  G4int     numSide,
                  G4int     numZPlanes,
            const G4double  zPlane[],
            const G4double  rInner[],
            const G4double  rOuter[] )

G4Polyhedra(const G4String& pName,
                  G4double  phiStart,
                  G4double  phiTotal,
                  G4int     numSide,
                  G4int     numRZ,
            const G4double  r[],
            const G4double  z[] )
```



In the picture:
`phiStart = -1/4*Pi, phiTotal= 5/4*Pi, numSide = 3, nunZPlanes = 7, rInner = { 0, 0, 0, 0, 0, 0, 0 }, rOuter = { 0, 15, 15, 4, 4, 10, 10 }, z = { 0, 5, 8, 13 , 30, 32, 35 }`

where:

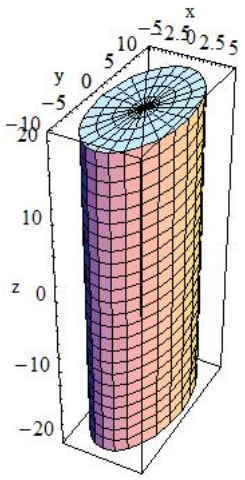
`phiStart`	Initial Phi starting angle
`phiTotal`	Total Phi angle
`numSide`	Number of sides
`numZPlanes`	Number of z planes
`numRZ`	Number of corners in r,z space
`zPlane`	Position of z planes
`rInner`	Tangent distance to inner surface
`rOuter`	Tangent distance to outer surface
`r`	r coordinate of corners
`z`	z coordinate of corners

Tube with an elliptical cross section:

A **tube with an elliptical cross section** (ELTU) can be defined as follows:

```
G4EllipticalTube(const G4String& pName,
                 G4double  Dx,
                 G4double  Dy,
                 G4double  Dz)
```

The equation of the surface in x/y is `1.0 = (x/dx)**2 +(y/dy)**2`



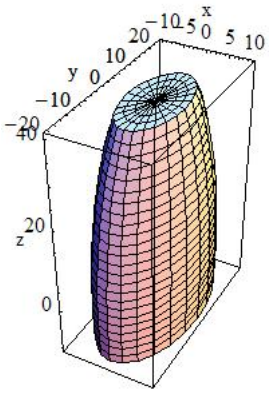
In the picture
`Dx = 5, Dy = 10, Dz = 20`

Dx	Half length X	Dy	Half length Y	Dz	Half length Z

General Ellipsoid:

The general **ellipsoid** with possible cut in `Z` can be defined as follows:

```
G4Ellipsoid(const G4String& pName,
                  G4double  pxSemiAxis,
                  G4double  pySemiAxis,
                  G4double  pzSemiAxis,
                  G4double  pzBottomCut=0,
                  G4double  pzTopCut=0)
```



In the picture:
`pxSemiAxis = 10, pySemiAxis = 20, pzSemiAxis = 50, pzBottomCut = -10, pzTopCut = 40`

A general (or triaxial) ellipsoid is a quadratic surface which is given in Cartesian coordinates by:

```
1.0 = (x/pxSemiAxis)**2 + (y/pySemiAxis)**2 + (z/pzSemiAxis)**2
```

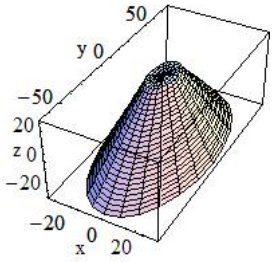
where:

pxSemiAxis	Semiaxis in X
pySemiAxis	Semiaxis in Y
pzSemiAxis	Semiaxis in Z
pzBottomCut	lower cut plane level, z
pzTopCut	upper cut plane level, z

Cone with Elliptical Cross Section:

A **cone with an elliptical cross section** can be defined as follows:

```
G4EllipticalCone(const G4String& pName,
                       G4double  pxSemiAxis,
                       G4double  pySemiAxis,
                       G4double  zMax,
                       G4double  pzTopCut)
```



In the picture:
`pxSemiAxis = 30/75, pySemiAxis = 60/75, zMax = 50, pzTopCut = 25`

where:

pxSemiAxis	Semiaxis in X
pySemiAxis	Semiaxis in Y
zMax	Height of elliptical cone
pzTopCut	upper cut plane level

An elliptical cone of height `zMax`, with two bases at `-pzTopCut` and `+pzTopCut`, semiaxis `pxSemiAxis`, and semiaxis `pySemiAxis` is given by the parametric equations:

```
x = pxSemiAxis * ( zMax - u ) / u * Cos(v)
y = pySemiAxis * ( zMax - u ) / u * Sin(v)
z = u
```

Where `v` is between 0 and 2π, and `u` between `-pzTopCut` and `+pzTopCut` respectively.

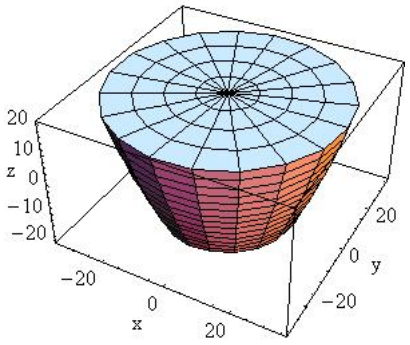
Paraboloid, a solid with parabolic profile:

A **solid with parabolic profile** and possible cuts along the `Z` axis can be defined as follows:

```
G4Paraboloid(const G4String& pName,
                   G4double  Dz,
                   G4double  R1,
                   G4double  R2)
```

The equation for the solid is:

```
rho**2 <= k1 * z + k2;
    -dz <= z <= dz
r1**2 = k1 * (-dz) + k2
r2**2 = k1 * ( dz) + k2
```



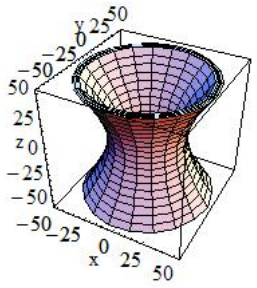
In the picture:
`R1 = 20, R2 = 35, Dz = 20`

`Dz`	Half length Z	`R1`	Radius at -Dz	`R2`	Radius at +Dz greater than R1

Tube with Hyperbolic Profile:

A **tube with a hyperbolic profile** (HYPE) can be defined as follows:

```
G4Hype(const G4String& pName,
             G4double  innerRadius,
             G4double  outerRadius,
             G4double  innerStereo,
             G4double  outerStereo,
             G4double  halfLenZ)
```



In the picture:
`innerStereo = 0.7, outerStereo = 0.7, halfLenZ = 50, innerRadius = 20, outerRadius = 30`

`G4Hype` is shaped with curved sides parallel to the `z`-axis, has a specified half-length along the `z` axis about which it is centred, and a given minimum and maximum radius.

A minimum radius of `0` defines a filled Hype (with hyperbolic inner surface), i.e. inner radius = 0 AND inner stereo angle = 0.

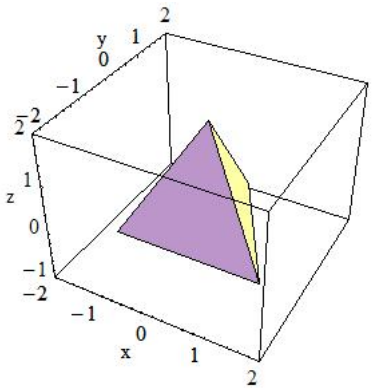
The inner and outer hyperbolic surfaces can have different stereo angles. A stereo angle of `0` gives a cylindrical surface:

`innerRadius`	Inner radius
`outerRadius`	Outer radius
`innerStereo`	Inner stereo angle in radians
`outerStereo`	Outer stereo angle in radians
`halfLenZ`	Half length in Z

Tetrahedra:

A **tetrahedra** solid can be defined as follows:

```
G4Tet(const G4String& pName,
            G4ThreeVector  anchor,
            G4ThreeVector  p2,
            G4ThreeVector  p3,
            G4ThreeVector  p4,
            G4bool         *degeneracyFlag=0)
```



In the picture:
`anchor = {0, 0, sqrt(3)}, p2 = { 0, 2*sqrt(2/3), -1/sqrt(3) }, p3 = { -sqrt(2), -sqrt(2/3),-1/sqrt(3) }, p4 = { sqrt(2), -sqrt(2/3) , -1/sqrt(3) }`

The solid is defined by 4 points in space:

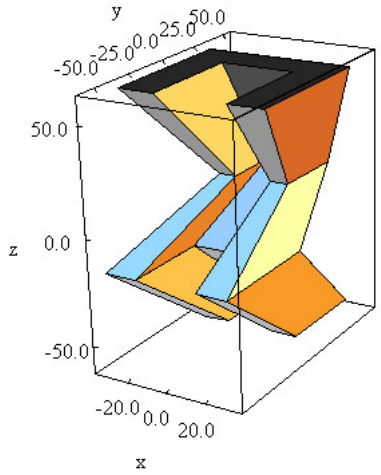
`anchor`	Anchor point
`p2`	Point 2
`p3`	Point 3
`p4`	Point 4
`degeneracyFlag`	Flag indicating degeneracy of points

Extruded Polygon:

The extrusion of an arbitrary polygon (**extruded solid**) with fixed outline in the defined `Z` sections can be defined as follows (in a general way, or as special construct with two `Z` sections):

```
G4ExtrudedSolid(const G4String& pName,
     std::vector<G4TwoVector> polygon,
        std::vector<ZSection> zsections)

G4ExtrudedSolid(const G4String& pName,
     std::vector<G4TwoVector> polygon,
                       G4double     hz,
     G4TwoVector off1, G4double scale1,
     G4TwoVector off2, G4double scale2)
```



In the picture:

```
poligon = {-30,-30},{-30,30},{30,30},{30,-30},
     {15,-30},{15,15},{-15,15},{-15,
     -30}
zsections = [-60,{0,30},0.8], [-15, {0,-30},1.],
     [10,{0,0},0.6], [60,{0,30},1.2]
```

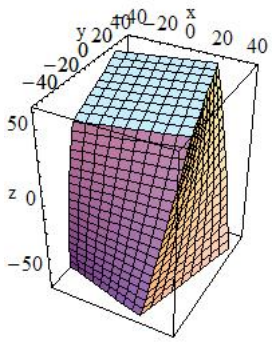
The z-sides of the solid are the scaled versions of the same polygon.

`polygon`	the vertices of the outlined polygon defined in clock-wise order
`zsections`	the z-sections defined by z position in increasing order
`hz`	Half length in Z
`off1, off2`	Offset of the side in -hz and +hz respectively
`scale1, scale2`	Scale of the side in -hz and +hz respectively

Box Twisted:

A **box twisted** along one axis can be defined as follows:

```
G4TwistedBox(const G4String& pName,
                   G4double  twistedangle,
                   G4double  pDx,
                   G4double  pDy,
                   G4double  pDz)
```



In the picture:

```
twistedangle = 30*Degree, pDx = 30,
pDy =40, pDz = 60
```

`G4TwistedBox` is a box twisted along the z-axis. The twist angle cannot be greater than 90 degrees:

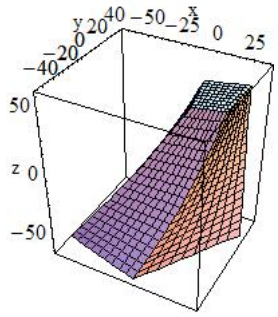
twistedangle	Twist angle
pDx	Half x length
pDy	Half y length
pDz	Half z length

Trapezoid Twisted along One Axis:

trapezoid twisted along one axis can be defined as follows:

```
G4TwistedTrap(const G4String& pName,
                    G4double  twistedangle,
                    G4double  pDxx1,
                    G4double  pDxx2,
                    G4double  pDy,
                    G4double   pDz)

G4TwistedTrap(const G4String& pName,
                    G4double  twistedangle,
                    G4double  pDz,
                    G4double  pTheta,
                    G4double  pPhi,
                    G4double  pDy1,
                    G4double  pDx1,
                    G4double  pDx2,
                    G4double  pDy2,
                    G4double  pDx3,
                    G4double  pDx4,
                    G4double  pAlph)
```



In the picture:
`pDx1 = 30, pDx2 = 40, pDy1 = 40, pDx3 = 10, pDx4 = 14, pDy2 = 16, pDz = 60, pTheta = 20*Degree, pDphi = 5*Degree, pAlph = 10*Degree, twistedangle = 30*Degree`

The first constructor of `G4TwistedTrap` produces a regular trapezoid twisted along the `z`-axis, where the caps of the trapezoid are of the same shape and size.

The second constructor produces a generic trapezoid with polar, azimuthal and tilt angles.

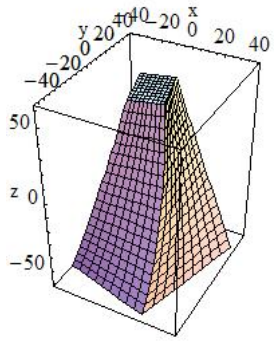
The twist angle cannot be greater than 90 degrees:

twistedangle	Twisted angle
pDx1	Half x length at y=-pDy
pDx2	Half x length at y=+pDy
pDy	Half y length
pDz	Half z length
pTheta	Polar angle of the line joining the centres of the faces at -/+pDz
pDy1	Half y length at -pDz
pDx1	Half x length at -pDz, y=-pDy1
pDx2	Half x length at -pDz, y=+pDy1
pDy2	Half y length at +pDz
pDx3	Half x length at +pDz, y=-pDy2
pDx4	Half x length at +pDz, y=+pDy2
pAlph	Angle with respect to the y axis from the centre of the side

Twisted Trapezoid with x and y dimensions varying along z:

A **twisted trapezoid** with the `x` and `y` dimensions **varying along** `z` can be defined as follows:

```
G4TwistedTrd(const G4String& pName,
                   G4double  pDx1,
                   G4double  pDx2,
                   G4double  pDy1,
                   G4double  pDy2,
                   G4double  pDz,
                   G4double  twistedangle)
```



In the picture:
`dx1 = 30, dx2 = 10, dy1 = 40, dy2 = 15, dz = 60, twistedangle = 30*Degree`

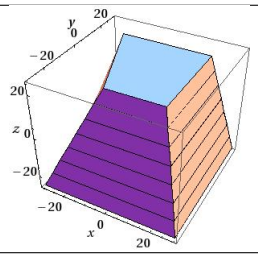
where:

`pDx1`	Half x length at the surface positioned at -dz
`pDx2`	Half x length at the surface positioned at +dz
`pDy1`	Half y length at the surface positioned at -dz
`pDy2`	Half y length at the surface positioned at +dz
`pDz`	Half z length
`twistedangle`	Twisted angle

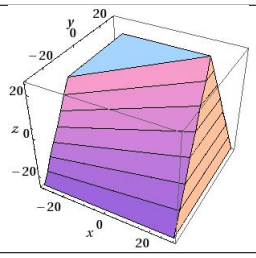
Generic trapezoid with optionally collapsing vertices:

An **arbitrary trapezoid** with up to 8 vertices standing on two parallel planes perpendicular to the `Z` axis can be defined as follows:

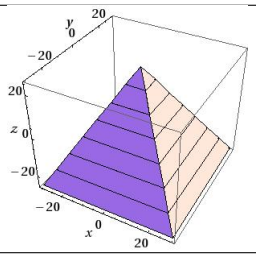
```
G4GenericTrap(const G4String& pName,
                    G4double  pDz,
              const std::vector<G4TwoVector>& vertices)
```



In the picture:
`pDz = 25 vertices = {-30, -30}, {-30, 30}, {30, 30}, {30, -30} {-5, -20}, {-20, 20}, {20, 20}, {20, -20}`



In the picture:
`pDz = 25 vertices = {-30, -30}, {-30,30}, {30,30}, {30,-30} {-20,-20},{-20, 20}, {20,20}, {20, 20}`



In the picture:
`pDz = 25 vertices = {-30,-30}, {-30,30}, {30,30}, {30,-30} {0,0}, {0,0}, {0,0}, {0,0}`

where:

pDz	Half z length
vertices	The (x,y) coordinates of vertices

The order of specification of the coordinates for the vertices in G4GenericTrap is important. The first four points are the vertices sitting on the -hz plane; the last four points are the vertices sitting on the +hz plane.

The order of defining the vertices of the solid is the following:

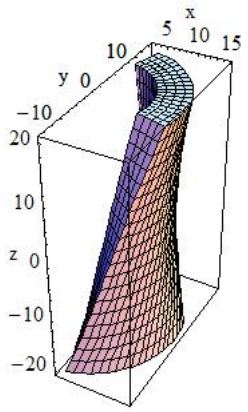
- point 0 is connected with points 1,3,4
- point 1 is connected with points 0,2,5
- point 2 is connected with points 1,3,6
- point 3 is connected with points 0,2,7
- point 4 is connected with points 0,5,7
- point 5 is connected with points 1,4,6
- point 6 is connected with points 2,5,7
- point 7 is connected with points 3,4,6

Points can be identical in order to create shapes with less than 8 vertices; the only limitation is to have at least one triangle at +hz or -hz; the lateral surfaces are not necessarily planar. Not planar lateral surfaces are represented by a surface that linearly changes from the edge on -hz to the corresponding edge on +hz; it represents a *sweeping* surface with twist angle linearly dependent on Z, but it is not a real twisted surface mathematically described by equations as for the other *twisted* solids described in this chapter.

Tube Section Twisted along Its Axis:

A **tube section twisted** along its axis can be defined as follows:

```
G4TwistedTubs(const G4String& pName,
                    G4double  twistedangle,
                    G4double  endinnerrad,
                    G4double  endouterrad,
                    G4double  halfzlen,
                    G4double  dphi)
```



In the picture:
endinnerrad = 10, endouterrad = 15,
halfzlen = 20, dphi = 90*Degree,
twistedangle = 60*Degree

G4TwistedTubs is a sort of twisted cylinder which, placed along the z-axis and divided into phi-segments is shaped like an hyperboloid, where each of its segmented pieces can be tilted with a stereo angle.

It can have inner and outer surfaces with the same stereo angle:

twistedangle	Twisted angle
endinnerrad	Inner radius at endcap
endouterrad	Outer radius at endcap
halfzlen	Half z length
dphi	Phi angle of a segment

Additional constructors are provided, allowing the shape to be specified either as:

- the number of segments in phi and the total angle for all segments, or
- a combination of the above constructors providing instead the inner and outer radii at z=0 with different z-lengths along negative and positive z-axis.

Solids made by Boolean operations

Simple solids can be combined using Boolean operations. For example, a cylinder and a half-sphere can be combined with the union Boolean operation.

Creating such a new *Boolean* solid, requires:

- Two solids
- A Boolean operation: union, intersection or subtraction.
- Optionally a transformation for the second solid.

The solids used should be either CSG solids (for examples a box, a spherical shell, or a tube) or another Boolean solid: the product of a previous Boolean operation. An important purpose of Boolean solids is to allow the description of solids with peculiar shapes in a simple and intuitive way, still allowing an efficient geometrical navigation inside them.

Note: The constituent solids of a Boolean operation should possibly *avoid* be composed by sharing all or part of their surfaces. This precaution is necessary in order to avoid the generation of 'fake' surfaces due to precision loss, or errors in the final visualization of the Boolean shape. In particular, if any one of the *subtractor* surfaces is coincident with a surface of the *subtractee*, the result is undefined. Moreover, the final Boolean solid should represent a single 'closed' solid, i.e. a Boolean operation between two solids which are disjoint or far apart each other, is *not* a valid Boolean composition.

Note: The tracking cost for navigating in a Boolean solid in the current implementation, is proportional to the number of constituent solids. So care must be taken to avoid extensive, unecessary use of Boolean solids in performance-critical areas of a geometry description, where each solid is created from Boolean combinations of many other solids.

Examples of the creation of the simplest Boolean solids are given below:

```
G4Box* box =
  new G4Box("Box",20*mm,30*mm,40*mm);
G4Tubs* cyl =
  new G4Tubs("Cylinder",0,50*mm,50*mm,0,twopi);  // r:      0 mm -> 50 mm
                                                 // z:   -50 mm -> 50 mm
                                                 // phi:   0 ->  2 pi
G4UnionSolid* union =
  new G4UnionSolid("Box+Cylinder", box, cyl);
G4IntersectionSolid* intersection =
  new G4IntersectionSolid("Box*Cylinder", box, cyl);
G4SubtractionSolid* subtraction =
  new G4SubtractionSolid("Box-Cylinder", box, cyl);
```

where the union, intersection and subtraction of a box and cylinder are constructed.

The more useful case where one of the solids is displaced from the origin of coordinates also exists. In this case the second solid is positioned relative to the coordinate system (and thus relative to the first). This can be done in two ways:

- Either by giving a rotation matrix and translation vector that are used to transform the coordinate system of the second solid to the coordinate system of the first solid. This is called the *passive* method.
- Or by creating a transformation that moves the second solid from its desired position to its standard position, e.g., a box's standard position is with its centre at the origin and sides parallel to the three axes. This is called the *active* method.

In the first case, the translation is applied first to move the origin of coordinates. Then the rotation is used to rotate the coordinate system of the second solid to the coordinate system of the first.

```
G4RotationMatrix* yRot = new G4RotationMatrix;  // Rotates X and Z axes only
yRot->rotateY(M_PI/4.*rad);                     // Rotates 45 degrees
G4ThreeVector zTrans(0, 0, 50);

G4UnionSolid* unionMoved =
  new G4UnionSolid("Box+CylinderMoved", box, cyl, yRot, zTrans);
//
// The new coordinate system of the cylinder is translated so that
// its centre is at +50 on the original Z axis, and it is rotated
// with its X axis halfway between the original X and Z axes.

// Now we build the same solid using the alternative method
//
G4RotationMatrix invRot = yRot->invert();
G4Transform3D transform(invRot, zTrans);
G4UnionSolid* unionMoved =
  new G4UnionSolid("Box+CylinderMoved", box, cyl, transform);
```

Note that the first constructor that takes a pointer to the rotation-matrix (`G4RotationMatrix*`), does NOT copy it. Therefore once used a rotation-matrix to construct a Boolean solid, it must NOT be modified.

In contrast, with the alternative method shown, a `G4Transform3D` is provided to the constructor by value, and its transformation is stored by the Boolean solid. The user may modify the `G4Transform3D` and eventually use it again.

When positioning a volume associated to a Boolean solid, the relative center of coordinates considered for the positioning is the one related to the *first* of the two constituent solids.

Tessellated Solids

In Geant4 it is also implemented a class `G4TessellatedSolid` which can be used to generate a generic solid defined by a number of facets (`G4VFacet`). Such constructs are especially important for conversion of complex geometrical shapes imported from CAD systems bounded with generic surfaces into an approximate description with facets of defined dimension (see *Tessellated solid*).

They can also be used to generate a solid bounded with a generic surface made of planar facets. It is important that the supplied facets shall form a fully enclose space to represent the solid.

Two types of facet can be used for the construction of a `G4TessellatedSolid`: a triangular facet (`G4TriangularFacet`) and a quadrangular facet (`G4QuadrangularFacet`).

An example on how to generate a simple tessellated shape is given below.

Listing 4.1: Example of geometries imported from CAD system and converted to tessellated solids.

```
// First declare a tessellated solid
//
G4TessellatedSolid solidTarget = new G4TessellatedSolid("Solid_name");
```

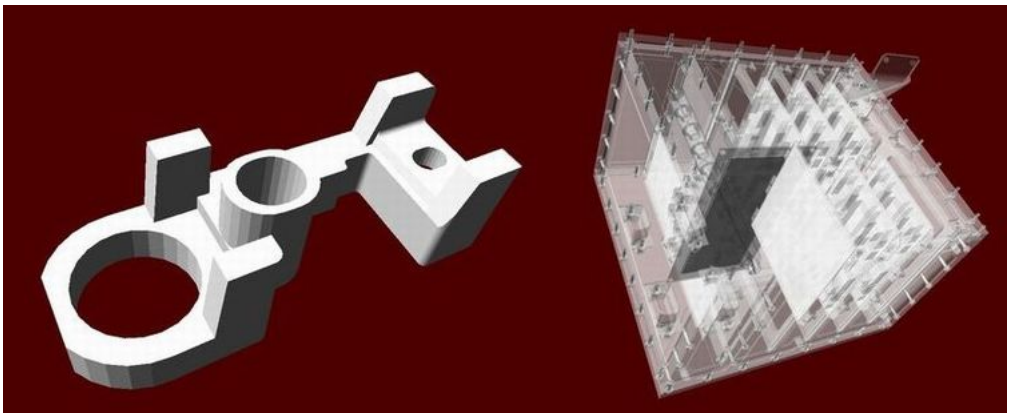


Fig. 4.1: Example of geometries imported from CAD system and converted to tessellated solids.

```
// Define the facets which form the solid
//
G4double targetSize = 10*cm ;
G4TriangularFacet *facet1 = new
G4TriangularFacet (G4ThreeVector(-targetSize,-targetSize,        0.0),
                   G4ThreeVector(+targetSize,-targetSize,        0.0),
                   G4ThreeVector(        0.0,        0.0,+targetSize),
                   ABSOLUTE);
G4TriangularFacet *facet2 = new
G4TriangularFacet (G4ThreeVector(+targetSize,-targetSize,        0.0),
                   G4ThreeVector(+targetSize,+targetSize,        0.0),
                   G4ThreeVector(        0.0,        0.0,+targetSize),
                   ABSOLUTE);
G4TriangularFacet *facet3 = new
G4TriangularFacet (G4ThreeVector(+targetSize,+targetSize,        0.0),
                   G4ThreeVector(-targetSize,+targetSize,        0.0),
                   G4ThreeVector(        0.0,        0.0,+targetSize),
                   ABSOLUTE);
G4TriangularFacet *facet4 = new
G4TriangularFacet (G4ThreeVector(-targetSize,+targetSize,        0.0),
                   G4ThreeVector(-targetSize,-targetSize,        0.0),
                   G4ThreeVector(        0.0,        0.0,+targetSize),
                   ABSOLUTE);
G4QuadrangularFacet *facet5 = new
G4QuadrangularFacet (G4ThreeVector(-targetSize,-targetSize,        0.0),
                     G4ThreeVector(-targetSize,+targetSize,        0.0),
                     G4ThreeVector(+targetSize,+targetSize,        0.0),
                     G4ThreeVector(+targetSize,-targetSize,        0.0),
                     ABSOLUTE);

// Now add the facets to the solid
//
solidTarget->AddFacet((G4VFacet*) facet1);
solidTarget->AddFacet((G4VFacet*) facet2);
solidTarget->AddFacet((G4VFacet*) facet3);
solidTarget->AddFacet((G4VFacet*) facet4);
solidTarget->AddFacet((G4VFacet*) facet5);

Finally declare the solid is complete
//
solidTarget->SetSolidClosed(true);
```

The `G4TriangularFacet` class is used for the contruction of `G4TessellatedSolid`. It is defined by three vertices, which shall be supplied in *anti-clockwise order* looking from the outside of the solid where it belongs. Its constructor looks like:

```
G4TriangularFacet ( const G4ThreeVector     Pt0,
                    const G4ThreeVector     vt1,
                    const G4ThreeVector     vt2,
                          G4FacetVertexType fType )
```

i.e., it takes 4 parameters to define the three vertices:

G4FacetVertexType	ABSOLUTE in which case Pt0, vt1 and vt2 are the three vertices in anti-clockwise order looking from the outside.
G4FacetVertexType	RELATIVE in which case the first vertex is Pt0, the second vertex is Pt0+vt1 and the third vertex is Pt0+vt2, all in anti-clockwise order when looking from the outside.

The G4QuadrangularFacet class can be used for the contruction of G4TessellatedSolid as well. It is defined by four vertices, which shall be in the same plane and be supplied in *anti-clockwise order* looking from the outside of the solid where it belongs. Its constructor looks like:

```
G4QuadrangularFacet ( const G4ThreeVector     Pt0,
                      const G4ThreeVector     vt1,
                      const G4ThreeVector     vt2,
                      const G4ThreeVector     vt3,
                            G4FacetVertexType fType )
```

i.e., it takes 5 parameters to define the four vertices:

G4FacetVertexType	ABSOLUTE in which case Pt0, vt1, vt2 and vt3 are the four vertices required in anti-clockwise order when looking from the outside.
G4FacetVertexType	RELATIVE in which case the first vertex is Pt0, the second vertex is Pt0+vt, the third vertex is Pt0+vt2 and the fourth vertex is Pt0+vt3, in anti-clockwise order when looking from the outside.

Importing CAD models as tessellated shapes

Tessellated solids can also be used to import geometrical models from CAD systems (see *fig-geom-solid-1*). In order to do this, it is required to convert first the CAD shapes into tessellated surfaces. A way to do this is to save the shapes in the geometrical model as STEP files and convert them to tessellated (faceted surfaces) solids, using a tool which allows such conversion. At the time of writing, at least two tools are available for such purpose: STViewer (part of the STEP-Tools development suite) or FASTRAD. This strategy allows to import any shape with some degree of approximation; the converted CAD models can then be imported through GDML (Geometry Description Markup Language) into Geant4 and be represented as G4TessellatedSolid shapes.

Other tools which can be used to generate meshes to be then imported in Geant4 as tessellated solids are:

- InStep - A free STL to GDML conversion tool.
- SALOME - Open-source software allowing to import STEP/BREP/IGES/STEP/ACIS formats, mesh them and export to STL.
- ESABASE2 - Space environment analysis CAD, basic modules free for academic non-commercial use. Can import STEP files and export to GDML shapes or complete geometries.
- CADMesh - Tool based on the VCG Library to read STL files and import in Geant4.
- Cogenda - Commercial TCAD software for generation of 3D meshes through the module Gds2Mesh and final export to GDML.

Unified Solids

An alternative implementation for some of the cited geometrical primitives is provided since release 10.0 of Geant4. The solids included in release 10.2 are: Box, Cons, Polycone, GenericPolycone, Polyhedra, Paraboloid, Orb, Sphere,

Tet, Trd, Trap, GenericTrap, Tubs, Torus, ExtrudedSolid and MultiUnion.

The new `G4MultiUnion` structure, in particular, allows for the description of a Boolean union of many displaced solids at once, therefore representing volumes with the same associated material. *NOTE: MultiUnion structures can only be defined for usage with USolids primitives enabled !* An example on how to define a simple MultiUnion structure is given here:

```
#include "G4MultiUnion.hh"

// Define two -UBox- shapes
//
G4Box* box1 = new G4Box("Box1", 5.*mm, 5.*mm, 10.*mm);
G4Box* box2 = new G4Box("Box2", 5.*mm, 5.*mm, 10.*mm);

// Define displacements for the shapes
//
G4RotationMatrix rotm  = G4RotationMatrix();
G4ThreeVector position1 = G4ThreeVector(0.,0.,1.);
G4ThreeVector position2 = G4ThreeVector(0.,0.,2.);
G4Transform3D tr1 = G4Transform3D(rotm,position1);
G4Transform3D tr2 = G4Transform3D(rotm,position2);

// Initialise a MultiUnion structure
//
G4MultiUnion* munion_solid = new G4MultiUnion("Boxes_Union");

// Add the shapes to the structure
//
munion_solid->AddNode(*box1,tr1);
munion_solid->AddNode(*box2,tr2);

// Finally close the structure
//
munion_solid->Voxelize();

// Associate it to a logical volume as a normal solid
//
G4LogicalVolume* lvol =
new G4LogicalVolume(munion_solid,         // its solid
                    munion_mat,           // its material
                    "Boxes_Union_LV");    // its name
```

The code for the USolids primitives originated as part of the AIDA Unified Solids Library and is now integrated in the VecGeom library (the vectorized geometry library for particle-detector simulation); it is provided for experimental use and can be activated in place of the original primitives defined in Geant4, by selecting the appropriate compilation flag when configuring the Geant4 libraries installation. The installation allows to build against an external system installation of the VecGeom library, therefore the appropriate installation path must also be provided during the installation configuration:

```
-DGEANT4_USE_USOLIDS="all"      // to replace all available shapes
-DGEANT4_USE_USOLIDS="box;tubs" // to replace only individual shapes
```

The original API for all geometrical primitives is preserved.

4.1.3 Logical Volumes

The Logical Volume manages the information associated with detector elements represented by a given Solid and Material, independently from its physical position in the detector.

`G4LogicalVolumes` must be allocated using 'new' in the user's program; they get registered to a `G4LogicalVolumeStore` at construction, which will also take care to deallocate them at the end of the job, if not done already in the user's code.

A Logical Volume knows which physical volumes are contained within it. It is uniquely defined to be their mother volume. A Logical Volume thus represents a hierarchy of unpositioned volumes whose positions relative to one another are well defined. By creating Physical Volumes, which are placed instances of a Logical Volume, this hierarchy or tree can be repeated.

A Logical Volume also manages the information relative to the Visualization attributes (*Visualization Attributes*) and user-defined parameters related to tracking, electro-magnetic field or cuts (through the `G4UserLimits` interface).

By default, tracking optimization of the geometry (voxelization) is applied to the volume hierarchy identified by a logical volume. It is possible to change the default behavior by choosing not to apply geometry optimization for a given logical volume. This feature does not apply to the case where the associated physical volume is a parameterised volume; in this case, optimization is always applied.

```
G4LogicalVolume( G4VSolid*             pSolid,
                 G4Material*           pMaterial,
                 const G4String&       Name,
                 G4FieldManager*       pFieldMgr=0,
                 G4VSensitiveDetector* pSDetector=0,
                 G4UserLimits*         pULimits=0,
                 G4bool                Optimise=true )
```

Through the logical volume it is also possible to *tune* the granularity of the optimisation algorithm to be applied to the sub-tree of volumes represented. This is possible using the methods:

```
G4double GetSmartless() const
void SetSmartless(G4double s)
```

The default *smartless* value is *2* and controls the average number of slices per contained volume which are used in the optimisation. The smaller the value, the less fine grained optimisation grid is generated; this will translate in a possible reduction of memory consumed for the optimisation of that portion of geometry at the price of a slight CPU time increase at tracking time. Manual tuning of the optimisation is in general not required, since the optimal granularity level is computed automatically and adapted to the specific geometry setup; however, in some cases (like geometry portions with 'dense' concentration of volumes distributed in a non-uniform way), it may be necessary to adopt manual tuning for helping the optimisation process in dealing with the most critical areas. By setting the verbosity to *2* through the following UI run-time command:

```
/run/verbose 2
```

a statistics of the memory consumed for the allocated optimisation nodes will be displayed volume by volume, allowing to easily identify the critical areas which may eventually require manual intervention.

The logical volume provides a way to estimate the *mass* of a tree of volumes defining a detector or sub-detector. This can be achieved by calling the method:

```
G4double GetMass(G4bool forced=false)
```

The mass of the logical volume tree is computed from the estimated geometrical volume of each solid and material associated with the logical volume and its daughters. Note that this computation may require a considerable amount of time, depending on the complexity of the geometry tree. The returned value is cached by default and can be used for successive calls, unless recomputation is forced by providing `true` for the boolean argument `forced` in input. Computation should be forced if the geometry setup has changed after the previous call.

Finally, the Logical Volume manages the information relative to the Envelopes hierarchy required for fast Monte Carlo parameterisations (*Parameterization*).

Sub-detector Regions

In complex geometry setups, such as those found in large detectors in particle physics experiments, it is useful to think of specific Logical Volumes as representing parts (sub-detectors) of the entire detector setup which perform specific

functions. In such setups, the processing speed of a real simulation can be increased by assigning specific production *cuts* to each of these detector parts. This allows a more detailed simulation to occur only in those regions where it is required.

The concept of detector *Region* is introduced to address this need. Once the final geometry setup of the detector has been defined, a region can be specified by constructing it with:

```
G4Region( const G4String& rName )
```

where:

rName	String identifier for the detector region

G4Regions must be allocated using 'new' in the user's program; they get registered to a G4RegionStore at construction, which will also take care to deallocate them at the end of the job, if not done already in the user's code.

A G4Region must then be assigned to a logical volume, in order to make it a *Root Logical Volume*:

```
G4Region* emCalorimeter = new G4Region("EM-Calorimeter");
emCalorimeterLV->SetRegion(emCalorimeter);
emCalorimeter->AddRootLogicalVolume(emCalorimeterLV);
```

A root logical volume is the first volume at the top of the hierarchy to which a given region is assigned. Once the region is assigned to the root logical volume, the information is automatically propagated to the volume tree, so that each daughter volume shares the same region. Propagation on a tree branch will be interrupted if an already existing root logical volume is encountered.

A specific *Production Cut* can be assigned to the region, by defining and assigning to it a G4ProductionCut object

```
emCalorimeter->SetProductionCuts(emCalCuts);
```

Set production threshold (SetCut methods) describes how to define a production cut. The same region can be assigned to more than one root logical volume, and root logical volumes can be removed from an existing region. A logical volume can have only *one* region assigned to it. Regions will be automatically registered in a store which will take care of destroying them at the end of the job. A default region with a default production cut is automatically created and assigned to the world volume.

Regions can also become 'envelopes' for fast-simulation; can be assigned user-limits or generic user-information (G4VUserRegionInformation); can be associated to specific stepping-actions (G4UserSteppingAction) or have assigned a local magnetic-field (local fields specifically associated to logical volumes take precedence anyhow).

4.1.4 Physical Volumes

Physical volumes represent the spatial positioning of the volumes describing the detector elements. Several techniques can be used. They range from the simple placement of a single copy to the repeated positioning using either a simple linear formula or a user specified function.

Any physical volume must be allocated using 'new' in the user's program; they get registered to a G4PhysicalVolumeStore at construction, which will also take care to deallocate them at the end of the job, if not done already in the user's code.

The simple placement involves the definition of a transformation matrix for the volume to be positioned. Repeated positioning is defined using the number of times a volume should be replicated at a given distance along a given direction. Finally it is possible to define a parameterised formula to specify the position of multiple copies of a volume. Details about these methods are given below.

Note: For geometries which vary between runs and for which components of the old geometry setup are explicitely -deleted-, it is required to consider the proper order of deletion (which is the exact inverse of the actual construction, i.e., first delete physical volumes and then logical volumes). Deleting a logical volume does NOT delete its daughter volumes.

It is not necessary to delete the geometry setup at the end of a job, the system will take care to free the volume and solid stores at the end of the job. The user has to take care of the deletion of any additional transformation or rotation matrices allocated dinamically in his/her own application.

Placements: single positioned copy

In this case, the Physical Volume is created by associating a Logical Volume with a Rotation Matrix and a Translation vector. The Rotation Matrix represents the rotation of the reference frame of the considered volume relatively to its mother volume's reference frame. The Translation Vector represents the translation of the current volume in the reference frame of its mother volume.

Transformations including reflections are not allowed.

To create a Placement one must construct it using:

```
G4PVPlacement(       G4RotationMatrix*  pRot,
                const G4ThreeVector&     tlate,
                      G4LogicalVolume*   pCurrentLogical,
                const G4String&          pName,
                      G4LogicalVolume*   pMotherLogical,
                      G4bool             pMany,
                      G4int              pCopyNo,
                      G4bool             pSurfChk=false )
```

where:

`pRot`	Rotation with respect to its mother volume
`tlate`	Translation with respect to its mother volume
`pCurrentLogical`	The associated Logical Volume
`pName`	String identifier for this placement
`pMotherLogical`	The associated mother volume
`pMany`	For future use. Can be set to false
`pCopyNo`	Integer which identifies this placement
`pSurfChk`	if true activates check for overlaps with existing volumes

Care must be taken because the rotation matrix is not copied by a `G4PVPlacement`. So the user must not modify it after creating a Placement that uses it. However the same rotation matrix can be re-used for many volumes.

Currently Boolean operations are not implemented at the level of physical volume. So `pMany` must be false. However, an alternative implementation of Boolean operations exists. In this approach a solid can be created from the union, intersection or subtraction of two solids. See *Solids made by Boolean operations* above for an explanation of this.

The mother volume must be specified for all volumes *except* the world volume.

An alternative way to specify a Placement utilizes a different method to place the volume. The solid itself is moved by rotating and translating it to bring it into the system of coordinates of the mother volume. If compared to the previous construct, the transformation in this case is generated by specifying the same translation with respect to its mother volume and the inverse of the rotation matrix. This *active* method can be utilized using the following constructor:

```
G4PVPlacement(       G4Transform3D      solidTransform,
                     G4LogicalVolume*   pCurrentLogical,
               const G4String&          pName,
                     G4LogicalVolume*   pMotherLogical,
                     G4bool             pMany,
                     G4int              pCopyNo,
                     G4bool             pSurfChk=false )
```

An alternative method to specify the mother volume is to specify its placed physical volume. It can be used in either of the above methods of specifying the placement's position and rotation. The effect will be exactly the same as for using the mother logical volume.

Note that a Placement Volume can still represent multiple detector elements. This can happen if several copies exist of the mother logical volume. Then different detector elements will belong to different branches of the tree of the hierarchy of geometrical volumes.

Repeated volumes

In this case, a single Physical Volume represents multiple copies of a volume within its mother volume, allowing to save memory. This is normally done when the volumes to be positioned follow a well defined rotational or translational symmetry along a Cartesian or cylindrical coordinate. The Repeated Volumes technique is available for most volumes described by CSG solids.

Replicas:

Replicas are *repeated volumes* in the case when the multiple copies of the volume are all identical. The coordinate axis and the number of replicas need to be specified for the program to compute at run time the transformation matrix corresponding to each copy.

```
G4PVReplica( const G4String&          pName,
                   G4LogicalVolume*   pCurrentLogical,
                   G4LogicalVolume*   pMotherLogical, // OR G4VPhysicalVolume*
             const EAxis              pAxis,
             const G4int              nReplicas,
             const G4double           width,
             const G4double           offset=0 )
```

where:

`pName`	String identifier for the replicated volume
`pCurrentLogical`	The associated Logical Volume
`pMotherLogical`	The associated mother volume
`pAxis`	The axis along with the replication is applied
`nReplicas`	The number of replicated volumes
`width`	The width of a single replica along the axis of replication
`offset`	Possible offset associated to mother offset along the axis of replication

`G4PVReplica` represents `nReplicas` volumes differing only in their positioning, and completely **filling** the containing mother volume. Consequently if a `G4PVReplica` is 'positioned' inside a given mother it **MUST** be the mother's only daughter volume. Replica's correspond to divisions or slices that completely fill the mother volume and have no offsets. For Cartesian axes, slices are considered perpendicular to the axis of replication.

The replica's positions are calculated by means of a linear formula. Replication may occur along:

- *Cartesian axes* `(kXAxis,kYAxis,kZAxis)`
  The replications, of specified width have coordinates of form `(-width*(nReplicas-1)*0.5+n*width,0,0)` where `n=0.. nReplicas-1` for the case of `kXAxis`, and are unrotated.

- *Radial axis (cylindrical polar) ‘‘(kRho)‘‘*
  The replications are cons/tubs sections, centred on the origin and are unrotated.
  They have radii of `width*n+offset` to `width*(n+1)+offset` where `n=0..nReplicas-1`
- *Phi axis (cylindrical polar) ‘‘(kPhi)‘‘*
  The replications are *phi sections* or *wedges*, and of cons/tubs form.
  They have `phi` of `offset+n*width` to `offset+(n+1)*width` where `n=0..nReplicas-1`

The coordinate system of the replicas is at the centre of each replica for the cartesian axis. For the radial case, the coordinate system is unchanged from the mother. For the `phi` axis, the new coordinate system is rotated such that the X axis bisects the angle made by each wedge, and Z remains parallel to the mother's Z axis.

The solid associated via the replicas' logical volume should have the dimensions of the first volume created and must be of the correct symmetry/type, in order to assist in good visualisation.

ex. For X axis replicas in a box, the solid should be another box with the dimensions of the replications. (same Y & Z dimensions as mother box, X dimension = mother's X dimension/nReplicas).

Replicas may be placed inside other replicas, provided the above rule is observed. Normal placement volumes may be placed inside replicas, provided that they do not intersect the mother's or any previous replica's boundaries. Parameterised volumes may not be placed inside.

Because of these rules, it is not possible to place any other volume inside a replication in `radius`.

The world volume *cannot* act as a replica, therefore it cannot be sliced.

During tracking, the translation + rotation associated with each `G4PVReplica` object is modified according to the currently 'active' replication. The solid is not modified and consequently has the wrong parameters for the cases of `phi` and `r` replication and for when the cross-section of the mother is not constant along the replication.

Example

Listing 4.2: An example of simple replicated volumes with `G4PVReplica`.

```
G4PVReplica repX("Linear Array",
                 pRepLogical,
                 pContainingMotherBox,
                 kXAxis, 5, 10*mm);

G4PVReplica repR("RSlices",
                 pRepRLogical,
                 pContainingMotherTub,
                 kRho, 5, 10*mm, 0);

G4PVReplica repZ("ZSlices",
                 pRepZLogical,
                 pContainingMotherTub,
                 kZAxis, 5, 10*mm);

G4PVReplica repPhi("PhiSlices",
                   pRepPhiLogical,
                   pContainingMotherTub,
                   kPhi, 4, M_PI*0.5*rad, 0);
```

`RepX` is an array of 5 replicas of width 10*mm, positioned inside and completely filling the volume pointed by `pContainingMotherBox`. The mother's X length must be 5*10*mm=50*mm (for example, if the mother's solid were a Box of half lengths [25,25,25] then the replica's solid must be a box of half lengths [25,25,5]).

If the containing mother's solid is a tube of radius 50*mm and half Z length of 25*mm, `RepR` divides the mother tube into 5 cylinders (hence the solid associated with `pRepRLogical` must be a tube of radius 10*mm, and half Z length 25*mm); `repZ` divides the tube into 5 shorter cylinders (the solid associated with `pRepZLogical` must be a tube of radius 10*mm, and half Z length 5*mm); finally, `repPhi` divides the tube into 4 tube segments with full angle of 90 degrees (the solid associated with `pRepPhiLogical` must be a tube segment of radius 10*mm, half Z length 5*mm and delta phi of M_PI*0.5*rad).

No further volumes may be placed inside these replicas. To do so would result in intersecting boundaries due to the `r` replications.

Parameterised Volumes:

Parameterised Volumes are *repeated volumes* in the case in which the multiple copies of a volume can be different in size, solid type, or material. The solid's type, its dimensions, the material and the transformation matrix can all be parameterised in function of the copy number, both when a strong symmetry exist and when it does not. The user implements the desired parameterisation function and the program computes and updates automatically at run time the information associated to the Physical Volume.

An example of creating a parameterised volume (by dimension and position) exists in basic example B2b. The implementation is provided in the two classes `B2bDetectorConstruction` and `B2bChamberParameterisation`.

To create a parameterised volume, one must first create its logical volume like `trackerChamberLV` below. Then one must create his own parameterisation class (*B2bChamberParameterisation*) and instantiate an object of this class (`chamberParam`). We will see how to create the parameterisation below.

Listing 4.3: An example of Parameterised volumes.

```
// Tracker segments

// An example of Parameterised volumes
// Dummy values for G4Tubs -- modified by parameterised volume

G4Tubs* chamberS
  = new G4Tubs("tracker",0, 100*cm, 100*cm, 0.*deg, 360.*deg);
fLogicChamber
  = new G4LogicalVolume(chamberS,fChamberMaterial,"Chamber",0,0,0);

G4double firstPosition = -trackerSize + chamberSpacing;
G4double firstLength   = trackerLength/10;
G4double lastLength    = trackerLength;

G4VPVParameterisation* chamberParam =
  new B2bChamberParameterisation(
                               NbOfChambers,   // NoChambers
                               firstPosition,  // Z of center of first
                               chamberSpacing, // Z spacing of centers
                               chamberWidth,   // chamber width
                               firstLength,    // initial length
                               lastLength);    // final length

// dummy value : kZAxis -- modified by parameterised volume

new G4PVParameterised("Chamber",       // their name
                      fLogicChamber,   // their logical volume
                      trackerLV,       // Mother logical volume
                      kZAxis,          // Are placed along this axis
                      NbOfChambers,    // Number of chambers
                      chamberParam,    // The parametrisation
                      fCheckOverlaps); // checking overlaps
```

The general constructor is:

```
G4PVParameterised( const G4String&              pName,
                         G4LogicalVolume*       pCurrentLogical,
                         G4LogicalVolume*       pMotherLogical, // OR G4VPhysicalVolume*
                   const EAxis                  pAxis,
                   const G4int                  nReplicas,
                         G4VPVParameterisation* pParam,
                         G4bool                 pSurfChk=false )
```

Note that for a parameterised volume the user must always specify a mother volume. So the world volume can *never*

be a parameterised volume, nor it can be sliced. The mother volume can be specified either as a physical or a logical volume.

`pAxis` specifies the tracking optimisation algorithm to apply: if a valid axis (the axis along which the parameterisation is performed) is specified, a simple one-dimensional voxelisation algorithm is applied; if "kUndefined" is specified instead, the default three-dimensional voxelisation algorithm applied for normal placements will be activated. In the latter case, more voxels will be generated, therefore a greater amount of memory will be consumed by the optimisation algorithm.

`pSurfChk` if `true` activates a check for overlaps with existing volumes or paramaterised instances.

The parameterisation mechanism associated to a parameterised volume is defined in the parameterisation class and its methods. Every parameterisation must create two methods:

- `ComputeTransformation` defines where one of the copies is placed,
- `ComputeDimensions` defines the size of one copy, and
- a constructor that initializes any member variables that are required.

An example is `B2bChamberParameterisation` that parameterises a series of tubes of different sizes

Listing 4.4: An example of Parameterised tubes of different sizes.

```
class B2bChamberParameterisation : public G4VPVParameterisation
{
  ...
  void ComputeTransformation(const G4int       copyNo,
                             G4VPhysicalVolume *physVol) const;

  void ComputeDimensions(G4Tubs&             trackerLayer,
                         const G4int            copyNo,
                         const G4VPhysicalVolume *physVol) const;
  ...
}
```

These methods works as follows:

The `ComputeTransformation` method is called with a copy number for the instance of the parameterisation under consideration. It must compute the transformation for this copy, and set the physical volume to utilize this transformation:

```
void B2bChamberParameterisation::ComputeTransformation
(const G4int copyNo, G4VPhysicalVolume *physVol) const
{
  // Note: copyNo will start with zero!
  G4double Zposition = fStartZ + copyNo * fSpacing;
  G4ThreeVector origin(0,0,Zposition);
  physVol->SetTranslation(origin);
  physVol->SetRotation(0);
}
```

Note that the translation and rotation given in this scheme are those for the frame of coordinates (the *passive* method). They are **not** for the *active* method, in which the solid is rotated into the mother frame of coordinates.

Similarly the `ComputeDimensions` method is used to set the size of that copy.

```
void B2bChamberParameterisation::ComputeDimensions
(G4Tubs& trackerChamber, const G4int copyNo, const G4VPhysicalVolume*) const
{
  // Note: copyNo will start with zero!
  G4double rmax = fRmaxFirst + copyNo * fRmaxIncr;
  trackerChamber.SetInnerRadius(0);
  trackerChamber.SetOuterRadius(rmax);
  trackerChamber.SetZHalfLength(fHalfWidth);
  trackerChamber.SetStartPhiAngle(0.*deg);
```

```
  trackerChamber.SetDeltaPhiAngle(360.*deg);
}
```

The user must ensure that the type of the first argument of this method (in this example `G4Tubs &`) corresponds to the type of object the user give to the logical volume of parameterised physical volume.

More advanced usage allows the user:

- to change the type of solid by creating a `ComputeSolid` method, or
- to change the material of the volume by creating a `ComputeMaterial` method. This method can also utilise information from a parent or other ancestor volume (see the Nested Parameterisation below.)

for the parameterisation.

Example `examples/extended/runAndEvent/RE02` shows a simple parameterisation by material. A more complex example is provided in `examples/extended/medical/DICOM`, where a phantom grid of cells is built using a parameterisation by material defined through a map.

Note: Currently for many cases it is not possible to add daughter volumes to a parameterised volume. Only parameterised volumes all of whose solids have the same size are allowed to contain daughter volumes. When the size or type of solid varies, adding daughters is not supported. So the full power of parameterised volumes can be used only for "leaf" volumes, which contain no other volumes.

Note: A hierarchy of volumes included in a parameterised volume cannot vary. Therefore, it is not possible to implement a parameterisation which can modify the hierachy of volumes included inside a specific parameterised copy.

Note: For parameterisations of tubes or cons, where the starting `Phi` and its `DeltaPhi` angles vary, it is possible to optimise the regeneration of the trigonometric parameters of the shape, by invoking `SetStartPhiAngle(newPhi, false); SetDeltaPhiAngle (newDPhi)`, i.e. by specifying with `false` flag to skip the computation of the parameters which will be later on properly initialised with the call for `DeltaPhi`.

Note: For multi-threaded applications, one must be careful in the implementation of the parameterisation functions for the geometrical objects being created in the parameterisation. In particular, when parameterising by the type of a solid, it is assumed that the solids being parameterised are being declared thread-local in the user's parameterisation class and allocated just once.

Advanced parameterisations for 'nested' parameterised volumes

A different type of parameterisation enables a user to have the daughter's material also depend on the copy number of the parent when a parameterised volume (daughter) is located inside another (parent) repeated volume. The parent volume can be a replica, a parameterised volume, or a division if the key feature of modifying its contents is utilised. (Note: a 'nested' parameterisation inside a placement volume is not supported, because all copies of a placement volume must be identical at all levels.)

In such a " nested" parameterisation , the user must provide a `ComputeMaterial` method that utilises the new argument that represents the touchable history of the parent volume:

```
// Sample Parameterisation
class SampleNestedParameterisation : public G4VNestedParameterisation
{
```

```
 public:
   // .. other methods ...
    // Mandatory method, required and reason for this class
    virtual G4Material* ComputeMaterial(G4VPhysicalVolume *currentVol,
                                        const G4int no_lev,
                                        const G4VTouchable *parentTouch);
 private:
    G4Material *material1, *material2;
};
```

The implementation of the method can utilise any information from a parent or other ancestor volume of its parameterised physical volume, but typically it will use only the copy number:

```
G4Material*
SampleNestedParameterisation::ComputeMaterial(G4VPhysicalVolume *currentVol,
                                              const G4int no_lev,
                                              const G4VTouchable *parentTouchable)
{
   G4Material *material=0;

   // Get the information about the parent volume
   G4int no_parent= parentTouchable->GetReplicaNumber();
   G4int no_total= no_parent + no_lev;
   // A simple 'checkerboard' pattern of two materials
   if( no_total / 2 == 1 ) material= material1;
   else  material= material2;
   // Set the material to the current logical volume
   G4LogicalVolume* currentLogVol= currentVol->GetLogicalVolume();
   currentLogVol->SetMaterial( material );
   return material;
}
```

Nested parameterisations are suitable for the case of regular, 'voxel' geometries in which a large number of 'equal' volumes are required, and their only difference is in their material. By creating two (or more) levels of parameterised physical volumes it is possible to divide space, while requiring only limited additional memory for very fine-level optimisation. This provides fast navigation. Alternative implementations, taking into account the regular structure of such geometries in navigation are under study.

Divisions of Volumes

Divisions in Geant4 are implemented as a specialized type of parameterised volumes.

They serve to divide a volume into identical copies along one of its axes, providing the possibility to define an *offset*, and without the limitation that the daugthers have to fill the mother volume as it is the case for the replicas. In the case, for example, of a tube divided along its radial axis, the copies are not strictly identical, but have increasing radii, although their widths are constant.

To divide a volume it will be necessary to provide:

1. the axis of division, and
2. either
   - the number of divisions (so that the width of each division will be automatically calculated), or
   - the division width (so that the number of divisions will be automatically calculated to fill as much of the mother as possible), or
   - both the number of divisions and the division width (this is especially designed for the case where the copies do not fully fill the mother).

An *offset* can be defined so that the first copy will start at some distance from the mother wall. The dividing copies will be then distributed to occupy the rest of the volume.

There are three constructors, corresponding to the three input possibilities described above:

- Giving only the number of divisions:

```
G4PVDivision( const G4String& pName,
                    G4LogicalVolume* pCurrentLogical,
                    G4LogicalVolume* pMotherLogical,
              const EAxis pAxis,
              const G4int nDivisions,
              const G4double offset )
```

- Giving only the division width:

```
G4PVDivision( const G4String& pName,
                    G4LogicalVolume* pCurrentLogical,
                    G4LogicalVolume* pMotherLogical,
              const EAxis pAxis,
              const G4double width,
              const G4double offset )
```

- Giving the number of divisions and the division width:

```
G4PVDivision( const G4String& pName,
                    G4LogicalVolume* pCurrentLogical,
                    G4LogicalVolume* pMotherLogical,
              const EAxis pAxis,
              const G4int nDivisions,
              const G4double width,
              const G4double offset )
```

where:

`pName`	String identifier for the replicated volume
`pCurrentLogical`	The associated Logical Volume
`pMotherLogical`	The associated mother Logical Volume
`pAxis`	The axis along which the division is applied
`nDivisions`	The number of divisions
`width`	The width of a single division along the axis
`offset`	Possible offset associated to the mother along the axis of division

The parameterisation is calculated automatically using the values provided in input. Therefore the dimensions of the solid associated with `pCurrentLogical` will not be used, but recomputed through the `G4VParameterisation::ComputeDimension()` method.

Since `G4VPVParameterisation` may have different `ComputeDimension()` methods for each solid type, the user must provide a solid that is of the same type as of the one associated to the mother volume.

As for any replica, the coordinate system of the divisions is related to the centre of each division for the cartesian axis. For the radial axis, the coordinate system is the same of the mother volume. For the phi axis, the new coordinate system is rotated such that the X axis bisects the angle made by each wedge, and Z remains parallel to the mother's Z axis.

As divisions are parameterised volumes with constant dimensions, they may be placed inside other divisions, except in the case of divisions along the radial axis.

It is also possible to place other volumes inside a volume where a division is placed.

The list of volumes that currently support divisioning and the possible division axis are summarised below:

G4Box	kXAxis, kYAxis, kZAxis
G4Tubs	kRho, kPhi, kZAxis
G4Cons	kRho, kPhi, kZAxis
G4Trd	kXAxis, kYAxis, kZAxis
G4Para	kXAxis, kYAxis, kZAxis
G4Polycone	kRho, kPhi, kZAxis
G4Polyhedra	kRho, kPhi, kZAxis (*)

(*) - G4Polyhedra:

- kPhi - the number of divisions has to be the same as solid sides, (i.e. numSides), the width will *not* be taken into account.

In the case of division along kRho of G4Cons, G4Polycone, G4Polyhedra, if width is provided, it is taken as the width at the -Z radius; the width at other radii will be scaled to this one.

Examples are given below in listings *Example-parameterised* and *Example-Chamber*.

Listing 4.5: An example of a box division along different axes, with or without offset.

```
G4Box* motherSolid = new G4Box("motherSolid", 0.5*m, 0.5*m, 0.5*m);
G4LogicalVolume* motherLog = new G4LogicalVolume(motherSolid, material, "mother",0,0,0);
G4Para* divSolid = new G4Para("divSolid", 0.512*m, 1.21*m, 1.43*m);
G4LogicalVolume* childLog = new G4LogicalVolume(divSolid, material, "child",0,0,0);

G4PVDivision divBox1("division along X giving nDiv",
                     childLog, motherLog, kXAxis, 5, 0.);

G4PVDivision divBox2("division along X giving width and offset",
                     childLog, motherLog, kXAxis, 0.1*m, 0.45*m);

G4PVDivision divBox3("division along X giving nDiv, width and offset",
                     childLog, motherLog, kXAxis, 3, 0.1*m, 0.5*m);
```

- divBox1 is a division of a box along its X axis in 5 equal copies. Each copy will have a dimension in meters of [0.2, 1., 1.].
- divBox2 is a division of the same box along its X axis with a width of 0.1 meters and an offset of 0.5 meters. As the mother dimension along X of 1 meter (0.5*m of halflength), the division will be sized in total 1 - 0.45 = 0.55 meters. Therefore, there's space for 5 copies, the first extending from -0.05 to 0.05 meters in the mother's frame and the last from 0.35 to 0.45 meters.
- divBox3 is a division of the same box along its X axis in 3 equal copies of width 0.1 meters and an offset of 0.5 meters. The first copy will extend from 0. to 0.1 meters in the mother's frame and the last from 0.2 to 0.3 meters.

Listing 4.6: An example of division of a polycone.

```
G4double* zPlanem = new G4double[3];
          zPlanem[0]= -1.*m;
          zPlanem[1]= -0.25*m;
          zPlanem[2]=  1.*m;
G4double* rInnerm = new G4double[3];
          rInnerm[0]=0.;
          rInnerm[1]=0.1*m;
          rInnerm[2]=0.5*m;
G4double* rOuterm  = new G4double[3];
          rOuterm[0]=0.2*m;
          rOuterm[1]=0.4*m;
          rOuterm[2]=1.*m;
G4Polycone* motherSolid = new G4Polycone("motherSolid", 20.*deg, 180.*deg,
                                         3, zPlanem, rInnerm, rOuterm);
G4LogicalVolume* motherLog = new G4LogicalVolume(motherSolid, material, "mother",0,0,0);
```

```
G4double* zPlaned = new G4double[3];
          zPlaned[0]= -3.*m;
          zPlaned[1]= -0.*m;
          zPlaned[2]=  1.*m;
G4double* rInnerd = new G4double[3];
          rInnerd[0]=0.2;
          rInnerd[1]=0.4*m;
          rInnerd[2]=0.5*m;
G4double* rOuterd  = new G4double[3];
          rOuterd[0]=0.5*m;
          rOuterd[1]=0.8*m;
          rOuterd[2]=2.*m;
G4Polycone* divSolid = new G4Polycone("divSolid", 0.*deg, 10.*deg,
                                      3, zPlaned, rInnerd, rOuterd);
G4LogicalVolume* childLog = new G4LogicalVolume(divSolid, material, "child",0,0,0);

G4PVDivision divPconePhiW("division along phi giving width and offset",
                          childLog, motherLog, kPhi, 30.*deg, 60.*deg);

G4PVDivision divPconeZN("division along Z giving nDiv and offset",
                           childLog, motherLog, kZAxis, 2, 0.1*m);
```

- `divPconePhiW` is a division of a polycone along its `phi` axis in equal copies of width 30 degrees with an offset of 60 degrees. As the mother extends from 0 to 180 degrees, there's space for 4 copies. All the copies have a starting angle of 20 degrees (as for the mother) and a `phi` extension of 30 degrees. They are rotated around the `Z` axis by 60 and 30 degrees, so that the first copy will extend from 80 to 110 and the last from 170 to 200 degrees.
- `divPconeZN` is a division of the same polycone along its `Z` axis. As the mother polycone has two sections, it will be divided in two one-section polycones, the first one extending from -1 to -0.25 meters, the second from -0.25 to 1 meters. Although specified, the offset will not be used.

Note: Divisions for polycone and polyhedra are *NOT* possible in a multi-threaded application.

4.1.5 Touchables: Uniquely Identifying a Volume

Introduction to Touchables

A *touchable* for a volume serves the purpose of providing a unique identification for a detector element. This can be useful for description of the geometry alternative to the one used by the Geant4 tracking system, such as a Sensitive Detectors based read-out geometry, or a parameterised geometry for fast Monte Carlo. In order to create a *touchable volume*, several techniques can be implemented: for example, in Geant4 touchables are implemented as solids associated to a transformation-matrix in the global reference system, or as a hierarchy of physical volumes up to the root of the geometrical tree.

A touchable is a geometrical entity (volume or solid) which has a unique placement in a detector description. It is represented by an abstract base class which can be implemented in a variety of ways. Each way must provide the capabilities of obtaining the transformation and solid that is described by the touchable.

What can a Touchable do?

All `G4VTouchable` implementations must respond to the two following "requests", where in all cases, by `depth` it is meant the number of levels *up* in the tree to be considered (the default and current one is `0`):

1. `GetTranslation(depth)`
2. `GetRotation(depth)`

that return the components of the volume's transformation.

Additional capabilities are available from implementations with more information. These have a default implementation that causes an exception.

Several capabilities are available from touchables with physical volumes:

1. `GetSolid(depth)` gives the solid associated to the touchable.
2. `GetVolume(depth)` gives the physical volume.
3. `GetReplicaNumber(depth)` or `GetCopyNumber(depth)` which return the copy number of the physical volume (replicated or not).

Touchables that store volume hierarchy (history) have the whole stack of parent volumes available. Thus it is possible to add a little more state in order to extend its functionality. We add a "pointer" to a level and a member function to move the level in this stack. Then calling the above member functions for another level the information for that level can be retrieved.

The top of the history tree is, by convention, the world volume.

1. `GetHistoryDepth()` gives the depth of the history tree.
2. `MoveUpHistory(num)` moves the current pointer inside the touchable to point `num` levels up the history tree. Thus, e.g., calling it with `num=1` will cause the internal pointer to move to the mother of the current volume.

> **Warning:** this function changes the state of the touchable and can cause errors in tracking if applied to Pre/Post step touchables.

These methods are valid only for the *touchable-history* type, as specified also below.

An update method, with different arguments is available, so that the information in a touchable can be updated:

1. `UpdateYourself(vol, history)` takes a physical volume pointer and can additionally take a `NavigationHistory` pointer.

Touchable history holds stack of geometry data

As shown in Sections *Logical Volumes* and *Physical Volumes*, a logical volume represents unpositioned detector elements, and a physical volume can represent multiple detector elements. On the other hand, touchables provide a unique identification for a detector element. In particular, the Geant4 transportation process and the tracking system exploit touchables as implemented in `G4TouchableHistory`. The touchable history is the minimal set of information required to specify the full genealogy of a given physical volume (up to the root of the geometrical tree). These touchable volumes are made available to the user at every step of the Geant4 tracking in `G4VUserSteppingAction`.

To create/access a `G4TouchableHistory` the user must message `G4Navigator` which provides the method `CreateTouchableHistoryHandle()`:

```
G4TouchableHistoryHandle CreateTouchableHistoryHandle() const;
```

this will return a handle to the touchable.

The methods that differentiate the touchable-history from other touchables (since they have meaning only for this type...), are:

```
G4int GetHistoryDepth()  const;
G4int MoveUpHistory( G4int num_levels = 1 );
```

The first method is used to find out how many levels deep in the geometry tree the current volume is. The second method asks the touchable to eliminate its deepest level.

As mentioned above, `MoveUpHistory(num)` significantly modifies the state of a touchable.

4.1.6 Creating an Assembly of Volumes

`G4AssemblyVolume` is a helper class which allows several logical volumes to be combined together in an arbitrary way in 3D space. The result is a placement of a normal logical volume, but where final physical volumes are many.

However, an *assembly* volume does not act as a real mother volume, being an envelope for its daughter volumes. Its role is over at the time the placement of the logical assembly volume is done. The physical volume objects become independent copies of each of the assembled logical volumes.

This class is particularly useful when there is a need to create a regular pattern in space of a complex component which consists of different shapes and can't be obtained by using replicated volumes or parametrised volumes (see also *Assembly of volumes*. Careful usage of `G4AssemblyVolume` must be considered though, in order to avoid cases of "proliferation" of physical volumes all placed in the same mother.

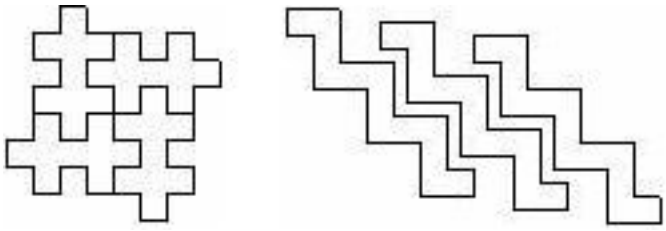


Fig. 4.2: Examples of *assembly* of volumes.

Filling an assembly volume with its "daughters"

Participating logical volumes are represented as a triplet of <logical volume, translation, rotation> (`G4AssemblyTriplet` class).

The adopted approach is to place each participating logical volume with respect to the assembly's coordinate system, according to the specified translation and rotation.

Assembly volume placement

An assembly volume object is composed of a set of logical volumes; imprints of it can be made inside a mother logical volume.

Since the assembly volume class generates physical volumes during each imprint, the user has no way to specify identifiers for these. An internal counting mechanism is used to compose uniquely the names of the physical volumes created by the invoked `MakeImprint(...)` method(s).

The name for each of the physical volume is generated with the following format:

```
av_WWW_impr_XXX_YYY_ZZZ
```

where:

- WWW - assembly volume instance number
- XXX - assembly volume imprint number
- YYY - the name of the placed logical volume
- ZZZ - the logical volume index inside the assembly volume

It is however possible to access the constituent physical volumes of an assembly and eventually customise ID and copy-number.

Destruction of an assembly volume

At destruction all the generated physical volumes and associated rotation matrices of the imprints will be destroyed. A list of physical volumes created by `MakeImprint()` method is kept, in order to be able to cleanup the objects when not needed anymore. This requires the user to keep the assembly objects in memory during the whole job or during the life-time of the `G4Navigator`, logical volume store and physical volume store may keep pointers to physical volumes generated by the assembly volume.

The `MakeImprint()` method will operate correctly also on transformations including reflections and can be applied also to recursive assemblies (i.e., it is possible to generate imprints of assemblies including other assemblies). Giving `true` as the last argument of the `MakeImprint()` method, it is possible to activate the volumes overlap check for the assembly's constituents (the default is `false`).

At destruction of a `G4AssemblyVolume`, all its generated physical volumes and rotation matrices will be freed.

Example

This example shows how to use the `G4AssemblyVolume` class. It implements a layered detector where each layer consists of 4 plates.

In the code below, at first the world volume is defined, then solid and logical volume for the plate are created, followed by the definition of the assembly volume for the layer.

The assembly volume for the layer is then filled by the plates in the same way as normal physical volumes are placed inside a mother volume.

Finally the layers are placed inside the world volume as the imprints of the assembly volume (see *Example*).

Listing 4.7: An example of usage of the `G4AssemblyVolume` class.

```
static unsigned int layers = 5;

void TstVADetectorConstruction::ConstructAssembly()
{
  // Define world volume
  G4Box* WorldBox = new G4Box( "WBox", worldX/2., worldY/2., worldZ/2. );
  G4LogicalVolume*   worldLV  = new G4LogicalVolume( WorldBox, selectedMaterial, "WLog", 0, 0,␣
↪0);
  G4VPhysicalVolume* worldVol = new G4PVPlacement(0, G4ThreeVector(), "WPhys",worldLV,
                                                  0, false, 0);

  // Define a plate
  G4Box* PlateBox = new G4Box( "PlateBox", plateX/2., plateY/2., plateZ/2. );
  G4LogicalVolume* plateLV = new G4LogicalVolume( PlateBox, Pb, "PlateLV", 0, 0, 0 );

  // Define one layer as one assembly volume
  G4AssemblyVolume* assemblyDetector = new G4AssemblyVolume();

  // Rotation and translation of a plate inside the assembly
  G4RotationMatrix Ra;
  G4ThreeVector Ta;
  G4Transform3D Tr;

  // Rotation of the assembly inside the world
  G4RotationMatrix Rm;

  // Fill the assembly by the plates
  Ta.setX( caloX/4. ); Ta.setY( caloY/4. ); Ta.setZ( 0. );
  Tr = G4Transform3D(Ra,Ta);
  assemblyDetector->AddPlacedVolume( plateLV, Tr );

  Ta.setX( -1*caloX/4. ); Ta.setY( caloY/4. ); Ta.setZ( 0. );
  Tr = G4Transform3D(Ra,Ta);
```

```
  assemblyDetector->AddPlacedVolume( plateLV, Tr );

  Ta.setX( -1*caloX/4. ); Ta.setY( -1*caloY/4. ); Ta.setZ( 0. );
  Tr = G4Transform3D(Ra,Ta);
  assemblyDetector->AddPlacedVolume( plateLV, Tr );

  Ta.setX( caloX/4. ); Ta.setY( -1*caloY/4. ); Ta.setZ( 0. );
  Tr = G4Transform3D(Ra,Ta);
  assemblyDetector->AddPlacedVolume( plateLV, Tr );

  // Now instantiate the layers
  for( unsigned int i = 0; i < layers; i++ )
  {
    // Translation of the assembly inside the world
    G4ThreeVector Tm( 0,0,i*(caloZ + caloCaloOffset) - firstCaloPos );
    Tr = G4Transform3D(Rm,Tm);
    assemblyDetector->MakeImprint( worldLV, Tr );
  }
}
```

The resulting detector will look as in *Assembly*, below:

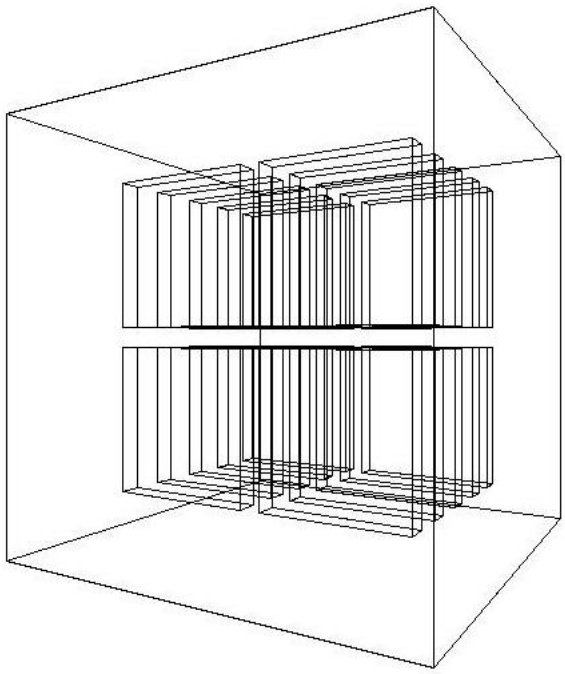


Fig. 4.3: The geometry corresponding to the *example*.

4.1.7 Reflecting Hierarchies of Volumes

Hierarchies of volumes based on *CSG* or *specific* solids can be reflected by means of the `G4ReflectionFactory` class and `G4ReflectedSolid`, which implements a solid that has been shifted from its original reference frame to a new 'reflected' one. The reflection transformation is applied as a decomposition into rotation and translation transformations.

The factory is a singleton object which provides the following methods:

```
G4PhysicalVolumesPair Place(const G4Transform3D&   transform3D,
                            const G4String&        name,
                                  G4LogicalVolume* LV,
                                  G4LogicalVolume* motherLV,
                                  G4bool           isMany,
                                  G4int            copyNo,
                                  G4bool           surfCheck=false)

G4PhysicalVolumesPair Replicate(const G4String&        name,
                                      G4LogicalVolume* LV,
                                      G4LogicalVolume* motherLV,
                                      EAxis            axis,
                                      G4int            nofReplicas,
                                      G4double         width,
                                      G4double         offset=0)

G4PhysicalVolumesPair Divide(const G4String&        name,
                                   G4LogicalVolume* LV,
                                   G4LogicalVolume* motherLV,
                                   EAxis            axis,
                                   G4int            nofDivisions,
                                   G4double         width,
                                   G4double         offset);
```

The method `Place()` used for placements, evaluates the passed transformation. In case the transformation contains a reflection, the factory will act as follows:

1. Performs the transformation decomposition.
2. Creates a new reflected solid and logical volume, or retrieves them from a map if the reflected object was already created.
3. Transforms the daughters (if any) and place them in the given mother.

If successful, the result is a pair of physical volumes, where the second physical volume is a placement in a reflected mother. Optionally, it is also possible to force the overlaps check at the time of placement, by activating the `surfCheck` flag.

The method `Replicate()` creates replicas in the given mother. If successful, the result is a pair of physical volumes, where the second physical volume is a replica in a reflected mother.

The method `Divide()` creates divisions in the given mother. If successful, the result is a pair of physical volumes, where the second physical volume is a division in a reflected mother. There exists also two more variants of this method which may specify or not width or number of divisions.

Note: In order to reflect hierarchies containing divided volumes, it is necessary to explicitely instantiate a concrete *division* factory -before- applying the actual reflection: (i.e. - `G4PVDivisionFactory::GetInstance();`).

Note: Reflection of generic parameterised volumes is not possible yet.

Listing 4.8: An example of usage of the G4ReflectionFactory class.

```
#include "G4ReflectionFactory.hh"

// Calor placement with rotation

G4double calThickness = 100*cm;
G4double Xpos = calThickness*1.5;
G4RotationMatrix* rotD3 = new G4RotationMatrix();
rotD3->rotateY(10.*deg);
```

```
G4VPhysicalVolume* physiCalor =
    new G4PVPlacement(rotD3,                    // rotation
                      G4ThreeVector(Xpos,0.,0.), // at (Xpos,0,0)
                      logicCalor,     // its logical volume (defined elsewhere)
                      "Calorimeter",  // its name
                      logicHall,      // its mother volume (defined elsewhere)
                      false,          // no boolean operation
                      0);             // copy number

// Calor reflection with rotation
//
G4Translate3D translation(-Xpos, 0., 0.);
G4Transform3D rotation = G4Rotate3D(*rotD3);
G4ReflectX3D  reflection;
G4Transform3D transform = translation*rotation*reflection;

G4ReflectionFactory::Instance()
         ->Place(transform,     // the transformation with reflection
                 "Calorimeter", // the actual name
                 logicCalor,    // the logical volume
                 logicHall,     // the mother volume
                 false,         // no boolean operation
                 1,             // copy number
                 false);        // no overlap check triggered

// Replicate layers
//
G4ReflectionFactory::Instance()
         ->Replicate("Layer",    // layer name
                     logicLayer, // layer logical volume (defined elsewhere)
                     logicCalor, // its mother
                     kXAxis,     // axis of replication
                     5,          // number of replica
                     20*cm);     // width of replica
```

4.1.8 The Geometry Navigator

Navigation through the geometry at tracking time is implemented by the class `G4Navigator`. The navigator is used to locate points in the geometry and compute distances to geometry boundaries. At tracking time, the navigator is intended to be the only point of interaction with tracking.

Internally, the G4Navigator has several private helper/utility classes:

- **G4NavigationHistory** - stores the compounded transformations, replication/parameterisation information, and volume pointers at each level of the hierarchy to the current location. The volume types at each level are also stored - whether normal (placement), replicated or parameterised.
- **G4NormalNavigation** - provides location & distance computation functions for geometries containing 'placement' volumes, with no voxels.
- **G4VoxelNavigation** - provides location and distance computation functions for geometries containing 'placement' physical volumes with voxels. Internally a stack of voxel information is maintained. Private functions allow for isotropic distance computation to voxel boundaries and for computation of the 'next voxel' in a specified direction.
- **G4ParameterisedNavigation** - provides location and distance computation functions for geometries containing parameterised volumes with voxels. Voxel information is maintained similarly to `G4VoxelNavigation`, but computation can also be simpler by adopting voxels to be one level deep only (*unrefined*, or 1D optimisation)
- **G4ReplicaNavigation** - provides location and distance computation functions for replicated volumes.

In addition, the navigator maintains a set of flags for exiting/entry optimisation. A navigator is not a singleton class; this is mainly to allow a design extension in future (e.g geometrical event biasing).

Navigation and Tracking

The main functions required for tracking in the geometry are described below. Additional functions are provided to return the net transformation of volumes and for the creation of touchables. None of the functions implicitly requires that the geometry be described hierarchically.

- **SetWorldVolume()**
  Sets the first volume in the hierarchy. It must be unrotated and untranslated from the origin.
- **LocateGlobalPointAndSetup()**
  Locates the volume containing the specified global point. This involves a traverse of the hierarchy, requiring the computation of compound transformations, testing replicated and parameterised volumes (etc). To improve efficiency this search may be performed relative to the last, and this is the recommended way of calling the function. A 'relative' search may be used for the first call of the function which will result in the search defaulting to a search from the root node of the hierarchy. Searches may also be performed using a `G4TouchableHistory`.
- **LocateGlobalPointAndUpdateTouchableHandle()**
  First, search the geometrical hierarchy like the above method `LocateGlobalPointAndSetup()`. Then use the volume found and its navigation history to update the touchable.
- **ComputeStep()**
  Computes the distance to the next boundary intersected along the specified unit direction from a specified point. The point must be have been located prior to calling `ComputeStep()`.
  When calling `ComputeStep()`, a proposed physics step is passed. If it can be determined that the first intersection lies at or beyond that distance then `kInfinity` is returned. In any case, if the returned step is greater than the physics step, the physics step must be taken.
- **SetGeometricallyLimitedStep()**
  Informs the navigator that the last computed step was taken in its entirety. This enables entering/exiting optimisation, and should be called prior to calling `LocateGlobalPointAndSetup()`.
- **CreateTouchableHistory()**
  Creates a `G4TouchableHistory` object, for which the caller has deletion responsibility. The 'touchable' volume is the volume returned by the last Locate operation. The object includes a copy of the current NavigationHistory, enabling the efficient relocation of points in/close to the current volume in the hierarchy.

As stated previously, the navigator makes use of utility classes to perform location and step computation functions. The different navigation utilities manipulate the `G4NavigationHistory` object.

In `LocateGlobalPointAndSetup()` the process of locating a point breaks down into three main stages - optimisation, determination that the point is contained with a subtree (mother and daughters), and determination of the actual containing daughter. The latter two can be thought of as scanning first 'up' the hierarchy until a volume that is guaranteed to contain the point is found, and then scanning 'down' until the actual volume that contains the point is found.

In `ComputeStep()` three types of computation are treated depending on the current containing volume:

- The volume contains normal (placement) daughters (or none)
- The volume contains a single parameterised volume object, representing many volumes
- The volume is a replica and contains normal (placement) daughters

Using the navigator to locate points

More than one navigator object can be created inside an application; these navigators can act independently for different purposes. The main navigator which is *activated* automatically at the startup of a simulation program is the navigator used for the *tracking* and attached the world volume of the main tracking (or *mass*) geometry.

The navigator for tracking can be retrieved at any state of the application by messagging the `G4TransportationManager`:

```
G4Navigator* tracking_navigator =
  G4TransportationManager::GetInstance()->GetNavigatorForTracking();
```

This also allows to retrieve at any time a pointer to the world volume assigned for tracking:

```
G4VPhysicalVolume* tracking_world = tracking_navigator->GetWorldVolume();
```

The navigator for tracking also retains all the information of the current history of volumes transversed at a precise moment of the tracking during a run. Therefore, if the navigator for tracking is used during tracking for locating a generic point in the tree of volumes, the actual particle gets also -relocated- in the specified position and tracking will be of course affected !

In order to avoid the problem above and provide information about location of a point without affecting the tracking, it is suggested to either use an alternative `G4Navigator` object (which can then be assigned to the world-volume), or access the information through the step.

If the user instantiates an alternative `G4Navigator`, ownership is retained by the user's code, and the navigator object should be deleted by that code.

Using the 'step' to retrieve geometrical information

During the tracking run, geometrical information can be retrieved through the touchable handle associated to the current step. For example, to identify the exact copy-number of a specific physical volume in the mass geometry, one should do the following:

```
// Given the pointer to the step object ...
//
G4Step* aStep = ..;

// ... retrieve the 'pre-step' point
//
G4StepPoint* preStepPoint = aStep->GetPreStepPoint();

// ... retrieve a touchable handle and access to the information
//
G4TouchableHandle theTouchable = preStepPoint->GetTouchableHandle();
G4int copyNo = theTouchable->GetCopyNumber();
G4int motherCopyNo = theTouchable->GetCopyNumber(1);
```

To determine the exact position in global coordinates in the mass geometry and convert to local coordinates (local to the current volume):

```
G4ThreeVector worldPosition = preStepPoint->GetPosition();
G4ThreeVector localPosition = theTouchable->GetHistory()->
              GetTopTransform().TransformPoint(worldPosition);
```

Using an alternative navigator to locate points

In order to know (when in the `idle` state of the application) in which physical volume a given point is located in the detector geometry, it is necessary to create an alternative navigator object first and assign it to the world volume:

```
G4Navigator* aNavigator = new G4Navigator();
aNavigator->SetWorldVolume(worldVolumePointer);
```

Then, locate the point `myPoint` (defined in global coordinates), retrieve a *touchable handle* and do whatever you need with it:

```
aNavigator->LocateGlobalPointAndSetup(myPoint);
G4TouchableHistoryHandle aTouchable =
     aNavigator->CreateTouchableHistoryHandle();

   // Do whatever you need with it ...
```

```
// ... convert point in local coordinates (local to the current volume)
//
G4ThreeVector localPosition = aTouchable->GetHistory()->
             GetTopTransform().TransformPoint(myPoint);

// ... convert back to global coordinates system
G4ThreeVector globalPosition = aTouchable->GetHistory()->
             GetTopTransform().Inverse().TransformPoint(localPosition);
```

If outside of the tracking run and given a generic local position (local to a given volume in the geometry tree), it is -not- possible to determine a priori its global position and convert it to the global coordinates system. The reason for this is rather simple, nobody can guarantee that the given (local) point is located in the right -copy- of the physical volume ! In order to retrieve this information, some extra knowledge related to the absolute position of the physical volume is required first, i.e. one should first determine a global point belonging to that volume, eventually making a dedicated scan of the geometry tree through a dedicated `G4Navigator` object and then apply the method above after having created the touchable for it.

Navigation in parallel geometries

Since release 8.2 of Geant4, it is possible to define geometry trees which are `parallel` to the tracking geometry and having them assigned to navigator objects that transparently communicate in sync with the normal tracking geometry.

Parallel geometries can be defined for several uses (fast shower parameterisation, geometrical biasing, particle scoring, readout geometries, etc ...) and can *overlap* with the mass geometry defined for the tracking. The `parallel` transportation will be activated only after the registration of the parallel geometry in the detector description setup; see Section *Parallel Geometries* for how to define a parallel geometry and register it to the run-manager.

The `G4TransportationManager` provides all the utilities to verify, retrieve and activate the navigators associated to the various parallel geometries defined.

Fast navigation in regular patterned geometries and phantoms

Since release 9.1 of Geant4, a specialised navigation algorithm has been introduced to allow for optimal memory use and extremely efficient navigation in geometries represented by a regular pattern of volumes and particularly three-dimensional grids of boxes. A typical application of this kind is the case of DICOM phantoms for medical physics studies.

The class `G4RegularNavigation` is used and automatically activated when such geometries are defined. It is required to the user to implement a parameterisation of the kind `G4PhantomParameterisation` and place the parameterised volume containing it in a container volume, so that all cells in the three-dimensional grid (*voxels*) completely fill the container volume. This way the location of a point inside a voxel can be done in a fast way, transforming the position to the coordinate system of the container volume and doing a simple calculation of the kind:

```
copyNo_x = (localPoint.x()+fVoxelHalfX*fNoVoxelX)/(fVoxelHalfX*2.)
```

where `fVoxelHalfX` is the half dimension of the voxel along `X` and `fNoVoxelX` is the number of voxels in the `X` dimension. Voxel `0` will be the one closest to the corner `(fVoxelHalfX*fNoVoxelX, fVoxelHalfY*fNoVoxelY, fVoxelHalfZ*fNoVoxelZ)`.

Having the voxels filling completely the container volume allows to avoid the lengthy computation of `ComputeStep()` and `ComputeSafety` methods required in the traditional navigation algorithm. In this case, when a track is inside the parent volume, it has always to be inside one of the voxels and it will be only necessary to calculate the distance to the walls of the current voxel.

Skipping borders of voxels with same material

Another speed optimisation can be provided by skipping the frontiers of two voxels which the same material assigned, so that bigger steps can be done. This optimisation may be not very useful when the number of materials is very big (in which case the probability of having contiguous voxels with same material is reduced), or when the physical step is small compared to the voxel dimensions (very often the case of electrons). The optimisation can be switched off in such cases, by invoking the following method with argument `skip = 0`:

Phantoms with only one material

If you want to describe a phantom of a unique material, you may spare some memory by not filling the set of indices of materials of each voxel. If the method `SetMaterialIndices()` is not invoked, the index for all voxels will be 0, that is the first (and unique) material in your list.

```
G4RegularParameterisation::SetSkipEqualMaterials( G4bool skip );
```

Example

To use the specialised navigation, it is required to first create an object of type `G4PhantomParameterisation`:

```
G4PhantomParameterisation* param = new G4PhantomParameterisation();
```

Then, fill it with the all the necessary data:

```
// Voxel dimensions in the three dimensions
//
G4double halfX = ...;
G4double halfY = ...;
G4double halfZ = ...;
param->SetVoxelDimensions( halfX, halfY, halfZ );

// Number of voxels in the three dimensions
//
G4int nVoxelX = ...;
G4int nVoxelY = ...;
G4int nVoxelZ = ...;
param->SetNoVoxel( nVoxelX, nVoxelY, nVoxelZ );

// Vector of materials of the voxels
//
std::vector < G4Material* > theMaterials;
theMaterials.push_back( new G4Material( ...
theMaterials.push_back( new G4Material( ...
param->SetMaterials( theMaterials );

// List of material indices
// For each voxel it is a number that correspond to the index of its
// material in the vector of materials defined above;
//
size_t* mateIDs = new size_t[nVoxelX*nVoxelY*nVoxelZ];
mateIDs[0] = n0;
mateIDs[1] = n1;
...
param->SetMaterialIndices( mateIDs );
```

Then, define the volume that contains all the voxels:

```
G4Box* cont_solid = new G4Box("PhantomContainer",nVoxelX*halfX.,nVoxelY*halfY.,nVoxelZ*halfZ);
G4LogicalVolume* cont_logic =
  new G4LogicalVolume( cont_solid,
           matePatient,        // material is not relevant here...
           "PhantomContainer",
           0, 0, 0 );
G4VPhysicalVolume * cont_phys =
  new G4PVPlacement(rotm,                 // rotation
            pos,                  // translation
            cont_logic,           // logical volume
```

```
            "PhantomContainer",     // name
            world_logic,            // mother volume
            false,                  // No op. bool.
            1);                     // Copy number
```

The physical volume should be assigned as the container volume of the parameterisation:

```
param->BuildContainerSolid(cont_phys);

// Assure that the voxels are completely filling the container volume
//
param->CheckVoxelsFillContainer( cont_solid->GetXHalfLength(),
                                 cont_solid->GetyHalfLength(),
                                 cont_solid->GetzHalfLength() );

// The parameterised volume which uses this parameterisation is placed
// in the container logical volume
//
G4PVParameterised * patient_phys =
  new G4PVParameterised("Patient",               // name
                        patient_logic,           // logical volume
                        cont_logic,              // mother volume
            kXAxis,                   // optimisation hint
                        nVoxelX*nVoxelY*nVoxelZ, // number of voxels
                        param);                  // parameterisation

// Indicate that this physical volume is having a regular structure
//
patient_phys->SetRegularStructureId(1);
```

An example showing the application of the optimised navigation algorithm for phantoms geometries is available in `examples/extended/medical/DICOM`. It implements a real application for reading `DICOM` images and convert them to Geant4 geometries with defined materials and densities, allowing for different implementation solutions to be chosen (non-optimised, classical 3D optimisation, nested parameterisations and use of `G4PhantomParameterisation`).

Run-time commands

When running in *verbose* mode (i.e. the default, `G4VERBOSE` set while installing the Geant4 kernel libraries), the navigator provides a few commands to control its behavior. It is possible to select different verbosity levels (up to 5), with the command:

```
geometry/navigator/verbose [verbose_level]
```

or to force the navigator to run in *check* mode:

```
geometry/navigator/check_mode [true/false]
```

The latter will force more strict and less tolerant checks in step/safety computation to verify the correctness of the solids' response in the geometry.

By combining *check_mode* with verbosity level-1, additional verbosity checks on the response from the solids can be activated.

Setting Geometry Tolerance to be relative

The tolerance value defining the accuracy of tracking on the surfaces is by default set to a reasonably small value of *10E-9 mm*. Such accuracy may be however redundant for use on simulation of detectors of big size or macroscopic

dimensions. Since release 9.0, it is possible to specify the surface tolerance to be relative to the extent of the world volume defined for containing the geometry setup.

The class `G4GeometryManager` can be used to activate the computation of the surface tolerance to be relative to the geometry setup which has been defined. It can be done this way:

```
G4GeometryManager::GetInstance()->SetWorldMaximumExtent(WorldExtent);
```

where, `WorldExtent` is the actual maximum extent of the world volume used for placing the whole geometry setup.

Such call to `G4GeometryManager` must be done **before** defining any geometrical component of the setup (solid shape or volume), and can be done only **once**!

The class `G4GeometryTolerance` is to be used for retrieving the actual values defined for tolerances, surface (Cartesian), angular or radial respectively:

```
G4GeometryTolerance::GetInstance()->GetSurfaceTolerance();
G4GeometryTolerance::GetInstance()->GetAngularTolerance();
G4GeometryTolerance::GetInstance()->GetRadialTolerance();
```

4.1.9 A Simple Geometry Editor

GGE is the acronym for Geant4 Graphical Geometry Editor. GGE aims to assist physicists who have a little knowledge on C++ and the Geant4 toolkit to construct his or her own detector geometry. In essence, GGE is made up of a set of tables which can contain all relevant parameters to construct a simple detector geometry. Tables for scratch or compound materials, tables for logical and physical volumes are provided. From the values in the tables, C++ source codes are automatically generated.

GGE provides methods to:

1. construct a detector geometry including `G4Element`, `G4Material`, `G4Solids`, `G4LogicalVolume`, `G4PVPlacement`, etc.
2. view the detector geometry using existing visualization system, DAWN
3. keep the detector object in a persistent way, either in GDML format (currently only logical volumes are supported) or Java serialized format.
4. produce corresponding C++ codes after the norm of Geant4 toolkit
5. make a Geant4 executable, in collaboration with another component of MOMO, i.e., GPE, or Geant4 Physics Editor.

GGE can be found in the standard Geant4 source package under the directory `environments/MOMO/MOMO.jar`. JRE (Java Run-time Environment) is prerequisite to run MOMO.jar, Java archive file of MOMO. MOMO contains GGE, GPE, GAG and other helper tools. Further information is available from the Web pages below.

MOMO = GGE + GPE + GAG: http://www-geant4.kek.jp/~yoshidah

Materials: elements and mixtures

GGE provides the database of elements in the form of the periodic table, from which users can select element(s) to construct new materials. They can be loaded, used, edited and saved as Java persistent objects or in a GDML file. In `enviroments/MOMO`, a pre-constructed database of materials taken from the PDG book, `PDG.xml` is present.

Users can also create new materials either from scratch or by combining other materials.

- By selecting an element in the periodic table, default values as shown below are copied to a row in the table.

Use	Name	A	Z	Density	Unit	State	Temperature	Unit	Pressure	Unit

Use marks the used materials. Only the elements and materials used in the logical volumes are kept in the detector object and are used to generate C++ constructors.

- By selecting multiple elements in the periodic table, a material from a combination of elements is assigned to a row of the compound material table. The minimum actions user have to do is to give a name to the material and define its density.

Use	Name	Elements	Density	Unit	State	Temperature	Unit	Pressure	Unit

By clicking the column **Elements**, a new window is open to select one of two methods:

- Add an element, giving its fraction by weight
- Add an element, giving its number of atoms.

Solids

The most popular CSG solids (`G4Box`, `G4Tubs`, `G4Cons`, `G4Trd`) and specific solids (Pcons, Pgons) are supported. All relevant parameters of such a solid can be specified in the parameter table, which pops up upon selection.

Color, or the visualization attribute of a logical volume can be created, using color chooser panel. Users can view each solid using DAWN.

Logical Volume

GGE can specify the following items:

Name	Solid	Material	VisAttribute

The lists of solid types, names of the materials defined in the material tables, and names of user-defined visualization attributes are shown automatically in respective table cell for user's choices.

The construction and assignment of appropriate entities for `G4FieldManager` and `G4VSensitiveDetector` are left to the user.

Physical Volume

Geant4 enables users to create a physical volume in different ways; the mother volume can be either a logical or a physical one, spatial rotation can be either with respect to the volume or to the frame to which the volume is attached. GGE is prepared for such four combinatorial cases to construct a physical volume.

Five simple cases of creating physical volumes are supported by GGE. Primo, a single copy of a physical volume can be created by a translation and rotation. Secondo, repeated copies can be created by repeated linear translations. A logical volume is translated in a Cartesian direction, starting from the initial position, with a given step size. Mother volume can be either another logical volume or a physical volume.

Name	LogicalVolume	Type and name of MotherVolume	Many	X0, Y0, Z0	Direction	StepSize	Unit	CopyNumber

Third, repeated copies are created by rotation around an axis, placing an object repeatedly on a "cylindrical" pattern. Fourth, replicas are created by slicing a volume along a Cartesian direction. Fifth, replicas are created by cutting a volume cylindrically.

Generation of C++ code:

User has to type in a class name to his geometry, for example, `MyDetectorConstruction`. Then, with a mouse button click, source codes in the form of an include file and a source file are created and shown in the editor panel. In this example, they are `MyDetectorConstruction.cc` and `MyDetectorConstruction.hh` files. They reflect all current user modifications in the tables in real-time.

Visualization

The whole geometry can be visualized after the compilation of the source code `MyDetectorConstruction.cc` with appropriate parts of Geant4. (In particular only the geometry and visualization, together with the small other parts they depend on, are needed.) MOMO provides Physics Editor to create standard electromagnetic physics and a minimum main program. See the on-line document in MOMO.

4.1.10 Converting Geometries from Geant3.21

Approach

G3toG4 is the Geant4 facility to convert GEANT 3.21 geometries into Geant4. This is done in two stages:

1. The user supplies a GEANT 3.21 RZ-file (.rz) containing the initialization data structures. An executable `rztog4` reads this file and produces an ASCII *call list* file containing instructions on how to build the geometry. The source code of `rztog4` is FORTRAN.
2. A call list interpreter (`G4BuildGeom.cc`) reads these instructions and builds the geometry in the user's client code for Geant4.

Importing converted geometries into Geant4

Two examples of how to use the call list interpreter are supplied in the directory `examples/extended/g3tog4`:

1. `cltog4` is a simple example which simply invokes the call list interpreter method `G4BuildGeom` from the `G3toG4DetectorConstruction` class, builds the geometry and exits.
2. `clGeometry`, is more complete and is patterned as for the basic Geant4 examples. It also invokes the call list interpreter, but in addition, allows the geometry to be visualized and particles to be tracked.

To compile and build the G3toG4 libraries, you need to have set in your environment the variable `G4LIB_BUILD_G3TOG4` at the time of installation. The G3toG4 libraries are not built by default. Then, simply type:

```
gmake
```

from the top-level `source/g3tog4` directory.

To build the converter executable `rztog4`, simply type:

```
gmake bin
```

To make everything, simply type:

```
gmake global
```

To remove all `G3toG4` libraries, executables and .d files, simply type:

```
gmake clean
```

Current Status

The package has been tested with the geometries from experiments like: BaBar, CMS, Atlas, Alice, Zeus, L3, and Opal.

Here is a comprehensive list of features supported and not supported or implemented in the current version of the package:

- Supported shapes: all GEANT 3.21 shapes except for `GTRA`, `CTUB`.
- `PGON`, `PCON` are built using the *specific* solids `G4Polycone` and `G4Polyhedra`.
- GEANT 3.21 `MANY` feature is only partially supported. `MANY` positions are resolved in the `G3toG4MANY()` function, which has to be processed before `G3toG4BuildTree()` (it is not called by default). In order to resolve `MANY`, the user code has to provide additional info using `G4gsbool(G4String volName, G4String manyVolName)` function for all the overlapping volumes. Daughters of overlapping volumes are then resolved automatically and should not be specified via `Gsbool`.
  Limitation: a volume with a `MANY` position can have only this one position; if more than one position is needed a new volume has to be defined (`gsvolu()`) for each position.
- `GSDV*` routines for dividing volumes are implemented, using `G4PVReplica`s, for shapes:
  - `BOX`, `TUBE`, `TUBS`, `PARA` - all axes;
  - `CONE`, `CONS` - axes 2, 3;
  - `TRD1`, `TRD2`, `TRAP` - axis 3;
  - `PGON`, `PCON` - axis 2;
  - `PARA` -axis 1; axis 2,3 for a special case
- `GSPOSP` is implemented via individual logical volumes for each instantiation.
- `GSROTM` is implemented. Reflections of hierachies based on plain CSG solids are implemented through the `G3Division` class.
- Hits are not implemented.
- Conversion of GEANT 3.21 magnetic field is currently not supported. However, the usage of magnetic field has to be turned on.

4.1.11 Detecting Overlapping Volumes

The problem of overlapping volumes

Volumes are often positioned within other volumes with the intent that one is fully contained within the other. If, however, a volume extends beyond the boundaries of its mother volume, it is defined as overlapping. It may also be intended that volumes are positioned within the same mother volume such that they do not intersect one another. When such volumes do intersect, they are also defined as overlapping.

The problem of detecting overlaps between volumes is bounded by the complexity of the solid model description. Hence it requires the same mathematical sophistication which is needed to describe the most complex solid topology, in general. However, a tunable accuracy can be obtained by approximating the solids via first and/or second order surfaces and checking their intersections.

In general, the most powerful clash detection algorithms are provided by CAD systems, treating the intersection between the solids in their topological form.

Detecting overlaps at construction

The Geant4 geometry modeler provides the ability to detect overlaps of placed volumes (normal placements or parameterised) at the time of construction. This check is optional and can be activated when instantiating a placement (see `G4PVPlacement` constructor in *Placements: single positioned copy*) or a parameterised volume (see `G4PVParameterised` constructor in *Repeated volumes*).

The positioning of that specific volume will be checked against all volumes in the same hierarchy level and its mother volume. Depending on the complexity of the geometry being checked, the check may require considerable CPU time;

it is therefore suggested to use it only for debugging the geometry setup and to apply it only to the part of the geometry setup which requires debugging.

The classes `G4PVPlacement` and `G4PVParameterised` also provide a method:

```
G4bool CheckOverlaps(G4int res=1000, G4double tol=0., G4bool verbose=true)
```

which will force the check for the specified volume, and can be therefore used to verify for overlaps also once the geometry is fully built. The check verifies if each placed or parameterised instance is overlapping with other instances or with its mother volume. A default resolution for the number of points to be generated and verified is provided. The method returns `true` if an overlap occurs. It is also possible to specify a "tolerance" by which overlaps not exceeding such quantity will not be reported; by default, all overlaps are reported.

Detecting overlaps: built-in kernel commands

Built-in run-time commands to activate verification tests for the user-defined geometry are also provided

```
geometry/test/run
--> to start verification of geometry for overlapping regions
    recursively through the volumes tree.
geometry/test/recursion_start [int]
--> to set the starting depth level in the volumes tree from where
    checking overlaps. Default is level '0' (i.e. the world volume).
    The new settings will then be applied to any recursive test run.
geometry/test/recursion_depth [int]
--> to set the total depth in the volume tree for checking overlaps.
    Default is '-1' (i.e. checking the whole tree).
    Recursion will stop after having reached the specified depth (the
    default being the full depth of the geometry tree).
    The new settings will then be applied to any recursive test run.
geometry/test/tolerance [double] [unit]
--> to define tolerance by which overlaps should not be reported.
    Default is '0'.
geometry/test/verbosity [bool]
--> to set verbosity mode. Default is 'true'.
geometry/test/resolution [int]
--> to establish the number of points on surface to be generated
    and checked for each volume. Default is '10000'.
geometry/test/maximum_errors [int]
--> to fix the threshold for the number of errors to be reported
    for a single volume. By default, for each volume, reports stop
    after the first error reported.
```

To detect overlapping volumes, the built-in UI commands use the random generation of points on surface technique described above. It allows to detect with high level of precision any kind of overlaps, as depicted below. For example, consider *figure-overlap*:

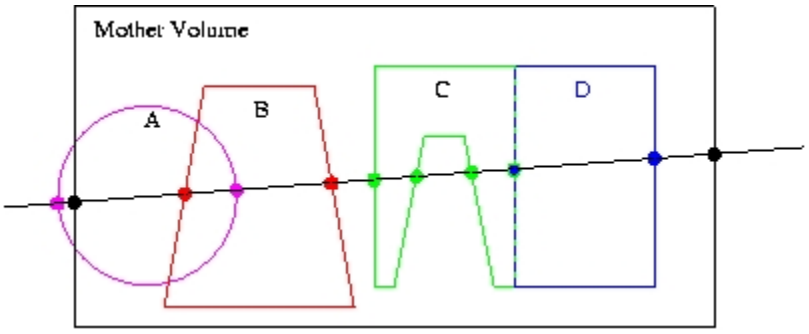


Fig. 4.4: Different cases of placed volumes overlapping each other.

Here we have a line intersecting some physical volume (large, black rectangle). Belonging to the volume are four

daughters: A, B, C, and D. Indicated by the dots are the intersections of the line with the mother volume and the four daughters.

This example has two geometry errors. First, volume A sticks outside its mother volume (this practice, sometimes used in GEANT3.21, is not allowed in Geant4). This can be noticed because the intersection point (leftmost magenta dot) lies outside the mother volume, as defined by the space between the two black dots.

The second error is that daughter volumes A and B overlap. This is noticeable because one of the intersections with A (rightmost magenta dot) is inside the volume B, as defined as the space between the red dots. Alternatively, one of the intersections with B (leftmost red dot) is inside the volume A, as defined as the space between the magenta dots.

Another difficult issue is roundoff error. For example, daughters C and D lie precisely next to each other. It is possible, due to roundoff, that one of the intersections points will lie just slightly inside the space of the other. In addition, a volume that lies tightly up against the outside of its mother may have an intersection point that just slightly lies outside the mother.

Finally, notice that no mention is made of the possible daughter volumes of A, B, C, and D. To keep the code simple, only the immediate daughters of a volume are checked at one pass. To test these "granddaughter" volumes, the daughters A, B, C, and D each have to be tested themselves in turn. To make this automatic, a recursive algorithm is applied; it first tests the target volume, then it loops over all daughter volumes and calls itself.

Note: for a complex geometry, checking the entire volume hierarchy can be extremely time consuming.

Using the visualization driver: DAVID

The Geant4 visualization offers a powerful debugging tool for detecting potential intersections of physical volumes. The Geant4 DAVID visualization tool can infact automatically detect the overlaps between the volumes defined in Geant4 and converted to a graphical representation for visualization purposes. The accuracy of the graphical representation can be tuned onto the exact geometrical description. In the debugging, physical-volume surfaces are automatically decomposed into 3D polygons, and intersections of the generated polygons are investigated. If a polygon intersects with another one, physical volumes which these polygons belong to are visualized in color (red is the default). The *DAVID-overlap* figure below is a sample visualization of a detector geometry with intersecting physical volumes highlighted:

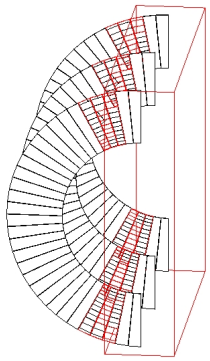


Fig. 4.5: A geometry with overlapping volumes highlighted by DAVID.

At present physical volumes made of the following solids can be debugged: `G4Box`, `G4Cons`, `G4Para`, `G4Sphere`, `G4Trd`, `G4Trap`, `G4Tubs`. (Existence of other solids is harmless.)

Visual debugging of physical-volume surfaces is performed with the DAWNFILE driver defined in the visualization category and with the two application packages, i.e. Fukui Renderer "DAWN" and a visual intersection debugger "DAVID". DAWN and DAVID can be downloaded from the Web.

How to compile Geant4 with the DAWNFILE driver incorporated is described in *The Visualization Drivers*.

If the DAWNFILE driver, DAWN and DAVID are all working well in your host machine, the visual intersection debugging of physical-volume surfaces can be performed as follows:

Run your Geant4 executable, invoke the DAWNFILE driver, and execute visualization commands to visualize your detector geometry:

```
Idle> /vis/open DAWNFILE
.....(setting camera etc)...
Idle> /vis/drawVolume
Idle> /vis/viewer/update
```

Then a file "g4.prim", which describes the detector geometry, is generated in the current directory and DAVID is invoked to read it. (The description of the format of the file g4.prim can be found from the DAWN web site documentation.)

If DAVID detects intersection of physical-volume surfaces, it automatically invokes DAWN to visualize the detector geometry with the intersected physical volumes highlighted (See the above sample visualization).

If no intersection is detected, visualization is skipped and the following message is displayed on the console:

```
-----------------------------------------------------
!!! Number of intersected volumes : 0 !!!
!!! Congratulations ! \(^o^)/         !!!
-----------------------------------------------------
```

If you always want to skip visualization, set an environmental variable as follows beforehand:

```
%  setenv DAVID_NO_VIEW  1
```

To control the precision associated to computation of intersections (default precision is set to 9), it is possible to use the environmental variable for the DAWNFILE graphics driver, as follows:

```
%  setenv G4DAWNFILE_PRECISION  10
```

If necessary, re-visualize the detector geometry with intersected parts highlighted. The data are saved in a file "g4david.prim" in the current directory. This file can be re-visualized with DAWN as follows:

```
% dawn g4david.prim
```

It is also helpful to convert the generated file g4david.prim into a VRML-formatted file and perform interactive visualization of it with your WWW browser. The file conversion tool `prim2wrml` can be downloaded from the DAWN web site download pages.

For more details, see the documentation of DAVID mentioned above.

4.1.12 Dynamic Geometry Setups

Geant4 can handle geometries which vary in time (e.g. a geometry varying between two runs in the same job).

It is considered a change to the geometry setup, whenever for the same physical volume:

- the shape or dimension of its related solid is modified;
- the positioning (translation or rotation) of the volume is changed;
- the volume (or a set of volumes, tree) is removed/replaced or added.

Whenever such a change happens, the geometry setup needs to be first "opened" for the change to be applied and afterwards "closed" for the optimisation to be reorganised.

In the general case, in order to notify the Geant4 system of the change in the geometry setup, the `G4RunManager` has to be messaged once the new geometry setup has been finalised:

```
G4RunManager::GeometryHasBeenModified();
```

The above notification needs to be performed also if a material associated to a *positioned* volume is changed, in order to allow for the internal materials/cuts table to be updated. However, for relatively complex geometries the re-optimisation step may be extremely inefficient, since it has the effect that the whole geometry setup will be re-optimised and re-initialised. In cases where only a limited portion of the geometry has changed, it may be suitable to apply the re-optimisation only to the affected portion of the geometry (subtree).

Since release 7.1 of the Geant4 toolkit, it is possible to apply re-optimisation local to the subtree of the geometry which has changed. The user will have to explicitly "open/close" the geometry providing a pointer to the top physical volume concerned:

Listing 4.9: Opening and closing a portion of the geometry without notifying the `G4RunManager`.

```
#include "G4GeometryManager.hh"

// Open geometry for the physical volume to be modified ...
//
G4GeometryManager::OpenGeometry(physCalor);

// Modify dimension of the solid ...
//
physCalor->GetLogicalVolume()->GetSolid()->SetXHalfLength(12.5*cm);

// Close geometry for the portion modified ...
//
G4GeometryManager::CloseGeometry(physCalor);
```

If the existing geometry setup is modified locally in more than one place, it may be convenient to apply such a technique only once, by specifying a physical volume on top of the hierarchy (subtree) containing all changed portions of the setup.

An alternative solution for dealing with dynamic geometries is to specify NOT to apply optimisation for the subtree affected by the change and apply the general solution of invoking the `G4RunManager`. In this case, a performance penalty at run-time may be observed (depending on the complexity of the not-optimised subtree), considering that, without optimisation, intersections to all volumes in the subtree will be explicitely computed each time.

Note: in multi-threaded runs, dynamic geometries are only allowed for runs consisting only of one event.

4.1.13 Importing XML Models Using GDML

Geometry Description Markup Language (GDML) is a markup language based on XML and suited for the description of detector geometry models. It allows for easy exchange of geometry data in a *human-readable* XML-based description and structured formatting.

The GDML parser is a component of Geant4 which can be built and installed as an optional choice. It allows for importing and exporting GDML files, following the schema specified in the GDML documentation. The installation of the plugin is optional and requires the installation of the XercesC DOM parser.

Examples of how to import and export a detector description model based on GDML, and also how to extend the GDML schema, are provided and can be found in `examples/extended/persistency/gdml`.

4.1.14 Importing ASCII Text Models

Since release 9.2 of Geant4, it is also possible to import geometry setups based on a plain text description, according to a well defined syntax for identifying the different geometrical entities (solids, volumes, materials and volume

attributes) with associated parameters. An example showing how to define a geometry in plain text format and import it in a Geant4 application is shown in `examples/extended/persistency/P03`. The example also covers the case of associating a sensitive detector to one of the volumes defined in the text geometry, the case of mixing C++ and text geometry definitions and the case of defining new tags in the text format so that regions and cuts by region can be defined in the text file. It also provides an example of how to write a geometry text file from the in-memory Geant4 geometry. For the details on the format see the dedicated manual.

4.1.15 Saving geometry tree objects in binary format

The Geant4 geometry tree can be stored in the Root binary file format using the *Root-I/O* technique provided by in Root. Such a binary file can then be used to quickly load the geometry into the memory or to move geometries between different Geant4 applications.

See *Object Persistency* for details and references.

4.2 Material

4.2.1 General considerations

In nature, materials (chemical compounds, mixtures) are made of elements, and elements are made of isotopes. Geant4 has three main classes designed to reflect this organization. Each of these classes has a table, which is a static data member, used to keep track of the instances of the respective classes created.

`G4Isotope` This class describes the properties of atoms: atomic number, number of nucleons, mass per mole, etc.

`G4Element` This class describes the properties of elements: effective atomic number, effective number of nucleons, effective mass per mole, number of isotopes, shell energy, and quantities like cross section per atom, etc.

`G4Material` This class describes the macroscopic properties of matter: density, state, temperature, pressure, and macroscopic quantities like radiation length, mean free path, dE/dx, etc.

Only the `G4Material` class is visible to the rest of the toolkit and used by the tracking, the geometry and the physics. It contains all the information relevant to its constituent elements and isotopes, while at the same time hiding their implementation details.

4.2.2 Introduction to the Classes

G4Isotope

A `G4Isotope` object has a name, atomic number, number of nucleons, mass per mole, and an index in the table. The constructor automatically stores "this" isotope in the isotopes table, which will assign it an index number. The `G4Isotope` objects are owned by the isotopes table, and must not be deleted by user code.

G4Element

A `G4Element` object has a name, symbol, effective atomic number, effective number of nucleons, effective mass of a mole, an index in the elements table, the number of isotopes, a vector of pointers to such isotopes, and a vector of relative abundances referring to such isotopes (where relative abundance means the number of atoms per volume). In addition, the class has methods to add, one by one, the isotopes which are to form the element.

The constructor automatically stores "this" element in the elements table, which will assign it an index number. The `G4Element` objects are owned by the elements table, and must not be deleted by user code.

A G4Element object can be constructed by directly providing the effective atomic number, effective number of nucleons, and effective mass of a mole, if the user explicitly wants to do so. Alternatively, a G4Element object can be constructed by declaring the number of isotopes of which it will be composed. The constructor will "new" a vector of pointers to G4Isotopes and a vector of doubles to store their relative abundances. Finally, the method to add an isotope must be invoked for each of the desired (pre-existing) isotope objects, providing their addresses and relative abundances. At the last isotope entry, the system will automatically compute the effective atomic number, effective number of nucleons and effective mass of a mole, and will store "this" element in the elements table.

A few quantities, with physical meaning or not, which are constant in a given element, are computed and stored here as "derived data members".

Using the internal Geant4 database, a G4Element can be accessed by atomic number or by atomic symbol ("Al", "Fe", "Pb"...). In that case G4Element will be found from the list of existing elements or will be constructed using data from the Geant4 database, which is derived from the NIST database of elements and isotope compositions. Thus, the natural isotope composition can be built by default. The same element can be created as using the NIST database with the natural composition of isotopes and from scratch in user code with user defined isotope composition.

G4Material

A G4Material object has a name, density, physical state, temperature and pressure (by default the standard conditions), the number of elements and a vector of pointers to such elements, a vector of the fraction of mass for each element, a vector of the atoms (or molecules) numbers of each element, and an index in the materials table. In addition, the class has methods to add, one by one, the elements which will comprise the material.

The constructor automatically stores "this" material in the materials table, which will assign it an index number. The G4Material objects are owned by the materials table, and must not be deleted by user code.

A G4Material object can be constructed by directly providing the resulting effective numbers, if the user explicitly wants to do so (an underlying element will be created with these numbers). Alternatively, a G4Material object can be constructed by declaring the number of elements of which it will be composed. The constructor will "new" a vector of pointers to G4Element and a vector of doubles to store their fraction of mass. Finally, the method to add an element must be invoked for each of the desired (pre-existing) element objects, providing their addresses and mass fractions. At the last element entry, the system will automatically compute the vector of the number of atoms of each element per volume, the total number of electrons per volume, and will store "this" material in the materials table. In the same way, a material can be constructed as a mixture of other materials and elements.

It should be noted that if the user provides the number of atoms (or molecules) for each element comprising the chemical compound, the system automatically computes the mass fraction. A few quantities, with physical meaning or not, which are constant in a given material, are computed and stored here as "derived data members".

Some materials are included in the internal Geant4 database, which were derived from the NIST database of material properties. Additionally a number of materials friquently used in HEP is included in the database. Materials are interrogated or constructed by their *names* (*Geant4 Material Database*). There are UI commands for the material category, which provide an interactive access to the database. If material is created using the NIST database by it will consist by default of elements with the natural composition of isotopes.

Final Considerations

The classes will automatically decide if the total of the mass fractions is correct, and perform the necessary checks. The main reason why a fixed index is kept as a data member is that many cross section and energy tables will be built in the physics processes "by rows of materials (or elements, or even isotopes)". The tracking gives the physics process the address of a material object (the material of the current volume). If the material has an index according to which the cross section table has been built, then direct access is available when a number in such a table must be accessed. We get directly to the correct row, and the energy of the particle will tell us the column. Without such an index, every access to the cross section or energy tables would imply a search to get to the correct material's row. More details will be given in the section on processes.

Isotopes, elements and materials must be instantiated dynamically in the user application; they are automatically registered in internal stores and the system takes care to free the memory allocated at the end of the job.

4.2.3 Recipes for Building Elements and Materials

The *example below* illustrates the different ways to define materials.

Listing 4.10: A program which illustrates the different ways to define materials.

```
#include "G4Isotope.hh"
#include "G4Element.hh"
#include "G4Material.hh"
#include "G4UnitsTable.hh"

int main() {
G4String name, symbol;             // a=mass of a mole;
G4double a, z, density;            // z=mean number of protons;
G4int iz, n;                       // iz=nb of protons  in an isotope;
                                   // n=nb of nucleons in an isotope;
G4int ncomponents, natoms;
G4double abundance, fractionmass;
G4double temperature, pressure;

G4UnitDefinition::BuildUnitsTable();

// define Elements
a = 1.01*g/mole;
G4Element* elH  = new G4Element(name="Hydrogen",symbol="H" , z= 1., a);

a = 12.01*g/mole;
G4Element* elC  = new G4Element(name="Carbon"  ,symbol="C" , z= 6., a);

a = 14.01*g/mole;
G4Element* elN  = new G4Element(name="Nitrogen",symbol="N" , z= 7., a);

a = 16.00*g/mole;
G4Element* elO  = new G4Element(name="Oxygen"  ,symbol="O" , z= 8., a);

a = 28.09*g/mole;
G4Element* elSi = new G4Element(name="Silicon", symbol="Si", z=14., a);

a = 55.85*g/mole;
G4Element* elFe = new G4Element(name="Iron"    ,symbol="Fe", z=26., a);

a = 183.84*g/mole;
G4Element* elW = new G4Element(name="Tungsten" ,symbol="W",  z=74., a);

a = 207.20*g/mole;
G4Element* elPb = new G4Element(name="Lead"    ,symbol="Pb", z=82., a);

// define an Element from isotopes, by relative abundance
G4Isotope* U5 = new G4Isotope(name="U235", iz=92, n=235, a=235.01*g/mole);
G4Isotope* U8 = new G4Isotope(name="U238", iz=92, n=238, a=238.03*g/mole);

G4Element* elU  = new G4Element(name="enriched Uranium", symbol="U", ncomponents=2);
elU->AddIsotope(U5, abundance= 90.*perCent);
elU->AddIsotope(U8, abundance= 10.*perCent);

G4cout << *(G4Isotope::GetIsotopeTable()) << G4endl;
G4cout << *(G4Element::GetElementTable()) << G4endl;
```

```
// define simple materials
density = 2.700*g/cm3;
a = 26.98*g/mole;
G4Material* Al = new G4Material(name="Aluminum", z=13., a, density);
```

```
density = 1.390*g/cm3;
a = 39.95*g/mole;
G4Material* lAr = new G4Material(name="liquidArgon", z=18., a, density);

density = 8.960*g/cm3;
a = 63.55*g/mole;
G4Material* Cu = new G4Material(name="Copper"   , z=29., a, density);

// define a material from elements.   case 1: chemical molecule
density = 1.000*g/cm3;
G4Material* H2O = new G4Material(name="Water", density, ncomponents=2);
H2O->AddElement(elH, natoms=2);
H2O->AddElement(elO, natoms=1);

density = 1.032*g/cm3;
G4Material* Sci = new G4Material(name="Scintillator", density, ncomponents=2);
Sci->AddElement(elC, natoms=9);
Sci->AddElement(elH, natoms=10);

density = 2.200*g/cm3;
G4Material* SiO2 = new G4Material(name="quartz", density, ncomponents=2);
SiO2->AddElement(elSi, natoms=1);
SiO2->AddElement(elO , natoms=2);

density = 8.280*g/cm3;
G4Material* PbWO4= new G4Material(name="PbWO4", density, ncomponents=3);
PbWO4->AddElement(elO , natoms=4);
PbWO4->AddElement(elW , natoms=1);
PbWO4->AddElement(elPb, natoms=1);

// define a material from elements.   case 2: mixture by fractional mass
density = 1.290*mg/cm3;
G4Material* Air = new G4Material(name="Air  "   , density, ncomponents=2);
Air->AddElement(elN, fractionmass=0.7);
Air->AddElement(elO, fractionmass=0.3);

// define a material from elements and/or others materials (mixture of mixtures)
density = 0.200*g/cm3;
G4Material* Aerog = new G4Material(name="Aerogel", density, ncomponents=3);
Aerog->AddMaterial(SiO2, fractionmass=62.5*perCent);
Aerog->AddMaterial(H2O , fractionmass=37.4*perCent);
Aerog->AddElement (elC , fractionmass= 0.1*perCent);
```

```
// examples of gas in non STP conditions
density     = 27.*mg/cm3;
pressure    = 50.*atmosphere;
temperature = 325.*kelvin;
G4Material* CO2 = new G4Material(name="Carbonic gas", density, ncomponents=2,
                                 kStateGas,temperature,pressure);
CO2->AddElement(elC, natoms=1);
CO2->AddElement(elO, natoms=2);

density     = 0.3*mg/cm3;
pressure    = 2.*atmosphere;
temperature = 500.*kelvin;
G4Material* steam = new G4Material(name="Water steam ", density, ncomponents=1,
                                   kStateGas,temperature,pressure);
steam->AddMaterial(H2O, fractionmass=1.);

// What about vacuum ?  Vacuum is an ordinary gas with very low density
density     = universe_mean_density;                //from PhysicalConstants.h
pressure    = 1.e-19*pascal;
temperature = 0.1*kelvin;
new G4Material(name="Galactic", z=1., a=1.01*g/mole, density,
                  kStateGas,temperature,pressure);
```

```
density     = 1.e-5*g/cm3;
pressure    = 2.e-2*bar;
temperature = STP_Temperature;                    //from PhysicalConstants.h
G4Material* beam = new G4Material(name="Beam ", density, ncomponents=1,
                                  kStateGas,temperature,pressure);
beam->AddMaterial(Air, fractionmass=1.);

// print the table of materials
G4cout << *(G4Material::GetMaterialTable()) << G4endl;

return EXIT_SUCCESS;
}
```

As can be seen in the later examples, a material has a state: solid (the default), liquid, or gas. The constructor checks the density and automatically sets the state to gas below a given threshold (10 mg/cm3).

In the case of a gas, one may specify the temperature and pressure. The defaults are STP conditions defined in `PhysicalConstants.hh`.

An element must have the number of nucleons >= number of protons >= 1.

A material must have non-zero values of density, temperature and pressure.

Materials can also be defined using the internal Geant4 database. *Materials example 2* illustrates how to do this for the same materials used in *Materials example 1*. There are also UI commands which allow the database to be accessed. *The list of currently avalable material names* (*Geant4 Material Database*) is extended permanently.

Listing 4.11: A program which shows how to define materials from the internal database.

```
#include "globals.hh"
#include "G4Material.hh"
#include "G4NistManager.hh"

int main() {
  G4NistManager* man = G4NistManager::Instance();
  man->SetVerbose(1);

  // define elements
  G4Element* C  = man->FindOrBuildElement("C");
  G4Element* Pb = man->FindOrBuildMaterial("Pb");

  // define pure NIST materials
  G4Material* Al = man->FindOrBuildMaterial("G4_Al");
  G4Material* Cu = man->FindOrBuildMaterial("G4_Cu");

  // define NIST materials
  G4Material* H2O  = man->FindOrBuildMaterial("G4_WATER");
  G4Material* Sci  = man->FindOrBuildMaterial("G4_PLASTIC_SC_VINYLTOLUENE");
  G4Material* SiO2 = man->FindOrBuildMaterial("G4_SILICON_DIOXIDE");
  G4Material* Air  = man->FindOrBuildMaterial("G4_AIR");

  // HEP materials
  G4Material* PbWO4  = man->FindOrBuildMaterial("G4_PbWO4");
  G4Material* lAr    = man->FindOrBuildMaterial("G4_lAr");
  G4Material* vac    = man->FindOrBuildMaterial("G4_Galactic");

  // define gas material at non STP conditions (T = 120K, P=0.5atm)
  G4Material* coldAr = man->ConstructNewGasdMaterial("ColdAr","G4_Ar",120.*kelvin,0.
↪5*atmosphere);

  // print the table of materials
  G4cout << *(G4Material::GetMaterialTable()) << G4endl;

  return EXIT_SUCCESS;
}
```

4.2.4 The Tables

Print a constituent

The following shows how to print a constituent:

```
G4cout << elU << G4endl;
G4cout << Air << G4endl;
```

Print the table of materials

The following shows how to print the table of materials:

```
G4cout << *(G4Material::GetMaterialTable()) << G4endl;
```

4.3 Electromagnetic Field

4.3.1 An Overview of Propagation in a Field

Geant4 is capable of describing and propagating in a variety of fields. Magnetic fields, electric fields, electromagnetic fields, and gravity fields, uniform or non-uniform, can specified for a Geant4 setup. The propagation of tracks inside them can be performed to a user-defined accuracy.

In order to propagate a track inside a field, the equation of motion of the particle in the field is integrated. In general, this is done using a Runge-Kutta method for the integration of ordinary differential equations. However, for specific cases where an analytical solution is known, it is possible to utilize this instead. Several Runge-Kutta methods are available, suitable for different conditions. In specific cases (such as a uniform field where the analytical solution is known) different solvers can also be used. In addition, when an approximate analytical solution is known, it is possible to utilize it in an iterative manner in order to converge to the solution to the precision required. This latter method is currently implemented and can be used particularly well for magnetic fields that are almost uniform.

Once a method is chosen that calculates the track's propagation in a specific field, the curved path is broken up into linear chord segments. These chord segments are determined so that they closely approximate the curved path. The chords are then used to interrogate the Navigator as to whether the track has crossed a volume boundary. Several parameters are available to adjust the accuracy of the integration and the subsequent interrogation of the model geometry.

How closely the set of chords approximates a curved trajectory is governed by a parameter called the *miss distance* (also called the *chord distance*). This is an upper bound for the value of the sagitta - the distance between the 'real' curved trajectory and the approximate linear trajectory of the chord. By setting this parameter, the user can control the precision of the volume interrogation. Every attempt has been made to ensure that all volume interrogations will be made to an accuracy within this *miss distance*.

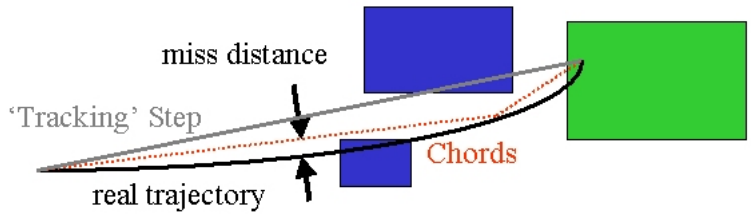


Fig. 4.6: The curved trajectory will be approximated by chords, so that the maximum estimated distance and chord is less than the *miss distance*.

In addition to the *miss distance* there are two more parameters which the user can set in order to adjust the accuracy (and performance) of tracking in a field. In particular these parameters govern the accuracy of the intersection with a volume boundary and the accuracy of the integration of other steps. As such they play an important role for tracking.

The *delta intersection* parameter is the accuracy to which an intersection with a volume boundary is calculated. If a candidate boundary intersection is estimated to have a precision better than this, it is accepted. This parameter is especially important because it is used to limit a bias that our algorithm (for boundary crossing in a field) exhibits. This algorithm calculates the intersection with a volume boundary using a chord between two points on the curved particle trajectory. As such, the intersection point is always on the 'inside' of the curve. By setting a value for this parameter that is much smaller than some acceptable error, the user can limit the effect of this bias on, for example, the future estimation of the reconstructed particle momentum.

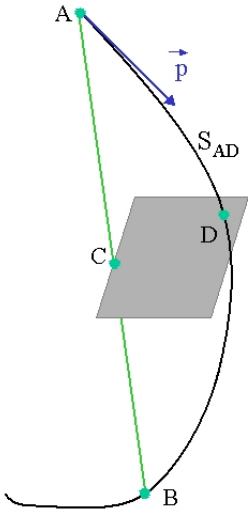


Fig. 4.7: The distance between the calculated chord intersection point C and a computed curve point D is used to determine whether C is an accurate representation of the intersection of the curved path ADB with a volume boundary. Here CD is likely too large, and a new intersection on the chord AD will be calculated.

The *delta one step* parameter is the accuracy for the endpoint of 'ordinary' integration steps, those which do not intersect a volume boundary. This parameter is a limit on the estimated error of the endpoint of each physics step. It can be seen as akin to a statistical uncertainty and is not expected to contribute any systematic behavior to physical quantities. In contrast, the bias addressed by *delta intersection* is clearly correlated with potential systematic errors in the momentum of reconstructed tracks. Thus very strict limits on the intersection parameter should be used in tracking detectors or wherever the intersections are used to reconstruct a track's momentum.

Delta intersection and *delta one step* are parameters of the Field Manager; the user can set them according to the demands of his application. Because it is possible to use more than one field manager, different values can be set for different detector regions.

Note that reasonable values for the two parameters are strongly coupled: it does not make sense to request an accuracy of 1 nm for *delta intersection* and accept 100 &#956m for the *delta one step* error value. Nevertheless *delta intersection* is the more important of the two. It is recommended that these parameters should not differ significantly - certainly not by more than an order of magnitude.

4.3.2 Practical Aspects

Creating a Magnetic Field for a Detector

The simplest way to define a field for a detector involves the following steps:

1. create a field:

```
G4UniformMagField* magField
  = new G4UniformMagField(G4ThreeVector(0.,0.,fieldValue));
```

2. set it as the default field:

```
G4FieldManager* fieldMgr
  = G4TransportationManager::GetTransportationManager()
    ->GetFieldManager();
   fieldMgr->SetDetectorField(magField);
```

3. create the objects which calculate the trajectory:

```
fieldMgr->CreateChordFinder(magField);
```

To change the accuracy of volume intersection use the `SetDeltaChord` method:

```
fieldMgr->GetChordFinder()->SetDeltaChord( G4double newValue);
```

Since 10.0 version, it is also possible to perform all three steps above at once using the `G4GlobalMagFieldMessenger` class:

```
G4ThreeVector fieldValue = G4ThreeVector();
fMagFieldMessenger = new G4GlobalMagFieldMessenger(fieldValue);
fMagFieldMessenger->SetVerboseLevel(1);
```

The messenger creates the global uniform magnetic field, which is activated (set to the `G4TransportationManager` object) only when the `fieldValue` is non zero vector. The messenger class setter functions can be then used to change the field value (and activate or inactivate the field again) or the level of output messages. The messenger also takes care of deleting the field.

As its class name suggests, the messenger creates also UI commands which can be used to change the field value and the verbose level interactively or from a macro:

```
/globalField/setValue vx vy vz unit
/globalField/verbose level
```

Creating a Field for a Part of the Volume Hierarchy

It is possible to create a field for a part of the detector. In particular it can describe the field (with pointer fEmField, for example) inside a logical volume and all its daughters. This can be done by simply creating a `G4FieldManager` and attaching it to a logical volume (with pointer, logicVolumeWithField, for example) or set of logical volumes.

```
G4bool allLocal = true;
logicVolumeWithField->SetFieldManager(localFieldManager, allLocal);
```

Using the second parameter to `SetFieldManager` you choose whether daughter volumes of this logical volume will also be given this new field. If it has the value `true`, the field will be assigned also to its daughters, and all their sub-volumes. Else, if it is `false`, it will be copied only to those daughter volumes which do not have a field manager already.

Creating an Electric or Electromagnetic Field

The design and implementation of the *Field* category allows and enables the use of an electric or combined electromagnetic field. These fields can also vary with time, as can magnetic fields.

Source listing *example-emfield-1* shows how to define a uniform electric field for the whole of a detector.

Listing 4.12: How to define a uniform electric field for the whole of a detector, extracted from example in examples/extended/field/field02 .

```
// in the header file (or first)
#include "G4EqMagElectricField.hh"
#include "G4UniformElectricField.hh"

...
G4ElectricField*        fEMfield;
G4EqMagElectricField*   fEquation;
G4MagIntegratorStepper* fStepper;
G4FieldManager*         fFieldMgr;
G4double                fMinStep ;
G4ChordFinder*          fChordFinder ;

// in the source file

{
  fEMfield = new G4UniformElectricField(
              G4ThreeVector(0.0,100000.0*kilovolt/cm,0.0));

  // Create an equation of motion for this field
  fEquation = new G4EqMagElectricField(fEMfield);

  G4int nvar = 8;
  fStepper = new G4ClassicalRK4( fEquation, nvar );

  // Get the global field manager
  fFieldManager= G4TransportationManager::GetTransportationManager()->
       GetFieldManager();
  // Set this field to the global field manager
  fFieldManager->SetDetectorField(fEMfield );

  fMinStep     = 0.010*mm ; // minimal step of 10 microns

  fIntgrDriver = new G4MagInt_Driver(fMinStep,
                                     fStepper,
                                     fStepper->GetNumberOfVariables() );

  fChordFinder = new G4ChordFinder(fIntgrDriver);
  fFieldManager->SetChordFinder( fChordFinder );
}
```

An example with an electric field is examples/extended/field/field02, where the class F02ElectricFieldSetup demonstrates how to set these and other parameters, and how to choose different Integration Steppers. An example with a uniform gravity field (G4UniformGravityField) is examples/extended/field/field06.

The user can also create their own type of field, inheriting from `G4VField`, and an associated Equation of Motion class (inheriting from `G4EqRhs`) to simulate other types of fields.

Choosing a Stepper

Runge-Kutta integration is used to compute the motion of a charged track in a general field. There are many general steppers from which to choose, of low and high order, and specialized steppers for pure magnetic fields. By default, Geant4 uses the classical fourth-order Runge-Kutta stepper, which is general purpose and robust. If the field is known to have specific properties, lower or higher order steppers can be used to obtain the same quality results using fewer computing cycles.

In particular, if the field is calculated from a field map, a lower order stepper is recommended. The less smooth the field is, the lower the order of the stepper that should be used. The choice of lower order steppers includes the third order stepper `G4SimpleHeum`, the second order `G4ImplicitEuler` and `G4SimpleRunge`, and the first order `G4ExplicitEuler`. A first order stepper would be useful only for very rough fields. For somewhat smooth fields

(intermediate), the choice between second and third order steppers should be made by trial and error. Trying a few different types of steppers for a particular field or application is suggested if maximum performance is a goal.

The choice of stepper depends on the type of field: magnetic or general. A general field can be an electric or electromagnetic field, it can be a magnetic field or a user-defined field (which requires a user-defined equation of motion.) For a general field several steppers are available as alternatives to the default (`G4ClassicalRK4`):

```
G4int nvar = 8;  // To integrate time & energy
                 //    in addition to position, momentum
G4EqMagElectricField* fEquation= new G4EqMagElectricField(fEMfield);

fStepper = new G4SimpleHeum( fEquation, nvar );
         //  3rd  order, a good alternative to ClassicalRK
fStepper = new G4SimpleRunge( fEquation, nvar );
         //  2nd  order, for less smooth fields
fStepper = new G4CashKarpRKF45( fEquation );
         // 4/5th order for very smooth fields
```

Specialized steppers for pure magnetic fields are also available. They take into account the fact that a local trajectory in a slowly varying field will not vary significantly from a helix. Combining this in with a variation the Runge-Kutta method can provide higher accuracy at lower computational cost when large steps are possible.

```
G4Mag_UsualEqRhs*
   fEquation = new G4Mag_UsualEqRhs(fMagneticField);
fStepper = new G4HelixImplicitEuler( fEquation );
// Note that for magnetic field that do not vary with time,
//  the default number of variables suffices.

// or ..
fStepper = new G4HelixExplicitEuler( fEquation );
fStepper = new G4HelixSimpleRunge( fEquation );
```

A new stepper for propagation in magnetic field is available in release 9.3. Choosing the G4NystromRK4 stepper provides accuracy near that of G4ClassicalRK4 (4th order) with a significantly reduced cost in field evaluation. Using a novel analytical expression for estimating the error of a proposed step and the Nystrom reuse of the mid-point field value, it requires only 2 additional field evaluations per attempted step, in place of 10 field evaluations of ClassicalRK4 (which uses the general midpoint method for estimating the step error.)

```
G4Mag_UsualEqRhs*
   pMagFldEquation = new G4Mag_UsualEqRhs(fMagneticField);
fStepper = new G4NystromRK4( pMagFldEquation );
```

It is proposed as an alternative stepper in the case of a pure magnetic field. It is not applicable for the simulation of electric or full electromagnetic or other types of field. For a pure magnetic field, results should be fully compatible with the results of ClassicalRK4 in nearly all cases. (The only potential exceptions are large steps for tracks with small momenta - which cannot be integrated well by any RK method except the Helical extended methods.)

You can choose an alternative stepper either when the field manager is constructed or later. At the construction of the ChordFinder it is an optional argument:

```
G4ChordFinder( G4MagneticField* itsMagField,
               G4double         stepMinimum = 1.0e-2 * mm,
               G4MagIntegratorStepper* pItsStepper = 0 );
```

To change the stepper at a later time use

```
pChordFinder->GetIntegrationDriver()
            ->RenewStepperAndAdjust( newStepper );
```

How to Adjust the Accuracy of Propagation

In order to obtain a particular accuracy in tracking particles through an electromagnetic field, it is necessary to adjust the parameters of the field propagation module. In the following section, some of these additional parameters are discussed.

When integration is used to calculate the trajectory, it is necessary to determine an acceptable level of numerical imprecision in order to get performant simulation with acceptable errors. The parameters in Geant4 tell the field module what level of integration inaccuracy is acceptable.

In all quantities which are integrated (position, momentum, energy) there will be errors. Here, however, we focus on the error in two key quantities: the position and the momentum. (The error in the energy will come from the momentum integration).

Three parameters exist which are relevant to the integration accuracy. DeltaOneStep is a distance and is roughly the position error which is acceptable in an integration step. Since many integration steps may be required for a single physics step, DeltaOneStep should be a fraction of the average physics step size. The next two parameters impose a further limit on the relative error of the position/momentum inaccuracy. EpsilonMin and EpsilonMax impose a minimum and maximum on this relative error - and take precedence over DeltaOneStep. (Note: if you set EpsilonMin=EpsilonMax=your-value, then all steps will be made to this relative precision.

Listing 4.13: How to set accuracy parameters for the 'global' field of the setup.

```
G4FieldManager *globalFieldManager;

G4TransportationManager *transportMgr=
     G4TransportationManager::GetTransportationManager();

globalFieldManager = transportMgr->GetFieldManager();
                        // Relative accuracy values:
G4double minEps= 1.0e-5;  //   Minimum & value for smallest steps
G4double maxEps= 1.0e-4;  //   Maximum & value for largest steps

globalFieldManager->SetMinimumEpsilonStep( minEps );
globalFieldManager->SetMaximumEpsilonStep( maxEps );
globalFieldManager->SetDeltaOneStep( 0.5e-3 * mm );  // 0.5 micrometer

G4cout << "EpsilonStep: set min= " << minEps << " max= " << maxEps << G4endl;
```

We note that the relevant parameters above limit the inaccuracy in each step. The final inaccuracy due to the full trajectory will accumulate!

The exact point at which a track crosses a boundary is also calculated with finite accuracy. To limit this inaccuracy, a parameter called DeltaIntersection is used. This is a maximum for the inaccuracy of a single boundary crossing. Thus the accuracy of the position of the track after a number of boundary crossings is directly proportional to the number of boundaries.

Reducing the number of field calls to speed-up simulation

An additional method to reduce the number of field evaluations is to utilise the new class G4CachedMagneticField class. It is applicable only for pure magnetic fields which do not vary with time.

```
G4MagneticField * pMagField;  // Your field - Defined elsewhere

G4double                distanceConst = 2.5 * millimeter;
G4MagneticField * pCachedMagField= new G4CachedMagneticField(  pMagField,  distanceConst);
```

Choosing different accuracies for the same volume

It is possible to create a FieldManager which has different properties for particles of different momenta (or depending on other parameters of a track). This is useful, for example, in obtaining high accuracy for 'important' tracks (e.g. muons) and accept less accuracy in tracking others (e.g. electrons). To use this, you must create your own field manager which uses the method

```
void ConfigureForTrack( const G4Track * );
```

to configure itself using the parameters of the current track. An example of this will be available in examples/extended/field05.

Parameters that must scale with problem size

The default settings of this module are for problems with the physical size of a typical high energy physics setup, that is, distances smaller than about one kilometer. A few parameters are necessary to carry this information to the magnetic field module, and must typically be rescaled for problems of vastly different sizes in order to get reasonable performance and robustness. Two of these parameters are the maximum acceptable step and the minimum step size.

The **maximum acceptable step** should be set to a distance larger than the biggest reasonable step. If the apparatus in a setup has a diameter of two meters, a likely maximum acceptable steplength would be 10 meters. A particle could then take large spiral steps, but would not attempt to take, for example, a 1000-meter-long step in the case of a very low-density material. Similarly, for problems of a planetary scale, such as the earth with its radius of roughly 6400 km, a maximum acceptabe steplength of a few times this value would be reasonable.

An upper limit for the size of a step is a parameter of `G4PropagatorInField`, and can be set by calling its `SetLargestAcceptableStep` method.

The **minimum step size** is used during integration to limit the amount of work in difficult cases. It is possible that strong fields or integration problems can force the integrator to try very small steps; this parameter stops them from becoming unnecessarily small.

Trial steps smaller than this parameter will be treated with less accuracy, and may even be ignored, depending on the situation.

The minimum step size is a parameter of the MagInt_Driver, but can be set in the contstructor of G4ChordFinder, as in the source listing above.

Known Issues

Currently it is computationally expensive to change the *miss distance* to very small values, as it causes tracks to be limited to curved sections whose 'bend' is smaller than this value. (The bend is the distance of the mid-point from the chord between endpoints.) For tracks with small curvature (typically low momentum particles in strong fields) this can cause a large number of steps

- even in areas where there are no volumes to intersect (something that is expected to be addressed in future development, in which the safety will be utilized to partially alleviate this limitation)
- especially in a region near a volume boundary (in which case it is necessary in order to discover whether a track might intersect a volume for only a short distance.)

Requiring such precision at the intersection is clearly expensive, and new development would be necessary to minimize the expense.

By contrast, changing the intersection parameter is less computationally expensive. It causes further calculation for only a fraction of the steps, in particular those that intersect a volume boundary.

4.3.3 Spin Tracking

The effects of a particle's motion on the precession of its spin angular momentum in slowly varying external fields are simulated. The relativistic equation of motion for spin is known as the BMT equation. The equation demonstrates a remarkable property; in a purely magnetic field, in vacuum, and neglecting small anomalous magnetic moments, the particle's spin precesses in such a manner that the longitudinal polarization remains a constant, whatever the motion of the particle. But when the particle interacts with electric fields of the medium and multiple scatters, the spin, which is related to the particle's magnetic moment, does not participate, and the need thus arises to propagate it independent of the momentum vector. In the case of a polarized muon beam, for example, it is important to predict the muon's spin direction at decay-time in order to simulate the decay electron (Michel) distribution correctly.

In order to track the spin of a particle in a magnetic field, you need to code the following:

1. in your DetectorConstruction

   ```
   #include "G4Mag_SpinEqRhs.hh"
   ```

   G4Mag_EqRhs* fEquation = new G4Mag_SpinEqRhs(magField);
   G4MagIntegratorStepper* pStepper = new G4ClassicalRK4(fEquation,12); // notice the 12
2. in your PrimaryGenerator

   ```
   particleGun->SetParticlePolarization(G4ThreeVector p)
   ```

   for example:

   ```
   particleGun->
   SetParticlePolarization(-(particleGun->GetParticleMomentumDirection()));

   // or
   particleGun->
   SetParticlePolarization(particleGun->GetParticleMomentumDirection()
                                        .cross(G4ThreeVector(0.,1.,0.)));
   ```

   where you set the initial spin direction.

While the G4Mag_SpinEqRhs class constructor

```
G4Mag_SpinEqRhs::G4Mag_SpinEqRhs( G4MagneticField* MagField )
   : G4Mag_EqRhs( MagField )
{
    anomaly = 1.165923e-3;
}
```

sets the muon anomaly by default, the class also comes with the public method:

```
inline void SetAnomaly(G4double a) { anomaly = a; }
```

with which you can set the magnetic anomaly to any value you require.

The code has been rewritten (in Release 9.5) such that field tracking of the spin can now be done for charged and neutral particles with a magnetic moment, for example spin tracking of ultra cold neutrons. This requires the user to set `EnableUseMagneticMoment`, a method of the `G4Transportation` process. The force resulting from the term, MUSDOTNABLAB, is not yet implemented in Geant4 (for example, simulated trajectory of a neutral hydrogen atom trapped by its magnetic moment in a gradient B-field.)

4.4 Hits

4.4.1 Hit

A hit is a snapshot of the physical interaction of a track in the sensitive region of a detector. In it you can store information associated with a `G4Step` object. This information can be

- the position and time of the step,
- the momentum and energy of the track,
- the energy deposition of the step,
- geometrical information,

or any combination of the above.

G4VHit

`G4VHit` is an abstract base class which represents a hit. You must inherit this base class and derive your own concrete hit class(es). The member data of your concrete hit class can be, and should be, your choice.

As with `G4THitsCollection`, authors of subclasses must declare templated `G4Allocator`s for their class. They must also implement *operator new()* and *operator delete()* which use these allocators.

`G4VHit` has two virtual methods, `Draw()` and `Print()`. To draw or print out your concrete hits, these methods should be implemented. How to define the drawing method is described in *Polylines, Markers and Text*.

G4THitsCollection

`G4VHit` is an abstract class from which you derive your own concrete classes. During the processing of a given event, represented by a `G4Event` object, many objects of the hit class will be produced, collected and associated with the event. Therefore, for each concrete hit class you must also prepare a concrete class derived from `G4VHitsCollection`, an abstract class which represents a vector collection of user defined hits.

`G4THitsCollection` is a template class derived from `G4VHitsCollection`, and the concrete hit collection class of a particular `G4VHit` concrete class can be instantiated from this template class. Each object of a hit collection must have a unique name for each event.

`G4Event` has a `G4HCofThisEvent` class object, that is a container class of collections of hits. Hit collections are stored by their pointers, whose type is that of the base class.

An example of a concrete hit class

The example shows an example of a concrete hit class.

Listing 4.14: An example of a concrete hit class.

```
//============ header file =====================

#ifndef ExN04TrackerHit_h
#define ExN04TrackerHit_h 1

#include "G4VHit.hh"
#include "G4THitsCollection.hh"
#include "G4Allocator.hh"
#include "G4ThreeVector.hh"

class ExN04TrackerHit : public G4VHit
{
```

```
  public:

      ExN04TrackerHit();
      ~ExN04TrackerHit();
      ExN04TrackerHit(const ExN04TrackerHit &right);
      const ExN04TrackerHit& operator=(const ExN04TrackerHit &right);
      int operator==(const ExN04TrackerHit &right) const;

      inline void * operator new(size_t);
      inline void operator delete(void *aHit);

      void Draw() const;
      void Print() const;

  private:
      G4double edep;
      G4ThreeVector pos;

  public:
      inline void SetEdep(G4double de)
      { edep = de; }
      inline G4double GetEdep() const
      { return edep; }
      inline void SetPos(G4ThreeVector xyz)
      { pos = xyz; }
      inline G4ThreeVector GetPos() const
      { return pos; }

};

typedef G4THitsCollection<ExN04TrackerHit> ExN04TrackerHitsCollection;

extern G4ThreadLocal G4Allocator<ExN04TrackerHit>* ExN04TrackerHitAllocator;

inline void* ExN04TrackerHit::operator new(size_t)
{
  if(!ExN04TrackerHitAllocator) ExN04TrackerHitAllocator = new G4Allocator<ExN04TrackerHit>
  return (void *) ExN04TrackerHitAllocator->MallocSingle();
}

inline void ExN04TrackerHit::operator delete(void *aHit)
{
  ExN04TrackerHitAllocator->FreeSingle((ExN04TrackerHit*) aHit);
}

#endif

//============ source file =====================

#include "ExN04TrackerHit.hh"

G4ThreadLocal G4Allocator<ExN04TrackerHit>* ExN04TrackerHit::ExN04TrackerHitAllocator = 0;

 ... snipped ...
```

`G4Allocator` is a class for fast allocation of objects to the heap through the paging mechanism. For details of `G4Allocator`, refer to *General management classes*. Use of `G4Allocator` is not mandatory, but it is recommended, especially for users who are not familiar with the C++ memory allocation mechanism or alternative tools of memory allocation. On the other hand, note that `G4Allocator` is to be used **only** for the concrete class that is **not** used as a base class of any other classes. For example, do **not** use the `G4Trajectory` class as a base class for a customized trajectory class, since `G4Trajectory` uses `G4Allocator`.

G4THitsMap

G4THitsMap is an alternative to G4THitsCollection. G4THitsMap does not demand G4VHit, but instead any variable which can be mapped with an integer key. Typically the key is a copy number of the volume, and the mapped value could for example be a double, such as the energy deposition in a volume. G4THitsMap is convenient for applications which do not need to output event-by-event data but instead just accumulate them. All the G4VPrimitiveScorer classes discussed in *G4MultiFunctionalDetector and G4VPrimitiveScorer* use G4THitsMap.

G4THitsMap is derived from the G4VHitsCollection abstract base class and all objects of this class are also stored in G4HCofThisEvent at the end of an event. How to access a G4THitsMap object is discussed in the following section (*G4MultiFunctionalDetector and G4VPrimitiveScorer*).

4.4.2 Sensitive detector

G4VSensitiveDetector

G4VSensitiveDetector is an abstract base class which represents a detector. The principal mandate of a sensitive detector is the construction of hit objects using information from steps along a particle track. The ProcessHits() method of G4VSensitiveDetector performs this task using G4Step objects as input. In the case of a "Readout" geometry, objects of the G4TouchableHistory class may be used as an optional input.

Your concrete detector class should be instantiated with the unique name of your detector. The name can be associated with one or more global names with "/" as a delimiter for categorizing your detectors. For example

```
myEMcal = new MyEMcal("/myDet/myCal/myEMcal");
```

where myEMcal is the name of your detector. The detector must be constructed in G4VUserDetectorConstruction::ConstructSDandField() method. It must be assigned to one or more G4LogicalVolume objects to set the sensitivity of these volumes. SUch assignment must be made in the same G4VUserDetectorConstruction::ConstructSDandField() method. The pointer should also be registered to G4SDManager, as described in *G4SDManager*.

G4VSensitiveDetector has three major virtual methods.

ProcessHits() This method is invoked by G4SteppingManager when a step is composed in the G4LogicalVolume which has the pointer to this sensitive detector. The first argument of this method is a G4Step object of the current step. The second argument is a G4TouchableHistory object for the Readout geometry described in the next section. The second argument is NULL if Readout geometry is not assigned to this sensitive detector. In this method, one or more G4VHit objects should be constructed if the current step is meaningful for your detector.

Initialize() This method is invoked at the beginning of each event. The argument of this method is an object of the G4HCofThisEvent class. Hit collections, where hits produced in this particular event are stored, can be associated with the G4HCofThisEvent object in this method. The hit collections associated with the G4HCofThisEvent object during this method can be used for during the event processing digitization.

EndOfEvent() This method is invoked at the end of each event. The argument of this method is the same object as the previous method. Hit collections occasionally created in your sensitive detector can be associated with the G4HCofThisEvent object.

4.4.3 G4SDManager

G4SDManager is the singleton manager class for sensitive detectors.

Activation/inactivation of sensitive detectors

The user interface commands `activate` and `inactivate` are available to control your sensitive detectors. For example:

```
/hits/activate detector_name
/hits/inactivate detector_name
```

where `detector_name` can be the detector name or the category name.

For example, if your EM calorimeter is named

```
/myDet/myCal/myEMcal
/hits/inactivate myCal
```

will inactivate all detectors belonging to the `myCal` category.

Access to the hit collections

Hit collections are accessed for various cases.

- Digitization
- Event filtering in `G4VUserStackingAction`
- "End of event" simple analysis
- Drawing / printing hits

The following is an example of how to access the hit collection of a particular concrete type:

```
G4SDManager* fSDM = G4SDManager::GetSDMpointer();
G4RunManager* fRM = G4RunManager::GetRunManager();
G4int collectionID = fSDM->GetCollectionID("collection_name");
const G4Event* currentEvent = fRM->GetCurrentEvent();
G4HCofThisEvent* HCofEvent = currentEvent->GetHCofThisEvent();
MyHitsCollection* myCollection = (MyHitsCollection*)(HC0fEvent->GetHC(collectionID));
```

4.4.4 `G4MultiFunctionalDetector` and `G4VPrimitiveScorer`

`G4MultiFunctionalDetector` is a concrete class derived from `G4VSensitiveDetector`. Instead of implementing a user-specific detector class, `G4MultiFunctionalDetector` allows the user to register `G4VPrimitiveScorer` classes to build up the sensitivity. `G4MultiFunctionalDetector` should be instantiated in the users detector construction with its unique name and should be assigned to one or more `G4LogicalVolume`s.

`G4VPrimitiveScorer` is an abstract base class representing a class to be registered to `G4MultiFunctionalDetector` that creates a `G4THitsMap` object of one physics quantity for an event. Geant4 provides many concrete primitive scorer classes listed in *Concrete classes of G4VPrimitiveScorer*, and the user can also implement his/her own primitive scorers. Each primitive scorer object must be instantiated with a name that must be unique among primitive scorers registered in a `G4MultiFunctionalDetector`. Please note that a primitive scorer object must **not** be shared by more than one `G4MultiFunctionalDetector` object.

As mentioned in *Hit*, each `G4VPrimitiveScorer` generates one `G4THitsMap` object per event. The name of the map object is the same as the name of the primitive scorer. Each of the concrete primitive scorers listed in *Concrete classes of G4VPrimitiveScorer* generates a `G4THitsMap<G4double>` that maps a `G4double` value to its key integer number. By default, the key is taken as the copy number of the `G4LogicalVolume` to which `G4MultiFunctionalDetector` is assigned. In case the logical volume is uniquely placed in its mother volume and the mother is replicated, the copy number of its mother volume can be taken by setting the second argument of the `G4VPrimitiveScorer` constructor, "*depth*" to 1, i.e. one level up. Furthermore, in case the key must consider

more than one copy number of a different geometry hierarchy, the user can derive his/her own primitive scorer from the provided concrete class and implement the `GetIndex(G4Step*)` virtual method to return the unique key.

This *example* shows an example of primitive sensitivity class definitions.

Listing 4.15: An example of defining primitive sensitivity classes taken from `RE06DetectorConstruction`.

```
void RE06DetectorConstruction::SetupDetectors()
{
  G4String filterName, particleName;

  G4SDParticleFilter* gammaFilter =
    new G4SDParticleFilter(filterName="gammaFilter",particleName="gamma");
  G4SDParticleFilter* electronFilter =
    new G4SDParticleFilter(filterName="electronFilter",particleName="e-");
  G4SDParticleFilter* positronFilter =
    new G4SDParticleFilter(filterName="positronFilter",particleName="e+");
  G4SDParticleFilter* epFilter = new G4SDParticleFilter(filterName="epFilter");
  epFilter->add(particleName="e-");
  epFilter->add(particleName="e+");


  for(G4int i=0;i<3;i++)
  {
   for(G4int j=0;j<2;j++)
   {
    // Loop counter j = 0 : absorber
    //                = 1 : gap
    G4String detName = calName[i];
    if(j==0)
    { detName += "_abs"; }
    else
    { detName += "_gap"; }
    G4MultiFunctionalDetector* det = new G4MultiFunctionalDetector(detName);

    //  The second argument in each primitive means the "level" of geometrical hierarchy,
    // the copy number of that level is used as the key of the G4THitsMap.
    //  For absorber (j = 0), the copy number of its own physical volume is used.
    //  For gap (j = 1), the copy number of its mother physical volume is used, since there
    // is only one physical volume of gap is placed with respect to its mother.
    G4VPrimitiveScorer* primitive;
    primitive = new G4PSEnergyDeposit("eDep",j);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSNofSecondary("nGamma",j);
    primitive->SetFilter(gammaFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSNofSecondary("nElectron",j);
    primitive->SetFilter(electronFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSNofSecondary("nPositron",j);
    primitive->SetFilter(positronFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSMinKinEAtGeneration("minEkinGamma",j);
    primitive->SetFilter(gammaFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSMinKinEAtGeneration("minEkinElectron",j);
    primitive->SetFilter(electronFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSMinKinEAtGeneration("minEkinPositron",j);
    primitive->SetFilter(positronFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSTrackLength("trackLength",j);
    primitive->SetFilter(epFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSNofStep("nStep",j);
    primitive->SetFilter(epFilter);
    det->RegisterPrimitive(primitive);
```

```
    G4SDManager::GetSDMpointer()->AddNewDetector(det);
    if(j==0)
    { layerLogical[i]->SetSensitiveDetector(det); }
    else
    { gapLogical[i]->SetSensitiveDetector(det); }
   }
  }
}
```

Each `G4THitsMap` object can be accessed from `G4HCofThisEvent` with a unique collection ID number. This ID number can be obtained from `G4SDManager::GetCollectionID()` with a name of `G4MultiFunctionalDetector` and `G4VPrimitiveScorer` connected with a slush ("/"). `G4THitsMap` has a [] operator taking the key value as an argument and returning **the pointer** of the value. Please note that the [] operator returns **the pointer** of the value. If you get zero from the [] operator, it does **not** mean the value is zero, but that the provided key does not exist. The value itself is accessible with an astarisk ("*"). It is advised to check the validity of the returned pointer before accessing the value. `G4THitsMap` also has a += operator in order to accumulate event data into run data. The *example* shows the use of `G4THitsMap`.

Listing 4.16: An example of accessing to `G4THitsMap` objects.

```
#include "ExN07Run.hh"
#include "G4Event.hh"
#include "G4HCofThisEvent.hh"
#include "G4SDManager.hh"

ExN07Run::ExN07Run()
{
  G4String detName[6] = {"Calor-A_abs","Calor-A_gap","Calor-B_abs","Calor-B_gap",
                         "Calor-C_abs","Calor-C_gap"};
  G4String primNameSum[6] = {"eDep","nGamma","nElectron","nPositron","trackLength","nStep"};
  G4String primNameMin[3] = {"minEkinGamma","minEkinElectron","minEkinPositron"};

  G4SDManager* SDMan = G4SDManager::GetSDMpointer();
  G4String fullName;
  for(size_t i=0;i<6;i++)
  {
    for(size_t j=0;j<6;j++)
    {
      fullName = detName[i]+"/"+primNameSum[j];
      colIDSum[i][j] = SDMan->GetCollectionID(fullName);
    }
    for(size_t k=0;k<3;k++)
    {
      fullName = detName[i]+"/"+primNameMin[k];
      colIDMin[i][k] = SDMan->GetCollectionID(fullName);
    }
  }
}


void ExN07Run::RecordEvent(const G4Event* evt)
{
  G4HCofThisEvent* HCE = evt->GetHCofThisEvent();
  if(!HCE) return;
  numberOfEvent++;
  for(size_t i=0;i<6;i++)
  {
    for(size_t j=0;j<6;j++)
    {
      G4THitsMap<G4double>* evtMap = (G4THitsMap<G4double>*)(HCE->GetHC(colIDSum[i][j]));
      mapSum[i][j] += *evtMap;
    }
    for(size_t k=0;k<3;k++)
    {
```

```
      G4THitsMap<G4double>* evtMap = (G4THitsMap<G4double>*)(HCE->GetHC(colIDMin[i][k]));
      std::map<G4int,G4double*>::iterator itr = evtMap->GetMap()->begin();
      for(; itr != evtMap->GetMap()->end(); itr++)
      {
        G4int key = (itr->first);
        G4double val = *(itr->second);
        G4double* mapP = mapMin[i][k][key];
        if( mapP && (val>*mapP) ) continue;
        mapMin[i][k].set(key,val);
      }
    }
  }
}
```

4.4.5 Concrete classes of `G4VPrimitiveScorer`

With Geant4 version 8.0, several concrete primitive scorer classes are provided, all of which are derived from the `G4VPrimitiveScorer` abstract base class and which are to be registered to `G4MultiFunctionalDetector`. Each of them contains one `G4THitsMap` object and scores a simple double value for each key.

Track length scorers

`G4PSTrackLength` The track length is defined as the sum of step lengths of the particles inside the cell. By default, the track weight is not taken into account, but could be used as a multiplier of each step length if the `Weighted()` method of this class object is invoked.

`G4PSPassageTrackLength` The passage track length is the same as the track length in `G4PSTrackLength`, except that only tracks which pass through the volume are taken into account. It means newly-generated or stopped tracks inside the cell are excluded from the calculation. By default, the track weight is not taken into account, but could be used as a multiplier of each step length if the `Weighted()` method of this class object is invoked.

Deposited energy scorers

`G4PSEnergyDeposit` This scorer stores a sum of particles' energy deposits at each step in the cell. The particle weight is multiplied at each step.

`G4PSDoseDeposit` In some cases, dose is a more convenient way to evaluate the effect of energy deposit in a cell than simple deposited energy. The dose deposit is defined by the sum of energy deposits at each step in a cell divided by the mass of the cell. The mass is calculated from the density and volume of the cell taken from the methods of `G4VSolid` and `G4LogicalVolume`. The particle weight is multiplied at each step.

Current and flux scorers

There are two different definitions of a particle's flow for a given geometry. One is a current and the other is a flux. In our scorers, the current is simply defined as the number of particles (with the particle's weight) at a certain surface or volume, while the flux takes the particle's injection angle to the geometry into account. The current and flux are usually defined at a surface, but volume current and volume flux are also provided.

`G4PSFlatSurfaceCurrent` Flat surface current is a surface based scorer. The present implementation is limited to scoring only at the -Z surface of a `G4Box` solid. The quantity is defined by the number of tracks that reach the surface. The user must choose a direction of the particle to be scored. The choices are fCurrent_In, fCurrent_Out, or fCurrent_InOut, one of which must be entered as the second argument of the constructor. Here, fCurrent_In scores incoming particles to the cell, while fCurrent_Out scores only outgoing particles from the

cell. fCurrent_InOut scores both directions. The current is multiplied by particle weight and is normalized for a unit area.

G4PSSphereSurfaceCurrent Sphere surface current is a surface based scorer, and similar to the G4PSFlatSurfaceCurrent. The only difference is that the surface is defined at the **inner surface** of a G4Sphere solid.

G4PSPassageCurrent Passage current is a volume-based scorer. The current is defined by the number of tracks that pass through the volume. A particle weight is applied at the exit point. A passage current is defined for a volume.

G4PSFlatSurfaceFlux Flat surface flux is a surface based flux scorer. The surface flux is defined by the number of tracks that reach the surface. The expression of surface flux is given by the sum of W/cos(t)/A, where W, t and A represent particle weight, injection angle of particle with respect to the surface normal, and area of the surface. The user must enter one of the particle directions, fFlux_In, fFlux_Out, or fFlux_InOut in the constructor. Here, fFlux_In scores incoming particles to the cell, while fFlux_Out scores outgoing particles from the cell. fFlux_InOut scores both directions.

G4PSCellFlux Cell flux is a volume based flux scorer. The cell flux is defined by a track length (L) of the particle inside a volume divided by the volume (V) of this cell. The track length is calculated by a sum of the step lengths in the cell. The expression for cell flux is given by the sum of (W*L)/V, where W is a particle weight, and is multiplied by the track length at each step.

G4PSPassageCellFlux Passage cell flux is a volume based scorer similar to `G4PSCellFlux`. The only difference is that tracks which pass through a cell are taken into account. It means generated or stopped tracks inside the volume are excluded from the calculation.

Other scorers

G4PSMinKinEAtGeneration This scorer records the minimum kinetic energy of secondary particles at their production point in the volume in an event. This primitive scorer does not integrate the quantity, but records the minimum quantity.

G4PSNofSecondary This class scores the number of secondary particles generated in the volume. The weight of the secondary track is taken into account.

G4PSNofStep This class scores the number of steps in the cell. A particle weight is not applied.

G4PSCellCharge This class scored the total charge of particles which has stoped in the volume.

4.4.6 `G4VSDFilter` and its derived classes

`G4VSDFilter` is an abstract class that represents a track filter to be associated with `G4VSensitiveDetector` or `G4VPrimitiveScorer`. It defines a virtual method

```
G4bool Accept(const G4Step*)
```

that should return *true* if this particular step should be scored by the `G4VSensitiveDetector` or `G4VPrimitiveScorer`.

While the user can implement his/her own filter class, Geant4 version 8.0 provides the following concrete filter classes:

G4SDChargedFilter All charged particles are accepted.

G4SDNeutralFilter All neutral particles are accepted.

G4SDParticleFilter Particle species which are registered to this filter object by `Add("particle_name")` are accepted. More than one species can be registered.

G4SDKineticEnergyFilter A track with kinetic energy greater than or equal to EKmin and smaller than EKmin is accepted. EKmin and EKmax should be defined as arguments of the constructor. The default values of EKmin and EKmax are zero and DBL_MAX.

G4SDParticleWithEnergyFilter Combination of `G4SDParticleFilter` and `G4SDParticleWithEnergyFilter`.

The use of the G4SDParticleFilter class is demonstrated in *example-hits-2*, where filters which accept gamma, electron, positron and electron/positron are defined.

4.4.7 Muiltiple sensitive detectors associated to a single logical-volume

From Geant4 Version 10.3 it is possible to attach multiple sensitive detectors to a single geometrical element. This is achieved via the use of a special proxy class, to which multiple child sensitive detectors are attached: G4MultiSensitiveDetector. The kernel still sees a single sensitive detector associated to any given logical-volume, but the proxy will dispatch the calls from kernel to all the attached child sensitive detectors.

When using the G4VUserDetectorConstruction::SetSensitiveDetector(...) utility method the handling of multiple sensitive detectors is done automatically. Multiple calls to the method passing the same logical volume will trigger the creation and setup of an instance of G4MultiSensitiveDetector.

For more complex use cases it may be necessary to manually instantiate and setup an instance of G4MultiSensitiveDetector. For this advanced use case you can refer to the implementation of the G4VUserDetectorConstruction::SetSensitiveDetector(G4LogicalVolume* logVol, G4VSensitiveDetector* aSD) utility method.

Listing 4.17: An example of the use of G4MultiSensitiveDetector.

```
void MyDetectorConstruction::ConstructSDandField()
{
  auto sdman = G4SDManager::GetSDMpointer();
  //...
  auto mySD = new mySD1("/SD1");
  sdman->AddNewDetector(mySD);//Note we explictly add the SD to the manager
  SetSensitiveDetector("LogVolName",mySD);
  auto mySD2 = new MySD2("/SD2");
  sdman->AddNewDetector(mySD2);
  //This will trigger atuomatic creation and setup of proxy
  SetSensitiveDetector("LogVolName",mySD2);
  //...
}
```

4.5 Digitization

4.5.1 Digi

A hit is created by a sensitive detector when a step goes through it. Thus, the sensitive detector is associated to the corresponding G4LogicalVolume object(s). On the other hand, a digit is created using information of hits and/or other digits by a digitizer module. The digitizer module is not associated with any volume, and you have to implicitly invoke the Digitize() method of your concrete G4VDigitizerModule class.

Typical usages of digitizer module include:

- simulate ADC and/or TDC
- simulate readout scheme
- generate raw data
- simulate trigger logics
- simulate pile up

G4VDigi

G4VDigi is an abstract base class which represents a digit. You have to inherit this base class and derive your own

concrete digit class(es). The member data of your concrete digit class should be defined by yourself. `G4VDigi` has two virtual methods, `Draw()` and `Print()`.

As with `G4VHit`, authors of subclasses must declare templated `G4Allocators` for their digit class. They must also implement *operator new()* and *operator delete()* which use these allocators.

G4TDigiCollection

`G4TDigiCollection` is a template class for digits collections, which is derived from the abstract base class `G4VDigiCollection`. `G4Event` has a `G4DCofThisEvent` object, which is a container class of collections of digits. The usages of `G4VDigi` and `G4TDigiCollection` are almost the same as `G4VHit` and `G4THitsCollection`, respectively, explained in the previous section.

As with `G4THitsCollection`, authors of subclasses must declare templated `G4Allocators` for their collection class. They must also implement *operator new()* and *operator delete()* which use these allocators.

4.5.2 Digitizer module

G4VDigitizerModule

`G4VDigitizerModule` is an abstract base class which represents a digitizer module. It has a pure virtual method, `Digitize()`. A concrete digitizer module must have an implementation of this virtual method. The Geant4 kernel classes do not have a "built-in" invocation to the `Digitize()` method. You have to implement your code to invoke this method of your digitizer module.

In the `Digitize()` method, you construct your `G4VDigi` concrete class objects and store them to your `G4TDigiCollection` concrete class object(s). Your collection(s) should be associated with the `G4DCofThisEvent` object.

G4DigiManager

`G4DigiManager` is the singleton manager class of the digitizer modules. All of your concrete digitizer modules should be registered to `G4DigiManager` with their unique names.

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
MyDigitizer * myDM = new MyDigitizer( "/myDet/myCal/myEMdigiMod" );
fDM->AddNewModule(myDM);
```

Your concrete digitizer module can be accessed from your code using the unique module name.

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
MyDigitizer * myDM = fDM->FindDigitizerModule( "/myDet/myCal/myEMdigiMod" );
myDM->Digitize();
```

Also, `G4DigiManager` has a `Digitize()` method which takes the unique module name.

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
MyDigitizer * myDM = fDM->Digitize( "/myDet/myCal/myEMdigiMod" );
```

How to get hitsCollection and/or digiCollection

`G4DigiManager` has the following methods to access to the hits or digi collections of the currently processing event or of previous events.

First, you have to get the collection ID number of the hits or digits collection.

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
G4int myHitsCollID = fDM->GetHitsCollectionID( "hits_collection_name" );
G4int myDigiCollID = fDM->GetDigiCollectionID( "digi_collection_name" );
```

Then, you can get the pointer to your concrete `G4THitsCollection` object or `G4TDigiCollection` object of the currently processing event.

```
MyHitsCollection * HC = fDM->GetHitsCollection( myHitsCollID );
MyDigiCollection * DC = fDM->GetDigiCollection( myDigiCollID );
```

In case you want to access to the hits or digits collection of previous events, add the second argument.

```
MyHitsCollection * HC = fDM->GetHitsCollection( myHitsCollID, n );
MyDigiCollection * DC = fDM->GetDigiCollection( myDigiCollID, n );
```

where, `n` indicates the hits or digits collection of the n^{th} previous event.

4.6 Object Persistency

4.6.1 Persistency in Geant4

Object persistency is provided by Geant4 as an optional category, so that the user may run Geant4 with or without an object database management system (ODBMS).

When a usual (transient) object is created in C++, the object is placed onto the application heap and it ceases to exist when the application terminates. Persistent objects, on the other hand, live beyond the termination of the application process and may then be accessed by other processes (in some cases, by processes on other machines).

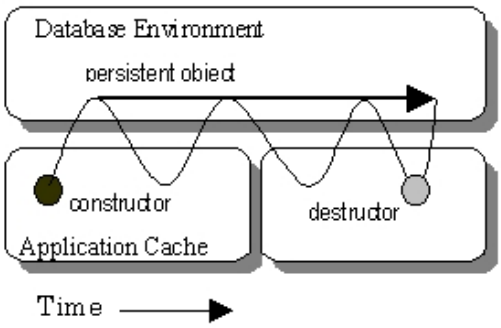


Fig. 4.8: Persistent object.

C++ does not have, as an intrinsic part of the language, the ability to store and retrieve persistent objects. Geant4 provides an abstract framework for persistency of hits, digits and events.

Two examples demonstrating an implementation of object persistency using one of the tools accessible through the available interface, is provided in `examples/extended/persistency`.

4.6.2 Using Root-I/O for persistency of Geant4 objects

Object persistency of Geant4 objects is also possible by using the Root-I/O features through Root (since release `v6.04/08`).

The basic steps that one needs to do in order to use Root-I/O for arbitrary C++ classes is:

1. Generate the dictionary for the given classes from Root (this usually is done by adding the appropriate command to the makefile)
2. Add initialization of Root-I/O and loading of the generated dictionary for the given classes in the appropriate part of the code
3. Whenever the objects to be persistified are available, call the `WriteObject` method of `TFile` with the pointer to the appropriate object as argument (usually it is some sort of container, like `std::vector` containing the collection of objects to be persistified)

The two examples (`P01` and `P02`) provided in `examples/extended/persistency` demonstrate how to perform object persistency using the Root-I/O mechanism for storing hits and geometry description.

4.7 Parallel Geometries

4.7.1 A parallel world

Occasionally, it is not straightforward to define geometries for sensitive detectors, importance geometries or envelopes for shower parameterization to be coherently assigned to volumes in the tracking (mass) geometry. The parallel navigation functionality introduced since release 8.2 of Geant4, allows the user to define more than one world simultaneously. The `G4CoupledTransportation` process will see all worlds simultaneously; steps will be limited by every boundaries of the mass and parallel geometries. `G4Transportation` is automatically replaced `G4CoupledTransportation`.

In a parallel world, the user can define volumes in arbitrary manner with sensitivity, regions, shower parameterization setups, and/or importance weight for biasing. Volumes in different worlds may overlap.

Any kind of `G4VSensitiveDetector` object can be defined in volumes in a parallel world, exactly at the same manner for the mass geometry. `G4Step` object given as an argument of `ProcessHit()` method contains geometrical information of the associated world.

Here are restrictions to be considered for the parallel geometry:

- Production thresholds and EM field are used only from the mass geometry. Even if such *physical* quantities are defined in a parallel world, they do not affect to the simulation.
- Although all worlds will be comprehensively taken care by the `G4CoupledTransportation` process for the navigation, each parallel world must have its own unique object of `G4ParallelWorldProcess` process.
- Volumes in a parallel world may have materials. Such materials overwrite the materials defined in the mass geometry if the "`layered mass geometry`" switch of the `G4ParallelWorldProcess` constructor is set.

4.7.2 Defining a parallel world

A parallel world should be defined in the `Construct()` virtual method of the user's class derived from the abstract base class `G4VUserParallelWorld`. If needed, sensitive detectors must be defined in the `ConstructSD()` method of the same derived class. Please note that EM field cannot be defined in a parallel world.

Listing 4.18: An example header file of a concrete user parallel world class.

```
#ifndef MyParallelWorld_h
#define MyParallelWorld_h 1

#include "globals.hh"
#include "G4VUserParallelWorld.hh"

class MyParallelWorld : public G4VUserParallelWorld
```

```
{
  public:
    MyParallelWorld(G4String worldName);
    virtual ~MyParallelWorld();

  public:
    virtual void Construct();
    virtual void ConstructSD();
};

#endif
```

A parallel world must have its unique name, which should be set to the `G4VUserParallelWorld` base class as an argument of the base class constructor.

The world physical volume of the parallel world is provided by the `G4RunManager` as a clone of the mass geometry. In the `Construct()` virtual method of the user's class, the pointer to this cloned world physical volume is available through the `GetWorld()` method defined in the base class. The user should fill the volumes in the parallel world by using this provided world volume. For a logical volume in a parallel world, the material pointer can be `nullptr`. Even if specified a valid material pointer, unless `"layered mass geometry"` switch of the `G4ParallelWorldProcess` constructor is set, it will not be taken into account by any physics process.

Listing 4.19: An example source code of a concrete user parallel world class.

```
#include "MyParallelWorld.hh"
#include "G4LogicalVolume.hh"
#include "G4VPhysicalVolume.hh"
#include "G4Box.hh"
#include "G4PVPlacement.hh"

MyParallelWorld::MyParallelWorld(G4String worldName)
:G4VUserParallelWorld(worldName)
{;}

MyParallelWorld::~MyParallelWorld()
{;}

void MyParallelWorld::Construct()
{
  G4VPhysicalVolume* ghostWorld = GetWorld();
  G4LogicalVolume* worldLogical = ghostWorld->GetLogicalVolume();

  // place volumes in the parallel world here. For example ...
  //
  G4Box * ghostSolid = new G4Box("GhostdBox", 60.*cm, 60.*cm, 60.*cm);
  G4LogicalVolume * ghostLogical
        = new G4LogicalVolume(ghostSolid, 0, "GhostLogical", 0, 0, 0);
  new G4PVPlacement(0, G4ThreeVector(), ghostLogical,
                    "GhostPhysical", worldLogical, 0, 0);
}
```

In case the user needs to define more than one parallel worlds, each of them must be implemented through its dedicated class. Each parallel world should be registered to the mass geometry class using the method `RegisterParallelWorld()` available through the class `G4VUserDetectorConstruction`. The registration must be done before the mass world is registed to the `G4RunManager`.

Listing 4.20: Typical implementation in the `main()` to define a parallel world.

```
// RunManager construction
//
G4RunManager* runManager = new G4RunManager;

// mass world
//
```

```
MyDetectorConstruction* massWorld = new MyDetectorConstruction;

// parallel world
//
G4String paraWorldName = "ParallelWorld";
massWorld->RegisterParallelWorld(new MyParallelWorld(paraWorldName));

// set mass world to run manager
//
runManager->SetUserInitialization(massWorld);

// physics list
//
G4VModularPhysicsList* physicsList = new FTFP_BERT;
physicsList->RegisterPhysics(new G4ParallelWorldPhysics(paraWorldName));
runManager->SetUserInitialization(physicsList);
```

4.7.3 Layered mass geometry

If "`layered mass geometry`" switch of the `G4ParallelWorldProcess` constructor is set, that parallel world is conceptually layered on top of the mass geometry. If more than one parallel worlds are defined, later-defined world comes on top of others. A track will see the material of the top layer, if it is `nullptr`, then one layer beneath. Thus, user has to make sure volumes in a parallel world should have `nullptr` as their materials except for volumes he/she really wants to overwrite.

Listing 4.21: Typical implementation in the `main()` to define a layered mass geometry.

```
// RunManager construction
//
G4RunManager* runManager = new G4RunManager;

// mass world
//
MyDetectorConstruction* massWorld = new MyDetectorConstruction;

// parallel world
//
G4String paraWorldName = "ParallelWorld";
massWorld->RegisterParallelWorld(new MyParallelWorld(paraWorldName));

// set mass world to run manager
//
runManager->SetUserInitialization(massWorld);

// physics list
//
G4VModularPhysicsList* physicsList = new FTFP_BERT;
physicsList->RegisterPhysics(new G4ParallelWorldPhysics(paraWorldName,true));
runManager->SetUserInitialization(physicsList);
```

For an information to advanced users, instead of using `G4ParallelWorldPhysics` physics constructor, once can define `G4ParallelWorldProcess` in his/her physics list and assign it only to some selected kind of particle types. In this case, this parallel world will be seen only by these kinds of particles.

4.8 Command-based scoring

4.8.1 Introduction

Command-based scoring in Geant4 utilizes parallel navigation in a parallel world volume as descibed in the previous sections. Through interactive commands, the user can define :

- A parallel world for scoring and three-dimensional mesh in it
- Arbitrary number of physics quantities to be scored and filters

After scoring (i.e. a run), the user can visualize the score and dump scores into a file. All available UI commands are listed in List of built-in commands.

Command-based scoring is an optional functionality and the user has to explicity define its use in the `main()`. To do this, the method `G4ScoringManager::GetScoringManager()` must be invoked right after the instantiation of `G4RunManager`. The scoring manager is a singleton object, and the pointer accessed above should not be deleted by the user.

Listing 4.22: A user `main()` to use the command-based scoring

```
#include "G4RunManager.hh"
#include "G4ScoringManager.hh"

int main(int argc,char** argv)
{
 // Construct the run manager
 G4RunManager * runManager = new G4RunManager;

 // Activate command-based scorer
 G4ScoringManager::GetScoringManager();

 ...
}
```

4.8.2 Defining a scoring mesh

To define a scoring mesh, the user has to specify the following.

- Shape and name of the 3D scoring mesh. Currently, box is the only available shape.
- Size of the scoring mesh. Mesh size must be specified as “half width” similar to the arguments of `G4Box` .
- Number of bins for each axes. Note that too high number causes immense memory consumption.
- Optionally, position and rotation of the mesh. If not specified, the mesh is positioned at the center of the world volume without rotation.

For a scoring mesh the user can have arbitrary number of quantities to be scored for each cell of the mesh. For each scoring quantity, the use can set one filter. Please note that `/score/filter` affects on the preceding scorer. Names of scorers and filters must be unique for the mesh. It is possible to define more than one scorer of same kind with different names and, likely, with different filters.

Defining a scoring mesh and scores in the mesh should terminate with the `/score/close` command. The following sample UI commands define a scoring mesh named `boxMesh_1`, size of which is 2 m * 2 m * 2 m, and sliced into 30 cells along each axes. For each cell energy deposition, number of steps of gamma, number of steps of electron and number of steps of positron are scored.

Listing 4.23: UI commands to define a scoring mesh and scorers

```
#
# define scoring mesh
#
```

```
/score/create/boxMesh boxMesh_1
/score/mesh/boxSize 100. 100. 100. cm
/score/mesh/nBin 30 30 30
#
# define scorers and filters
#
/score/quantity/energyDeposit eDep
/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+
#
/score/close
#
```

4.8.3 Drawing scores

Once scores are filled, it is possible to visualize the scores. The score is drawn on top of the mass geometry with the current visualization settings.

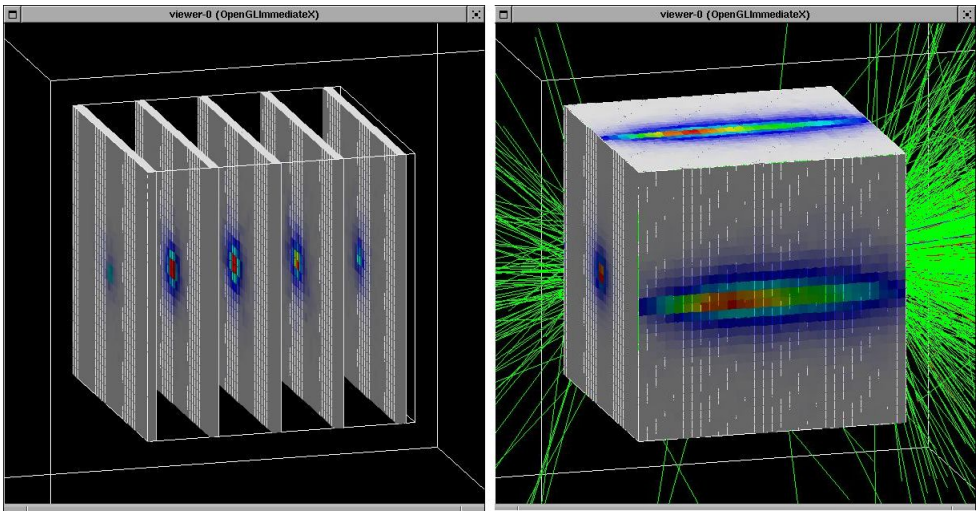


Fig. 4.9: Drawing scores in slices (left) and projection (right).

Scored data can be visualized using the commands `/score/drawProjection` and `/score/drawColumn`. For details, see examples/extended/runAndEvent/RE03.

By default, entries are linearly mapped to colors (gray - blue - green - red). This color mapping is implemented in `G4DefaultLinearColorMap` class, and registered to `G4ScoringManager` with the color map name `"defaultLinearColorMap"`. The user may alternate color map by implementing a customised color map class derived from `G4VScoreColorMap` and register it to `G4ScoringManager`. Then, for each `draw` command, one can specify the preferred color map.

4.8.4 Writing scores to a file

It is possible to dump a score in a mesh (`/score/dumpQuantityToFile` command) or all scores in a mesh (`/score/dumpAllQuantitiesToFile` command) to a file. The default file format is the simple CSV. To alternate

the file format, one should overwrite `G4VScoreWriter` class and register it to `G4ScoringManager`. The scoring manager takes ownership of the registered writer, and will delete it at the end of the job.

Please refer to `/examples/extended/runAndEvent/RE03` for details.

CHAPTER

FIVE

TRACKING AND PHYSICS

5.1 Tracking

5.1.1 Basic Concepts

Philosophy of Tracking

All Geant4 processes, including the transportation of particles, are treated generically. In spite of the name "*tracking*", particles are not *transported* in the tracking category. `G4TrackingManager` is an interface class which brokers transactions between the event, track and tracking categories. An instance of this class handles the message passing between the upper hierarchical object, which is the event manager, and lower hierarchical objects in the tracking category. The event manager is a singleton instance of the `G4EventManager` class.

The tracking manager receives a track from the event manager and takes the actions required to finish tracking it. `G4TrackingManager` aggregates the pointers to `G4SteppingManager`, `G4Trajectory` and `G4UserTrackingAction`. Also there is a "use" relation to `G4Track` and `G4Step`.

`G4SteppingManager` plays an essential role in tracking the particle. It takes care of all message passing between objects in the different categories relevant to transporting a particle (for example, geometry and interactions in matter). Its public method `Stepping()` steers the stepping of the particle. The algorithm to handle one step is given below.

1. If the particle stop (i.e. zero kinetic energy), each active atRest process proposes a step length in time based on the interaction it describes. And the process proposing the smallest step length will be invoked.
2. Each active discrete or continuous process must propose a step length based on the interaction it describes. The smallest of these step lengths is taken.
3. The geometry navigator calculates "Safety", the distance to the next volume boundary. If the minimum physical-step-length from the processes is shorter than "Safety", the physical-step-length is selected as the next step length. In this case, no further geometrical calculations will be performed.
4. If the minimum physical-step-length from the processes is longer than "Safety", the distance to the next boundary is re-calculated.
5. The smaller of the minimum physical-step-length and the geometric step length is taken.
6. All active continuous processes are invoked. Note that the particle's kinetic energy will be updated only after all invoked processes have completed. The change in kinetic energy will be the sum of the contributions from these processes.
7. The current track properties are updated before discrete processes are invoked. In the same time, the secondary particles created by processes are stored in SecondaryList. The updated properties are:
    - the kinetic energy of the current track particle (note that 'sumEnergyChange' is the sum of the energy difference before and after each process invocation)
    - position and time
8. The kinetic energy of the particle is checked to see whether or not it has been terminated by a continuous process. If the kinetic energy goes down to zero, atRest processes will be applied at the next step if applicable.
9. The discrete process is invoked. After the invocation,
    - the energy, position and time of the current track particle are updated, and

    - the secondaries are stored in SecondaryList.
10. The track is checked to see whether or not it has been terminated by the discrete process.
11. "Safety" is updated.
12. If the step was limited by the volume boundary, push the particle into the next volume.
13. Handle hit information.
14. Invoke the user intervention `G4UserSteppingAction`.
15. Save data to Trajectory.
16. Update the mean free paths of the discrete processes.
17. If the parent particle is still alive, reset the maximum interaction length of the discrete process which has occurred.
18. One step completed.

What is a Process?

Only processes can change information of `G4Track` and add secondary tracks via `ParticleChange`. `G4VProcess` is a base class of all processes and it has 3 kinds of `DoIt` and `GetPhysicalInteraction` methods in order to describe interactions generically. If a user want to modify information of `G4Track`, he (or she) SHOULD create a special process for the purpose and register the process to the particle.

What is a Track?

`G4Track` keeps 'current' information of the particle. (i.e. energy,momentum, position ,time and so on) and has 'static' information (i.e. mass, charge, life and so on) also. Note that `G4Track` keeps information at the beginning of the step while the `AlongStepDoIt`s are being invoked for the step in progress.After finishing all `AlongStepDoIt`s, `G4Track` is updated. On the other hand, `G4Track` is updated after each invocation of a `PostStepDoIt`.

What is a Step?

`G4Step` stores the transient information of a step. This includes the two endpoints of the step, `PreStepPoint` and `PostStepPoint`, which contain the points' coordinates and the volumes containing the points. `G4Step` also stores the change in track properties between the two points. These properties, such as energy and momentum, are updated as the various active processes are invoked.

What is a ParticleChange?

Processes do NOT change any information of `G4Track` directly in their `DoIt`. Instead, they proposes changes as a result of interactions by using `ParticleChange`. After each `DoIt`, `ParticleChange` updates `PostStepPoint` based on proposed changes. Then, `G4Track` is updated after finishing all `AlongStepDoIt`s and after each `PostStepDoIt`.

5.1.2 Access to Track and Step Information

How to Get Track Information

Track information may be accessed by invoking various `Get` methods provided in the `G4Track` class. For details, see the G4Track.hh header file in `$G4INCLUDE`. Typical information available includes:

- (x,y,z)
- Global time (time since the event was created)
- Local time (time since the track was created)
- Proper time (time in its rest frame since the track was created)
- Momentum direction (unit vector)
- Kinetic energy
- Accumulated geometrical track length
- Accumulated true track length
- Pointer to dynamic particle
- Pointer to physical volume
- Track ID number

- Track ID number of the parent
- Current step number
- Track status
- (x,y,z) at the start point (vertex position) of the track
- Momentum direction at the start point (vertex position) of the track
- Kinetic energy at the start point (vertex position) of the track
- Pointer to the process which created the current track

How to Get Step Information

Step and step-point information can be retrieved by invoking various `Get` methods provided in the `G4Step`/`G4StepPoint` classes..

Information in G4Step includes:

- Pointers to `PreStep` and `PostStepPoint`
- Geometrical step length (step length before the correction of multiple scattering)
- True step length (step length after the correction of multiple scattering)
- Increment of position and time between `PreStepPoint` and `PostStepPoint`
- Increment of momentum and energy between `PreStepPoint` and `PostStepPoint`. (Note: to get the energy deposited in the step, you cannot use this 'Delta energy'. You have to use 'Total energy deposit' as below.)
- Pointer to `G4Track`
- Total energy deposited during the step - this is the sum of
  - the energy deposited by the energy loss process, and
  - the energy lost by secondaries which have NOT been generated because each of their energies was below the cut threshold
- Energy deposited not by ionization during the step
- Secondary tracks created during tracking for the current track.
  - NOTE: all secondaries are included. NOT only secondaries created in the CURRENT step.

Information in G4StepPoint (`PreStepPoint` and `PostStepPoint`) includes:

- (x, y, z, t)
- (px, py, pz, Ek)
- Momentum direction (unit vector)
- Pointers to physical volumes
- Safety
- Beta, gamma
- Polarization
- Step status
- Pointer to the physics process which defined the current step and its `DoIt` type
- Pointer to the physics process which defined the previous step and its `DoIt` type
- Total track length
- Global time (time since the current event began)
- Local time (time since the current track began)
- Proper time

How to Get "particle change"

Particle change information can be accessed by invoking various `Get` methods provided in the G4ParticleChange class. Typical information available includes:

- final momentum direction of the parent particle
- final kinetic energy of the parent particle
- final position of the parent particle
- final global time of the parent particle
- final proper time of the parent particle

- final polarization of the parent particle
- status of the parent particle (`G4TrackStatus`)
- true step length (this is used by multiple scattering to store the result of the transformation from the geometrical step length to the true step length)
- local energy deposited - this consists of either
  - energy deposited by the energy loss process, or
  - the energy lost by secondaries which have NOT been generated because each of their energies was below the cut threshold.
- number of secondaries particles
- list of secondary particles (list of `G4Track`)

5.1.3 Handling of Secondary Particles

Secondary particles are passed as `G4Tracks` from a physics process to tracking. `G4ParticleChange` provides the following four methods for a physics process:

- `AddSecondary( G4Track* aSecondary )`
- `AddSecondary( G4DynamicParticle* aSecondary )`
- `AddSecondary( G4DynamicParticle* aSecondary, G4ThreeVector position )`
- `AddSecondary( G4DynamicParticle* aSecondary, G4double time)`

In all but the first, the construction of `G4Track` is done in the methods using information given by the arguments.

5.1.4 User Actions

There are two classes which allow the user to intervene in the tracking. These are:

- `G4UserTrackingAction`, and
- `G4UserSteppingAction`.

Each provides methods which allow the user access to the Geant4 kernel at specific points in the tracking.

Note: Users SHOULD NOT (and CAN NOT) change `G4Track` in `UserSteppingAction`. Only the exception is the `TrackStatus`.

Note: Users have to be cautious to implement an unnatural/unphysical action in these user actions. See the section *Killing Tracks in User Actions and Energy Conservation* for more details.

5.1.5 Verbose Outputs

The verbose information output flag can be turned on or off. The amount of information printed about the track/step, from brief to very detailed, can be controlled by the value of the verbose flag, for example,

```
G4UImanager* UI = G4UImanager::GetUIpointer();

UI->ApplyCommand("/tracking/verbose 1");
```

5.1.6 Trajectory and Trajectory Point

G4Trajectory and G4TrajectoryPoint

G4Trajectory and G4TrajectoryPoint are default concrete classes provided by Geant4, which are derived from the G4VTrajectory and G4VTrajectoryPoint base classes, respectively. A G4Trajectory class object is created by G4TrackingManager when a G4Track is passed from the G4EventManager. G4Trajectory has the following data members:

- ID numbers of the track and the track's parent
- particle name, charge, and PDG code
- a collection of G4TrajectoryPoint pointers

G4TrajectoryPoint corresponds to a step point along the path followed by the track. Its position is given by a G4ThreeVector. A G4TrajectoryPoint class object is created in the AppendStep() method of G4Trajectory and this method is invoked by G4TrackingManager at the end of each step. The first point is created when the G4Trajectory is created, thus the first point is the original vertex.

The creation of a trajectory can be controlled by invoking G4TrackingManager::SetStoreTrajectory(G4bool). The UI command */tracking/storeTrajectory _bool_* does the same. The user can set this flag for each individual track from his/her G4UserTrackingAction::PreUserTrackingAction() method.

Note: The user should not create trajectories for secondaries in a shower due to the large amount of memory consumed.

All the created trajectories in an event are stored in G4TrajectoryContainer class object and this object will be kept by G4Event. To draw or print trajectories generated in an event, the user may invoke the DrawTrajectory() or ShowTrajectory() methods of G4VTrajectory, respectively, from his/her G4UserEventAction::EndOfEventAction(). The geometry must be drawn before the trajectory drawing. The color of the drawn trajectory depends on the particle charge:

- negative: red
- neutral: green
- positive: blue

Note: Due to improvements in G4Navigator, a track can execute more than one turn of its spiral trajectory without being broken into smaller steps as long as the trajectory does not cross a geometrical boundary. Thus a drawn trajectory may not be circular.

Customizing trajectory and trajectory point

G4Track and G4Step are transient classes; they are not available at the end of the event. Thus, the concrete classes G4VTrajectory and G4VTrajectoryPoint are the only ones a user may employ for end-of-event analysis or for persistency. As mentioned above, the default classes which Geant4 provides, i.e. G4Trajectory and G4TrajectoryPoint, have only very primitive quantities. The user can customize his/her own trajectory and trajectory point classes by deriving directly from the respective base classes.

To use the customized trajectory, the user must construct a concrete trajectory class object in the G4UserTrackingAction::PreUserTrackingAction() method and make its pointer available to G4TrackingManager by using the SetTrajectory() method. The customized trajectory point class object must be constructed in the AppendStep() method of the user's implementation of the trajectory class. This AppendStep() method will be invoked by G4TrackingManager.

To customize trajectory drawing, the user can override the DrawTrajectory() method in his/her own trajectory class.

When a customized version of G4Trajectory declares any new class variables, *operator new* and *operator delete* must be provided. It is also useful to check that the allocation size in *operator new* is equal to sizeof(G4Trajectory). These two points do not apply to G4VTrajectory because it has no *operator new* or *operator delete*.

5.2 Physics Processes

Physics processes describe how particles interact with a material. Seven major categories of processes are provided by Geant4:

1. electromagnetic,
2. hadronic,
3. decay,
4. photolepton-hadron,
5. optical,
6. parameterization, and
7. transportation.

The generalization and abstraction of physics processes is a key issue in the design of Geant4. All physics processes are treated in the same manner from the tracking point of view. The Geant4 approach enables anyone to create a process and assign it to a particle type. This openness should allow the creation of processes for novel, domain-specific or customised purposes by individuals or groups of users.

Each process has two groups of methods which play an important role in tracking, `GetPhysicalInteractionLength` (GPIL) and `DoIt`. The GPIL method gives the step length from the current space-time point to the next space-time point. It does this by calculating the probability of interaction based on the process's cross section information. At the end of this step the `DoIt` method should be invoked. The `DoIt` method implements the details of the interaction, changing the particle's energy, momentum, direction and position, and producing secondary tracks if required. These changes are recorded as `G4VParticleChange` objects (see *Particle Change*).

G4VProcess

`G4VProcess` is the base class for all physics processes. Each physics process must implement virtual methods of `G4VProcess` which describe the interaction (DoIt) and determine when an interaction should occur (GPIL). In order to accommodate various types of interactions `G4VProcess` provides three `DoIt` methods:

- `G4VParticleChange* AlongStepDoIt( const G4Track& track, const G4Step& stepData )`
  This method is invoked while `G4SteppingManager` is transporting a particle through one step. The corresponding `AlongStepDoIt` for each defined process is applied for every step regardless of which process produces the minimum step length. Each resulting change to the track information is recorded and accumulated in `G4Step`. After all processes have been invoked, changes due to `AlongStepDoIt` are applied to `G4Track`, including the particle relocation and the safety update. Note that after the invocation of `AlongStepDoIt`, the endpoint of the `G4Track` object is in a new volume if the step was limited by a geometric boundary. In order to obtain information about the old volume, `G4Step` must be accessed, since it contains information about both endpoints of a step.
- `G4VParticleChange* PostStepDoIt( const G4Track& track, const G4Step& stepData )`
  This method is invoked at the end point of a step, only if its process has produced the minimum step length, or if the process is forced to occur. `G4Track` will be updated after each invocation of `PostStepDoIt`, in contrast to the `AlongStepDoIt` method.
- `G4VParticleChange* AtRestDoIt( const G4Track& track, const G4Step& stepData )`
  This method is invoked only for stopped particles, and only if its process produced the minimum step length or the process is forced to occur.

For each of the above `DoIt` methods `G4VProcess` provides a corresponding pure virtual GPIL method:

- `G4double PostStepGetPhysicalInteractionLength( const G4Track& track, G4double previousStepSize, G4ForceCondition* condition )`

This method generates the step length allowed by its process. It also provides a flag to force the interaction to occur regardless of its step length.

- `G4double AlongStepGetPhysicalInteractionLength( const G4Track& track, G4double previousStepSize, G4double currentMinimumStep, G4double& proposedSafety, G4GPILSelection* selection )`
  This method generates the step length allowed by its process.
- `G4double AtRestGetPhysicalInteractionLength( const G4Track& track, G4ForceCondition* condition )`
  This method generates the step length in time allowed by its process. It also provides a flag to force the interaction to occur regardless of its step length.

Other pure virtual methods in `G4VProcess` follow:

- `virtual G4bool IsApplicable(const G4ParticleDefinition&)`
  returns true if this process object is applicable to the particle type.
- `virtual void PreparePhysicsTable(const G4ParticleDefinition&)` and
- `virtual void BuildPhysicsTable(const G4ParticleDefinition&)`
  is messaged by the process manager, whenever cross section tables should be prepared and rebuilt due to changing cut-off values. It is not mandatory if the process is not affected by cut-off values.
- `virtual void StartTracking()` and
- `virtual void EndTracking()`
  are messaged by the tracking manager at the beginning and end of tracking the current track.

Other base classes for processes

Specialized processes may be derived from seven additional virtual base classes which are themselves derived from `G4VProcess`. Three of these classes are used for simple processes:

`G4VRestProcess` Processes using only the `AtRestDoIt` method.
example: neutron capture

`G4VDiscreteProcess` Processes using only the `PostStepDoIt` method.
example: compton scattering, hadron inelastic interaction

The other four classes are provided for rather complex processes:

`G4VContinuousDiscreteProcess` Processes using both `AlongStepDoIt` and `PostStepDoIt` methods.
example: transportation, ionisation(energy loss and delta ray)

`G4VRestDiscreteProcess` Processes using both `AtRestDoIt` and `PostStepDoIt` methods.
example: positron annihilation, decay (both in flight and at rest)

`G4VRestContinuousProcess` Processes using both `AtRestDoIt` and `AlongStepDoIt` methods.

`G4VRestContinuousDiscreteProcess` Processes using `AtRestDoIt`, `AlongStepDoIt and` PostStepDoIt methods.

Particle change

`G4VParticleChange` and its descendants are used to store the final state information of the track, including secondary tracks, which has been generated by the `DoIt` methods. The instance of `G4VParticleChange` is the only object whose information is updated by the physics processes, hence it is responsible for updating the step. The stepping manager collects secondary tracks and only sends requests via particle change to update `G4Step`.

`G4VParticleChange` is introduced as an abstract class. It has a minimal set of methods for updating `G4Step` and handling secondaries. A physics process can therefore define its own particle change derived from `G4VParticleChange`. Three pure virtual methods are provided,

- `virtual G4Step* UpdateStepForAtRest( G4Step* step)`,
- `virtual G4Step* UpdateStepForAlongStep( G4Step* step )`, and
- `virtual G4Step* UpdateStepForPostStep( G4Step* step)`,

which correspond to the three `DoIt` methods of `G4VProcess`. Each derived class should implement these methods.

5.2.1 Electromagnetic Interactions

This section summarizes the electromagnetic (EM) physics processes which are provided with Geant4. Extended information are avalable at EM web pages. For details on the implementation of these processes please refer to the Physics Reference Manual.

To use the electromagnetic physics data files are needed. The user should set the environment variable G4LEDATA to the directory with this files. These files are distributed together with Geant4 and can be obtained via Geant4 download web page. For Geant4 version 10.3 G4EMLOW6.50 data set is required.

Electromagnetic Processes

The following is a summary of the electromagnetic processes available in Geant4.

- Photon processes
  - Gamma conversion (also called pair production, class name `G4GammaConversion`)
  - Photo-electric effect (class name `G4PhotoElectricEffect`)
  - Compton scattering (class name `G4ComptonScattering`)
  - Rayleigh scattering (class name `G4RayleighScattering`)
  - Muon pair production (class name `G4GammaConversionToMuons`)
- Electron/positron processes
  - Ionisation and delta ray production (class name `G4eIonisation`)
  - Bremsstrahlung (class name `G4eBremsstrahlung`)
  - e+e- pair production (class name `G4ePairProduction`)
  - Multiple scattering (class name `G4eMultipleScattering`)
  - Positron annihilation into two gammas (class name `G4eplusAnnihilation`)
  - Positron annihilation into two muons (class name `G4AnnihiToMuPair`)
  - Positron annihilation into hadrons (class name `G4eeToHadrons`)
- Muon processes
  - Ionisation and delta ray production (class name `G4MuIonisation`)
  - Bremsstrahlung (class name `G4MuBremsstrahlung`)
  - e+e- pair production (class name `G4MuPairProduction`)
  - Multiple scattering (class name `G4MuMultipleScattering`)
- Hadron/ion processes
  - Ionisation (class name `G4hIonisation`)
  - Ionisation for ions (class name `G4ionIonisation`)
  - Ionisation for heavy exotic particles (class name `G4hhIonisation`)
  - Ionisation for classical magnetic monopole (class name `G4mplIonisation`)
  - Multiple scattering (class name `G4hMultipleScattering`)
  - Bremsstrahlung (class name `G4hBremsstrahlung`)
  - e+e- pair production (class name `G4hPairProduction`)
- Coulomb scattering processes
  - Alternative process for simulation of single Coulomb scattering of all charged particles (class name `G4CoulombScattering`)
  - Alternative process for simulation of single Coulomb scattering of ions (class name `G4ScreenedNuclearRecoil`)
- Processes for simulation of polarized electron and gamma beams
  - Compton scattering of circularly polarized gamma beam on polarized target (class name `G4PolarizedCompton`)
  - Pair production induced by circularly polarized gamma beam (class name `G4PolarizedGammaConversion`)
  - Photo-electric effect induced by circularly polarized gamma beam (class name `G4PolarizedPhotoElectricEffect`)
  - Bremsstrahlung of polarized electrons and positrons (class name `G4ePolarizedBremsstrahlung`)

  - Ionisation of polarized electron and positron beam (class name `G4ePolarizedIonisation`)
  - Annihilation of polarized positrons (class name `G4eplusPolarizedAnnihilation`)
- Processes for simulation of X-rays and optical protons production by charged particles
  - Synchrotron radiation (class name `G4SynchrotronRadiation`)
  - Transition radiation (class name `G4TransitionRadiation`)
  - Cerenkov radiation (class name `G4Cerenkov`)
  - Scintillations (class name `G4Scintillation`)
- The processes described above use physics model classes, which may be combined according to particle energy. It is possible to change the energy range over which different models are valid, and to apply other models specific to particle type, energy range, and G4Region. The following alternative models are available in the standard EM sub-library:
  - Ionisation in thin absorbers (class name `G4PAIModel`)
  - Ionisation in thin absorbers (class name `G4PAIPhotModel`)
  - Ionisation in low-density media (class name `G4BraggIonGasModel`)
  - Ionisation in low-density media (class name `G4BetheBlochIonGasModel`)
  - Multiple scattering (class name `G4UrbanMscModel`)
  - Multiple scattering (class name `G4GoudsmitSaundersonMscModel`)
  - Multiple scattering (class name `G4WentzelVIModel`)
  - Multiple scattering (class name `G4LowEWentzelVIModel`)
  - Single scattering (class name `G4eCoulombScatteringModel`)
  - Single scattering (class name `G4eSingleCoulombScatteringModel`)

It is recommended to use physics constructor classes provided with reference physics lists (in subdirectory `source/physics_lists/constructors/electromagnetic` of the Geant4 source distribution):

- default EM physics, multiple scattering is simulated with "UseSafety" type of step limitation by combined `G4WentzelVIModel` and `G4eCoulombScatteringModel` for all particle types, for of e+- below 100 MeV `G4UrbanMscModel` is used, physics tables are built from 100 eV to 100 TeV, 7 bins per energy decade of physics tables are used (class name `G4EmStandardPhysics`)
- optional EM physics providing fast but less acurate electron transport due to "Simple" method of step limitation by multiple scattering, reduced step limitation by ionisation process and enabled "ApplyCuts" option (class name `G4EmStandardPhysics_option1`)
- optional EM physics providing fast but less acurate electron transport due to "Simple" method of step limitation by multiple scattering and reduced step limitation by ionisation process, `G4Generator2BS` angular generator for bremsstrahlung (class name `G4EmStandardPhysics_option2`)
- EM physics for simulation with high accuracy due to "UseDistanceToBoundary" multiple scattering step limitation and usage of `G4UrbanMscModel` for all charged particles, reduced *finalRange* parameter of stepping function optimized per particle type, alternative models `G4LivermorePhotoElectricModel` for photoelectric effect, `G4KleinNishinaModel` for Compton scattering, enabled Rayleigh scattering, enabled fluorescence, enabled nuclear stopping, `G4Generator2BS` angular generator for bremsstrahlung, `G4IonParameterisedLossModel` for ion ionisation, 20 bins per energy decade of physics tables, and 10 eV low-energy limit for tables (class name `G4EmStandardPhysics_option3`)
- Combination of EM models for simulation with high accuracy includes "UseDistanceToBoundary" multiple scattering step limitation, `RangeFactor = 0.02`, reduced *finalRange* parameter of stepping function optimized per particle type, enabled Rayleigh scattering, enabled fluorescence, enabled nuclear stopping, enable accurate angular generator for ionisation models, `G4LivermorePhotoElectricModel`, `G4LowEPComptonModel` below 20 MeV, `G4PenelopeGammaConversionModel` below 1 GeV, `G4PenelopeIonisationModel` fro electrons and positrons below 100 keV, `G4IonParameterisedLossModel` for ion ionisation, `G4Generator2BS` angular generator for bremsstrahlung, and 20 bins per energy decade of physics tables, (class name `G4EmStandardPhysics_option4`)
- Models based on Livermore data bases for electrons and gamma, enabled Rayleigh scattering, enabled fluorescence, enabled nuclear stopping, enable accurate angular generator for ionisation models, `G4IonParameterisedLossModel` for ion ionisation, and 20 bins per energy decade of physics tables, (`G4EmLivermorePhysics`);

- Models for simulation of linear polarized gamma based on Livermore data bases for electrons and gamma (G4EmLivermorePolarizedPhysics);
- Models based on Livermore data bases and new model for Compton scattering G4LowEPComptonModel, new low-energy model of multiple scatetring G4LowEWenzelMscModel (G4EmLowEPPhysics);
- Penelope2008 models for electrons, positrons and gamma, enabled Rayleigh scattering, enabled fluorescence, enabled nuclear stopping, enable accurate angular generator for ionisation models, G4IonParameterisedLossModel for ion ionisation, and 20 bins per energy decade of physics tables, (G4EmPenelopePhysics);
- Experimental physics with multiple scattering of e+- below 100 MeV simulated by G4GoudsmitSaundersonMscModel is done on top of the default EM physics (G4EmStandardPhysicsGS);
- Experimental physics with multiple scattering of e+- below 100 MeV simulated by a combination of G4WentzelVIModel and G4eCoulombScatteringModel is done on top of the default EM physics (G4EmStandardPhysicsGS);
- Experimental physics with single scattering models instead of multiple scattering is done on top of the default EM physics, for all leptons and hadrons G4eCoulombScatteringModel is used, for ions - G4IonCoulombScatteringModel (G4EmStandardPhysicsSS);
- Low-energy Geant4-DNA physics (G4EmDNAPhysics).
- Alternative low-energy Geant4-DNA physics constructors (G4EmDNAPhysics_optionX, where X is 1 to 7).Refer to Geant4-DNA section.

Examples of the registration of these physics constructor and construction of alternative combinations of options are shown in basic, extended and advanced examples, which can be found in the subdirectories examples/basic, examples/extended/electromagnetic and examples/advanced of the Geant4 source distribution. Examples illustrating the use of electromagnetic processes are available as part of the Geant4 release.

Options are available for steering of electromagnetic processes. These options may be invoked either by UI commands or by the new C++ interface class G4EmParameters. The interface G4EmParameters::Instance() is thread safe, EM parameters are shared between threads, and parameters are shared between all EM processes. Parameters may be modified at G4State_PreInit or G4State_Idle states of Geant4. Note, that when any of EM physics constructor is instantiated a default set of EM parameters for this EM physics configuration is defained. So, parameters modification should be applied only after. This class has the following public methods:

- Dump()
- StreamInfo(std::ostream&)
- SetDefaults()
- SetLossFluctuations(G4bool)
- SetBuildCSDARange(G4bool)
- SetLPM(G4bool)
- SetSpline(G4bool)
- SetCutAsFinalRange(G4bool)
- SetApplyCuts(G4bool)
- SetFluo(G4bool val)
- SetBeardenFluoDir(G4bool val)
- SetAuger(G4bool val)
- SetAugerCascade(G4bool val)
- SetPixe(G4bool val)
- SetDeexcitationIgnoreCut(G4bool val)
- SetLateralDisplacement(G4bool val)
- SetMuHadLateralDisplacement(G4bool val)
- SetLatDisplacementBeyondBoundary(G4bool val)
- ActivateAngularGeneratorForIonisation(G4bool val)
- SetUseMottCorrection(G4bool val)
- SetIntegral(G4bool val)
- SetBirksActive(G4bool val)

- SetEmSaturation(G4EmSaturation*)
- SetMinSubRange(G4double)
- SetMinEnergy(G4double)
- SetMaxEnergy(G4double)
- SetMaxEnergyForCSDARange(G4double)
- SetLowestEnergy(G4double)
- SetLowestMuHadEnergy(G4double)
- SetLinearLossLimit(G4double)
- SetBremsstrahlungTh(G4double)
- SetLambdaFactor(G4double)
- SetFactorForAngleLimit(G4double)
- SetMscThetaLimit(G4double)
- SetMscRangeFactor(G4double)
- SetMscMuHadRangeFactor(G4double)
- SetMscGeomFactor(G4double)
- SetMscSkin(G4double)
- SetStepFunction(G4double, G4double)
- SetStepFunctionMuHad(G4double, G4double)
- SetNumberOfBins(G4int)
- SetNumberOfBinsPerDecade(G4int)
- SetVerbose(G4int)
- SetWorkerVerbose(G4int)
- SetMscStepLimitType(G4MscStepLimitType val)
- SetMscMuHadStepLimitType(G4MscStepLimitType val)
- SetNuclearFormFactorType(G4NuclearFormFactorType val)
- SetPIXECrossSectionModel(const G4String&)
- SetPIXEElectronCrossSectionModel(const G4String&)
- AddPAIModel(const G4String& particle, const G4String& region, const G4String& type)
- AddMicroElecModel(const G4String& region)
- AddDNA(const G4String& region, const G4String& type)
- AddMsc(const G4String& region, const G4String& physics_type)
- SetSubCutoff(G4bool, const G4String& region)
- SetDeexActiveRegion(const G4String& region, G4bool, G4bool, G4bool)
- SetProcessBiasingFactor(const G4String& process, G4double, G4bool)
- ActivateForcedInteraction(const G4String& process, const G4String& region, G4double, G4bool)
- ActivateSecondaryBiasing(const G4String& process, const G4String& region, G4double, G4double)

The old interface class `G4EmProcessOptions` is still available but but is strongly recommended not to be used. It will be removed in the next major release.

The corresponding UI command can be accessed in the UI subdirectories "/process/eLoss", "/process/em", and "/process/msc". The following types of step limitation by multiple scattering are available:

- fSimple - simplified step limitation (used in _EMV and _EMX Physics Lists)
- fUseSafety - default
- fUseDistanceToBoundary - advance method of step limitation used in EM examples, required parameter *skin* > *0* , should be used for setup without magnetic field
- fUseSafetyPlus - advanced method may be used with magnetic field

G4EmCalculator is a class which provides access to cross sections and stopping powers. This class can be used anywhere in the user code provided the physics list has already been initialised (G4State_Idle). G4EmCalculator has "Get" methods which can be applied to materials for which physics tables are already built, and "Compute" methods which can be applied to any material defined in the application or existing in the Geant4 internal database. The public methods of this class are:

- GetDEDX(kinEnergy,particle,material,G4Region region=0)

- GetRangeFromRestrictedDEDX(kinEnergy,particle,material,G4Region* region=0)
- GetCSDARange(kinEnergy,particle,material,G4Region* region=0)
- GetRange(kinEnergy,particle,material,G4Region* region=0)
- GetKinEnergy(range,particle,material,G4Region* region=0)
- GetCrosSectionPerVolume(kinEnergy,particle,material,G4Region* region=0)
- GetShellIonisationCrossSectionPerAtom(particle,Z,shell,kinEnergy)
- GetMeanFreePath(kinEnergy,particle,material,G4Region* region=0)
- PrintDEDXTable(particle)
- PrintRangeTable(particle)
- PrintInverseRangeTable(particle)
- ComputeDEDX(kinEnergy,particle,process,material,cut=DBL_MAX)
- ComputeElectronicDEDX(kinEnergy,particle,material,cut=DBL_MAX)
- ComputeNuclearDEDX(kinEnergy,particle,material,cut=DBL_MAX)
- ComputeTotalDEDX(kinEnergy,particle,material,cut=DBL_MAX)
- ComputeCrossSectionPerVolume(kinEnergy,particle,process,material,cut=0)
- ComputeCrossSectionPerAtom(kinEnergy,particle,process,Z,A,cut=0)
- ComputeGammaAttenuationLength(kinEnergy,material)
- ComputeShellIonisationCrossSectionPerAtom(particle,Z,shell,kinEnergy)
- ComputeMeanFreePath(kinEnergy,particle,process,material,cut=0)
- ComputeEnergyCutFromRangeCut(range,particle,material)
- FindParticle(const G4String&)
- FindIon(G4int Z, G4int A)
- FindMaterial(const G4String&)
- FindRegion(const G4String&)
- FindCouple(const G4Material*, const G4Region* region=0)
- SetVerbose(G4int)

For these interfaces, particles, materials, or processes may be pointers or strings with names.

Low Energy Electromagnetic Library

A physical interaction is described by a process class which can handle physics models, described by model classes. The following is a summary of the Low Energy Electromagnetic physics models available in Geant4. Further information is available in the web pages of the Geant4 Low Energy Electromagnetic Physics Working Group, accessible from the Geant4 web site, "who we are" section, then "working groups".

The physics content of these models is documented in the Geant4 Physics Reference Manual. They are based on the Livermore data library, on the ICRU73 data tables or on the Penelope2008 Monte Carlo code. They adopt the same software design as the "standard" Geant4 electromagnetic models.

Examples of the registration of physics constructor with low-energy electromagnetic models are shown in Geant4 extended examples (`examples/extended/electromagnetic` in the Geant4 source distribution). Advanced examples (`examples/advanced` in the Geant4 source distribution) illustrate alternative instantiation of these processes. Both are available as part of the Geant4 release.

Production Cuts

Remember that production cuts for secondaries can be specified as range cuts, which are converted at initialisation time into energy thresholds for secondary gamma, electron, positron and proton production. The cut for proton is applied by elastic scattering processes to aal recoil ions.

A range cut value is set by default to 0.7 mm in Geant4 reference physics lists. This value can be specified in the optional SetCuts() method of the user Physics list or via UI commands. For eg. to set a range cut of 10 micrometers, one can use:

```
/run/setCut  0.01 mm
```

or, for a given particle type (for eg. electron),

```
/run/setCutForAGivenParticle e- 0.01 mm
```

If a range cut equivalent to an energy lower than 990 eV is specified, the energy cut is still set to 990 eV. In order to decrease this value (for eg. down to 250 eV, in order to simulate low energy emission lines of the fluorescence spectrum), one may use the following UI command before the "/run/initialize" command:

```
/cuts/setLowEdge 250 eV
```

or alternatively directly in the user Physics list, in the optional SetCuts() method, using:

```
G4ProductionCutsTable::GetProductionCutsTable()->SetEnergyRange(250*eV, 1*GeV);
```

A command is also available in order to disable usage of production threshold for fluorescence and Auger electron production:

```
/process/em/deexcitationIgnoreCut true
```

Angular Generators

For part of EM processes it is possible to factorise out sampling of secondary energy and direction. Using an interface `G4VEmModel` base class `SetAngularDistribution(G4VEmAngularDistribution*)` it is possible to substitute default angular generator of a model. Angular generators in standard and lowenergy sub-packages follow the same abstract interface.

For photoelectric models several angular generators are available:

- G4SauterGavrilaAngularDistribution (default);
- G4PhotoElectricAngularGeneratorSauterGavrila;
- G4PhotoElectricAngularGeneratorPolarized.

For bremsstrahlung models following angular generators are available:

- G4DipBustGenerator (default);
- G4ModifiedTsai;
- G4Generator2BS;
- G4Generator2BN;
- G4PenelopeBremsstrahlungAngular.

For models of ionisation a new optional angular generator is available:

- G4DeltaAngle.

Electromagnetics secondary biasing

It may be useful to create more than one secondary at an interaction. For example, electrons incident on a target in a medical linac produce photons through bremsstrahlung. The variance reduction technique of bremsstrahlung splitting involves choosing N photons from the expected distribution, and assigning each a weight of $1/N$.

Similarly, if the secondaries are not important, one can kill them with a survival probability of $1/N$. The weight of the survivors is increased by a factor N. This is known as Russian roulette.

Neither biasing technique is applied if the resulting daughter particles would have a weight below $1/N$, in the case of brem splitting, or above 1, in the case of Russian roulette.

These techniques can be enabled in Geant4 electromagnetics with the macro commands

```
/process/em/setSecBiasing processName Region factor energyLimit energyUnit
```

where: *processName* is the name of the process to apply the biasing to; *Region* is the region in which to apply biasing; *factor* is the inverse of the brem splitting or Russian roulette factor (1/*N*); *energyLimit energyUnit* is the high energy limit. If the first secondary has energy above this limit, no biasing is applied.

For example,

```
/process/em/setSecBiasing eBrem target 10 100 MeV
```

will result in electrons undergoing bremsstrahlung in the target region being split 10 times (if the first photon sampled has an energy less than 100 MeV).

Note that the biasing needs to be specified for each process individually. To apply Russian Roulette to daughter electrons from interactions of photons, issue the macro command for the processes phot, compt, conv.

Reference: BEAMnrc Users Manual, D.W.O Rogers, B. Walters, I. Kawrakow. NRCC Report PIRS-0509(A)revL, available at http://www.irs.inms.nrc.ca/inms/irs/BEAM/beamhome.html

Livermore Data Based Models

- **Photon models**
  - Photo-electric effect (class `G4LivermorePhotoElectricModel`)
  - Polarized Photo-electric effect (class `G4LivermorePolarizedPhotoElectricModel`)
  - Compton scattering (class `G4LivermoreComptonModel`)
  - Compton scattering (class `G4LowEPComptonModel`)
  - Polarized Compton scattering (class `G4LivermorePolarizedComptonModel`)
  - Rayleigh scattering (class `G4LivermoreRayleighModel`)
  - Polarized Rayleigh scattering (class `G4LivermorePolarizedRayleighModel`)
  - Gamma conversion (also called pair production, class `G4LivermoreGammaConversionModel`)
  - Nuclear gamma conversion (class `G4LivermoreNuclearGammaConversionModel`)
  - Radiative correction for pair production (class `G4LivermoreGammaConversionModelRC`)
  - Polarized gamma conversion (class `G4LivermorePolarizedGammaConversionModel`)
- **Electron models**
  - Bremsstrahlung (class `G4LivermoreBremsstrahlungModel`)
  - Ionisation and delta ray production (class `G4LivermoreIonisationModel`)

ICRU73 Based Ion Model

Ionisation and delta ray production (class G4IonParametrisedLossModel)

The ion model uses data files initially converted from the ICRU 73 report. Later authors of the ICRU 73 report provided Geant4 recomputated tables for more combinations of projectile and target ions. In 2015 newer calculation results were provided. The algorith of selection of ion stopping powers applying following condition: if a projectile/target combionation exists in the data base and the projectile energy is below 1 GeV/nucleon then tabulated data is used, otherwise applies a Bethe-Bloch based formalism. For compounds, ICRU 73 stopping powers are employed if the material name coincides with the name of Geant4 NIST materials (e.g. G4_WATER). Elemental materials are matched to the corresponding ICRU 73 stopping powers by means of the atomic number of the material. The material name may be arbitrary in this case. For a list of applicable materials, the user is referred to the ICRU 73 report.

The model requires data files to be copied by the user to his/her code repository. These files are distributed together with the Geant4 release. The user should set the environment variable G4LEDATA to the directory where he/she has copied the files.

The model is dedicated to be used with the G4ionIonisation process and its applicability is restricted to G4GenericIon particles. The ion model is not used by default by this process and must be instantiated and registered by the user:

```
G4ionIonisation* ionIoni = new G4ionIonisation();
ionIoni -> SetEmModel(new G4IonParametrisedLossModel());
```

Penelope2008 Based Models

- **Photon models**
  - Compton scattering (class `G4PenelopeComptonModel`)
  - Rayleigh scattering (class `G4PenelopeRayleighModel`)
  - Gamma conversion (also called pair production, class `G4PenelopeGammaConversionModel`)
  - Photo-electric effect (class `G4PenelopePhotoElectricModel`)
- **Electron models**
  - Bremsstrahlung (class `G4PenelopeBremsstrahlungModel`)
  - Ionisation and delta ray production (class `G4PenelopeIonisationModel`)
- **Positron models**
  - Bremsstrahlung (class `G4PenelopeBremsstrahlungModel`)
  - Ionisation and delta ray production (class `G4PenelopeIonisationModel`)
  - Positron annihilation (class `G4PenelopeAnnihilationModel`)

All Penelope models can be applied up to a maximum energy of 100 GeV, although it is advisable not to use them above a few hundreds of MeV.

Options are available in the all Penelope Models, allowing to set (and retrieve) the verbosity level of the model, namely the amount of information which is printed on the screen.

- SetVerbosityLevel(G4int)
- GetVerbosityLevel()

The default verbosity level is 0 (namely, no textual output on the screen). The default value should be used in general for normal runs. Higher verbosity levels are suggested only for testing and debugging purposes.

The verbosity scale defined for all Penelope processes is the following:

- 0 = no printout on the screen (default)
- 1 = issue warnings only in the case of energy non-conservation in the final state (should never happen)
- 2 = reports full details on the energy budget in the final state
- 3 = writes also informations on cross section calculation, data file opening and sampling of atoms
- 4 = issues messages when entering in methods

Very Low energy Electromagnetic Processes (Geant4-DNA extension)

The Geant4 low energy electromagnetic Physics package has been extended down to energies of a few electronVolts suitable for the simulation of radiation effects in liquid water for applications in micro/nanodosimetry at the cellular and sub-cellular level. These developments take place in the framework of the on-going Geant4-DNA project (see more in the Geant4-DNA web pages or in the web pages of the Geant4 Low Energy Electromagnetic Physics Working Group).

The Geant4-DNA process and model classes apply to electrons, protons, hydrogen, alpha particles and their charge states, in liquid water ("G4_WATER" material).

Electron processes and models

- Elastic scattering:
  - process class is G4DNAElastic

  - four alternative model classes are: G4DNAScreenedRutherfordElasticModel or G4DNAChampionElasticModel (default) or G4DNAUeharaScreenedRutherfordElasticModel or G4DNACPA100ElasticModel
- Excitation
  - process class is G4DNAExcitation
  - model class is G4DNABornExcitationModel (default) or G4DNAEmfietzoglouExcitationModel or G4DNACPA100ExcitationModel
- Ionisation
  - process class is G4DNAIonisation
  - model class is G4DNABornIonisationModel (default) or G4DNAEmfietzoglouIonisationModel or G4DNACPA100IonisationModel
- Attachment
  - process class is G4DNAAttachment
  - model class is G4DNAMeltonAttachmentModel
- Vibrational excitation
  - process class is G4DNAVibExcitation
  - model class is G4DNASancheExcitationModel

Proton processes and models

- Elastic scattering:
  - process class is G4DNAElastic
  - G4DNAIonElasticModel
- Excitation
  - process class is G4DNAExcitation
  - two complementary model classes are G4DNAMillerGreenExcitationModel (below 500 keV) and G4DNABornExcitationModel (above)
- Ionisation
  - process class is G4DNAIonisation
  - two complementary model classes are G4DNARuddIonisationModel (below 500 keV) and G4DNABornIonisationModel (above)
- Charge decrease
  - process class is G4DNAChargeDecrease
  - model class is G4DNADingfelderChargeDecreaseModel

Hydrogen processes and models

- Elastic scattering :
  - process class is G4DNAElastic
  - G4DNAIonElasticModel
- Excitation
  - process class is G4DNAExcitation
  - model class is G4DNAMillerGreenExcitationModel
- Ionisation
  - process class is G4DNAIonisation
  - model class is G4DNARuddIonisationModel
- Charge increase
  - process class is G4DNAChargeIncrease
  - model class is G4DNADingfelderChargeIncreaseModel

Helium (neutral) processes and models

- Elastic scattering :
  - process class is G4DNAElastic
  - G4DNAIonElasticModel
- Excitation
  - process class is G4DNAExcitation

    - model class is G4DNAMillerGreenExcitationModel
- Ionisation
    - process class is G4DNAIonisation
    - model class is G4DNARuddIonisationModel
- Charge increase
    - process class is G4DNAChargeIncrease
    - model class is G4DNADingfelderChargeIncreaseModel

Helium+ (ionized once) processes and models

- Elastic scattering :
    - process class is G4DNAElastic
    - G4DNAIonElasticModel
- Excitation
    - process class is G4DNAExcitation
    - model class is G4DNAMillerGreenExcitationModel
- Ionisation
    - process class is G4DNAIonisation
    - model classes is G4DNARuddIonisationModel
- Charge increase
    - process class is G4DNAChargeIncrease
    - model classes is G4DNADingfelderChargeIncreaseModel
- Charge decrease
    - process class is G4DNAChargeDecrease
    - model classes is G4DNADingfelderChargeDecreaseModel

Helium++ (ionised twice) processes and models

- Elastic scattering :
    - process class is G4DNAElastic
    - G4DNAIonElasticModel
- Excitation
    - process class is G4DNAExcitation
    - model classes is G4DNAMillerGreenExcitationModel
- Ionisation
    - process class is G4DNAIonisation
    - model classes is G4DNARuddIonisationModel
- Charge decrease
    - process class is G4DNAChargeDecrease
    - model classes is G4DNADingfelderChargeDecreaseModel

Li, Be, B, C, N, O, Si, Fe processes and models

- Ionisation
    - process class is G4DNAIonisation
    - model class is G4DNARuddIonisationExtendedModel

Examples of the registration of these processes in a physics list are given in the G4EmDNAPhysics* constructors (in `source/physics_lists/constructors/electromagnetic` in the Geant4 source distribution). An example of the usage of these constructors in a physics list is given in the "dnaphysics" extended example, which explains how to extract basic information from Geant4-DNA Physics processes.

Geant4-DNA physics constructors are described at the Geant4-DNA website.

The "microdosimetry" extended example illustrates how to combine Geant4-DNA processes with Standard electromagnetic processes (combination of discrete and condensed history Geant4 electromagnetic processes at different scales).

A set of Geant4-DNA models applicable to biological materials is available since release 10.4. These

models are named G4DNAPTBElasticModel, G4DNAPTBExcitationModel, G4DNAPTBIonisationModel and G4DNAPTBAugerModel. They can be used for electrons in THF, PY, PU, TMP precursors and in backbone, cytosine, thymine, adenine, guanine materials of DNA. The G4DNAPTBIonisationModel can also be used with protons in THF, PY and TMP. Their usage is illustrated in the "icsd" extended example.

Since Geant4 release 10.1, Geant4-DNA can also be used for the modelling of water radiolysis (physico-chemistry and chemistry stages). Three extended examples, "chem1", "chem2", "chem3" and "chem4" illustrate this. More information is available from the Geant4-DNA website.

To run the Geant4-DNA extension, data files need to be copied by the user to his/her code repository. These files are distributed together with the Geant4 release. The user should set the environment variable G4LEDATA to the directory where he/she has copied the files.

A full list of publications regarding Geant4-DNA is directly available from the Geant4-DNA website or from the Geant4@IN2P3 web site).

Atomic Deexcitation

A unique interface named G4VAtomicDeexcitation is available in Geant4 for the simulation of atomic deexcitation using Standard, Low Energy and Very Low Energy electromagnetic processes. Atomic deexcitation includes fluorescence and Auger electron emission induced by photons, electrons and ions (PIXE); see more details in:

A. Mantero et al., PIXE Simulation in Geant4 , X-Ray Spec. , 40, 135-140, 2011.

It can be activated for processes producing vacancies in atomic shells. Currently these processes are the photoelectric effect, ionization and Compton scattering.

Activation of atomic deexcitation

The activation of atomic deexcitation in continuous processes in a user physics list can be done through the following G4EmParameters class methods described above or via UI commands:

```
/process/em/deexcitation region true true true
/process/em/fluo true
/process/em/auger true
/process/em/pixe true
```

One can define parameters in the G4State_PreInit or G4State_Idle states. Fluorescence from photons and electrons is activated by default in Option3, Option4, Livermore and Penelope physics constructors, while Auger production and PIXE are not.

The alternative set of data by Bearden et al. (1967) for the modelling of fluorescence lines had been added to the G4LEDATA archive. This set can be selected via UI command:

```
/process/em/fluoBearden true
```

Another important UI commands enable simulation of the full Auger and/or fluorescence cascade:

```
/process/em/augerCascade true
/process/em/deexcitationIgnoreCut true
```

How to change ionisation cross section models ?

The user can also select which cross section model to use in order to calculate shell ionisation cross sections for generating PIXE:

```
/process/em/pixeXSmodel     name
/process/em/pixeElecXSmodel name
```

where the name can be "Empirical", "ECPSSR_FormFactor" or "ECPSSR_Analytical" corresponds to different PIXE cross sections. Following shell cross sections models are available : "ECPSSR_Analytical" models derive from an analytical calculation of the ECPSSR theory (see *A. Mantero et al., X-Ray Spec.40 (2011) 135-140*) and it reproduces K and L shell cross sections over a wide range of energies; "ECPSSR_FormFactor" models derive from A. Taborda et al. calculations (see *A. Taborda et al., X-Ray Spec. 40 (2011) 127-134*) of ECPSSR values directly form Form Factors and it covers K, L shells on the range 0.1-100 MeV and M shells in the range 0.1-10 MeV; the "empirical" models are from Paul "reference values" (for protons and alphas for K-Shell) and Orlic empirical model for L shells (only for protons and ions with Z>2). The later ones are the models used by default. Out of the energy boundaries, "ECPSSR_Analytical" model is used. We recommend to use default settings if not sure what to use.

Example

The **TestEm5 extended/electromagetic example** shows how to simulate atomic deexcitation (see for eg. the pixe.mac macro).

Very Low energy Electromagnetic Processes in Silicon for microelectronics application (Geant4-MuElec extension)

(Previously named Geant4-MuElec)

The Geant4 low energy electromagnetic Physics package has been extended down to energies of a few electronVolts suitable for the simulation of radiation effects in highly integrated microelectronic components.

The Geant4-MicroElec process and model classes apply to electrons, protons and heavy ions in silicon.

Electron processes and models

- Elastic scattering :
  - process class is G4MicroElastic
  - model class is G4MicroElecElasticModel
- Ionization
  - process class is G4MicroElecInelastic
  - model class is G4MicroElecInelasticModel

Proton processes and models

- Ionisation
  - process class is G4MicroElecInelastic
  - model class is G4MicroElecInelasticModel

Heavy ion processes and models

- Ionization
  - process class is G4MicroElecInelastic
  - model class is G4MicroElecInelasticModel

A full list of publications regarding Geant4-MicroElec is directly available from the Geant4-MicroElec website.

New Compton model by Monash U., Australia

A new Compton scattering model for unpolarised photons has been developed in the relativistic impulse approximation. The model was developed as an alternative to low energy electromagnetic Compton scattering models developed from Ribberfors' Compton scattering framework (Livermore, Penelope Compton models). The model class is named named G4LowEPComptonModel.

G4LowEPComptonModel has been added to the physics constructor G4EmStandardPhysics_option4, containing the most accurate models from the Standard and Low Energy Electromagnetic physics working groups.

Multi-scale Processes

Hadron Impact Ionisation and PIXE

The **G4hImpactIonisation** process deals with ionisation by impact of hadrons and alpha particles, and the following generation of **PIXE** (Particle Induced X-ray Emission). This process and related classes can be found in *source/processes/electromagnetic/pii*.

Further documentation about PIXE simulation with this process is available here.

A detailed description of the related physics features can be found in:

M. G. Pia et al., PIXE Simulation with Geant4 , IEEE Trans. Nucl. Sci. , vol. 56, no. 6, pp. 3614-3649, 2009.

A brief summary of the related physics features can be found in the Geant4 Physics Reference Manual.

An example of how to use this process is shown below. A more extensive example is available in the eRosita Geant4 advanced example (see *examples/advanced/eRosita* in your Geant4 installation source).

```
#include "G4hImpactIonisation.hh"
[...]

void eRositaPhysicsList::ConstructProcess()
{

[...]

  theParticleIterator->reset();
  while( (*theParticleIterator)() )
    {
      G4ParticleDefinition* particle = theParticleIterator->value();
      G4ProcessManager* processManager = particle->GetProcessManager();
      G4String particleName = particle->GetParticleName();

      if (particleName == "proton")
        {
          // Instantiate the G4hImpactIonisation process
          G4hImpactIonisation* hIonisation = new G4hImpactIonisation();

          // Select the cross section models to be applied for K, L and M shell vacancy creation
          // (here the ECPSSR model is selected for K, L and M shell; one can mix and match
          // different models for each shell)
          hIonisation->SetPixeCrossSectionK("ecpssr");
          hIonisation->SetPixeCrossSectionL("ecpssr");
          hIonisation->SetPixeCrossSectionM("ecpssr");

          // Register the process with the processManager associated with protons
          processManager -> AddProcess(hIonisation, -1, 2, 2);
        }
     }
}
```

Available cross section model options

The following cross section model options are available:

- protons
  - K shell
    - `ecpssr` *(based on the ECPSSR theory)*
    - `ecpssr_hs` *(based on the ECPSSR theory, with Hartree-Slater correction)*
    - `ecpssr_ua` *(based on the ECPSSR theory, with United Atom Hartree-Slater correction)*
    - `ecpssr_he` *(based on the ECPSSR theory, with high energy correction)*
    - `pwba` *(plane wave Born approximation)*
    - `paul` *(based on the empirical model by Paul and Sacher)*

        * `kahoul` *(based on the empirical model by Kahoul et al.)*
    - L shell
        * `ecpssr`
        * `ecpssr_ua`
        * `pwba`
        * `miyagawa` *(based on the empirical model by Miyagawa et al.)*
        * `orlic` *(based on the empirical model by Orlic et al.)*
        * `sow` *(based on the empirical model by Sow et al.)*
    - M shell
        * `ecpssr`
        * `pwba`
- alpha particles
    - K shell
        * `ecpssr`
        * `ecpssr_hs`
        * `pwba`
        * `paul` *(based on the empirical model by Paul and Bolik)*
    - L shell
        * `ecpssr`
        * `pwba`
    - M shell
        * `ecpssr`
        * `pwba`

PIXE data library

The *G4hImpactIonisation* process uses a **PIXE Data Library.**

The PIXE Data Library is distributed in the Geant4 **G4PII** data set, which must be downloaded along with Geant4 source code.

The **G4PIIDATA** environment variable must be defined to refer to the location of the G4PII PIXE data library in your filesystem; for instance, if you use a c-like shell:

```
setenv G4PIIDATA path_to_where_G4PII_has_been_downloaded
```

Further documentation about the PIXE Data Library is available here.

5.2.2 Hadronic Interactions

This section briefly introduces the hadronic physics processes installed in Geant4. For details of the implementation of hadronic interactions available in Geant4, please refer to the **Physics Reference Manual**.

Treatment of Cross Sections

Cross section data sets

Each hadronic process object (derived from `G4HadronicProcess`) may have one or more cross section data sets associated with it. The term "data set" is meant, in a broad sense, to be an object that encapsulates methods and data for calculating total cross sections for a given process. The methods and data may take many forms, from a simple equation using a few hard-wired numbers to a sophisticated parameterisation using large data tables. Cross section data sets are derived from the abstract class `G4VCrossSectionDataSet`, and are required to implement the following methods:

```
G4bool IsApplicable( const G4DynamicParticle*, const G4Element* )
```

This method must return `True` if the data set is able to calculate a total cross section for the given particle and material, and `False` otherwise.

```
G4double GetCrossSection( const G4DynamicParticle*, const G4Element* )
```

This method, which will be invoked only if `True` was returned by `IsApplicable`, must return a cross section, in Geant4 default units, for the given particle and material.

```
void BuildPhysicsTable( const G4ParticleDefinition& )
```

This method may be invoked to request the data set to recalculate its internal database or otherwise reset its state after a change in the cuts or other parameters of the given particle type.

```
void DumpPhysicsTable( const G4ParticleDefinition& ) = 0
```

This method may be invoked to request the data set to print its internal database and/or other state information, for the given particle type, to the standard output stream.

Cross section data store

Cross section data sets are used by the process for the calculation of the physical interaction length. A given cross section data set may only apply to a certain energy range, or may only be able to calculate cross sections for a particular type of particle. The class `G4CrossSectionDataStore` has been provided to allow the user to specify, if desired, a series of data sets for a process, and to arrange the priority of data sets so that the appropriate one is used for a given energy range, particle, and material. It implements the following public methods:

```
G4CrossSectionDataStore()

~G4CrossSectionDataStore()
```

and

```
G4double GetCrossSection( const G4DynamicParticle*, const G4Element* )
```

For a given particle and material, this method returns a cross section value provided by one of the collection of cross section data sets listed in the data store object. If there are no known data sets, a `G4Exception` is thrown and `DBL_MIN` is returned. Otherwise, each data set in the list is queried, in reverse list order, by invoking its `IsApplicable` method for the given particle and material. The first data set object that responds positively will then be asked to return a cross section value via its `GetCrossSection` method. If no data set responds positively, a `G4Exception` is thrown and `DBL_MIN` is returned.

```
void AddDataSet( G4VCrossSectionDataSet* aDataSet )
```

This method adds the given cross section data set to the end of the list of data sets in the data store. For the evaluation of cross sections, the list has a LIFO (Last In First Out) priority, meaning that data sets added later to the list will have priority over those added earlier to the list. Another way of saying this, is that the data store, when given a `GetCrossSection` request, does the `IsApplicable` queries in the reverse list order, starting with the last data set in the list and proceeding to the first, and the first data set that responds positively is used to calculate the cross section.

```
void BuildPhysicsTable( const G4ParticleDefinition& aParticleType )
```

This method may be invoked to indicate to the data store that there has been a change in the cuts or other parameters of the given particle type. In response, the data store will invoke the `BuildPhysicsTable` of each of its data sets.

```
void DumpPhysicsTable( const G4ParticleDefinition& )
```

This method may be used to request the data store to invoke the DumpPhysicsTable method of each of its data sets.

Default cross sections

The defaults for total cross section data and calculations have been encapsulated in the singleton class G4HadronCrossSections. Each hadronic process: G4HadronInelasticProcess, G4HadronElasticProcess, G4HadronFissionProcess, and G4HadronCaptureProcess, comes already equipped with a cross section data store and a default cross section data set. The data set objects are really just shells that invoke the singleton G4HadronCrossSections to do the real work of calculating cross sections.

The default cross sections can be overridden in whole or in part by the user. To this end, the base class G4HadronicProcess has a get method:

```
G4CrossSectionDataStore* GetCrossSectionDataStore()
```

which gives public access to the data store for each process. The user's cross section data sets can be added to the data store according to the following framework:

```
G4Hadron...Process aProcess(...)

MyCrossSectionDataSet myDataSet(...)

aProcess.GetCrossSectionDataStore()->AddDataSet( &MyDataSet )
```

The added data set will override the default cross section data whenever so indicated by its IsApplicable method.

In addition to the get method, G4HadronicProcess also has the method

```
void SetCrossSectionDataStore( G4CrossSectionDataStore* )
```

which allows the user to completely replace the default data store with a new data store.

It should be noted that a process does not send any information about itself to its associated data store (and hence data set) objects. Thus, each data set is assumed to be formulated to calculate cross sections for one and only one type of process. Of course, this does not prevent different data sets from sharing common data and/or calculation methods, as in the case of the G4HadronCrossSections class mentioned above. Indeed, G4VCrossSectionDataSet specifies only the abstract interface between physics processes and their data sets, and leaves the user free to implement whatever sort of underlying structure is appropriate.

The current implementation of the data set G4HadronCrossSections reuses the total cross-sections for inelastic and elastic scattering, radiative capture and fission as used with **GHEISHA** to provide cross-sections for calculation of the respective mean free paths of a given particle in a given material.

Cross-sections for low energy neutron transport

The cross section data for low energy neutron transport are organized in a set of files that are read in by the corresponding data set classes at time zero. Hereby the file system is used, in order to allow highly granular access to the data. The ''root" directory of the cross-section directory structure is accessed through an environment variable, G4NEUTRONHPDATA, which is to be set by the user. The classes accessing the total cross-sections of the individual processes, i.e., the cross-section data set classes for low energy neutron transport, are G4NeutronHPElasticData, G4NeutronHPCaptureData, G4NeutronHPFissionData, and G4NeutronHPInelasticData.

For detailed descriptions of the low energy neutron total cross-sections, they may be registered by the user as described above with the data stores of the corresponding processes for neutron interactions.

It should be noted that using these total cross section classes does not require that the neutron_hp models also be used. It is up to the user to decide whethee this is desirable or not for his particular problem.

A prototype of the compact version of neutron cross sections derived from HP database are provided with new classes `G4NeutronHPElasticData`, `G4NeutronCaptureXS`, `G4NeutronElasticXS`, and `G4NeutronInelasticXS`.

Cross-sections for low-energy charged particle transport

The cross-section data for low-energy charged particle transport are organized in a set of files that are read at initialization, similarly to the case of low-energy neutron transport. The "root" directory of the cross-section directory structure is accessed through an environment variable, `G4PARTICLEHPDATA`, which has to be set by the user. This variable has to point to the directory where the low-energy charged particle data have been installed, e.g. `G4TENDL1.3` for the Geant4 release `10.3` (note that the download of this data library from the Geant4 web site is not done automatically, i.e. it must be done manually by the user):

```
export G4PARTICLEHPDATA=/your/path/G4TENDL1.3/.
```

It is expected that the directory `$G4PARTICLEHPDATA` has the following five subdirectories, corresponding to the charged particles that can be handled by the low-energy charged particle transport: `Proton/`, `Deuteron/`, `Triton/`, `He3/`, `Alpha/`. It is possible for the user to overwrite the default directory structure with individual environment variables pointing to custom data libraries for each particle type. This is considered an advanced/expert user feature. These directories are set by the following environment variables: `G4PROTONHPDATA`, for proton; `G4DEUTERONHPDATA`, for deuteron; `G4TRITONHPDATA`, for triton; `G4HE3HPDATA`, for He3; `G4ALPHAHPDATA`, for alpha. Note that if any of these variables is not defined explicitly, e.g. `G4TRITONHPDATA`, then the corresponding data library is expected to be a subdirectory of `$G4PARTICLEHPDATA/`, e.g. `$G4PARTICLEHPDATA/Triton/`. If instead all the above five environmental variables are set, then `G4PARTICLEHPDATA` does not need to be specified; even if it is set, then its value will be ignored (because the per-particle ones take precedence).

Hadrons at Rest

List of implemented "Hadron at Rest" processes

The following process classes have been implemented:

- pi-, K-, sigma-, xi-, omega- absorption (class name `G4HadronicAbsorptionBertini`)
- neutron capture (class name `G4HadronCaptureProcess`)
- anti-proton, anti-sigma+, anti-deuteron, anti-triton, anti-alpha, anti-He3 annihilation (class name `G4HadronicAbsorptionFritiof`)
- mu- capture (class name `G4MuonMinusCapture`)

Capture of low-energy negatively charged particles is a complex process involving formation of mesonic atoms, X-ray cascade and Auger cascade, nuclear interaction. In the case of mu- there is also a probability to decay from K-shell of mesonic atom. To handle this a base process class `G4HadronicStoppingProcess` is used.

For the case of neutrons, Geant4 offer simulation down to thermal energies. The capture cross section generally increases when neutron energy descreases and there are many nuclear resonances. In Geant4 neutron capture cross sections are parameterized using ENDF database.

Hadrons in Flight

What processes do you need?

For hadrons in motion, there are four physics process classes. *Hadronic processes table* shows each process and the particles for which it is relevant.

G4HadronElasticProcess	pi+, pi-, K^+, K^0_S, K^0_L, K^-, p, p-bar, n, n-bar, lambda, lambda-bar, $Sigma^+$, $Sigma^-$, $Sigma^+$-bar, $Sigma^-$-bar, Xi^0, Xi^-, Xi^0-bar, Xi^--bar
G4HadronInelasticProcess	pi+, pi-, K^+, K^0_S, K^0_L, K^-, p, p-bar, n, n-bar, lambda, lambda-bar, $Sigma^+$, $Sigma^-$, $Sigma^+$-bar, $Sigma^-$-bar, Xi^0, Xi^-, Xi^0-bar, Xi^--bar
G4HadronFissionProcess	all
G4CaptureProcess	n, n-bar

Table: Hadronic processes and relevant particles.

How to register Models

To register an inelastic process model for a particle, a proton for example, first get the pointer to the particle's process manager:

```
G4ParticleDefinition *theProton = G4Proton::ProtonDefinition();
G4ProcessManager *theProtonProcMan = theProton->GetProcessManager();
```

Create an instance of the particle's inelastic process:

```
G4ProtonInelasticProcess *theProtonIEProc = new G4ProtonInelasticProcess();
```

Create an instance of the model which determines the secondaries produced in the interaction, and calculates the momenta of the particles, for instance the Bertini cascade model:

```
G4CascadeInterface *theProtonIE = new G4CascadeInterface();
```

Register the model with the particle's inelastic process:

```
theProtonIEProc->RegisterMe( theProtonIE );
```

Finally, add the particle's inelastic process to the list of discrete processes:

```
theProtonProcMan->AddDiscreteProcess( theProtonIEProc );
```

The particle's inelastic process class, G4ProtonInelasticProcess in the example above, derives from the G4HadronicInelasticProcess class, and simply defines the process name and calls the G4HadronicInelasticProcess constructor. All of the specific particle inelastic processes derive from the G4HadronicInelasticProcess class, which calls the PostStepDoIt function, which returns the particle change object from the G4HadronicProcess function GeneralPostStepDoIt. This class also gets the mean free path, builds the physics table, and gets the microscopic cross section. The G4HadronicInelasticProcess class derives from the G4HadronicProcess class, which is the top level hadronic process class. The G4HadronicProcess class derives from the G4VDiscreteProcess class. The inelastic, elastic, capture, and fission processes derive from the G4HadronicProcess class. This pure virtual class also provides the energy range manager object and the RegisterMe access function.

In-flight, final-state hadronic models derive, directly or indirectly, from the G4InelasticInteraction class, which is an abstract base class since the pure virtual function ApplyYourself is not defined there. G4InelasticInteraction itself derives from the G4HadronicInteraction abstract base class. This class is the base class for all the model classes. It sorts out the energy range for the models and provides class utilities. The G4HadronicInteraction class provides the Set/GetMinEnergy and the Set/GetMaxEnergy functions which determine the minimum and maximum energy range for the model. An energy range can be set for a specific element, a specific material, or for general applicability:

```
void SetMinEnergy( G4double anEnergy, G4Element *anElement )
void SetMinEnergy( G4double anEnergy, G4Material *aMaterial )
void SetMinEnergy( const G4double anEnergy )
void SetMaxEnergy( G4double anEnergy, G4Element *anElement )
```

```
void SetMaxEnergy( G4double anEnergy, G4Material *aMaterial )
void SetMaxEnergy( const G4double anEnergy )
```

Which models are there, and what are the defaults

In Geant4, any model can be run together with any other model without the need for the implementation of a special interface, or batch suite, and the ranges of applicability for the different models can be steered at initialisation time. This way, highly specialised models (valid only for one material and particle, and applicable only in a very restricted energy range) can be used in the same application, together with more general code, in a coherent fashion.

Each model has an intrinsic range of applicability, and the model chosen for a simulation depends very much on the use-case. Consequently, there are no ''defaults''. However, physics lists are provided which specify sets of models for various purposes.

Two types of hadronic shower models have been implemented: data driven models and theory driven models.

- Data driven models are available for the transport of low energy neutrons in matter in sub-directory `hadronics/models/neutron_hp`. The modeling is based on the data formats of **ENDF/B-VI**, and all distributions of this standard data format are implemented. The data sets used are selected from data libraries that conform to these standard formats. The file system is used in order to allow granular access to, and flexibility in, the use of the cross sections for different isotopes, and channels. The energy coverage of these models is from thermal energies to 20 MeV.
- Theory driven models are available for inelastic scattering in a first implementation, covering the full energy range of LHC experiments. They are located in sub-directory `hadronics/models/generator`. The current philosophy implies the usage of parton string models at high energies, of intra-nuclear transport models at intermediate energies, and of statistical break-up models for de-excitation.

High-precision neutron interactions (NeutronHP)

Nuclear models fail (sometimes catastrophically) at predicting with reasonable accuracies the nuclear cross sections of neutrons (and other particles). For this reason, all physical quantities relevant for an accurate modeling of nuclear reactions in Monte Carlo simulations need to be provided as a database which includes, ideally:

- cross sections
- angular distributions of the emitted particles
- energy spectra of the emitted particles
- energy-angle correlated spectrum (double-differential cross sections, DDX)
- neutrons per fission
- fission spectra
- fission product yields
- photo production data

For the case of neutron induced reactions, such databases are called “evaluated data”, in the sense that they contain recommended values for different quantities that rely on compilations of experimental nuclear data and usually completed with theoretical predictions, benchmarked against available experimental data (i.e. integral and differential experiments) when possible. It should be noticed that the information available varies from isotope to isotope and can be incomplete or totally missing.

The G4NeutronHP package in GEANT4 allows using evaluated nuclear data libraries in the G4NDL format. GEANT4 users should know that any simulation involving neutrons with energies below 20 MeV and not using the G4NeutronHP package can lead to unreliable results. GEANT4 users are therefore encouraged to use it, although they should be aware of the limitations of using evaluated nuclear data libraries.

An example about how to implement the G4NeutronHP package into physics list in a GEANT4 application can be found in the example case (among others distributed with GEANT4) `extended/radioactivedecay/rdecay02`. Three different processes are included in that example: elastic, capture and inelastic. The inelastic

reactions in G4NeutronHP are all reactions except elastic, capture and fission, so fission should also be included in the physics list, if needed, and it is done in the same way as it is done for the other three.

The G4NeutronHP package must be used together with evaluated nuclear data libraries. They are distributed by the GEANT4 collaboration (http://geant4.web.cern.ch/geant4/support/download.shtml) and from the IAEA nuclear data web site (http://www-nds.iaea.org/geant4/) where a larger set of different libraries, including isotopes with Z > 92, is available.

The evaluated nuclear data libraries do differ and thus the results of the Monte Carlo simulations will depend on the library used. It is a safe practice to perform simulations with (at least) two different libraries for estimating the uncertainties associated to the nuclear data.

Together with a good implementation of the physics list, users must be very careful with the definition of the materials performed in a Monte Carlo simulation when low energy neutron transport is relevant. In contrast to other kind of simulations, the isotopic composition of the elements which compose the different materials can strongly affect the obtained simulation results. Because of this, it is strongly recommended to define specifically the isotopic composition of each element used in the simulation, as it is described in the GEANT4 user's manual. In principle, such a practice is not mandatory if natural isotopic compositions are used, since GEANT4 contains them in their databases. However, by defining them explicitly some unexpected problems may be avoided and a better control of the simulation will be achieved.

It is highly recommended or mandatory to set the following UNIX environment variables running a GEANT4 application:

`G4NEUTRONHPDATA` [path to the G4NDL format data libraries] (mandatory).

`G4NEUTRONHP_SKIP_MISSING_ISOTOPES=1` It sets to zero the cross section of the isotopes which are not present in the neutron library. If GEANT4 doesn't find an isotope, then it looks for the natural composition data of that element. Only if the element is not found then the cross section is set to zero. On the contrary, if this variable is not defined, GEANT4 looks then for the neutron data of another isotope close in Z and A, which will have completely different nuclear properties and lead to incorrect results (highly recommended).

`G4NEUTRONHP_DO_NOT_ADJUST_FINAL_STATE=1` If this variable is not defined, a GEANT4 model that attempts to satisfy the energy and momentum conservation in some nuclear reactions, by generating artificial gamma rays. By setting such a variable one avoids the correction and leads to the result obtained with the ENDF-6 libraries. Even though energy and momentum conservation are desirable, the ENDF-6 libraries do not provide the necessary correlations between secondary particles for satisfying them in all cases. On the contrary, ENDF-6 libraries intrinsically violate energy and momentum conservation for several processes and have been built for preserving the overall average quantities such as average energy releases, average number of secondaries... (highly recommended).

`AllowForHeavyElements=1` Activates the physics for isotopes with Z>92 (recommended).

The G4NDL format libraries are based on the ENDF-6 format libraries, which contain evaluated (i.e. recommended) nuclear data prepared for their use in transport codes. These data are essentially nuclear reaction cross sections together with the distribution in energy and angle of the secondary reaction products. As a consequence of how the data is written in the ENDF files, there are some features that may be or may be not expected in the results of a Monte Carlo calculation.

The information concerning the creation of the reaction products can be incomplete and/or uncorrelated, in the sense that is described below:

1. **Incomplete information.**
   This applies when there is no information about how to generate a secondary particle. As an example, it is possible to have only the cross section data of an (n,p) reaction, without any information concerning the energy and angle of the secondary proton. In this case GEANT4 will produce the proton considering that it is emitted isotropically in the center of mass frame, with an energy which is deduced from assuming that the residual nucleus is in its ground state.

2. **Uncorrelated information.**
   This applies when:

(a) The energy and angle distributions of a reaction product may be uncorrelated. As a consequence, the reaction products can be generated with an unphysical energy-angle relationship.
(b) The energy-angle distributions of different reaction products of a certain reaction are always uncorrelated. As an example, consider that in a (n, 2p) reaction at a certain neutron energy both resulting protons can be emitted with energies ranging from 0 to 5MeV. In this case the energy and angle of each proton will be sampled independently of the energy and angle of the other proton, so there will be events in which both protons will be emitted with energies close to 5 MeV and there will also be events in which both protons will be emitted with energies close to 0 MeV. As a consequence, energy and angular momentum won't be conserved event by event. However, energy will be conserved in average and the resulting proton energy spectrum will be correctly produced.

3. **Concatenated reactions.**
There are some cases where several nuclear reactions are put together as if they were a single reaction (`MT=5` reaction, in ENDF-6 format nomenclature). In those cases the information consists in a cross section, which is the sum of all of them, plus a reaction product yield and energy-angle distributions for each secondary particle. In this case the amount of each secondary particle produced has to be sampled every time the reaction occurs, and it is done independently of the amount of the other secondary particles produced.
Thus, in this case neither the energy and angular momentum nor the number of nucleons is conserved event by event, but all the quantities should be conserved in average. As a consequence, it is also not possible to deduce which are the residual nuclei produced, since no information is available concerning what are the specific nuclear reactions which take place. It has to be said that sometimes ENDF libraries include the residual nuclei as an outgoing particle. However, GEANT4 does not manage that information, at present. This situation is quite uncommon in neutron data libraries up to 20 MeV. However, it is quite common to find it in charged particle libraries below 20 MeV or in neutron libraries above 20 MeV.

As a consequence of what has been presented above, some general features can be expected in the results of a Monte Carlo calculation performed with the G4NeutronHP package:

- The neutron transport, which means how the neutron looses energy in the collisions, when and how it is absorbed..., is quite trustable, since the main purpose of the ENDF neutron libraries is to perform this neutron transport.
- The production of neutrons due to neutron induced nuclear reactions is usually trustable, with the exception of the energy-angle correlations when several neutrons are produced in the same nuclear reaction.
- The results concerning the production of charged particles have to be always questioned. A look into the ENDF format library used can indicate which results are trustable and which are not. This can be done, for example, in http://t2.lanl.gov/data/data.html, among other websites.
- The results concerning the production of γ-rays have to be questioned always. For example, the information on the number and energies of γ-rays emitted in the neutron capture process is incomplete for almost all the nuclei and is frequently also uncorrelated. When the information is available, it will be used, but one can obtain results which are quite far from reality on an event by event basis: the total energy of the cascade won't be correct in many cases and only some specific γ-rays which are stored in the neutron databases will be emitted. If there isn't any information concerning these γ-rays, GEANT4 will use a simple a model instead which is generally missing the relevant spectroscopic information. The results concerning the generation of residual nuclei (for example, in activation calculations) are usually trustable, with the exception of libraries with MT=5 reactions, as described above (*Uncorrelated*).

As a general conclusion, users should always be critical with the results obtained with Monte Carlo simulation codes, and this also applies to GEANT4. They have to anticipate which results can be trusted and which results should be questioned. For the particular case of the a closer look into the underlying evaluated nuclear datain the ENDF format libraries will allow to check what is the information available in a certain library for some specific isotope and a certain reaction. There are several public nuclear data sites like http://t2.lanl.gov/data/data.html.

The transport of very low energy neutrons (below 5 eV) has to be performed using the thermal neutron data libraries. At these energies, the fact that the nuclei are in atoms which form part of a certain molecule inside a material (crystal lattice, liquid, plastic...) plays an important role, since there can be a transference of momentum between the neutron and the whole structure of the material, not only with the nucleus. This is of particular importance for material used as neutron moderators, i.e., materials with low A (mass number) used to decrease the incident neutron energy in only a

few collisions. Since the property is related to the nucleus in the material, as an example, there is the need for having different thermal libraries for Hydrogen in polyethylene, Hydrogen in water and so on.

If neutron collisions at these energies are relevant for the problem to be simulated, thermal libraries should be used for the materials if they are available. If they are not, the results obtained from the simulation will not be trustable in the neutron energy range below 5 eV, especially when using low mass elements in the simulation.

To use the thermal libraries the following lines should be included in the physics list:

```
G4HadronElasticProcess* theNeutronElasticProcess = new G4HadronElasticProcess;
// Cross Section Data set
G4NeutronHPElasticData* theHPElasticData = new G4NeutronHPElasticData;
theNeutronElasticProcess->AddDataSet(theHPElasticData);
G4NeutronHPThermalScatteringData* theHPThermalScatteringData = new␣
↪G4NeutronHPThermalScatteringData;
theNeutronElasticProcess->AddDataSet(theHPThermalScatteringData);
// Models
G4NeutronHPElastic* theNeutronElasticModel = new G4NeutronHPElastic;
theNeutronElasticModel->SetMinEnergy(4.0*eV);
theNeutronElasticProcess->RegisterMe(theNeutronElasticModel);
G4NeutronHPThermalScattering* theNeutronThermalElasticModel = new G4NeutronHPThermalScattering;
theNeutronThermalElasticModel->SetMaxEnergy(4.0*eV);
theNeutronElasticProcess->RegisterMe(theNeutronThermalElasticModel);
// Apply Processes to Process Manager of Neutron
G4ProcessManager* pmanager = G4Neutron::Neutron()->GetProcessManager();
pmanager->AddDiscreteProcess(theNeutronElasticProcess);
```

And the materials should be defined with a specific name. For example, to use the thermal library for Hydrogen in water, the water should be defined as:

```
G4Element* elTSHW = new G4Element("TS_H_of_Water", "H_WATER", 1.0, 1.0079*g/mole);
G4Material* matH2O_TS = new G4Material("Water_TS", density=1.0*g/cm3, ncomponents=2);
matH2O_TS->AddElement(elTSHW,natoms=2);
matH2O_TS->AddElement(elO,natoms=1);
```

where the important thing is the name **"TS_H_of_Water"**, which is a specific name used by G4NeutronHP. In order to see which thermal libraries are available, they can be found in the G4NDL4.0/ThermalScattering folder (or equivalent, for other neutron libraries). Then, one has to look into the `G4NeutronHPThermalScatteringNames.cc` source file, under `source/processes/hadronic/models/neutron_hp/src`. There are some lines similar to:

```
names.insert(std::pair<G4String,G4String>("TS_H_of_Water", "h_water"));
```

where **"TS_H_of_Water"** means Hydrogen in water. Names similar to **"TS_H_of_Water"** like **"TS_C_of_Graphite"** or **"TS_H_of_Polyethylene"** can be found and used in the same way as described above.

High-precision charged particle interactions (ParticleHP)

Due to the coupling between the configuration for neutrons and charged particles in ParticleHP, the default one is not the recommended one from the physics point of view for all particles. A consistent configuration with thorough testing will hopefully be introduced in the next release. For the time being, in order to improve the physics performance for primary charged particles the following environment variable should be set:

```
export DO_NOT_SET_PHP_AS_HP=1
```

Note that this environmental variable is a configuration option which is used only at compilation, not at run time, and it affects both primary neutrons and charged particles. It is not expected to dramatically change the behaviour for neutrons.

For further improvement with projectile charged particles, it is also recommended to set the following environmental variable used at run-time:

```
export G4PHP_DO_NOT_ADJUST_FINAL_STATE=1
```

which avoids the default adjustment of the final state to ensure better conservation laws (for charge, energy, momentum, baryon number).

The adjustment of the final state is recommended for realistic detector response in the case of neutron interactions. For the use-case of reactor physics and dosimetry, where *average* quantities are important, not adjusting the final state (i.e. setting the above environment variable) improves accuracy.

Note that, for the time being, setting `G4PHP_DO_NOT_ADJUST_FINAL_STATE` affects both primary neutrons and charged particles, so be careful which is the use-case you are interested in. To summarize: if you use ParticleHP for primary neutrons, you can safely take the default; no harm is expected if you build ParticleHP with `DO_NOT_SET_PHP_AS_HP` set; be very careful instead if you set `G4PHP_DO_NOT_ADJUST_FINAL_STATE`. If you use ParticleHP for primary charged particles, then it is recommended to build with `DO_NOT_SET_PHP_AS_HP` set, and then run with `DO_NOT_SET_PHP_AS_HP` set.

Switching statistical nuclear de-excitation models

Nuclear reactions at intermediate energies (from a few MeV to a few GeV) are typically modelled in two stages. The first, fast reaction stage is described by a dynamical model (quantum molecular dynamics, intranuclear cascade, pre-compound, etc.) and often results in the production of one or several excited nuclei. The second reaction stage describes the de-excitation of the excited nuclei and it is usually handled by statistical de-excitation models. The models for the two reaction stages can in principle be chosen independently, but the current design of the Geant4 hadronics framework makes it difficult to do this at the physics-list level. However, another solution exists.

Geant4 provides several nuclear de-excitation modules. The default one is `G4ExcitationHandler`, which is described in detail in the Physics Reference Manual. The Bertini-style `G4CascadeInterface` uses an internal de-excitation model. The *ABLA V3* model is also available.

Options are available for steering of the pre-compound model and the de-excitation module. These options may be invoked by the new C++ interface class `G4DeexPrecoParameters`. The interface `G4NuclearLevelData::Instance()->GetParameters()` is thread safe, parameters are shared between threads, and parameters are shared between all de-excitation and pre-compound classes. Parameters may be modified at G4State_PreInit state of Geant4. This class has the following public methods:

- Dump()
- StreamInfo(std::ostream&)
- SetLevelDensity(G4double)
- SetR0(G4double)
- SetTransitionsR0(G4double)
- SetFermiEnergy(G4double)
- SetPrecoLowEnergy(G4double)
- SetPhenoFactor(G4double)
- SetMinExcitation(G4double)
- SetMaxLifeTime(G4double)
- SetMinExPerNucleounForMF(G4double)
- SetMinEForMultiFrag(G4double)
- SetMinZForPreco(G4int)
- SetMinAForPreco(G4int)
- SetPrecoModelType(G4int)
- SetDeexModelType(G4int)
- SetNeverGoBack(G4bool)
- SetUseSoftCutoff(G4bool)

- SetUseCEM(G4bool)
- SetUseHETC(G4bool)
- SetUseAngularGen(G4bool)
- SetUseLongFiles(G4bool)
- SetCorrelatedGamma(G4bool)
- SetStoreAllLevels(G4bool)
- SetDeexChannelType(G4DeexChannelType)

It is possible to replace the default de-excitation model with *ABLA V3* for any intranuclear-cascade model in Geant4 except `G4CascadeInterface`. The easiest way to do this is to call the `SetDeExcitation()` method of the relevant intranuclear-cascade-model interface. This can be done even if you are using one of the reference physics lists. The technique is the following.

For clarity's sake, assume you are using the *FTFP_INCLXX* physics list, which uses *INCL++*, the Liege Intranuclear Cascade model (`G4INCLXXInterface`) at intermediate energies. You can couple *INCL++* to *ABLA V3* by adding a run action (*Usage of User Actions*) and adding the following code snippet to `BeginOfRunAction()`.

Listing 5.1: Coupling the INCL++ model to ABLA V3

```
#include "G4HadronicInteraction.hh"
#include "G4HadronicInteractionRegistry.hh"
#include "G4INCLXXInterface.hh"
#include "G4AblaInterface.hh"

void MyRunAction::BeginOfRunAction(const G4Run*)
{
  // Get hold of pointers to the INCL++ model interfaces
  std::vector<G4HadronicInteraction *> interactions = G4HadronicInteractionRegistry::Instance()
    ->FindAllModels(G4INCLXXInterfaceStore::GetInstance()->getINCLXXVersionName());
  for(std::vector<G4HadronicInteraction *>::const_iterator iInter=interactions.begin(),␣
↪e=interactions.end();
      iInter!=e; ++iInter) {
    G4INCLXXInterface *theINCLInterface = static_cast<G4INCLXXInterface*>(*iInter);
    if(theINCLInterface) {
      // Instantiate the ABLA model
      G4HadronicInteraction *interaction = G4HadronicInteractionRegistry::Instance()->FindModel(
↪"ABLA");
      G4AblaInterface *theAblaInterface = static_cast<G4AblaInterface*>(interaction);
      if(!theAblaInterface)
        theAblaInterface = new G4AblaInterface;
      // Couple INCL++ to ABLA
      G4cout << "Coupling INCLXX to ABLA" << G4endl;
      theINCLInterface->SetDeExcitation(theAblaInterface);
    }
  }
}
```

This technique may be applied to any intranuclear-cascade model (i.e. models that inherit from `G4VIntraNuclearTransportModel`), except `G4CascadeInterface`. For example, if your physics list relies on the *Binary-Cascade* model (e.g. *FTF_BIC*), you'll need to do

```
// Get hold of a pointer to the Binary-Cascade model interface
std::vector<G4HadronicInteraction *> interactions = G4HadronicInteractionRegistry::Instance()
  ->FindAllModels("Binary Cascade");
for(std::vector<G4HadronicInteraction *>::const_iterator iInter=interactions.begin(),␣
↪e=interactions.end();
    iInter!=e; ++iInter) {
  G4BinaryCascade *theBICInterface = static_cast<G4BinaryCascade*>(*iInter);
  if(theBICInterface) {

    // Instantiate ABLA V3 as in the example above
    // [...]

    // Couple BIC to ABLA
```

```
    theBICInterface->SetDeExcitation(theAblaInterface);
  }
}
```

5.2.3 Particle Decay Process

This section briefly introduces decay processes installed in Geant4. For details of the implementation of particle decays, please refer to the Physics Reference Manual.

Particle Decay Class

Geant4 provides a `G4Decay` class for both `at rest` and `in flight` particle decays. `G4Decay` can be applied to all particles except:

massless particles, i.e., `G4ParticleDefinition::thePDGMass <= 0`
particles with "negative" life time, i.e., `G4ParticleDefinition::thePDGLifeTime < 0`
shortlived particles, i.e., `G4ParticleDefinition::fShortLivedFlag = True`

Decay for some particles may be switched on or off by using `G4ParticleDefinition::SetPDGStable()` as well as `ActivateProcess()` and `InActivateProcess()` methods of `G4ProcessManager`.

`G4Decay` proposes the step length (or step time for `AtRest`) according to the lifetime of the particle unless `PreAssignedDecayProperTime` is defined in `G4DynamicParticle`.

The `G4Decay` class itself does not define decay modes of the particle. Geant4 provides two ways of doing this:

- using `G4DecayChannel` in `G4DecayTable`, and
- using `thePreAssignedDecayProducts` of `G4DynamicParticle`

The `G4Decay` class calculates the `PhysicalInteractionLength` and boosts decay products created by `G4VDecayChannel` or event generators. See below for information on the determination of the decay modes.

An object of `G4Decay` can be shared by particles. Registration of the decay process to particles in the `ConstructPhysics` method of *PhysicsList* (see *How to Specify Physics Processes*) is shown in the *example*.

Listing 5.2: Registration of the decay process to particles in the `ConstructPhysics` method of *PhysicsList*.

```
#include "G4Decay.hh"
void MyPhysicsList::ConstructGeneral()
{
  // Add Decay Process
  G4Decay* theDecayProcess = new G4Decay();
  theParticleIterator->reset();
  while( (*theParticleIterator)() ){
    G4ParticleDefinition* particle = theParticleIterator->value();
    G4ProcessManager* pmanager = particle->GetProcessManager();
    if (theDecayProcess->IsApplicable(*particle)) {
      pmanager ->AddProcess(theDecayProcess);
      // set ordering for PostStepDoIt and AtRestDoIt
      pmanager ->SetProcessOrdering(theDecayProcess, idxPostStep);
      pmanager ->SetProcessOrdering(theDecayProcess, idxAtRest);
    }
  }
}
```

Decay Table

Each particle has its G4DecayTable, which stores information on the decay modes of the particle. Each decay mode, with its branching ratio, corresponds to an object of various "decay channel" classes derived from G4VDecayChannel. Default decay modes are created in the constructors of particle classes. For example, the decay table of the neutral pion has G4PhaseSpaceDecayChannel and G4DalitzDecayChannel as follows:

```
// create a decay channel
G4VDecayChannel* mode;
// pi0 -> gamma + gamma
mode = new G4PhaseSpaceDecayChannel("pi0",0.988,2,"gamma","gamma");
table->Insert(mode);
// pi0 -> gamma + e+ + e-
mode = new G4DalitzDecayChannel("pi0",0.012,"e-","e+");
table->Insert(mode);
```

Decay modes and branching ratios defined in Geant4 are listed in *Definition of a particle*.

Branching ratios and life time can be set in tracking time.

```
// set lifetime
G4Neutron::Neutron()->SetPDGLifeTime(885.7*second);
// allow neutron decay
G4Neutron::Neutron()->SetPDGStable(false);
```

Branching ratios and life time can be modified by using user commands, also.

Example: Set 100% br for dalitz decay of pi0

```
Idle> /particle/select pi0
Idle> /particle/property/decay/select 0
Idle> /particle/property/decay/br 0
Idle> /particle/property/decay/select 1
Idle> /particle/property/decay/br 1
Idle> /particle/property/decay/dump
      G4DecayTable:  pi0
         0:  BR:  0  [Phase Space]   :   gamma gamma
         1:  BR:  1  [Dalitz Decay]  :   gamma e- e+
```

Pre-assigned Decay Modes by Event Generators

Decays of heavy flavor particles such as B mesons are very complex, with many varieties of decay modes and decay mechanisms. There are many models for heavy particle decay provided by various event generators and it is impossible to define all the decay modes of heavy particles by using G4VDecayChannel. In other words, decays of heavy particles cannot be defined by the Geant4 decay process, but should be defined by event generators or other external packages. Geant4 provides two ways to do this: pre-assigned decay mode and external decayer.

In the latter approach, the class G4VExtDecayer is used for the interface to an external package which defines decay modes for a particle. If an instance of G4VExtDecayer is attached to G4Decay, daughter particles will be generated by the external decay handler.

In the former case, decays of heavy particles are simulated by an event generator and the primary event contains the decay information. G4VPrimaryGenerator automatically attaches any daughter particles to the parent particle as the PreAssignedDecayProducts member of G4DynamicParticle. G4Decay adopts these pre-assigned daughter particles instead of asking G4VDecayChannel to generate decay products.

In addition, the user may assign a pre-assigned decay time for a specific track in its rest frame (i.e. decay time is defined in the proper time) by using the G4PrimaryParticle::SetProperTime() method. G4VPrimaryGenerator sets the PreAssignedDecayProperTime member of G4DynamicParticle. G4Decay uses this decay time instead of the life time of the particle type.

5.2.4 Gamma-nuclear and Lepto-nuclear Processes

Gamma-nuclear and lepto-nuclear reactions are handled in Geant4 as hybrid processes which typically require both electromagnetic and hadronic models for their implementation. While neutrino-induced reactions are not currently provided, the Geant4 hadronic framework is general enough to include their future implementation as a hybrid of weak and hadronic models.

The general scheme followed is to factor the full interaction into an electromagnetic (or weak) vertex, in which a virtual particle is generated, and a hadronic vertex in which the virtual particle interacts with a target nucleus. In most cases the hadronic vertex is implemented by an existing Geant4 model which handles the intra-nuclear propagation.

The cross sections for these processes are parameterizations, either directly of data or of theoretical distributions determined from the integration of lepton-nucleon cross sections double differential in energy loss and momentum transfer.

For the most part gammas can be treated as hadrons and in fact they interact that way with the nucleus when the Bertini-style cascade `G4CascadeInterface` and *QGSP* models are used. These models may be assigned to `G4PhotoNuclearProcess` as shown in the following partial code:

```
G4TheoFSGenerator* theHEModel = new G4TheoFSGenerator;
G4QGSModel* theStringModel = new G4QGSModel<G4GammaParticipants>;
G4ExcitedStringDecay* theStringDecay =
  new G4ExcitedStringDecay(theFragmentation=new G4QGSMFragmentation);
theStringModel->SetFragmentationModel(theStringDecay);

theHEModel->SetHighEnergyGenerator(theStringModel);
theHEModel->SetTransport(new G4GeneratorPrecompoundInterface);
theHEModel->SetMinEnergy(8*GeV);

G4CascadeInterface* theLEModel = new G4CascadeInterface;
theLEModel->SetMaxEnergy(10*GeV);

G4PhotoNuclearProcess* thePhotoNuclearProcess = new G4PhotoNuclearProcess;
thePhotoNuclearProcess->RegisterMe(theLEModel);
thePhotoNuclearProcess->RegisterMe(theHEModel);

G4ProcessManager* procMan = G4Gamma::Gamma()->GetProcessManager();
procMan->AddDiscreteProcess(thePhotoNuclearProcess);
```

Electro-nuclear reactions in Geant4 are handled by the classes `G4ElectronNuclearProcess` and `G4PositronNuclearProcess`, which are both implmented by `G4ElectroVDNuclearModel`. This model consists of three sub-models: code which generates the virtual photon from the lepton-nucleus vertex, the Bertini-style cascade to handle the low and medium energy photons, and the FTFP model to handle the high energy photons.

Muon-nuclear reactions are handled similarly. The process `G4MuonNuclearProcess` can be assigned the `G4MuonVDNuclearModel` which in turn is implemented by three sub-models: virtual gamma generation code, Bertini-style cascade and the FTFP model.

5.2.5 Optical Photon Processes

A photon is considered to be *optical* when its wavelength is much greater than the typical atomic spacing. In GEANT4 optical photons are treated as a class of particle distinct from their higher energy *gamma* cousins. This implementation allows the wave-like properties of electromagnetic radiation to be incorporated into the optical photon process. Because this theoretical description breaks down at higher energies, there is no smooth transition as a function of energy between the optical photon and gamma particle classes.

For the simulation of optical photons to work correctly in GEANT4, they must be imputed a linear polarization. This is unlike most other particles in GEANT4 but is automatically and correctly done for optical photons that are generated as secondaries by existing processes in GEANT4. Not so, if the user wishes to start optical photons as primary particles. In this case, the user must set the linear polarization using particle gun methods, the General Particle Source, or his/her

PrimaryGeneratorAction. For an unpolarized source, the linear polarization should be sampled randomly for each new primary photon.

The GEANT4 catalogue of processes at optical wavelengths includes refraction and reflection at medium boundaries, bulk absorption, Mie and Rayleigh scattering. Processes which produce optical photons include the Cerenkov effect and scintillation. Optical photons are generated in GEANT4 without energy conservation and their energy must therefore not be tallied as part of the energy balance of an event.

The optical properties of the medium which are key to the implementation of these types of processes are stored as entries in a `G4MaterialPropertiesTable` which is linked to the `G4Material` in question. These properties may be constants or they may be expressed as a function of the photon's energy. This table is a private data member of the `G4Material` class. The `G4MaterialPropertiesTable` is implemented as a hash directory, in which each entry consists of a *value* and a *key*. The key is used to quickly and efficiently retrieve the corresponding value. All values in the dictionary are either instantiations of `G4double` or the class `G4MaterialPropertyVector`, and all keys are of type `G4String`.

A `G4MaterialPropertyVector` is a typedef of G4PhysicsOrderedFreeVector. The entries are a pair of numbers, which in the case of an optical property, are the photon energy and corresponding property value. It is possible for the user to add as many material (optical) properties to the material as he wishes using the methods supplied by the `G4MaterialPropertiesTable` class. An example of this is shown in *the example*. In this example the interpolation of the G4MaterialPropertyVector is to be done by a spline fit. The default is a linear interpolation.

Listing 5.3: Optical properties added to a `G4MaterialPropertiesTable` and linked to a `G4Material`

```
const G4int NUMENTRIES = 32;

G4double ppckov[NUMENTRIES] = {2.034*eV, ......, 4.136*eV};
G4double rindex[NUMENTRIES] = {1.3435, ......, 1.3608};
G4double absorption[NUMENTRIES] = {344.8*cm, ......, 1450.0*cm];

G4MaterialPropertiesTable *MPT = new G4MaterialPropertiesTable();

MPT -> AddConstProperty("SCINTILLATIONYIELD",100./MeV);

MPT -> AddProperty("RINDEX",ppckov,rindex,NUMENTRIES}->SetSpline(true);
MPT -> AddProperty("ABSLENGTH",ppckov,absorption,NUMENTRIES}->SetSpline(true);

scintillator -> SetMaterialPropertiesTable(MPT);
```

Generation of Photons in `processes/electromagnetic/xrays` - Cerenkov Effect

The radiation of Cerenkov light occurs when a charged particle moves through a dispersive medium faster than the group velocity of light in that medium. Photons are emitted on the surface of a cone, whose opening angle with respect to the particle's instantaneous direction decreases as the particle slows down. At the same time, the frequency of the photons emitted increases, and the number produced decreases. When the particle velocity drops below the local speed of light, the radiation ceases and the emission cone angle collapses to zero. The photons produced by this process have an inherent polarization perpendicular to the cone's surface at production.

The flux, spectrum, polarization and emission of Cerenkov radiation in the `AlongStepDoIt` method of the class `G4Cerenkov` follow well-known formulae, with two inherent computational limitations. The first arises from step-wise simulation, and the second comes from the requirement that numerical integration calculate the average number of Cerenkov photons per step. The process makes use of a `G4PhysicsTable` which contains incremental integrals to expedite this calculation. The Cerenkov process in Geant4 is limited to normally dispersive media, i.e., dn(E)/dE GE 0.

The time and position of Cerenkov photon emission are calculated from quantities known at the beginning of a charged particle's step. The step is assumed to be rectilinear even in the presence of a magnetic field. The user may limit the step size by specifying a maximum (average) number of Cerenkov photons created during the step,

using the `SetMaxNumPhotonsPerStep(const G4int NumPhotons)` method. The actual number generated will necessarily be different due to the Poissonian nature of the production. In the present implementation, the production density of photons is distributed evenly along the particle's track segment, even if the particle has slowed significantly during the step. The step can also be limited with the `SetMaxBetaChangePerStep` method, where the argument is the allowed change in percent).

The frequently very large number of secondaries produced in a single step (about 300/cm in water), compelled the idea in GEANT3.21 of suspending the primary particle until all its progeny have been tracked. Despite the fact that GEANT4 employs dynamic memory allocation and thus does not suffer from the limitations of GEANT3.21 with its fixed large initial ZEBRA store, GEANT4 nevertheless provides for an analogous functionality with the public method `SetTrackSecondariesFirst`. An example of the registration of the Cerenkov process is given in *the example*.

Listing 5.4: Registration of the Cerenkov process in `PhysicsList`.

```
#include "G4Cerenkov.hh"

void ExptPhysicsList::ConstructOp(){

  G4Cerenkov*   theCerenkovProcess = new G4Cerenkov("Cerenkov");

  G4int MaxNumPhotons = 300;

  theCerenkovProcess->SetTrackSecondariesFirst(true);
  theCerenkovProcess->SetMaxBetaChangePerStep(10.0);
  theCerenkovProcess->SetMaxNumPhotonsPerStep(MaxNumPhotons);

  theParticleIterator->reset();
  while( (*theParticleIterator)() ){
    G4ParticleDefinition* particle = theParticleIterator->value();
    G4ProcessManager* pmanager = particle->GetProcessManager();
    G4String particleName = particle->GetParticleName();
    if (theCerenkovProcess->IsApplicable(*particle)) {
      pmanager->AddProcess(theCerenkovProcess);
      pmanager->SetProcessOrdering(theCerenkovProcess,idxPostStep);
    }
  }
}
```

Generation of Photons in `processes/electromagnetic/xrays` - Scintillation

Every scintillating material has a characteristic light yield, `SCINTILLATIONYIELD`, and an intrinsic resolution, `RESOLUTIONSCALE`, which generally broadens the statistical distribution of generated photons. A wider intrinsic resolution is due to impurities which are typical for doped crystals like NaI(Tl) and CsI(Tl). On the other hand, the intrinsic resolution can also be narrower when the Fano factor plays a role. The actual number of emitted photons during a step fluctuates around the mean number of photons with a width given by `ResolutionScale*sqrt(MeanNumberOfPhotons)`. The average light yield, `MeanNumberOfPhotons`, has a linear dependence on the local energy deposition, but it may be different for minimum ionizing and non-minimum ionizing particles.

A scintillator is also characterized by its photon emission spectrum and by the exponential decay of its time spectrum. In GEANT4 the scintillator can have a fast and a slow component. The relative strength of the fast component as a fraction of total scintillation yield is given by the `YIELDRATIO`. Scintillation may be simulated by specifying these empirical parameters for each material. It is sufficient to specify in the user's `DetectorConstruction` class a relative spectral distribution as a function of photon energy for the scintillating material. An example of this is shown in *the example*.

Listing 5.5: Specification of scintillation properties in `DetectorConstruction`.

```
const G4int NUMENTRIES = 9;
G4double Scnt_PP[NUMENTRIES] = { 6.6*eV, 6.7*eV, 6.8*eV, 6.9*eV,
                                 7.0*eV, 7.1*eV, 7.2*eV, 7.3*eV, 7.4*eV };

G4double Scnt_FAST[NUMENTRIES] = { 0.000134, 0.004432, 0.053991, 0.241971,
                                   0.398942, 0.000134, 0.004432, 0.053991,
                                   0.241971 };
G4double Scnt_SLOW[NUMENTRIES] = { 0.000010, 0.000020, 0.000030, 0.004000,
                                   0.008000, 0.005000, 0.020000, 0.001000,
                                   0.000010 };

G4Material* Scnt;
G4MaterialPropertiesTable* Scnt_MPT = new G4MaterialPropertiesTable();

Scnt_MPT->AddProperty("FASTCOMPONENT", Scnt_PP, Scnt_FAST, NUMENTRIES);
Scnt_MPT->AddProperty("SLOWCOMPONENT", Scnt_PP, Scnt_SLOW, NUMENTRIES);

Scnt_MPT->AddConstProperty("SCINTILLATIONYIELD", 5000./MeV);
Scnt_MPT->AddConstProperty("RESOLUTIONSCALE", 2.0);
Scnt_MPT->AddConstProperty("FASTTIMECONSTANT",  1.*ns);
Scnt_MPT->AddConstProperty("SLOWTIMECONSTANT", 10.*ns);
Scnt_MPT->AddConstProperty("YIELDRATIO", 0.8);

Scnt->SetMaterialPropertiesTable(Scnt_MPT);
```

In cases where the scintillation yield of a scintillator depends on the particle type, different scintillation processes may be defined for them. How this yield scales to the one specified for the material is expressed with the `ScintillationYieldFactor` in the user's `PhysicsList` as shown in the *example* below. In those cases where the fast to slow excitation ratio changes with particle type, the method `SetScintillationExcitationRatio` can be called for each scintillation process (see the advanced underground_physics example). This overwrites the `YieldRatio` obtained from the `G4MaterialPropertiesTable`.

Listing 5.6: Implementation of the scintillation process in `PhysicsList`.

```
G4Scintillation* theMuonScintProcess = new G4Scintillation("Scintillation");

theMuonScintProcess->SetTrackSecondariesFirst(true);
theMuonScintProcess->SetScintillationYieldFactor(0.8);

theParticleIterator->reset();
while( (*theParticleIterator)() ){
  G4ParticleDefinition* particle = theParticleIterator->value();
  G4ProcessManager* pmanager = particle->GetProcessManager();
  G4String particleName = particle->GetParticleName();
  if (theMuonScintProcess->IsApplicable(*particle)) {
     if (particleName == "mu+") {
        pmanager->AddProcess(theMuonScintProcess);
        pmanager->SetProcessOrderingToLast(theMuonScintProcess, idxAtRest);
        pmanager->SetProcessOrderingToLast(theMuonScintProcess, idxPostStep);
     }
  }
}
```

A Gaussian-distributed number of photons is generated according to the energy lost during the step. A resolution scale of 1.0 produces a statistical fluctuation around the average yield set with `AddConstProperty("SCINTILLATIONYIELD")`, while values > 1 broaden the fluctuation. A value of zero produces no fluctuation. Each photon's frequency is sampled from the empirical spectrum. The photons originate evenly along the track segment and are emitted uniformly into 4PI with a random linear polarization and at times characteristic for the scintillation component.

When there are multiple scintillators in the simulation and/or when the scintillation yield is a non-linear function of the

energy deposited, the user can also define an array of total scintillation light yields as a function of the energy deposited and particle type. The available particles are protons, electrons, deuterons, tritons, alphas, and carbon ions. These are the particles known to significantly effect the scintillation light yield, of for example, BC501A (NE213/EJ301) liquid organic scintillator and BC420 plastic scintillator as function of energy deposited.

The method works as follows:

1. In the user's physics lists, the user must set a G4bool flag that allows scintillation light emission to depend on the energy deposited by particle type:

```
theScintProcess->SetScintillationByParticleType(true);
```

2. The user must also specify and add, via the AddProperty method of the MPT, the scintillation light yield as function of incident particle energy with new keywords, for example: PROTONSCINTILLATIONYIELD etc. and pairs of protonEnergy and scintLightYield.

Generation of Photons in `processes/optical` - Wavelength Shifting

Wavelength Shifting (WLS) fibers are used in many high-energy particle physics experiments. They absorb light at one wavelength and re-emit light at a different wavelength and are used for several reasons. For one, they tend to decrease the self-absorption of the detector so that as much light reaches the PMTs as possible. WLS fibers are also used to match the emission spectrum of the detector with the input spectrum of the PMT.

A WLS material is characterized by its photon absorption and photon emission spectrum and by a possible time delay between the absorption and re-emission of the photon. Wavelength Shifting may be simulated by specifying these empirical parameters for each WLS material in the simulation. It is sufficient to specify in the user's `DetectorConstruction` class a relative spectral distribution as a function of photon energy for the WLS material. WLSABSLENGTH is the absorption length of the material as a function of the photon's energy. WLSCOMPONENT is the relative emission spectrum of the material as a function of the photon's energy, and WLSTIMECONSTANT accounts for any time delay which may occur between absorption and re-emission of the photon. An example is shown in the *example*.

Listing 5.7: Specification of WLS properties in `DetectorConstruction`.

```
const G4int nEntries = 9;

G4double PhotonEnergy[nEntries] = { 6.6*eV, 6.7*eV, 6.8*eV, 6.9*eV,
                                    7.0*eV, 7.1*eV, 7.2*eV, 7.3*eV, 7.4*eV };

G4double RIndexFiber[nEntries] =
          { 1.60, 1.60, 1.60, 1.60, 1.60, 1.60, 1.60, 1.60, 1.60 };
G4double AbsFiber[nEntries] =
          {0.1*mm,0.2*mm,0.3*mm,0.4*cm,1.0*cm,10*cm,1.0*m,10.0*m,10.0*m};
G4double EmissionFiber[nEntries] =
          {0.0, 0.0, 0.0, 0.1, 0.5, 1.0, 5.0, 10.0, 10.0 };

 G4Material* WLSFiber;
 G4MaterialPropertiesTable* MPTFiber = new G4MaterialPropertiesTable();

 MPTFiber->AddProperty("RINDEX",PhotonEnergy,RIndexFiber,nEntries);
 MPTFiber->AddProperty("WLSABSLENGTH",PhotonEnergy,AbsFiber,nEntries);
 MPTFiber->AddProperty("WLSCOMPONENT",PhotonEnergy,EmissionFiber,nEntries);
 MPTFiber->AddConstProperty("WLSTIMECONSTANT", 0.5*ns);

 WLSFiber->SetMaterialPropertiesTable(MPTFiber);
```

The process is defined in the PhysicsList in the usual way. The process class name is G4OpWLS. It should be instantiated with theWLSProcess = new G4OpWLS("OpWLS") and attached to the process manager of the optical photon as a DiscreteProcess. The way the WLSTIMECONSTANT is used depends on the time profile method chosen by the user. If in the PhysicsList theWLSProcess->UseTimeGenerator("exponential") option is set, the time

delay between absorption and re-emission of the photon is sampled from an exponential distribution, with the decay term equal to WLSTIMECONSTANT. If, on the other hand, theWLSProcess->UseTimeGenerator("delta") is chosen, the time delay is a delta function and equal to WLSTIMECONSTANT. The default is "delta" in case the G4OpWLS::UseTimeGenerator(const G4String name) method is not used.

Tracking of Photons in `processes/optical`

Absorption

The implementation of optical photon bulk absorption, `G4OpAbsorption`, is trivial in that the process merely kills the particle. The procedure requires the user to fill the relevant `G4MaterialPropertiesTable` with empirical data for the absorption length, using `ABSLENGTH` as the property key in the public method `AddProperty`. The absorption length is the average distance traveled by a photon before being absorpted by the medium; i.e. it is the mean free path returned by the `GetMeanFreePath` method.

Rayleigh Scattering

The differential cross section in Rayleigh scattering, dSIGMA/dOMEGA, is proportional to $1+\cos^2$(THETAS), where THETAS is the polar of the new polarization vector with respect to the old polarization vector. The `G4OpRayleigh` scattering process samples this angle accordingly and then calculates the scattered photon's new direction by requiring that it be perpendicular to the photon's new polarization in such a way that the final direction, initial and final polarizations are all in one plane. This process thus depends on the particle's polarization (spin). The photon's polarization is a data member of the `G4DynamicParticle` class.

A photon which is not assigned a polarization at production, either via the `SetPolarization` method of the `G4PrimaryParticle` class, or indirectly with the `SetParticlePolarization` method of the `G4ParticleGun` class, may not be Rayleigh scattered. Optical photons produced by the `G4Cerenkov` process have inherently a polarization perpendicular to the cone's surface at production. Scintillation photons have a random linear polarization perpendicular to their direction.

The process requires a `G4MaterialPropertiesTable` to be filled by the user with Rayleigh scattering length data. The Rayleigh scattering attenuation length is the average distance traveled by a photon before it is Rayleigh scattered in the medium and it is the distance returned by the `GetMeanFreePath` method. The `G4OpRayleigh` class provides a `RayleighAttenuationLengthGenerator` method which calculates the attenuation coefficient of a medium following the Einstein-Smoluchowski formula whose derivation requires the use of statistical mechanics, includes temperature, and depends on the isothermal compressibility of the medium. This generator is convenient when the Rayleigh attenuation length is not known from measurement but may be calculated from first principles using the above material constants. For a medium named *Water* and no Rayleigh scattering attenutation length specified by the user, the program automatically calls the `RayleighAttenuationLengthGenerator` which calculates it for 10 degrees Celsius liquid water.

Mie Scattering

Mie Scattering (or Mie solution) is an analytical solution of Maxwell's equations for scattering of optical photons by spherical particles. It is significant only when the radius of the scattering object is of order of the wave length. The analytical expressions for Mie Scattering are very complicated since they are a series sum of Bessel functions. One common approximation made is call *Henyey-Greenstein (HG)*. The implementation in Geant4 follows the HG approximation (for details see the Physics Reference Manual) and the treatment of polarization and momentum are similar to that of Rayleigh scattering. We require the final polarization direction to be perpendicular to the momentum direction. We also require the final momentum, initial polarization and final polarization to be in the same plane.

The process requires a G4MaterialPropertiesTable to be filled by the user with Mie scattering length data (entered with the name: MIEHG) analogous to Rayleigh scattering. The Mie scattering attenuation length is the average distance traveled by a photon before it is Mie scattered in the medium and it is the distance returned by the GetMeanFreePath method. In practice, the user not only needs to provide the attenuation length of Mie scattering, but also needs to provide the constant parameters of the approximation: g_f, g_b, and r_f. (with AddConstProperty and with the

names: MIEHG_FORWARD, MIEHG_BACKWARD, and MIEHG_FORWARD_RATIO, respectively; see Novice Example N06.)

Boundary Process

Reference: E. Hecht and A. Zajac, Optics *[Hecht1974]*

For the simple case of a perfectly smooth interface between two dielectric materials, all the user needs to provide are the refractive indices of the two materials stored in their respective `G4MaterialPropertiesTable`. In all other cases, the optical boundary process design relies on the concept of *surfaces*. The information is split into two classes. One class in the material category keeps information about the physical properties of the surface itself, and a second class in the geometry category holds pointers to the relevant physical and logical volumes involved and has an association to the physical class. Surface objects of the second type are stored in a related table and can be retrieved by either specifying the two ordered pairs of physical volumes touching at the surface, or by the logical volume entirely surrounded by this surface. The former is called a *border surface* while the latter is referred to as the *skin surface*. This second type of surface is useful in situations where a volume is coded with a reflector and is placed into many different mother volumes. A limitation is that the skin surface can only have one and the same optical property for all of the enclosed volume's sides. The border surface is an ordered pair of physical volumes, so in principle, the user can choose different optical properties for photons arriving from the reverse side of the same interface. For the optical boundary process to use a border surface, the two volumes must have been positioned with `G4PVPlacement`. The ordered combination can exist at many places in the simulation. When the surface concept is not needed, and a perfectly smooth surface exists beteen two dielectic materials, the only relevant property is the index of refraction, a quantity stored with the material, and no restriction exists on how the volumes were positioned.

When an optical photon arrives at a boundary it is absorbed if the medium of the volume being left behind has no index of refraction defined. A photon is also absorbed in case of a dielectric-dielectric polished or ground surface when the medium about to be entered has no index of refraction. It is absorbed for backpainted surfaces when the surface has no index of refraction. If the geometry boundary has a *border surface* this surface takes precedence, otherwise the program checks for *skin surfaces*. The *skin surface* of the daughter volume is taken if a daughter volume is entered else the program checks for a *skin surface* of the current volume. When the optical photon leaves a volume without entering a daughter volume the *skin surface* of the current volume takes precedence over that of the volume about to be entered.

The physical surface object also specifies which model the boundary process should use to simulate interactions with that surface. In addition, the physical surface can have a material property table all its own. The usage of this table allows all specular constants to be wavelength dependent. In case the surface is painted or wrapped (but not a cladding), the table may include the thin layer's index of refraction. This allows the simulation of boundary effects at the intersection between the medium and the surface layer, as well as the Lambertian reflection at the far side of the thin layer. This occurs within the process itself and does not invoke the `G4Navigator`. Combinations of surface finish properties, such as *polished* or *ground* and *front painted* or *back painted*, enumerate the different situations which can be simulated.

When a photon arrives at a medium boundary its behavior depends on the nature of the two materials that join at that boundary. Medium boundaries may be formed between two dielectric materials or a dielectric and a metal. In the case of two dielectric materials, the photon can undergo total internal reflection, refraction or reflection, depending on the photon's wavelength, angle of incidence, and the refractive indices on both sides of the boundary. Furthermore, reflection and transmission probabilites are sensitive to the state of linear polarization. In the case of an interface between a dielectric and a metal, the photon can be absorbed by the metal or reflected back into the dielectric. If the photon is absorbed it can be detected according to the photoelectron efficiency of the metal.

As expressed in Maxwell's equations, Fresnel reflection and refraction are intertwined through their relative probabilities of occurrence. Therefore neither of these processes, nor total internal reflection, are viewed as individual processes deserving separate class implementation. Nonetheless, an attempt was made to adhere to the abstraction of having independent processes by splitting the code into different methods where practicable.

One implementation of the `G4OpBoundaryProcess` class employs the UNIFIED model [A. Levin and C. Moisan, A More Physical Approach to Model the Surface Treatment of Scintillation Counters and its Implementation into DETECT, TRIUMF Preprint TRI-PP-96-64, Oct. 1996] of the DETECT program [G.F. Knoll, T.F. Knoll and T.M.

Henderson, Light Collection Scintillation Detector Composites for Neutron Detection, IEEE Trans. Nucl. Sci., 35 (1988) 872.]. It applies to dielectric-dielectric interfaces and tries to provide a realistic simulation, which deals with all aspects of surface finish and reflector coating. The surface may be assumed as smooth and covered with a metallized coating representing a specular reflector with given reflection coefficient, or painted with a diffuse reflecting material where Lambertian reflection occurs. The surfaces may or may not be in optical contact with another component and most importantly, one may consider a surface to be made up of micro-facets with normal vectors that follow given distributions around the nominal normal for the volume at the impact point. For very rough surfaces, it is possible for the photon to inversely aim at the same surface again after reflection of refraction and so multiple interactions with the boundary are possible within the process itself and without the need for relocation by `G4Navigator`.

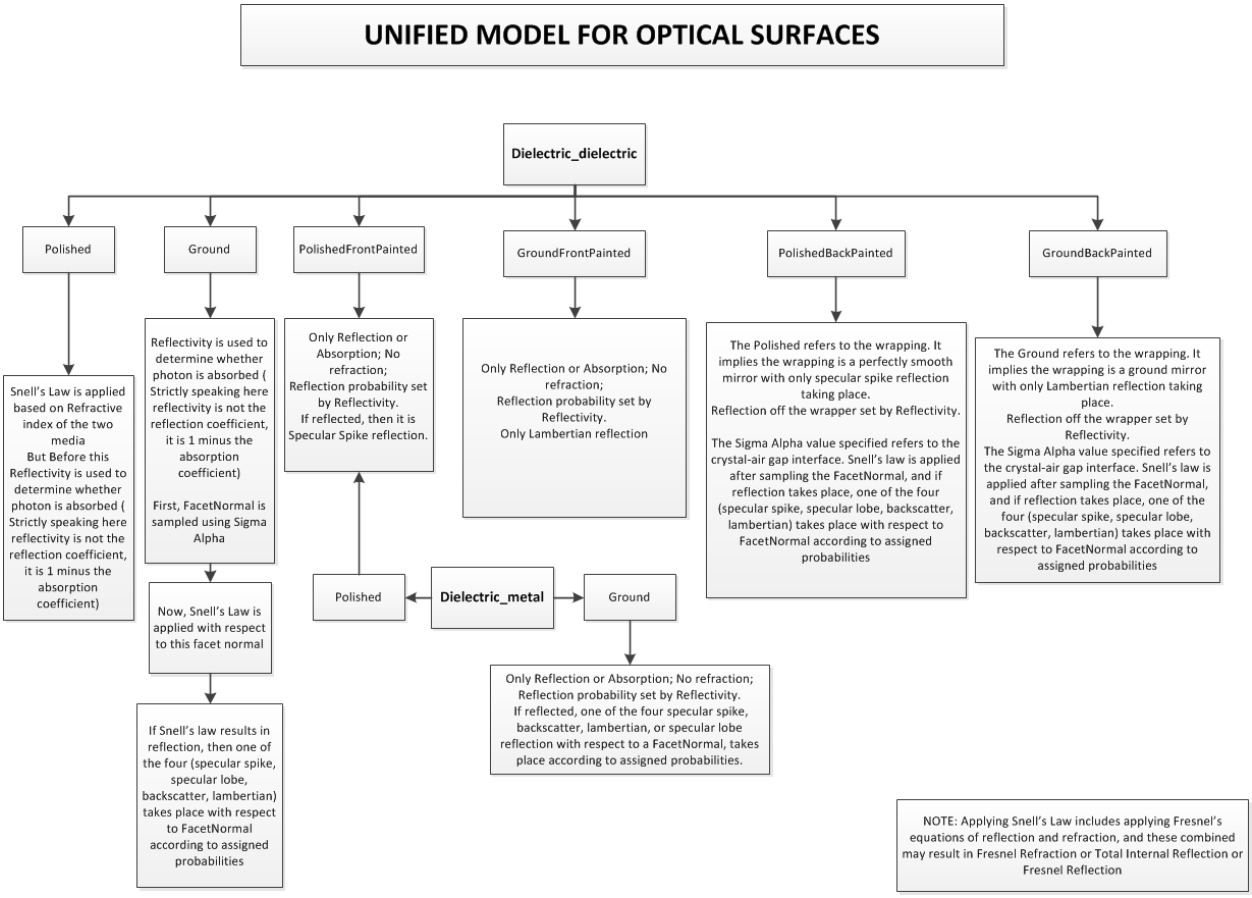


Fig. 5.1: Diagram of the UNIFIED Model for Optical Surfaces (courtesy A. Shankar)

The UNIFIED model (*figure*) provides for a range of different reflection mechanisms. The specular lobe constant represents the reflection probability about the normal of a micro facet. The specular spike constant, in turn, illustrates the probability of reflection about the average surface normal. The diffuse lobe constant is for the probability of internal Lambertian reflection, and finally the back-scatter spike constant is for the case of several reflections within a deep groove with the ultimate result of exact back-scattering. The four probabilities must add up to one, with the diffuse lobe constant being implicit. The reader may consult the reference for a thorough description of the model.

Listing 5.8: Dielectric-dielectric surface properties defined via the `G4OpticalSurface`.

```
G4VPhysicalVolume* volume1;
G4VPhysicalVolume* volume2;

G4OpticalSurface* OpSurface = new G4OpticalSurface("name");

G4LogicalBorderSurface* Surface = new
```

```
  G4LogicalBorderSurface("name",volume1,volume2,OpSurface);

G4double sigma_alpha = 0.1;

OpSurface -> SetType(dielectric_dielectric);
OpSurface -> SetModel(unified);
OpSurface -> SetFinish(groundbackpainted);
OpSurface -> SetSigmaAlpha(sigma_alpha);

const G4int NUM = 2;

G4double pp[NUM] = {2.038*eV, 4.144*eV};
G4double specularlobe[NUM] = {0.3, 0.3};
G4double specularspike[NUM] = {0.2, 0.2};
G4double backscatter[NUM] = {0.1, 0.1};
G4double rindex[NUM] = {1.35, 1.40};
G4double reflectivity[NUM] = {0.3, 0.5};
G4double efficiency[NUM] = {0.8, 0.1};

G4MaterialPropertiesTable* SMPT = new G4MaterialPropertiesTable();

SMPT -> AddProperty("RINDEX",pp,rindex,NUM);
SMPT -> AddProperty("SPECULARLOBECONSTANT",pp,specularlobe,NUM);
SMPT -> AddProperty("SPECULARSPIKECONSTANT",pp,specularspike,NUM);
SMPT -> AddProperty("BACKSCATTERCONSTANT",pp,backscatter,NUM);
SMPT -> AddProperty("REFLECTIVITY",pp,reflectivity,NUM);
SMPT -> AddProperty("EFFICIENCY",pp,efficiency,NUM);

OpSurface -> SetMaterialPropertiesTable(SMPT);
```

The original GEANT3.21 implementation of this process is also available via the GLISUR methods flag. [GEANT Detector Description and Simulation Tool, Application Software Group, Computing and Networks Division, CERN, PHYS260-6 tp 260-7.].

Listing 5.9: Dielectric metal surface properties defined via the `G4OpticalSurface`.

```
G4LogicalVolume* volume_log;

G4OpticalSurface* OpSurface = new G4OpticalSurface("name");

G4LogicalSkinSurface* Surface = new
  G4LogicalSkinSurface("name",volume_log,OpSurface);

OpSurface -> SetType(dielectric_metal);
OpSurface -> SetFinish(ground);
OpSurface -> SetModel(glisur);

G4double polish = 0.8;

G4MaterialPropertiesTable *OpSurfaceProperty = new G4MaterialPropertiesTable();

OpSurfaceProperty -> AddProperty("REFLECTIVITY",pp,reflectivity,NUM);
OpSurfaceProperty -> AddProperty("EFFICIENCY",pp,efficiency,NUM);

OpSurface -> SetMaterialPropertiesTable(OpSurfaceProperty);
```

The reflectivity off a metal surface can also be calculated by way of a complex index of refraction. Instead of storing the REFLECTIVITY directly, the user stores the real part (REALRINDEX) and the imaginary part (IMAGINARYRINDEX) as a function of photon energy separately in the G4MaterialPropertyTable. Geant4 then calculates the reflectivity depending on the incident angle, photon energy, degree of TE and TM polarization, and this complex refractive index.

The program defaults to the GLISUR model and *polished* surface finish when no specific model and surface finish is specified by the user. In the case of a dielectric-metal interface, or when the GLISUR model is specified, the only

surface finish options available are *polished* or *ground*. For dielectric-metal surfaces, the `G4OpBoundaryProcess` also defaults to unit reflectivity and zero detection efficiency. In cases where the user specifies the UNIFIED model (*fig-unified-surface-model*), but does not otherwise specify the model reflection probability constants, the default becomes Lambertian reflection.

Martin Janecek and Bill Moses (Lawrence Berkeley National Laboratory) built an instrument for measuring the angular reflectivity distribution inside of BGO crystals with common surface treatments and reflectors applied. These results have been incorporate into the Geant4 code. A third class of reflection type besides dielectric_metal and dielectric_dielectric is added: dielectric_LUT. The distributions have been converted to 21 look-up-tables (LUT); so far for 1 scintillator material (BGO) x 3 surface treatments x 7 reflector materials. The modified code allows the user to specify the surface treatment (rough-cut, chemically etched, or mechanically polished), the attached reflector (Lumirror, Teflon, ESR film, Tyvek, or TiO2 paint), and the bonding type (air-coupled or glued). The glue used is MeltMount, and the ESR film used is VM2000. Each LUT consists of measured angular distributions with 4º by 5º resolution in theta and phi, respectively, for incidence angles from 0º to 90º degrees, in 1º-steps. The code might in the future be updated by adding more LUTs, for instance, for other scintillating materials (such as LSO or NaI). To use these LUT the user has to download them from Geant4 Software Download and set an environment variable, `G4REALSURFACEDATA`, to the directory of `geant4/data/RealSurface1.0`. For details see: M. Janecek, W. W. Moses, IEEE Trans. Nucl. Sci. 57 (3) (2010) 964-970.

The enumeration G4OpticalSurfaceFinish has been extended to include (what follows should be a 2 column table):

```
polishedlumirrorair,          // mechanically polished surface, with lumirror
polishedlumirrorglue,         // mechanically polished surface, with lumirror & meltmount
polishedair,                  // mechanically polished surface
polishedteflonair,            // mechanically polished surface, with teflon
polishedtioair,               // mechanically polished surface, with tio paint
polishedtyvekair,             // mechanically polished surface, with tyvek
polishedvm2000air,            // mechanically polished surface, with esr film
polishedvm2000glue,           // mechanically polished surface, with esr film & meltmount
etchedlumirrorair,            // chemically etched surface, with lumirror
etchedlumirrorglue,           // chemically etched surface, with lumirror & meltmount
etchedair,                    // chemically etched surface
etchedteflonair,              // chemically etched surface, with teflon
etchedtioair,                 // chemically etched surface, with tio paint
etchedtyvekair,               // chemically etched surface, with tyvek
etchedvm2000air,              // chemically etched surface, with esr film
etchedvm2000glue,             // chemically etched surface, with esr film & meltmount
groundlumirrorair,            // rough-cut surface, with lumirror
groundlumirrorglue,           // rough-cut surface, with lumirror & meltmount
groundair,                    // rough-cut surface
groundteflonair,              // rough-cut surface, with teflon
groundtioair,                 // rough-cut surface, with tio paint
groundtyvekair,               // rough-cut surface, with tyvek
groundvm2000air,              // rough-cut surface, with esr film
groundvm2000glue              // rough-cut surface, with esr film & meltmount
```

To use a look-up-table, all the user needs to specify for an `G4OpticalSurface` is: `SetType(dielectric_LUT), SetModel(LUT)` and for example, `SetFinish(polishedtyvekair)`.

5.2.6 Parameterization

In this section we describe how to use the parameterization or "fast simulation" facilities of GEANT4. Examples are provided in the **examples/novice/N05 directory**.

Generalities:

The Geant4 parameterization facilities allow you to shortcut the detailed tracking in a given volume and for given particle types in order for you to provide your own implementation of the physics and of the detector response.

Parameterisations are bound to a **"G4Region"** object, which, in the case of fast simulation is also called an **envelope**. Prior to release 8.0, parameterisations were bound to a `G4LogicalVolume`, the root of a volume hierarchy. These root volumes are now attributes of the `G4Region`. Envelopes often correspond to the volumes of sub-detectors: electromagnetic calorimeters, tracking chambers, etc. With GEANT4 it is also possible to define envelopes by overlaying a parallel or "ghost" geometry as discussed in ' Parameterisation Using Ghost Geometries <#sect.PhysProc.Param.Ghost>'__.

In GEANT4, parameterisations have three main features. You must specify:

- the particle types for which your parameterisation is valid;
- the dynamics conditions for which your parameterisation is valid and must be triggered;
- the parameterisation itself: where the primary will be killed or moved, whether or not to create it or create secondaries, etc., and where the detector response will be computed.

GEANT4 will message your parameterisation code for each step starting in any root G4LogicalVolume (including daughters. sub-daughters, etc. of this volume) of the `G4Region`. It will proceed by first asking the available parameterisations for the current particle type if one of them (and only one) wants to issue a trigger. If so it will invoke its parameterisation. In this case, the tracking ***will not apply physics*** to the particle in the step. Instead, the UserSteppingAction will be invoked.

Parameterisations look like a "user stepping action" but are more advanced because:

- parameterisation code is messaged only in the G4Region to which it is bound;
- parameterisation code is messaged anywhere in the G4Region , that is, any volume in which the track is located;
- GEANT4 will provide information to your parameterisation code about the current root volume of the G4Region in which the track is travelling.

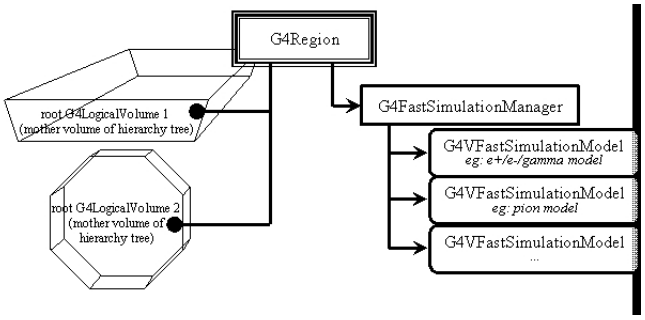
Overview of Parameterisation Components

The GEANT4 components which allow the implementation and control of parameterisations are:

`G4VFastSimulationModel` This is the abstract class for the implementation of parameterisations. You must inherit from it to implement your concrete parameterisation model.

`G4FastSimulationManager` The `G4VFastSimulationModel` objects are attached to the `G4Region` through a `G4FastSimulationManager`. This object will manage the list of models and will message them at tracking time.

`G4Region/Envelope` As mentioned before, an envelope in GEANT4 is a `G4Region`. The parameterisation is bound to the `G4Region` by setting a `G4FastSimulationManager` pointer to it.
The figure below shows how the `G4VFastSimulationModel` and `G4FastSimulationManager` objects are bound to the `G4Region`. Then for all root G4LogicalVolume's held by the G4Region, the fast simulation code is active.



`G4FastSimulationManagerProcess` This is a `G4VProcess`. It provides the interface between the tracking and the parameterisation. It must be set in the process list of the particles you want to parameterise.

`G4GlobalFastSimulationManager` This a singleton class which provides the management of the `G4FastSimulationManager` objects and some ghost facilities.

The G4VFastSimulationModel Abstract Class

Constructors:

The G4VFastSimulationModel class has two constructors. The second one allows you to get started quickly:

G4VFastSimulationModel(const G4String& aName): Here aName identifies the parameterisation model.

G4VFastSimulationModel(const G4String& aName, G4Region*, G4bool IsUnique=false): In addition to the model name, this constructor accepts a G4Region pointer. The needed G4FastSimulationManager object is constructed if necessary, passing to it the G4Region pointer and the boolean value. If it already exists, the model is simply added to this manager. Note that the G4VFastSimulationModel object *will not keep track* of the G4Region passed in the constructor. The boolean argument is there for optimization purposes: if you know that the G4Region has a unique root G4LogicalVolume, uniquely placed, you can set the boolean value to "true".

Virtual methods:

The G4VFastSimulationModel has three pure virtual methods which must be overriden in your concrete class:

G4VFastSimulationModel(const G4String& aName): Here aName identifies the parameterisation model.

G4bool ModelTrigger(const G4FastTrack&): You must return "true" when the dynamic conditions to trigger your parameterisation are fulfilled. G4FastTrack provides access to the current G4Track, gives simple access to the current root G4LogicalVolume related features (its G4VSolid, and G4AffineTransform references between the global and the root G4LogicalVolume local coordinates systems) and simple access to the position and momentum expressed in the root G4LogicalVolume coordinate system. Using these quantities and the G4VSolid methods, you can for example easily check how far you are from the root G4LogicalVolume boundary.

G4bool IsApplicable(const G4ParticleDefinition&): In your implementation, you must return "true" when your model is applicable to the G4ParticleDefinition passed to this method. The G4ParticleDefinition provides all intrinsic particle information (mass, charge, spin, name ...).
If you want to implement a model which is valid only for certain particle types, it is recommended for efficiency that you use the static pointer of the corresponding particle classes.
As an example, in a model valid for *gamma*s only, the IsApplicable() method should take the form:

```
#include "G4Gamma.hh"

G4bool MyGammaModel::IsApplicable(const G4ParticleDefinition& partDef)
{
  return &partDef == G4Gamma::GammaDefinition();
}
```

G4bool ModelTrigger(const G4FastTrack&): You must return "true" when the dynamic conditions to trigger your parameterisation are fulfilled. The G4FastTrack provides access to the current G4Track, gives simple access to envelope related features (G4LogicalVolume, G4VSolid, and G4AffineTransform references between the global and the envelope local coordinates systems) and simple access to the position and momentum expressed in the envelope coordinate system. Using these quantities and the G4VSolid methods, you can for example easily check how far you are from the envelope boundary.

void DoIt(const G4FastTrack&, G4FastStep&): The details of your parameterisation will be implemented in this method. The G4FastTrack reference provides the input information, and the final state of the particles after parameterisation must be returned through the G4FastStep reference. Tracking for the final state particles is requested after your parameterisation has been invoked.

The G4FastSimulationManager Class:

G4FastSimulationManager functionnalities regarding the use of ghost volumes are explained in *Parameterisation Using Ghost Geometries*.

Constructor:

`G4FastSimulationManager(G4Region *anEnvelope, G4bool IsUnique=false)`: This is the only constructor. You specify the G4Region by providing its pointer. The G4FastSimulationManager object will bind itself to this G4Region. If you know that this G4Region has a single root G4LogicalVolume, placed only once, you can set the IsUnique boolean to "true" to allow some optimization.
Note that if you choose to use the G4VFastSimulationModel(const G4String&, G4Region*, G4bool) constructor for your model, the G4FastSimulationManager will be constructed using the given G4Region* and G4bool values of the model constructor.

G4VFastSimulationModel object management:

The following two methods provide the usual management functions.

- `void AddFastSimulationModel( G4VFastSimulationModel*)`
- `RemoveFastSimulationModel( G4VFastSimulationModel*)`

The `G4FastSimulationManagerProcess` Class

This G4VProcess serves as an interface between the tracking and the parameterisation. At tracking time, it collaborates with the G4FastSimulationManager of the current volume, if any, to allow the models to trigger. If no manager exists or if no model issues a trigger, the tracking goes on normally.

In the present implementation, you must set this process in the G4ProcessManager of the particles you parameterise to enable your parameterisation.

The processes ordering is:

```
[n-3] ...
[n-2] Multiple Scattering
[n-1] G4FastSimulationManagerProcess
[ n ] G4Transportation
```

This ordering is important if you use ghost geometries, since the G4FastSimulationManagerProcess will provide navigation in the ghost world to limit the step on ghost boundaries.

The G4FastSimulationManager must be added to the process list of a particle as a continuous and discrete process if you use ghost geometries for this particle. You can add it as a discrete process if you don't use ghosts.

The following code registers the G4FastSimulationManagerProcess with all the particles as a discrete and continuous process:

```
void MyPhysicsList::addParameterisation()
{
  G4FastSimulationManagerProcess*
    theFastSimulationManagerProcess = new G4FastSimulationManagerProcess();
  theParticleIterator->reset();
  while( (*theParticleIterator)() )
    {
      G4ParticleDefinition* particle = theParticleIterator->value();
      G4ProcessManager* pmanager = particle->GetProcessManager();
      pmanager->AddProcess(theFastSimulationManagerProcess, -1, 0, 0);
    }
}
```

The `G4GlobalFastSimulationManager` Singleton Class

This class is a singleton which can be accessed as follows:

```
#include "G4GlobalFastSimulationManager.hh"
...
...
G4GlobalFastSimulationManager* globalFSM;
globalFSM = G4GlobalFastSimulationManager::getGlobalFastSimulationManager();
...
...
```

Presently, you will mainly need to use the GlobalFastSimulationManager if you use ghost geometries.

Parameterisation Using Ghost Geometries

In some cases, volumes of the tracking geometry do not allow envelopes to be defined. This may be the case with a geometry coming from a CAD system. Since such a geometry is flat, a parallel geometry must be used to define the envelopes.

Another interesting case involves defining an envelope which groups the electromagnetic and hadronic calorimeters of a detector into one volume. This may be useful when parameterizing the interaction of charged pions. You will very likely not want electrons to see this envelope, which means that ghost geometries have to be organized by particle flavours.

Using ghost geometries implies some more overhead in the parameterisation mechanism for the particles sensitive to ghosts, since navigation is provided in the ghost geometry by the G4FastSimulationManagerProcess. Usually, however, only a few volumes will be placed in this ghost world, so that the geometry computations will remain rather cheap.

In the existing implementation (temporary implementation with G4Region but before parallel geometry implementation), you may only consider ghost G4Regions with just one root G4LogicalVolume. The G4GlobalFastSimulationManager provides the construction of the ghost geometry by making first an empty "clone" of the world for tracking provided by the construct() method of your G4VUserDetectorConstruction concrete class. You provide the placement of the G4Region root G4LogicalVolume relative to the ghost world coordinates in the G4FastSimulationManager objects. A ghost G4Region is recognized by the fact that its associated G4FastSimulationManager retains a non-empty list of placements.

The G4GlobalFastSimulationManager will then use both those placements and the IsApplicable() methods of the models attached to the G4FastSimulationManager objects to build the flavour-dependant ghost geometries.

Then at the beginning of the tracking of a particle, the appropriate ghost world, if any, will be selected.

The steps required to build one ghost G4Region are:

1. built the ghost G4Region : myGhostRegion;
2. build the root G4LogicalVolume: myGhostLogical, set it to myGhostRegion;
3. build a G4FastSimulationManager object, myGhostFSManager, giving myGhostRegion as argument of the constructor;
4. give to the G4FastSimulationManager the placement of the myGhostLogical, by invoking for the G4FastSimulationManager method:

   ```
   AddGhostPlacement(G4RotationMatrix*, const G4ThreeVector&);
   ```

   or:

   ```
   AddGhostPlacement(G4Transform3D*);
   ```

   where the rotation matrix and translation vector of the 3-D transformation describe the placement relative to the ghost world coordinates.
5. build your G4VFastSimulationModel objects and add them to the myGhostFSManager. *The IsApplicable() methods of your models will be used by the G4GlobalFastSimulationManager to build the ghost geometries corresponding to a given particle type.*

6. Invoke the G4GlobalFastSimulationManager method:

```
G4GlobalFastSimulationManager::getGlobalFastSimulationManager()->
     CloseFastSimulation();
```

This last call will cause the G4GlobalFastSimulationManager to build the flavour-dependent ghost geometries. This call must be done before the RunManager closes the geometry. (It is foreseen that the run manager in the future will invoke the CloseFastSimulation() to synchronize properly with the closing of the geometry).

Visualization facilities are provided for ghosts geometries. After the CloseFastSimulation() invocation, it is possible to ask for the drawing of ghosts in an interactive session. The basic commands are:

- `/vis/draw/Ghosts particle_name`

  which makes the drawing of the ghost geometry associated with the particle specified by name in the command line.

- `/vis/draw/Ghosts`

  which draws all the ghost geometries.

Gflash Parameterization

This section describes how to use the Gflash library. Gflash is a concrete parameterization which is based on the equations and parameters of the original Gflash package from H1(hep-ex/0001020, Grindhammer & Peters, see physics manual) and uses the "fast simulation" facilities of GEANT4 described above. Briefly, whenever a e-/e+ particle enters the calorimeter, it is parameterized if it has a minimum energy and the shower is expected to be contained in the calorimeter (or " parameterization envelope"). If this is fulfilled the particle is killed, as well as all secondaries, and the energy is deposited according to the Gflash equations. An example, provided in **examples/extended/parametrisation/gflash/**, shows how to interface Gflash to your application. The simulation time is measured, so the user can immediately see the speed increase resulting from the use of Gflash.

Using the Gflash Parameterisation

To use Gflash "out of the box" the following steps are necessary:

- The user must add the fast simulation process to his process manager:

```
void MyPhysicsList::addParameterisation()
{
  G4FastSimulationManagerProcess*
    theFastSimulationManagerProcess = new G4FastSimulationManagerProcess();
  theParticleIterator->reset();
  while( (*theParticleIterator)() )
    {
      G4ParticleDefinition* particle = theParticleIterator->value();
      G4ProcessManager* pmanager = particle->GetProcessManager();
      pmanager->AddProcess(theFastSimulationManagerProcess, -1, 0, 0);
    }
}
```

- The envelope in which the parameterization should be performed must be specified (below: G4Region m_calo_region) and the GFlashShowerModel must be assigned to this region. Furthermore, the classes GFlashParticleBounds (which provides thresholds for the parameterization like minimal energy etc.), GflashHitMaker(a helper class to generate hits in the sensitive detector) and GFlashHomoShowerParamterisation (which does the computations) must be constructed (by the user at the moment) and assigned to the GFlashShowerModel. Please note that at the moment only homogeneous calorimeters are supported.

```
m_theFastShowerModel = new GFlashShowerModel("fastShowerModel",m_calo_region);
m_theParametrisation = new GFlashHomoShowerParamterisation(matManager->getMaterial(mat));
m_theParticleBounds  = new GFlashParticleBounds();
m_theHMaker          = new GFlashHitMaker();
m_theFastShowerModel->SetParametrisation(*m_theParametrisation);
m_theFastShowerModel->SetParticleBounds(*m_theParticleBounds) ;
m_theFastShowerModel->SetHitMaker(*m_theHMaker);
```

The user must also set the material of the calorimeter, since the computation depends on the material.

- It is mandatory to use G4VGFlashSensitiveDetector as (additional) base class for the sensitive detector.

```
class ExGflashSensitiveDetector: public G4VSensitiveDetector ,public␣
↪G4VGFlashSensitiveDetector
```

Here it is necessary to implement a separate interface, where the GFlash spots are processed.

```
(ProcessHits(G4GFlashSpot*aSpot ,G4TouchableHistory* ROhist))
```

A separate interface is used, because the Gflash spots naturally contain less information than the full simulation.

Since the parameters in the Gflash package are taken from fits to full simulations with Geant3, some retuning might be necessary for good agreement with Geant4 showers. For experiment-specific geometries some retuning might be necessary anyway. The tuning is quite complicated since there are many parameters (some correlated) and cannot be described here (see again hep-ex/0001020). For brave users the Gflash framework already forsees the possibility of passing a class with the (users) parameters,**GVFlashHomoShowerTuning**, to the GFlashHomoShowerParamterisation constructor. The default parameters are the original Gflash parameters:

```
GFlashHomoShowerParameterisation(G4Material * aMat, GVFlashHomoShowerTuning * aPar = 0);
```

Now there is also a preliminary implemenation of a parameterization for sampling calorimeters.

The user must specify the active and passive material, as well as the thickness of the active and passive layer.

The sampling structure of the calorimeter is taken into account by using an "effective medium" to compute the shower shape.

All material properties needed are calculated automatically. If tuning is required, the user can pass his own parameter set in the class **GFlashSamplingShowerTuning**. Here the user can also set his calorimeter resolution.

All in all the constructor looks the following:

```
GFlashSamplingShowerParamterisation(G4Material * Mat1, G4Material * Mat2,G4double d1,G4double d2,
GVFlashSamplingShowerTuning * aPar = 0);
```

An implementation of some tools that should help the user to tune the parameterization is forseen.

5.2.7 Transportation Process

To be delivered by J. Apostolakis (John.Apostolakis@cern.ch).

5.3 Particles

5.3.1 Basic concepts

There are three levels of classes to describe particles in Geant4.

`G4ParticleDefinition` defines a particle
`G4DynamicParticle` describes a particle interacting with materials
`G4Track` describes a particle traveling in space and time

`G4ParticleDefinition` aggregates information to characterize a particle's properties, such as name, mass, spin, life time, and decay modes. `G4DynamicParticle` aggregates information to describe the dynamics of particles, such as energy, momentum, polarization, and proper time, as well as "particle definition" information. `G4Track` (see *Tracking*) includes all information necessary for tracking in a detector simulation, such as time, position, and step, as well as "dynamic particle" information.

5.3.2 Definition of a particle

There are a large number of elementary particles and nuclei. Geant4 provides the `G4ParticleDefinition` class to represent particles, and various particles, such as the electron, proton, and gamma have their own classes derived from `G4ParticleDefinition`.

We do not need to make a class in Geant4 for every kind of particle in the world. There are more than 100 types of particles defined in Geant4 by default. Which particles should be included, and how to implement them, is determined according to the following criteria. (Of course, the user can define any particles he wants. Please see the **User's Guide: For ToolKit Developers**).

Particle List in Geant4

This list includes all particles in Geant4 and you can see properties of particles such as

- PDG encoding
- mass and width
- electric charge
- spin, isospin and parity
- magnetic moment
- quark contents
- life time and decay modes

Here is a list of particles in Geant4. This list is generated automatically by using Geant4 functionality, so listed values are same as those in your Geant4 application (as far as you do not change source codes).

Categories

- gluon / quarks / di-quarks
- leptons
- mesons
- baryons
- ions
- others

Classification of particles

1. elementary particles which should be tracked in Geant4 volumes
   All particles that can fly a finite length and interact with materials in detectors are included in this category. In addition, some particles with a very short lifetime are included for user's convenience.
   (a) **stable** particles
   Stable means that the particle can not decay, or has a very small possibility to decay in detectors, e.g., gamma, electron, proton, and neutron.
   (b) **long life** ($>10^{-14}$sec) particles
   Particles which may travel a finite length, e.g., muon, charged pions.
   (c) **short life** particles that decay immediately in Geant4
   For example, pi^0, eta

(d) **K:sup:‘0‘ system**
K^0 “decays” immediately into K^0_S or K^0_L, and then K^0_S/ K^0_L decays according to its life time and decay modes.
(e) **optical photon**
Gamma and optical photon are distinguished in the simulation view, though both are the same particle (photons with different energies). For example, optical photon is used for Cerenkov light and scintillation light.
(f) **geantino/charged geantino**
Geantino and charged geantino are virtual particles for simulation which do not interact with materials and undertake transportation processes only.

2. nuclei
Any kinds of nucleus can be used in Geant4, such as alpha(He-4), uranium-238 and excited states of carbon-14. In addition, Geant4 provides hyper-nuclei. Nuclei in Geant4 are divided into two groups from the viewpoint of implementation.
(a) **light nuclei**
Light nuclei frequently used in simulation, e.g., alpha, deuteron, He3, triton.
(b) **heavy nuclei** (including hyper-nuclei)
Nuclei other than those defined in the previous category.
(c) **light anti-nuclei**
Light anti-nuclei for example anti-alpha.
Note that G4ParticleDefinition represents nucleus state and G4DynamicParticle represents atomic state with some nucleus. Both alpha particle with charge of +2e and helium atom with no charge aggregates the same “particle definition” of G4Alpha, but different G4DynamicParticle objects should be assigned to them. (Details can be found below)
3. short-lived particles
Particles with very short life time decay immediately and are never tracked in the detector geometry. These particles are usually used only inside physics processes to implement some models of interactions. `G4VShortLivedParticle` is provided as the base class for these particles. All classes related to particles in this category can be found in `shortlived` sub-directory under the `particles` directory.
(a) **quarks/di-quarks**: For example, all 6 quarks.
(b) **gluon**
(c) **baryon excited states** with very short life: For example, spin 3/2 baryons and anti-baryons
(d) **meson excited states** with very short life: For example, spin 1 vector bosons

Implementation of particles

Single object created in the initialization: Categories a, b-1

These particles are frequently used for tracking in Geant4. An individual class is defined for each particle in these categories. The object in each class is unique. The user can get pointers to these objects by using static methods in their own classes. The unique object for each class is created when its static method is called in the “initialization phase:.

On-the-fly creation: Category b-2

Ions will travel in a detector geometry and should be tracked, however, the number of ions which may be used for hadronic processes is so huge that ions are dynamically created by requests from processes (and users). Each ion corresponds to one object of the `G4Ions` class. `G4IonTable` class is a dictionary for ions. `G4IonTable::GetIon()` method to create ions on the fly. (`G4IonTable::FIndIon()` method returns pointer to the specified ion. If the ion does not exists, it returns zero without creating any ion.

`G4NucleiPropertiesTableAME03` contains a table of mesaured mass values of about 3100 stable nuclei (ground states). `G4NucleiPropertiesTheoreticalTable` theoretical mass values of about 8000 nuclei (ground states). `G4IsotopeTable` describes properties of ions (exited energy, decay modes, life time and magnetic moments), which are used to create ions. `G4NuclideTable` is provided as a list of nuclei in Geant4. It

contains about 2900 ground states and 4000 excited states. Users can register his/her G4IsotopeTable to the G4IonTable.

Processes attached to heavy ions are same as those for G4GenericIon class. In other words, you need to create G4GenericIon and attach processes to it if you want to use heavy ions.

G4ParticleGun can shoot any heavy ions with /gun/ions command after ion is selected by /gun/particle command.

Dynamic creation by processes: Category c

Particle types in this category are are not created by default, but will only be created by request from processes or directly by users. Each shortlived particle corresponds to one object of a class derived from G4VshortLivedParticle, and it will be created dynamically during the initialization phase.

G4ParticleDefinition

The G4ParticleDefinition class has ‘‘read-only” properties to characterize individual particles, such as name, mass, charge, spin, and so on. These properties are set during initialization of each particle. Methods to get these properties are listed in the *table*.

G4String GetParticleName()	particle name
G4double GetPDGMass()	mass
G4double GetPDGWidth()	decay width
G4double GetPDGCharge()	electric charge
G4double GetPDGSpin()	spin
G4double GetPDGMagneticMoment()	magnetic moment (0: not defined or no magnetic moment)
G4int GetPDGiParity()	parity (0:not defined)
G4int GetPDGiConjugation()	charge conjugation (0:not defined)
G4double GetPDGIsospin()	iso-spin
G4double GetPDGIsospin3()	3rd-component of iso-spin
G4int GetPDGiGParity()	G-parity (0:not defined)
G4String GetParticleType()	particle type
G4String GetParticleSubType()	particle sub-type
G4int GetLeptonNumber()	lepton number
G4int GetBaryonNumber()	baryon number
G4int GetPDGEncoding()	particle encoding number by PDG
G4int GetAntiPDGEncoding()	encoding for anti-particle of this particle

Table: Methods to get particle properties.

The next *table* shows the methods of G4ParticleDefinition for getting information about decay modes and the life time of the particle.

G4bool GetPDGStable()	stable flag
G4double GetPDGLifeTime()	life time
G4DecayTable* GetDecayTable()	decay table

Table: Methods to get particle decay modes and life time.

Users can modify these properties, though the other properties listed above can not be change without rebuilding the libraries.

Each particle has its own G4ProcessManger object that manages a list of processes applicable to the particle.(see *Managing Processes*)

5.3.3 Dynamic particle

The G4DynamicParticle class has kinematics information for the particle and is used for describing the dynamics of physics processes. The properties in G4DynamicParticle are listed in the *table*.

G4double theDynamicalMass	dynamical mass
G4ThreeVector theMomentumDirection	normalized momentum vector
G4ParticleDefinition* theParticleDefinition	definition of particle
G4double theDynamicalSpin	dynamical spin (i.e. total angular momentum as a ion/atom)
G4ThreeVector thePolarization	polarization vector
G4double theMagneticMoment	dynamical magnetic moment (i.e. total magnetic moment as a ion/atom)
G4double theKineticEnergy	kinetic energy
G4double theProperTime	proper time
G4double theDynamicalCharge	dynamical electric charge (i.e. total electric charge as a ion/atom)
G4ElectronOccupancy* theElectronOccupancy	electron orbits for ions

Table: Methods to set/get values.

Here, the dynamical mass is defined as the mass for the dynamic particle. For most cases, it is same as the mass defined in G4ParticleDefinition class (i.e. mass value given by GetPDGMass() method). However, there are two exceptions.

- resonance particle
- ions

Resonance particles have large mass width and the total energy of decay products at the center of mass system can be different event by event.

As for ions, G4ParticleDefintion defines a nucleus and G4DynamicParticle defines an atom. G4ElectronOccupancy describes state of orbital electrons. So, the dynamic mass can be different from the PDG mass by the mass of electrons (and their binding energy). In addition, the dynamical charge, spin and magnetic moment are those of the atom/ion (i.e. including nucleus and orbit electrons).

Decay products of heavy flavor particles are given in many event generators. In such cases, G4VPrimaryGenerator sets this information in *thePreAssignedDecayProducts. In addition, decay time of the particle can be set arbitrarily time by using PreAssignedDecayProperTime.

5.4 Production Threshold versus Tracking Cut

5.4.1 General considerations

We have to fulfill two contradictory requirements. It is the responsibility of each individual **process** to produce secondary particles according to its own capabilities. On the other hand, it is only the Geant4 kernel (i.e., tracking) which can ensure an overall coherence of the simulation.

The general principles in Geant4 are the following:

1. Each **process** has its intrinsic limit(s) to produce secondary particles.
2. All particles produced (and accepted) will be tracked up to **zero range**.

3. Each **particle** has a suggested cut in range (which is converted to energy for all materials), and defined via a `SetCut()` method (see *Range Cuts*).

Points 1 and 2 imply that the cut associated with the **particle** is a (recommended) **production** threshold of secondary particles.

5.4.2 Set production threshold (`SetCut` methods)

As already mentioned, each kind of particle has a suggested production threshold. Some of the processes will not use this threshold (e.g., decay), while other processes will use it as a default value for their intrinsic limits (e.g., ionisation and bremsstrahlung).

See *Range Cuts* to see how to set the production threshold.

5.4.3 Apply cut

The `DoIt` methods of each process can produce secondary particles. Two cases can happen:

- a process sets its intrinsic limit greater than or equal to the recommended production threshold. OK. Nothing has to be done (nothing can be done !).
- a process sets its intrinsic limit smaller than the production threshold (for instance 0).

The list of secondaries is sent to the *SteppingManager* via a *ParticleChange* object.

Before being recopied to the temporary stack for later tracking, the particles below the production threshold will be kept or deleted according to the safe mechanism explained hereafter.

- The *ParticleDefinition* (or *ParticleWithCuts*) has a boolean data member: `ApplyCut`.
- `ApplyCut` is OFF: do nothing. All the secondaries are stacked (and then tracked later on), regardless of their initial energy. The Geant4 kernel respects the best that the physics can do, but neglects the overall coherence and the efficiency. Energy conservation is respected as far as the processes know how to handle correctly the particles they produced! This is the main used during Geant4 tracking.
- `ApplyCut` in ON: this feature is not normally used but is potentially available; the *TrackingManager* checks the range of each secondary against the production threshold and against the safety. The particle is stacked if `range > min(cut,safety)`.
  - If not, check if the process has nevertheless set the flag "good for tracking" and then stack it (see *Why produce secondaries below threshold in some processes?* below for the explanation of the `GoodForTracking` flag).
  - If not, recuperate its kinetic energy in the `localEnergyDeposit`, and set `tkin=0`.
  - Then check in the *ProcessManager* if the vector of *ProcessAtRest* is not empty. If yes, stack the particle for performing the "Action At Rest" later. If not, and only in this case, abandon this secondary.

With this sophisticated mechanism we have the global cut that we wanted, but with energy conservation, and we respect boundary constraint (safety) and the wishes of the processes (via "good for tracking"). Note, that for electromagnetic processes for gamma incident a specific `ApplyCut` option is used which gurantees energy balance and is more efficient because secondary tracks are not produced at all.

5.4.4 Why produce secondaries below threshold in some processes?

A process may have good reasons to produce particles below the recommended threshold:

- checking the range of the secondary versus geometrical quantities like safety may allow one to realize the possibility that the produced particle, even below threshold, will reach a sensitive part of the detector;
- another example is the gamma conversion: the positron is always produced, even at zero energy, for further annihilation;

- if a process is rare there is not practical reason make it complicate checking cut value.

These secondary particles are sent to the "Stepping Manager" with a flag `GoodForTracking` to pass the filter explained in the previous section (even when `ApplyCut` is ON).

5.4.5 Cuts in stopping range or in energy?

The cuts in stopping range allow one to say that the energy has been released at the correct space position, limiting the approximation within a given distance. On the contrary, cuts in energy imply accuracies of the energy depositions which depend on the material.

5.4.6 Summary

In summary, we do not have tracking cuts; we only have production thresholds in range. All particles produced and accepted are tracked up to zero range.

It must be clear that the overall coherency that we provide cannot go beyond the capability of processes to produce particles down to the recommended threshold.

In other words a process can produce the secondaries down to the recommended threshold, and by interrogating the geometry, or by realizing when mass-to-energy conversion can occur, recognize when particles below the threshold have to be produced.

5.4.7 Special tracking cuts

One may need to cut given particle types in given volumes for optimisation reasons. This decision is under user control, and can happen for particles during tracking as well.

The user must be able to apply these special cuts only for the desired particles and in the desired volumes, without introducing an overhead for all the rest.

The approach is as follows:

- special user cuts are registered in the *UserLimits* class (or its descendant), which is associated with the logical volume class.
  The current default list is:
  - max allowed step size
  - max total track length
  - max total time of flight
  - min kinetic energy
  - min remaining range

  The user can instantiate a *UserLimits* object only for the desired logical volumes and do the association.
  The first item (max step size) is automatically taken into account by the G4 kernel while the others items must be managed by the user, as explained below.
  Example(see basic/B2/B2a or B2b): in the Tracker region, in order to force the step size not to exceed one half of the Tracker chamber thickness (`chamberWidth`), it is enough to put the following code in `B2aDetectorConstruction::DefineVolumes()`:

```
G4double maxStep = 0.5*chamberWidth;
fStepLimit = new G4UserLimits(maxStep);
trackerLV->SetUserLimits(fStepLimit);
```

  and in `PhysicsList`, the process `G4StepLimiter` needs to be attached to each particle's process manager where step limitation in the Tracker region is required:

```
// Step limitation seen as a process
G4StepLimiter* stepLimiter = new G4StepLimiter();
pmanager->AddDiscreteProcess(StepLimiter);
```

If a provided Geant4 physics list is used, as FTFP_BERT in B2 example, then the `G4StepLimiterPhysics`, which will take care of attaching the `G4StepLimiter` process to all particles, can be added to the physics list in the `main()` function:

```
G4VModularPhysicsList* physicsList = new FTFP_BERT;
physicsList->RegisterPhysics(new G4StepLimiterPhysics());
runManager->SetUserInitialization(physicsList);
```

The `G4UserLimits` class is in `source/global/management`.

- Concerning the others cuts, the user must define dedicaced process(es). He registers this process (or its descendant) only for the desired particles in their process manager. He can apply his cuts in the `DoIt` of this process, since, via `G4Track`, he can access the logical volume and *UserLimits*.
  An example of such process (called *UserSpecialCuts*) is provided in the repository, but not inserted in any process manager of any particle.
  Example: neutrons. One may need to abandon the tracking of neutrons after a given time of flight (or a charged particle in a magnetic field after a given total track length ... etc ...).
  Example(see basic/B2/B2a or B2b): in the Tracker region, in order to force the total time of flight of the neutrons not to exceed 10 milliseconds, put the following code in `B2aDetectorConstruction::DefineVolumes()`:

```
G4double maxTime = 10*ms;
fStepLimit = new G4UserLimits(DBL_MAX,DBL_MAX,maxTime);
trackerLV->SetUserLimits(fStepLimit);
```

and put the following code in a physics list:

```
G4ProcessManager* pmanager = G4Neutron::Neutron->GetProcessManager();
pmanager->AddProcess(new G4UserSpecialCuts(),-1,-1,1);
```

If a provided Geant4 physics list is used, then a `SpecialCutsBuilder` class can be defined in a similar way as `G4StepLimiterPhysics` and added to the physics list in the `main()` function:

```
G4VModularPhysicsList* physicsList = new FTFP_BERT;
physicsList->RegisterPhysics(new SpecialCutsBuilder());
runManager->SetUserInitialization(physicsList);
```

(The default `G4UserSpecialCuts` class is in `source/processes/transportation`.)

5.5 Cuts per Region

5.5.1 General Concepts

Beginning with Geant4 version 5.1, the concept of a region has been defined for use in geometrical descriptions. Details about regions and how to use them are available in *Sub-detector Regions*. As an example, suppose a user defines three regions, corresponding to the tracking volume, the calorimeter and the bulk structure of a detector. For performance reasons, the user may not be interested in the detailed development of electromagnetic showers in the insensitive bulk structure, but wishes to maintain the best possible accuracy in the tracking region. In such a use case, Geant4 allows the user to set different production thresholds ("cuts") for each geometrical region. This ability, referred to as "cuts per region", is also a new feature provided by the Geant4 5.1 release. The general concepts of production thresholds were presented in the *Production Threshold versus Tracking Cut*.

Please note that this new feature is intended only for users who

1. are simulating the most complex geometries, such as an LHC detector, and
2. are experienced in simulating electromagnetic showers in matter.

We strongly recommend that results generated with this new feature be compared with results using the same geometry and uniform production thresholds. Setting completely different cut values for individual regions may break the coherent and comprehensive accuracy of the simulation. Therefore cut values should be carefully optimized, based on a comparison with results obtained using uniform cuts.

5.5.2 Default Region

The world volume is treated as a region by default. A `G4Region` object is automatically assigned to the world volume and is referred to as the "default region". The production cuts for this region are the defaults which are defined in the *UserPhysicsList*. Unless the user defines different cut values for other regions, the cuts in the default region will be used for the entire geometry.

Please note that the default region and its default production cuts are created and set automatically by `G4RunManager`. The user is **not** allowed to set a region to the world volume, **nor** to assign other production cuts to the default region.

5.5.3 Assigning Production Cuts to a Region

In the `SetCuts()` method of the user's physics list, the user must first define the default cuts. Then a `G4ProductionCuts` object must be created and initialized with the cut value desired for a given region. This object must in turn be assigned to the region object, which can be accessed by name from the `G4RegionStore`. An example `SetCuts()` code follows.

Listing 5.10: Setting production cuts to a region

```
void MyPhysicsList::SetCuts()
{
  // default production thresholds for the world volume
  SetCutsWithDefault();

  // Production thresholds for detector regions
  G4Region* region;
  G4String regName;
  G4ProductionCuts* cuts;

  regName = "tracker";
  region = G4RegionStore::GetInstance()->GetRegion(regName);
  cuts = new G4ProductionCuts;
  cuts->SetProductionCut(0.01*mm); // same cuts for gamma, e- and e+
  region->SetProductionCuts(cuts);

  regName = "calorimeter";
  region = G4RegionStore::GetInstance()->GetRegion(regName);
  cuts = new G4ProductionCuts;
  cuts->SetProductionCut(0.01*mm,G4ProductionCuts::GetIndex("gamma"));
  cuts->SetProductionCut(0.1*mm,G4ProductionCuts::GetIndex("e-"));
  cuts->SetProductionCut(0.1*mm,G4ProductionCuts::GetIndex("e+"));
  region->SetProductionCuts(cuts);
}
```

5.6 Physics Table

5.6.1 General Concepts

In Geant4, physics processes use many tables of cross sections, energy losses and other physics values. Before the execution of an event loop, `PreparePhysicsTable()` and `BuildPhysicsTable()` methods of `G4VProcess`

are invoked for all processes and as a part of initialisation procedure cross section tables are prepared. Energy loss processes calculate cross section and/or energy loss values for each pair of material and production cut value used in geometry for a give run. A change in production cut values therefore require these cross sections to be re-calculated. Cross sections for hadronic processes and gamma processes do not depend on the production cut but sampling of final state may depend on cuts, so full re-initilisation is performed.

The `G4PhysicsTable` class is used to handle cross section tables. `G4PhysicsTable` is a collection of instances of `G4PhysicsVector` (and derived classes), each of which has cross section values for a particle within a given energy range traveling in a material. By default the linear interpolation is used, alternatively spline may be used if the flag of spline is activated by *SetSpline* method of the `G4PhysicsVector`

5.6.2 Material-Cuts Couple

Users can assign different production cuts to different regions (see *Cuts per Region*). This means that if the same material is used in regions with different cut values, the processes need to prepare several different cross sections for that material.

The `G4ProductionCutsTable` has `G4MaterialCutsCouple` objects, each of which consists of a material paired with a cut value. These `G4MaterialCutsCouple`s are numbered with an index which is the same as the index of a `G4PhysicsVector` for the corresponding `G4MaterialCutsCouple`in the `G4PhysicsTable`. The list of *MaterialCutsCouple*s used in the current geometry setup is updated before starting the event loop in each run.

5.6.3 File I/O for the Physics Table

Calculated physics tables for electromagnetic processes can be stored in files. The user may thus eliminate the time required for the calculation of physics tables by retrieving them from the files.

Using the built-in user command "**storePhysicsTable**" (see *Built-in Commands*), stores physics tables in files. Information on materials and cuts defined in the current geometry setup are stored together with physics tables because calculated values in the physics tables depend on *MaterialCutsCouple*. Note that physics tables are calculated before the event loop, not in the initialization phase. So, at least one event must be executed before using the "**storePhysicsTable**" command.

Calculated physics tables can be retrieved from files by using the "**retrievePhysicsTable**" command. Materials and cuts from files are compared with those defined in the current geometry setup, and only physics vectors corresponding to the *MaterialCutsCouple*s used in the current setup are restored. Note that nothing happens just after the "**retrievePhysicsTable**" command is issued. Restoration of physics tables will be executed in parallel with the calculation of physics tables.

5.6.4 Building the Physics Table

In the `G4RunManagerKernel::RunInitialization()` method, after the list of *MaterialCutsCouple*s is updated, the `G4VUserPhysicsList::BuildPhysicsTable()` method is invoked to build physics tables for all processes.

Initially, the `G4VProcess::PreparePhysicsTable()` method is invoked. Each process creates `G4PhysicsTable` objects as necessary. It then checks whether the *MaterialCutsCouple*s have been modified after a run to determine if the corresponding physics vectors can be used in the next run or need to be re-calculated.

Next, the `G4VProcess::RetrievePhysicsTable()` method is invoked if the `G4VUserPhysicsList::fRetrievePhysicsTable` flag is asserted. After checking materials and cuts in files, physics vectors corresponding to the *MaterialCutsCouple*s used in the current setup are restored.

Finally, the `G4VProcess::BuildPhysicsTable()` method is invoked and only physics vectors which need to be re-calculated are built.

At the end of program G4PhysicsTable should be deleted. Before deletion of a table it should be cleaned up using the method G4PhysicsTable::clearAndDestroy(). This method should be called in a middle of the run if an old table is removed and a new one is created.

5.7 User Limits

5.7.1 General Concepts

The user can define artificial limits affecting to the Geant4 tracking.

```
G4UserLimits(G4double uStepMax = DBL_MAX,
             G4double uTrakMax = DBL_MAX,
             G4double uTimeMax = DBL_MAX,
             G4double uEkinMin = 0.,
             G4double uRangMin = 0. );
```

uStepMax	Maximum step length
uTrakMax	Maximum total track length
uTimeMax	Maximum global time for a track
uEkinMin	Minimum remaining kinetic energy for a track
uRangMin	Minimum remaining range for a track

Note that uStepMax is affecting to each step, while all other limits are affecting to a track.

The user can assign G4UserLimits to logical volume and/or to a region. User limits assigned to logical volume do not propagate to daughter volumes, while User limits assigned to region propagate to daughter volumes unless daughters belong to another region. If both logical volume and associated region have user limits, those of logical volume win.

A G4UserLimits object must be instantiated for the duration of whatever logical volume or region to which it is assigned. It is the responsibility of the user's code to delete the object *after* the assigned volume(s)/region(s) have been deleted.

5.7.2 Processes co-working with G4UserLimits

In addition to instantiating G4UserLimits and setting it to logical volume or region, the user has to assign the following process(es) to particle types he/she wants to affect. If none of these processes is assigned, that kind of particle is not affected by G4UserLimits.

Limitation to step (uStepMax) G4StepLimiter process must be defined to affected particle types. This process limits a step, but it does not kill a track.

Limitations to track (uTrakMax, uTimeMax, uEkinMin, uRangMin) G4UserSpecialCuts process must be defined to affected particle types. This process limits a step and kills the track when the track comes to one of these limits. Step limitation occurs only for the final step.

Example of G4UserLimits can be found in examples/basic/B2 : see B2aDetectorConstruction (or B2bDetectorConstruction). The G4StepLimiter process is added in the Geant4 physics list via the G4StepLimiterPhysics class in the main() function in exampleB4a.cc (or exampleB4b.cc).

5.8 Track Error Propagation

The error propagation package serves to propagate one particle together with its error from a given trajectory state until a user-defined target is reached (a surface, a volume, a given track length,...).

5.8.1 Physics

The error propagator package computes the average trajectory that a particle would follow. This means that the physics list must have the following characteristics:

- No multiple scattering
- No random fluctuations for energy loss
- No creation of secondary tracks
- No hadronic processes

It has also to be taken into account that when the propagation is done backwards (in the direction opposed to the one the original track traveled) the energy loss has to be changed into an energy gain.

All this is done in the `G4ErrorPhysicsList` class, that is automatically set by `G4ErrorPropagatorManager` as the GEANT4 physics list. It sets `G4ErrorEnergyLoss` as unique electromagnetic process. This process uses the GEANT4 class `G4EnergyLossForExtrapolator` to compute the average energy loss for forwards or backwards propagation. To avoid getting too different energy loss calculation when the propagation is done forwards (when the energy at the beginning of the step is used) or backwards (when the energy at the end of the step is used, always smaller than at the beginning) `G4ErrorEnergyLoss` computes once the energy loss and then replaces the original energy loss by subtracting/adding half of this value (what is approximately the same as computing the energy loss with the energy at the middle of the step). In this way, a better calculation of the energy loss is obtained with a minimal impact on the total CPU time.

The user may use his/her own physics list instead of `G4ErrorPhysicsList`. As it is not needed to define a physics list when running this package, the user may have not realized that somewhere else in his/her application it has been defined; therefore a warning will be sent to advert the user that he is using a physics list different to `G4ErrorPhysicsList`. If a new physics list is used, it should also initialize the `G4ErrorMessenger` with the classes that serve to limit the step:

```
G4ErrorEnergyLoss* eLossProcess = new G4ErrorEnergyLoss;
G4ErrorStepLengthLimitProcess* stepLengthLimitProcess = new G4ErrorStepLengthLimitProcess;
G4ErrorMagFieldLimitProcess* magFieldLimitProcess = new G4ErrorMagFieldLimitProcess;
new G4ErrorMessenger( stepLengthLimitProcess, magFieldLimitProcess, eLossProcess );
```

To ease the use of this package in the reconstruction code, the physics list, whether `G4ErrorPhysicsList` or the user's one, will be automatically initialized before starting the track propagation if it has not been done by the user.

5.8.2 Trajectory state

The user has to provide the particle trajectory state at the initial point. To do this it has to create an object of one of the children classes of `G4ErrorTrajState`, providing:

- Particle type
- Position
- Momentum
- Trajectory error matrix

```
G4ErrorTrajState( const G4String& partType,
                  const G4Point3D& pos,
                  const G4Vector3D& mom,
                  const G4ErrorTrajErr& errmat = G4ErrorTrajErr(5,0) );
```

A particle trajectory is characterized by five independent variables as a function of one parameter (e.g. the path length). Among the five variables, one is related to the curvature (to the absolute value of the momentum), two are related to the direction of the particle and the other two are related to the spatial location.

There are two possible representations of these five parameters in the error propagator package: as a free trajectory state, class `G4ErrorTrajStateFree`, or as a trajectory state on a surface, class `G4ErrorTrajStateonSurface`.

Free trajectory state

In the free trajectory state representation the five trajectory parameters are

- G4double fInvP
- G4double fLambda
- G4double fPhi
- G4double fYPerp
- G4double fZPerp

where `fInvP` is the inverse of the momentum. `fLambda` and `fPhi` are the dip and azimuthal angles related to the momentum components in the following way:

`p\_x = p cos(lambda) cos(phi) p\_y = p cos(lambda) sin(phi) p\_z = p sin(lambda)`, that is, `lambda = 90 - theta`, where `theta` is the usual angle with respect to the Z axis.

`fYperp` and `fZperp` are the coordinates of the trajectory in a local orthonormal reference frame with the X axis along the particle direction, the Y axis being parallel to the X-Y plane (obtained by the vectorial product of the global Z axis and the momentum).

Trajectory state on a surface

In the trajectory state on a surface representation the five trajectory parameters are

- G4double fInvP
- G4double fPV
- G4double fPW
- G4double fV
- G4double fW

where `fInvP` is the inverse of the momentum; `fPV` and `fPW` are the momentum components in an orthonormal coordinate system with axis U, V and W; `fV` and `fW` are the position components on this coordinate system.

For this representation the user has to provide the plane where the parameters are calculated. This can be done by providing two vectors, V and W, contained in the plane:

```
G4ErrorSurfaceTrajState( const G4String& partType,
                         const G4Point3D& pos,
                         const G4Vector3D& mom,
                         const G4Vector3D& vecV,
                         const G4Vector3D& vecW,
                         const G4ErrorTrajErr& errmat = G4ErrorTrajErr(5,0) );
```

or by providing a plane

```
G4ErrorSurfaceTrajState( const G4String& partType,
                         const G4Point3D& pos,
                         const G4Vector3D& mom,
                         const G4Plane3D& plane,
                         const G4ErrorTrajErr& errmat = G4ErrorTrajErr(5,0) );
```

In this second case the vector V is calculated as the vector in the plane perpendicular to the global vector X (if the plane normal is equal to X, Z is used instead) and W is calculated as the vector in the plane perpendicular to V.

5.8.3 Trajectory state error

The 5X5 error matrix should also be provided at the creation of the trajectory state as a `G4ErrorTrajErr` object. If it is not provided a default object will be created filled with null values.

Currently the `G4ErrorTrajErr` is a `G4ErrorSymMatrix`, a simplified version of `CLHEP HepSymMatrix`.

The error matrix is given in units of GeV and cm. Therefore you should do the conversion if your code is using other units.

5.8.4 Targets

The user has to define up to where the propagation must be done: the target. The target can be a surface `G4ErrorSurfaceTarget`, which is not part of the GEANT4 geometry. It can also be the surface of a GEANT4 volume `G4ErrorGeomVolumeTarget`, so that the particle will be stopped when it enters this volume. Or it can be that the particle is stopped when a certain track length is reached, by implementing a `G4ErrorTrackLengthTarget`.

Surface target

When the user chooses a `G4ErrorSurfaceTarget` as target, the track is propagated until the surface is reached. This surface is not part of GEANT4 geometry, but usually traverses many GEANT4 volumes. The class `G4ErrorNavigator` takes care of the double navigation: for each step the step length is calculated as the minimum of the step length in the full geometry (up to a GEANT4 volume surface) and the distance to the user-defined surface. To do it, `G4ErrorNavigator` inherits from `G4Navigator` and overwrites the methods `ComputeStep()` and `ComputeSafety()`. Two types of surface are currently supported (more types could be easily implemented at user request): plane and cylindrical.

Plane surface target

`G4ErrorPlaneSurfaceTarget` implements an infinite plane surface. The surface can be given as the four coefficients of the plane equation `ax+by+cz+d = 0`:

```
G4ErrorPlaneSurfaceTarget(G4double a=0,
                          G4double b=0,
                          G4double c=0,
                          G4double d=0);
```

or as the normal to the plane and a point contained in it:

```
G4ErrorPlaneSurfaceTarget(const G4Normal3D &n,
                          const G4Point3D &p);
```

or as three points contained in it:

```
G4ErrorPlaneSurfaceTarget(const G4Point3D &p1,
                          const G4Point3D &p2,
                          const G4Point3D &p3);
```

Cylindrical surface target

G4ErrorCylSurfaceTarget implements an infinite-length cylindrical surface (a cylinder without end-caps). The surface can be given as the radius, the translation and the rotation

```
G4ErrorCylSurfaceTarget( const G4double& radius,
                         const G4ThreeVector& trans=G4ThreeVector(),
                         const G4RotationMatrix& rotm=G4RotationMatrix() );
```

or as the radius and the affine transformation

```
G4ErrorCylSurfaceTarget( const G4double& radius,
                         const G4AffineTransform& trans );
```

Geometry volume target

When the user chooses a `G4ErrorGeomVolumeTarget` as target, the track is propagated until the surface of a GEANT4 volume is reached. User can choose if the track will be stopped only when the track enters the volume, only when the track exits the volume or in both cases.

The object has to be instantiated giving the name of a logical volume existing in the geometry:

```
G4ErrorGeomVolumeTarget( const G4String& name );
```

Track Length target

When the user chooses a `G4ErrorTrackLengthTarget` as target, the track is propagated until the given track length is reached.

The object has to be instantiated giving the value of the track length:

```
G4ErrorTrackLengthTarget(const G4double maxTrkLength );
```

It is implemented as a `G4VDiscreteProcess` and it limits the step in `PostStepGetPhysicalInteractionLength`. To ease its use, the process is registered to all particles in the constructor.

5.8.5 Managing the track propagation

The user needs to propagate just one track, so there is no need of run and events. neither of `G4VPrimaryGeneratorAction`. `G4ErrorPropagator` creates a track from the information given in the `G4ErrorTrajState` and manages the step propagation. The propagation is done by the standard GEANT4 methods, invoking `G4SteppingManager::Stepping()` to propagate each step.

After one step is propagated, `G4ErrorPropagator` takes cares of propagating the track errors for this step, what is done by `G4ErrorTrajStateFree::PropagateError()`. The equations of error propagation are only implemented in the representation of `G4ErrorTrajStateFree`. Therefore if the user has provided instead a `G4ErrorTrajStateOnSurface` object, it will be transformed into a `G4ErrorTrajStateFree` at the beginning of tracking, and at the end it is converted back into `G4ErrorTrajStateOnSurface` on the target surface (on the normal plane to the surface at the final point).

The user `G4VUserTrackingAction::PreUserTrackingAction( const G4Track* )` and `G4VUserTrackingAction::PreUserTrackingAction( const G4Track* )` are also invoked at the beginning and at the end of the track propagation.

G4ErrorPropagator stops the tracking when one of the three conditions is true:

- Energy is exhausted
- World boundary is reached
- User-defined target is reached

In case the defined target is not reached, G4ErrorPropagator::Propagate() returns a negative value.

The propagation of a trajectory state until a user defined target can be done by invoking the method of G4ErrorPropagatorManager

```
G4int Propagate( G4ErrorTrajState* currentTS, const G4ErrorTarget* target,
                 G4ErrorMode mode = G4ErrorMode_PropForwards );
```

You can get the pointer to the only instance of G4ErrorPropagatorManager with

```
G4ErrorPropagatorManager* g4emgr = G4ErrorPropagatorManager::GetErrorPropagatorManager();
```

Another possibility is to invoke the propagation step by step, returning control to the user after each step. This can be done with the method

```
G4int PropagateOneStep( G4ErrorTrajState* currentTS,
                        G4ErrorMode mode = G4ErrorMode_PropForwards );
```

In this case you should register the target first with the command

```
G4ErrorPropagatorData::GetG4ErrorPropagatorData()->SetTarget( theG4eTarget );
```

Error propagation

As in the GEANT3-based GEANE package, the error propagation is based on the equations of the European Muon Collaboration, that take into account:

- Error from curved trajectory in magnetic field
- Error from multiple scattering
- Error from ionization

The formulas assume propagation along an helix. This means that it is necessary to make steps small enough to assure magnetic field constantness and not too big energy loss.

5.8.6 Limiting the step

There are three ways to limit the step. The first one is by using a fixed length value. This can be set by invoking the user command:

```
G4UImanager::GetUIpointer()->ApplyCommand("/geant4e/limits/stepLength MY_VALUE MY_UNIT");
```

The second one is by setting the maximum percentage of energy loss in the step (or energy gain is propagation is backwards). This can be set by invoking the user command:

```
G4UImanager::GetUIpointer()->ApplyCommand("/geant4e/limits/energyLoss MY_VALUE");
```

The last one is by setting the maximum difference between the value of the magnetic field at the beginning and at the end of the step. Indeed what is limited is the curvature, or exactly the value of the magnetic field divided by the value of the momentum transversal to the field. This can be set by invoking the user command:

```
G4UImanager::GetUIpointer()->ApplyCommand("/geant4e/limits/magField MY_VALUE");
```

The classes that limit the step are implemented as GEANT4 processes. Therefore, the invocation of the above-mentioned commands should only be done after the initialization (for example after `G4ErrorPropagatorManager::InitGeant4e()`.

5.9 Exotic Physics

The Geant4 toolkit has recently been extended to include "exotic physics". This covers the area of phonon propagation and crystal channelling. These two domains are applicable for Dark Matter experiments (phonon excitation) and beam extraction and collimation (crystal channelling). The framework within Geant4 is similar in that a macroscopic periodic crystal lattice is required for both and wave functions are propagated within the medium (rather than discrete particles as in the case of conventional Geant4). Contained here is a brief description of how to modify a Geant4 application to include the crystal as both a material and a geometry (plane orientations).

5.9.1 Physics

For a more complete description and understanding the user is referred to the extended examples category "exotic-physics" and the references therein.

5.9.2 Material

The implementation of solid-state processes in Geant4 requires the addition of two important features, the crystal unit cell with all its parameters and the support for other data required by the processes. The extended data for a material is stored in a class derived from the virtual class `G4VMaterialExtension`. The `G4ExtenededMaterial` class collects the pointers to concrete instances of `G4VMaterialExtension`. The `G4CrystalExtension` class is a derived class of `G4VMaterialExtension` and collects information on the physics properties of a perfect crystal. In particular, the class contains a pointer to a `G4CrystalUnitCell` object, the elasticity tensor, a map of `G4CrystalAtomBase` objects associated with a `G4Element` and a vector of `G4AtomicBond`. The `G4CrystalUnitCell` class collects information on the mathematical description of the crystal unit cell, i.e. the sizes and the angles of the unit cell, the space group, the Bravais lattice and the lattice system, and methods for the calculation of the volume in the direct and reciprocal space, the spacing between two planes, the angle between two planes, and for the filling of the reduced elasticity tensor. The `G4CrystalExtension` constructor takes as argument a pointer to a `G4Material` object and has to be registered to the `G4ExtendedMaterial` to which it is attached. The `G4CrystalAtomBase` class stores the position of atoms in the crystal unit cell. Since the `G4CrystalAtomBase` class is mapped to a `G4Element` in the `G4CrystalMaterial`, each `G4Element` should have an associated `G4CrystalAtomBase`. The `G4AtomicBond` class contains information on the atomic bond in the crystal. For each instance of the class two `G4Elements` have to be specified as well as the atom number in the `G4CrystalAtomBase` associated to the `G4Element`.

Code Implementation

5.9.3 Geometry

The `G4LogicalCrystalVolume` accepts only a pointer to a `G4CrystalExtension` in its constructor and stores the definition of the orientation of the crystalline structure with respect to the solid to which it is attached. By convention, the crystal < 100 > direction is by default set parallel to the {[1,0,0]} direction in the Geant4 reference system, and the < 010 > axis lays on the plane which contains the [1,0,0] and [0,1,0] directions in the Geant4 reference system.

CHAPTER

SIX

USER ACTIONS

6.1 User Actions

Geant4 has two user initialization classes and one user action class whose methods the user must override in order to implement a simulation. They require the user to define the detector, specify the physics to be used, and define how initial particles are to be generated. These classes are described in *Mandatory User Actions and Initializations*.

Additionally, users may define any of several optional user actions, to collect data during event generation from steps, tracks, or whole events, to accumulate data during runs, or to modify the state of new tracks as they are created. These user actions are described in *Optional User Actions*.

To support the accumulation of data in the actions mentioned above, users may define subclasses for some of the container objects used during event generation and tracking. These are described in *User Information Classes*.

6.2 Mandatory User Actions and Initializations

Three user initialization class objects are registered with the run manager (*Manage the run procedures*) in the user's `main()` program, which takes ownership. The user must not delete these objects directly, and they must be created using 'new'. Within the `G4UserActionInitialization` class (*User Action Initialization*), the user must instantiate and register a concrete `G4VUserPrimaryGeneratorAction` subclass, which generates the primary particles for each event.

6.2.1 G4VUserDetectorConstruction

Listing 6.1: `G4VUserDetectorConstruction`

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

  public:
    virtual G4VPhysicalVolume* Construct() = 0;
    virtual void ConstructSDandField() = 0;
};
```

In the `Construct()` method, material and geometry has to be descrived. Detailed discussions on material and geometry are given in *How to Specify Materials in the Detector* and *How to Define a Detector Geometry*. Detector sensitivity and electromagnetic field should be defined in `ConstructSDandField()`, as objects defined in this method are thread-local if they are used in multi-threaded mode. Detailed discussions on Detector sensitivity and electromagnetic field are given in *Hits* and *Electromagnetic Field*.

6.2.2 Physics Lists

The concept of a physics list arises from the fact that Geant4 can not offer a single modeling algorithm to cover the entire energy domain from zero to the TeV scale, for all known processes and particles. Instead, a combination of ideas and approaches is typically used to perform a simulation task.

A schematic view of the Geant4 modeling of the processes of particle passage through matter may be presented as follows:

- Physics Model = final state generator
- Physics Process = cross section + model
- Physics List = list of processes for each particle

The "patchwork" concept is especially true in the Geant4 hadronic physics domain: models are valid only over finite energy ranges, and there maybe competing models in the same range or one model maybe working better than the other for a specific group of particles, while its competitor may be better for other species. For this reason models have to be combined to cover the large energy range; every two adjacent models may have an overlap in their validity range.

G4VUserPhysicsList

This is an abstract class for constructing particles and processes. An introduction into the concept of the Geant4 Physics List and the Geant4 Physics Processes is also given in *How to Specify Physics Processes* and further in *Physics Processes*.

While the fabrication of a physics list is, in principle, a choice of a user, the toolkit is distributed with a number of pre-fabricated physics lists for the convenience of many user applications. These physics lists are supported by the Geant4 development team and can be recommended for specific physics tasks. However, based on the interests and needs of a specific project, a user may want to implement her or his own custom physics list.

The following sections offer several examples that show how to instantiate or select one or another pre-fabricated Physics List from the Geant4 standard collection, as well as guidance composing a custom Physics List from pre-facbricated components or even entirely from scratch.

To view the contents of a Physics List, there are two useful methods: `DumpList()` and `DumpCutValueTable(G4int flag)`.

Reference Physics Lists

Number of ready to use Physics Lists are available with Geant4 kernel. Below an example of instantiation of FTFP_BERT Physics List class is shown. The full set of reference Physics Lists is described in Geant4 web.

Listing 6.2: Creating FTFP_BERT Physics List.

```
G4int verbose = 1;
FTFP_BERT* physlist = new FTFP_BERT(verbose);
runManager->SetUserInitialization(physlist);
```

Building Physics List Using Factory

Geant4 provides a class `G4PhysListFactory` allowing to defined Physics List by its name. The last for characters in the name defines an electromagnetic (EM) physics options. By default standard EM physics is used, "_EMV" corresponding to standard option1, "_EMX" - to standard option2, "_LIV" to EM Livermore physics, "_PEN" - to EM Penelope physics.

Listing 6.3: Creating Physics List by name.

```
G4int verbose = 1;
G4PhysListFactory factory;
G4VModularPhysicsList* physlist = factory.GetReferencePhysList("FTFP_BERT_EMV");
physlist.SetVerboseLevel(verbose);
runManager->SetUserInitialization(physlist);
```

The class `G4PhysListFactory` provides also another interface allowing to defined Physics List by the environment variable *PHYSLIST*.

Listing 6.4: Creating Physics List by name.

```
G4int verbose = 1;
G4PhysListFactory factory;
G4VModularPhysicsList* physlist = factory.ReferencePhysList();
physlist.SetVerboseLevel(verbose);
runManager->SetUserInitialization(physlist);
```

Building Physics List from Physics Builders

Technically speaking, one can implement physics list in a "flat-out" manner, i.e. specify all necessary particles and associated processes in a single piece of code, as it will be shown later in this document. However, for practical purposes it is often more convenient to group together certain categories and make implementation more modular.

One very useful concept is a Modular Physics List, `G4VModularPhysicsList`, that is a sub-class of `G4VUserPhysicsLists` and allows a user to organize physics processes into "building blocks", or "modules", then compose a physics list of such modules. The concept allows to group together, at a relatively high level, desired combinations of selected particles and related processes. One of the advantages of such approach is that it allows to combine pre-fabricated physics modules that are centrally provided by Geant4 kernel with user's applications.

`G4ModularPhysicsList` has all the functionalities as `G4VUserPhysicsList` class, plus several additional functionalities. One of the important methods is `RegisterPhysics(G4VPhysicsConstructor* )` for "registering" the above mentioned pre-fabriced physics modules. There also methods for removing or replacing physics modules.

Example below shows how `G4VModularPhysList` can be implemented.

Listing 6.5: Creating Physics List by name.

```
MyPhysicsList::MyPhysicsList():G4VModularPhysicsList()
{
  G4DataQuestionaire it(photon, neutron, no, no, no, neutronxs);
  G4cout << "<<< Geant4 Physics List: MyPhysicsList " <<G4endl;
  G4cout <<G4endl;
  defaultCutValue = 0.7*mm;
  G4int ver = 1;
  SetVerboseLevel(ver);

  // EM Physics
  RegisterPhysics( new G4EmStandardPhysics(ver) );

  // Synchroton Radiation & GN Physics
  RegisterPhysics( new G4EmExtraPhysics(ver) );
  // Decays
  RegisterPhysics( new G4DecayPhysics(ver) );

  // Hadron physics
  RegisterPhysics( new G4HadronElasticPhysicsXS(ver) );
  RegisterPhysics( new G4QStoppingPhysics(ver) );
  RegisterPhysics( new G4IonBinaryCascadePhysics(ver) );
```

```
  RegisterPhysics( new G4HadronInelasticQBBC(ver));

  // Neutron tracking cut
  RegisterPhysics( new G4NeutronTrackingCut(ver) );
}
```

Note that each module to be registered with a Modular Physics List is a `G4VPhysicsConstructor` (or a derived object), i.e. a "sublist" that holds groups of particles and accompanying physics processes. A user can find these and other similar modules in the source/physics_lists/list area of Geant4 core code, and can combine selected ones with custom modules, if desired.

In order to compose a custom physics module, two mandatory methods of a `G4VPhysicsConstructor` must be implemented: `ConstructParticle()` and `ConstructProcess()`; beyond that the implementation can be structured according to the developer's taste.

Another useful concept in the modular approach to composing a Physics List is the concept of so called "builders". This concept allows to encapsulate certain implementation details into smaller-scale software components, and offers the flexibility of re-using those component in different modules. At the general level, the scheme is this:

- Particles (hadrons) are created, and physics models to be used to simulate applicable processes are specified, usually in a particular range of validity.
- Physics processes for each particle type in the builder are created, and each process is outfitted with one or more hadronic physics models, as specified.
- If necessary, a cross section data set for a given particle type is added.

This concept is widely used through the Geant4 hadronic domain, but the idea would be equally applicable in the electromagnetic area.

All builders can be found in the source/physics_lists/builders directory. There are basically two types of builders:

- Particle Builders
- Particle-Model Builders

A particle builder is somewhat "superior" here, as it specifies a particle or a group of particles, what category of processes are applicable, how to outfit a process with specified model(s), and how processes are to be registered with the `G4ProcessManager`. A particle-model builder instantiates a given model and implements details of associating it with one or more processes applicable to a given particle type. Some models can not be instantiated through a single interface class, but instead they need, in turn, to be composed from several components (examples are QGS and FTF).

Useful example bulders to review and to consider as inspirations can be the following:

- G4PiKBuilder (.hh and .cc) - groups pions and kaons, together with a list of associated hadronic processes.
- G4BertiniPiKBuilder (.hh and .cc) - instrantiates Bertini cascade model and implements how to outfit pion and kaon physics processes with this model. It also sets default validity range for the model.
- G4FTFPPiKBuilder (.hh and .cc) - composes a high energy FTF-based model and implements how to outfit hadronic processes for pions and kaons with the model. This example illustrates that a hadronic model does not always have a single interface class, but it needs to be created from several components. In particular, in this builder a "high energy generator" object (G4TheoFSGenerator) is created and is outfited with G4FTFModel string model (which also gives this builder its name), we well as string fragmentation algorithm and intra-nuclear transport model. Please note that the quasi-elastic scattering is not set as FTF model has its own mechanism for it. A cross-section data set is specified for pions. A default validity range is also specified.

One detail to remember is that, in principle, the validity range for a given model can be setup for each particle type individually. But in these referenced applications the validity range is setup to be the same for a group of particles (i.e. for a number of corresponding inelastic hadronic processes). Once a builder is instantiated, one can override the default validity range (via SetMinEnery or SetMaxEnergy methods), but the new value will be, again, given to a group of particles/processes. Also note that the validity range can be overriden only before calling the Build() method of a builder. Again, the approach is just a specifics of this particular implementation. Obviously, if a limited validity range

is selected for a specific particle/model/process, one has to supplement another model or several models, to cover the entire range.

One more useful class is the ‘‘ G4PhysicsListHelper‘‘ which is a service class that wraps around the technicalities of the physics process registering in Geant4 and allows a user to easily associate a process with a particles, without knowing many details about various types of processes (discrete, continuous, etc.) and their internal ordering with `G4ProcessManager`. Curious users may eventually want to go deeper into details of G4ProcessManager class and, in particular, its group of `AddProcess(...)` methods, as it is the basis of G4PhysicsListHelper implementation. But for practical purposes, the use of G4PhysicsListHelper is likely to be sufficient in most cases.

Other useful details, including several elements of the software design philosophy and class diagrams, are given in *How to Specify Physics Processes*.

Building Physics List from Scratch

The user must derive a concrete class from `G4VUserPhysicsList` and implement three virtual methods:

- `ConstructParticle()` to instantiate each requested particle type;
- `ConstructPhysics()` to instantiate the desired physics processes and register each of them;
- `SetCuts(G4double aValue)` to set a cut value in range for all particles in the particle table, which invokes the rebuilding of the physics table.

At early stage of the initialisation of Geant4 the method `ConstructParticle()` of `G4VUserPhysicsList` is invoked. The `ConstructProcess()` method must always invoke the `AddTransportation()` method in order to insure particle transportation. `AddTransportation()` must never be overridden. This is done automatically if `G4VUserPhysicsList` inherits of `G4VModularPhysicsList`. It is recommended for users as the most robust interface to Physics List. Geant4 examples demonstrate different methods how to create user Physics List.

6.2.3 User Action Initialization

All user action classes must be defined through the protected method `SetUserAction()`. `Build()` methos should be used for defining user action classes for worker threads as well as for the sequential mode. `BuildForMaster()` should be used only for defining UserRunAction for the master thread. `BuildForMaster()` is not invoked in the sequential mode. In case the user uses his/her own `SteppingVerbose` class, it must be instantiated in the method `InitializeSteppingVerbose()` and returned.

G4VUserActionInitialization

Listing 6.6: `G4VUserActionInitialization`

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

  public:
    virtual void Build() const = 0;
    virtual void BuildForMaster() const;
    virtual G4VSteppingVerbose* InitializeSteppingVerbose() const;

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

```
    void SetUserAction(G4UserSteppingAction*) const;
};
```

G4VUserPrimaryGeneratorAction

Listing 6.7: G4VUserPrimaryGeneratorAction

```
class G4VUserPrimaryGeneratorAction
{
  public:
    G4VUserPrimaryGeneratorAction();
    virtual ~G4VUserPrimaryGeneratorAction();

  public:
    virtual void GeneratePrimaries(G4Event* anEvent) = 0;
};
```

6.3 Optional User Actions

There are five virtual classes whose methods the user may override in order to gain control of the simulation at various stages. Each method of each action class has an empty default implementation, allowing the user to inherit and implement desired classes and methods.

Objects of user action classes must be registered with G4RunManager (*Manage the run procedures*), which takes ownership of them. The user must not delete these objects directly, and they must be created using 'new'.

6.3.1 Usage of User Actions

G4UserRunAction

This class has three virtual methods which are invoked by G4RunManager for each run:

- **GenerateRun()** This method is invoked at the beginning of BeamOn. Because the user can inherit the class G4Run and create his/her own concrete class to store some information about the run, the GenerateRun() method is the place to instantiate such an object. It is also the ideal place to set variables which affect the physics table (such as production thresholds) for a particular run, because GenerateRun() is invoked before the calculation of the physics table.
- **BeginOfRunAction()** This method is invoked before entering the event loop. A typical use of this method would be to initialize and/or book histograms for a particular run. This method is invoked after the calculation of the physics tables.
- **EndOfRunAction()** This method is invoked at the very end of the run processing. It is typically used for a simple analysis of the processed run.

Listing 6.8: G4UserRunAction

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

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

```
    virtual void EndOfRunAction(const G4Run*);
};
```

G4UserEventAction

This class has two virtual methods which are invoked by `G4EventManager` for each event:

- **`beginOfEventAction()`** This method is invoked before converting the primary particles to `G4Track` objects. A typical use of this method would be to initialize and/or book histograms for a particular event.
- **`endOfEventAction()`** This method is invoked at the very end of event processing. It is typically used for a simple analysis of the processed event. If the user wants to keep the currently processing event until the end of the current run, the user can invoke `fpEventManager->KeepTheCurrentEvent();` so that it is kept in `G4Run` object. This should be quite useful if you simulate quite many events and want to visualize only the most interest ones after the long execution. Given the memory size of an event and its contents may be large, it is the user's responsibility not to keep unnecessary events.

Listing 6.9: `G4UserEventAction`

```
class G4UserEventAction
{
  public:
      G4UserEventAction() {;}
      virtual ~G4UserEventAction() {;}
      virtual void BeginOfEventAction(const G4Event*);
      virtual void EndOfEventAction(const G4Event*);
  protected:
      G4EventManager* fpEventManager;
};
```

G4UserStackingAction

This class has three virtual methods, `ClassifyNewTrack`, `NewStage` and `PrepareNewEvent` which the user may override in order to control the various track stacking mechanisms. ExampleN04 could be a good example to understand the usage of this class.

`ClassifyNewTrack()` is invoked by `G4StackManager` whenever a new `G4Track` object is "pushed" onto a stack by `G4EventManager`. `ClassifyNewTrack()` returns an enumerator, `G4ClassificationOfNewTrack`, whose value indicates to which stack, if any, the track will be sent. This value should be determined by the user. `G4ClassificationOfNewTrack` has four possible values:

- `fUrgent` - track is placed in the *urgent* stack
- `fWaiting` - track is placed in the *waiting* stack, and will not be simulated until the *urgent* stack is empty
- `fPostpone` - track is postponed to the next event
- `fKill` - the track is deleted immediately and not stored in any stack.

These assignments may be made based on the origin of the track which is obtained as follows:

```
G4int parent_ID = aTrack->get_parentID();
```

where

- `parent_ID = 0` indicates a primary particle
- `parent_ID > 0` indicates a secondary particle
- `parent_ID < 0` indicates postponed particle from previous event.

`NewStage()` is invoked when the *urgent* stack is empty and the *waiting* stack contains at least one `G4Track` object. Here the user may kill or re-assign to different stacks all the tracks in the *waiting* stack by calling the

`stackManager->ReClassify()` method which, in turn, calls the `ClassifyNewTrack()` method. If no user action is taken, all tracks in the *waiting* stack are transferred to the *urgent* stack. The user may also decide to abort the current event even though some tracks may remain in the *waiting* stack by calling `stackManager->clear()`. This method is valid and safe only if it is called from the `G4UserStackingAction` class. A global method of event abortion is

```
G4UImanager * UImanager = G4UImanager::GetUIpointer();
UImanager->ApplyCommand("/event/abort");
```

`PrepareNewEvent()` is invoked at the beginning of each event. At this point no primary particles have been converted to tracks, so the *urgent* and *waiting* stacks are empty. However, there may be tracks in the *postponed-to-next-event* stack; for each of these the `ClassifyNewTrack()` method is called and the track is assigned to the appropriate stack.

Listing 6.10: `G4UserStackingAction`

```
#include "G4ClassificationOfNewTrack.hh"

class G4UserStackingAction
{
  public:
      G4UserStackingAction();
      virtual ~G4UserStackingAction();
  protected:
      G4StackManager * stackManager;

  public:
//------------------------------------------------------------
// virtual methods to be implemented by user
//------------------------------------------------------------
//
      virtual G4ClassificationOfNewTrack
        ClassifyNewTrack(const G4Track*);
//
//------------------------------------------------------------
//
      virtual void NewStage();
//
//------------------------------------------------------------
//
      virtual void PrepareNewEvent();
//
//------------------------------------------------------------

};
```

G4UserTrackingAction

Listing 6.11: `G4UserTrackingAction`

```
//------------------------------------------------------------
//
// G4UserTrackingAction.hh
//
// Description:
// This class represents actions taken place by the user at
// the start/end point of processing one track.
//
//------------------------------------------------------------

////////////////////////
class G4UserTrackingAction
```

```
//////////////////////////
{

//--------
   public:
//--------

// Constructor & Destructor
   G4UserTrackingAction(){};
   virtual ~G4UserTrackingAction(){}

// Member functions
   virtual void PreUserTrackingAction(const G4Track*){}
   virtual void PostUserTrackingAction(const G4Track*){}

//-----------
   protected:
//-----------

// Member data
   G4TrackingManager* fpTrackingManager;

};
```

G4UserSteppingAction

Listing 6.12: G4UserSteppingAction

```
//---------------------------------------------------------------
//
//  G4UserSteppingAction.hh
//
//  Description:
//    This class represents actions taken place by the user at each
//    end of stepping.
//
//---------------------------------------------------------------

//////////////////////////
class G4UserSteppingAction
//////////////////////////
{

//--------
   public:
//--------

// Constructor and destructor
   G4UserSteppingAction(){}
   virtual ~G4UserSteppingAction(){}

// Member functions
   virtual void UserSteppingAction(const G4Step*){}

//-----------
   protected:
//-----------

// Member data
   G4SteppingManager* fpSteppingManager;

};
```

6.3.2 Killing Tracks in User Actions and Energy Conservation

In either of user action classes described in the previous section, the user can implement an unnatural/unphysical action. A typical example is to kill a track, which is under the simulation, in the user stepping action. In this case the user have to be cautious of the total energy conservation. The user stepping action itself does not take care the energy or any physics quantity associated with the killed track. Therefore if the user want to keep the total energy of an event in this case, the lost track energy need to be recorded by the user.

The same is true for user stacking or tracking actions. If the user has killed a track in these actions the all physics information associated with it would be lost and, for example, the total energy conservation be broken.

If the user wants the Geant4 kernel to take care the total energy conservation automatically when he/she has killed artificially a track, the user has to use a killer process. For example if the user uses G4UserLimits and G4UserSpecialCuts process, energy of the killed track is added to the total energy deposit.

6.4 User Information Classes

Additional user information can be associated with various Geant4 classes. There are basically two ways for the user to do this:

- derive concrete classes from base classes used in Geant4. These are classes for run, hit, digit, trajectory and trajectory point, which are discussed in *Optional User Actions* for G4Run, *Hits* for G4VHit, *Digitization* for G4VDigit, and *Tracking* for G4VTrajectory and G4VTrajectoryPoint
- create concrete classes from provided abstract base classes and associate them with classes used in Geant4. Geant4 classes which can accommodate user information classes are G4Event, G4Track, G4PrimaryVertex, G4PrimaryParticle and G4Region. These classes are discussed here.

6.4.1 G4VUserEventInformation

`G4VUserEventInformation` is an abstract class from which the user can derive his/her own concrete class for storing user information associated with a G4Event class object. It is the user's responsibility to construct a concrete class object and set the pointer to a proper G4Event object.

Within a concrete implementation of G4UserEventAction, the SetUserEventInformation() method of G4EventManager may be used to set a pointer of a concrete class object to G4Event, given that the G4Event object is available only by "pointer to const". Alternatively, the user may modify the GenerateEvent() method of his/her own RunManager to instantiate a G4VUserEventInformation object and set it to G4Event.

The concrete class object is deleted by the Geant4 kernel when the associated G4Event object is deleted.

6.4.2 G4VUserTrackInformation

This is an abstract class from which the user can derive his/her own concrete class for storing user information associated with a G4Track class object. It is the user's responsibility to construct a concrete class object and set the pointer to the proper G4Track object.

Within a concrete implementation of G4UserTrackingAction, the SetUserTrackInformation() method of G4TrackingManager may be used to set a pointer of a concrete class object to G4Track, given that the G4Track object is available only by "pointer to const".

The ideal place to copy a G4VUserTrackInformation object from a mother track to its daughter tracks is `G4UserTrackingAction::PostUserTrackingAction()`.

Listing 6.13: Copying `G4VUserTrackInformation` from mother to daughter tracks

```
void RE01TrackingAction::PostUserTrackingAction(const G4Track* aTrack)
{
  G4TrackVector* secondaries = fpTrackingManager->GimmeSecondaries();
  if(secondaries)
  {
    RE01TrackInformation* info = (RE01TrackInformation*)(aTrack->GetUserInformation());
    size_t nSeco = secondaries->size();
    if(nSeco>0)
    {
      for(size_t i=0; i < nSeco; i++)
      {
        RE01TrackInformation* infoNew = new RE01TrackInformation(info);
        (*secondaries)[i]->SetUserInformation(infoNew);
      }
    }
  }
}
```

The concrete class object is deleted by the Geant4 kernel when the associated G4Track object is deleted. In case the user wants to keep the information, it should be copied to a trajectory corresponding to the track.

6.4.3 G4VUserPrimaryVertexInformation and G4VUserPrimaryTrackInformation

These abstract classes allow the user to attach information regarding the generated primary vertex and primary particle. Concrete class objects derived from these classes should be attached to `G4PrimaryVertex` and `G4PrimaryParticle` class objects, respectively.

The concrete class objects are deleted by the Geant4 Kernel when the associated G4PrimaryVertex or G4PrimaryParticle class objects are deleted along with the deletion of G4Event.

6.4.4 G4VUserRegionInformation

This abstract base class allows the user to attach information associated with a region. For example, it would be quite beneficial to add some methods returning a boolean flag to indicate the characteristics of the region (e.g. tracker, calorimeter, etc.). With this example, the user can easily and quickly identify the detector component.

Listing 6.14: A sample region information class

```
class RE01RegionInformation : public G4VUserRegionInformation
{
  public:
    RE01RegionInformation();
    ~RE01RegionInformation();
    void Print() const;

  private:
    G4bool isWorld;
    G4bool isTracker;
    G4bool isCalorimeter;

  public:
    inline void SetWorld(G4bool v=true) {isWorld = v;}
    inline void SetTracker(G4bool v=true) {isTracker = v;}
    inline void SetCalorimeter(G4bool v=true) {isCalorimeter = v;}
    inline G4bool IsWorld() const {return isWorld;}
    inline G4bool IsTracker() const {return isTracker;}
    inline G4bool IsCalorimeter() const {return isCalorimeter;}
};
```

The following code is an example of a stepping action. Here, a track is suspended when it enters the "calorimeter region" from the "tracker region".

Listing 6.15: Sample use of a region information class

```
void RE01SteppingAction::UserSteppingAction(const G4Step * theStep)
{
  // Suspend a track if it is entering into the calorimeter

  // check if it is alive
  G4Track * theTrack = theStep->GetTrack();
  if(theTrack->GetTrackStatus()!=fAlive) { return; }

  // get region information
  G4StepPoint * thePrePoint = theStep->GetPreStepPoint();
  G4LogicalVolume * thePreLV = thePrePoint->GetPhysicalVolume()->GetLogicalVolume();
  RE01RegionInformation* thePreRInfo
   = (RE01RegionInformation*)(thePreLV->GetRegion()->GetUserInformation());
  G4StepPoint * thePostPoint = theStep->GetPostStepPoint();
  G4LogicalVolume * thePostLV = thePostPoint->GetPhysicalVolume()->GetLogicalVolume();
  RE01RegionInformation* thePostRInfo
   = (RE01RegionInformation*)(thePostLV->GetRegion()->GetUserInformation());

  // check if it is entering to the calorimeter volume
  if(!(thePreRInfo->IsCalorimeter()) && (thePostRInfo->IsCalorimeter()))
  { theTrack->SetTrackStatus(fSuspend); }
}
```

6.5 Multiple User Actions

Starting from Geant4 Version 10.3 it is possible to attach multiple instances of the same type of user action to a single run manager. This is achieved via the use of a special proxy classes to which multiple child user actions are attached. This is allowed for run-, event-, tracking- and stepping-type user actions (`G4UserRunAction`, `G4UserEventAction`,`G4UserTrackingAction`,`G4UserSteppingAction`).

The kernel still sees a single user action of each type, the proxy will forward the calls from kernel to all the attached child user actions.

Listing 6.16: An example of the use of the use of multiple user-actions.

```
#include "G4MultiRunAction.hh"
#include "G4MultiEventAction.hh"
#include "G4MultiTrackingAction.hh"
#include "G4MultiSteppingAction.hh"
//...
void MyUserActionInitialization::Build()
{
  //...
  // Example with multiple-event action, similartly
  // for the other cases
  // multi- user actions extend std::vector
  auto multiAction = new G4MultiEventAction { new MyEventAction1, new MyEventAction2 } ;
  //...
  multiAction->push_back( new MyEventAction3 );
  SetUserAction( multiAction );
  //...
}
```

6.5.1 Exceptions

This functionality is not implemented for the the stacking user action and primary generation action. There is no multiple `G4UserStackingAction` equivalent since this would require a complex handling of the case in which conflicting classifications are issued. For the case of `G4VUserPrimaryGeneratorAction` the use case of the multiple user actions is already addressed by the design of the class itself. User can implement one or more generators in the actions.

For the case of `G4MultiRunAction` only one of the child user actions can implement the `G4UserRunAction::GenerateRun()` method returning a non null, user derived `G4Run` object, otherwise an exception is thrown.

CHAPTER

SEVEN

CONTROL

7.1 Built-in Commands

Geant4 has various built-in user interface commands, each of which corresponds roughly to a Geant4 category. These commands can be used

- interactively via a (Graphical) User Interface - (G)UI,
- in a macro file via /control/execute <command>,
- within C++ code with the ApplyCommand method of G4UImanager.

Note: The availability of individual commands, the ranges of parameters, the available candidates on individual command parameters **vary** according to the implementation of your application and may even **vary** dynamically during the execution of your job.

The following is a short summary of available commands. You can also see the all available commands by executeing 'help' in your UI session.

- List of built-in commands

7.2 User Interface - Defining New Commands

7.2.1 G4UImessenger

`G4UImessenger` is a base class which represents a messenger that delivers command(s) to the destination class object. Concrete messengers are instantiated by, and owned by, the functional classes for which they provide a user interface; messengers should be deleted by those classes in their own destructors.

Your concrete messenger should have the following functionalities.

- Construct your command(s) in the constructor of your messenger.
- Destruct your command(s) in the destructor of your messenger.

These requirements mean that your messenger should keep all pointers to your command objects as its data members.

You can use `G4UIcommand` derived classes for the most frequent types of command. These derived classes have their own conversion methods according to their types, and they make implementation of the `SetNewValue()` and `GetCurrentValue()` methods of your messenger much easier and simpler.

`G4UIcommand` objects are owned by the messenger. If instantiated via *new*, they should be deleted in the messenger destructor.

For complicated commands which take various parameters, you can use the G4UIcommand base class, and construct G4UIparameter objects by yourself. You don't need to delete G4UIparameter object(s).

In the SetNewValue() and GetCurrentValue() methods of your messenger, you can compare the G4UIcommand pointer given in the argument of these methods with the pointer of your command, because your messenger keeps the pointers to the commands. Thus, you don't need to compare by command name. Please remember, in the cases where you use G4UIcommand derived classes, you should store the pointers with the types of these derived classes so that you can use methods defined in the derived classes according to their types without casting.

G4UImanager/G4UIcommand/G4UIparameter have very powerful type and range checking routines. You are strongly recommended to set the range of your parameters. For the case of a numerical value (int or double), the range can be given by a G4String using C++ notation, e.g., "X > 0 && X < 10". For the case of a string type parameter, you can set a candidate list. Please refer to the detailed descriptions below.

GetCurrentValue() will be invoked after the user's application of the corresponding command, and before the SetNewValue() invocation. This GetCurrentValue() method will be invoked only if

- at least one parameter of the command has a range
- at least one parameter of the command has a candidate list
- at least the value of one parameter is omitted and this parameter is defined as omittable and currentValueAsDefault

For the first two cases, you can re-set the range or the candidate list if you need to do so, but these "re-set" parameters are needed only for the case where the range or the candidate list varies dynamically.

A command can be "state sensitive", i.e., the command can be accepted only for a certain G4ApplicationState(s). For example, the /run/beamOn command should not be accepted when Geant4 is processing another event ("G4State_EventProc" state). You can set the states available for the command with the AvailableForStates() method.

7.2.2 G4UIcommand and its derived classes

Methods available for all derived classes

These are methods defined in the G4UIcommand base class which should be used from the derived classes.

- void SetGuidance(char*)
  Define a guidance line. You can invoke this method as many times as you need to give enough amount of guidance. Please note that the first line will be used as a title head of the command guidance.
- void availableForStates(G4ApplicationState s1,...)
  If your command is valid only for certain states of the Geant4 kernel, specify these states by this method. Currently available states are G4State_PreInit, G4State_Init, G4State_Idle, G4State_GeomClosed, and G4State_EventProc. Refer to the *Geant4 as a state machine* for meaning of each state. Please note that the Pause state had been removed from G4ApplicationState.
- void SetRange(char* range)
  Define a range of the parameter(s). Use C++ notation, e.g., "x > 0 && x < 10", with variable name(s) defined by the SetParameterName() method. For the case of a G4ThreeVector, you can set the relation between parameters, e.g., "x > y".

G4UIdirectory

This is a G4UIcommand derived class for defining a directory containing commands. It is owned by, and should be deleted in the destructor of, the associated G4UImessenger class, after all of its contained commands have been deleted.

- G4UIdirectory(char* directoryPath)
  Constructor. Argument is the (full-path) directory, which must begin and terminate with "/:.

G4UIcmdWithoutParameter

This is a `G4UIcommand` derived class for a command which takes no parameter.

- `G4UIcmdWithoutParameter(char* commandPath, G4UImessenger* theMessenger)`
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.

G4UIcmdWithABool

This is a `G4UIcommand` derived class which takes one boolean type parameter.

- `G4UIcmdWithABool(char* commandpath,G4UImanager* theMessenger)`
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- `void SetParameterName(char* paramName, G4bool omittable)`
  Define the name of the boolean parameter and set the omittable flag. If omittable is true, you should define the default value using the next method.
- `void SetDefaultValue(G4bool defVal)`
  Define the default value of the boolean parameter.
- `G4bool GetNewBoolValue(G4String paramString)`
  Convert `G4String` parameter value given by the `SetNewValue()` method of your messenger into boolean.
- `G4String convertToString(G4bool currVal)`
  Convert the current boolean value to `G4String` whichshould be returned by the `GetCurrentValue()` method of your messenger.

G4UIcmdWithAnInteger

This is a `G4UIcommand` derived class which takes one integer type parameter.

- `G4UIcmdWithAnInteger(char* commandpath, G4UImanager* theMessenger)`
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- `void SetParameterName(char* paramName, G4bool omittable)`
  Define the name of the integer parameter and set the omittable flag. If omittable is true, you should define the default value using the next method.
- `void SetDefaultValue(G4int defVal)`
  Define the default value of the integer parameter.
- `G4int GetNewIntValue(G4String paramString)`
  Convert `G4String` parameter value given by the `SetNewValue()` method of your messenger into integer.
- `G4String convertToString(G4int currVal)`
  Convert the current integer value to `G4String`, which should be returned by the `GetCurrentValue()` method of your messenger.

G4UIcmdWithADouble

This is a `G4UIcommand` derived class which takes one double type parameter.

- `G4UIcmdWithADouble(char* commandpath, G4UImanager* theMessenger)`
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- `void SetParameterName(char* paramName, G4bool omittable)`
  Define the name of the double parameter and set the omittable flag. If omittable is true, you should define the default value using the next method.
- `void SetDefaultValue(G4double defVal)`
  Define the default value of the double parameter.
- `G4double GetNewDoubleValue(G4String paramString)`
  Convert `G4String` parameter value given by the `SetNewValue()` method of your messenger into double.
- `G4String convertToString(G4double currVal)`
  Convert the current double value to `G4String` which should be returned by the `GetCurrentValue()` method of your messenger.

G4UIcmdWithAString

This is a G4UIcommand derived class which takes one string type parameter.

- G4UIcmdWithAString(char* commandpath, G4UImanager* theMessenger)
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- void SetParameterName(char* paramName, G4bool omittable)
  Define the name of the string parameter and set the omittable flag. If omittable is true, you should define the default value using the next method.
- void SetDefaultValue(char* defVal)
  Define the default value of the string parameter.
- void SetCandidates(char* candidateList)
  Define a candidate list which can be taken by the parameter. Each candidate listed in this list should be separated by a single space. If this candidate list is given, a string given by the user but which is not listed in this list will be rejected.

G4UIcmdWith3Vector

This is a G4UIcommand derived class which takes one three vector parameter.

- G4UIcmdWith3Vector(char* commandpath, G4UImanager* theMessenger)
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- void SetParameterName(char* paramNamX, char* paramNamY, char* paramNamZ, G4bool omittable)
  Define the names of each component of the three vector and set the omittable flag. If omittable is true, you should define the default value using the next method.
- void SetDefaultValue(G4ThreeVector defVal)
  Define the default value of the three vector.
- G4ThreeVector GetNew3VectorValue(G4String paramString)
  Convert the G4String parameter value given by the SetNewValue() method of your messenger into a G4ThreeVector.
- G4String convertToString(G4ThreeVector currVal)
  Convert the current three vector to G4String, which should be returned by the GetCurrentValue() method of your messenger.

G4UIcmdWithADoubleAndUnit

This is a G4UIcommand derived class which takes one double type parameter and its unit.

- G4UIcmdWithADoubleAndUnit(char* commandpath, G4UImanager* theMessenger)
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- void SetParameterName(char* paramName, G4bool omittable)
  Define the name of the double parameter and set the omittable flag. If omittable is true, you should define the default value using the next method.
- void SetDefaultValue(G4double defVal)
  Define the default value of the double parameter.
- void SetUnitCategory(char* unitCategory)
  Define acceptable unit category.
- void SetDefaultUnit(char* defUnit)
  Define the default unit. Please use this method and the SetUnitCategory() method alternatively.
- G4double GetNewDoubleValue(G4String paramString)
  Convert G4String parameter value given by the SetNewValue() method of your messenger into double. Please note that the return value has already been multiplied by the value of the given unit.
- G4double GetNewDoubleRawValue(G4String paramString)
  Convert G4String parameter value given by the SetNewValue() method of your messenger into double but without multiplying the value of the given unit.
- G4double GetNewUnitValue(G4String paramString)
  Convert G4String unit value given by the SetNewValue() method of your messenger into double.
- G4String convertToString(G4bool currVal, char* unitName)

Convert the current double value to a G4String, which should be returned by the GetCurrentValue() method of your messenger. The double value will be divided by the value of the given unit and converted to a string. Given unit will be added to the string.

G4UIcmdWith3VectorAndUnit

This is a G4UIcommand derived class which takes one three vector parameter and its unit.

- G4UIcmdWith3VectorAndUnit(char* commandpath, G4UImanager* theMessenger)
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- void SetParameterName(char* paramNamX, char* paramNamY, char* paramNamZ, G4bool omittable)
  Define the names of each component of the three vector and set the omittable flag. If omittable is true, you should define the default value using the next method.
- void SetDefaultValue(G4ThreeVector defVal)
  Define the default value of the three vector.
- void SetUnitCategory(char* unitCategory)
  Define acceptable unit category.
- void SetDefaultUnit(char* defUnit)
  Define the default unit. Please use this method and the SetUnitCategory() method alternatively.
- G4ThreeVector GetNew3VectorValue(G4String paramString)
  Convert a G4String parameter value given by the SetNewValue() method of your messenger into a G4ThreeVector. Please note that the return value has already been multiplied by the value of the given unit.
- G4ThreeVector GetNew3VectorRawValue(G4String paramString)
  Convert a G4String parameter value given by the SetNewValue() method of your messenger into three vector, but without multiplying the value of the given unit.
- G4double GetNewUnitValue(G4String paramString)
  Convert a G4String unit value given by the SetNewValue() method of your messenger into a double.
- G4String convertToString(G4ThreeVector currVal, char* unitName)
  Convert the current three vector to a G4String which should be returned by the GetCurrentValue() method of your messenger. The three vector value will be divided by the value of the given unit and converted to a string. Given unit will be added to the string.

Additional comments on the SetParameterName() method

You can add one additional argument of G4bool type for every SetParameterName() method mentioned above. This additional argument is named currentAsDefaultFlag and the default value of this argument is false. If you assign this extra argument as true, the default value of the parameter will be overriden by the current value of the target class.

7.2.3 An example messenger

This example is of G4ParticleGunMessenger, which is made by inheriting G4UIcommand.

Listing 7.1: An example of G4ParticleGunMessenger.hh.

```
#ifndef G4ParticleGunMessenger_h
#define G4ParticleGunMessenger_h 1

class G4ParticleGun;
class G4ParticleTable;
class G4UIcommand;
class G4UIdirectory;
class G4UIcmdWithoutParameter;
class G4UIcmdWithAString;
class G4UIcmdWithADoubleAndUnit;
class G4UIcmdWith3Vector;
```

```
class G4UIcmdWith3VectorAndUnit;

#include "G4UImessenger.hh"
#include "globals.hh"

class G4ParticleGunMessenger: public G4UImessenger
{
  public:
    G4ParticleGunMessenger(G4ParticleGun * fPtclGun);
    ~G4ParticleGunMessenger();

  public:
    void SetNewValue(G4UIcommand * command,G4String newValues);
    G4String GetCurrentValue(G4UIcommand * command);

  private:
    G4ParticleGun * fParticleGun;
    G4ParticleTable * particleTable;

  private: //commands
    G4UIdirectory *             gunDirectory;
    G4UIcmdWithoutParameter *   listCmd;
    G4UIcmdWithAString *        particleCmd;
    G4UIcmdWith3Vector *        directionCmd;
    G4UIcmdWithADoubleAndUnit * energyCmd;
    G4UIcmdWith3VectorAndUnit * positionCmd;
    G4UIcmdWithADoubleAndUnit * timeCmd;

};

#endif
```

Listing 7.2: An example of `G4ParticleGunMessenger.cc`.

```
#include "G4ParticleGunMessenger.hh"
#include "G4ParticleGun.hh"
#include "G4Geantino.hh"
#include "G4ThreeVector.hh"
#include "G4ParticleTable.hh"
#include "G4UIdirectory.hh"
#include "G4UIcmdWithoutParameter.hh"
#include "G4UIcmdWithAString.hh"
#include "G4UIcmdWithADoubleAndUnit.hh"
#include "G4UIcmdWith3Vector.hh"
#include "G4UIcmdWith3VectorAndUnit.hh"
#include <iostream.h>

G4ParticleGunMessenger::G4ParticleGunMessenger(G4ParticleGun * fPtclGun)
:fParticleGun(fPtclGun)
{
  particleTable = G4ParticleTable::GetParticleTable();

  gunDirectory = new G4UIdirectory("/gun/");
  gunDirectory->SetGuidance("Particle Gun control commands.");

  listCmd = new G4UIcmdWithoutParameter("/gun/list",this);
  listCmd->SetGuidance("List available particles.");
  listCmd->SetGuidance(" Invoke G4ParticleTable.");

  particleCmd = new G4UIcmdWithAString("/gun/particle",this);
  particleCmd->SetGuidance("Set particle to be generated.");
  particleCmd->SetGuidance(" (geantino is default)");
  particleCmd->SetParameterName("particleName",true);
  particleCmd->SetDefaultValue("geantino");
  G4String candidateList;
  G4int nPtcl = particleTable->entries();
```

```
  for(G4int i=0;i<nPtcl;i++)
  {
    candidateList += particleTable->GetParticleName(i);
    candidateList += " ";
  }
  particleCmd->SetCandidates(candidateList);

  directionCmd = new G4UIcmdWith3Vector("/gun/direction",this);
  directionCmd->SetGuidance("Set momentum direction.");
  directionCmd->SetGuidance("Direction needs not to be a unit vector.");
  directionCmd->SetParameterName("Px","Py","Pz",true,true);
  directionCmd->SetRange("Px != 0 || Py != 0 || Pz != 0");

  energyCmd = new G4UIcmdWithADoubleAndUnit("/gun/energy",this);
  energyCmd->SetGuidance("Set kinetic energy.");
  energyCmd->SetParameterName("Energy",true,true);
  energyCmd->SetDefaultUnit("GeV");
  energyCmd->SetUnitCandidates("eV keV MeV GeV TeV");

  positionCmd = new G4UIcmdWith3VectorAndUnit("/gun/position",this);
  positionCmd->SetGuidance("Set starting position of the particle.");
  positionCmd->SetParameterName("X","Y","Z",true,true);
  positionCmd->SetDefaultUnit("cm");
  positionCmd->SetUnitCandidates("micron mm cm m km");

  timeCmd = new G4UIcmdWithADoubleAndUnit("/gun/time",this);
  timeCmd->SetGuidance("Set initial time of the particle.");
  timeCmd->SetParameterName("t0",true,true);
  timeCmd->SetDefaultUnit("ns");
  timeCmd->SetUnitCandidates("ns ms s");

  // Set initial value to G4ParticleGun
  fParticleGun->SetParticleDefinition( G4Geantino::Geantino() );
  fParticleGun->SetParticleMomentumDirection( G4ThreeVector(1.0,0.0,0.0) );
  fParticleGun->SetParticleEnergy( 1.0*GeV );
  fParticleGun->SetParticlePosition(G4ThreeVector(0.0*cm, 0.0*cm, 0.0*cm));
  fParticleGun->SetParticleTime( 0.0*ns );
}
```

```
G4ParticleGunMessenger::~G4ParticleGunMessenger()
{
  delete listCmd;
  delete particleCmd;
  delete directionCmd;
  delete energyCmd;
  delete positionCmd;
  delete timeCmd;
  delete gunDirectory;
}

void G4ParticleGunMessenger::SetNewValue(
  G4UIcommand * command,G4String newValues)
{
  if( command==listCmd )
  { particleTable->dumpTable(); }
  else if( command==particleCmd )
  {
    G4ParticleDefinition* pd = particleTable->findParticle(newValues);
    if(pd != NULL)
    { fParticleGun->SetParticleDefinition( pd ); }
  }
  else if( command==directionCmd )
  { fParticleGun->SetParticleMomentumDirection(directionCmd->
     GetNew3VectorValue(newValues)); }
  else if( command==energyCmd )
  { fParticleGun->SetParticleEnergy(energyCmd->
     GetNewDoubleValue(newValues)); }
```

```
  else if( command==positionCmd )
  { fParticleGun->SetParticlePosition(
     directionCmd->GetNew3VectorValue(newValues)); }
  else if( command==timeCmd )
  { fParticleGun->SetParticleTime(timeCmd->
     GetNewDoubleValue(newValues)); }
}

G4String G4ParticleGunMessenger::GetCurrentValue(G4UIcommand * command)
{
  G4String cv;

  if( command==directionCmd )
  { cv = directionCmd->ConvertToString(
     fParticleGun->GetParticleMomentumDirection()); }
  else if( command==energyCmd )
  { cv = energyCmd->ConvertToString(
     fParticleGun->GetParticleEnergy(),"GeV"); }
  else if( command==positionCmd )
  { cv = positionCmd->ConvertToString(
     fParticleGun->GetParticlePosition(),"cm"); }
  else if( command==timeCmd )
  { cv = timeCmd->ConvertToString(
     fParticleGun->GetParticleTime(),"ns"); }
  else if( command==particleCmd )
  { // update candidate list
    G4String candidateList;
    G4int nPtcl = particleTable->entries();
    for(G4int i=0;i<nPtcl;i++)
    {
      candidateList += particleTable->GetParticleName(i);
      candidateList += " ";
    }
    particleCmd->SetCandidates(candidateList);
  }
  return cv;
}
```

7.2.4 How to control the output of G4cout/G4cerr

Instead of *std::cout* and *std::cerr*, Geant4 uses `G4cout` and `G4cerr`. Output streams from `G4cout/G4cerr` are handled by `G4UImanager` which allows the application programmer to control the flow of the stream. Output strings may therefore be displayed on another window or stored in a file. This is accomplished as follows:

1. Derive a class from `G4UIsession` and implement the two methods:

   ```
   G4int ReceiveG4cout(const G4String& coutString);
   G4int ReceiveG4cerr(const G4String& cerrString);
   ```

   These methods receive the string stream of `G4cout` and `G4cerr`, respectively. The string can be handled to meet specific requirements. The following sample code shows how to make a log file of the output stream:

   ```
   ostream logFile;
   logFile.open("MyLogFile");
   G4int MySession::ReceiveG4cout(const G4String& coutString)
   {
     logFile << coutString << flush;
     return 0;
   }
   ```

2. Set the destination of `G4cout/G4cerr` using `G4UImanager::SetCoutDestination(session)`. Typically this method is invoked from the constructor of `G4UIsession` and its derived classes, such as `G4UIGAG/G4UIteminal`. This method sets the destination of `G4cout/G4cerr` to the session. For example, when the following code appears in the constructor of `G4UIterminal`, the method

`SetCoutDestination(this)` tells *UImanager* that this instance of `G4UIterminal` receives the stream generated by `G4cout`.

```
G4UIterminal::G4UIterminal()
{
  UI = G4UImanager::GetUIpointer();
  UI->SetCoutDestination(this);
  //  ...
}
```

Similarly, `UI->SetCoutDestination(NULL)` must be added to the destructor of the class.

3. Write or modify the main program. To modify `exampleB1` to produce a log file, derive a class as described in step 1 above, and add the following lines to the main program:

```
#include "MySession.hh"
main()
{
  // get the pointer to the User Interface manager
  G4UImanager* UI = G4UImanager::GetUIpointer();
  // construct a session which receives G4cout/G4cerr
  MySession * LoggedSession = new MySession;
  UI->SetCoutDestination(LoggedSession);
  // session->SessionStart(); // not required in this case
  // .... do simulation here ...

  delete LoggedSession;
  return 0;
}
```

Note: `G4cout/G4cerr` should not be used in the constructor of a class if the instance of the class is intended to be used as `static`. This restriction comes from the language specification of C++. See the documents below for details:

- M.A.Ellis, B.Stroustrup, "Annotated C++ Reference Manual", Section 3.4 *[Ellis1990]*
- P.J.Plauger, "The Draft Standard C++ Library" *[Plauger1995]*

CHAPTER

EIGHT

VISUALIZATION

8.1 Introduction to Visualization

The Geant4 visualization system was developed in response to a diverse set of requirements:

1. Quick response to study geometries, trajectories and hits
2. High-quality output for publications
3. Flexible camera control to debug complex geometries
4. Tools to show volume overlap errors in detector geometries
5. Interactive picking to get more information on visualized objects

No one graphics system is ideal for all of these requirements, and many of the large software frameworks into which Geant4 has been incorporated already have their own visualization systems, so Geant4 visualization was designed around an abstract interface that supports a diverse family of graphics systems. Some of these graphics systems use a graphics library compiled with Geant4, such as OpenGL, Qt, while others involve a separate application, such as HepRApp or DAWN.

Most examples include a vis.mac to perform typical visualization for that example. The macro includes optional code which you can uncomment to activate additional visualization features.

8.1.1 What Can be Visualized

Simulation data can be visualized:

- Detector components
  - A hierarchical structure of physical volumes
  - A piece of physical volume, logical volume, and solid
- Particle trajectories and tracking steps
- Hits of particles in detector components
- Scoring data

Other user defined objects can be visualized:

- Polylines, such as coordinate axes
- 3D Markers, such as eye guides
- Text, descriptive character strings, comments or titles
- Scales
- Logos

8.1.2 You have a Choice of Visualization Drivers

The many graphics systems that Geant4 supports are complementary to each other.

- OpenGL
  - View directly from Geant4
  - Requires addition of GL libraries that are freely avialable for all operating systems (and pre-installed on many)
  - Rendered, photorealistic image with some interactive features
  - zoom, rotate, translate
  - Fast response (can usually exploit full potential of graphics hardware)
  - Print to EPS (vector and pixel graphics)
- Qt
  - View directly from Geant4
  - Requires addition of Qt and GL libraries that are freely available on most operating systems
  - Rendered, photorealistic image
  - Many interactive features
  - zoom, rotate, translate
  - Fast response (can usually exploit full potential of graphics hardware)
  - Expanded printing ability (vector and pixel graphics)
  - Easy interface to make movies
- OpenInventor
  - View directly from Geant4
  - Requires addition of OpenInventor libraries (freely available for most Linux systems).
  - Rendered, photorealistic image
  - Many interactive features
  - zoom, rotate, translate
  - click to "see inside" opaque volumes
  - Fast response (can usually exploit full potential of graphics hardware)
  - Expanded printing ability (vector and pixel graphics)
- HepRep
  - Create a file to view in a HepRep browser such as HepRApp, FRED or WIRED4
  - Requires a HepRep browser (above options work on any operating system)
  - Wireframe or simple area fills (not photorealistic)
  - Many interactive features
  - zoom, rotate, translate
  - click to show attributes (momentum, etc.)
  - special projections (FishEye, etc.)
  - control visibility from hierarchical (tree) view of data
  - Hierarchical view of the geometry
  - Export to many vector graphic formats (PostScript, PDF, etc.)
- DAWN
  - Create a file to view in the DAWN Renderer
  - Requires DAWN, available for all Linux and Windows systems.
  - Rendered, photorealistic image
  - No interactive features
  - Highest quality technical rendering - output to vector PostScript
- VRML
  - Create a file to view in any VRML browser (some as web browser plug-ins).
  - Requires VRML browser (many different choices for different operating systems).
  - Rendered, photorealistic image with some interactive features
  - zoom, rotate, translate
  - Limited printing ability (pixel graphics, not vector graphics)
- RayTracer
  - Create a jpeg file
  - Forms image by using Geant4's own tracking to follow photons through the detector
  - Can show geometry but not trajectories
  - Can render any geometry that Geant4 can handle (such as Boolean solids)

  - Supports shadows, transparency and mirrored surfaces
- gMocren
  - Create a gMocren file suiable for viewing in the gMocren volume data visualization application
  - Represents three dimensional volume data such as radiation therapy dose
  - Can also include geometry and trajectory information
- ASCIITree
  - Text dump of the geometry hierarchy
  - Not graphical
  - Control over level of detail to be dumped
  - Can calculate mass and volume of any hierarchy of volumes
- Wt (WARNING: this driver is experimental and should be used with caution)
  - View directly from Geant4 across a Web browser.
  - Requires addition of Wt librarie that is freely available on most operating systems.
  - Require a Web browser with WebGL enable.
  - Rendered, photorealistic image
  - Many interactive features
  - zoom, rotate, translate
  - Fast response (can usually exploit full potential of graphics hardware)

8.1.3 Choose the Driver that Meets Your Needs

- If you want very responsive photorealistic graphics (and have the OpenGL libraries installed)
  - OpenGL is a good solution (if you have the Motif extensions, this also gives GUI control)
- If you want to have the User Interface and all Visualization windows in the same window
  - Only Qt can do that
- If you want very responsive photorealistic graphics plus more interactivity (and have the OpenInventor or Qt libraries installed)
  - OpenInventor or Qt are good solutions
- If you want GUI control, very responsive photorealistic graphics plus more interactivity (and have the Qt libraries installed).
  - Qt is a good solution
- If you want GUI control, want to be able to pick on items to inquire about them (identity, momentum, etc.), perhaps want to render to vector formats, and a wireframe look will do
  - HepRep will meet your needs
- If you want to render highest quality photorealistic images for use in a poster or a technical design report, and you can live without quick rotate and zoom
  - DAWN is the way to go
- If you want to render to a 3D format that others can view in a variety of commodity browsers (including some web browser plug-ins)
  - VRML is the way to go
- If you want to visualize a geometry that the other visualization drivers can't handle, or you need transparency or mirrors, and you don't need to visualize trajectories
  - RayTracer will do it
- If you want to visualization volume data, such as radiation therapy dose distributions
  - gMocren will meet your needs
- If you just want to quickly check the geometry hierarchy, or if you want to calculate the volume or mass of any geometry hierarchy
  - ASCIITree will meet your needs
- If you to interact with your application with a Web Broswer
  - Wt will do it.

> **Warning:** this driver is experimental and should be used with caution

- You can also add your own visualization driver.
  - Geant4's visualization system is modular. By creating just three new classes, you can direct Geant4 information to your own visualization system.

8.1.4 Controlling Visualization

Your Geant4 code stays basically the same no matter which driver you use.

Visualization is performed either with commands or from C++ code.

- Some visualization drivers work directly from Geant4
  - OpenGL
  - Qt
  - OpenInventor
  - RayTracer
  - ASCIITree
  - Wt (WARNING: this driver is experimental and should be used with caution)
- For other visualization drivers, you first have Geant4 produce a file, and then you have that file rendered by another application (which may have GUI control)
  - HepRep
  - DAWN
  - VRML
  - gMocren

8.1.5 Visualization Details

The following sections of this guide cover the details of Geant4 visualization:

- Adding Visualization to Your Executable
- The Visualization Drivers
- Controlling Visualization from Commands
- Controlling Visualization from Compiled Code
- Visualization Attributes
- Enhanced Trajectory Drawing
- Polylines, Markers and Text
- Making a Movie

Other useful references for Geant4 visualization outside of this user guide:

- Introduction to Geant4 Visualization (pdf, ppt)
- Geant4 Visualization Commands (pdf, ppt)
- Geant4 Advanced Visualization (pdf, ppt)
- How to Make a Movie (pdf, ppt)
- Geant4 Visualization Tutorial using the HepRApp HepRep Browser
- Geant4 Visualization Tutorial using the OpenGL Event Display
- Geant4 Visualization Tutorial using the DAWN Event Display
- Macro files distributed in Geant4 source in basic examples, `vis.mac` and `examples/basic/B4/macros/visTutor/`.

8.2 Adding Visualization to Your Executable

This section explains how to incorporate your selected visualization drivers into the `main()` function and create an executable for it. In order to perform visualization with your Geant4 executable, you must compile it with support for the required visualization driver(s). You may be dazzled by the number of choices of visualization driver, but you need not use all of them at one time.

8.2.1 Installing Visualization Drivers

Depending on what has been installed on your system, several kinds of visualization driver are available. One or many drivers may be chosen for realization in compilation, depending on your visualization requirements. Features and notes on each driver are briefly described in *The Visualization Drivers*, along with links to detailed web pages for the various drivers.

Note that not all drivers can be installed on all systems; *Table of VisDrivers* in *The Visualization Drivers* lists all the available drivers and the platforms on which they can be installed. For any of the visualization drivers to work, the corresponding graphics system must be installed beforehand.

Visualization drivers that do not depend on external libraries are by default incorporated into Geant4 libraries during their installation. Here "installation of Geant4 libraries" means the generation of Geant4 libraries by compilation. The automatically incorporated visualization drivers are: DAWNFILE, HepRepFile, HepRepXML, RayTracer, VRML1FILE, VRML2FILE and ATree and GAGTree.

The OpenGL, Qt, OpenInventor and RayTracerX drivers are not incorporated by default. Nor are the DAWN-Network and VRML-Network drivers, because they require the network setting of the installed machine. These drivers must be selected when you build the Geant4 Toolkit itself. This proceedure is described in detail in the Installation Guide, to which you should refer.

8.2.2 How to Realize Visualization Drivers in an Executable

You can realize and use any of the visualization driver(s) you want in your Geant4 executable, provided they are among the set installed beforehand into the Geant4 libraries. A warning will appear if this is not the case.

In order to realize visualization drivers, you must instantiate and initialize a subclass of `G4VisManager` that implements the pure virtual function `RegisterGraphicsSystems()`. This subclass must be compiled in the user's domain to force the loading of appropriate libraries in the right order. The easiest way to do this is to use `G4VisExecutive`, a provided class with included implementation. `G4VisExecutive` is sensitive to the `G4VIS_USE...` variables mentioned below.

If you do wish to write your own subclass, you may do so. You will see how to do this by looking at `G4VisExecutive.icc`. A typical extract is:

```
...
  RegisterGraphicsSystem (new G4DAWNFILE);
...
#ifdef G4VIS_USE_OPENGLX
  RegisterGraphicsSystem (new G4OpenGLImmediateX);
  RegisterGraphicsSystem (new G4OpenGLStoredX);
#endif
...
```

If you wish to use `G4VisExecutive` but register an additional graphics system, `XXX` say, you may do so either before or after initializing:

```
visManager->RegisterGraphicsSytem(new XXX);
visManager->Initialize();
```

By default, you get the DAWNFILE, HepRepFile, RayTracer, VRML1FILE, VRML2FILE, ATree and GAGTree drivers. Additionally, you may choose from the OpenGL-Xlib, OpenGL-Motif, Qt, OpenInventor, RayTracerX, DAWN-Network and VRML-Network drivers, each of which can be set at "`Cmake`" or "`GNUMakefile step`", see *Usage of Geant4Config.cmake*. (Of course, this has to be chosen from the set incorporated into the Geant4 libraries during their compilation.)

For more details, see *The Visualization Drivers* and pages linked from there.

8.2.3 Visualization Manager

Visualization procedures are controlled by the "Visualization Manager", a class which must inherit from `G4VisManager` defined in the visualization category. Most users will find that they can just use the default visualization manager, `G4VisExecutive`. The Visualization Manager accepts users' requests for visualization, processes them, and passes the processed requirements to the abstract interface, i.e., to the currently selected visualization driver.

8.2.4 How to Write the `main()` Function

In order for your Geant4 executable to perform visualization, you must instantiate and initialize "your" Visualization Manager in the `main()` function. The core of the Visualization Manager is the class `G4VisManager`, defined in the visualization category. This class requires that one pure virtual function be implemented, namely, `void RegisterGraphicsSystems()`. The easiest way to do this is to use `G4VisExecutive`, as described above (but you may write your own class - see above).

The *example* shows the form of the `main()` function.

Listing 8.1: The form of the `main()` function.

```
//----- C++ source codes: Instantiation and initialization of G4VisManager

.....
// Your Visualization Manager
#include "G4VisExecutive.hh"
.....

// Instantiation and initialization of the Visualization Manager
#ifdef G4VIS_USE
G4VisManager* visManager = new G4VisExecutive;
// G4VisExecutive can take a verbosity argument - see /vis/verbose guidance.
// G4VisManager* visManager = new G4VisExecutive("Quiet");
visManager->Initialize();
#endif

.....
#ifdef G4VIS_USE
delete visManager;
#endif

//----- end of C++
```

Alternatively, you can implement an empty `RegisterGraphicsSystems()` function, and register visualization drivers you want directly in your `main()` function. See the next *example*.

Listing 8.2: An alternative style for the `main()` function.

```
//----- C++ source codes: How to register a visualization driver directly
//                        in main() function

.....
G4VisManager* visManager = new G4VisExecutive;
visManager -> RegisterGraphicsSystem (new MyGraphicsSystem);
```

```
.....
delete visManager

//----- end of C++
```

Do not forget to delete the instantiated Visualization Manager by yourself. Note that a graphics system for Geant4 Visualization may run as a different process. In that case, the destructor of `G4VisManager` might have to terminate the graphics system and/or close the connection.

We recommend that the instantiation, initialization, and deletion of the Visualization Manager be protected by C-pre-processor commands, as in the basic examples. To see the behaviour of C-pre-processor macro `G4VIS_USE` and `G4UI_USE`, see *Usage of Geant4Config.cmake*.

The *example* shows an example of the `main()` function available for Geant4 Visualization.

Listing 8.3: An example of the `main()` function available for Geant4 Visualization.

```
//----- C++ source codes: An example of main() for visualization
.....
#include "G4VisExecutive.hh"
#include "G4UIExecutive.hh"
.....

int main(int argc, char *argv[])
{
     // Run Manager
     G4RunManager * runManager = new G4RunManager;

     // Detector components
     runManager->set_userInitialization(new MyDetectorConstruction);
     runManager->set_userInitialization(new MyPhysicsList);

     // UserAction classes.
     runManager->set_userAction(new MyRunAction);
     runManager->set_userAction(new MyPrimaryGeneratorAction);
     runManager->set_userAction(new MyEventAction);
     runManager->set_userAction(new MySteppingAction);

#ifdef G4VIS_USE
     G4VisManager* visManager = new G4VisExecutive;
     visManager->Initialize(argc, argv);
#endif

     // Define (G)UI
#ifdef G4UI_USE
     G4UIExecutive * ui = new G4UIExecutive;
     ui->SessionStart();

     delete ui;
#endif
      delete runManager;

#ifdef G4VIS_USE
      delete visManager;
#endif

      return 0;
}

//----- end of C++
```

Useful information on incorporated visualization drivers can be displayed in initializing the Visualization Manager. This is done by setting the verbosity flag to an appropriate number or string:

```
Simple graded message scheme - give first letter or a digit:
 0) quiet,         // Nothing is printed.
 1) startup,       // Startup and endup messages are printed...
 2) errors,        // ...and errors...
 3) warnings,      // ...and warnings...
 4) confirmations, // ...and confirming messages...
 5) parameters,    // ...and parameters of scenes and views...
 6) all            // ...and everything available.
```

For example, in your `main()` function, write the following code:

```
...
G4VisManager* visManager = new G4VisExecutive("Quiet");
visManager->Initialize();
...
```

(This can also be set with the `/vis/verbose` command.)

8.3 The Visualization Drivers

As explained in the Introduction to Visualization , Geant4 provides many different choices of visualization systems. Features and notes on each driver are briefly described here along with links to detailed web pages for the various drivers.

Details are given below for:

- *OpenGL*
- *Qt*
- *OpenInventor*
- *OpenInventor Extended Viewer*
- *HepRepFile*
- *HepRepXML*
- *DAWN*
- *VRML*
- *RayTracer*
- *gMocren*
- *ASCIITree*
- *GAG Tree*
- *XML Tree*
- *Wt*

8.3.1 Availability of drivers on the supported systems

Table *VisDrivers* lists required graphics systems and supported platforms for the various visualization drivers

Driver	Required Graphics System	Platform
OpenGL-Xlib	OpenGL	Linux, UNIX, Mac with Xlib
OpenGL-Motif	OpenGL	Linux, UNIX, Mac with Motif
OpenGL-Win32	OpenGL	Windows
Qt	Qt, OpenGL	Linux, UNIX, Mac, Windows
Wt	Wt for server side/Web browser for clients.	Linux, UNIX, Mac, Windows
OpenInventor-X	OpenInventor (Coin3D), OpenGL	Linux, UNIX, Mac with Xlib and Motif
OpenInventor-X-Extended	OpenInventor (Coin3D), OpenGL	Linux, UNIX, Mac with Xlib and Motif
OpenInventor-Win32	OpenInventor, OpenGL	Windows
HepRep	HepRApp, FRED or WIRED4 HepRep Browser	Linux, UNIX, Mac, Windows
DAWNFILE	Fukui Renderer DAWN	Linux, UNIX, Mac, Windows
DAWN-Network	Fukui Renderer DAWN	Linux, UNIX
VRMLFILE	any VRML viewer	Linux, UNIX, Mac, Windows
VRML-Network	any network-enabled VRML viewer	Linux, UNIX
RayTracer	any JPEG viewer	Linux, UNIX, Mac, Windows
ASCIITree	none	Linux, UNIX, Mac, Windows
GAGTree	GAG	Linux, UNIX, Mac, Windows
XMLTree	any XML viewer	Linux, UNIX, Mac, Windows

Table: Required graphics systems and supported platforms for the various visualization drivers.

8.3.2 OpenGL

These drivers have been developed by John Allison and Andrew Walkden (University of Manchester). It is an interface to the de facto standard 3D graphics library, OpenGL. It is well suited for real-time fast visualization and demonstration. Fast visualization is realized with hardware acceleration, reuse of shapes stored in a display list, etc. NURBS visualization is also supported.

Several versions of the OpenGL drivers are prepared. Versions for Xlib, Motif, Qt and Win32 platforms are available by default. For each version, there are two modes: immediate mode and stored mode. The former has no limitation on data size, and the latter is fast for visualizing large data repetitively, and so is suitable for animation.

Output can be exported to EPS (both vector and pixel graphics) using vis/ogl/printEPS.

More information can be found here: *How to save a view to an image file*

If you want to open a OGL viewer, the generic way is:

```
/vis/open OGL
```

According to your G4VIS_USE... variables it will open the correct viewer. By default, it will be open in stored mode. You can specify to open an "OGLS" or "OGLI" viewer, or even "OGLSXm","OGLIXm",... If you don't have Motif or Qt, all control is done from Geant4 commands:

```
/vis/open OGLIX
/vis/viewer/set/viewpointThetaPhi 70 20
/vis/viewer/zoom 2
etc.
```

But if you have Motif libraries or Qt install, you can control Geant4 from Motif widgets or mouse with Qt:

```
/vis/open OGLSQt
```

The OpenGL driver added Smooth shading and Transparency since Geant4 release 8.0.

Further information (OpenGL and Mesa):

- http://www.opengl.org/
- http://www.mesa3d.org
- http://geant4.slac.stanford.edu/Presentations/vis/G4OpenGLTutorial/G4OpenGLTutorial.html using the OpenGL Graphics System

8.3.3 Qt

This driver has been developed by Laurent Garnier (IN2P3, LAL Orsay). It is an interface to the powerful application framework, Qt, now free on most platforms. This driver also requires the OpenGL library.

The Qt driver is well suited for real-time fast visualization and demonstration. Fast visualization is realized with hardware acceleration, reuse of shapes stored in a display list, etc. NURBS visualization is also supported. All OpenGL features are implemented in the Qt driver, but one also gets mouse control of rotation/translation/zoom, the ability to save your scene in many formats (both vector and pixel graphics) and an easy interface for making movies.

Two display modes are available: Immediate mode and Stored mode. The former has no limitation on data size, and the latter is fast for visualizing large data repetitively, and so is suitable for animation.

This driver has the feature to open a vis window into the UI window as a new tab. You can have as many tabs you want and mix them from Stored or Immediate mode. To see the visualization window in the UI:

```
/vis/open OGL  (Generic way. For Stored mode if you have define your G4VIS_USE_QT variable)
or
/vis/open OGLI  (for Immediate mode)
or
/vis/open OGLS  (for Stored mode)
or
/vis/open OGLIQt  (for Immediate mode)
or
/vis/open OGLSQt  (for Stored mode)
```

Further information (Qt):

- Qt
- Geant4 Visualization Tutorial using the Qt Driver

8.3.4 OpenInventor

These drivers were developed by Jeff Kallenbach (FNAL) and Guy Barrand (IN2P3) based on the Hepvis class library originated by Joe Boudreau (Pittsburgh University). The OpenInventor drivers and the Hepvis class library are based on the well-established OpenInventor technology for scientific visualization. They have high extendibility. They support high interactivity, e.g., attribute editing of picked objects. Some OpenInventor viewers support "stereoscopic" effects.

It is also possible to save a visualized 3D scene as an OpenInventor-formatted file, and re-visualize the scene afterwards.

Because it is connected directly to the Geant4 kernel, using same language as that kernel (C++), OpenInventor systems can have direct access to Geant4 data (geometry, trajectories, etc.).

Because OpenInventor uses OpenGL for rendering, it supports lighting and transparency.

OpenInventor provides thumbwheel control to rotate and zoom.

OpenInventor supports picking to ask about data. [Control Clicking] on a volume turns on rendering of that volume's daughters. [Shift Clicking] a daughter turns that rendering off: If modeling opaque solid, effect is like opening a box to look inside.

Further information (HEPVis and OpenScientist):

- Geant4 Inventor Visualization with OpenScientist
- Overall OpenScientist Home
- HEPVis

Further information (OpenInventor):

- http://oss.sgi.com/projects/inventor
- Josie Wernecke, "The Inventor Mentor", Addison Wesley (ISBN 0-201-62495-8)
- Josie Wernecke, "The Inventor Toolmaker", Addison Wesley (ISBN 0-201-62493-1)
- "The Open Inventor C++ Reference Manual", Addison Wesley (ISBN 0-201-62491-5)

8.3.5 OpenInventor Extended Viewer

This driver was developed by Rastislav Ondrasek, Pierre-Luc Gagnon and Frederick Jones (TRIUMF). It extends the functionality of the OpenInventor driver, described in the previous section, by adding a number of new features to the viewer.

At present this driver is supported only on Linux/Unix/MacOS platforms and is not available for Windows. It requires the Coin3D implementation of OpenInventor.

All of the viewer functions and behavior of the basic OpenInventor driver are included and remain unchanged. The added viewer functions are implemented via dropdown menu items, buttons, a new navigation panel, and keyboard and mouse inputs.

Reference path navigation

Most of the added features are concerned with navigation along a "reference path" which is a piecewise linear path through the geometry. The reference path can be any particle trajectory, which may be chosen in the application by an attaching a visualization attribute to it, or at run time by selecting a trajectory with the mouse. Via Load and Save menu items, a reference path can be read from a file and the current reference path can be written to a file.

Once a reference path is established, the viewer pops up a Navigation Panel showing a list of all elements in the geometry, ordered by their "distance" along the reference path (based on the perpendicular from the element center to the path).

Navigation controls

[L,R,U,D refer to the arrow keys on the keyboard]

- Select an element from the list: navigate along the path to the element's "location" (distance along the reference path).
- Shift-L and Shift-R: navigate to the previous or next element on the path (with wraparound).
- L and R: rotate 90 degrees around the vertical axis
- U and D: rotate 90 degrees around the path
- Ctrl-L and Ctrl-R: rotate 90 degrees around the horizontal axis

All these keys have a "repeat" function for continuous motion.

The rotation keys put the camera in a definite orientation, whereas The Shift-L and Shift-R keys can be used to "fly" along the path in whatever camera orientation is in effect. NOTE: if this appears to be "stuck", try switching from orthonormal camera to perspective camera ("cube" viewer button).

Menu Items:

- Tools / Go to start of reference path: useful if you get lost

- Tools / Invert reference path: flips the direction of travel and the distance readout

Reference path animation

This is a special mode which flys the camera steadily along the path, without wraparound. The controls are:

- Tools Menu - Animate Ref Particle: start animation mode
- Page-Up: increase speed
- Page-Down: decrease speed
- U (arrow key): raise camera
- D (arrow key): lower camera
- ESC: exit animation mode

For suitable geometries the U and D keys can be used to get "Star Wars" style fly-over and fly-under effects.

Bookmarks

At any time, the viewpoint and other camera parameters can be saved in a file as a labelled "bookmark". The view can then be restored later in the current run or in another run.

The default name for the bookmark file is ".bookmarkFile" The first time a viewpoint is saved, this file will be created if it does not already exist. When the viewer is first opened, it will automatically read this file if present and load the viewpoints into the left-hand panel of the viewer's auxiliary window.

Controls:

- Select viewpoint from list: restore this view
- Right-arrow VIEWER button: go to next viewpoint Left-arrow VIEWER button: go to next viewpoint
- "Floppy Disk" button: save current view. The user can type in a label for the view, or use the default label provided.
- File Menu - Open Viewpoint File: loads an existing bookmark file
- File Menu - New Viewpoint File: creates a new bookmark file for saving subsequent views

Special picking modes

Controls:

- "Console" VIEWER button: enable brief trajectory picking and mouse-over element readout For trajectories, the list of all trajectory points is replaced by the first and last point only, allowing easier identification of the particle without scrolling back. Passing the mouse over an element will give a readout of the volume name, material, and position on the reference path.
- "Star" VIEWER button: select new reference path The cursor will change to a small cross (+) after which a trajectory can be selected to become the new reference path.

Convenience feature

It is now possible to escape from the Open Inventor viewer without using the mouse.

In addition to the File - Escape menu item, pressing the "e" key on the keyboard will exit from the viewer's X event loop. The viewer will become inactive and control will return to the Geant4 UI prompt.

8.3.6 HepRepFile

The HepRepFile driver creates a HepRep XML file in the HepRep1 format suitable for viewing with the HepRApp HepRep Browser.

The HepRep graphics format is further described at http://www.slac.stanford.edu/~perl/heprep .

To write just the detector geometry to this file, use the command:

```
/vis/viewer/flush
```

Or, to also include trajectories and hits (after the appropriate /vis/viewer/add/trajectories or /vis/viewer/add/hits commands), just issue:

```
/run/beamOn 1
```

HepRepFile will write a file called G4Data0.heprep to the current directory. Each subsequent file will have a file name like G4Data1.heprep, G4Data2.heprep, etc.

View the file using the HepRApp HepRep Browser, available from:

http://www.slac.stanford.edu/~perl/HepRApp/ .

HepRApp allows you to pick on volumes, trajectories and hits to find out their associated HepRep Attributes, such as volume name, particle ID, momentum, etc. These same attributes can be displayed as labels on the relevant objects, and you can make visibility cuts based on these attributes ("show me only the photons", or "omit any volumes made of iron").

HepRApp can read heprep files in zipped format as well as unzipped, so you can save space by applying gzip to the heprep file. This will reduce the file to about five percent of its original size.

Several commands are available to override some of HepRepFile's defaults

- You can specify a different directory for the heprep output files by using the setFileDir command, as in:

  ```
  /vis/heprep/setFileDir <someOtherDir/someOtherSubDir>
  ```

- You can specify a different file name (the part before the number) by using the setFileName command, as in:

  ```
  /vis/heprep/setFileName <my_file_name>
  ```

  which will produce files named <my_file_name>0.heprep, <my_file_name>1.heprep, etc.
- You can specify that each file should overwrite the previous file (always rewriting to the same file name) by using the setOverwrite command, as in:

  ```
  /vis/heprep/setOverwrite true
  ```

  This may be useful in some automated applications where you always want to see the latest output file in the same location.
- Geant4 visualization supports a concept called "culling", by which certain parts of the detector can be made invisible. Since you may want to control visibility from the HepRep browser, turning on visibility of detector parts that had defaulted to be invisible, the HepRepFile driver does not omit these invisible detector parts from the HepRep file. But for very large files, if you know that you will never want to make these parts visible, you can choose to have them left entirely out of the file. Use the /vis/heprep/setCullInvisibles command, as in:

  ```
  /vis/heprep/setCullInvisibles true
  ```

Further information:

- HepRApp Users Home Page: http://www.slac.stanford.edu/~perl/HepRApp/
- HepRep graphics format: http://www.slac.stanford.edu/~perl/heprep
- Geant4 Visualization Tutorial using the HepRApp HepRep Browser: http://geant4.slac.stanford.edu/Presentations/vis/G4HepRAppTutorial/G4HepRAppTutorial.html

8.3.7 HepRepXML

The HepRepXML driver creates a HepRep file in the HepRep2 format suitable for viewing with the WIRED4 Plugin to the JAS3 Analysis System or the FRED event display.

This driver can write both Binary HepRep (.bheprep) and XML HepRep (.heprep) files. Binary HepRep files are a one-to-one translation of XML HepRep files, but they are considerably shorter and faster to parse by a HepRepViewer such as WIRED 4.

Both Binary HepRep and XML HepRep can be compressed using the standard zlib library if linked into Geant4 using G4LIB_USE_ZLIB. If a standard zlib is not available (WIN32-VC for instance) you should also set G4LIB_BUILD_ZLIB to build G4zlib included with Geant4.

HepRep files (Binary and XML) can contain multiple HepRep events/geometries. If the file contains more than one HepRep it is not strictly XML anymore. Files can be written in .heprep.zip, .heprep.gz or .heprep format and their binary versions .bheprep.zip, .bheprep.gz or .bheprep.

The .heprep.zip is the default for file output, the .heprep is the default for stdout and stderr.

(Optional) To set the filename with a particular extension such as: .heprep.zip, .heprep.gz, .heprep, .bheprep.zip, .bheprep.gz or .bheprep use for instance:

```
/vis/scene/create filename.bheprep.zip
```

(Optional) To create separate files for each event, you can set a suffix such as "-0001" to start writing files from filename-0001.bheprep.zip to filename-9999.bheprep.zip (or up), while "-55-sub" will start write files filename-55-sub.bheprep.zip to filename-99-sub.bheprep.zip (or up).

```
/vis/heprep/setEventNumberSuffix -0001
```

(Note: suffix has to contain at least one digit)

(Optional) To route the HepRep XML output to stdout (or stderr), by default uncompressed, use:

```
/vis/scene/create stdout
```

(Optional) To add attributes to each point on a trajectory, use:

```
/vis/heprep/addPointAttributes 1
```

Be aware that this may increase the size of the output dramatically.

(Optional) You may use the commands:

```
/vis/viewer/zoom                          to set an initial zoom factor

/vis/viewer/set/viewpointThetaPhi         to set an initial view point

/vis/heprep/setCoordinateSystem uvw       to change the coordinate system, where uvw
                                                can be "xyz", "zxy", ...
```

(Optional) You may decide to write .zip files with events and geometry separated (but linked). This results in a smaller zip file, as the geometry is only written once. Use the command:

```
/vis/heprep/appendGeometry false
```

(Optional) To close the file, remove the SceneHandler, use:

```
/vis/sceneHandler/remove scene-handler-0
```

Limitations: Only one SceneHandler can exist at any time, connected to a single Viewer. Since the HepRep format is a model rather than a view this is not a real limitation. In WIRED 4 you can create as many views (SceneHandlers) as you like.

Further information:

- WIRED4 Plugin to the JAS3 Analysis System
- FRED event display
- HepRep graphics format

8.3.8 DAWN

The DAWN drivers are interfaces to Fukui Renderer DAWN, which has been developed by Satoshi Tanaka, Minato Kawaguti et al (Fukui University). It is a vectorized 3D PostScript processor, and so well suited to prepare technical high quality outputs for presentation and/or documentation. It is also useful for precise debugging of detector geometry. Remote visualization, off-line re-visualization, cut view, and many other useful functions of detector simulation are supported. A DAWN process is automatically invoked as a co-process of Geant4 when visualization is performed, and 3D data are passed with inter-process communication, via a file, or the TCP/IP socket.

When Geant4 Visualization is performed with the DAWN driver, the visualized view is automatically saved to a file named `g4.eps` in the current directory, which describes a vectorized (Encapsulated) PostScript data of the view.

There are two kinds of DAWN drivers, the DAWNFILE driver and the DAWN-Network driver. The DAWNFILE driver is usually recommended, since it is faster and safer in the sense that it is not affected by network conditions.

The DAWNFILE driver sends 3D data to DAWN via an intermediate file, named `g4.prim` in the current directory. The file `g4.prim` can be re-visualized later without the help of Geant4. This is done by invoking DAWN by hand:

```
% dawn g4.prim
```

DAWN files can also serve as input to two additional programs:

- A standalone program, DAWNCUT, can perform a planar cut on a DAWN image. DAWNCUT takes as input a .prim file and some cut parameters. Its output is a new .prim file to which the cut has been applied.
- Another standalone program, DAVID, can show you any volume overlap errors in your geometry. DAVID takes as input a .prim file and outputs a new .prim file in which overlapping volumes have been highlighted. The use of DAVID is described in section *Detecting Overlapping Volumes* of this manual.

The DAWN-Network driver is almost the same as the DAWNFILE driver except that

- 3D data are passed to DAWN via the TCP/IP the socket (default) or the named pipe, and that,

If you have not set up network configurations of your host machine, set the environment variable `G4DAWN_NAMED_PIPE` to "1", e.g., `% setenv G4DAWN_NAMED_PIPE 1`. This setting switches the default socket connection to the named-pipe connection within the same host machine. The DAWN-Network driver also saves the 3D data to the file `g4.prim` in the current directory.

8.3.9 Remote Visualization with the DAWN-Network Driver

Visualization in Geant4 is considered to be "remote" when it is performed on a machine other than the Geant4 host. Some of the visualization drivers support this feature.

Usually, the visualization host is your local host, while the Geant4 host is a remote host where you log in, for example, with the `telnet` command. This enables distributed processing of Geant4 visualization, avoiding the transfer of large amounts of visualization data to your terminal display via the network. This section describes how to perform remote Geant4 visualization with the DAWN-Network driver. In order to do it, you must install the Fukui Renderer DAWN on your local host beforehand.

The following steps realize remote Geant4 visualization viewed by DAWN.

1. Invoke DAWN with "-G" option on your local host:

   ```
   Local_Host> dawn -G
   ```

   This invokes DAWN with the network connection mode.
2. Login to the remote host where a Geant4 executable is placed.
3. Set an environment variable on the remote host as follows:

```
Remote_Host> setenv G4DAWN_HOST_NAME local_host_name
```

For example, if you are working in the local host named "arkoop.kek.jp", set this environment variable as follows:

```
Remote_Host> setenv G4DAWN_HOST_NAME arkoop.kek.jp
```

This tells a Geant4 process running on the remote host where Geant4 Visualization should be performed, i.e., where the visualized views should be displayed.

4. Invoke a Geant4 process and perform visualization with the DAWN-Network driver. For example:

```
Idle> /vis/open DAWN
Idle> /vis/drawVolume
Idle> /vis/viewer/flush
```

In step 4, 3D scene data are sent from the remote host to the local host as DAWN-formatted data, and the local DAWN will visualize the data. The transferred data are saved as a file named `g4.prim` in the current directory of the local host.

Further information:

- Fukui Renderer DAWN: http://geant4.kek.jp/GEANT4/vis/DAWN/About_DAWN.html
- The DAWNFILE driver: http://geant4.kek.jp/GEANT4/vis/GEANT4/DAWNFILE_driver.html
- The DAWN-Network driver: http://geant4.kek.jp/GEANT4/vis/GEANT4/DAWNNET_driver.html
- Environmental variables to customize DAWN and DAWN drivers: http://geant4.kek.jp/GEANT4/vis/DAWN/DAWN_ENV.html , http://geant4.kek.jp/GEANT4/vis/GEANT4/g4vis_on_linux.html
- DAWN format (g4.prim format) manual: http://geant4.kek.jp/GEANT4/vis/DAWN/G4PRIM_FORMAT_24/
- Geant4 Fukui University Group Home Page: http://geant4.kek.jp/GEANT4/vis/
- DAWNCUT: http://geant4.kek.jp/GEANT4/vis/DAWN/About_DAWNCUT.html
- DAVID: http://geant4.kek.jp/GEANT4/vis/DAWN/About_DAVID.html
- Geant4 Visualization Tutorial using the DAWN Renderer: http://geant4.slac.stanford.edu/Presentations/vis/GDAWNTutorial/G4DAWNTutorial.html

8.3.10 VRML

These drivers were developed by Satoshi Tanaka and Yasuhide Sawada (Fukui University). They generate VRML files, which describe 3D scenes to be visualized with a proper VRML viewer, at either a local or a remote host. It realizes virtual-reality visualization with your WWW browser. There are many excellent VRML viewers, which enable one to perform interactive spinning of detectors, walking and/or flying inside detectors or particle showers, interactive investigation of detailed detector geometry etc.

There are two kinds of VRML drivers: the VRMLFILE driver, and the VRML-Network driver. The VRMLFILE driver is usually recommended, since it is faster and safer in the sense that it is not affected by network conditions.

The VRMLFILE driver sends 3D data to your VRML viewer, which is running on the same host machine as Geant4, via an intermediate file named `g4.wrl` created in the current directory. This file can be re-visualization afterwards. In visualization, the name of the VRML viewer should be specified by setting the environment variable `G4VRML_VIEWER` beforehand. For example,

```
% setenv G4VRML_VIEWER  "netscape"
```

Its default value is `NONE`, which means that no viewer is invoked and only the file `g4.wrl` is generated.

Remote Visualization with the VRML-Network Driver

Visualization in Geant4 is considered to be "remote" when it is performed on a machine other than the Geant4 host. Some of the visualization drivers support this feature.

Usually, the visualization host is your local host, while the Geant4 host is a remote host where you log in, for example, with the `telnet` command. This enables distributed processing of Geant4 visualization, avoiding the transfer of large amounts of visualization data to your terminal display via the network.

In order to perform remote visualization with the VRML-Network driver, the following must be installed on your local host beforehand:

1. a VRML viewer
2. the Java application `g4vrmlview`.

The Java application `g4vrmlview` is included as part of the Geant4 package and is located at:

```
source/visualization/VRML/g4vrmlview/
```

Installation instructions for `g4vrmlview` can be found in the `README` file there, or on the WWW page below.

The following steps realize remote Geant4 visualization displayed with your local VRML browser:

1. Invoke the `g4vrmlview` on your local host, giving a VRML viewer name as its argument:

   ```
   Local_Host> java g4vrmlview  VRML_viewer_name
   ```

   For example, if you want to use the Netscape browser as your VRML viewer, execute `g4vrmlview` as follows:

   ```
   Local_Host> java g4vrmlview  netscape
   ```

   Of course, the command path to the VRML viewer should be properly set.
2. Log in to the remote host where a Geant4 executable is placed.
3. Set an environment variable on the remote host as follows:

   ```
   Remote_Host> setenv G4VRML_HOST_NAME local_host_name
   ```

   For example, if you are working on the local host named "arkoop.kek.jp", set this environment variable as follows:

   ```
   Remote_Host> setenv G4VRML_HOST_NAME arkoop.kek.jp
   ```

   This tells a Geant4 process running on the remote host where Geant4 Visualization should be performed, i.e., where the visualized views should be displayed.
4. Invoke a Geant4 process and perform visualization with the VRML-Network driver. For example:

   ```
   Idle> /vis/open VRML2
   Idle> /vis/drawVolume
   Idle> /vis/viewer/update
   ```

In step 4, 3D scene data are sent from the remote host to the local host as VRML-formatted data, and the VRML viewer specified in step 3 is invoked by the `g4vrmlview` process to visualize the VRML data. The transferred VRML data are saved as a file named `g4.wrl` in the current directory of the local host.

Further information:

- http://geant4.kek.jp/GEANT4/vis/GEANT4/VRML_net_driver.html

Further information (VRML drivers):

- http://geant4.kek.jp/GEANT4/vis/GEANT4/VRML_file_driver.html
- http://geant4.kek.jp/GEANT4/vis/GEANT4/VRML_net_driver.html

Sample VRML files:

- http://geant4.kek.jp/GEANT4/vis/GEANT4/VRML2_FIG/

Further information (VRML language and browsers):

- http://www.vrmlsite.com/

8.3.11 RayTracer

This driver was developed by Makoto Asai and Minamimoto (Hirosihma Instutute of Technology). It performs ray-tracing visualization using the tracking routines of Geant4. It is, therefore, available for every kinds of shapes/solids which Geant4 can handle. It is also utilized for debugging the user's geometry for the tracking routines of Geant4. It is well suited for photo-realistic high quality output for presentation, and for intuitive debugging of detector geometry. It produces a JPEG file. This driver is by default listed in the available visualization drivers of user's application.

Some pieces of geometries may fail to show up in other visualization drivers (due to algorithms those drivers use to compute visualizable shapes and polygons), but RayTracer can handle any geometry that the Geant4 navigator can handle.

Because RayTracer in essence takes over Geant4's tracking routines for its own use, RayTracer cannot be used to visualize Trajectories or hits.

An X-Window version, called RayTracerX, can be selected by setting `GEANT4_USE_RAYTRACER_X11` (for CMake) at Geant4 library build time and application (user code) build time (assuming you use the standard visualization manager, `G4VisExecutive`, or an equally smart vis manager). RayTracerX builds the same jpeg file as RayTracer, but simultaneously renders to screen so you can watch as rendering grows progressively smoother.

RayTracer has its own built-in commands - `/vis/rayTracer/...`. Alternatively, you can treat it as a normal vis system and use `/vis/viewer/...` commands, e.g:

```
/vis/open RayTracerX
/vis/drawVolume
/vis/viewer/set/viewpointThetaPhi 30 30
/vis/viewer/refresh
```

The view parameters are translated into the necessary RayTracer parameters.

RayTracer is compute intensive. If you are unsure of a good viewing angle or zoom factor, you might be advised to choose them with a faster renderer, such as OpenGL, and transfer the view parameters with `/vis/viewer/copyViewFrom`:

```
/vis/open OGL
/vis/drawVolume
/vis/viewer/zoom  # plus any /vis/viewer/commands that get you the view you want.
/vis/open RayTracerX
/vis/viewer/copyViewFrom viewer-0
/vis/viewer/refresh
```

8.3.12 gMocren

The gMocrenFile driver creates a gdd file suitable for viewing with the gMocren volume visualizer. gMocren, a sophisticated tool for rendering volume data, can show volume data such as Geant4 dose distrubutions overlaid with scoring grids, trajectories and detector geometry. gMocren provides additional advanced functionality such as transfer functions, colormap editing, image rotation, image scaling, and image clipping.

gMocren is further described at http://geant4.kek.jp/gMocren/. At this link you will find the gMocren download, the user manual, a tutorial and some example gdd data files.

Please note that the gMocren file driver is currently considered a Beta release. Users are encouraged to try this driver, and feedback is welcome, but users should be aware that features of this driver may change in upcoming releases.

To send volume data from Geant4 scoring to a gMocren file, the user needs to tell the gMocren driver the name of the specific scoring volume that is to be displayed. For scoring done in C++, this is the name of the sensitive volume. For command-based scoring, this is the name of the scoring mesh.

```
/vis/gMocren/setVolumeName <volume_name>
```

The following is an example of the minimum command sequence to send command-based scoring data to the a gMocren file:

```
# an example of a command-based scoring definition
/score/create/boxMesh scoringMesh       # name of the scoring mesh
/score/mesh/boxSize 10. 10. 10. cm      # dimension of the scoring mesh
/score/mesh/nBin 10 10 10               # number of divisions of the scoring mesh
/score/quantity/energyDeposit eDep      # quantity to be scored
/score/close
# configuration of the gMocren-file driver
/vis/scene/create
/vis/open gMocrenFile
/vis/gMocren/setVolumeName scoringMesh
```

To add detector geometry to this file:

```
/vis/viewer/flush
```

To add trajectories and primitive scorer hits to this file:

```
/vis/scene/add/trajectories
/vis/scene/add/pshits
/run/beamOn 1
```

gMocrenFile will write a file named G4_00.gd to the current directory. Subsequent draws will create files named g4_01.gdd, g4_02.gdd, etc. An alternate output directory can be specified with an environment variable:

```
export G4GMocrenFile_DEST_DIR=<someOtherDir/someOtherSubDir/>
```

View the resuling gMocren files with the gMocren viewer, available from: http://geant4.kek.jp/gMocren/.

8.3.13 Wt

> **Warning:** this driver is experimental and should be used with caution.

This driver has been developed by Laurent Garnier (IN2P3, LAL Orsay). It provide an interface to a geant4 application inside a Web browser. This driver also requires the Wt library and a Web browser with WebGL enable. See if your Web browser support WebGL on Wikipedia#WebGL#Support

The Wt driver is well suited for real-time fast visualization and demonstration. Available as experimental in Geant4.10 version, all OpenGL features are not implemented but basics interactions as mouse control of rotation/translation/zoom are present.

Wt driver rely on WebGL, it aims to render the same way as Qt, but inside a Web browser. The use of WebGL (instead of OpenGL for Qt), allow it to be available wherever a Web browser with WebGL is activate.

Sources files:

See CMake configuration in order to compile Geant4 with Wt support.

As a Geant4 with Wt driver application will be available inside a Web browser, your need at first to launch a web server in order to be able to see the web page. Hopefully, Wt came with its own web server included. This web server will be multi-user, that means that you could have many users using your application from everywhere. As the support for Wt driver is experimental, the multi-user aspect is not well manage. In Geant4.10, many users will have access at the same Run manager at the same time and evn to the files and datas, this could cause some troubles.

As a Geant4 application using Wt driver is a client/server application, the way to build the main function is a bit different.

Listing 8.4: The typical `main()` routine available for visualization with Wt driver.

```
//----- C++ source codes: main() function for visualization
#ifdef G4VIS_USE
#include "G4VisExecutive.hh"
#endif

 // Wt includes
#if defined(G4UI_USE_WT)
#include <Wt/WApplication>
#include <Wt/WEnvironment>
#include <Wt/WServer>


// Main Wt driver function. It will be call once by user launching the application
// Inside this function, you have to put all your Geant4 initialisation
// (as in main() function on other graphic drivers)
Wt::WApplication *createApplication(const Wt::WEnvironment& env)
{

 // Create a new instance of Wt::Application
 Wt::WApplication* myApp = new Wt::WApplication(env);

 // Set title and styleSheet
 wApp->setTitle( "Geant4 on the web" );
 wApp->useStyleSheet("extkitchen.css");


 // Get the pointer to the User Interface manager
 G4UImanager* UImanager = G4UImanager::GetUIpointer();

 char* name = "ExampleN03 \0";
 G4UIExecutive* ui = new G4UIExecutive(1,&name, "Wt");

 // Start the session
 ui->SessionStart();
 delete ui;

 return myApp;
}
#endif


int main(int argc,char** argv) {

.....

  // Instantiation and initialization of the Visualization Manager
#ifdef G4VIS_USE
  // visualization manager
  G4VisManager* visManager = new G4VisExecutive;
  // G4VisExecutive can take a verbosity argument - see /vis/verbose guidance.
  // G4VisManager* visManager = new G4VisExecutive("Quiet");
  visManager->Initialize();
#endif

.....

// replace the "normal" user interface by the Web server
#ifndef G4UI_USE_WT
 // Get the pointer to the User Interface manager
 G4UImanager* UImanager = G4UImanager::GetUIpointer();
#else
 try {
```

```
   // Create a Wt::WServer
   Wt::WServer server(argv[0]);

   server.setServerConfiguration(argc, argv, WTHTTP_CONFIGURATION);

   server.addEntryPoint(Wt::Application, createApplication);

   // Run it !
   if (server.start()) {
     int sig = Wt::WServer::waitForShutdown();

     server.stop();
   }
 } catch (Wt::WServer::Exception& e) {
   std::cerr << e.what() << "\n";
   return 1;
 } catch (std::exception& e) {
   std::cerr << "exception: " << e.what() << "\n";
   return 1;
 }

 // Wait for clients
 Wt::WRun(argc, argv, &createApplication);

 // Job termination
 #ifdef G4VIS_USE
  delete visManager;
#endif

.....
#endif

 return 0;
}

//----- end of C++
```

This driver will display the UI and vis window inside a Web browser page. As with Qt driver, you can have as many tabs with viewer you want. To see the visualization window:

```
/vis/open OGL
```

other parameters as OGLI, OGLS, OGLIWt, OGLSWt will have all the same effect

Execution of the server: As your application will contain a web server, you will have to launch the web server first and set some specific arguments for internet :

- docroot: document root for static files as css, images...
- http-address: The address where this application will be deploy. (eg:0.0.0.0)
- http-port: HTTP port (e.g. 80)

More informations on Wt web site The command line for launching your application will be the following:

```
myExample --docroot "where your ressources are" --http-address 0.0.0.0 --http-port 8080
```

Execution of a client: All clients can reach your application server at the following address:

```
http://0.0.0.0:8080 (for users on the same computer as the server)
http://Your.Server.Ip:8080 (for external users)
```

If this address is unreachable, check if the specify port is not already in use and is fully open.

Further information (Wt):

- Wt

8.3.14 Visualization of detector geometry tree

ASCIITREE is a visualization driver that is not actually graphical but that dumps the volume hierarchy as a simple text tree.

Each call to /vis/viewer/flush or /vis/drawTree will dump the tree.

ASCIITree has command to control its verbosity, `/vis/ASCIITree/verbose`. The verbosity value controls the amount of information available, e.g., physical volume name alone, or also logical volume and solid names. If the volume is "sensitive" and/or has a "readout geometry", this may also be indicated. Also, the mass of the physical volume tree(s) can be printed (but beware - higher verbosity levels can be computationally intensive).

At verbosity level 4, ASCIITree calculates the mass of the complete geometry tree taking into account daughters up to the depth specified for each physical volume. The calculation involves subtracting the mass of that part of the mother that is occupied by each daughter and then adding the mass of the daughter, and so on down the hierarchy.

```
/vis/ASCIITree/Verbose 4
/vis/viewer/flush
"HadCalorimeterPhysical":0 / "HadCalorimeterLogical" / "HadCalorimeterBox"(G4Box),
                             1.8 m3 , 11.35 g/cm3
"HadCalColumnPhysical":-1 (10 replicas) / "HadCalColumnLogical" / "HadCalColumnBox"(G4Box),
                                          180000 cm3, 11.35 g/cm3
"HadCalCellPhysical":-1 (2 replicas) / "HadCalCellLogical" / "HadCalCellBox"(G4Box),
                                       90000 cm3, 11.35 g/cm3
"HadCalLayerPhysical":-1 (20 replicas) / "HadCalLayerLogical" / "HadCalLayerBox"(G4Box),
                                         4500 cm3, 11.35 g/cm3
"HadCalScintiPhysical":0 / "HadCalScintiLogical" / "HadCalScintiBox"(G4Box),
                           900 cm3, 1.032 g/cm3

Calculating mass(es)...
Overall volume of "worldPhysical":0, is 2400 m3
Mass of tree to unlimited depth is 22260.5 kg
```

Some more examples of ASCIITree in action:

```
Idle> /vis/ASCIITree/verbose 1
Idle> /vis/drawTree
#  Set verbosity with "/vis/ASCIITree/verbose "
#    <  10: - does not print daughters of repeated placements, does not repeat replicas.
#    >= 10: prints all physical volumes.
#  The level of detail is given by verbosity%10:
#  for each volume:
#    >=  0: physical volume name.
#    >=  1: logical volume name (and names of sensitive detector and readout geometry, if any).
#    >=  2: solid name and type.
#    >=  3: volume and density.
#    >=  5: daughter-subtracted volume and mass.
#  and in the summary at the end of printing:
#    >=  4: daughter-included mass of top physical volume(s) in scene to depth specified.
.....
"Calorimeter", copy no. 0, belongs to logical volume "Calorimeter"
  "Layer", copy no. -1, belongs to logical volume "Layer" (10 replicas)
    "Absorber", copy no. 0, belongs to logical volume "Absorber"
      "Gap", copy no. 0, belongs to logical volume "Gap"
.....
Idle> /vis/ASCIITree/verbose 15
Idle> /vis/drawTree
....
 "tube_phys":0 / "tube_L" / "tube"(G4Tubs), 395841 cm3, 1.782 mg/cm3,
                            9.6539e-08 mm3, 1.72032e-10 mg
   "divided_tube_phys":0 / "divided_tube_L" / "divided_tube"(G4Tubs), 65973.4 cm3,
                           1.782 mg/cm3, 7587.54 cm3, 13.521 g
     "divided_tube_inset_phys":0 / "divided_tube_inset_L" / "divided_tube_inset"(G4Tubs),
                                   58385.9 cm3, 1.782 mg/cm3, 6.03369e-09 mm3, 1.0752e-11 mg
       "sub_divided_tube_phys":0 / "sub_divided_tube_L" / "sub_divided_tube"(G4Tubs),
```

```
                                  14596.5 cm3, 1.782 mg/cm3, 12196.5 cm3, 21.7341 g
.....
Calculating mass(es)...
Overall volume of "expHall_P":0, is 8000 m3  and the daughter-included mass to unlimited depth
                                 is 78414 kg
.....
```

For the complete list of commands and options, see the Control... UICommands section of this user guide.

8.3.15 GAG Tree

The GAGTree driver provides a listing of the detector geometry tree within GAG, the Geant Adaptive GUI, from the `environments/MOMO/MOMO.jar` file present under the Geant4 source distribution. GAG allows "folding/unfolding" a part of the geometry tree, using the *Tree Widget* in Java:

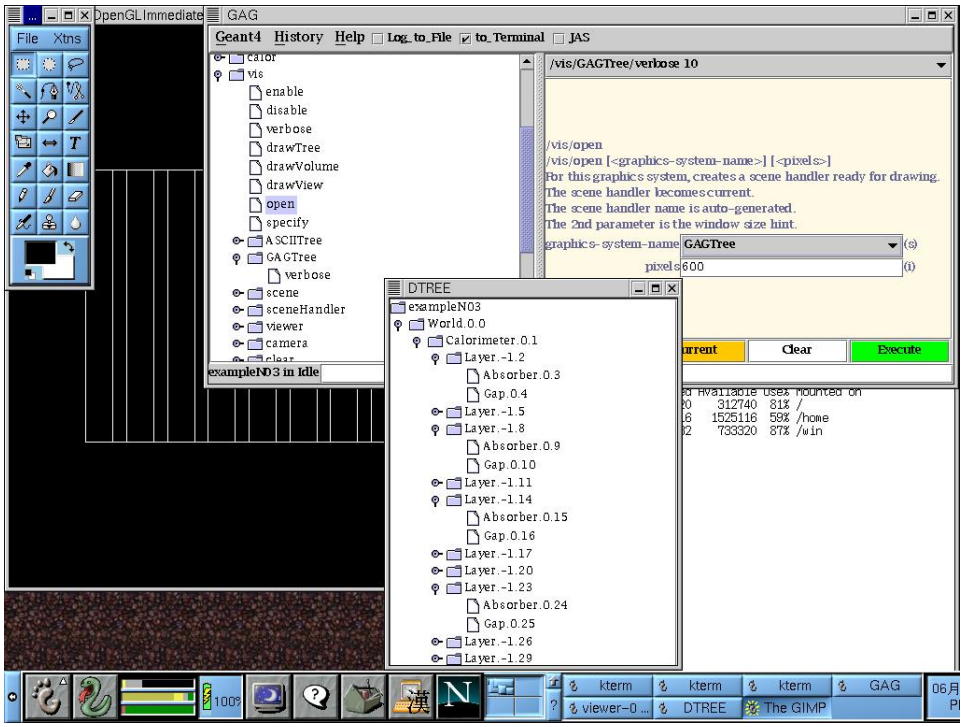


Fig. 8.1: GAG tree folding/unfolding.

8.3.16 XML Tree

The XML description of the geometry tree can be created in Geant4 by the XML Tree driver. The XML source can also be edited on the fly. The created XML files are visualizable with any XML browser (in Windows, a good XML viewer is *XML Notepad*).

- Folding and un-folding (*Figure*):
- Searching a string (*Figure*):

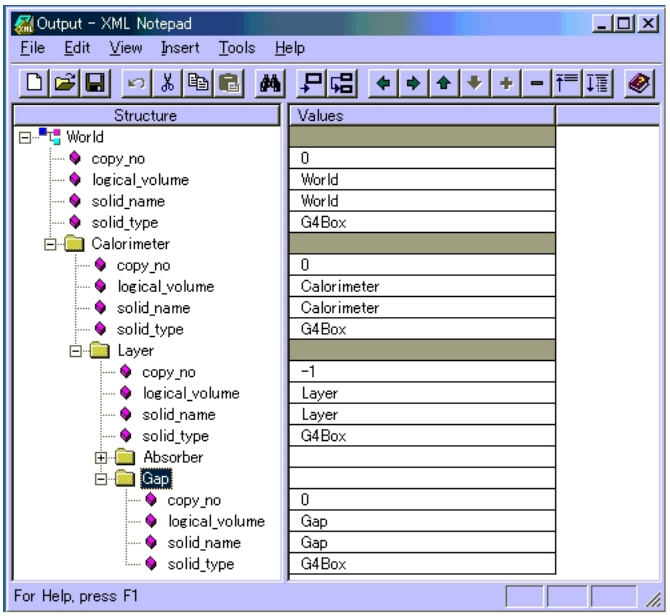


Fig. 8.2: XML Tree folding and un-folding.

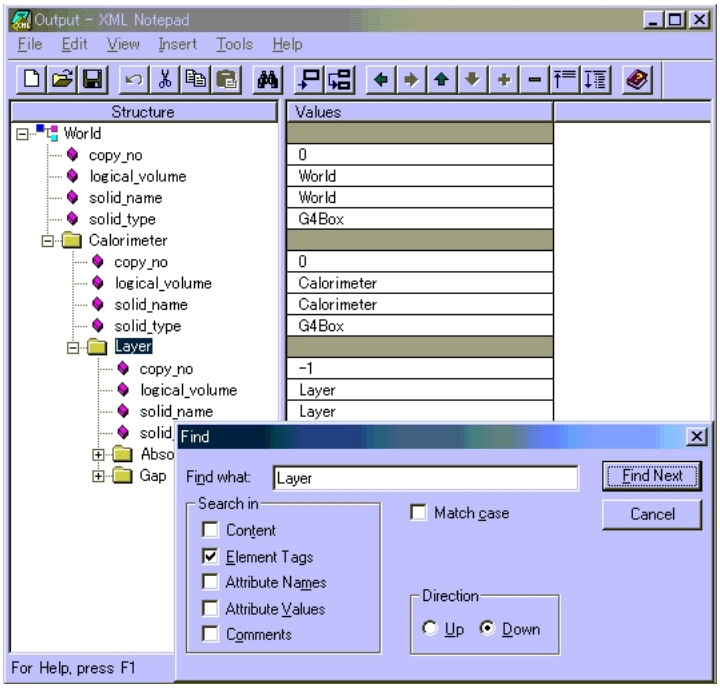


Fig. 8.3: XML Tree searching for a string.

8.4 Controlling Visualization from Commands

This section describes just a few of the more commonly used visualization commands. For the complete list of commands and options, see the Control... UICommands section of this user guide.

For simplicity, this section assumes that the Geant4 executable was compiled incorporating the DAWNFILE and the OpenGL-Xlib drivers. For details on creating an executable for visualization see *Adding Visualization to Your Executable*.

Warning: THIS SECTION IS NOT A COMPLETE DESCRIPTION OF ALL VISUALISATION COMMANDS; THEY ARE TOO NUMEROUS AND CONTINUALLY EVOLVING. PLEASE REFER TO THE COMMAND GUIDANCE, Control... UICommands OR SIMPLY TYPE "ls vis" OR "help". SOME VIEWERS, NOTABLE Qt, OFFER INTERACTIVE GUIDANCE UNDER THE "Help" MENU."

8.4.1 Scene, scene handler, and viewer

In using the visualization commands, it is useful to know the concept of "scene", "scene handler", and "viewer". A "scene" is a set of visualizable raw 3D data. A "scene handler" is a graphics-data modeler, which processes raw data in a scene for later visualization. And a "viewer" generates images based on data processed by a scene handler. Roughly speaking, a set of a scene handler and a viewer corresponds to a visualization driver.

The steps of performing Geant4 visualization are explained below, though some of these steps may be done for you so that in practice you may use as few as just two commands (such as /vis/open OGLIX plus /vis/drawVolume). The seven steps of visualization are:

Step		Command	Alternative command
1	Create a scene handler and a viewer	/vis/sceneHandler/create /vis/viewer/create	/vis/open
2	Create an empty scene	/vis/scene/create	/vis/drawVolume
3	Add raw 3D data to the created scene	/vis/scene/add/volume	
4	Attach the current scene to the current scene handler	/vis/sceneHandler/attach	
5	Set camera parameters, drawing style (wireframe/surface), etc	E.g., /vis/viewer/set/viewpoint	
6	Make the viewer execute visualization	/vis/viewer/refresh	
7	Declare the end of visualization for flushing	/vis/viewer/flush	

For details about the commands, see below.

These seven steps can be controlled explicitly to create multiple scenes and multiple viewers, each with its own set of parameters, with easy switching from one scene to another. But for the most common case of just having one scene and one viewer, many steps are handled implicitly for you.

8.4.2 Create a scene handler and a viewer: `/vis/open` command

Command "`/vis/open`" creates a scene handler and a viewer, which corresponds to Step 1.

Command: `/vis/open [driver_tag_name]`

- **Argument**
  A name of (a mode of) an available visualization driver.

- **Action**
  Create a visualization driver, i.e. a set of a scene hander and a viewer.
- **Example: Create an OpenGL generic driver with its immediate mode**
  `Idle> /vis/open OGLI`
- **Additional notes**
  For immediate viewers, such as OGLI, your geometry will immediately be rendered in the new GL window

How to list available driver_tag_name:

```
Idle> help /vis/open
```

or:

```
Idle> help /vis/sceneHandler/create
```

The list is, for example, displayed as follows:

```
.....
Candidates : DAWNFILE OGL
.....
```

For additional options, see the Control...UICommands section of this user guide.

8.4.3 Create an empty scene: `/vis/scene/create` command

Command "`/vis/scene/create`" creates an empty scene, which corresponds to Step 2.

Command: `/vis/scene/create [scene_name]`

- **Argument**
  A name for this scene. Created for you if you don't specify one.

8.4.4 Visualization of a physical volume: `/vis/drawVolume` command

Command "`/vis/drawVolume`" adds a physical volume to the scene. It also does some of the other steps, if you haven't done them explicitly. It takes care of steps 2, 3, 4 and 6. Command "`/vis/viewer/flush`" should follow in order to do the final Step 7.

Commands:

```
/vis/drawVolume [physical-volume-name]
.....
Idle> /vis/viewer/flush
```

- **Argument**
  A physical-volume name. The default value is "world", which is omittable.
- **Action**
  Creates a scene consisting of the given physical volume and asks the current viewer to draw it. The scene becomes current. Command "`/vis/viewer/flush`" should follow this command in order to declare end of visualization.
- **Example: Visualization of the whole world with coordinate axes**

```
Idle> /vis/drawVolume
Idle> /vis/scene/add/axes 0 0 0 500 mm
Idle> /vis/viewer/flush
```

8.4.5 Visualization of a logical volume: `/vis/specify` command

Command "`/vis/specify`" visualizes a logical volume. If allows you to control how much details is shown and whether to show booleans, voxels and readout geometries. It also does some of the other steps, if you haven't done them explicitly. It takes care of steps 2, 3, 4 and 6. Command "`/vis/viewer/flush`" should follow the command in order to do the final Step 7.

Command: `/vis/specify [logical-volume-name][depth-of-descent] [booleans-flag] [voxels-flag] [readout-flag]`

- **Argument**
  A logical-volume name.
- **Action**
  Creates a scene consisting of the given logical volume and asks the current viewer to draw it. The scene becomes current.
- **Example (visualization of a selected logical volume with coordinate axes)**

```
Idle> /vis/specify Absorber
Idle> /vis/scene/add/axes 0 0 0 500 mm
Idle> /vis/scene/add/text 0 0 0 mm 40 -100 -200 LogVol:Absorber
Idle> /vis/viewer/flush
```

For more options, see the Control...UICommands section of this user guide.

8.4.6 Visualization of trajectories: `/vis/scene/add/trajectories` command

Command "`/vis/scene/add/trajectories [smooth] [rich]`" adds trajectories to the current scene. The optional parameters "smooth" and/or "rich" (you may specify either, both or neither) invoke, if "smooth" is specified, the storing and displaying of extra points on curved trajectories and, if "rich" is specified, the storing, for possible subsequent selection and display, of additional information, such as volume names, creator process, energy deposited, global time. Be aware, of course, that this imposes computational and memory overheads. Note that this automatically issues the appropriate "`/tracking/storeTrajectory`" command so that trajectories are stored (by default they are not). The visualization is performed with the command "`/run/beamOn`" unless you have non-default values for /vis/scene/endOfEventAction or /vis/scene/endOfRunAction (described below).

Command: `/vis/scene/add/trajectories [smooth] [rich]`

- **Action**
  The command adds trajectories to the current scene. Trajectories are drawn at end of event when the scene in which they are added is current.
- **Example: Visualization of trajectories**

```
Idle> /vis/scene/add/trajectories
Idle> /run/beamOn 10
```

- **Additional note 1**
  See the section *Controlling from Commands* Enhanced Trajectory Drawing for details on how to control how trajectories are color-coded.
- **Additional note 2**
  Events may be kept and reviewed at end of run with:

```
Idle> /vis/reviewKeptEvents
```

  Keep all events with:

```
Idle> /vis/scene/endOfEventAction accumulate [maxNumber]
```

  (see *End of Event and Run*)
  or keep some chosen subset with:

```
G4EventManager::GetEventManager()->KeepTheCurrentEvent();
```

as described in *G4UserRunAction example*.
To suppress drawing during a run

```
Idle> /vis/disable
Idle> /run/beamOn 10000
```

then at end of run

```
Idle> /vis/enable
Idle> /vis/reviewKeptEvents
```

- **Additional note 3**
  Visualising events as they are being generated inevitably slows the simulation. Visualisation can be suspended with `/vis/disable` as suggested above. You may also switch off trajectory production with `/tracking/storeTrajectory 0`. When using OpenGL, the following can help:

```
Idle> /vis/ogl/flushAt [ endOfEvent endOfRun eachPrimitive NthPrimitive NthEvent never ]
```

  By default, this value is set to /vis/ogl/flushAt NthEvent 100

For more options, see the Control...UICommands section of this user guide.

8.4.7 Visualization of hits: `/vis/scene/add/hits` command

Command "`/vis/scene/add/hits`" adds hits to the current scene, assuming that you have a hit class and that the hits have visualization information. The visualization is performed with the command "`/run/beamOn`" unless you have non-default values for /vis/scene/endOfEventAction or /vis/scene/endOfRunAction (described above).

8.4.8 Visualization of Scored Data

Scored data can be visualized using the commands "`/score/drawProjection`" and "`/score/drawColumn`". For details, see examples/extended/runAndEvent/RE03.

8.4.9 HepRep Attributes for Hits

The HepRep file formats, HepRepFile and HepRepXML, attach various attributes to hits such that you can view these attributes, label trajectories by these attributes or make visibility cuts based on these attributes. Examples of adding HepRep attributes to hit classes can be found in examples /extended/analysis/A01 and /extended/runAndEvent/RE01.

For example, in example RE01's class RE01CalorimeterHit.cc, available attributes will be:

- Hit Type
- Track ID
- Z Cell ID
- Phi Cell ID
- Energy Deposited
- Energy Deposited by Track
- Position
- Logical Volume

You can add additional attributes of your choosing by modifying the relevant part of the hit class (look for the methods GetAttDefs and CreateAttValues).

8.4.10 Basic camera workings: `/vis/viewer/` commands

Commands in the command directory "`/vis/viewer/`" set camera parameters and drawing style of the current viewer, which corresponds to Step 5. Note that the camera parameters and the drawing style should be set separately for each viewer. They can be initialized to the default values with command "`/vis/viewer/reset`". Some visualization systems, such as the VRML and HepRep browsers also allow camera control from the standalone graphics application.

Just a few of the camera commands are described here. For more commands, see the Control...UICommands section of this user guide.

The view is defined by a target point (initially at the centre of the extent of all objects in the scene), an up-vector and a viewpoint direction - see *view parameters* figure. By default, the up-Vector is parallel to the y-axis and the viewpoint direction is parallel to the z-axis, so the the view shows the x-axis to the right and the y-axis upwards - a projection on to the canonical x-y plane - see *default parameters* figure.

The target point can be changed with a `/vis/viewer/set` command or with the `/vis/viewer/pan` commands. The up-vector and the viewpoint direction can also be changed with `/vis/viewer/set` commands. Care must be taken to avoid having the two vectors parallel, for in that case the view is undefined.

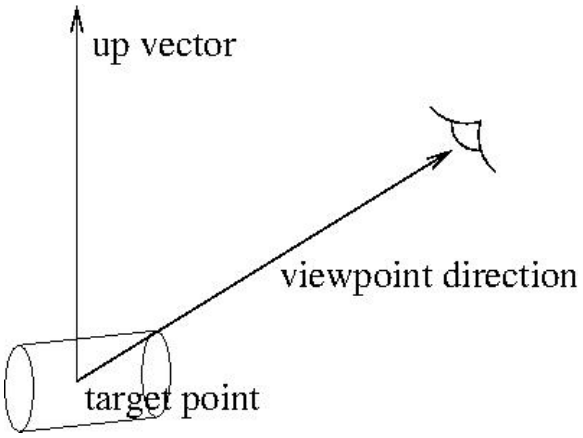


Fig. 8.4: Up-vector and viewpoint direction

Command: `/vis/viewer/set/viewpointThetaPhi [theta] [phi] [deg|rad]`

- **Arguments**
  Arguments "theta" and "phi" are polar and azimuthal camera angles, respectively. The default unit is "degree".
- **Action**
  Set a view point in direction of (theta, phi).
- **Example: Set the viewpoint in direction of (70 deg, 20 deg) /**

```
Idle> /vis/viewer/set/viewpointThetaPhi 70 20
```

- **Additional notes**
  Camera parameters should be set for each viewer. They are initialized with command "`/vis/viewer/reset`". Alternatively, they can be copied from another viewer with the command "`/vis/viewer/copyViewFrom viewer-0`", for example.

Command: `/vis/viewer/zoom [scale_factor]`

- **Argument**
  The scale factor. The command multiplies magnification of the view by this factor.
- **Action**
  Zoom up/down of view.

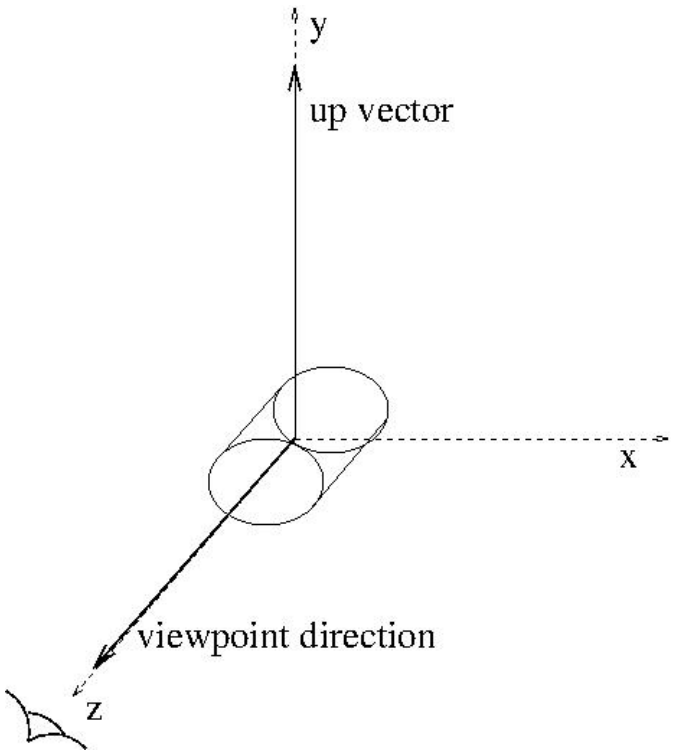


Fig. 8.5: The default view

- **Example: Zoom up by factor 1.5**

```
Idle> /vis/viewer/zoom 1.5
```

- **Additional notes**
  A similar pair of commands, scale and scaleTo allow non-uniform scaling (i.e., zoom differently along different axes). For details of this and lots of other commands, see the Control. . . UICommands section of this user guide. Some viewers have limits to how large the zoom factor can be. This problem can be circumnavigated to some degree by using `zoom` and `scale` together. If

```
Idle> /vis/viewer/zoomTo 1e10
```

  does not work, please try

```
Idle> /vis/viewer/scaleTo 1e5 1e5 1e5
Idle> /vis/viewer/zoomTo 1e5
```

  Of course, with such high zoom factors, you might want to know whither you are zooming. Use "`/vis/viewer/set/targetPoint`"
  Camera parameters should be set for each viewer. They are initialized with command "`/vis/viewer/reset`". Alternatively, they can be copied from another viewer with the command "`/vis/viewer/copyViewFrom viewer-0`", for example.

Command: `/vis/viewer/set/style [style_name]`

- **Arguments**
  Candidate values of the argument are "wireframe" and "surface". ("w" and "s" also work.)
- **Action**
  Set a drawing style to wireframe or surface.
- **Example: Set the drawing style to "surface"**

```
Idle> /vis/viewer/set/style surface
```

- **Additional notes**
  The style of some geometry components may have been forced one way or the other through calls in compiled code. The set/style command will NOT override such force styles.
  Drawing style should be set for each viewer. The drawing style is initialized with command "`/vis/viewer/reset`". Alternatively, it can be copied from another viewer with the command "`/vis/viewer/set/all viewer-0`", for example.

8.4.11 Declare the end of visualization for flushing: `/vis/viewer/flush` command

Command: `/vis/viewer/flush`

- **Action**
  Declare the end of visualization for flushing.
- **Additional notes**
  Command "`/vis/viewer/flush`" should follow "`/vis/drawVolume`", "`/vis/specify`", etc in order to complete visualization. It corresponds to Step 7.
  The flush is done automatically after every /run/beamOn command unless you have non-default values for /vis/scene/endOfEventAction or /vis/scene/endOfRunAction (described above).

8.4.12 End of Event Action and End of Run Action: `/vis/viewer/endOfEventAction` and `/vis/viewer/endOfRunAction` commands

By default, a separate picture is created for each event. You can change this behavior to accumulate multiple events, or even multiple runs, in a single picture.

Command: `/vis/scene/endOfEventAction [refresh|accumulate]`

- **Action**
  Control how often the picture should be cleared. `refresh` means each event will be written to a new picture. `accumulate` means events will be accumulated into a single picture. Picture will be flushed at end of run, unless you have also set `/vis/scene/endOfRunAction accumulate`
- **Additional note**
  You may instead choose to use update commands from your BeginOfRunAction or EndOfEventAction, as in early examples, but now the vis manager ia able to do most of what most users require through the above commands.

Command: `/vis/scene/endOfRunAction [refresh|accumulate]`

- **Action**
  Control how often the picture should be cleared. `refresh` means each run will be written to a new picture. `accumulate` means runs will be accumulated into a single picture. To start a new picture, you must explicitly issue `/vis/viewer/refresh`, `/vis/viewer/update` or `/vis/viewer/flush`

8.4.13 HepRep Attributes for Trajectories

The HepRep file formats, HepRepFile and HepRepXML, attach various attributes to trajectories such that you can view these attributes, label trajectories by these attributes or make visibility cuts based on these attributes. If you use the default Geant4 trajectory class from /tracking/src/G4Trajectory.cc (this is what you get with the plain `/vis/scene/add/trajectories` command), available attributes will be:

- Track ID

- Parent ID
- Particle Name
- Charge
- PDG Encoding
- Momentum 3-Vector
- Momentum magnitude
- Number of points

Using `/vis/scene/add/trajectories rich` will get you additional attributes. You may also add additional attributes of your choosing by modifying the relevant part of G4Trajectory (look for the methods GetAttDefs and CreateAttValues). If you are using your own trajectory class, you may want to consider copying these methods from G4Trajectory.

8.4.14 How to save a view.

```
/vis/viewer/save
```

This will save to a file that can be read in again with

```
/control/execute
```

If you save several views you may "fly through" them with

```
/vis/viewer/interpolate
```

See *Making a Movie*.

(Use the Geant4 "help" command to see details.)

8.4.15 How to save a view to an image file

Most of the visualization drivers offer ways to save visualized views to PostScript (PS) or Encapsulated PostScript (EPS). Some, in addition, offer Portable Document Format (PDF). OpenGL offers a big range of formats - see below.

- **DAWNFILE**
  The DAWNFILE driver, which co-works with Fukui Renderer DAWN, generates "vectorized" PostScript data with "analytical hidden-line/surface removal", and so it is well suited for technical high-quality outputs for presentation, documentation, and debugging geometry. In the default setting of the DAWNFILE drivers, EPS files named "`g4_00.eps, g4_01.eps, g4_02.eps,...`" are automatically generated in the current directory each time when visualization is performed, and then a PostScript viewer "`gv`"is automatically invoked to visualize the generated EPS files.
  For large data sets, it may take time to generate the vectorized PostScript data. In such a case, visualize the 3D scene with a faster visualization driver beforehand for previewing, and then use the DAWNFILE drivers. For example, the following visualizes the whole detector with the OpenGL-Xlib driver (immediate mode) first, and then with the DAWNFILE driver to generate an EPS file `g4_XX.eps` to save the visualized view:

```
# Invoke the OpenGL visualization driver in its immediate mode
/vis/open OGLIX

# Camera setting
/vis/viewer/set/viewpointThetaPhi 20 20

# Camera setting
/vis/drawVolume
/vis/viewer/flush

# Invoke the DAWNFILE visualization driver
```

```
/vis/open DAWNFILE

# Camera setting
/vis/viewer/set/viewpointThetaPhi 20 20

# Camera setting
/vis/drawVolume
/vis/viewer/flush
```

This is a good example to show that the visualization drivers are complementary to each other.

- **OpenInventor**
  In the OpenInventor drivers, you can simply click the "Print" button on their GUI to generate a PostScript file as a hard copy of a visualized view.
- **OpenGL**
  The OpenGL drivers can also generate image files, either from a pull-down menu (Motif and Qt drivers) or with `/vis/ogl/export`. Available formats are: eps ps pdf svg bmp cur dds icns ico jp2 jpeg jpg pbm pgm png ppm tif tiff wbmp webp xbm xpm. The default is pdf. It can generate either vector or bitmap PostScript data with `/vis/ogl/set/printMode` ("vectored" or "pixmap"). You can change the filename by `/vis/ogl/set/printFilename` And the print size by `/vis/ogl/set/printSize` In generating vectorized PostScript data, hidden-surface removal is performed based on the painter's algorithm after dividing facets of shapes into small sub-triangles.
  Note that a fundamental limitation of the gl2ps library used for this printing causes the `/vis/viewer/set/hiddenMarker` command to be ignored. Trajectories will always be fully drawn in the printEPS output even when the hiddenMarker hidden line removal option has been set to hide these trajectories in the corresponding OpenGL view.
  The `/vis/ogl/set/printSize` command can be used to print EPS files even larger than the current screen resolution. This can allow creation of very large images, suitable for creation of posters, etc. The only size limitation is the graphics card's viewport dimension: GL_MAX_VIEWPORT_DIMS

```
# Invoke the OpenGL visualization driver in its stored mode
/vis/open OGLSX

# Camera setting
/vis/viewer/set/viewpointThetaPhi 20 20

# Camera setting
/vis/drawVolume
/vis/viewer/flush

# set print mode to vectored
#/vis/ogl/set/printMode vectored

# set print size larger than screen
/vis/ogl/set/printSize 2000 2000

# print
/vis/ogl/export
```

- **HepRep**
  The HepRApp HepRep Browser and WIRED4 JAS Plug-In can generate a wide variety of bitmap and vector output formats including PostScript and PDF.

8.4.16 Culling

"Culling" means to skip visualizing parts of a 3D scene. Culling is useful for avoiding complexity of visualized views, keeping transparent features of the 3D scene, and for quick visualization.

Geant4 Visualization supports the following 3 kinds of culling:

- Culling of invisible physical volumes

- Culling of low density physical volumes.
- Culling of covered physical volumes by others

In order that one or all types of the above culling are on, i.e., activated, the global culling flag should also be on.

The *table* summarizes the default culling policies.

Culling Type	Default Value
global	ON
invisible	ON
low density	OFF
covered daughter	OFF

Table: The default culling policies.

The default threshold density of the low-density culling is 0.01 g/cm^3.

The default culling policies can be modified with the following visualization commands. (Below the argument `flag` takes a value of `true` or `false`.)

```
# global
/vis/viewer/set/culling  global  flag

# invisible
/vis/viewer/set/culling  invisible  flag

# low density
#   "value" is a proper value of a treshold density
#   "unit" is either g/cm3, mg/cm3 or kg/m3
/vis/viewer/set/culling  density  flag  value  unit

# covered daughter
/vis/viewer/set/culling  coveredDaughters  flag     density
```

The HepRepFile graphic system will, by default, include culled objects in the file so that they can still be made visible later from controls in the HepRep browser. If this behavior would cause files to be too large, you can instead choose to have culled objects be omitted from the HepRep file. See details in the HepRepFile Driver section of this user guide.

8.4.17 Cut view

Sectioning

"Sectioning" means to make a thin slice of a 3D scene around a given plane. At present, this function is supported by the OpenGL drivers. The sectioning is realized by setting a sectioning plane before performing visualization. The sectioning plane can be set by the command,

```
/vis/viewer/set/sectionPlane on x y z units nx ny nz
```

where the vector (x,y,z) defines a point on the sectioning plane, and the vector (nx,ny,nz) defines the normal vector of the sectioning plane. For example, the following sets a sectioning plane to a yz plane at x = 2 cm:

```
Idle> /vis/viewer/set/sectionPlane  on  2.0  0.0  0.0  cm  1.0  0.0  0.0
```

Cutting away

"Cutting away" means to remove a half space, defined with a plane, from a 3D scene.

- Cutting away is supported by the DAWNFILE driver "off-line". Do the following:
  - Perform visualization with the DAWNFILE driver to generate a file `g4.prim`, describing the whole 3D scene.

  - Make the application "DAWNCUT" read the generated file to make a view of cutting away.

  See the following WWW page for details: http://geant4.kek.jp/GEANT4/vis/DAWN/About_DAWNCUT.html
- Alternatively, add up to three cutaway planes:

```
/vis/viewer/addCutawayPlane 0 0 0 m 1 0 0
/vis/viewer/addCutawayPlane 0 0 0 m 0 1 0
...
```

  and, for more that one plane, you can change the mode to
  - "add" or, equivalently, "union" (default) or
  - "multiply" or, equivalently, "intersection":

```
/vis/viewer/set/cutawayMode multiply
```

  To de-activate:

```
/vis/viewer/clearCutawayPlanes
```

  OpenGL supports this feature.

8.4.18 Multithreading commands

Multithreading

Visualising events inevitably slows things down. With multithreading this effect is all the greater. See *Adding trajectories*, Additional Note 3, for some advice. If you wish to continue visualising, multithreading mode offers the following fine tuning.

Since Geant4 version 10.2, in multithreading mode, events generated by worker threads are put in a queue and extracted by a special visualisation thread. If the queue gets full, workers are suspended until the visualisation thread catches up. To mitigate or avoid this try using

```
/vis/multithreading/maxEventQueueSize <N>
/vis/multithreading/actionOnEventQueueFull <wait|discard>
```

(See command guidance for details.)

8.5 Controlling Visualization from Compiled Code

While a Geant4 simulation is running, visualization can be performed without user intervention. This is accomplished by calling methods of the Visualization Manager from methods of the user action classes (`G4UserRunAction` and `G4UserEventAction`, for example). In this section methods of the class `G4VVisManager`, which is part of the `graphics_reps` category, are described and examples of their use are given.

8.5.1 G4VVisManager

The Visualization Manager is implemented by classes `G4VisManager` and `G4VisExecutive`. See *Adding Visualization to Your Executable*. In order that your Geant4 be compilable either with or without the visualization category, you should not use these classes directly in your C++ source code, other than in the `main()` function. Instead, you should use their abstract base class `G4VVisManager`, defined in the `intercoms` category.

The pointer to the concrete instance of the real Visualization Manager can be obtained as follows:

```
//----- Getting a pointer to the concrete Visualization Manager instance
G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
```

The method G4VVisManager::GetConcreteInstance() returns NULL if Geant4 is not ready for visualization. Thus your C++ source code should be protected as follows:

```
//----- How to protect your C++ source codes in visualization
if (pVVisManager) {
    ....
    pVVisManager ->Draw (...);
    ....
}
```

8.5.2 Visualization of detector components

If you have already constructed detector components with logical volumes to which visualization attributes are properly assigned, you are almost ready for visualizing detector components. All you have to do is to describe proper visualization commands within your C++ codes, using the ApplyCommand() method.

For example, the following is sample C++ source codes to visualize the detector components:

```
//----- C++ source code: How to visualize detector components (2)
//                       ... using visualization commands in source codes

G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance() ;

if(pVVisManager)
{
    ... (camera setting etc) ...
    G4UImanager::GetUIpointer()->ApplyCommand("/vis/drawVolume");
    G4UImanager::GetUIpointer()->ApplyCommand("/vis/viewer/flush");
}

//-----  end of C++ source code
```

In the above, you should also describe /vis/open command somewhere in your C++ codes or execute the command from (G)UI at the executing stage.

8.5.3 Visualization of trajectories

In order to visualize trajectories, you can use the method void G4Trajectory::DrawTrajectory() defined in the tracking category. In the implementation of this method, the following drawing method of G4VVisManager is used:

```
//----- A drawing method of G4Polyline
virtual void G4VVisManager::Draw (const G4Polyline&, ...) ;
```

The real implementation of this method is described in the class G4VisManager.

At the end of one event, a set of trajectories can be stored as a list of G4Trajectory objects. Therefore you can visualize trajectories, for example, at the end of each event, by implementing the method MyEventAction::EndOfEventAction() as follows:

```
//----- C++ source codes
void ExN03EventAction::EndOfEventAction(const G4Event* evt)
{
  .....
  // extract the trajectories and draw them
  if (G4VVisManager::GetConcreteInstance())
    {
     G4TrajectoryContainer* trajectoryContainer = evt->GetTrajectoryContainer();
     G4int n_trajectories = 0;
     if (trajectoryContainer) n_trajectories = trajectoryContainer->entries();
```

```
    for (G4int i=0; i < n_trajectories; i++)
       { G4Trajectory* trj=(G4Trajectory*)((*(evt->GetTrajectoryContainer()))[i]);
         if (drawFlag == "all") trj->DrawTrajectory(50);
         else if ((drawFlag == "charged")&&(trj->GetCharge() != 0.))
                                trj->DrawTrajectory(50);
         else if ((drawFlag == "neutral")&&(trj->GetCharge() == 0.))
                                trj->DrawTrajectory(50);
       }
  }
}
//----- end of C++ source codes
```

8.5.4 Enhanced trajectory drawing

It is possible to use the enhanced trajectory drawing functionality in compiled code as well as from commands. Multiple trajectory models can be instantiated, configured and registered with G4VisManager. For details, see the section on *Controlling from Compiled Code*.

8.5.5 HepRep Attributes for Trajectories

The HepRep file formats, HepRepFile and HepRepXML, attach various attributes to trajectories such that you can view these attributes, label trajectories by these attributes or make visibility cuts based on these attributes. If you use the default Geant4 trajectory class, from /tracking/src/G4Trajectory.cc, available attributes will be:

- Track ID
- Parent ID
- Particle Name
- Charge
- PDG Encoding
- Momentum 3-Vector
- Momentum magnitude
- Number of points

You can add additional attributes of your choosing by modifying the relevant part of G4Trajectory (look for the methods GetAttDefs and CreateAttValues). If you are using your own trajectory class, you may want to consider copying these methods from G4Trajectory.

8.5.6 Visualization of hits

Hits are visualized with classes `G4Square` or `G4Circle`, or other user-defined classes inheriting the abstract base class `G4VMarker` (*Polylines, Markers and Text*). Drawing methods for hits are not supported by default. Instead, ways of their implementation are guided by virtual methods, `G4VHit::Draw()` and `G4VHitsCollection::DrawAllHits()`, of the abstract base classes `G4VHit` and `G4VHitsCollection`. These methods are defined as empty functions in the `digits+hits` category. You can overload these methods, using the following drawing methods of class `G4VVisManager`, in order to visualize hits:

```
//----- Drawing methods of G4Square and G4Circle
virtual void G4VVisManager::Draw (const G4Circle&, ...) ;
virtual void G4VVisManager::Draw (const G4Square&, ...) ;
```

The real implementations of these `Draw()` methods are described in class `G4VisManager`.

The overloaded implementation of `G4VHits::Draw()` will be held by, for example, class *MyTrackerHits* inheriting `G4VHit` as follows:

```
//----- C++ source codes: An example of giving concrete implementation of
//       G4VHit::Draw(), using  class MyTrackerHit : public G4VHit {...}
//
void MyTrackerHit::Draw()
{
   G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
   if(pVVisManager)
   {
     // define a circle in a 3D space
     G4Circle circle(pos);
     circle.SetScreenSize(0.3);
     circle.SetFillStyle(G4Circle::filled);

     // make the circle red
     G4Colour colour(1.,0.,0.);
     G4VisAttributes attribs(colour);
     circle.SetVisAttributes(attribs);

     // make a 3D data for visualization
     pVVisManager->Draw(circle);
   }
 }

 //----- end of C++ source codes
```

The overloaded implementation of `G4VHitsCollection::DrawAllHits()` will be held by, for example, class *MyTrackerHitsCollection* inheriting class `G4VHitsCollection` as follows:

```
//----- C++ source codes: An example of giving concrete implementation of
//       G4VHitsCollection::Draw(),
//       using  class MyTrackerHit : public G4VHitsCollection{...}
//
void MyTrackerHitsCollection::DrawAllHits()
{
  G4int n_hit = theCollection.entries();
  for(G4int i=0;i < n_hit;i++)
  {
    theCollection[i].Draw();
  }
}

//----- end of C++ source codes
```

Thus, you can visualize hits as well as trajectories, for example, at the end of each event by implementing the method `MyEventAction::EndOfEventAction()` as follows:

```
void MyEventAction::EndOfEventAction()
{
  const G4Event* evt = fpEventManager->GetConstCurrentEvent();

  G4SDManager * SDman = G4SDManager::GetSDMpointer();
  G4String colNam;
  G4int trackerCollID = SDman->GetCollectionID(colNam="TrackerCollection");
  G4int calorimeterCollID = SDman->GetCollectionID(colNam="CalCollection");

  G4TrajectoryContainer * trajectoryContainer = evt->GetTrajectoryContainer();
  G4int n_trajectories = 0;
  if(trajectoryContainer)
  { n_trajectories = trajectoryContainer->entries(); }

  G4HCofThisEvent * HCE = evt->GetHCofThisEvent();
  G4int n_hitCollection = 0;
  if(HCE)
  { n_hitCollection = HCE->GetCapacity(); }

  G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
```

```
  if(pVVisManager)
  {

    // Declare begininng of visualization
    G4UImanager::GetUIpointer()->ApplyCommand("/vis/scene/notifyHandlers");

    // Draw trajectories
    for(G4int i=0; i < n_trajectories; i++)
    {
        (*(evt->GetTrajectoryContainer()))[i]->DrawTrajectory();
    }

    // Construct 3D data for hits
    MyTrackerHitsCollection* THC
      = (MyTrackerHitsCollection*)(HCE->GetHC(trackerCollID));
    if(THC) THC->DrawAllHits();
    MyCalorimeterHitsCollection* CHC
      = (MyCalorimeterHitsCollection*)(HCE->GetHC(calorimeterCollID));
    if(CHC) CHC->DrawAllHits();

    // Declare end of visualization
    G4UImanager::GetUIpointer()->ApplyCommand("/vis/viewer/update");

  }

}

//----- end of C++ codes
```

You can re-visualize a physical volume, where a hit is detected, with a highlight color, in addition to the whole set of detector components. It is done by calling a drawing method of a physical volume directly. The method is:

```
//----- Drawing methods of a physical volume
virtual void Draw (const G4VPhysicalVolume&, ...) ;
```

This method is, for example, called in a method `MyXXXHit::Draw()`, describing the visualization of hits with markers. The following is an example for this:

```
//----- C++ source codes: An example of visualizing hits with
void MyCalorimeterHit::Draw()
{
  G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
  if(pVVisManager)
  {
    G4Transform3D trans(rot,pos);
    G4VisAttributes attribs;
    G4LogicalVolume* logVol = pPhys->GetLogicalVolume();
    const G4VisAttributes* pVA = logVol->GetVisAttributes();
    if(pVA) attribs = *pVA;
    G4Colour colour(1.,0.,0.);
    attribs.SetColour(colour);
    attribs.SetForceSolid(true);

    //----- Re-visualization of a selected physical volume with red color
    pVVisManager->Draw(*pPhys,attribs,trans);

  }
}

//----- end of C++ codes
```

8.5.7 HepRep Attributes for Hits

The HepRep file formats, HepRepFile and HepRepXML, attach various attributes to hits such that you can view these attributes, label trajectories by these attributes or make visibility cuts based on these attributes. Examples of adding HepRep attributes to hit classes can be found in examples /extended/analysis/A01 and /extended/runAndEvent/RE01.

For example, in example RE01's class RE01CalorimeterHit.cc, available attributes will be:

- Hit Type
- Track ID
- Z Cell ID
- Phi Cell ID
- Energy Deposited
- Energy Deposited by Track
- Position
- Logical Volume

You can add additional attributes of your choosing by modifying the relevant part of the hit class (look for the methods GetAttDefs and CreateAttValues).

8.5.8 Visualization of text

In Geant4 Visualization, a text, i.e., a character string, is described by class `G4Text` inheriting `G4VMarker` as well as `G4Square` and `G4Circle`. Therefore, the way to visualize text is the same as for hits. The corresponding drawing method of `G4VVisManager` is:

```
//----- Drawing methods of G4Text
virtual void G4VVisManager::Draw (const G4Text&, ...);
```

The real implementation of this method is described in class `G4VisManager`.

8.5.9 Visualization of polylines and tracking steps

Polylines, i.e., sets of successive line segments, are described by class `G4Polyline`. For `G4Polyline`, the following drawing method of class `G4VVisManager` is prepared:

```
//----- A drawing method of G4Polyline
 virtual void G4VVisManager::Draw (const G4Polyline&, ...) ;
```

The real implementation of this method is described in class `G4VisManager`.

Using this method, C++ source codes to visualize `G4Polyline` are described as follows:

```
//----- C++ source code: How to visualize a polyline
 G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();

 if (pVVisManager) {
     G4Polyline polyline ;

     ..... (C++ source codes to set vertex positions, color, etc)

     pVVisManager -> Draw(polyline);

 }

 //----- end of C++ source codes
```

Tracking steps are able to be visualized based on the above visualization of `G4Polyline`. You can visualize tracking steps at each step automatically by writing a proper implementation of class *MySteppingAction* inheriting `G4UserSteppingAction`, and also with the help of the Run Manager.

First, you must implement a method, `MySteppingAction::UserSteppingAction()`. A typical implementation of this method is as follows:

```
//----- C++ source code: An example of visualizing tracking steps
void MySteppingAction::UserSteppingAction()
{
    G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();

    if (pVVisManager) {

      //----- Get the Stepping Manager
      const G4SteppingManager* pSM = GetSteppingManager();

      //----- Define a line segment
      G4Polyline polyline;
      G4double charge = pSM->GetTrack()->GetDefinition()->GetPDGCharge();
      G4Colour colour;
      if      (charge < 0.) colour = G4Colour(1., 0., 0.);
      else if (charge < 0.) colour = G4Colour(0., 0., 1.);
      else                  colour = G4Colour(0., 1., 0.);
      G4VisAttributes attribs(colour);
      polyline.SetVisAttributes(attribs);
      polyline.push_back(pSM->GetStep()->GetPreStepPoint()->GetPosition());
      polyline.push_back(pSM->GetStep()->GetPostStepPoint()->GetPosition());

      //----- Call a drawing method for G4Polyline
      pVVisManager -> Draw(polyline);

    }
}

//----- end of C++ source code
```

Next, in order that the above C++ source code works, you have to pass the information of the *MySteppingAction* to the Run Manager in the `main()` function:

```
//----- C++ source code: Passing what to do at each step to the Run Manager

int main()
{
   ...

   // Run Manager
   G4RunManager * runManager = new G4RunManager;

   // User initialization classes
   ...
   runManager->SetUserAction(new MySteppingAction);
   ...
}

//----- end of C++ source code
```

Thus you can visualize tracking steps with various visualization attributes, e.g., color, at each step, automatically.

As well as tracking steps, you can visualize any kind 3D object made of line segments, using class `G4Polyline` and its drawing method, defined in class `G4VVisManager`. See, for example, the implementation of the `/vis/scene/add/axes` command.

8.5.10 Visualization User Action

You can implement the `Draw` method of `G4VUserVisAction`, e.g., the class definition could be:

```
class StandaloneVisAction: public G4VUserVisAction {
  void Draw();
};
```

and the implementation:

```
void StandaloneVisAction::Draw() {
  G4VVisManager* pVisManager = G4VVisManager::GetConcreteInstance();
  if (pVisManager) {

    // Simple box...
    pVisManager->Draw(G4Box("box",2*m,2*m,2*m),
                      G4VisAttributes(G4Colour(1,1,0)));

    // Boolean solid...
    G4Box boxA("boxA",3*m,3*m,3*m);
    G4Box boxB("boxB",1*m,1*m,1*m);
    G4SubtractionSolid subtracted("subtracted_boxes",&boxA,&boxB,
                       G4Translate3D(3*m,3*m,3*m));
    pVisManager->Draw(subtracted,
                      G4VisAttributes(G4Colour(0,1,1)),
                      G4Translate3D(6*m,6*m,6*m));
  }
}
```

Explicit use of polyhedron objects is equivalent, e.g.:

```
// Same, but explicit polyhedron...
G4Polyhedron* pA = G4Box("boxA",3*m,3*m,3*m).CreatePolyhedron();
G4Polyhedron* pB = G4Box("boxB",1*m,1*m,1*m).CreatePolyhedron();
pB->Transform(G4Translate3D(3*m,3*m,3*m));
G4Polyhedron* pSubtracted = new G4Polyhedron(pA->subtract(*pB));
G4VisAttributes subVisAtts(G4Colour(0,1,1));
pSubtracted->SetVisAttributes(&subVisAtts);
pVisManager->Draw(*pSubtracted,G4Translate3D(6*m,6*m,6*m));
delete pA;
delete pB;
delete pSubtracted;
```

If efficiency is an issue, create the objects in the constructor, delete them in the destructor and draw them in your `Draw` method. Anyway, an instance of your class needs to be registered with the vis manager, e.g.:

```
...
G4VisManager* visManager = new G4VisExecutive;
visManager->Initialize ();

visManager->SetUserAction
  (new StandaloneVisAction,
   G4VisExtent(-5*m,5*m,-5*m,5*m,-5*m,5*m));  // 2nd argument optional.
...
```

then activate by adding to a scene, e.g:

```
/control/verbose 2
/vis/verbose c
/vis/open OGLSXm
/vis/scene/create
#/vis/scene/add/userAction
/vis/scene/add/userAction -10 10 -10 10 -10 10 m
#/vis/scene/add/axes 0 0 0 10 m
#/vis/scene/add/scale 10 m
```

```
/vis/sceneHandler/attach
/vis/viewer/refresh
```

The extent can be added on registration or on the command line or neither (if the extent of the scene is set by other components). Your `Draw` method will be called whenever needed to refresh the screen or rebuild a graphics database, for any chosen viewer. The scene can be attached to any scene handler and your drawing will be shown.

8.5.11 Standalone Visualization

The above raises the possibility of using Geant4 as a "standalone" graphics package without invoking the run manager. The following main program, together with a user visualization action and a macro file, will allow you to view your drawing interactively on any of the supported graphics systems.

```
#include "globals.hh"
#include "G4VisExecutive.hh"
#include "G4VisExtent.hh"
#include "G4UImanager.hh"
#include "G4UIterminal.hh"
#include "G4UItcsh.hh"

#include "StandaloneVisAction.hh"

int main() {

  G4VisManager* visManager = new G4VisExecutive;
  visManager->Initialize ();

  visManager->SetUserAction
    (new StandaloneVisAction,
     G4VisExtent(-5*m,5*m,-5*m,5*m,-5*m,5*m));  // 2nd argument optional.

  G4UImanager* UI = G4UImanager::GetUIpointer ();
  UI->ApplyCommand ("/control/execute standalone.g4m");

  G4UIsession* session = new G4UIterminal(new G4UItcsh);
  session->SessionStart();

  delete session;
  delete visManager;
}
```

8.6 Visualization Attributes

Visualization attributes are extra pieces of information associated with the visualizable objects. This information is necessary only for visualization, and is not included in geometrical information such as shapes, position, and orientation. Typical examples of visualization attributes are Color, Visible/Invisible, Wireframe/Solid. For example, in visualizing a box, the Visualization Manager must know its colour. If an object to be visualized has not been assigned a set of visualization attributes, then an appropriate default set is used automatically.

A set of visualization attributes is held by an instance of class `G4VisAttributes` defined in the `graphics_reps` category. In the following, we explain the main fields of the `G4VisAttributes` one by one.

8.6.1 Visibility

Visibility is a boolean flag to control the visibility of objects that are passed to the Visualization Manager for visualization. Visibility is set with the following access function:

```
void G4VisAttributes::SetVisibility (G4bool visibility);
```

If you give `false` to the argument, and if culling is activated (see below), visualization is skipped for objects for which this set of visualization attributes is assigned. The default value of visibility is `true`.

Note that whether an object is visible or not is also affected by the current culling policy, which can be tuned with visualization commands.

By default the following public static function is defined:

```
static const G4VisAttributes& GetInvisible();
```

which returns a reference to a const object in which visibility is set to `false`. It can be used as follows:

```
experimentalHall_logical -> SetVisAttributes (G4VisAttributes::GetInvisible());
```

Direct access to the public static const data member `G4VisAttributes::Invisible` is also possible but deprecated on account of initialisation issues with dynamic libraries.

8.6.2 Colour

Construction

Class `G4Colour` (an equivalent class name, `G4Color`, is also available) has 4 fields, which represent the RGBA (red, green, blue, and alpha) components of colour. Each component takes a value between 0 and 1. If an irrelevant value, i.e., a value less than 0 or greater than 1, is given as an argument of the constructor, such a value is automatically clipped to 0 or 1. Alpha is opacity. Its default value `1` means "opaque".

A `G4Colour` object is instantiated by giving red, green, and blue components to its constructor, i.e.,

```
G4Colour::G4Colour ( G4double r = 1.0,
                     G4double g = 1.0,
                     G4double b = 1.0,
                     G4double a = 1.0);
                             // 0<=red, green, blue <= 1.0
```

The default value of each component is 1.0. That is to say, the default colour is "white" (opaque).

For example, colours which are often used can be instantiated as follows:

```
G4Colour  white   ()               ;  // white
G4Colour  white   (1.0, 1.0, 1.0) ;  // white
G4Colour  gray    (0.5, 0.5, 0.5) ;  // gray
G4Colour  black   (0.0, 0.0, 0.0) ;  // black
G4Colour  red     (1.0, 0.0, 0.0) ;  // red
G4Colour  green   (0.0, 1.0, 0.0) ;  // green
G4Colour  blue    (0.0, 0.0, 1.0) ;  // blue
G4Colour  cyan    (0.0, 1.0, 1.0) ;  // cyan
G4Colour  magenta (1.0, 0.0, 1.0) ;  // magenta
G4Colour  yellow  (1.0, 1.0, 0.0) ;  // yellow
```

It is also possible to instantiate common colours through static public data member functions:

```
static const G4Colour& White   ();
static const G4Colour& Gray    ();
static const G4Colour& Grey    ();
static const G4Colour& Black   ();
static const G4Colour& Red     ();
static const G4Colour& Green   ();
static const G4Colour& Blue    ();
```

```
static const G4Colour& Cyan    ();
static const G4Colour& Magenta ();
static const G4Colour& Yellow  ();
```

For example, a local G4Colour could be constructed as:

```
G4Colour myRed(G4Colour::Red());
```

After instantiation of a G4Colour object, you can access to its components with the following access functions:

```
G4double G4Colour::GetRed   () const ; // Get the red   component.
G4double G4Colour::GetGreen () const ; // Get the green component.
G4double G4Colour::GetBlue  () const ; // Get the blue  component.
```

Colour Map

G4Colour also provides a static colour map, giving access to predefined G4Colour's through a G4String key. The default mapping is:

```
G4String           G4Colour
-----------------------------------
white              G4Colour::White   ()
gray               G4Colour::Gray    ()
grey               G4Colour::Grey    ()
black              G4Colour::Black   ()
red                G4Colour::Red     ()
green              G4Colour::Green   ()
blue               G4Colour::Blue    ()
cyan               G4Colour::Cyan    ()
magenta            G4Colour::Magenta ()
yellow             G4Colour::Yellow  ()
```

Colours can be retrieved through the GetColour method:

```
bool G4Colour::GetColour(const G4String& key, G4Colour& result)
```

For example:

```
G4Colour myColour(G4Colour::Black());
if (G4Colour::GetColour("red", myColour)) {
  // Successfully retrieved colour "red". myColour is now red
}
else {
  // Colour did not exist in map. myColour is still black
}
```

To see a list of available named colours, /vis/list. These names may also be used to specify colours in many /vis commands.

If the key is not registered in the colour map, a warning message is printed and the input colour is not changed. The colour map is case insensitive.

It is also possible to load user defined G4Colour's into the map through the public AddToMap method. For example:

```
G4Colour myColour(0.2, 0.2, 0.2, 1);
G4Colour::AddToMap("custom", myColour);
```

This loads a user defined G4Colour with key "custom" into the colour map.

It is also possible to use the colours in g4tools:

```
#include "tools/colors"
...
G4Colour niceColour = tools::get_color_aquamarine<G4Colour>();
```

Colour and G4VisAttributes

Class G4VisAttributes holds its colour entry as an object of class G4Colour. A G4Colour object is passed to a G4VisAttributes object with the following access functions:

```
//----- Set functions of G4VisAttributes.
void G4VisAttributes::SetColour (const G4Colour& colour);
void G4VisAttributes::SetColor (const G4Color& color );
```

We can also set RGBA components directly:

```
//----- Set functions of G4VisAttributes
void G4VisAttributes::SetColour ( G4double red   ,
                                  G4double green ,
                                  G4double blue  ,
                                  G4double alpha = 1.0);

void G4VisAttributes::SetColor  ( G4double red   ,
                                  G4double green ,
                                  G4double blue  ,
                                  G4double alpha = 1.);
```

The following constructor with G4Colour as its argument is also supported:

```
//----- Constructor of G4VisAttributes
G4VisAttributes::G4VisAttributes (const G4Colour& colour);
```

Note that colour assigned to a G4VisAttributes object is not always the colour that ultimately appears in the visualization. The ultimate appearance may be affected by shading and lighting models applied in the selected visualization driver or stand-alone graphics system.

8.6.3 Forcing attributes

As you will see later, you can select a "drawing style" from various options. For example, you can select your detector components to be visualized in "wireframe" or with "surfaces". In the former, only the edges of your detector are drawn and so the detector looks transparent. In the latter, your detector looks opaque with shading effects.

The forced wireframe and forced solid styles make it possible to mix the wireframe and surface visualization (if your selected graphics system supports such visualization). For example, you can make only the outer wall of your detector "wired" (transparent) and can see inside in detail.

Forced wireframe style is set with the following access function:

```
void G4VisAttributes::SetForceWireframe (G4bool force);
```

If you give true as the argument, objects for which this set of visualization attributes is assigned are always visualized in wireframe even if in general, the surface drawing style has been requested. The default value of the forced wireframe style is false.

Similarly, forced solid style, i.e., to force that objects are always visualized with surfaces, is set with:

```
void G4VisAttributes::SetForceSolid (G4bool force);
```

The default value of the forced solid style is `false`, too.

You can also force auxiliary edges to be visible. Normally they are not visible unless you set the appropriate view parameter. Forcing the auxiliary edges to be visible means that auxiliary edges will be seen whatever the view parameters.

Auxiliary edges are not genuine edges of the volume. They may be in a curved surface made out of polygons, for example, or in plane surface of complicated shape that has to be broken down into simpler polygons. HepPolyhedron breaks all surfaces into triangles or quadrilaterals. There will be auxiliary edges for any volumes with a curved surface, such as a tube or a sphere, or a volume resulting from a Boolean operation. Normally, they are not shown, but sometimes it is useful to see them. In particular, a sphere, because it has no egdes, will not be seen in wireframe mode in some graphics systems unless requested by the view parameters or forced, as described here.

To force auxiliary edges to be visible, use:

```
void G4VisAttributes::SetForceAuxEdgeVisible (G4bool force);
```

The default value of the force auxiliary edges visible flag is `false`.

For volumes with edges that are parts of a circle, such as a tube (G4Tubs), etc., it is possible to force the precision of polyhedral representation for visualisation. This is recommended for volumes containing only a small angle of circle, for example, a thin tube segment.

For visualisation, a circle is represented by an N-sided polygon. The default is 24 sides or segments. The user may change this for all volumes in a particular viewer at run time with /vis/viewer/set/lineSegmentsPerCircle; alternatively it can be forced for a particular volume with:

```
void G4VisAttributes::SetForceLineSegmentsPerCircle (G4int nSegments);
```

8.6.4 Other attributes

Here is a list of Set methods for class `G4VisAttributes`:

```
void SetVisibility          (G4bool);
void SetDaughtersInvisible  (G4bool);
void SetColour              (const G4Colour&);
void SetColor               (const G4Color&);
void SetColour              (G4double red, G4double green, G4double blue,
                             G4double alpha = 1.);
void SetColor               (G4double red, G4double green, G4double blue,
                             G4double alpha = 1.);
void SetLineStyle           (LineStyle);
void SetLineWidth           (G4double);
void SetForceWireframe      (G4bool);
void SetForceSolid          (G4bool);
void SetForceAuxEdgeVisible (G4bool);
void SetForceLineSegmentsPerCircle (G4int nSegments);
// Allows choice of circle approximation.  A circle of 360 degrees
// will be composed of nSegments line segments.  If your solid has
// curves of D degrees that you need to divide into N segments,
// specify nSegments = N * 360 / D.
void SetStartTime           (G4double);
void SetEndTime             (G4double);
void SetAttValues           (const std::vector<G4AttValue>*);
void SetAttDefs             (const std::map<G4String,G4AttDef>*);
```

8.6.5 Constructors of G4VisAttributes

The following constructors are supported for class `G4VisAttributes`:

```
//----- Constructors of class G4VisAttributes
G4VisAttributes (void);
G4VisAttributes (G4bool visibility);
G4VisAttributes (const G4Colour& colour);
G4VisAttributes (G4bool visibility, const G4Colour& colour);
```

8.6.6 How to assign G4VisAttributes to a logical volume

In constructing your detector components, you may assign a set of visualization attributes to each "logical volume" in order to visualize them later (if you do not do this, the graphics system will use a default set). You cannot make a solid such as `G4Box` hold a set of visualization attributes; this is because a solid should hold only geometrical information. At present, you cannot make a physical volume hold one, but there are plans to design a memory-efficient way to do it; however, you can visualize a transient piece of solid or physical volume with a temporary assigned set of visualization attributes.

Class `G4LogicalVolume` holds a pointer of `G4VisAttributes`. This field is set and referenced with the following access functions:

```
//----- Set functions of G4VisAttributes
void G4VisAttributes::SetVisAttributes (const G4VisAttributes* pVA);
void G4VisAttributes::SetVisAttributes (const G4VisAttributes& VA);

//----- Get functions of G4VisAttributes
const G4VisAttributes* G4VisAttributes::GetVisAttributes () const;
```

The following is sample C++ source codes for assigning a set of visualization attributes with cyan colour and forced wireframe style to a logical volume:

```
//----- C++ source codes: Assigning G4VisAttributes to a logical volume
...
     // Instantiation of a logical volume
myTargetLog = new G4LogicalVolume( myTargetTube,BGO, "TLog", 0, 0, 0);
...
     // Instantiation of a set of visualization attributes with cyan colour
G4VisAttributes * calTubeVisAtt = new G4VisAttributes(G4Colour(0.,1.,1.));
     // Set the forced wireframe style
calTubeVisAtt->SetForceWireframe(true);
     // Assignment of the visualization attributes to the logical volume
myTargetLog->SetVisAttributes(calTubeVisAtt);

//----- end of C++ source codes
```

Note that the life of the visualization attributes must be at least as long as the objects to which they are assigned; it is the users' responsibility to ensure this, and to delete the visualization attributes when they are no longer needed (or just leave them to die at the end of the job).

8.6.7 Additional User-Defined Attributes

Geant4 Trajectories and Hits can be assigned additional arbitrary attributes that will be displayed when you click on the relevant object in the WIRED or FRED HepRep browsers. WIRED then lets you label objects by any of these attributes or cut visibility based on these attributes.

Define the attributes with lines such as:

```
std::map<G4String,G4AttDef>* store = G4AttDefStore::GetInstance("G4Trajectory",isNew);
G4String PN("PN");
(*store)[PN] = G4AttDef(PN,"Particle Name","Physics","","G4String");
G4String IMom("IMom");
```

```
(*store)[IMom] = G4AttDef(IMom, "Momentum of track at start of trajectory", "Physics", "",
                                "G4ThreeVector");
```

Then fill the attributes with lines such as:

```
std::vector<G4AttValue>* values = new std::vector<G4AttValue>;
values->push_back(G4AttValue("PN",ParticleName,""));
s.seekp(std::ios::beg);
s << G4BestUnit(initialMomentum,"Energy") << std::ends;
values->push_back(G4AttValue("IMom",c,""));
```

See geant4/source/tracking/src/G4Trajectory.cc for a good example.

`G4AttValue` objects are light, containing just the value; for the long description and other sharable information the `G4AttValue` object refers to a `G4AttDef` object. They are based on the HepRep standard described at http://www.slac.stanford.edu/~perl/heprep/. Geant4 also provides an `G4AttDefStore`.

Geant4 provides some default examples of the use of this facility in the trajectory classes in /source/tracking such as `G4Trajectory`, `G4SmoothTrajectory`. `G4Trajectory::CreateAttValues` shows how `G4AttValue` objects can be made and `G4Trajectory::GetAttDefs` shows how to make the corresponding `G4AttDef` objects and use the `G4AttDefStore`. Note that the "user" of CreateAttValues guarantees to destroy them; this is a way of allowing creation on demand and leaving the `G4Trajectory` object, for example, free of such objects in memory. The comments in `G4VTrajectory.hh` explain further and additional insights might be obtained by looking at two methods which use them, namely `G4VTrajectory::DrawTrajectory` and `G4VTrajectory::ShowTrajectory`.

Hits classes in examples /extended/analysis/A01 and /extended/runAndEvent/RE01 show how to do the same for your hits. The base class no-action methods CreateAttValues and GetAttDefs should be overridden in your concrete class. The comments in `G4VHit.hh` explain further.

In addition, the user is free to add a `G4std::vector<G4AttValue>*` and a `G4std::vector<G4AttDef>*` to a `G4VisAttributes` object as could, for example, be used by a `G4LogicalVolume` object.

At the time of writing, only the HepRep graphics systems are capable of displaying the G4AttValue information, but this information will become useful for all Geant4 visualization systems through improvements in release 8.1 or later.

8.7 Enhanced Trajectory Drawing

8.7.1 Default Configuration

Trajectory drawing styles are specified through trajectory drawing models. Each drawing model has a default configuration provided through a G4VisTrajContext object. The default context settings are shown below.

Property	Default Setting
Line colour	grey
Line visibility	true
Draw line	true
Draw auxiliary points	false
Auxiliary point type	squares
Auxiliary point size	2 pixels or mm*
Auxiliary point size type	screen
Auxiliary point fill style	filled
Auxiliary point colour	magenta
Auxiliary point visibility	true
Draw step point	false
Step point type	circles
Step point size	2 pixels or mm*
Step point size type	screen
Step point fill style	filled
Step point colour	yellow
Step point visibility	true
Time slice interval	0

* Depending on size type. If size type == screen, pixels are assumed and no unit need be supplied. If size type == world, a unit must be supplied, e.g., 10 cm.

Note: Different visualisation drivers handle trajectory configuration in different ways, so trajectories may not necessarily get displayed as you have configured them.

8.7.2 Trajectory Drawing Models

A trajectory drawing model can override the default context according to the properties of a given trajectory. The following models are supplied with the Geant4 distribution:

- G4TrajectoryGenericDrawer (generic)
- G4TrajectoryDrawByCharge (drawByCharge)
- G4TrajectoryDrawByParticleID (drawByParticleID)
- G4TrajectoryDrawByOriginVolume (drawByOriginVolume)
- G4TrajectoryDrawByTouchedVolume (drawByTouchedVolume)
- G4TrajectoryDrawByAttribute (drawByAttribute)

Both the context and model properties can be configured by the user. The models are described briefly below, followed by some example configuration commands.

G4TrajectoryGenericDrawer

This model simply draws all trajectories in the same style, with the properties provided by the context.

G4TrajectoryDrawByCharge

This is the default model - if no model is specified by the user, this model will be constructed automatically. The trajectory lines are coloured according to charge, with all other configuration parameters provided by the default context. The default colouring scheme is shown below.

```
Charge          Colour
----------------------
```

```
1               blue
-1              red
0               green
```

G4TrajectoryDrawByParticleID

This model colours trajectory lines according to particle type. All other configuration parameters are provided by the default context. Chosen particle types can be highlighted with specified colours. By default, trajectories are coloured according to the scheme below and any other particle in the default colour (grey). (All may be overridden by the set command.)

```
Particle        Colour
----------------------
gamma           green
e-              red
e+              blue
pi+             magenta
pi-             magenta
proton          cyan
neutron         yellow
```

G4TrajectoryDrawByOriginVolume

This model colours trajectory lines according to the trajectory's originating volume name. The volume can be either a logical or physical volume. Physical volume takes precedence over logical volume. All trajectories are coloured grey by default.

G4TrajectoryDrawByTouchedVolume

This model colours trajectory lines if it touches one or more volumes according to the physical volume name(s). It requires rich trajectories, G4RichTrajectory (`/vis/scene/add/trajectories rich`). All trajectories are coloured grey by default.

G4TrajectoryDrawByAttribute

This model draws trajectories based on the HepRep style attributes associated with trajectories. Each attribute drawer can be configured with interval and/or single value data. A new context object is created for each interval/single value. This makes it possible to have different step point markers etc, as well as line colour for trajectory attributes falling into different intervals, or matching single values. The single value data should override the interval data, allowing specific values to be highlighted. Units should be specified on the command line if the attribute unit is specified either as a G4BestUnit or if the unit is part of the value string.

8.7.3 Controlling from Commands

Multiple trajectory models can be created and configured using commands in the "`/vis/modeling/trajectories/`" directory. It is then possible to list available models and select one to be current.

Model configuration commands are generated dynamically when a model is instantiated. These commands apply directly to that instance. This makes it possible to have multiple instances of the drawByCharge model for example, each independently configurable through it's own set of commands.

See the interactive help for more information on the available commands.

Example commands

- Create a generic model named generic-0 by default

```
/vis/modeling/trajectories/create/generic
```

- Configure context to colour all trajectories red

```
/vis/modeling/trajectories/generic-0/default/setLineColour red
```

- Create a drawByCharge model named drawCharge-0 by default (Subsequent models will be named drawByCharge-1, drawByCharge-2, etc.)

```
/vis/modeling/trajectories/create/drawByCharge
```

- Create a drawByCharge model named testChargeModel

```
/vis/modeling/trajectories/create/drawByCharge testChargeModel
```

- Configure drawByCharge-0 model

```
/vis/modeling/trajectories/drawByCharge-0/set 1 red
/vis/modeling/trajectories/drawByCharge-0/set -1 red
/vis/modeling/trajectories/drawByCharge-0/set 0 white
```

- Configure testCharge model through G4Colour components

```
/vis/modeling/trajectories/testChargeModel/setRGBA 1 0 1 1 1
/vis/modeling/trajectories/testChargeModel/setRGBA -1 0.5 0.5 0.5 1
/vis/modeling/trajectories/testChargeModel/setRGBA 0 1 1 0 1
```

- Create a drawByParticleID model named drawByParticleID-0

```
/vis/modeling/trajectories/create/drawByParticleID
```

- Configure drawByParticleID-0 model

```
/vis/modeling/trajectories/drawByParticleID-0/set gamma red
/vis/modeling/trajectories/drawByParticleID-0/setRGBA e+ 1 0 1 1
```

- List available models

```
/vis/modeling/trajectories/list
```

- select drawByParticleID-0 to be current

```
/vis/modeling/trajectories/select drawByParticleID-0
```

- Create a drawByAttribute model named drawByAttribute-0

```
/vis/modeling/trajectories/create/drawByAttribute
```

- Configure drawByAttribute-0 model

```
/vis/modeling/trajectories/drawByAttribute-0/verbose true
```

- Select attribute “CPN”

```
/vis/modeling/trajectories/drawByAttribute-0/setAttribute CPN
```

- Configure single value data

```
/vis/modeling/trajectories/drawByAttribute-0/addValue brem_key  eBrem
/vis/modeling/trajectories/drawByAttribute-0/addValue annihil_key annihil
/vis/modeling/trajectories/drawByAttribute-0/addValue decay_key Decay
/vis/modeling/trajectories/drawByAttribute-0/addValue muIon_key muIoni
/vis/modeling/trajectories/drawByAttribute-0/addValue eIon_key  eIoni

/vis/modeling/trajectories/drawByAttribute-0/brem_key/setLineColour     red
/vis/modeling/trajectories/drawByAttribute-0/annihil_key/setLineColour  green
/vis/modeling/trajectories/drawByAttribute-0/decay_key/setLineColour    cyan
/vis/modeling/trajectories/drawByAttribute-0/eIon_key/setLineColour     yellow
/vis/modeling/trajectories/drawByAttribute-0/muIon_key/setLineColour magenta
```

- Create a drawByAttribute model named drawByAttribute-1

```
/vis/modeling/trajectories/create/drawByAttribute
```

- Select "IMag" attribute

```
/vis/modeling/trajectories/drawByAttribute-1/setAttribute IMag
```

- Configure interval data

```
/vis/modeling/trajectories/drawByAttribute-1/addInterval interval1 0.0 keV 2.5MeV
/vis/modeling/trajectories/drawByAttribute-1/addInterval interval2 2.5 MeV 5 MeV
/vis/modeling/trajectories/drawByAttribute-1/addInterval interval3 5 MeV 7.5 MeV
/vis/modeling/trajectories/drawByAttribute-1/addInterval interval4 7.5 MeV 10 MeV
/vis/modeling/trajectories/drawByAttribute-1/addInterval interval5 10 MeV 12.5 MeV
/vis/modeling/trajectories/drawByAttribute-1/addInterval interval6 12.5 MeV 10000 MeV

/vis/modeling/trajectories/drawByAttribute-1/interval1/setLineColourRGBA 0.8 0 0.8 1
/vis/modeling/trajectories/drawByAttribute-1/interval2/setLineColourRGBA 0.23 0.41 1 1
/vis/modeling/trajectories/drawByAttribute-1/interval3/setLineColourRGBA 0 1 0 1
/vis/modeling/trajectories/drawByAttribute-1/interval4/setLineColourRGBA 1 1 0 1
/vis/modeling/trajectories/drawByAttribute-1/interval5/setLineColourRGBA 1 0.3 0 1
/vis/modeling/trajectories/drawByAttribute-1/interval6/setLineColourRGBA 1 0 0 1
```

- Create a drawByEncounteredVolume model named drawByEncounteredVolume-0

```
/vis/modeling/trajectories/create/drawByEncounteredVolume
```

- Change the color for a specific encountered shape

```
/vis/modeling/trajectories/drawByEncounteredVolume-0/set Shape1 cyan
```

8.7.4 Controlling from Compiled Code

It is possible to use the enhanced trajectory drawing functionality in compiled code as well as from commands. Multiple trajectory models can be instantiated, configured and registered with G4VisManager. Once registered, the models are owned by G4VisManager, and must not be deleted by the user.

Only one model may be current. For example:

```
G4VisManager* visManager = new G4VisExecutive;
visManager->Initialize();

G4TrajectoryDrawByParticleID* model = new G4TrajectoryDrawByParticleID;
G4TrajectoryDrawByParticleID* model2 = new G4TrajectoryDrawByParticleID("test");

model->SetDefault("cyan");
model->Set("gamma", "green");
model->Set("e+", "magenta");
model->Set("e-", G4Colour(0.3, 0.3, 0.3));

visManager->RegisterModel(model);
visManager->RegisterModel(model2);

visManager->SelectTrajectoryModel(model->Name());
```

8.7.5 Drawing by time

To draw by time, you need to use G4RichTrajectory, for example:

```
/vis/scene/add/trajectories rich
```

or

```
/vis/scene/add/trajectories rich smooth
```

When you run, you need to create a trajectory model and set the time slice interval (remembering that paticles are often relativistic and travel 30 cm/ns):

```
/vis/modeling/trajectories/create/drawByCharge
/vis/modeling/trajectories/drawByCharge-0/default/setDrawStepPts true
/vis/modeling/trajectories/drawByCharge-0/default/setStepPtsSize 5
/vis/modeling/trajectories/drawByCharge-0/default/setDrawAuxPts true
/vis/modeling/trajectories/drawByCharge-0/default/setAuxPtsSize 5
/vis/modeling/trajectories/drawByCharge-0/default/setTimeSliceInterval 0.1 ns
/vis/modeling/trajectories/list
```

and use a graphics driver that can display by time:

```
/vis/open OGL
/vis/drawVolume
/vis/scene/add/trajectories rich
/vis/ogl/set/startTime 0.5 ns
/vis/ogl/set/endTime 0.8 ns
/run/beamOn
/vis/viewer/refresh
```

A good way to see the particles moving through the detector is:

```
/vis/ogl/set/fade 1
/vis/ogl/set/displayHeadTime true
/control/alias timeRange 1
/control/loop movie.loop startTime -{timeRange} 40 0.1
```

where `fade` gives a vapour-trail effect, `displayHeadTime` displays the time of the leading edge as 2D text, and `movie.loop` is a macro file:

```
/vis/ogl/set/startTime {startTime} ns {timeRange} ns
```

From there, it's straightforward to *Making a Movie* make a movie.

8.8 Trajectory Filtering

Trajectory filtering allows you to visualise a subset of available trajectories. This can be useful if you only want to view interesting trajectories and discard uninteresting ones. Trajectory filtering can be run in two modes:

- **Soft filtering**: In this mode, uninteresting trajectories are marked invisible. Hence, they are still written, but (depending on the driver) will not be displayed. Some drivers, for example the HepRepFile driver, will allow you to selectively view these soft filtered trajectories
- **Hard filtering**: In this mode, uninteresting trajectories are not drawn at all. This mode is especially useful if the job produces huge graphics files, dominated by data from uninteresting trajectories.

Trajectory filter models are used to apply filtering according to specific criteria. The following models are currently supplied with the Geant4 distribution:

- G4TrajectoryChargeFilter (chargeFilter)
- G4TrajectoryParticleFilter (particleFilter)
- G4TrajectoryOriginVolumeFilter (originVolumeFilter)
- G4TrajectoryTouchedVolumeFilter (touchedVolumeFilter)
- G4TrajectoryAttributeFilter (attributeFilter)

Multiple filters are automatically chained together, and can configured either interactively or in commands or in compiled code. The filters can be inverted, set to be inactive or set in a verbose mode. The above models are described briefly below, followed by some example configuration commands.

G4TrajectoryChargeFilter

This model filters trajectories according to charge. In standard running mode, only trajectories with charges matching those registered with the model will pass the filter.

G4TrajectoryParticleFilter

This model filters trajectories according to particle type. In standard running mode, only trajectories with particle types matching those registered with the model will pass the filter.

G4TrajectoryOriginVolumeFilter

This model filters trajectories according to originating volume name. In standard running mode, only trajectories with originating volumes matching those registered with the model will pass the filter.

G4TrajectoryTouchedVolumeFilter

This model filters trajectories that touch one or more volumes according to the physical volume name(s). It requires rich trajectories, G4RichTrajectory (`/vis/scene/add/trajectories rich`). In standard running mode, only trajectories that touch volumes matching those registered with the model will pass the filter.

G4TrajectoryAttributeFilter

This model filters trajectories based on the HepRep style attributes associated with trajectories. Each attribute drawer can be configured with interval and/or single value data. Single value data should override the interval data. Units should be specified on the command line if the attribute unit is specified either as a G4BestUnit or if the unit is part of the value string.

8.8.1 Controlling from Commands

Multiple trajectory filter models can be created and configured using commands in the "`/vis/filtering/trajectories/`" directory. All generated filter models are chained together automatically.

Model configuration commands are generated dynamically when a filter model is instantiated. These commands apply directly to that instance.

See the interactive help for more information on the available commands.

8.8.2 Example commands

```
# Create a particle filter. Configure to pass only gammas. Then
# invert to pass anything other than gammas. Set verbose printout,
# and then deactivate filter

/vis/filtering/trajectories/create/particleFilter
/vis/filtering/trajectories/particleFilter-0/add gamma
/vis/filtering/trajectories/particleFilter-0/invert true
/vis/filtering/trajectories/particleFilter-0/verbose true
/vis/filtering/trajectories/particleFilter-0/active false
```

```
# Create a charge filter. Configure to pass only neutral trajectories.
# Set verbose printout. Reset filter and reconfigure to pass only
# negativly charged trajectories.

/vis/filtering/trajectories/create/chargeFilter
/vis/filtering/trajectories/chargeFilter-0/add 0
/vis/filtering/trajectories/chargeFilter-0/verbose true
```

```
/vis/filtering/trajectories/chargeFilter-0/reset true
/vis/filtering/trajectories/chargeFilter-0/add -1
```

```
# Create an attribute filter named attributeFilter-0
/vis/filtering/trajectories/create/attributeFilter

# Select attribute "IMag"
/vis/filtering/trajectories/attributeFilter-0/setAttribute IMag

# Select trajectories with 2.5 MeV <= IMag< 1000 MeV
/vis/filtering/trajectories/attributeFilter-0/addInterval 2.5 MeV 1000 MeV
```

```
# List filters
/vis/filtering/trajectories/list
```

Note that although `particleFilter-0` and `chargeFilter-0` are automatically chained, `particleFilter-0` will not have any effect since it is has been deactivated.

8.8.3 Hit and Digi Filtering

The attribute based filtering can be used on hits and digitisations as well as trajectories. To active the interactive attribute based hit filtering, a filter call should be added to the "Draw" method of the hit (or digi) class:

```
void MyHit::Draw()
{
   ...
   if (! pVVisManager->FilterHit(*this)) return;
   ...
}
```

Interactive filtering can then be done through the commands in `/vis/filtering/hits` or `digi`.

8.9 Polylines, Markers and Text

Polylines, markers and text are defined in the `graphics_reps` category, and are used only for visualization (*Controlling Visualization from Compiled Code*). Users may create any of these objects with local scope; once drawn, they may safely be deleted or allowed to go out of scope.

8.9.1 Polylines

A polyline is a set of successive line segments. It is defined with a class `G4Polyline` defined in the `graphics_reps` category. A polyline is used to visualize tracking steps, particle trajectories, coordinate axes, and any other user-defined objects made of line segments.

`G4Polyline` is defined as a list of `G4Point3D` objects, i.e., vertex positions. The vertex positions are set to a `G4Polyline` object with the `push_back()` method.

For example, an x-axis with length 5 cm and with red color is defined in shown in the *example*.

Listing 8.5: Defining an x-axis with length 5 cm and with colour red.

```
//----- C++ source codes: An example of defining a line segment
// Instantiate an emply polyline object
G4Polyline  x_axis;

// Set red line colour
```

```
G4Colour         red(1.0, 0.0, 0.0);
G4VisAttributes  att(red);
x_axis.SetVisAttributes(&att);

// Set vertex positions
x_axis.push_back( G4Point3D(0., 0., 0.) );
x_axis.push_back( G4Point3D(5.*cm, 0., 0.) );

//----- end of C++ source codes
```

8.9.2 Markers

Here we explain how to use 3D markers in Geant4 Visualization.

What are Markers?

Markers set marks at arbitrary positions in the 3D space. They are often used to visualize hits of particles at detector components. A marker is a 2-dimensional primitive with shape (square, circle, etc), color, and special properties (a) of always facing the camera and (b) of having the possibility of a size defined in screen units (pixels). Here "size" means "overall size", e.g., diameter of circle and side of square (but diameter and radius access functions are defined to avoid ambiguity).

So the user who constructs a marker should decide whether or not it should be visualized to a given size in world coordinates by setting the world size. Alternatively, the user can set the screen size and the marker is visualized to its screen size. Finally, the user may decide not to set any size; in that case, it is drawn according to the sizes specified in the default marker specified in the class `G4ViewParameters`.

By default, "square" and "circle" are supported in Geant4 Visualization. The former is described with class `G4Square`, and the latter with class `G4Circle`:

Marker Type	Class Name
circle	`G4Circle`
right square	`G4Square`

These classes are inherited from class `G4VMarker`. They have constructors as follows:

```
//----- Constructors of G4Circle and G4Square
G4Circle::G4Circle (const G4Point3D& pos );
G4Square::G4Square (const G4Point3D& pos);
```

Access functions of class `G4VMarker` are summarized below.

Access functions of markers

The *example* shows the access functions inherited from the base class `G4VMarker`.

Listing 8.6: The access functions inherited from the base class `G4VMarker`.

```
//----- Set functions of G4VMarker
void G4VMarker::SetPosition( const G4Point3D& );
void G4VMarker::SetWorldSize( G4double );
void G4VMarker::SetWorldDiameter( G4double );
void G4VMarker::SetWorldRadius( G4double );
void G4VMarker::SetScreenSize( G4double );
void G4VMarker::SetScreenDiameter( G4double );
void G4VMarker::SetScreenRadius( G4double );
void G4VMarker::SetFillStyle( FillStyle );
// Note: enum G4VMarker::FillStyle {noFill, hashed, filled};

//----- Get functions of G4VMarker
```

```
G4Point3D G4VMarker::GetPosition () const;
G4double G4VMarker::GetWorldSize () const;
G4double G4VMarker::GetWorldDiameter () const;
G4double G4VMarker::GetWorldRadius () const;
G4double G4VMarker::GetScreenSize () const;
G4double G4VMarker::GetScreenDiameter () const;
G4double G4VMarker::GetScreenRadius () const;
FillStyle G4VMarker::GetFillStyle () const;
// Note: enum G4VMarker::FillStyle {noFill, hashed, filled};
```

The next *example* shows sample C++ source code to define a very small red circle, i.e., a dot with diameter 1.0 pixel. Such a dot is often used to visualize a hit.

Listing 8.7: Sample C++ source code to define a very small red circle.

```
//----- C++ source codes: An example of defining a red small maker
G4Circle circle(position); // Instantiate a circle with its 3D
                           // position. The argument "position"
                           // is defined as G4Point3D instance
circle.SetScreenDiameter (1.0); // Should be circle.SetScreenDiameter
                                //  (1.0 * pixels) - to be implemented
circle.SetFillStyle (G4Circle::filled); // Make it a filled circle
G4Colour colour(1.,0.,0.);              // Define red color
G4VisAttributes attribs(colour);        // Define a red visualization attribute
circle.SetVisAttributes(attribs);       // Assign the red attribute to the circle
//----- end of C++ source codes
```

8.9.3 Text

Text, i.e., a character string, is used to visualize various kinds of description, particle name, energy, coordinate names etc. Text is described by the class `G4Text` . The following constructors are supported:

```
//----- Constructors of G4Text
G4Text (const G4String& text);
G4Text (const G4String& text, const G4Point3D& pos);
```

where the argument `text` is the text (string) to be visualized, and `pos` is the 3D position at which the text is visualized.

Text is currently drawn only by the OpenGL drivers, such as OGLIX, OGLIXm and OpenInventor. It is not yet supported on other drivers, including the Windows OpenGL drivers, HepRep, etc.

Note that class `G4Text` also inherits `G4VMarker`. Size of text is recognized as "font size", i.e., height of the text. All the access functions defined for class `G4VMarker` mentioned above are available. In addition, the following access functions are available, too:

```
//----- Set functions of G4Text
void G4Text::SetText ( const G4String& text ) ;
void G4Text::SetOffset ( double dx, double dy ) ;

//----- Get functions of G4Text
G4String G4Text::GetText () const;
G4double G4Text::GetXOffset () const;
G4double G4Text::GetYOffset () const;
```

Method `SetText()` defines text to be visualized, and `GetText()` returns the defined text. Method `SetOffset()` defines x (horizontal) and y (vertical) offsets in the screen coordinates. By default, both offsets are zero, and the text starts from the 3D position given to the constructor or to the method `G4VMarker:SetPosition()`. Offsets should be given with the same units as the one adopted for the size, i.e., world-size or screen-size units.

The *example* shows sample C++ source code to define text with the following properties:

- Text: “Welcome to Geant4 Visualization”
- Position: (0.,0.,0.) in the world coordinates
- Horizontal offset: 10 pixels
- Vertical offset: -20 pixels
- Colour: blue (default)

Listing 8.8: An example of defining text.

```
//----- C++ source codes: An example of defining a visualizable text

//----- Instantiation
G4Text text ;
text.SetText ( "Welcome to Geant4 Visualization");
text.SetPosition ( G4Point3D(0.,0.,0.) );
// These three lines are equivalent to:
//  G4Text text ( "Welcome to Geant4 Visualization",
//                G4Point3D(0.,0.,0.) );

//----- Size (font size in units of pixels)
G4double fontsize = 24.; // Should be 24. * pixels - to be implemented.
text.SetScreenSize ( fontsize );

//----- Offsets
G4double x_offset = 10.; // Should be 10. * pixels - to be implemented.
G4double y_offset = -20.; // Should be -20. * pixels - to be implemented.
text.SetOffset( x_offset, y_offset );

//----- Color (Blue is the default setting, and so the codes below are omissible)
G4Colour blue( 0., 0., 1. );
G4VisAttributes att ( blue );
text.SetVisAttributes ( att );

//----- end of C++ source codes
```

8.10 Making a Movie

These instructions are suggestive only. The following procedures have not been tested on all platforms. There are clearly some instructions that apply only to Unix-like systems with an X-Windows based windowing system. However, it should not be difficult to take the ideas presented here and extend them to other platforms and systems.

The procedures described here need graphics drivers that can produce picture files that can be converted to a form suitable for an MPEG encoder. There may be other ways of capturing the screen images and we would be happy to hear about them. Graphics drivers currently capable of producing picture files are: More informations about MPEG encoder

Driver	**File type**
DAWNFILE	prim then eps using **dawn**
HepRepFile	HepRep1
HepRep	HepRep2
OGLX	jpeg, eps, pdf, ppm, . . .
Qt	jpeg, eps, pdf, ppm, . . .
RayTracer	jpeg
VRMLFILE	vrml

So far, only DAWNFILE, OGLX, OGLQt and RayTracer have been “road tested”. Once in a standard format, such as eps, the **convert** program from ImageMagick can convert to ppm files suitable for **ppmtompeg** available here: http://netpbm.sourceforge.net/

8.10.1 Using "/vis/viewer/interpolate".

By saving views with "/vis/viewer/save" (see *How to save a view.*) one may "fly through" them with "/vis/viewer/interpolate". One of the options in "/vis/viewer/interpolate" is to export image files (OpenGL and Qt only) see *OGLX* below) that may then be used to make a movie.

(Use the Geant4 "help" command to see details.)

For example, with iMovie (Apple Mac) one may import the created files (PDF recommended) and by reducing the cliplength to 0.1 s (that seems to be the minimum) one may make a reasonable movie.

8.10.2 OGLX

Make a macro something like this:

```
/control/verbose 2
/vis/open OGL 600x600-0+0
/vis/drawVolume
/vis/viewer/reset
/vis/viewer/set/style surface
/vis/viewer/set/projection perspective 50 deg
/control/alias phi 30
/control/loop movie.loop theta 0 360 1
```

which invokes movie.loop, which is something like:

```
/vis/viewer/set/viewpointThetaPhi {theta} {phi}
/vis/viewer/zoom 1.005
/vis/ogl/printEPS
```

This produces lots of eps files. Then. . .

```
make_mpeg2encode_parfile.sh G4OpenGL_*eps
```

Then edit mpeg2encode.par to specify file type and size, etc.:

```
$ diff mpeg2encode.par~ mpeg2encode.par
7c7
< 1         /* input picture file format: 0=*.Y,*.U,*.V, 1=*.yuv, 2=*.ppm */
---
> 2         /* input picture file format: 0=*.Y,*.U,*.V, 1=*.yuv, 2=*.ppm */
15,17c15,17
<    /* horizontal_size */
<   /* vertical_size */
< 8         /* aspect_ratio_information 1=square pel, 2=4:3, 3=16:9, 4=2.11:1 */
---
> 600   /* horizontal_size */
> 600  /* vertical_size */
> 1         /* aspect_ratio_information 1=square pel, 2=4:3, 3=16:9, 4=2.11:1 */
```

Then convert to ppm:

```
for i in G4OpenGL*eps;
  do j=`basename $i .eps`; command="convert $i $j.ppm"; echo $command; $command; done
```

Then

```
mpeg2encode mpeg2encode.par G4OpenGL.mpg
```

Then, on Mac, for example:

```
open G4OpenGL.mpg
```

opens a movie player.

8.10.3 Qt

The Qt driver provides one of the easiest ways to make a movie. Of course, you first need to add the Qt libraries and link with Qt, but once you have that, Qt provides a ready-made function to store all updates of the OpenGL frame into the movie format. You then use loops (as defined in OGLX section above) or even move/rotate/zoom you scene by mouse actions to form your movie.

The Qt driver automatically handles all of the movie-making steps described in the OGLX section of this document - storing the files, converting them and assembling the finished movie. You just have to take care of installing an mpeg_encoder.

To make a movie :

- Right click to display a context menu, "Action"-<"Movie parameters".
- Select MPEG encoder path if it was not found.
- Select the name of the output movie.
- Let go! Hit SPACE to Start/Pause recording, RETURN to STOP

Then, open your movie (on Mac, for example):

```
open G4OpenGL.mpg
```

opens a QuickTime player.

8.10.4 DAWNFILE

You need to invoke **dawn** in "direct" mode, which picks up parameters from .DAWN_1.history, and suppress the GUI:

```
alias dawn='dawn -d'
export DAWN_BATCH=1
```

Change OGL to DAWNFILE in the above set of Geant4 commands and run. Then convert to ppm files as above:

```
for i in g4_*.eps;
  do j=`basename $i .eps`; command="convert $i $j.ppm"; echo $command; $command; done
```

Then make a .par file:

```
make_mpeg2encode_parfile.sh g4_*ppm
```

and edit mpeg2encode.par:

```
$ diff mpeg2encode.par~ mpeg2encode.par
7c7
< 1         /* input picture file format: 0=*.Y,*.U,*.V, 1=*.yuv, 2=*.ppm */
---
> 2         /* input picture file format: 0=*.Y,*.U,*.V, 1=*.yuv, 2=*.ppm */
9c9
< 1         /* number of first frame */
---
> 0         /* number of first frame */
15,16c15,16
<    /* horizontal_size */
<   /* vertical_size */
```

```
---
> 482   /* horizontal_size */
> 730  /* vertical_size */
```

Then encode and play:

```
mpeg2encode mpeg2encode.par DAWN.mpg
open DAWN.mpg
```

8.10.5 RayTracerX

```
/control/verbose 2
/vis/open RayTracerX 600x600-0+0
# (Raytracer doesn't need a scene; smoother not to /vis/drawVolume.)
/vis/viewer/reset
/vis/viewer/set/style surface
/vis/viewer/set/projection perspective 50 deg
/control/alias phi 30
/control/loop movie.loop theta 0 360 1
```

where movie.loop is as above. This produces lots of jpeg files (but takes 3 days!!!). Then...

```
make_mpeg2encode_parfile.sh g4RayTracer*jpeg
```

Then edit mpeg2encode.par to specify file type and size, etc.:

```
$ diff mpeg2encode.par.orig mpeg2encode.par
7c7
< 1         /* input picture file format: 0=*.Y,*.U,*.V, 1=*.yuv, 2=*.ppm */
---
> 2         /* input picture file format: 0=*.Y,*.U,*.V, 1=*.yuv, 2=*.ppm */
15,17c15,17
<    /* horizontal_size */
<   /* vertical_size */
< 8         /* aspect_ratio_information 1=square pel, 2=4:3, 3=16:9, 4=2.11:1 */
---
> 600   /* horizontal_size */
> 600  /* vertical_size */
> 1         /* aspect_ratio_information 1=square pel, 2=4:3, 3=16:9, 4=2.11:1 */
```

Then convert to ppm, encode and play:

```
for i in g4*jpeg;
  do j=`basename $i .jpeg`; command="convert $i $j.ppm"; echo $command; $command; done
mpeg2encode mpeg2encode.par g4RayTracer.mpg
open g4RayTracer.mpg
```

CHAPTER

NINE

ANALYSIS

9.1 Introduction

The new analysis category based on g4tools was added in the Geant4 9.5 release with the aim to provide the users a "light" analysis tool available directly with Geant4 installation without a need to link their Geant4 application with an external analysis package. It consists of the analysis manager classes and it includes also the g4tools package.

g4tools provides code to write and read histograms and ntuples in several formats: ROOT, XML AIDA format and CSV (comma-separated values format). It is a part of inlib and exlib libraries, that include also other facilities like fitting and plotting.

The analysis classes provide a uniform, user-friendly interface to g4tools and hide the differences according to a selected output technology from the user. They take care of a higher-level management of the g4tools objects (files, histograms and ntuples), handle allocation and removal of the objects in memory and provide the access methods to them via indexes. They are fully integrated in the Geant4 framework: they follow Geant4 coding style and also implement the built-in Geant4 user interface commands that can be used by users to define or configure their analysis objects.

An example of use of analysis manager classes is provided in basic example B4, in the `B4RunAction` and `B4EventAction` classes.

9.2 Analysis Manager Classes

The analysis manager classes provide uniform interfaces to the g4tools package and hide the differences between use of g4tools classes for the supported output formats (ROOT, AIDA XML and CSV).

An analysis manager class is available for each supported output format:

- `G4CsvAnalysisManager`
- `G4RootAnalysisManager`
- `G4XmlAnalysisManager`

For a simplicity of use, each analysis maneger provides the complete access to all interfaced functions though it is implemented via a more complex design.

The managers are implemented as singletons. User code will access a pointer to a single instance of the desired manager. The manager has to be created and deleted from the user code. All objects created via analysis manager are deleted automatically with the manager. The concrete types of the analysis manager as well as the handled g4tools objects, are hidden behind a namespace which is selected by including a dedicated include file. This allows the user to use all output technologies in an identical way via these generic types:

- `G4AnalysisManger`: the public analysis interface
- `G4AnaH1[2,3]`: one[two,three]-dimensional histogram

- `G4AnaP1[2]`: one[two]-dimensional profile
- `G4Ntuple`: ntuple

In addition to the `G4AnalysisManager` functions, a set of Geant4 UI commands for creating histograms and setting their properties is implemented in associated messenger classes.

9.2.1 Analysis Manager

To use Geant4 analysis, an instance of the analysis manager must be created. The analysis manager object is created with the first call to `G4AnalysisManager::Instance()`, the next calls to this function will just provide the pointer to this analysis manager object. The client code is responsible for deleting the created object what is in our example done in the run action destructor.

The example of the code for creating the analysis manager extracted from the basic B4 example is given below:

```
#include "B4Analysis.hh"

B4RunAction::B4RunAction()
 : G4UserRunAction()
{
  // Create analysis manager
  auto analysisManager = G4AnalysisanalysisManagerager::Instance();
  analysisManager->SetVerboseLevel(1);
  analysisManager->SetFirstHistoId(1);
}

B4RunAction::~B4RunAction()
{
  delete G4AnalysisManager::Instance();
}
```

It is recommended, but not necessary, to create the analysis manager in the user run action constructor and delete it in its destructor. This guarantees correct behavior in multi-threading mode. The code specific to the output format is hidden in `B4Analysis.hh` where the selection of the output format takes place.

```
#ifndef B4Analysis_h
#define B4Analysis_h 1

#include "g4root.hh"
//#include "g4xml.hh"
//#include "g4csv.hh"
#endif
```

The level of informative printings can be set by `SetVerboseLevel(G4int)`. Currently the levels from 0 (default) up to 4 are supported.

The verbose level can be also set via the UI command:

```
/analysis/verbose level
```

9.2.2 Files handling

The analysis manager can handle only one base file at a time. Below we give an example of opening and closing a file extracted from the basic example B4:

```
#include "B4Analysis.hh"

void B4RunAction::BeginOfRunAction(const G4Run* run)
{
  // Get analysis manager
```

```
  auto analysisManager = G4AnalysisanalysisManagerager::Instance();

  // Open an output file
  analysisManager->OpenFile("B4");
}

void B4RunAction::EndOfRunAction(const G4Run* aRun)
{
  // Save histograms
  auto analysisManager = G4AnalysisanalysisManagerager::Instance();
  analysisManager->Write();
  analysisManager->CloseFile();
}
```

The following functions are defined for handling files:

```
G4bool OpenFile(const G4String& fileName = "");
G4bool Write();
G4bool CloseFile();
```

The file name can be defined either directly with `OpenFile(const G4String&)` call or separately via `SetFileName(const G4String&)` function before calling `OpenFile()`. It is not possible to change the file name when a file is open and not yet closed. If a file extension is not specified in `fileName`, it is automatically completed according to a selected output format.

The file can be optionally structured in sub-directories. Currently only one directory for histograms and/or one directory for ntuples are supported. The directories are created automatically if their names are set to non-empty string values via `SetHistoDirectoryName(const G4String&)` and/or `SetNtupleDirectoryName(const G4String&)`. This setting is ignored with the output formats which do not support this feature (XML, CSV).

The following commands for handling files and directories are available

```
/analysis/setFileName name           # Set name for the output file
/analysis/setHistoDirName name       # Set name for the histograms directory
/analysis/setNtupleDirName name      # Set name for the histograms directory
```

Depending on the selected output format more files can be generated when more than one ntuple is defined in a user application. This is the case of XML and CSV, which do not allow writing more than one ntuple in a file. The ntuple file name is then generated automatically from the base file name and the ntuple name.

The analysis manager can handle only one base file at a time, but several base files can be generated sucessively from Geant4 session, typically one file is saved per run. A new file can be open only after a previous file was closed. An example of generated more files per session is provided in basic/B5 example and its run2.mac macro. Appending existing files is not supported. When an existing file is open again, its content is overwritten.

9.2.3 Histograms

The code for handling histograms given in the following example is extracted the B4 example classes. In this example, the histograms are created in the run action constructor and they are filled in the end of event.

```
#include "B4Analysis.hh"

B4RunAction::B4RunAction()
 : G4UserRunAction()
{
  // Create analysis manager
  // ...

  // Creating histograms
  analysisManager->CreateH1("1","Edep in absorber", 100, 0., 800*MeV);
  analysisManager->CreateH1("2","Edep in gap", 100, 0., 100*MeV);
```

```
}

void B4aEventAction::EndOfEventAction(const G4Run* aRun)
{
  // Fill histograms
  auto analysisManager = G4AnalysisanalysisManagerager::Instance();
  analysisManager->FillH1(1, fEnergyAbs);
  analysisManager->FillH1(2, fEnergyGap);
}
```

Creating Histograms

A one-dimensional (1D) histogram can be created with one of these two `G4AnalysisManager` functions:

```
G4int CreateH1(const G4String& name, const G4String& title,
               G4int nbins, G4double xmin, G4double xmax,
               const G4String& unitName = "none",
               const G4String& fcnName = "none",
               const G4String& binSchemeName = "linear");

G4int CreateH1(const G4String& name, const G4String& title,
               const std::vector<G4double>& edges,
               const G4String& unitName = "none",
               const G4String& fcnName = "none");
```

where `name` and `title` parameters are self-descriptive. The histogram edgeds can be defined either via the `nbins`, `xmin` and `xmax` parameters (first function) representing the number of bins, the minimum and maximum histogram values, or via the `const std::vector<G4double>& edges` parameter (second function) representing the edges defined explicitly. The other parameters in both functions are optional and their meaning is explained in *Histograms Properties*.

Two-dimensional (2D) and three-dimensional (3D) histograms can be created with one of these two functions analogous to those for 1D histograms:

```
G4int CreateH2(const G4String& name, const G4String& title,
               G4int nxbins, G4double xmin, G4double xmax,
               G4int nybins, G4double ymin, G4double ymax,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none",
               const G4String& xbinScheme = "linear",
               const G4String& ybinScheme = "linear");

G4int CreateH2(const G4String& name, const G4String& title,
               const std::vector<G4double>& xedges,
               const std::vector<G4double>& yedges,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none");
```

```
G4int CreateH3(const G4String& name, const G4String& title,
               G4int nxbins, G4double xmin, G4double xmax,
               G4int nybins, G4double ymin, G4double ymax,
               G4int nzbins, G4double zmin, G4double zmax,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& zunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none",
               const G4String& zfcnName = "none",
```

```
                const G4String& xbinSchemeName = "linear",
                const G4String& ybinSchemeName = "linear",
                const G4String& zbinSchemeName = "linear");

G4int CreateH3(const G4String& name, const G4String& title,
                const std::vector<G4double>& xedges,
                const std::vector<G4double>& yedges,
                const std::vector<G4double>& zedges,
                const G4String& xunitName = "none",
                const G4String& yunitName = "none",
                const G4String& zunitName = "none",
                const G4String& xfcnName = "none",
                const G4String& yfcnName = "none",
                const G4String& zfcnName = "none");
```

The meaning of parameters is the same as in the functions for 1D histograms, they are just applied in x, y and z dimensions.

The histograms created with `G4AnalysisManager` get automatically attributed an integer identifier which value is returned from the "Create" function. The default start value is 0 and it is incremented by 1 for each next created histogram. The numbering of 2D and 3D histograms is independent from 1D histograms and so the first created 2D (or 3D) histogram identifier is equal to the start value even when several 1D histograms have been already created.

The start histogram identifier value can be changed either with the `SetFirstHistoId(G4int)` method, which applies the new value to all histogram types, or with the `SetFirstHNId(G4int)`, where `N = 1, 2, 3` methods, which apply the new value only to the relevant histogram type. The first method is demonstrated in the example.

The histogram names "1", "2" in the demonstrated example are defined to correspond the histograms identifiers in a similar way as in `extended/analysis/AnaEx01` example. This choice is however fully in hands of the user who can prefer longer and more meaningful names.

All histograms created by `G4AnalysisManager` are automatically deleted with deleting the `G4AnalysisManager` object.

Histograms can be also created via UI commands. The commands to create 1D histogram:

```
/analysis/h1/create               # Create 1D histogram
   name title [nbin min max] [unit] [fcn] [binScheme]
```

The commands to create 2D histogram:

```
/analysis/h2/create               # Create 2D histogram
  name title [nxbin xmin xmax xunit xfcn xbinScheme nybin ymin ymax yunit yfcn yBinScheme]
```

The commands to create 3D histogram:

```
/analysis/h3/create               # Create 3D histogram
  name title [nxbin xmin xmax xunit xfcn xbinScheme nybin ymin ymax
  yunit yfcn yBinScheme nzbin zmin zmax zunit zfcn zBinScheme]
```

Configuring Histograms

The properties of already created histograms can be changed with use of one of these two functions sets. For 1D histograms:

```
G4bool SetH1(G4int id,
             G4int nbins, G4double xmin, G4double xmax,
             const G4String& unitName = "none",
             const G4String& fcnName = "none",
             const G4String& binSchemeName = "linear");
```

```
G4bool SetH1(G4int id,
             const std::vector<G4double>& edges,
             const G4String& unitName = "none",
             const G4String& fcnName = "none");
```

for 2D histograms:

```
G4bool SetH2(G4int id,
             G4int nxbins, G4double xmin, G4double xmax,
             G4int nybins, G4double ymin, G4double ymax,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none",
             const G4String& xbinSchemeName = "linear",
             const G4String& ybinSchemeName = "linear");

G4bool SetH2(G4int id,
             const std::vector<G4double>& xedges,
             const std::vector<G4double>& yedges,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none");
```

and for 3D histograms:

```
G4bool SetH3(G4int id,
             G4int nxbins, G4double xmin, G4double xmax,
             G4int nzbins, G4double zmin, G4double zmax,
             G4int nybins, G4double ymin, G4double ymax,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& zunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none",
             const G4String& zfcnName = "none",
             const G4String& xbinSchemeName = "linear",
             const G4String& ybinSchemeName = "linear",
             const G4String& zbinSchemeName = "linear");

G4bool SetH3(G4int id,
             const std::vector<G4double>& xedges,
             const std::vector<G4double>& yedges,
             const std::vector<G4double>& zedges,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& zunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none",
             const G4String& zfcnName = "none");
```

The histogram is accessed via its integer identifier. The meaning of the other parameters is the same as in "Create" functions.

Histogram properties can be also defined via UI commands. The commands to define 1D histogram

```
/analysis/h1/set id nbin min max [unit] [fcn] [binScheme] # Set parameters
```

The commands to define 2D histogram:

```
# Set parameters for the 2D histogram of #id
/analysis/h2/set
  id nxbin xmin xmax xunit xfcn xbinScheme nybin ymin ymax yunit yfcn yBinScheme
```

```
# Set parameters per dimension
/analysis/h2/setX id nbin min max [unit] [fcn] [binScheme] # Set x-parameters
/analysis/h2/setY id nbin min max [unit] [fcn] [binScheme] # Set y-parameters
```

The commands to define 3D histogram:

```
# Set parameters for the 3D histogram of #id
/analysis/h3/set =
  id nxbin xmin xmax xunit xfcn xbinScheme nybin ymin ymax yunit yfcn yBinScheme nzbin zmin zmax␣
↪zunit zfcn zBinScheme

# Set parameters per dimension
/analysis/h3/setX id nbin min max [unit] [fcn] [binScheme] # Set x-parameters
/analysis/h3/setY id nbin min max [unit] [fcn] [binScheme] # Set y-parameters
/analysis/h3/setY id nbin min max [unit] [fcn] [binScheme] # Set z-parameters
```

A limited set of parameters for histograms plotting, the histogram and the histogram axis titles, can be also defined via functions

```
G4bool SetH1Title(G4int id, const G4String& title);
G4bool SetH1XAxisTitle(G4int id, const G4String& title);
G4bool SetH1YAxisTitle(G4int id, const G4String& title);
//
G4bool SetH2Title(G4int id, const G4String& title);
G4bool SetH2XAxisTitle(G4int id, const G4String& title);
G4bool SetH2YAxisTitle(G4int id, const G4String& title);
G4bool SetH2ZAxisTitle(G4int id, const G4String& title);
//
G4bool SetH3Title(G4int id, const G4String& title);
G4bool SetH3XAxisTitle(G4int id, const G4String& title);
G4bool SetH3YAxisTitle(G4int id, const G4String& title);
G4bool SetH3ZAxisTitle(G4int id, const G4String& title);
```

The corresponding UI commands

```
/analysis/h1/setTitle id title                 # Set title for the 1D histogram of #id
/analysis/h1/setXaxis id title                 # Set x-axis title for the 1D histogram
/analysis/h1/setYaxis id title                 # Set y-axis title for the 1D histogram
```

The same set of commands is available for the other histogram types and profiles, under the appropriate directory.

Filling Histograms

The histogram values can be filled using the functions

```
G4bool FillH1(G4int id, G4double value,
              G4double weight = 1.0);
G4bool FillH2(G4int id, G4double xvalue, G4double yvalue,
              G4double weight = 1.0);
G4bool FillH3(G4int id,
              G4double xvalue, G4double yvalue, G4double zvalue,
              G4double weight = 1.0);
```

where the weight can be given optionally.

The histograms can be also scaled with a given factor using the functions:

```
G4bool ScaleH1(G4int id, G4double factor);
G4bool ScaleH2(G4int id, G4double factor);
G4bool ScaleH3(G4int id, G4double factor);
```

Accessing Histograms

Besides the fast access to histograms via their integer identifiers, the histograms can be also accessed by their names using the `G4AnalysisManager` function providing the conversion from a name in a histogram identifier:

```
G4int GetH1Id(const G4String& name, G4bool warn = true) const;
G4int GetH2Id(const G4String& name, G4bool warn = true) const;
G4int GetH3Id(const G4String& name, G4bool warn = true) const;
```

If a histogram with a given name is not found, a warning is issued unless it is explicitly disabled by the user. This way is however less efficient and it is not recommended for frequently called functions as e.g. `Fill()`.

The analysis manager provides also the direct access to the g4tools histogram objects. The concrete histogram type is hidden behind a selected namespace. In example B4, the g4tools histogram functions `mean()` and `rms()` are called:

```
auto analysisManager = G4AnalysisanalysisManagerager::Instance();
if ( analysisManager->GetH1(1) ) {
  G4cout << "\n ----> print histograms statistic \n" << G4endl;
  G4cout << " EAbs : mean = " << analysisManager->GetH1(1)->mean()
         << " rms = " << analysisManager->GetH1(1)->rms(),
         << G4endl;
  // ...
}
```

Activation of Histograms

The activation option allows the user to activate only selected histograms. When this option is activated, only the histograms marked as activated are returned, filled or saved in a file. This feature is intensively used in extended/electromagnetic examples where all histograms are first created inactivated:

```
auto analysisManager = G4AnalysisanalysisManagerager::Instance();
analysisManager->SetActivation(true);
// define histogram parameters name, title, nbins, vmin, vmax
G4int id = analysisManager->CreateH1(name, title, nbins, vmin, vmax);
analysisManager->SetH1Activation(id, false);
```

and then selected histograms are activated in macros, using the analysis "set" command

```
/analysis/h1/set 1 100 0   50 cm        #track length of primary
/analysis/h1/set 2 100 0  300 none      #nb steps of primary
```

The activation option is not switched on by default. It has to be activated either via analysisManager `SetActivation(true)` call as above or via the UI command:

```
/analysis/setActivation true|false   # Set activation option
```

When no parameters need to be changed a histogram can be activated using "setActivation" command:

```
/analysis/h1/setActivation id true|false     # Set activation to histogram #id
/analysis/h1/setActivationToAll  true|false  # Set activation to all 1D histograms.
```

Histograms Properties

The following properties, additional to those defined in g4tools, can be added to histograms via `G4AnalysisManager`:

- *Unit*: if a histogram is defined with a unit, all filled values are automatically converted to this defined unit and the unit is added to the histogram axis title.

- *Function*: if a histogram is defined with a function, the function is automatically executed on the filled values and its name is added to the histogram axis title. When a histogram is defined with both unit and function the unit is applied first. The available functions: `log`, `log10`, `exp`.
- *Binning scheme*: user can select logarithmic binning scheme besides the linear one (default). The available binning schemes: `linear`, `log`.
- *Activation*: see previous section.
- *ASCII option*: if activated the histogram is also printed in an ASCII file when Write() function is called.
- *Plotting option*: if activated the histogram is plotted in a file of Postscript format when Write() function is called. See more details in *Plotting*.

9.2.4 Profiles

Profile histograms (profiles) are used to display the mean value of Y and its error for each bin in X. The displayed error is by default the standard error on the mean (i.e. the standard deviation divided by the sqrt(n).) An example of use of 1D profiles can be found in `extended/electromagnetic/TestEm2`. Though the functions for creating and manipulating profiles are very similar to those for histograms, they are described in this section.

Creating Profiles

A one-dimensional (1D) profile can be created with one of these two `G4AnalysisManager` functions

```
G4int CreateP1(const G4String& name, const G4String& title,
               G4int nbins, G4double xmin, G4double xmax,
               G4double ymin = 0, G4double ymax = 0,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none",
               const G4String& xbinSchemeName = "linear");

G4int CreateP1(const G4String& name, const G4String& title,
               const std::vector<G4double>& edges,
               G4double ymin = 0, G4double ymax = 0,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none");
```

where `name` and `title` parameters are self-descriptive. The profile edgeds can be defined either via the `nbins`, `xmin` and `xmax` parameters (first function) representing the number of bins, the minimum and maximum profile values, or via the `const std::vector<G4double>& edges` parameter (second function) representing the edges defined explicitly. If `ymin` and `ymax` parameters are provides, only values between these limnits will be considered at filling time. The other parameters in both functions are optional and their meaning is explained in *Profiles Properties*.

A two-dimensional (2D) profile can be created with one of these two functions analogous to those for 1D profiles:

```
G4int CreateP2(const G4String& name, const G4String& title,
               G4int nxbins, G4double xmin, G4double xmax,
               G4int nybins, G4double ymin, G4double ymax,
               G4double zmin = 0, G4double zmax = 0,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& zunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none",
               const G4String& zfcnName = "none",
               const G4String& xbinSchemeName = "linear",
               const G4String& ybinSchemeName = "linear");
```

```
G4int CreateP2(const G4String& name, const G4String& title,
               const std::vector<G4double>& xedges,
               const std::vector<G4double>& yedges,
               G4double zmin = 0, G4double zmax = 0,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& zunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none",
               const G4String& zfcnName = "none");
```

The meaning of parameters is the same as in the functions for 1D profiles, they are just applied in x, y and z dimensions.

The profiles created with `G4AnalysisManager` get automatically attributed an integer identifier which value is returned from the "Create" function. The default start value is 0 and it is incremented by 1 for each next created profile. The numbering of 2D profiles is independent from 1D profiles and so the first created 2D profile identifier is equal to the start value even when several 1D profiles have been already created.

The start profile identifier value can be changed either with the `SetFirstProfileId(G4int)` method, which applies the new value to both 1D and 2D profile types, or with the `SetFirstPNId(G4int)`, where `N = 1, 2` methods, which apply the new value only to the relevant profile type.

All profiles created by `G4AnalysisManager` are automatically deleted with deleting the `G4AnalysisManager` object.

Profiles can be also created via UI commands. The commands to create 1D profile

```
/analysis/p1/create               # Create 1D profile
  name title [nxbin xmin xmax xunit xfcn xbinScheme ymin ymax yunit yfcn]
```

The commands to create 2D profile:

```
/analysis/p2/create               # Create 2D profile
  name title [nxbin xmin xmax xunit xfcn xbinScheme nybin ymin ymax yunit yfcn yBinScheme zmin␣
↪zmax zunit zfcn]
```

Configuring Profiles

The properties of already created profiles can be changed with use of one of these two functions sets. For 1D profiles

```
G4bool SetP1(G4int id,
             G4int nbins, G4double xmin, G4double xmax,
             G4double ymin = 0, G4double ymax = 0,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none",
             const G4String& xbinSchemeName = "linear");

G4bool SetP1(G4int id,
             const std::vector<G4double>& edges,
             G4double ymin = 0, G4double ymax = 0,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none");
```

and for 2D profiles:

```
G4bool SetP2(G4int id,
             G4int nxbins, G4double xmin, G4double xmax,
             G4int nybins, G4double ymin, G4double ymax,
             G4double zmin = 0, G4double zmax = 0,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& zunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none",
             const G4String& zfcnName = "none",
             const G4String& xbinSchemeName = "linear",
             const G4String& ybinSchemeName = "linear");

G4bool SetP2(G4int id,
             const std::vector<G4double>& xedges,
             const std::vector<G4double>& yedges,
             G4double zmin = 0, G4double zmax = 0,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& zunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none",
             const G4String& zfcnName = "none");
```

The profile is accessed via its integer identifier. The meaning of the other parameters is the same as in "Create" functions.

Profiles properties can be also defined via UI commands. The commands to define 1D profile

```
/analysis/p1/set                  # Set parameters for the 1D histogram of #id
  id nxbin xmin xmax xunit xfcn xbinScheme ymin ymax yunit yfcn
```

The commands to create or define 2D profile:

```
/analysis/p2/set                  # Set parameters for the 2D profile of #id
  id nxbin xmin xmax xunit xfcn xbinScheme nybin ymin ymax yunit yfcn yBinScheme zmin zmax zunit␣
↪zfcn
```

A limited set of parameters for profiles plotting, the profile and the profile axis titles, can be also defined via functions

```
G4bool SetP1Title(G4int id, const G4String& title);
G4bool SetP1XAxisTitle(G4int id, const G4String& title);
G4bool SetP1YAxisTitle(G4int id, const G4String& title);
//
G4bool SetP2Title(G4int id, const G4String& title);
G4bool SetP2XAxisTitle(G4int id, const G4String& title);
G4bool SetP2YAxisTitle(G4int id, const G4String& title);
G4bool SetP2ZAxisTitle(G4int id, const G4String& title);
```

The parameters can be also set via the same set of UI commands as the histogram parameters available under the appropriate directory.

Filling Profiles

The profile values can be filled using the functions:

```
G4bool FillP1(G4int id,
              G4double xvalue, G4double yvalue,
              G4double weight = 1.0);
G4bool FillP2(G4int id,
              G4double xvalue, G4double yvalue, G4double zvalue,
              G4double weight = 1.0);
```

where the weight can be given optionally.

The profiles can be also scaled with a given factor using the functions:

```
G4bool ScaleP1(G4int id, G4double factor);
G4bool ScaleP2(G4int id, G4double factor);
```

Profiles Properties

All histogram features described in sections "Accessing Histograms", "Activation of Histograms" and "Histograms Properties" (*Accessing Histograms*, *Activation of Histograms* and *Histograms Properties*) are also available for profiles.

9.2.5 Plotting

Since Geant4 10.2 version it is possible to produce a graphics output file in the Postscript format containing selected histograms and profiles. Histograms and profiles plotting can be activated using G4AnalysisManager functions:

```
auto analysisManager = G4AnalysisanalysisManagerager::Instance();
analysisManager->SetH1Plotting(id, true);
   // etc for H2, H3, P1, P2
```

or using the UI commands

```
/analysis/h1/setPlotting id true|false     # (In)Activate plottig for 1D histogram #id
/analysis/h1/setPlottingToAll true|false   # (In)Activate plottig for all 1D histograms.
       # etc. for h2, h3, p1, p2
```

If Geant4 libraries are built with support for Freetype font rendering, user can choose from three plotting styles:

- ROOT_default: ROOT style with high resolution fonts (default)
- hippodraw: hippodraw style with high resolution fonts
- inlib_default: PAW style with low resolution fonts")

otherwise only the `inlib_default` style with low resolution fonts is available.

The page size of the graphics output is fixed to A4 format. Users can choose the page layout which is defined by the number columns and the number of rows in a page. Depending on the selected plotting style, the maximum number of plots is limited to 3 columns x 5 rows for the styles with high resolution fonts and to 2 columns x 3 rows for the `inlib_default` style.

Finally, users can also customize the plot dimensions, which represent the plotter window size (width and height) in pixels.

The customization of the plotting can be done via the UI commands in `/analysis/plot` directory:

```
/analysis/plot/setStyle  styleName
/analysis/plot/setLayout  columns rows
/analysis/plot/setDimensions  width height
```

Opening more configuration parameters for users customisation can be considered in future according to the users feedback.

9.2.6 Ntuples

In the following example the code for handling ntuples extracted from basic example B4, from the `B4RunAction` and `B4EventAction` classes, is presented.

```
#include "B4Analysis.hh"

B4RunAction::B4RunAction()
 : G4UserRunAction()
{
  // Create analysis manager
  // ...

  // Create ntuple
  man->CreateNtuple("B4", "Edep and TrackL");
  man->CreateNtupleDColumn("Eabs");
  man->CreateNtupleDColumn("Egap");
  man->FinishNtuple();
}

void B4EventAction::EndOfEventAction(const G4Run* aRun)
{
  G4AnalysisManager* man = G4AnalysisManager::Instance();
  man->FillNtupleDColumn(0, fEnergyAbs);
  man->FillNtupleDColumn(1, fEnergyGap);
  man->AddNtupleRow();
}
```

Since 10.0 release, there is no limitation for the number of ntuples that can be handled by `G4AnalysisManager`. Handling of two ntuples is demostrated in extended analysis/AnaEx01 example.

Creating Ntuples

An ntuple can be created using the following set of functions:

```
G4int CreateNtuple(const G4String& name, const G4String& title);

// Create columns in the last created ntuple
G4int CreateNtupleXColumn(const G4String& name);
void  FinishNtuple();

// Create columns in the ntuple with given id
G4int CreateNtupleXColumn(G4int ntupleId, const G4String& name);
void  FinishNtuple(G4int ntupleId);
```

The first set is demonstrated in the example. The columns can take the values of `G4int`, `G4float`, `G4double` or `G4String` type which is also reflected in the `CreateNtupleXColumn()` function names. where `X` can be `I, F, D or S`.

It is also possible to define ntuple columns of `std::vector` of `G4int`, `G4float` or `G4double` values using the functions:

```
// Create columns of vector in the last created ntuple
G4int CreateNtupleXColumn(
        const G4String& name, std::vector<Xtype>& vector);

// Create columns of vector in the ntuple with given id
G4int CreateNtupleXColumn(G4int ntupleId,
        const G4String& name, std::vector<Xtype>& vector);
```

where `[X, Xtype]` can be `[I, G4int]`, `[F, G4float]` or `[D, G4double]`.

When all ntuple columns are created, the ntuple has to be closed using `FinishNtuple()` function.

The ntuples created with `G4AnalysisManager` get automatically attributed an integer identifier which value is returned from the "Create" function. The default start value is 0 and it is incremented by 1 for each next created ntuple. The start ntuple identifier value can be changed with the `SetFirstNtupleId(G4int)` function.

The integer identifiers are also attributed to the ntuple columns. The numbering of ntuple columns is independent for each ntuple, the identifier default start value is 0 and it is incremented by 1 for each next created column regardless its type (*I*, *F*, *D* or *S*). (If the third ntuple column of a different type than `double` (`int` or `float`) is created in the demonstrated example, its identifier will have the value equal 2.) The start ntuple column identifier value can be changed with the `SetFirstNtupleColumnId(G4int)` function.

When calls to `CreateNtuple-Column()` and `FinishNtuple()` succeed the call to `CreateNtuple()`, the `ntupleId` argument need not to be specified even when creating several ntuples. However this order is not enforced and the second set of functions with `ntupleId` argument is provided to allow the user to create the ntuples and their columns in whatever order.

All ntuples and ntuple columns created by `G4AnalysisManager` are automatically deleted with deleting the `G4AnalysisManager` object.

Filling Ntuples

The ntuple values can be filled using the functions:

```
// Methods for ntuple with id = FirstNtupleId
G4bool FillNtupleIColumn(G4int id, G4int value);
G4bool FillNtupleFColumn(G4int id, G4float value);
G4bool FillNtupleDColumn(G4int id, G4double value);
G4bool FillNtupleSColumn(G4int id, const G4String& value);
G4bool AddNtupleRow();

// Methods for ntuple with id > FirstNtupleId (when more ntuples exist)
G4bool FillNtupleIColumn(G4int ntupleId, G4int columnId, G4int value);
G4bool FillNtupleFColumn(G4int ntupleId, G4int columnId, G4float value);
G4bool FillNtupleDColumn(G4int ntupleId, G4int columnId, G4double value);
G4bool FillNtupleSColumn(G4int ntupleId, G4int id, const G4String& value);
G4bool AddNtupleRow(G4int ntupleId);
```

If only one ntuple is defined in the user application, the ntuple identifier, `ntupleId`, need not to be specified and the first set can be used. The second set of functions has to be used otherwise. When all ntuple columns are filled, the ntuple fill has to be closed by calling `AddNtupleRow()`.

Accessing Ntuples

The ntuples g4tools objects can be accessed by their identifier. The concrete ntuple type is hidden behind a selected namespace:

```
auto analysisManager = G4AnalysisanalysisManagerager::Instance();
// If only one ntuple is defined
G4Ntuple* ntuple = analysisManager->GetNtuple();
// If more ntuples
G4int ntuple id = ...;
G4Ntuple* ntuple = analysisManager->GetNtuple(ntupleId);
```

9.2.7 Parallel Processing

As well as all other Geant4 categories, the analysis code has been adapted for multi-threading. In multi-threading mode, the analysis manager instances are internally created on the master and thread workers and data accounting is processed in parallel on workers threads.

Histograms produced on thread workers are automatically merged on `Write()` call and the result is written in a master file. Merging is protected by a mutex locking, using `G4AutoLock` utility.

Ntuples produced on thread workers are, by default, written on separate files, which names are generated automatically from a base file name, a thread identifier and eventually also an ntuple name. Since Geant4 version 10.3 it is possible to activate merging of ntuples with ROOT output type:

```
auto analysisManager = G4AnalysisManager::Instance();
analysisManager->SetNtupleMerging(true);
```

The ntuples produced on workers will be then progressively being merged to the main ntuples on the master. By default, the ntuples are written at the same file as the final histograms. Users can also select merging in a given number of files or change the default basket size value (32000):

```
auto analysisManager = G4AnalysisManager::Instance();
G4int nofReducedNtupleFiles = 2;
G4int basketSize = 64000;
analysisManager->SetNtupleMerging(true, nofReducedNtupleFiles, basketSize);
```

No merging of ntuples is provided with CSV and AIDA XML formats.

No changes are required in the user client analysis code for migration to multi-threading. It is however recommended to instantiate and delete the analysis manager in the user run action constructor and destructor respectively. The master instance is necessary when histograms are used in the user application or if merging ntuples is selected (available only with ROOT output); in case only ntuples are in use and merging is not activated, the master instance need not to be created.

To simplify the scaling of a Geant4 application across nodes on a cluster Geant4 provides the support of MPI. In particular it is possible to run a hybrid MPI/MT application that uses MPI to scale across nodes and MT to scale across cores. This is demonstrated in the extended example `parallel/MPI/exMPI03` which includes usage of Geant4 analysis.

9.2.8 Coexistence of Several Managers

The specific manager classes are singletons and so it is not possible to create more than one instance of an analysis manager of one type, eg. `G4RootAnalysisManager`. However two analysis manager objects of different types can coexist. Then instead of the generic `G4AnalysisManager` typedef the concrete type of each manager has to be given explicitly.

```
#include "G4CsvAnalysisManager.hh"
#include "G4XmlAnalysisManager.hh"

G4CsvAnalysisManager* csvManager = G4CsvAnalysisManager::Instance();
G4XmlAnalysisManager* xmlManager = G4XmlAnalysisManager::Instance();
```

Or:

```
#include "g4csv_defs.hh"
#include "g4xml_defs.hh"

G4Csv::G4AnalysisManager* csvManager = G4Csv::G4AnalysisManager::Instance();
G4Xml::G4AnalysisManager* xmlManager = G4Xml::G4AnalysisManager::Instance();
```

9.2.9 Supported Features and Limitations

The analysis category based on g4tools is provided with certain limitations that can be reduced according to the feedback from Geant4 users and developers.

Below is a summary of currently supported features in Root, Csv and Xml manager classes:

- Histogram types: 1D, 2D, 3D of `double`

- Profile types: 1D, 2D of `double`
- Ntuple column types: `int`, `float`, `double`, `G4String`, `std::vector<int>`, `std::vector<float>`, `std::vector<double>`
- Optional directory structure limited to one directory for histograms and/or one for ntuples

9.3 Analysis Reader Classes

The analysis reader classes allow to read in g4analysis objects from the files generated by the analysis manager(s) during processing Geant4 application.

An analysis reader class is available for each supported output format:

- `G4CsvAnalysisReader`
- `G4RootAnalysisReader`
- `G4XmlAnalysisReader`

For a simplicity of use, each analysis manager provides the complete access to all interfaced functions though it is implemented via a more complex design.

The readers are implemented as singletons. User code will access a pointer to a single instance of the desired reader object. The reader has to be created and deleted from the user code. All objects created via analysis reader are deleted automatically with the manager. The concrete types of the analysis reader as well as the handled g4tools objects, are hidden behind a namespace which is selected by including a dedicated include file. This allows the user to use all output technologies in an identical way via these generic types:

- `G4AnalysisReader`: the public reader interface
- `G4AnaH1[2,3]`: one[two,three]-dimensional histogram
- `G4AnaP1[2]`: one[two]-dimensional profile
- `G4RNtuple`: read ntuple

While the histograms and profiles objects handled by the analysis reader are of the same type as those handled by the analysis manager, the redaer's ntuple type is different.

All objects read with `G4AnalysisReader` (histograms, profiles and ntuples) get automatically attributed an integer identifier which value is returned from the "Read" ot "GetNtuple" function. The default start value is 0 and it is incremented by 1 for each next created object. The numbering each object type is independent from other objects types and also from the numbering of the same object type in analysis manager. The start identifier value can be changed in the same way as with the analysis manager (see *Creating Histograms*).

The read objects can be accessed in the analysis reader via their integer identifiers or by their names in the same way as in the analysis manager (see *Accessing Histograms*). Note that the type of read ntuple is different from the ntuple type in the analysis manager.

The specific manager classes are singletons and so it is not possible to create more than one instance of an analysis reader of one type, eg. `G4RootAnalysisReader`. However two analysis reader objects of different types can coexist. Then instead of the generic `G4AnalysisReader` typedef the concrete type of each manager has to be given explicitly in a similar way as for the analysis managers (see *Coexistence of Several Managers*).

As well as all other Geant4 categories, the analysis code has been adapted for multi-threading. In multi-threading mode, the analysis reader instances are internally created on the master or thread workers, depending on the client code call, and data reading can be processed in parallel on workers threads.

9.3.1 Analysis Reader

For reading in the output files created with `G4AnalysisManager`, an instance of the analysis reader must be created. The analysis reader object is created with the first call to `G4AnalysisReader::Instance()`, the next

calls to this function will just provide the pointer to this analysis manager object. The client code is responsible for deleting the created object.

The example of the code for creating the analysis reader is given below:

```
#include "g4root.hh"
//#include "g4csv.hh"
//#include "g4xml.hh"

// Create (or get) analysis reader
G4AnalysisReader* analysisReader = G4AnalysisReader::Instance();
analysisReader->SetVerboseLevel(1);

// code to read data

// Delete analysis reader
delete G4AnalysisReader::Instance();
```

The level of informative printings can be set by `SetVerboseLevel(G4int)`. Currently the levels from 0 (default) up to 4 are supported.

9.3.2 Files handling

The name of file to be read can be specified either via `G4AnalysisReader::SetFileName()` function, or directly when reading an object. It is possible to change the base file name at any time. The analysis reader can handle more than one file at same time.

```
G4AnalysisReader* analysisReader = G4AnalysisReader::Instance();
// Define a base file name
analysisReader->SetFileName("MyFileName");
```

The following functions are defined for handling files:

```
void SetFileName(const G4String& fileName);
G4String GetFileName() const;
```

A file is open only when any "Read" function is called. When more objects are read from the same file (Xml, Root), the file is open only once. When reading an object without specifying the file name explicitly in "Read" call, the object is searched in all open files in the order of their creation time.

9.3.3 Histograms and Profiles

In the following example the code for reading an histogram is presented.

```
// Code to create (or get) analysis reader
G4AnalysisReader* analysisReader = G4AnalysisReader::Instance();

// Define a base file name
analysisReader->SetFileName("MyFileName");

// Read 1D histogram of "Edep" name
G4int h1Id = analysisReader->ReadH1("Edep");
if ( h1Id >= 0 ) {
  G4H1* h1 = analysisReader->GetH1(h1Id);
  if ( h1 ) {
    G4cout << "   H1: "
           << "   mean: " << h1->mean() << " rms: " << h1->rms() << G4endl;
  }
}
```

```
// Delete analysis reader
delete G4AnalysisReader::Instance();
```

The histograms and profiles can be read with these `G4AnalysisReader` functions:

```
G4int ReadH1(const G4String& h1Name, const G4String& fileName = "");
G4int ReadH2(const G4String& h2Name, const G4String& fileName = "");
G4int ReadH3(const G4String& h3Name, const G4String& fileName = "");
G4int ReadP1(const G4String& h1Name, const G4String& fileName = "");
G4int ReadP2(const G4String& h2Name, const G4String& fileName = "");
```

where `hNname` is the name of the object to be read from a file. The file name can be defined explicitly for each reading object.

All histograms and profiles created by `G4AnalysisReader` are automatically deleted with deleting the `G4AnalysisReader` object.

9.3.4 Ntuples

In the following example the code for reading ntuples is presented.

```
// Code to create (or get) analysis reader
G4AnalysisReader* analysisReader = G4AnalysisReader::Instance();

// Define a base file name
analysisReader->SetFileName("MyFileName");

// Read ntuple
G4int ntupleId = analysisReader->GetNtuple("TrackL");;
if ( ntupleId >= 0 ) {
  G4double trackL;
  analysisReader->SetNtupleDColumn("Labs", trackL);
  G4cout << "Ntuple TrackL, reading selected column Labs" << G4endl;
  while ( analysisReader->GetNtupleRow() ) {
      G4cout << counter++ << "th entry: "
             << "  TrackL: " << trackL << std::endl;
  }
}

// Delete analysis reader
delete G4AnalysisReader::Instance();
```

When the ntuple columns are associated with the variables of the appropriate type, the ntuple they can be read in a loop with `GetNtupleRow()` function. The function returns true until all data are read in.

An overview of all available functions for ntuple reading is given below:

```
// Methods to read ntuple from a file
G4int GetNtuple(const G4String& ntupleName, const G4String& fileName = "");

// Methods for ntuple with id = FirstNtupleId
G4bool SetNtupleXColumn(const G4String& columnName, Xtype& value);
G4bool SetNtupleXColumn(const G4String& columnName, std::vector<Xtype>& vector);
G4bool GetNtupleRow();

// Methods for ntuple with id > FirstNtupleId
G4bool SetNtupleXColumn(G4int ntupleId,
                        const G4String& columnName, Xtype& value);
G4bool SetNtupleXColumn(G4int ntupleId,
                        const G4String& columnName, std::vector<Xtype>& vector);
G4bool GetNtupleRow(G4int ntupleId);
```

where [X, Xtype] in SetNtupleXColumn() can be [I, G4int], [F, G4float], [D, G4double] or [S, G4String]. The columns of std::vector type are not supported for G4String.

All ntuples and ntuple columns created by G4AnalysisReader are automatically deleted with deleting the G4AnalysisReader object.

9.4 Accumulables

The classes for users accumulables management were added in 10.2 release for the purpose of simplification of users application code. The accumulables objects are named variables registered to the accumulable manager, which provides the acces to them by name and performs their merging in multi-threading mode according to their defined merge mode. Their usage is demonstrated in the basic examples B1 and B3a.

To better reflect the meaning of these objects, the classes base name "Parameter" used in 10.2 was changed in "Accumulable" in 10.3. Further integration in the Geant4 framework is foreseen in the next Geant4 versions.

9.4.1 G4Accumulable<T>

G4Accumulable<T> templated class can be used instead of built-in types in order to facilitate merging of the values accumulated on workers to the master thread. The G4Accumulable<T> object has, besides its value of the templated type T, also a name, the initial value, which the value is set to in Reset() function and a merge mode, specifying the operation which is performed in Merge() function.

The accumulable object can be either instatiated using its constructor and registerd in G4AccumulablesManager explicitly, or it can be created using G4AccumulablesManager::CreateAccumulable() function, their registering is then automatic. The first way is used in the basic examples B1 and B3a:

```
// B1RunAction.hh
class B1RunAction : public G4UserRunAction
{
  // ...
  private:
    G4Accumulable<G4double> fEdep;
    G4Accumulable<G4double> fEdep2;
};

// B1RunAction.cc
B1RunAction::B1RunAction()
: G4UserRunAction(),
  fEdep("Edep", 0.),
  fEdep2("Edep2", 0.)    // the accumulable is initialized with a name and a value = initValue
                         // (the name can be omitted)
{
  // ..
  // Register accumulable to the accumulable manager
  G4AccumulableManager* accumulableManager = G4AccumulableManager::Instance();
  accumulableManager->RegisterAccumulable(fEdep);
  accumulableManager->RegisterAccumulable(fEdep2);
}
```

An alternative way of creating an accumulable using G4AccumulablesManager is demonstrated below:

```
// B1RunAction.cc
B1RunAction::B1RunAction()
: G4UserRunAction()
{
  // ..
  // Accumulables can be also created via accumulable manager
  G4AccumulableManager* accumulableManager = G4AccumulableManager::Instance();
```

```
  accumulableManager->CreateAccumulable<G4double>("EdepBis", 0.);
  accumulableManager->CreateAccumulable<G4double>("Edep2Bis", 0.);
}
```

The `G4AccumulablesManager` takes ownership of the accumulables created by its `CreateAccumulable()` function the accumulables allocated in the user code has to be deleted in the user code.

Since Geant4 10.3, the name of the accumulable can be omitted. A generic name "accumulable_N", where N is the current number of registered obects, will be then attributed.

In multi-threading mode all accumulables registered to `G4AccumulablesManager` accumulated on workers can be merged to the master thread by calling `G4AccumulablesManager::Merge()` function. This step may be not necessary in future after a planned closer integration of G4Accumulable classes in the Geant4 kernel.

```
// B1RunAction.cc
void B1RunAction::EndOfRunAction(const G4Run* run)
{
  // ...
  // Merge accumulables
  G4AccumulableManager* accumulableManager = G4AccumulableManager::Instance();
  accumulableManager->Merge();
}
```

The merging mode can be specified using the third (or the second one, if the name is omitted) `G4Accumulable<T>` constructor argument. The merge modes are defined in `G4MergeMode` class enumeration:

```
enum class G4MergeMode {
  kAddition,       // "Or" if boolean type
  kMultiplication, // "And" if boolean type
  kMaximum,        // "Or" if boolean type
  kMinimum         // "And" if boolean type
};
```

The default accumulable merge operation is addition.

The registered accumulables can be accessed via `G4AccumulablesManager` by name or by the id, attributed in the order of registering:

```
// ...
G4AccumulableManager* accumulableManager = G4AccumulableManager::Instance();
// Access accumulables by name
G4double edepBis  = accumulableManager->GetAccumulable<G4double>("EdepBis")->GetValue();
G4double edep2Bis = accumulableManager->GetAccumulable<G4double>("Edep2Bis")->GetValue();

// Access accumulables by id
G4VAccumulable* accumulable = accumulableManager->GetAccumulable(id);
```

9.4.2 User defined accumulables

Users can define their own accumulable class derived from `G4VAccumulable` abstract base class. An example of a `ProcessCounterAccumulable` class, implementing an accumulable holding a map of the processes occurences by the procesesses names, is given below. Such processes occurences map is used in several electromagnetic extended examples, e.g. TestEm1.

ProcCounterAccumulable.hh:

```
#include "G4VAccumulable.hh"
#include "globals.hh"
#include <map>
class ProcCounterAccumulable : public G4VAccumulable
{
```

```
  public:
    ProcCounterAccumulable(const G4String& name)
      : G4VAccumulable(name, 0), fProcCounter() {}
    virtual ~ProcCounterAccumulable() {}

    void CountProcesses(G4String procName);

    virtual void Merge(const G4VAccumulable& other);
    virtual void Reset();

  private:
    std::map<G4String,G4int> fProcCounter;
};
```

ProcCounterAccumulable.cc:

```
void ProcCounterAccumulable::Merge(const G4VAccumulable& other)
{
  const ProcCounterAccumulable& otherProcCounterAccumulable
    = static_cast<const ProcCounterAccumulable&>(other);

  std::map<G4String,G4int>::const_iterator it;
  for (it = otherProcCounterAccumulable.fProcCounter.begin();
       it != otherProcCounterAccumulable.fProcCounter.end(); ++it) {

    G4String procName = it->first;
    G4int otherCount  = it->second;
    if ( fProcCounter.find(procName) == fProcCounter.end()) {
      fProcCounter[procName] = otherCount;
    }
    else {
      fProcCounter[procName] += otherCount;
    }
  }
}

void ProcCounterAccumulable::Reset()
{
  fProcCounter.clear();
}
```

The implementation of the `CountProcesses()` function is identical as in `Run::CountProcesses()` function in TestEm1.

9.5 g4tools

`g4tools` is a "namespace protected" part of `inlib` and `exlib` which is of some interest for Geant4, mainly the histograms, the ntuples and the code to write them at the ROOT, AIDA XML and CSV file formats. The idea of `g4tools` is to cover, with a very light and easy to install package, what is needed to do analysis in a "Geant4 batch program".

As `g4tools` is distributed through Geant4 and in order to avoid potential namespace clashes with other codes that use the `inlib/exlib` to do Geant4 visualization (as for the `g4view` application or some of the exlib examples), the *inlib* and *exlib* namespaces had been automatically changed to *tools* in the `g4tools` distribution. Since in principle Geant4 users will not have to deal directly with the `g4tools` classes, but will manipulate histograms and ntuples through the `G4AnalysisManager`, we are not going to extensively document the `g4tools` classes here. Interested people are encouraged to go at the `inlib/exlib` web pages for that (see inlib/exlib site).

9.5.1 g4tools package

g4tools code is pure header

As explained in `inlib/exlib`, the code found in `g4tools` is "pure header". This comes from the need to have an easy way to build applications, as the `ioda` one, from smartphone, passing by tablets and up to various desktops (UNIX and Windows). For example, if building an application targeted to the Apple AppStore and GooglePlay, the simplest way is to pass through `Xcode` and the Android `make` system (or `Eclipse`), and having not to build libraries simplifies a lot the handling of all these IDEs for the same application. A fallback of that is that the installation of `g4tools` (if not using the one coming with Geant4) is straightforward, you simply unzip the file containing the source code! To build an application using `g4tools`, as for `inlib/exlib`, you simply have to declare to your build system the "-I" toward the unfolded directory and do "Build and Run".

g4tools test

`g4tools` comes with test programs of its own that may be useful in case of problems (for example porting on a not yet covered platform). You can build and run them with:

```
UNIX> <get g4tools.zip>
UNIX> <unzip g4tools.zip>
UNIX> cd g4tools/test/cpp
UNIX> ./build
UNIX> ./tools_test_histo
UNIX> ./tools_test_wroot
UNIX> etc...
```

and on Windows:

```
DOS> <setup VisualC++ so that CL.exe is in your PATH>
DOS> <get g4tools.zip>
DOS> <unzip g4tools.zip> (you can use the unzip.exe of CYGWIN)
DOS> cd g4tools\test\cpp
DOS> .\build.bat
DOS> .\tools_test_histo.exe
DOS> .\tools_test_wroot.exe
DOS> etc...
```

g4tools in Geant4

The `g4tools` header files are distributed in the Geant4 source in the `source/analysis/include/tools` directory and in the Geant4 installation, they are installed in `include/tools` directory. The `g4tools` test programs, included only in Geant4 development versions, can be downloaded with the `g4tools-[version].zip` file from the `inexlib` download page).

While the Geant4 analysis manager provides the methods for booking and filling the g4tools objects, it does not interface all public functions. Users can access the g4tools objects (see *Accessing Histograms*) and use the g4tools API described in the next section to get the needed informations.

9.5.2 User API

We describe here some of the public methods potentially seen by a user doing analysis.

Booking and filling

```
h1d(const std::string& title,unsigned int Xnumber,double Xmin,double Xmax);
h1d(const std::string& title,const std::vector<double>& edges);

bool fill(double X,double Weight = 1);
```

example:

```
#include <tools/histo/h1d>
#include <tools/randd>
...
tools::histo::h1d h("Gauss",100,-5,5);
tools::rgaussd rg(1,2);
for(unsigned int count=0;count<entries;count++) h.fill(rg.shoot(),1.4);
```

Mean and rms

```
tools::histo::h1d h("Gauss",100,-5,5);
...
std::cout << " mean " << h.mean() << ", rms " << h.rms() << std::endl;
```

Bin infos

When doing a:

```
bool fill(double X,double Weight = 1);
```

the histogram class maintains, for each bin, the number of entries, the sum of weights that we can note “Sw”, the sum of W by W “Sw2”, the sum of X by Weight “Sxw”, the sum of X by X by W “Sx2w”. Then bin method names reflect these notations, for example to get the 50 bin sum of X*X*W:

```
double Sx2w = h.bin_Sx2w(50);
```

and the same for the other sums:

```
double Sw = h.bin_Sw(50);
double Sw2 = h.bin_Sw2(50);
double Sxw = h.bin_Sxw(50);
unsigned int n = h.bin_entries(50);
```

You can have also all infos on all bins with:

```
tools::histo::h1d h(...);
...
const std::vector<unsigned int>& _entries = h.bins_entries();
const std::vector<double>& _bins_sum_w = h.bins_sum_w();
const std::vector<double>& _bins_sum_w2 = h.bins_sum_w2();
const std::vector< std::vector<double> >& _bins_sum_xw = h.bins_sum_xw();
const std::vector< std::vector<double> >& _bins_sum_x2w = h.bins_sum_x2w();
```

for example to dump bin 50 of an histo booked with 100 bins:

```
std::cout << "entries[50] = " << _entries[50] << std::endl;
std::cout << "  sum_w[50] = " << _bins_sum_w[50] << std::endl;
std::cout << " sum_w2[50] = " << _bins_sum_w2[50] << std::endl;
std::cout << " sum_xw[50] = " << _bins_sum_xw[50][0] << std::endl;   //0 = xaxis
std::cout << "sum_x2w[50] = " << _bins_sum_x2w[50][0] << std::endl;  //0 = xaxis
```

(Take care that the [0] entries in the upper vectors are for the "underflow bin" and the last one is for the "overflow bin").

All data

You can get all internal data of an histo through the histo_data class:

```
const tools::histo::h1d::hd_t& hdata = h.dac();  //dac=data access.
```

and then, for example, find back the bins infos with:

```
const std::vector<unsigned int>& _entries = hdata.m_bin_entries;
const std::vector<double>& _bins_sum_w = hdata.m_bin_Sw;
const std::vector<double>& _bins_sum_w2 = hdata.m_bin_Sw2;
const std::vector< std::vector<double> >& _bins_sum_xw = hdata.m_bin_Sxw;
const std::vector< std::vector<double> >& _bins_sum_x2w = hdata.m_bin_Sx2w;
// dump bin 50 :
std::cout << "entries[50] = " << _entries[50] << std::endl;
std::cout << "  sum_w[50] = " << _bins_sum_w[50] << std::endl;
std::cout << " sum_w2[50] = " << _bins_sum_w2[50] << std::endl;
std::cout << " sum_xw[50] = " << _bins_sum_xw[50][0] << std::endl;   //0 = xaxis
std::cout << "sum_x2w[50] = " << _bins_sum_x2w[50][0] << std::endl;  //0 = xaxis
```

See the tools/histo/histo_data class for all internal fields.

Projections

From a 2D histo, you can get the x projection with:

```
tools::histo::h1d* projection = tools::histo::projection_x(h2d,"ProjX");
...
delete projection;
```

See test/cpp/histo.cpp for example code. Other slicing and projection methods are:

```
// h2d -> h1d. (User gets ownership of the returned object).
h1d* slice_x(const h2d&,int y_beg_ibin,int y_end_ibin,const std::string& title);
h1d* projection_x(const h2d&,const std::string& title);
h1d* slice_y(const h2d&,int x_beg_ibin,int x_end_ibin,const std::string& title);
h1d* projection_y(const h2d&,const std::string& title);
// h2d -> p1d. (User gets ownership of the returned object).
p1d* profile_x(const h2d&,int y_beg_ibin,int y_end_ibin,const std::string& title);
p1d* profile_x(const h2d&,const std::string&);
p1d* profile_y(const h2d&,int x_beg_ibin,int x_end_ibin,const std::string& title);
p1d* profile_y(const h2d&,const std::string& title);
// h3d -> h2d. (User gets ownership of the returned object).
h2d* slice_xy(const h3d&,int z_beg_ibin,int z_end_ibin,const std::string& title);
h2d* projection_xy(const h3d&,const std::string& title);
h2d* slice_yz(const h3d&,int x_beg_ibin,int x_end_ibin,const std::string& title);
h2d* projection_yz(const h3d&,const std::string& title);
h2d* slice_xz(const h3d&,int y_beg_ibin,int y_end_ibin,const std::string& title);
h2d* projection_xz(const h3d&,const std::string& title);
```

CHAPTER

TEN

EXAMPLES

10.1 Introduction

The Geant4 toolkit includes several fully coded examples that demonstrate the implementation of the user classes required to build a customized simulation.

The new "basic" examples cover the most typical use-cases of a Geant4 application while keeping simplicity and ease of use. They are provided as a starting point for new Geant4 application developers.

A set of "extended" examples range from the simulation of a non-interacting particle and a trivial detector to the simulation of electromagnetic and hadronic physics processes in a complex detector. Some of these examples require some libraries in addition to those of Geant4.

The "advanced" examples cover the use-cases typical of a "toolkit"-oriented kind of development, where real complete applications for different simulation studies are provided.

All examples can be compiled and run without modification. Most of them can be run both in interactive and batch mode using the input macro files (`*.in`) and reference output files (`*.out`) provided. Most examples are run routinely as part of the validation, or testing, of official releases of the Geant4 toolkit.

The previous set of examples oriented to novice users, "novice", has been refactored in "basic" and "extended" examples sets in Geant4 10.0. The information about the original set of these examples can be found at the last section of this chapter.

10.2 Basic Examples

10.2.1 Basic Examples Summary

Descriptions of the 5 basic examples are provided here along with links to source code documentation automatically generated with Doxygen.

Example B1 (see also Doxygen page)

- Simple geometry with a few solids
- Geometry with simple placements (G4PVPlacement)
- Scoring total dose in a selected volume in user action classes
- Geant4 physics list (QBBC)

Example B2 (see also Doxygen page)

- Simplified tracker geometry with uniform magnetic field
- Geometry with simple placements (G4PVPlacement) and parameterisation (G4PVParameterisation)
- Scoring within tracker via G4 sensitive detector and hits
- Geant4 physics list (FTFP_BERT) with step limiter

- Started from novice N02 example

Example B3 (see also Doxygen page)

- Schematic Positron Emission Tomography system
- Geometry with simple placements with rotation (G4PVPlacement)
- Radioactive source
- Scoring within Crystals via G4 scorers
- Modular physics list built via builders provided in Geant4

Example B4 (see also Doxygen page)

- Simplified calorimeter with layers of two materials
- Geometry with replica (G4PVReplica)
- Scoring within layers in four ways: via user actions (a), via user own object (b), via G4 sensitive detector and hits (c) and via scorers (d)
- Geant4 physics list (FTFP_BERT)
- Saving histograms and ntuple in a file using Geant4 analysis tools
- UI commands defined using G4GenericMessenger
- Started from novice/N03 example

Example B5 (see also Doxygen page)

- A double-arm spectrometer with wire chambers, hodoscopes and calorimeters with a local constant magnetic field
- Geometry with placements with rotation, replicas and parameterisation
- Scoring within wire chambers, hodoscopes and calorimeters via G4 sensitive detector and hits
- Geant4 physics list (FTFP_BERT) with step limiter
- UI commands defined using G4GenericMessenger
- Saving histograms and ntuple in a file using Geant4 analysis tools
- Started from extended/analysis/A01

The next three tables display the "item charts" for the examples currently prepared in the basic level. *Examples B1 and B2*, *Examples B3 and B4*, and *Example B5*.

	Example B1	*Example B2*
Description	Simple application for accounting dose in a selected volume	Fixed target tracker geometry
Geometry	• solids: box, cons, trd • simple placements with translation	• solids: box, tubs • simple placements with translation (a) • parameterised volume (b) • uniform magnetic field
Physics	Geant4 physics list: QBBC	Geant4 physics list: FTFP_BERT
Primary generator	Particle gun	Particle gun
Scoring	User action classes	Sensitive detector & hits
Vis/GUI	Detector & trajectory drawing	• Detector, trajectory & hits drawing • GUI
Stacking		
Analysis		

Table: The "item chart" for basic level examples *B1* and *B2*.

	Example B3	*Example B4*
Description	Schematic Positron Emitted Tomography system	Simplified calorimeter with layers of two materials
Geometry	• solids: box, tubs • simple placements with rotation	• solids: box • simple placements with translation • replica • uniform magnetic field
Physics	Modular physics list with Geant4 builders	Geant4 physics list: FTFP_BERT
Primary generator	Radioactive source (particle gun with Fluor ions)	Particle gun
Scoring	Multi functional (sensitive) detector & scorers	• 1. User action classes • 2. User own object (runData) • 3. Sensitive detector & hits • 4. Multi functional (sensitive) detector & scorers
Vis/GUI	Detector, trajectory & hits drawing	• Detector, trajectory & hits drawing • GUI
Stacking	Killing all neutrina	
Analysis		Histograms 1D, ntuple

Table: The "item chart" for basic level examples *B3* and *B4*.

	Example B5
Description	Double-arm spectrometer with several detectors and a local constant magnetic field
Geometry	• solids: box, tubs • simple placements with rotation • replica • parameterised volume • local constant magnetic field • modifying geometry between runs
Physics	Geant4 physics list: FTFP_BERT
Primary generator	Particle gun
Scoring	Sensitive detectors & hits
Vis/GUI	• Detector, trajectory & hits drawing • User defined visualization attributes
Stacking	•
Analysis	• Histograms 1D, ntuple • Saving file per run

Table: The "item chart" for basic level example *B5*.

10.2.2 Basic Examples Macros

All basic examples can be run either interactively or in a batch mode (see section *How to Define the main() Program* and *How to Execute a Program*) and they are provided with the following set of macros:

- `init_vis.mac`
- `vis.mac`
- `[gui.mac]`
- `run1.mac`, `run2.mac`
- `exampleBN.in`

The selection is done automatically according to the application build configuration.

The `init_vis.mac` macro is always executed just after the Geant4 kernel and user application classes instantiation. It sets first some defaults, then performs Geant4 kernel initialization and finally calls the `vis.mac` macro with visualization setting.

The `vis.mac` macros in each of the examples all have the same structure - except for example B1, see below. There are only a few lines in each example with a setting different from the other examples and so they can be easily spotted when looking in the macro. Various commands are proposed in commented blocks of lines with explanations so that a user can just uncomment lines and observe the effect. Additionally, in example B4, there are some visualization tutorial macros in `macros/visTutor/`. See more on visualization in section *How to Visualize the Detector and Events* and chapter *Visualization*.

From Release 9.6 the `vis.mac` macro in example B1 has additional commands that demonstrate additional functionality of the vis system, such as displaying text, axes, scales, date, logo and shows how to change viewpoint and style. Consider copying these to your favourite example or application. To see even more commands use `help` or `ls` or browse the available UI commands in section *Built-in Commands*.

The `gui.mac` macros are provided in examples B2 and B4. This macro is automatically executed if Geant4 is built with any GUI session. See more on graphical user interfaces in section *How to Set Up an Interactive Session*.

When running interactively, the example program stops after processing the `init_vis.mac` macro and the Geant4 kernel initialization, invoked from the macro, with the prompt `Idle>`. At this stage users can type in the commands from `run1.mac` line by line (recommended when running the example for the first time) or execute all commands at once using the `"/control/execute run1.mac"` command.

The `run2.mac` macros define conditions for execution a run with a larger number of events and so they are recommended to be executed in a batch. The `exampleBN.in` macros are also supposed to be run in a batch mode and their outputs from the Geant4 system testing are available in the files `exampleBN.out`.

10.2.3 Multi-threading

Multi-threading mode

All basic examples have been migrated to multi-threading (MT). No special steps are needed to build the examples in multi-threading mode. They will automatically run in MT when they are built against the Geant4 libraries built with MT mode activated, otherwise they will run in sequential mode.

The choice of multi-threading mode is done be creating `G4MTRunManager` instead of `G4RunManager` in the example `main()`:

```
#ifdef G4MULTITHREADED
  G4MTRunManager* runManager = new G4MTRunManager;
#else
  G4RunManager* runManager = new G4RunManager;
#endif
```

The compiler flag `-DG4MULTITHREADED` is automatically set when building applications using Geant4's CMake (via GEANT4_USE_FILE) and GNUmake systems, and is listed in the flags reported by the –cflags option of the geant4-config program.

`Action Initialization` class [B1, B2, B3, B4, B4]

See B1, B2, B3, B4, B5

A newly introduced `BnActionInitialization` class derived from `G4VUserActionInitialization`, present in all basic examples, instantiates and registers all user action classes with the Geant4 kernel.

While in sequential mode the action classes are instatiated just once, via invocation of the method `BnActionInitialization::Build()`. In multi-threading mode the same method is invoked for each worker thread, so all user action classes are defined thread-locally.

A run action class is instantiated both thread-locally and globally; that is why its instance is created also in the method `BnActionInitialization::BuildForMaster()`, which is invoked only in multi-threading mode.

10.2.4 Example B1

See http://geant4.web.cern.ch/geant4/UserDocumentation/Doxygen/examples_doc/html/ExampleB1.html

Basic concept:

This example demonstrates a simple (medical) application within which users will familiarize themselves with simple placement, use the NIST material database, and can utilize electromagnetic and/or hadronic physics processes. Two items of information are collected in this example: the energy deposited and the total dose for a selected volume.

This example uses the Geant4 physics list QBBC, which is instantiated in the main() function. It requires data files for electromagnetic and hadronic processes. See more on installation of the datasets in Geant4 Installation Guide, Chapter 3.3: Note On Geant4 Datasets. The following datasets: G4LEDATA, G4LEVELGAMMADATA, G4NEUTRONXSDATA, G4SAIDXSDATA and G4ENSDFSTATEDATA are mandatory for this example.

Classes:

B1DetectorConstruction The geometry is constructed in the `B1DetectorConstruction` class. The setup consists of a box shaped envelope containing two volumes: a circular cone and a trapezoid.
Some common materials from medical applications are used. The envelope is made of water and the two inner volumes are made from tissue and bone materials. These materials are created using the `G4NistManager` class, which allows one to build a material from the NIST database using their names. Available materials and their compositions can be found in the Appendix *Geant4 Material Database*.
The physical volumes are made from Constructive Solid Geometry (CSG) solids and placed without rotation using the `G4PVPlacement` class.

B1PrimaryGeneratorAction The default kinematics is a 6 MeV gamma, randomly distributed in front of the envelope across 80% of the transverse (X,Y) plane. This default setting can be changed via the commands of the `G4ParticleGun` class.

B1SteppingAction It is in the `UserSteppingAction()` function that the energy deposition is collected for a selected volume.

B1EventAction The statistical event by event accumulation of energy deposition. At the end of event, the acummulated values are passed in `B1RunAction` and summed over the whole run.

B1RunAction Sums the event energy depositions. In multi-threading mode the energy deposition accumulated in `G4Accumulable` objects per worker is merged to the master. Information about the primary particle is printed in this class along with the computation of the dose. An example of creating and computing new units (e.g., dose) is also shown in the class constructor.
`G4Accumulable<G4double>` type instead of `G4double` is used for the `B1RunAction` data members in order to facilitate merging of the values accumulated on workers to the master. At present the accumulables have to be registered to `G4AccumulablesManager` and `G4ParametersManager::Merge()` has to be called from the users code. This is planned to be further simplified with a closer integration of `G4Accumulable` classes in the Geant4 kernel next year.

10.2.5 Example B2

See http://geant4.web.cern.ch/geant4/UserDocumentation/Doxygen/examples_doc/html/ExampleB2.html

This example simulates a simplified fixed target experiment. To demonstrate alternative ways of constructing the geometry two variants are provided: B2a (explicit construction) and B2b (parametrized volumes).

The set of available particles and their physics processes are defined in the FTFP_BERT physics list. This Geant4 physics list is instantiated in the main() function. It requires data files for electromagnetic and hadronic processes. See more on installation of the datasets in Geant4 Installation Guide, Chapter 3.3: Note On Geant4 Datasets. The following datasets: G4LEDATA, G4LEVELGAMMADATA, G4NEUTRONXSDATA, G4SAIDXSDATA and G4ENSDFSTATEDATA are mandatory for this example.

This example also illustrates how to introduce tracking constraints like maximum step length via `G4StepLimiter`, and minimum kinetic energy, etc., via the `G4UserSpecialCuts` processes. This is accomplished by adding `G4StepLimiterPhysics` to the physics list.

Classes:

B2[a, b]DetectorConstruction The setup consists of a target followed by six chambers of increasing transverse size at defined distances from the target. These chambers are located in a region called the Tracker region. Their shape are cylinders constructed as simple cylinders (in `B2aDetectorConstruction`) and as parametrised volumes (in `B2bDetectorConstruction`) - see also B2bChamberParameterisation class.
In addition, a global uniform transverse magnetic field can be applied using `G4GlobalMagFieldMessenger`, instantiated in `ConstructSDandField()` with a non zero

field value, or via an interactive command. An instance of the `B2TrackerSD` class is created and associated with each logical chamber volume (in B2a) and with the one `G4LogicalVolume` associated with `G4PVParameterised` (in B2b).
One can change the materials of the target and the chambers interactively via the commands defined in `B2aDetectorMessenger` (or `B2bDetectorMessenger`).
This example also illustrates how to introduce tracking constraints like maximum step length, minimum kinetic energy etc. via the G4UserLimits class and associated G4StepLimiter and G4UserSpecialCuts processes. The maximum step limit in the tracker region can be set by the interactive command defined in `B2aDetectorMessenger` (or `B2bDetectorMessenger`).

`B2PrimaryGeneratorAction` The primary generator action class employs the `G4ParticleGun`. The primary kinematics consists of a single particle which hits the target perpendicular to the entrance face. The type of the particle and its energy can be changed via the G4 built-in commands of the `G4ParticleGun` class.

`B2EventAction` The event number is written to the log file every requested number of events in `BeginOfEventAction()` and `EndOfEventAction()`. Moreover, for the first 100 events and every 100 events thereafter information about the number of stored trajectories in the event is printed as well as the number of hits stored in the `G4VHitsCollection`.

`B2RunAction` The run number is printed at `BeginOfRunAction()`, where the `G4RunManager` is also informed how to `SetRandomNumberStore` for storing initial random number seeds per run or per event.

`B2TrackerHit` The tracker hit class is derived from `G4VHit`. In this example, a tracker hit is a step by step record of the track identifier, the chamber number, the total energy deposit in this step, and the position of the energy deposit.

`B2TrackerSD` The tracker sensitive detector class is derived from `G4VSensitiveDetector`. In `ProcessHits()` - called from the Geant4 kernel at each step - it creates one hit in the selected volume so long as energy is deposited in the medium during that step. This hit is inserted in a HitsCollection. The HitsCollection is printed at the end of each event (via the method `B2TrackerSD::EndOfEvent()`), under the control of the "/hits/verbose 2" command.

10.2.6 Example B3

See http://geant4.web.cern.ch/geant4/UserDocumentation/Doxygen/examples_doc/html/ExampleB3.html

This example simulates a Schematic Positron Emission Tomography system. To demonstrate alternative ways of accumulation event statistics in a run two variants are provided: B3a (using new `G4Accumulable` class) and B3b (using `G4Run` class).

Classes:

`B3DetectorConstruction` Crystals are circularly arranged to form a ring. A number rings make up the full detector (gamma camera). This is done by positionning Crystals in Ring with an appropriate rotation matrix. Several copies of Ring are then placed in the full detector.
The Crystal material, Lu2SiO5, is not included in the G4Nist database. Therefore, it is explicitly built in `DefineMaterials()`.
Crystals are defined as scorers in `DetectorConstruction::CreateScorers()`. There are two G4MultiFunctionalDetector objects: one for the Crystal (EnergyDeposit), and one for the Patient (DoseDeposit).

`B3PhysicsList` The physics list contains standard electromagnetic processes and the radioactiveDecay module for GenericIon. It is defined in the `B3PhysicsList` class as a Geant4 modular physics list with registered Geant4 physics builders:

- `G4DecayPhysics`
- `G4RadioactiveDecayPhysics`
- `G4EmStandardPhysics`

`B3PrimaryGeneratorAction` The default particle beam is an ion (F18), at rest, randomly distributed within a zone inside a patient and is defined in `GeneratePrimaries()`.

`B3aEventAction`, `B3aRunAction` Energy deposited in crystals is summed by `G4Scorer`. At the end of event, the values acummulated in `B3aEventAction` are passed in `B3aRunAction` and summed over the whole

run. In multi-threading mode the data accumulated in `G4Accumulable` objects per workers is merged to the master in `B3aRunAction::EndOfRunAction()` and the final result is printed on the screen.
`G4Accumulable<>` type instead of `G4double` and `G4int` types is used for the `B3aRunAction` data members in order to facilitate merging of the values accumulated on workers to the master. At present the accumulables have to be registered to `G4AccumulablesManager` and `G4AccumulablesManager::Merge()` has to be called from the users code. This is planned to be further simplified with a closer integration of `G4Accumulable` classes in the Geant4 kernel next year.

B3bRun, B3bRunAction Energy deposited in crystals is summed by `G4Scorer`. `B3Run::RecordEvent()` collects information event by event from the hits collections, and accumulates statistics for `B3RunAction::EndOfRunAction()`. In multi-threading mode the statistics accumulated per worker is merged to the master in `Run::Merge()`.

B3StackingAction Beta decay of Fluorine generates a neutrino. One wishes not to track this neutrino; therefore one kills it immediately, before created particles are put in a stack.

10.2.7 Example B4

See http://geant4.web.cern.ch/geant4/UserDocumentation/Doxygen/examples_doc/html/ExampleB4.html

This example simulates a simple Sampling Calorimeter setup. To demonstrate several possible ways of data scoring, the example is provided in four variants: B4a, B4b, B4c, B4d. (See also examples/extended/electromagnetic/TestEm3).

The set of available particles and their physics processes are defined in the FTFP_BERT physics list. This Geant4 physics list is instantiated in the main() function. It requires data files for electromagnetic and hadronic processes. See more on installation of the datasets in Geant4 Installation Guide, Chapter 3.3: Note On Geant4 Datasets. The following datasets: G4LEDATA, G4LEVELGAMMADATA, G4NEUTRONXSDATA, G4SAIDXSDATA and G4ENSDFSTATEDATA are mandatory for this example.

Classes:

B4[c, d]DetectorConstruction The calorimeter is a box made of a given number of layers. A layer consists of an absorber plate and of a detection gap. The layer is replicated. In addition a transverse uniform magnetic field can be applied using `G4GlobalMagFieldMessenger`, instantiated in `ConstructSDandField()` with a non zero field value, or via interactive commands.

B4PrimaryGeneratorAction The primary generator action class uses `G4ParticleGun`. It defines a single particle which hits the calorimeter perpendicular to the input face. The type of the particle can be changed via the G4 built-in commands of the `G4ParticleGun` class.

B4RunAction It accumulates statistics and computes dispersion of the energy deposit and track lengths of charged particles with the aid of analysis tools. H1D histograms are created in `BeginOfRunAction()` for the energy deposit and track length in both Absorber and Gap volumes. The same values are also saved in an ntuple. The histograms and ntuple are saved in the output file in a format accoring to a selected technology in `B4Analysis.hh`. In `EndOfRunAction()`, the accumulated statistics and computed dispersion are printed. When running in multi-threading mode, the histograms accumulated on threads are automatically merged in a single output file, while the ntuple is written in files per thread.

Classes in B4a (scoring via user actions):

B4aSteppingAction In `UserSteppingAction()` the energy deposit and track lengths of charged particles in each step in the Absober and Gap layers are collected and subsequently recorded in `B4aEventAction`.

B4aEventAction It defines data members to hold the energy deposit and track lengths of charged particles in the Absorber and Gap layers. In `EndOfEventAction()`, these quantities are printed and filled in H1D histograms and ntuple to accumulate statistic and compute dispersion.

Classes in B4b (via user own object):

B4bRunData A data class, derived from `G4Run`, which defines data members to hold the energy deposit and track lengths of charged particles in the Absober and Gap layers. It is instantiated in

`B4bRunAction::GenerateRun`. The data are collected step by step in `B4bSteppingAction`, and the accumulated values are entered in histograms and an ntuple event by event in `B4bEventAction`.

`B4bSteppingAction` In `UserSteppingAction()` the energy deposit and track lengths of charged particles in Absorber and Gap layers are collected and subsequently recorded in `B4bRunData`.

`B4bEventAction` In `EndOfEventAction()`, the accumulated quantities of the energy deposit and track lengths of charged particles in Absorber and Gap layers are printed and then stored in `B4bRunData`.

Classes in B4c (via Geant4 sensitive detector and hits):

`B4cDetectorConstruction` In addition to materials, volumes and uniform magnetic field definitions as in `B4DetectorConstruction`, in `ConstructSDandField()` two instances of the `B4cCalorimeterSD` class are created and associated with Absorber and Gap volumes.

`B4cCalorHit` The calorimeter hit class is derived from `G4VHit`. It defines data members to store the energy deposit and track lengths of charged particles in a selected volume.

`B4cCalorimeterSD` The calorimeter sensitive detector class is derived from `G4VSensitiveDetector`. Two instances of this class are created in `B4cDetectorConstruction` and associated with Absorber and Gap volumes. In `Initialize()`, it creates one hit for each calorimeter layer and one more hit for accounting the total quantities in all layers. The values are accounted in hits in the `ProcessHits()` function, which is called by the Geant4 kernel at each step.

`B4cEventAction` In `EndOfEventAction()`, the accumulated quantities of the energy deposit and track lengths of charged particles in Absorber and Gap layers are printed and then stored in the hits collections.

Classes in B4d (via Geant4 scorers):

`B4dDetectorConstruction` In addition to materials, volumes and uniform magnetic field definitions as in `B4DetectorConstruction`, in `ConstructSDandField()` sensitive detectors of `G4MultiFunctionalDetector` type with primitive scorers are created and associated with Absorber and Gap volumes.

`B4dEventAction` In `EndOfEventAction()`, the accumulated quantities of the energy deposit and track lengths of charged particles in Absober and Gap layers are printed and then stored in the hits collections.

10.2.8 Example B5

See http://geant4.web.cern.ch/geant4/UserDocumentation/Doxygen/examples_doc/html/ExampleB5.html

This example simulates a a double-arm spectrometer with wire chambers, hodoscopes and calorimeters with a uniform local magnetic field.

The set of available particles and their physics processes are defined in the FTFP_BERT physics list. This Geant4 physics list is instantiated in the main() function. It requires data files for electromagnetic and hadronic processes. See more on installation of the datasets in Geant4 Installation Guide, Chapter 3.3: Note On Geant4 Datasets. The following datasets: G4LEDATA, G4LEVELGAMMADATA, G4NEUTRONXSDATA, G4SAIDXSDATA and G4ENSDFSTATEDATA are mandatory for this example.

This example also illustrates how to introduce tracking constraints like maximum step length via `G4StepLimiter`, and minimum kinetic energy, etc., via the `G4UserSpecialCuts` processes. This is accomplished by adding `G4StepLimiterPhysics` to the physics list.

This example can be built with excluding visualization and/or Geant4 user interface via `G4VIS_USE` and `G4UI_USE` compiler options (see exampleB5.cc). These options are defined by default with Geant4 configuration; they can be switched off at compilation time via the CMake options `G4VIS_NONE` or `G4UI_NONE` or via the environment variables of the same name if using GNUmake build.

Classes:

`B5DetectorConstruction` The spectrometer consists of two detector arms. One arm provides position and timing information of the incident particle while the other collects position, timing and energy information of the particle after it has been deflected by a magnetic field centered at the spectrometer pivot point.

First arm: box filled with air, also containing:
- 1 hodoscope (15 vertical strips of plastic scintillator)
- 1 drift chamber (horizontal argon gas layers with a "virtual wire" at the center of each layer)

Second arm: box filled with air, also containing:
- 1 hodoscope (25 vertical strips of plastic scintillator)
- 1 drift chamber (5 horizontal argon gas layers with a "virtual wire" at the center of each layer)
- 1 electromagnetic calorimeter: a box sub-divided along x,y and z axes into cells of CsI (see also B5CellParameterisation class)
- 1 hadronic calorimeter: a box sub-divided along x,y, and z axes into cells of lead, with a layer of plastic scintillator placed at the center of each cell

The magnetic field region is represented by an air-filled cylinder which contains the field (see B5MagneticField.). The maximum step limit in the magnetic field region is also set via the G4UserLimits class in a similar way as in Example B2.

The rotation angle of the second arm and the magnetic field value can be set via the interactive command defined using the `G4GenericMessenger` class.

`B5PrimaryGeneratorAction` The primary generator action class employs the G4ParticleGun. The primary kinematics consists of a single particle which is is sent in the direction of the first spectrometer arm.

The type of the particle and its several properties can be changed via the Geant4 built-in commands of the `G4ParticleGun` class or this example command defined using the `G4GenericMessenger` class.

`B5EventAction` An event consists of the generation of a single particle which is transported through the first spectrometer arm. Here, a scintillator hodoscope records the reference time of the particle before it passes through a drift chamber where the particle position is measured. Momentum analysis is performed as the particle passes through a magnetic field at the spectrometer pivot and then into the second spectrometer arm. In the second arm, the particle passes through another hodoscope and drift chamber before interacting in the electromagnetic calorimeter. Here it is likely that particles will induce electromagnetic showers. The shower energy is recorded in a three-dimensional array of CsI crystals. Secondary particles from the shower, as well as primary particles which do not interact in the CsI crystals, pass into the hadronic calorimeter. Here, the remaining energy is collected in a three-dimensional array of scintillator-lead sandwiches.

In first execution of `BeginOfEventAction()` the hits collections identifiers are saved in data members of the class and then used in `EndOfEventAction()` for accessing the hists collections and filling the accounted information in defined histograms and ntuples and printing its summary in a log file. The frequency of printing can be tuned with the built-in command **`"/run/printProgress frequency".`**

`B5RunAction` The run action class handles the histograms and ntuples with the aid of Geant4 analysis tools in a similar way as in Example B4. From Release 10.2 the vectors of energy deposits in Electromagnetic and Hadronic calorimeter cells are also stored in the ntuple.

Hit and Sensitive Detector Classes All the information required to simulate and analyze an event is recorded in hits. This information is recorded in the following sensitive detectors:
- Hodoscope (`B5HodoscopeSD`, `B5HodoscopeHit`)
  - particle time
  - strip ID, position and rotation
- Drift chamber: (`B5DriftChamberSD`, `B5DriftChamberHit`)
  - particle time
  - particle position
  - layer ID
- Electromagnetic calorimeter: (`B5EmCalorimeterSD`, `B5EmCalorimeterHit`)
  - energy deposited in cell
  - cell ID, position and rotation
- Hadronic calorimeter: (`B5HadCalorimeterSD`, `B5HadCalorimeterHit`)
  - energy deposited in cell
  - cell column ID and row ID, position and rotation

The hit classes include methods `GetAttDefs` and `CreateAttValues` to define and then fill extra "HepRep-style" Attributes that the visualization system can use to present extra information about the hits. For example, if you pick a `B5HadCalorimeterHit` in OpenGL or a HepRep viewer, you will be shown the hit's "Hit Type", "Column ID", "Row ID", "Energy Deposited" and "Position".

These attributes are essentially arbitrary extra pieces of information (integers, doubles or strings) that are carried through the visualization. Each attribute is defined once in `G4AttDef` object and then is filled for each hit in a `G4AttValue` object. These attributes can also be used by commands to filter which hits are drawn: `"/vis/filtering/hits/drawByAttribute"`.
Detector Geometry and trajectories also carry HepRep-style attributes, but these are filled automatically in the base classes. HepRep is further described at: http://www.slac.stanford.edu/~perl/heprep/

10.3 Extended Examples

10.3.1 Extended Example Summary

Geant4 extended examples serve three purposes:

- testing and validation of processes and tracking,
- demonstration of Geant4 tools, and
- extending the functionality of Geant4.

The code for these examples is maintained as part of the categories to which they belong. Links to descriptions of the examples are listed below.

Analysis

- AnaEx01 - histogram and tuple manipulations using Geant4 internal g4tools system
- AnaEx02 - histogram and tuple manipulations using ROOT
- AnaEx03 - histogram and tuple manipulations using the AIDA interface
- B1Con - modified basic example B1 showing how to use a Convergence Tester
- [A01] - this examples has been refactored in Example B5 in the basic set.

Common

- ReadMe - a set of common classes which can be reused in other examples demonstrating just a particular feature. This module is going to be enhanced in future.

Biasing

- Variance Reduction - examples (*B01*, *B02* and *B03*) on variance reduction techniques and scoring and application of Reverse MonteCarlo in Geant4 ReverseMC
- Generic biasing examples illustrate the usage of a biasing scheme implemented since version Geant4 10.0.
  - GB01 This example illustrates how to bias process cross-sections in this scheme.
  - GB02 Illustrates a force collision scheme similar to the MCNP one.
  - GB03 Illustrates geometry based biasing.
  - GB04 Illustrates a bremsstrahlung splitting.
  - GB05 Illustrates a "splitting by cross-section" technique: a splitting-based technique using absorption cross-section to control the neutron population.
  - GB06 Illustrates the usage of parallel geometries with generic biasing.

Electromagnetic

- TestEm0 - how to print cross-sections and stopping power used in input by the standard EM package

- TestEm1 - how to count processes, activate/inactivate them and survey the range of charged particles. How to define a maximum step size
- TestEm2 - shower development in an homogeneous material : longitudinal and lateral profiles
- TestEm3 - shower development in a sampling calorimeter : collect energy deposited, survey energy flow and print stopping power
- TestEm4 - 9 MeV point like photon source: plot spectrum of energy deposited in a single media
- TestEm5 - how to study transmission, absorption and reflection of particles through a single, thin or thick, layer.
- TestEm6 - physics list for rare, high energy, electromagnetic processes: gamma conversion and e+ annihilation into pair of muons
- TestEm7 - how to produce a Bragg curve in water phantom. How to compute dose in tallies
- TestEm8 - test of photo-absorption-ionisation model in thin absorbers, and transition radiation
- TestEm9 - shower development in a crystal calorimeter; cut-per-region
- TestEm10 - XTR transition radiation model, investigation of ionisation in thin absorbers
- TestEm11 - how to plot a depth dose profile in a rectangular box
- TestEm12 - how to plot a depth dose profile in spherical geometry : point like source
- TestEm13 - how to compute cross sections of EM processes from rate of transmission coefficient
- TestEm14 - how to compute cross sections of EM processes from direct evaluation of the mean-free path. How to plot final state
- TestEm15 - compute and plot final state of Multiple Scattering as an isolated process
- TestEm16 - simulation of synchrotron radiation
- TestEm17 - check the cross sections of high energy muon processes
- TestEm18 - energy lost by a charged particle in a single layer, due to ionization and bremsstrahlung

Check basic quantities	
Total cross sections, mean free paths ...	Em0, Em13, Em14
Stopping power, particle range ...	Em0, Em1, Em5, Em11, Em12
Final state : energy spectra, angular distributions	Em14
Energy loss fluctuations	Em18
Multiple Coulomb scattering	
as an isolated mechanism	Em15
as a result of particle transport	Em5
More global verifications	
Single layer: transmission, absorption, reflexion	Em5
Bragg curve, tallies	Em7
Depth dose distribution	Em11, Em12
Shower shapes, Moliere radius	Em2
Sampling calorimeters, energy flow	Em3
Crystal calorimeters	Em9
Other specialized programs	
High energy muon physics	Em17
Other rare, high energy processes	Em6
Synchrotron radiation	Em16
Transition radiation	Em8
Photo-absorption-ionization model	Em10

Table: TestEm by theme

Error Propagation

- ReadMe - error propagation utility

Event Generator

- exgps - illustrating the usage of the `G4GeneralParticleSource` utility
- particleGun - demonstrating three different ways of usage of `G4ParticleGun`, shooting primary particles in different cases
- userPrimaryGenerator - demonstrating how to create a primary event including several vertices and several primary particles per vertex
- HepMCEx01 - simplified collider detector using HepMC interface and stacking
- HepMCEx02 - connecting primary particles in Geant4 with various event generators using the HepMC interface
- MCTruth - demonstrating a mechanism for Monte Carlo truth handling using HepMC as the event record
- pythia - illustrating the usage of Pythia as Monte Carlo event generator, interfaced with Geant4, and showing how to implement an external decayer (example *decayer6*)

Exotic Physics

- Channeling - simulates channeling of 400 GeV/c protons in a bent crystal.
- Monopole - illustrating how to measure energy deposition in classical magnetic monopole
- Phonon - demonstrates simulation of phonon propagation in cryogenic crystals
- UCN - simulates the passage of ultra-cold neutrons (UCN) in a hollow pipe.

Fields

- BlineTracer - tracing and visualizing magnetic field lines
- field01 - tracking using magnetic field and field-dependent processes
- field02 - tracking using electric field and field-dependent processes
- field03 - tracking in a magnetic field where field associated with selected logical volumes varies
- field04 - definition of overlapping fields either magnetic, electric or both
- field05 - demonstration of "spin-frozen" condition, how to cancel the muon g-2 precession by applying an electric field
- field06 - exercising the capability of tracking massive particles in a gravity field

Geant3 to Geant4

- General ReadMe - converting simple geometries in Geant3.21 to their Geant4 equivalents (example *clGeometry*)

Geometry

- General ReadMe
- transforms - demonstrating various ways of definition of 3D transformations for placing volumes

Hadronic

- Hadr00 - example demonstrating the usage of G4PhysListFactory to build physics lists and usage of G4HadronicProcessStore to access the cross sections
- Hadr01 - example based on the application IION developed for simulation of proton or ion beam interaction with a water target. Different aspects of beam target interaction are included
- Hadr02 - example application providing simulation of ion beam interaction with different targets. Hadronic aspects of beam target interaction are demonstrated including longitudinal profile of energy deposition, spectra of secondary particles, isotope production spectra.

- Hadr03 - example demonstrating how to compute total cross section from the direct evaluation of the mean free path, how to identify nuclear reactions and how to plot energy spectrum of secondary particles
- Hadr04 - example focused on neutronHP physics, especially neutron transport, including thermal scattering
- Hadr05 - demonstrates the usage of G4GenericPhysicsList to build the concrete physics list at the run time
- Hadr06 - demonstrates survey of energy deposition and particle's flux from a hadronic cascade
- Hadr07 - demonstrates survey of energy deposition and particle's flux from a hadronic cascade. Show how to plot a depth dose profile in a rectangular box.
- FissionFragment - This example demonstrates the Fission Fragment model as used within the neutron_hp model. It will demostrate the capability for fission product containmentby the cladding in a water moderated sub-critical assembly. It could also be further extended to calculate the effective multiplication factor of the subcritical assembly for various loading schemes.
- NeutronSource - NeutronSource is an example of neutrons production. It illustrates the cooperative work of nuclear reactions and radioactive decay processes. It surveys energy deposition and particle's flux. It uses PhysicsConstructor objects.

Medical Applications

- DICOM - geometry set-up using the Geant4 interface to the DICOM image format
- electronScattering - benchmark on electron scattering
- electronScattering2 - benchmark on electron scattering (second way to implement the same benchmark as the above)
- fanoCavity - dose deposition in an ionization chamber by a monoenergetic photon beam
- fanoCavity2 - dose deposition in an ionization chamber by an extended one-dimensional monoenergetic electron source
- GammaTherapy - gamma radiation field formation in water phantom by electron beam hitting different targets
- dna - Set of examples using the Geant4-DNA physics processes and models.
  - dnaphysics - The dnaphysics example shows how to simulate track structures in liquid water using the Geant4-DNA physics processes and models.
  - microdosimetry - The microdosimetry example simulates the track of a 5 MeV proton in liquid water. Geant4 standard EM models are used in the World volume while Geant4-DNA models are used in a Target volume, declared as a Region.
  - range - Simulation of ranges.
  - svalue - This example shows how to simulate S-values in spheres of liquid water using the Geant4-DNA physics processes and models.
  - wvalue - This example shows how to simulate W-values in liquid water using the Geant4-DNA physics processes and models.
  - chem1 - Simple activation of the chemistry module.
  - chem2 - Usage of TimeStepAction in the chemistry module.
  - chem3 - Activate the full interactivity with the chemistry module.
  - chem4 - Simulation of G radiochemical yields with the chemistry module.
  - wholeNuclearDNA - Description of the full nucleus of a biological cell.
  - pdb4dna - Usage of the Protein Data Bank (PDB) file format to build geometries.
  - clustering - Clustering application for direct damage extraction.

Optical Photons

- General ReadMe
- OpNovice - simulation of optical photons generation and transport. (It was moved in extended examples from novice/N06 with removal of novice examples.)
- LXe - optical photons in a liquid xenon scintillator
- WLS - application simulating the propagation of photons inside a Wave Length Shifting (WLS) fiber

Parallel Computing

- General ReadMe
- MPI - interface and examples of applications (*exMPI01*, *exMPI02* and *exMPI03*) parallelized with different MPI compliant libraries, such as LAM/MPI, MPICH2, OpenMPI, etc.
- TBB - demonstrate how to interface a simple application with the Intel Threading Building Blocks library (TBB), and organise MT event-level parallelism as TBB tasks
- ThreadsafeScorers - demonstrates a very simple application where an energy deposit and # of steps is accounted in thread-local (i.e. one instance per thread) hits maps with underlying types of plain-old data (POD) and global (i.e. one instance) hits maps with underlying types of atomics.
- TopC - set of examples (*ParN02* and *ParN04*) derived from `novice` using parallelism at event level with the TopC application

Parameterisations

- Par01 - Demonstrates the use of parameterisation facilities. (It was moved in extended examples from novice/N05 with removal of novice examples.)
- Par02 - Shows how to do "track and energy smearing" in Geant4, in order to have a very fast simulation based on assumed detector resolutions.
- Gflash - Demonstrates the use of the GFLASH parameterisation library. It uses the GFLASH equations(hep-ex/0001020, Grindhammer & Peters) to parametrise electromagnetic showers in matter

Persistency

- General ReadMe
- GDML - examples set (*G01*, *G02*, *G03* and *G04*) illustrating import and export of a detector geometry with GDML, and how to extend the GDML schema or use the auxiliary information field for defining additional persistent properties
- P01 - storing calorimeter hits using reflection mechanism with Root
- P02 - storing detector description using reflection mechanism with Root
- P03 - illustrating import and export of a detector geometry using ASCII text description and syntax

Polarisation

- Pol01 - interaction of polarized beam (e.g. circularly polarized photons) with polarized target

Radioactive Decay

- rdecay01 - demonstrating basic functionality of the `G4RadioactiveDecay` process
- rdecay02 (Exrdm) - decays of radioactive isotopes as well as induced radioactivity resulted from nuclear interactions
- Activation - compute and plot time evolution of each nuclide in an hadronic cascade. Compute and plot activity of emerging particles.

Run & Event

- RE01 - information between primary particles and hits and usage of user-information classes
- RE02 - simplified fixed target application for demonstration of primitive scorers
- RE03 - use of UI-command based scoring; showing how to create parallel world(s) for defining scoring mesh(es)
- RE04 - demonstrating how to define a layered mass geometry in parallel world

- RE05 - demonstrating interfacing to the PYTHIA primary generator, definition of a 'readout' geometry, event filtering using the stacking mechanism. (It was moved in extended examples from novice/N04 with removal of novice examples.)
- RE06 - demonstrating how to modify part of the geometry setup at run-time, detector description parameterisation by materials, sharing of a sensitive detector definition for different sub-detectors, different geometrical regions definition with different production thresholds, customization of the G4Run (It was moved in extended examples from novice/N07 with removal of novice examples.)

Visualization

- General ReadMe - examples (*perspective*, *standalone* and *userVisAction*) of customisation for visualization

10.4 Advanced Examples

Geant4 advanced examples illustrate realistic applications of Geant4 in typical experimental environments. Most of them also show the usage of analysis tools (such as histograms, ntuples and plotting), various visualization features and advanced user interface facilities, together with the simulation core.

Note: Maintenance and updates of the code is under the responsibility of the authors. These applications are therefore not subject to regular system testing and no guarantee can be provided.

The advanced examples include:

- air_shower, Simulation of a Fresnel lens focusing direct or reflected UV light onto a photomultiplier. Object parameterisation and replication capabilities of Geant4 are used to describe the lens geometry. The example is inspired in the configuration of the ULTRA experiment (NIM A 570 (2007) 22).
- amsEcal, illustrating simulation in the AMS electro-magnetic calorimeter.
- brachytherapy, illustrating a typical medical physics application simulating energy deposit in a Phantom filled with soft tissue.
- ChargeExchangeMC, The program was used to simulate real experiments in Petersburg Nuclear Physics Institute (PNPI, Russia).
- composite_calorimeter, test-beam simulation of the CMS Hadron calorimeter at LHC.
- dna_physics, this example explains how to use Geant4-DNA physics for the very low energy transport of particles in liquid water. See more information at http://geant4-dna.org.
- eRosita, simplified version of the simulation of the shielding of the eROSITA X-ray mission; it demonstrates the simulation of PIXE (Particle Induced X-ray Emission) as described in M.G. Pia et al., PIXE simulation with Geant4, IEEE Trans. Nucl. Sci., vol. 56, no. 6, pp. 3614-3649, 2009.
- gammaknife, reproducing in details a gammaknife device for stereotactic radiosurgery. In particular, the gammaknife model C is simulated, which is characterized by a symmetrical displacement of the Co60 sources. Dose distributions are acquired in a water spherical phantom using voxelized geometries. The possibility to change the source pattern in order to simulate different gammaknife models is in development and new versions with these additional features will be released.
- gammaray_telescope, illustrating an application to typical gamma ray telescopes with a flexible configuration.
- hadrontherapy, is an example for people interested in Monte Carlo studies related to proton/ion therapy. Hadrontherapy permits the simulation of a typical hadron therapy beam line (with all its elements) and the calculation of fundamentals quantities of interest: 3D dose distributions, fluences, and average LET for both primary and secondary particles, etc.. A typical beamline for laser-driven ion beams is also included in this last version.
- human_phantom, implementing an Anthropomorphic Phantom body built importing the description from a GDML representation.
- iort_therapy, specifically developed to address typical needs related to the IntraOperative Radio-Therapy (IORT) technique. This technique delivers a single dose of radiation directly to the tumor bed, or to the exposed tumor, during surgery. The idea of iort_therapy is to provide a useful tool for Users interested to radiation dosimetry,

dose planning and radio-protection studies in IORT. In fact, the application allows to reconstruct dose distribution curves in water or other materials, to plan dose distribution in the tumor treatment region with different clinical set-up, and to optimize radio-protection of normal patient tissues simulating a composite metallic shielding disc. iort_therapy simulates the collimator beam line system of a typical medical mobile linac, the phantom, the detector and the composite metallic shielding disc. Via external macro commands it is possible to change the physic models, the collimator beam line, the phantom, the detector and shielding disc geometries, the visualization, the beam particle characteristics, and to activate the Graphical Users Interface (QT libraries are requested)

- lAr_calorimeter, simulating the Forward Liquid Argon Calorimeter (FCAL) of the ATLAS Detector at LHC.
- medical_linac, illustrating a typical medical physics application simulating energy deposit in a Phantom filled with water for a typical linac used for intensity modulated radiation therapy. The experimental set-up is very similar to one used in clinical practice.
- microbeam, simulates the cellular irradiation beam line installed on the AIFIRA electrostatic accelerator facility located at CENBG, Bordeaux-Gradignan, France.
- microelectronics, simulates the track of a 5 MeV proton in silicon using very low energy electromagnetic Geant4 MicroElec processes. It illustrates how to combine these discrete processes with usual Geant4 condensed history ones, using different processes for different regions of the geometry and different energy ranges.
- nanobeam, simulates the beam optics of the "nanobeam line" installed on the AIFIRA electrostatic accelerator facility located at CENBG, Bordeaux-Gradignan, France.
- purging_magnet, illustrating an application that simulates electrons traveling through a 3D magnetic field; used in a medical environment for simulating a strong purging magnet in a treatment head.
- radioprotection, illustrating the response characterization of a novel diamond microdosimeter for radiation protection in human space missions and aviation.
- underground_physics, illustrating an underground detector for dark matter searches.
- xray_fluorescence, illustrating the emission of X-ray fluorescence and PIXE.
- xray_telescope, illustrating an application for the study of the radiation background in a typical X-ray telescope.

10.5 Novice Examples

The old "novice" set of examples is now replaced with a new "basic" set, covering the most typical use-cases of a Geant4 application with keeping simplicity and ease of use.

The source code of the last version of the novice examples set (in 9.6.p02 release) can be viewed in the Geant4 LXR code browser

The new location of each example in 10.0 release:

- N01 - removed
- N02 - basic/B2
- N03 - basic/B4
- N04 - extended/runAndEvent/RE05
- N05 - extended/parameterisations/Par01
- N06 - extended/optical/OpNovice
- N07 - extended/runAndEvent/RE06

CHAPTER

ELEVEN

APPENDIX

11.1 CLHEP Foundation Library

CLHEP is a set of Class Libraries containing many basic classes for use in High Energy Physics.

Both a CLHEP Reference Guide and a User Guide are available.

Origin and current situation of CLHEP

CLHEP started in 1992 as a library for fundamental classes mostly needed for, and in fact derived from, the MC event generator MC++ written in C++. Since then various authors added classes to this package, including several contributions made by developers in the Geant4 Collaboration.

Geant4 and CLHEP

The Geant4 project contributed to the development of CLHEP. The random number package, physics units and constants, and some of the numeric and geometry classes had their origins in Geant4.

Geant4 also benefits from the development of CLHEP. In addition to the already mentioned classes for random numbers and numerics, we use the classes for points, vectors, and planes and their transformations in 3D space, and lorentz vectors and their transformations. Although these classes have Geant4 names like G4ThreeVector, these are just typedefs to the CLHEP classes.

Since release 9.5 of Geant4, the relevant classes of the CLHEP libraries are distributed as embedded module within Geant4. It is therefore no longer necessary to build and link against an external CLHEP installation (solution which is still supported as option).

11.2 Geant4Config.cmake CMake Config File

11.2.1 Usage of Geant4Config.cmake

`Geant4Config.cmake` is designed to be used with CMake's find_package command. When found, it sets several CMake variables and provides a mechanism for checking and activating optional features of Geant4. This allows you to use it in many ways in your CMake project to configure Geant4 for use by your application.

The most basic usage of `Geant4Config.cmake` in a CMakeLists.txt file is just to locate Geant4 with no requirements on its existence, version number or components:

```
find_package(Geant4)
```

If you must find Geant4, then you can use:

```
find_package(Geant4 REQUIRED)
```

This will cause CMake to fail with an error should an install of Geant4 not be located.

When an install of Geant4 is found, the module sets a sequence of CMake variables that can be used elsewhere in the project:

- `Geant4_FOUND` Set to CMake boolean true if an install of Geant4 was found.
- `Geant4_INCLUDE_DIRS` Set to a list of directories containing headers needed by Geant4. May contain paths to third party headers if these appear in the public interface of Geant4.
- `Geant4_LIBRARIES` Set to the list of libraries that need to be linked to an application using Geant4.
- `Geant4_DEFINITIONS` The list of compile definitions needed to compile an application using Geant4. This is most typically used to correctly activate UI and Visualization drivers.
- `Geant4_CXX_FLAGS` The compiler flags used to build this install of Geant4. Usually most important on Windows platforms.
- `Geant4_CXX_FLAGS_<CONFIG>` The compiler flags recommended for compiling Geant4 and applications in mode `CONFIG` (e.g. Release, Debug, etc). Usually most important on Windows platforms.
- `Geant4_CXXSTD` The C++ standard, e.g. "`c++11`" against which this install of Geant4 was compiled.
- `Geant4_TLS_MODEL` The thread-local storage model, e.g. "`initial-exec`" against which this install of Geant4 was compiled. Only set if the install was compiled with multithreading support.
- `Geant4_USE_FILE` A CMake script which can be included to handle certain CMake steps automatically. Most useful for very basic applications.
- `Geant4_builtin_clhep_FOUND` A CMake boolean which is set to true if this install of Geant4 was built using the internal CLHEP.
- `Geant4_system_clhep_ISGRANULAR` A CMake boolean which is set to true if this install of Geant4 was built using the system CLHEP and linked to the granular CLHEP libraries.
- `Geant4_builtin_expat_FOUND` A CMake boolean which is set to true if this install of Geant4 was built using the internal Expat.
- `Geant4_builtin_zlib_FOUND` A CMake boolean which is set to true if this install of Geant4 was built using the internal zlib.
- `Geant4_DATASETS` A CMake list of the names of the physics datasets used by physics models in Geant4. It is provided to help iterate over the `Geant4_DATASET_XXX_YYY` variables documented below.
- `Geant4_DATASET_<NAME>_ENVVAR` The name of the environment variable used by Geant4 to locate the dataset with name `<NAME>`.
- `Geant4_DATASET_<NAME>_PATH` The absolute path to the dataset with name `<NAME>`. Note that the setting of this variable does not guarantee the existence of the dataset, and no checking of the path is performed. This checking is not provided because the action you take on non-existing data will be application dependent. You can access the `Geant4_DATASET_XXX_YYY` variables in a CMake script in the following way:

```
find_package(Geant4_REQUIRED)                 # Find Geant4

foreach(dsname ${Geant4_DATASETS})            # Iterate over dataset names
  if(NOT EXISTS ${Geant4_DATASET_${dsname}_PATH})  # Check existence
    message(WARNING "${dsname} not located at ${Geant4_DATASET_${dsname}_PATH}")
  endif()
endforeach()
```

  A typical use case for these variables is to automatically set the dataset environment variables for your application without the need to preconfigure the environment. This could typically be via a shell script wrapper around your application, or runtime configuration of the application environment via the relevant C/C++ API for your system.

The typical usage of `find_package` and these variables to configure a build requiring Geant4 is thus:

```
find_package(Geant4 REQUIRED)                       # Find Geant4
include_directories(${Geant4_INCLUDE_DIRS})         # Add -I type paths
add_definitions(${Geant4_DEFINITIONS})              # Add -D type defs
set(CMAKE_CXX_FLAGS ${Geant4_CXX_FLAGS})            # Optional

add_executable(myg4app myg4app.cc)                  # Compile application
target_link_libraries(myg4app ${Geant4_LIBRARIES})  # Link it to Geant4
```

Alternatively, the CMake script pointed to by Geant4_USE_FILE may be included:

```
find_package(Geant4 REQUIRED)                        # Find Geant4
include(${Geant4_USE_FILE})                          # Auto configure includes/flags

add_executable(myg4app myg4app.cc)                   # Compile application
target_link_libraries(myg4app ${Geant4_LIBRARIES})   # Link it to Geant4
```

When included, the Geant4_USE_FILE script performs the following actions:

1. Adds the definitions in Geant4_DEFINITIONS to the global compile definitions.
2. Appends the directories listed in Geant4_INCLUDE_DIRS to those the compiler uses for search for include paths, marking them as system include directories.
3. Prepends Geant4_CXX_FLAGS to CMAKE_CXX_FLAGS, and similarly for the extra compiler flags for each build mode (Release, Debug etc).

This use file is very useful for basic applications, but if your use case requires finer control over compiler definitions, include paths and flags you should use the relevant Geant4_NAME variables directly.

By default, CMake will look in several platform dependent locations for the Geant4Config.cmake file (see find_package for listings). You can also specify the location directly when running CMake by setting the Geant4_DIR variable to the path of the directory holding Geant4Config.cmake. It may be set on the command line via a -D option, or by adding an entry to the CMake GUI. For example, if we have an install of Geant4 located in

```
+- opt/
   +- Geant4/
   +- lib/
      +- libG4global.so
      +- ...
      +- Geant4-G4VERSION/
         +- Geant4Config.cmake
```

then we would pass the argument -DGeant4_DIR=/opt/Geant4/lib/Geant4-G4VERSION to CMake. The CMAKE_PREFIX_PATH variable may also be used to point CMake to Geant4 by adding, to take the example above, /opt/Geant4 to the list of paths it holds. This may be set either on the command line or as a path-style UNIX environment variable.

You can also, if you wish, build an application against a build of Geant4 *without* installing it. If you look in the directory where you built Geant4 itself (e.g. on UNIX, where you ran make), you see there is a Geant4Config.cmake file. This is a perfectly valid file, so you can also point CMake to this file when building your application. Simply set Geant4_DIR to the directory where you built Geant4. This feature is most useful for Geant4 developers, but it can be useful if you cannot, or do not want to, install Geant4.

A version number may also be supplied to search for a Geant4 install *greater than or equal to* the supplied version, e.g.

```
find_package(Geant4 10.0 REQUIRED)
```

would make CMake search for a Geant4 install whose version number is greater than or equal to 10.0. An exact version number may also be specified:

```
find_package(Geant4 G4VERSION EXACT REQUIRED)
```

In both cases, CMake will fail with an error if a Geant4 install meeting these version requirements is not located.

Geant4 can be built with many optional components, and the presence of these can also be required by passing extra "component" arguments. For example, to require that Geant4 is found *and* that it support Qt UI and visualization, we can do

```
find_package(Geant4 REQUIRED qt)
```

In this case, if CMake finds a Geant4 install that does *not* support Qt, it will fail with an error. Multiple component arguments can be supplied, for example

```
find_package(Geant4 REQUIRED qt gdml)
```

requires that we find a Geant4 install that supports both Qt and GDML. If the component(s) is(are) found, any needed header paths, libraries and compile definitions required to use the component are appended to the variables `Geant_INCLUDE_DIRS`, `Geant4_LIBRARIES` and `Geant4_DEFINITIONS` respectively. Variables `Geant4_<COMPONENTNAME>_FOUND` are set to `TRUE` if component `COMPONENTNAME` is supported by the installation.

If you want to activate options only if they exist, you can use the pattern

```
find_package(Geant4 REQUIRED)
find_package(Geant4 QUIET COMPONENTS qt)
```

which will require CMake to locate a core install of Geant4, and then check for and activate Qt support if the install provides it, continuing without error otherwise. A key thing to note here is that you can call `find_package` multiple times to append configuration of components. If you use this pattern and need to check if a component was found, you can use the `Geant4_<COMPONENTNAME>_FOUND` variables described earlier to check the support.

The components which can be supplied to `find_package` for Geant4 are as follows:

static `Geant4_static_FOUND` is `TRUE` if the install of Geant4 provides static libraries.
Use of this component forces the variable `Geant4_LIBRARIES` to contain static libraries, if they are available. It can therefore be used to force static linking if your application requires this, but note that this does not guarantee that static version of third party libraries will be used.

multithreaded `Geant4_multithreaded_FOUND` is `TRUE` if the install of Geant4 was built with multithreading support.
Note that this only indicates availability of multithreading support and activates the required compiler definition to build a multithreaded Geant4 application. Multithreading in your application requires creation and usage of the appropriate C++ objects and interfaces as described in the Application Developers Guide.

usolids `Geant4_usolids_FOUND` is `TRUE` if the install of Geant4 was built with USolids replacing the Geant4 solids.
Note that this only indicates that the replacement of Geant4 solids with USolids has taken place. Further information on the use of USolids applications is provided in the Application Developers Guide.

gdml `Geant4_gdml_FOUND` is `TRUE` if the install of Geant4 was built with GDML support.

g3tog4 `Geant4_g3tog4_FOUND` is `TRUE` if the install of Geant4 provides the G3ToG4 library. If so, the G3ToG4 library is added to `Geant4_LIBRARIES`.

freetype `Geant4_freetype_FOUND` is `TRUE` if the install of Geant4 was built with Freetype support.

ui_tcsh `Geant4_ui_tcsh_FOUND` is `TRUE` if the install of Geant4 provides the TCsh command line User Interface. Using this component allows use of the TCsh command line interface in the linked application.

ui_win32 `Geant4_ui_win32_FOUND` is `TRUE` if the install of Geant4 provides the Win32 command line User Interface. Using this component allows use of the Win32 command line interface in the linked application.

motif `Geant4_motif_FOUND` is `TRUE` if the install of Geant4 provides the Motif(Xm) User Interface and Visualization driver. Using this component allows use of the Motif User Interface and Visualization Driver in the linked application.

qt `Geant4_qt_FOUND` is `TRUE` if the install of Geant4 provides the Qt4 User Interface and Visualization driver. Using this component allows use of the Qt User Interface and Visualization Driver in the linked application.

wt `Geant4_wt_FOUND` is `TRUE` if the install of Geant4 provides the Wt Web User Interface and Visualization driver. Using this component allows use of the Wt User Interface and Visualization Driver in the linked application.

vis_network_dawn `Geant4_vis_network_dawn_FOUND` is `TRUE` if the install of Geant4 provides the Client/Server network interface to DAWN visualization. Using this component allows use of the Client/Server

DAWN Visualization Driver in the linked application.

vis_network_vrml `Geant4_vis_network_vrml_FOUND` is `TRUE` if the install of Geant4 provides the Client/Server network interface to VRML visualization. Using this component allows use of the Client/Server VRML Visualization Driver in the linked application.

vis_opengl_x11 `Geant4_vis_opengl_x11_FOUND` is `TRUE` if the install of Geant4 provides the X11 interface to the OpenGL Visualization driver. Using this component allows use of the X11 OpenGL Visualization Driver in the linked application.

vis_opengl_win32 `Geant4_vis_opengl_win32_FOUND` is `TRUE` if the install of Geant4 provides the Win32 interface to the OpenGL Visualization driver. Using this component allows use of the Win32 OpenGL Visualization Driver in the linked application.

vis_openinventor `Geant4_vis_openinventor_FOUND` is `TRUE` if the install of Geant4 provides the OpenInventor Visualization driver. Using this component allows use of the OpenInventor Visualization Driver in the linked application.

ui_all Activates all available UI drivers. Does not set any variables, and never causes CMake to fail.

vis_all Activates all available Visualization drivers. Does not set any variables, and never causes CMake to fail.

Please note that whilst the above aims to give a complete summary of the functionality of `Geant4Config.cmake`, it only gives a sampling of the ways in which you may use it, and other CMake functionality, to configure your application. We also welcome feedback, suggestions for improvement and bug reports on `Geant4Config.cmake`.

11.2.2 Going further with CMake

The preceeding sections show the minimal CMake scripting required to configure, build and install an application linking against the Geant4 libraries. If your project requires more advanced configuration, CMake provides tools such as compiler/platform identification and location of additional libraries/executables to link to/use. As this document is specific to Geant4, we do not cover more advanced usage of CMake and recommend that you consult the online manuals and tutorials supplied by Kitware.

In particular, for the common use case of finding and using an external software package, see the documentation of the `find_package` command, overview of CMake's package location functionality, and the list of packages CMake knows about out of the box. Location and use of a required package works exactly as we have illustrated for Geant4. Simply add the required `find_package` call to your CMake script, and use the supplied variables or targets for headers paths and library linking, e.g.

```
find_package(Foo 1.2 REQUIRED)        # Find "Foo" of at least version 1.2
find_package(Bar 3.4 EXACT REQUIRED)  # Find "Bar" at exactly version 3.4

include_directories(${Foo_INCLUDE_DIRS}) # Foo's setup supplies a header path

add_library(MyLibrary SHARED MyLibrary.cc) # Define our library
target_link_libraries(MyLibrary           # Link it
  ${Foo_LIBRARIES}                        # Foo's setup supplies a library path
  Bar::Bar                                # Bar's setup supplies an "IMPORTED" target
  )                                       # which sets header and library paths automatically
```

You should consult the documentation of the packages your project requires to see if they supply suitable CMake configuration files. If they do not, then CMake provide documentation on writing modules to find packages that do not supply these files. Geant4 cannot provide support for any third party package your project uses, and any questions should be direct to that package's authors.

11.2.3 Building an Application against a Build of Geant4

A typical use case for Geant4 developers is to build small testing applications against a fresh build of Geant4. If rebuilds are frequent, then the testing application builds are also frequent.

CMake can be used to build these test applications using `find_package` and `Geant4Config.cmake`, as a special version of the latter is created in the Geant4 build directory. This sets up the variables to point to the headers in the Geant4 source directory, and the freshly built libraries in the current build directory.

Applications may therefore be built against a non-installed build of Geant4 by running CMake for the application and setting `Geant4_DIR` to point to the current build directory of Geant4.

11.3 GNUMake System: Makefiles and Environment Variables

This section describes how the Geant4 GNUMake infrastructure is implemented in Geant4 and provides a quick reference guide for the user about the most important environment variables that can be set to configure its behaviour.

This system is now deprecated, though it is still provided through the SVN repository for developers, and is installed by CMake to provide temporary backward compatibility for user applications.

11.3.1 Geant4Make System

As described in *How to Define the main() Program* of the Installation Guide, the GNUmake process in Geant4 is mainly controlled by the following GNUmake script files (`*.gmk` scripts are placed in `$G4INSTALL/config`):

- `architecture.gmk`: defining all the architecture specific settings and paths. System settings are stored in `$G4INSTALL/config/sys` in separate files.
- `common.gmk`: defining all general GNUmake rules for building objects and libraries.
- `globlib.gmk`: defining all general GNUmake rules for building compound libraries.
- `binmake.gmk`: defining the general GNUmake rules for building executables.
- `GNUmake` scripts: placed inside each directory in the G4 distribution and defining directives specific to build a library (or a set of sub-libraries) or and executable.

To build a single library (or a set of sub-libraries) or an executable, you must explicitly change your current directory to the one you're interested in and invoke the "`make`" command from there ("`make global`" for building a compound library). Here is a list of the basic commands or GNUmake "targets" one can invoke to build libraries and/or executables:

- `make`
  starts the compilation process for building a kernel library or a library associated with an example. Kernel libraries are built with maximum granularity, i.e. if a category is a compound, this command will build all the related sub-libraries, **not** the compound one. The top level `GNUmakefile` in `$G4INSTALL/source` will also build in this case a dependency map `libname.map` of each library to establish the linking order automatically at the `bin` step. The map will be placed in `$G4LIB/$G4SYSTEM`.
- `make global`
  starts the compilation process to build a single compound kernel library per category. If issued after "make", both 'granular' and 'compound' libraries will be available (NOTE: this will consistently increase the disk space required. Compound libraries will then be selected by default at link time, unless G4LIB_USE_GRANULAR is specified).
- `make bin` or `make` (only for examples/)
  starts the compilation process to build an executable. This command will build implicitly the library associated with the example and link the final application. It assumes **all** kernel libraries are already generated and placed in the correct `$G4INSTALL` path defined for them.
  The linking order is controlled automatically in case libraries have been built with maximum granularity, and the link list is generated on the fly.
- `make dll`
  On Windows systems this will start the compilation process to build single compound kernel library per category and generate Dynamic Link Libraries (DLLs). Once the libraries are generated, the process will imply also the deletion of all temporary files generated during the compilation.

lib/ bin/ and tmp/ directories

The $G4INSTALL environment variable specifies where the installation of the Geant4 toolkit should take place, therefore kernel libraries will be placed in $G4INSTALL/lib. The $G4WORKDIR environment variable is set by the user and specifies the path to the user working directory; temporary files (object-files and data products of the installation process of Geant4) will be placed in $G4WORKDIR/tmp, according to the system architecture used. Binaries will be placed in $G4WORKDIR/bin, according to the system architecture used. The path to $G4WORKDIR/bin/$G4SYSTEM should be added to $PATH in the user environment.

11.3.2 Environment variables

Here is a list of the most important environment variables defined within the Geant4 GNUmake infrastructure, with a short explanation of their use.

We recommend that those environment variables listed here and marked with (*) NOT be overriden or set (explicitly or by accident). They are already set and used internally in the default setup !

System configuration

$CLHEP_BASE_DIR Specifies the path where the CLHEP package is installed in your system.
$USOLIDS_BASE_DIR Specifies the path where the USolids package is installed in your system.
$G4SYSTEM Defines the architecture and compiler currently used.

Note: This variable is set automatically if the Configure script is adopted for the installation. This will result in the proper settings also for configuring the environment with the generated shell scripts env.[c]sh.

Installation paths

$G4INSTALL Defines the path where the Geant4 toolkit is located. It should be set by the system installer. By default, it sets to $HOME/geant4, assuming the Geant4 distribution is placed in $HOME .

$G4BASE (*) Defines the path to the source code. Internally used to define $CPPFLAGS and $LDFLAGS for -I and -L directives. It has to be set to $G4INSTALL/src.

$G4WORKDIR Defines the path for the user's workdir for Geant4. It is set by default to $HOME/geant4, assuming the user's working directory for Geant4 is placed in $HOME.

$G4INCLUDE Defines the path where source header files may be mirrored at installation by issuing gmake includes (default is set to $G4INSTALL/include)

$G4BIN, $G4BINDIR (*) Used by the system to specify the place where to store executables. By default they're set to $G4WORKDIR/bin and $G4BIN/$G4SYSTEM respectively. The path to $G4WORKDIR/bin/$G4SYSTEM should be added to $PATH in the user environment. $G4BIN can be overridden.

$G4TMP, $G4TMPDIR (*) Used by the system to specify the place where to store temporary files products of the compilation/build of a user application or test. By default they're set to $G4WORKDIR/tmp and $G4TMP/$G4SYSTEM respectively. $G4TMP can be overridden.

$G4LIB, $G4LIBDIR (*) Used by the system to specify the place where to install libraries. By default they're set to $G4INSTALL/lib and $G4LIB/$G4SYSTEM respectively. $G4LIB can be overridden.

Build specific

$G4TARGET Specifies the target (name of the source file defining the main()) of the application/example to be built. This variable is set automatically for the examples and tests placed in $G4INSTALL/examples .

$G4DEBUG Specifies to compile the code (libraries or examples) including symbolic information in the object code for debugging. The size of the generated object code can increase considerably. By default, code is compiled in optimised mode ($G4OPTIMISE set).

$G4OPTDEBUG Only available for the g++ compiler, specifies to compile the code (libraries or examples) in optimised mode, but including symbolic information in the object code for debugging.

$G4USE_STD11 Specifies to compile the code (libraries or examples) with C++11 Standard enabled on compilers supporting the C++11 Standard.

$G4NO_OPTIMISE Specifies to compile the code (libraries or examples) without compiler optimisation.

$G4PROFILE On Linux systems with the `g++` compiler, it allows to build libraries with profiling setup for monitoring with the `gprof` tool.

$G4_NO_VERBOSE Geant4 code is compiled by default in high verbosity mode (`$G4VERBOSE` flag set). For better performance, verbosity code can be left out by defining `$G4_NO_VERBOSE`.

$G4LIB_BUILD_SHARED Flag specifying if to build kernel libraries as shared libraries (libraries will be then used by default). If not set, static archive libraries are built by default.

$G4LIB_BUILD_STATIC Flag specifying if to build kernel libraries as static archive libraries in addition to shared libraries (in case `$G4LIB_BUILD_SHARED` is set as well).

$G4LIB_BUILD_DLL (*) Internal flag for specifying to build DLL kernel libraries for Windows systems. The flag is automatically set when requested to build DLLs.

$G4LIB_USE_DLL For Windows systems only. Flag to specify to build an application using the installed DLL kernel libraries for Windows systems. It is required to have this flag set in the environment in order to successfully build an application if the DLL libraries have been installed.

$G4LIB_USE_GRANULAR To force usage of "granular" libraries against "compound" libraries at link time in case both have been installed. The Geant4 building system chooses "compound" libraries by default, if installed.

UI specific

The most relevant flags for User Interface drivers are just listed here. A more detailed description is given also in section 2. of this User's Guide.

G4UI_USE_TERMINAL Specifies to use dumb terminal interface in the application to be built (default).

G4UI_USE_TCSH Specifies to use the tcsh-shell like interface in the application to be built.

G4UI_BUILD_XM_SESSION Specifies to include in kernel library the `XM` Motif-based user interfaces.

G4UI_USE_XM Specifies to use the `XM` interfaces in the application to be built.

G4UI_BUILD_WIN32_SESSION Specifies to include in kernel library the WIN32 terminal interface for Windows systems.

G4UI_USE_WIN32 Specifies to use the WIN32 interfaces in the application to be built on Windows systems.

G4UI_BUILD_QT_SESSION Specifies to include in kernel library the Qt terminal interface. `$QTHOME` should specify the path where Qt libraries and headers are installed

G4UI_USE_QT Specifies to use the Qt interfaces in the application to be built.

G4UI_NONE If set, no UI sessions nor any UI libraries are built. This can be useful when running a pure batch job or in a user framework having its own UI system.

Visualization specific

The most relevant flags for visualization graphics drivers are just listed here. A description of these variables is given also in section 2. of this User's Guide.

$G4VIS_BUILD_OPENGLX_DRIVER Specifies to build kernel library for visualization including the OpenGL driver with X11 extension. It requires `$OGLHOME` set (path to OpenGL installation).

$G4VIS_USE_OPENGLX Specifies to use OpenGL graphics with X11 extension in the application to be built.

$G4VIS_BUILD_OPENGLXM_DRIVER Specifies to build kernel library for visualization including the OpenGL driver with XM extension. It requires `$OGLHOME` set (path to OpenGL installation).

$G4VIS_USE_OPENGLXM Specifies to use OpenGL graphics with XM extension in the application to be built.

G4VIS_BUILD_OPENGLQT_DRIVER Specifies to build kernel library for visualization including the OpenGL driver with Qt extension. It requires `$QTHOME` set to specify the path where Qt libraries and headers are installed.

G4VIS_USE_OPENGLQT Specifies to use OpenGL graphics with Qt extension in the application to be built.

$G4VIS_BUILD_OI_DRIVER Specifies to build kernel library for visualization including the OpenInventor driver. It requires `$OIHOME` set (paths to the OpenInventor installation).

$G4VIS_USE_OI Specifies to use OpenInventor graphics in the application to be built.

$G4VIS_BUILD_OIX_DRIVER Specifies to build the driver for the free X11 version of OpenInventor.

$G4VIS_USE_OIX Specifies to use the free X11 version of OpenInventor.

$G4VIS_BUILD_RAYTRACERX_DRIVER Specifies to build kernel library for visualization including the Ray-Tracer driver with X11 extension. It requires `X11` installed in the system.
$G4VIS_USE_RAYTRACERX Specifies to use the X11 version of the Ray-Tracer driver.
$G4VIS_BUILD_OIWIN32_DRIVER Specifies to build the driver for the free X11 version of OpenInventor on Windows systems.
$G4VIS_USE_OIWIN32 Specifies to use the free X11 version of OpenInventor on Windows systems.
$G4VIS_BUILD_DAWN_DRIVER Specifies to build kernel library for visualization including the driver for DAWN.
$G4VIS_USE_DAWN Specifies to use DAWN as a possible graphics renderer in the application to be built.
$G4DAWN_HOST_NAME To specify the hostname for use with the DAWN-network driver.
$G4VIS_NONE If specified, no visualization drivers will be built or used.

Hadronic physics specific

$G4NEUTRONHP_USE_ONLY_PHOTONEVAPORATION When using high precision neutron code, user may choose to force the use of Photon Evaporation model instead of using the neutron capture final state data.
$G4NEUTRONHP_SKIP_MISSING_ISOTOPES User can force high precison neutron code to use only exact isotope data files instead of allowing nearby isotope files to be used. If the exact file is not available, the cross section will be set to zero and a warning message will be printed.
$G4NEUTRONHP_NEGLECT_DOPPLER Sets neglecting doppler broadening mode for boosting performance.

GDML, zlib and g3tog4 modules

$G4LIB_BUILD_GDML If set, triggers compilation of a plugin module `gdml` for allowing import/export of detector description setups (geometrical volumes, solids, materials, etc.). By default, the flag is not set; if set, the path to the installation of XercesC package must be specified through the variable `$XERCESCROOT`.
$G4LIB_USE_GDML Specifies to use the `gdml` module. The flag is automatically set if `$G4LIB_BUILD_GDML` is set in the environment.
$G4LIB_USE_USOLIDS Specifies to adopt the `USolids` primitives in place of the original Geant4 solids.
$G4LIB_BUILD_ZLIB If set, triggers compilation of a specific `zlib` module for the compression of output files (mainly in use currently for the HepRep graphics driver). By default, the flag is not set and the built-in system library for compression is adopted instead. Setting this flag will also implicitely set the flag below. On Windows systems, if OpenGL or OpenInventor visualization drivers are built, this module is automatically built.
$G4LIB_USE_ZLIB Specifies to use the `zlib` module, either system built-in or Geant4 specific.
$G4LIB_BUILD_G3TOG4 If set, triggers compilation of the `g3tog4` module for conversions of simple legacy geometries descriptions to Geant4. By default, the flag is not set and the module's library is not built. Setting this flag will also implicitely set the flag below.
$G4LIB_USE_G3TOG4 Specifies to use the `g3tog4` module, assuming the related library has been already installed.

Analysis specific

$G4ANALYSIS_USE Specifies to activate the appropriate environment for analysis, if an application includes code for histogramming based on `AIDA`. Additional setup variables are required (`$G4ANALYSIS_AIDA_CONFIG_CFLAGS`, `$G4ANALYSIS_AIDA_CONFIG_LIBS`) to define config options for AIDA ("`aida-config --cflags`" and "`aida-config --libs`"). See installation instructions of the specific analysis tools for details.

Directory paths to Physics Data

$G4NEUTRONHPDATA Path to external data set for Neutron Scattering processes.
$G4NEUTRONXSDATA Path to external data set for evaluated neutron cross-sections.
$G4LEDATA Path to external data set for low energy electromagnetic processes.
$G4PIIDATA Path to external data set for shell ionisation cross-sections.
$G4LEVELGAMMADATA Path to the data set for Photon Evaporation.
$G4RADIOACTIVEDATA Path to the data set for Radiative Decay processes.
$G4ENSDFSTATE1.0 Path to the data set for NuclideTable
$G4ABLADATA Path to nuclear shell effects data set for INCL/ABLA hadronic model.

$G4REALSURFACEDATA Path to the data set for measured optical surface reflectance for precise optical physics.

11.3.3 Linking External Libraries with Geant4

The Geant4 GNUmake infrastructure allows to extend the link list of libraries with external (or user defined) packages which may be required for some user's applications to generate the final executable.

Adding external libraries which do *not* use Geant4

In the `GNUmakefile` of your application, before including `binmake.gmk`, specify the extra library in `EXTRALIBS` either using the `-L...-l...` syntax or by specifying the full pathname, e.g.:

```
EXTRALIBS := -L<your-path>/lib -l<myExtraLib>
```

or

```
EXTRALIBS := <your-path>/lib/lib<myExtraLib>.a
```

You may also specify `EXTRA_LINK_DEPENDENCIES`, which is added to the dependency of the target executable, and you may also specify a rule for making it, e.g.:

```
EXTRA_LINK_DEPENDENCIES := <your-path>/lib/lib<myExtraLib>.a

<your-path>/lib/lib<myExtraLib>.a:
      cd <your-path>/lib; $(MAKE)
```

Note that you almost certainly need to augment `CPPFLAGS` for the header files of the external library, e.g.:

```
CPPFLAGS+=-I<your-path>/include
```

See the *example*.

Listing 11.1: An example of a customised GNUmakefile for an application or example using an external module not bound to Geant4.

```
# --------------------------------------------------------------------
# GNUmakefile for the application "sim" depending on module "Xplotter"
# --------------------------------------------------------------------

name := sim
G4TARGET := $(name)
G4EXLIB := true

CPPFLAGS  += -I$(HOME)/Xplotter/include
EXTRALIBS += -L$(HOME)/Xplotter/lib -lXplotter
EXTRA_LINK_DEPENDENCIES := $(HOME)/Xplotter/lib/libXplotter.a

.PHONY: all

all: lib bin

include $(G4INSTALL)/config/binmake.gmk

$(HOME)/Xplotter/lib/libXplotter.a:
        cd $(HOME)/Xplotter; $(MAKE)
```

Adding external libraries which use Geant4

In addition to the above, specify, in `EXTRALIBSSOURCEDIRS`, a list of directories containing source files in its `src/` subdirectory. Thus, your `GNUmakefile` might contain:

```
EXTRALIBS += $(G4WORKDIR)/tmp/$(G4SYSTEM)/<myApp>/lib<myApp>.a \
             -L<your-path>/lib -l<myExtraLib>
EXTRALIBSSOURCEDIRS += <your-path>/<myApp> <your-path>/<MyExtraModule>
EXTRA_LINK_DEPENDENCIES := $(G4WORKDIR)/tmp/$(G4SYSTEM)/<myApp>/lib<myApp>.a

MYSOURCES := $(wildcard <your-path>/<myApp>/src/*cc)
$(G4WORKDIR)/tmp/$(G4SYSTEM)/<myApp>/lib<myApp>.a: $(MYSOURCES)
      cd <your-path>/<myApp>; $(MAKE)
```

See the *example*.

Listing 11.2: An example of a customised GNUmakefile for an application or example using external modules bound to Geant4.

```
# --------------------------------------------------------------
# GNUmakefile for the application "phys" depending on module "reco"
# --------------------------------------------------------------

name := phys
G4TARGET := $(name)
G4EXLIB := true

EXTRALIBS += $(G4WORKDIR)/tmp/$(G4SYSTEM)/$(name)/libphys.a \
             -L$(HOME)/reco/lib -lreco
EXTRALIBSSOURCEDIRS += $(HOME)/phys $(HOME)/reco
EXTRA_LINK_DEPENDENCIES := $(G4WORKDIR)/tmp/$(G4SYSTEM)/$(name)/libphys.a

.PHONY: all
all: lib bin

include $(G4INSTALL)/config/binmake.gmk

MYSOURCES := $(wildcard $(HOME)/phys/src/*cc)
$(G4WORKDIR)/tmp/$(G4SYSTEM)/$(name)/libphys.a: $(MYSOURCES)
        cd $(HOME)/phys; $(MAKE)
```

11.4 Development and Debug Tools

Although not in the scope of this user manual, in this appendix section we provide a set of references to rather known and established development tools and environments we think are useful for code development in C++ in general. It's a rather limited list, far from being complete of course.

11.4.1 Unix/Linux

- The KDevelop environment on Linux systems.
- The GNU Data Display Debugger (DDD).
- Valgrind, a system for debugging and profiling Linux programs.
- Parasoft Insure++ run-time debugger and memory checker
- Parasoft C++ Test source code analyzer.
- MicroFocus Together Visual Modeling for Software Architecture Design tool.

11.4.2 Windows

- Microsoft Visual Studio development environment.
- Parasoft Insure++ run-time debugger and memory checker
- Parasoft C++ Test source code analyzer.
- Enterprise Architect UML Visual Modeling tool.
- MicroFocus Together Visual Modeling for Software Architecture Design tool.

11.5 Python Interface

Python is a popular scripting language with an interactive interpreter. Geant4Py, a Geant4-Python bridge, provides a bridge for Geant4 classes. This enables to directly access Geant4 classes from Python scripting. User applications can be easily configured with many Python third-party modules, such as PyROOT, on the Python software bus.

Geant4Py is supplied in the directory `environments/g4py/` of the Geant4 source package.

11.5.1 Installation

Software Requirements

Geant4Py requires the Boost-C++ external library, which helps Python binding of C++ codes.

Building Geant4Py module

Building system is completely migrated to CMake system. Before building library, **GEANT4_INSTALL** environment variable should be set to the path specified by "CMAKE_INSTALL_PREFIX" when building Geant4.

```
# export GEANT4_INSTALL="Geant4 install path" (zsh, bash)
# setenv GEANT4_INSTALL "Geant4 install path" (csh)
```

Then

```
# mkdir build
# cd build
# cmake ..
# make
# make install
```

If you want to run the testing component,

```
# cd build/tests
# make; make install
```

By default, Geant4Py is installed in "g4py"/lib(64) directory.

11.5.2 Using Geant4Py

PYTHONPATH environment variable should be set at tun time. **PYTHONPATH** environment variable indicates Python module search directories, given by a colon-separated list of directories. Practically, the variable is `(your g4py directory)/lib:(your g4py directory)/lib/examples:(your g4py directory)/lib/tests`.

Import Geant4

To use Geant4Py, you start with importing the module called "Geant4".

```
# python
Python 2.7.5 (default, Mar  9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from Geant4 import *

************************************************************
 Geant4 version Name: geant4-10-01         (5-December-2014)
                      Copyright : Geant4 Collaboration
                      Reference : NIM A 506 (2003), 250-303
                            WWW : http://cern.ch/geant4
************************************************************

Visualization Manager instantiating...
>>>
```

Access to Geant4 Globals

When importing the Geant4 module, the `G4RunManager` object will be automatically instantiated. Geant4 singleton objects are also automatically instantiated. These singleton objects can be accessed by "gXXXX" variables, like "gRunManager".

```
.                       gLossTableManager       gTerminate
gApplyUICommand         gMaterialTable          gTrackingManager
gControlExecute         gNistManager            gTransportationManager
gElementTable           gParticleIterator       gUImanager
gEmCalculator           gParticleTable          gVisManager
gEventManager           gProcessTable
gExceptionHandler       gProductionCutsTable
gG4Date                 gRunManager
gG4VERSION_NUMBER       gRunManagerKernel
gG4Version              gStackManager
gGeometryManager        gStartUISession
gGetCurrentValues       gStateManager
```

Call Geant4 Methods

Once a Python object of a Geant4 class instantiated, Geant4 methods can be directly called the same way as in C++.

```
>>> from Geant4 import *

************************************************************
 Geant4 version Name: geant4-10-01         (5-December-2014)
                      Copyright : Geant4 Collaboration
                      Reference : NIM A 506 (2003), 250-303
                            WWW : http://cern.ch/geant4
************************************************************

Visualization Manager instantiating...

>>> print gRunManager.GetVersionString()
 Geant4 version Name: geant4-10-01         (5-December-2014)
```

11.5.3 Site-modules

Geant4Py provides additional utility modules called "g4py" in the directory `site-modules`. It consists of predifined geometries, materials, physics lists, primary generator actions, and so on.

ezgeom module

The *ezgeom* module provides an alternative way of defining simple geometry. An example code for defining a simple geometry is shown here:

```
import g4py.ezgeom
from g4py.ezgeom import G4EzVolume

def ConstructGeom():
  print "* Constructing geometry..."
  # reset world material
  air= G4Material.GetMaterial("G4_AIR")
  g4py.ezgeom.SetWorldMaterial(air)

  # a target box is placed
  global target
  target= G4EzVolume("Target")
  au= G4Material.GetMaterial("G4_Au")
  target.CreateTubeVolume(au, 0., 1.*cm, 1.*mm)
  target.PlaceIt(G4ThreeVector(0.,0.,-10.*cm))
```

NISTmaterials module

The *NISTmaterials* module provides an instant use of Geant4 NIST materials. An example code for creating NIST materials:

```
from Geant4 import *
import g4py.NISTmaterials

g4py.NISTmaterials.Construct()
print Geant4.gMaterialTable
```

ParticleGun module

The *ParticleGun* module provides a primary generator action with `G4ParticleGun`. An example code is shown here:

```
import g4py.ParticleGun

# normal way for constructing user primary generator action
#pgPGA= g4py.ParticleGun.ParticleGunAction()
#gRunManager.SetUserAction(pgPGA)
#pg= pgPGA.GetParticleGun()

# 2nd way, short-cut way
pg= g4py.ParticleGun.Construct()

# set parameters of particle gun
pg.SetParticleByName("e-")
pg.SetParticleEnergy(300.*MeV)
primary_position= G4ThreeVector(0.,0., -14.9*cm)
primary_direction= G4ThreeVector(0.2, 0., 1.)
pg.SetParticlePosition(primary_position)
pg.SetParticleMomentumDirection(primary_direction)
```

11.5.4 Examples

There are some examples of Geant4Py in the directories "tests/" and "examples/".

In the "tests/" directory,

```
gtest01 : exposes a user application
gtest02 : test for using site-module packages
gtest03 : test for ezgeom package
gtest04 : test for getting command tree and command information
gtest05 : test for constructing CSG geometries in Python
gtest06 : test for constructing/visualizing boolean geoemtries
gtest07 : test for checking overlapped geometries
```

The "examples/" directory contains a set of examples of Geant4Py.

demos/water_phantom An example of "water phantom dosimetry". This demo program shows that a Geant4 application well coworks with Root on Python front end. VisManager, PrimaryGeneratorAction, UserAction-s, histogramming with Root are implemented in Python;

- dose calculation in a water phantom
- Python overloading of user actions
- on-line histogramming with Root
- visualization

education Educational examples with Graphical User Interface using TKinter

- lesson1
  The first version of the courseware of the mass attenuation coefficient.
- lesson2
  GUI interface of ExN03, which can control geometry configuration, intial particle condition, physics processes, cut value, magnetic field and visualization outputs.

emplot Examples of plotting photon cross sections and stopping powers with Root.

gdml Examples of writing/reading user's geometry to/from a GDML file

11.6 Geant4 Material Database

11.6.1 Simple Materials (Elements)

```
=========================================================
 Z   Name   density(g/cm^3)  I(eV)
=========================================================
 1   G4_H    8.3748e-05       19.2
 2  G4_He   0.000166322       41.8
 3  G4_Li         0.534         40
 4  G4_Be         1.848       63.7
 5   G4_B          2.37         76
 6   G4_C             2         81
 7   G4_N     0.0011652         82
 8   G4_O    0.00133151         95
 9   G4_F    0.00158029        115
10  G4_Ne   0.000838505        137
11  G4_Na         0.971        149
12  G4_Mg          1.74        156
13  G4_Al         2.699        166
14  G4_Si          2.33        173
15   G4_P           2.2        173
16   G4_S             2        180
17  G4_Cl    0.00299473        174
18  G4_Ar    0.00166201        188
19   G4_K         0.862        190
20  G4_Ca          1.55        191
```

```
21  G4_Sc          2.989        216
22  G4_Ti           4.54        233
23   G4_V           6.11        245
24  G4_Cr           7.18        257
25  G4_Mn           7.44        272
26  G4_Fe          7.874        286
27  G4_Co            8.9        297
28  G4_Ni          8.902        311
29  G4_Cu           8.96        322
30  G4_Zn          7.133        330
31  G4_Ga          5.904        334
32  G4_Ge          5.323        350
33  G4_As           5.73        347
34  G4_Se            4.5        348
35  G4_Br     0.0070721         343
36  G4_Kr    0.00347832         352
37  G4_Rb         1.532         363
38  G4_Sr          2.54         366
39   G4_Y         4.469         379
40  G4_Zr         6.506         393
41  G4_Nb          8.57         417
42  G4_Mo         10.22         424
43  G4_Tc          11.5         428
44  G4_Ru         12.41         441
45  G4_Rh         12.41         449
46  G4_Pd         12.02         470
47  G4_Ag          10.5         470
48  G4_Cd          8.65         469
49  G4_In          7.31         488
50  G4_Sn          7.31         488
51  G4_Sb         6.691         487
52  G4_Te          6.24         485
53   G4_I          4.93         491
54  G4_Xe    0.00548536         482
55  G4_Cs         1.873         488
56  G4_Ba           3.5         491
57  G4_La         6.154         501
58  G4_Ce         6.657         523
59  G4_Pr          6.71         535
60  G4_Nd           6.9         546
61  G4_Pm          7.22         560
62  G4_Sm          7.46         574
63  G4_Eu         5.243         580
64  G4_Gd        7.9004         591
65  G4_Tb         8.229         614
66  G4_Dy          8.55         628
67  G4_Ho         8.795         650
68  G4_Er         9.066         658
69  G4_Tm         9.321         674
70  G4_Yb          6.73         684
71  G4_Lu          9.84         694
72  G4_Hf         13.31         705
73  G4_Ta        16.654         718
74   G4_W          19.3         727
75  G4_Re         21.02         736
76  G4_Os         22.57         746
77  G4_Ir         22.42         757
78  G4_Pt         21.45         790
79  G4_Au         19.32         790
80  G4_Hg        13.546         800
81  G4_Tl         11.72         810
82  G4_Pb         11.35         823
83  G4_Bi         9.747         823
84  G4_Po          9.32         830
85  G4_At          9.32         825
86  G4_Rn    0.00900662         794
87  G4_Fr             1         827
```

```
88  G4_Ra             5        826
89  G4_Ac         10.07        841
90  G4_Th         11.72        847
91  G4_Pa         15.37        878
92   G4_U         18.95        890
93  G4_Np         20.25        902
94  G4_Pu         19.84        921
95  G4_Am         13.67        934
96  G4_Cm         13.51        939
97  G4_Bk            14        952
98  G4_Cf            10        966
```

11.6.2 NIST Compounds

```
============================================================
 Ncomp             Name      density(g/cm^3)  I(eV) ChFormula
============================================================
 6            G4_A-150_TISSUE      1.127      65.1
         1      0.101327
         6        0.7755
         7      0.035057
         8     0.0523159
         9      0.017422
        20      0.018378
 3               G4_ACETONE     0.7899      64.2
         6             3
         1             6
         8             1
 2             G4_ACETYLENE  0.0010967      58.2
         6             2
         1             2
 3               G4_ADENINE        1.6      71.4
         6             5
         1             5
         7             5
 7     G4_ADIPOSE_TISSUE_ICRP       0.95      63.2
         1         0.114
         6         0.598
         7         0.007
         8         0.278
        11         0.001
        16         0.001
        17         0.001
 4                   G4_AIR 0.00120479      85.7
         6      0.000124
         7      0.755268
         8      0.231781
        18      0.012827
 4               G4_ALANINE       1.42      71.9
         6             3
         1             7
         7             1
         8             2
 2          G4_ALUMINUM_OXIDE       3.97     145.2   Al_2O_3
        13             2
         8             3
 3                 G4_AMBER        1.1      63.2
         1       0.10593
         6      0.788974
         8      0.105096
 2               G4_AMMONIA 0.000826019      53.7
         7             1
         1             3
 3               G4_ANILINE     1.0235      66.2
```

```
        6             6
        1             7
        7             1
 2             G4_ANTHRACENE      1.283      69.5
        6            14
        1            10
 6             G4_B-100_BONE       1.45      85.9
        1     0.0654709
        6      0.536944
        7        0.0215
        8      0.032085
        9      0.167411
       20      0.176589
 3               G4_BAKELITE       1.25      72.4
        1      0.057441
        6      0.774591
        8      0.167968
 2         G4_BARIUM_FLUORIDE       4.89     375.9
       56             1
        9             2
 3          G4_BARIUM_SULFATE        4.5     285.7
       56             1
       16             1
        8             4
 2                G4_BENZENE    0.87865      63.4
        6             6
        1             6
 2         G4_BERYLLIUM_OXIDE       3.01      93.2
        4             1
        8             1
 3                    G4_BGO       7.13     534.1
       83             4
       32             3
        8            12
10             G4_BLOOD_ICRP       1.06      75.2
        1         0.102
        6          0.11
        7         0.033
        8         0.745
       11         0.001
       15         0.001
       16         0.002
       17         0.003
       19         0.002
       26         0.001
 8       G4_BONE_COMPACT_ICRU       1.85      91.9
        1         0.064
        6         0.278
        7         0.027
        8          0.41
       12         0.002
       15          0.07
       16         0.002
       20         0.147
 9      G4_BONE_CORTICAL_ICRP       1.92       110
        1         0.034
        6         0.155
        7         0.042
        8         0.435
       11         0.001
       12         0.002
       15         0.103
       16         0.003
       20         0.225
 2           G4_BORON_CARBIDE       2.52      84.7
        5             4
        6             1
```

```
2             G4_BORON_OXIDE      1.812      99.6
        5             2
        8             3
9              G4_BRAIN_ICRP       1.04      73.3
        1         0.107
        6         0.145
        7         0.022
        8         0.712
       11         0.002
       15         0.004
       16         0.002
       17         0.003
       19         0.003
2                 G4_BUTANE 0.00249343      48.3
        6             4
        1            10
3         G4_N-BUTYL_ALCOHOL     0.8098      59.9
        6             4
        1            10
        8             1
5                 G4_C-552       1.76      86.8
        1       0.02468
        6      0.501611
        8      0.004527
        9      0.465209
       14      0.003973
2       G4_CADMIUM_TELLURIDE        6.2     539.3
       48             1
       52             1
3       G4_CADMIUM_TUNGSTATE        7.9     468.3
       48             1
       74             1
        8             4
3       G4_CALCIUM_CARBONATE        2.8     136.4
       20             1
        6             1
        8             3
2        G4_CALCIUM_FLUORIDE       3.18       166
       20             1
        9             2
2           G4_CALCIUM_OXIDE        3.3     176.1
       20             1
        8             1
3         G4_CALCIUM_SULFATE       2.96     152.3
       20             1
       16             1
        8             4
3       G4_CALCIUM_TUNGSTATE      6.062       395
       20             1
       74             1
        8             4
2          G4_CARBON_DIOXIDE 0.00184212        85   CO_2
        6             1
        8             2
2    G4_CARBON_TETRACHLORIDE      1.594     166.3
        6             1
       17             4
3    G4_CELLULOSE_CELLOPHANE       1.42      77.6
        6             6
        1            10
        8             5
3      G4_CELLULOSE_BUTYRATE        1.2      74.6
        1      0.067125
        6      0.545403
        8      0.387472
4       G4_CELLULOSE_NITRATE       1.49        87
        1      0.029216
```

```
         6      0.271296
         7      0.121276
         8      0.578212
 5           G4_CERIC_SULFATE       1.03      76.7
         1      0.107596
         7        0.0008
         8      0.874976
        16      0.014627
        58      0.002001
 2         G4_CESIUM_FLUORIDE      4.115     440.7
        55             1
         9             1
 2            G4_CESIUM_IODIDE       4.51     553.1
        55             1
        53             1
 3            G4_CHLOROBENZENE     1.1058      89.1
         6             6
         1             5
        17             1
 3               G4_CHLOROFORM     1.4832       156
         6             1
         1             1
        17             3
10                 G4_CONCRETE        2.3     135.2
         1          0.01
         6         0.001
         8      0.529107
        11         0.016
        12         0.002
        13      0.033872
        14      0.337021
        19         0.013
        20         0.044
        26         0.014
 2              G4_CYCLOHEXANE      0.779      56.4
         6             6
         1            12
 3     G4_1,2-DICHLOROBENZENE     1.3048     106.5
         6             6
         1             4
        17             2
 4   G4_DICHLORODIETHYL_ETHER     1.2199     103.3
         6             4
         1             8
         8             1
        17             2
 3      G4_1,2-DICHLOROETHANE     1.2351     111.9
         6             2
         1             4
        17             2
 3           G4_DIETHYL_ETHER    0.71378        60
         6             4
         1            10
         8             1
 4  G4_N,N-DIMETHYL_FORMAMIDE     0.9487      66.6
         6             3
         1             7
         7             1
         8             1
 4      G4_DIMETHYL_SULFOXIDE     1.1014      98.6
         6             2
         1             6
         8             1
        16             1
 2                 G4_ETHANE 0.00125324      45.4
         6             2
         1             6
```

```
3           G4_ETHYL_ALCOHOL     0.7893      62.9
        6              2
        1              6
        8              1
3         G4_ETHYL_CELLULOSE       1.13      69.3
        1       0.090027
        6       0.585182
        8       0.324791
2              G4_ETHYLENE 0.00117497      50.7
        6              2
        1              4
8           G4_EYE_LENS_ICRP       1.07      73.3
        1          0.096
        6          0.195
        7          0.057
        8          0.646
       11          0.001
       15          0.001
       16          0.003
       17          0.001
2            G4_FERRIC_OXIDE        5.2     227.3
       26              2
        8              3
2             G4_FERROBORIDE       7.15       261
       26              1
        5              1
2           G4_FERROUS_OXIDE        5.7     248.6
       26              1
        8              1
7         G4_FERROUS_SULFATE      1.024      76.4
        1       0.108259
        7        2.7e-05
        8       0.878636
       11        2.2e-05
       16       0.012968
       17        3.4e-05
       26        5.4e-05
3               G4_FREON-12       1.12       143
        6       0.099335
        9       0.314247
       17       0.586418
3               G4_FREON-12B2        1.8     284.9
        6       0.057245
        9       0.181096
       35       0.761659
3               G4_FREON-13       0.95     126.6
        6       0.114983
        9       0.545621
       17       0.339396
3               G4_FREON-13B1        1.5     210.5
        6              1
        9              3
       35              1
3               G4_FREON-13I1        1.8     293.5
        6       0.061309
        9       0.290924
       53       0.647767
3   G4_GADOLINIUM_OXYSULFIDE       7.44     493.3
       64              2
        8              2
       16              1
2         G4_GALLIUM_ARSENIDE       5.31     384.9
       31              1
       33              1
5       G4_GEL_PHOTO_EMULSION     1.2914      74.8
        1        0.08118
        6        0.41606
```

```
        7       0.11124
        8       0.38064
       16       0.01088
6            G4_Pyrex_Glass       2.23       134
        5     0.0400639
        8      0.539561
       11     0.0281909
       13      0.011644
       14      0.377219
       19    0.00332099
5              G4_GLASS_LEAD       6.22     526.4
        8      0.156453
       14      0.080866
       22      0.008092
       33      0.002651
       82      0.751938
4             G4_GLASS_PLATE        2.4     145.4
        8        0.4598
       11     0.0964411
       14      0.336553
       20      0.107205
4               G4_GLUTAMINE       1.46      73.3
        6             5
        1            10
        7             2
        8             3
3                G4_GLYCEROL     1.2613      72.6
        6             3
        1             8
        8             3
4                 G4_GUANINE        2.2        75
        6             5
        1             5
        7             5
        8             1
4                   G4_GYPSUM       2.32     129.7
       20             1
       16             1
        8             6
        1             4
2               G4_N-HEPTANE    0.68376      54.4
        6             7
        1            16
2                G4_N-HEXANE     0.6603        54
        6             6
        1            14
4                  G4_KAPTON       1.42      79.6
        6            22
        1            10
        7             2
        8             5
3    G4_LANTHANUM_OXYBROMIDE       6.28     439.7
       57             1
       35             1
        8             1
3    G4_LANTHANUM_OXYSULFIDE       5.86     421.2
       57             2
        8             2
       16             1
2              G4_LEAD_OXIDE       9.53     766.7
        8      0.071682
       82      0.928318
3           G4_LITHIUM_AMIDE      1.178      55.5
        3             1
        7             1
        1             2
3         G4_LITHIUM_CARBONATE       2.11      87.9
```

```
        3             2
        6             1
        8             3
2        G4_LITHIUM_FLUORIDE      2.635        94
        3             1
        9             1
2         G4_LITHIUM_HYDRIDE       0.82      36.5
        3             1
        1             1
2          G4_LITHIUM_IODIDE      3.494     485.1
        3             1
       53             1
2           G4_LITHIUM_OXIDE      2.013      73.6
        3             2
        8             1
3     G4_LITHIUM_TETRABORATE       2.44      94.6
        3             2
        5             4
        8             7
9               G4_LUNG_ICRP       1.04      75.3
        1         0.105
        6         0.083
        7         0.023
        8         0.779
       11         0.002
       15         0.001
       16         0.002
       17         0.003
       19         0.002
5                G4_M3_WAX       1.05      67.9
        1      0.114318
        6      0.655824
        8     0.0921831
       12      0.134792
       20      0.002883
3     G4_MAGNESIUM_CARBONATE      2.958       118
       12             1
        6             1
        8             3
2      G4_MAGNESIUM_FLUORIDE          3     134.3
       12             1
        9             2
2         G4_MAGNESIUM_OXIDE       3.58     143.8
       12             1
        8             1
3   G4_MAGNESIUM_TETRABORATE       2.53     108.3
       12             1
        5             4
        8             7
2         G4_MERCURIC_IODIDE       6.36     684.5
       80             1
       53             2
2                 G4_METHANE 0.000667151      41.7
        6             1
        1             4
3                G4_METHANOL     0.7914      67.6
        6             1
        1             4
        8             1
5               G4_MIX_D_WAX       0.99      60.9
        1       0.13404
        6       0.77796
        8       0.03502
       12      0.038594
       22      0.014386
6             G4_MS20_TISSUE          1      75.1
        1      0.081192
```

```
        6      0.583442
        7      0.017798
        8      0.186381
       12      0.130287
       17        0.0009
9    G4_MUSCLE_SKELETAL_ICRP       1.05      75.3
        1         0.102
        6         0.143
        7         0.034
        8          0.71
       11         0.001
       15         0.002
       16         0.003
       17         0.001
       19         0.004
8    G4_MUSCLE_STRIATED_ICRU       1.04      74.7
        1      0.102102
        6      0.123123
        7      0.035035
        8       0.72973
       11      0.001001
       15      0.002002
       16      0.004004
       19      0.003003
4     G4_MUSCLE_WITH_SUCROSE       1.11      74.3
        1     0.0982341
        6      0.156214
        7      0.035451
        8      0.710101
4  G4_MUSCLE_WITHOUT_SUCROSE       1.07      74.2
        1      0.101969
        6      0.120058
        7      0.035451
        8      0.742522
2            G4_NAPHTHALENE      1.145      68.4
        6            10
        1             8
4           G4_NITROBENZENE    1.19867      75.8
        6             6
        1             5
        7             1
        8             2
2          G4_NITROUS_OXIDE 0.00183094      84.9
        7             2
        8             1
4             G4_NYLON-8062       1.08      64.3
        1      0.103509
        6      0.648416
        7     0.0995361
        8      0.148539
4              G4_NYLON-6-6       1.14      63.9
        6             6
        1            11
        7             1
        8             1
4             G4_NYLON-6-10       1.14      63.2
        1      0.107062
        6      0.680449
        7      0.099189
        8        0.1133
4         G4_NYLON-11_RILSAN      1.425      61.6
        1      0.115476
        6      0.720818
        7     0.0764169
        8     0.0872889
2                G4_OCTANE     0.7026      54.7
        6             8
```

```
        1            18
2               G4_PARAFFIN       0.93      55.9
        6            25
        1            52
2              G4_N-PENTANE     0.6262      53.6
        6             5
        1            12
8          G4_PHOTO_EMULSION      3.815       331
        1        0.0141
        6      0.072261
        7       0.01932
        8      0.066101
       16       0.00189
       35      0.349103
       47      0.474105
       53       0.00312
2 G4_PLASTIC_SC_VINYLTOLUENE      1.032      64.7
        6             9
        1            10
2       G4_PLUTONIUM_DIOXIDE      11.46     746.5
       94             1
        8             2
3       G4_POLYACRYLONITRILE       1.17      69.6
        6             3
        1             3
        7             1
3           G4_POLYCARBONATE        1.2      73.1
        6            16
        1            14
        8             3
3       G4_POLYCHLOROSTYRENE        1.3      81.7
        6             8
        1             7
       17             1
2            G4_POLYETHYLENE       0.94      57.4   (C_2H_4)_N-Polyethylene
        6             1
        1             2
3                  G4_MYLAR        1.4      78.7
        6            10
        1             8
        8             4
3              G4_PLEXIGLASS       1.19        74
        6             5
        1             8
        8             2
3        G4_POLYOXYMETHYLENE      1.425      77.4
        6             1
        1             2
        8             1
2           G4_POLYPROPYLENE        0.9      56.5   (C_2H_4)_N-Polypropylene
        6             2
        1             4
2             G4_POLYSTYRENE       1.06      68.7
        6             8
        1             8
2                 G4_TEFLON        2.2      99.1
        6             2
        9             4
3 G4_POLYTRIFLUOROCHLOROETHYLENE        2.1     120.7
        6             2
        9             3
       17             1
3       G4_POLYVINYL_ACETATE       1.19      73.7
        6             4
        1             6
        8             2
3       G4_POLYVINYL_ALCOHOL        1.3      69.7
```

```
        6             2
        1             4
        8             1
3       G4_POLYVINYL_BUTYRAL       1.12      67.2
        6             8
        1            14
        8             2
3      G4_POLYVINYL_CHLORIDE        1.3     108.2
        6             2
        1             3
       17             1
3 G4_POLYVINYLIDENE_CHLORIDE        1.7     134.3
        6             2
        1             2
       17             2
3 G4_POLYVINYLIDENE_FLUORIDE       1.76      88.8
        6             2
        1             2
        9             2
4   G4_POLYVINYL_PYRROLIDONE       1.25      67.7
        6             6
        1             9
        7             1
        8             1
2        G4_POTASSIUM_IODIDE       3.13     431.9
       19             1
       53             1
2         G4_POTASSIUM_OXIDE       2.32     189.9
       19             2
        8             1
2                 G4_PROPANE 0.00187939      47.1
        6             3
        1             8
2               G4_lPROPANE       0.43        52
        6             3
        1             8
3        G4_N-PROPYL_ALCOHOL     0.8035      61.1
        6             3
        1             8
        8             1
3               G4_PYRIDINE     0.9819      66.2
        6             5
        1             5
        7             1
2            G4_RUBBER_BUTYL       0.92      56.5
        1      0.143711
        6      0.856289
2          G4_RUBBER_NATURAL       0.92      59.8
        1      0.118371
        6      0.881629
3         G4_RUBBER_NEOPRENE       1.23        93
        1       0.05692
        6      0.542646
       17      0.400434
2         G4_SILICON_DIOXIDE       2.32     139.2   SiO_2
       14             1
        8             2
2          G4_SILVER_BROMIDE      6.473     486.6
       47             1
       35             1
2         G4_SILVER_CHLORIDE       5.56     398.4
       47             1
       17             1
3          G4_SILVER_HALIDES       6.47     487.1
       35      0.422895
       47      0.573748
       53      0.003357
```

```
2           G4_SILVER_IODIDE       6.01     543.5
       47             1
       53             1
9              G4_SKIN_ICRP       1.09      72.7
        1            0.1
        6          0.204
        7          0.042
        8          0.645
       11          0.002
       15          0.001
       16          0.002
       17          0.003
       19          0.001
3         G4_SODIUM_CARBONATE      2.532       125
       11             2
        6             1
        8             3
2           G4_SODIUM_IODIDE      3.667       452
       11             1
       53             1
2          G4_SODIUM_MONOXIDE       2.27     148.8
       11             2
        8             1
3           G4_SODIUM_NITRATE      2.261     114.6
       11             1
        7             1
        8             3
2                G4_STILBENE     0.9707      67.7
        6            14
        1            12
3                G4_SUCROSE     1.5805      77.5
        6            12
        1            22
        8            11
2               G4_TERPHENYL       1.24      71.7
        6            18
        1            14
9             G4_TESTIS_ICRP       1.04        75
        1         0.106
        6         0.099
        7          0.02
        8         0.766
       11         0.002
       15         0.001
       16         0.002
       17         0.002
       19         0.002
2     G4_TETRACHLOROETHYLENE      1.625     159.2
        6             2
       17             4
2        G4_THALLIUM_CHLORIDE      7.004     690.3
       81             1
       17             1
9          G4_TISSUE_SOFT_ICRP       1.03      72.3
        1         0.105
        6         0.256
        7         0.027
        8         0.602
       11         0.001
       15         0.002
       16         0.003
       17         0.002
       19         0.002
4       G4_TISSUE_SOFT_ICRU-4          1      74.9
        1         0.101
        6         0.111
        7         0.026
```

```
        8         0.762
4          G4_TISSUE-METHANE 0.00106409      61.2
        1      0.101869
        6      0.456179
        7      0.035172
        8       0.40678
4          G4_TISSUE-PROPANE 0.00182628      59.5
        1      0.102672
        6       0.56894
        7      0.035022
        8      0.293366
2        G4_TITANIUM_DIOXIDE       4.26     179.5
       22             1
        8             2
2                G4_TOLUENE     0.8669      62.5
        6             7
        1             8
3       G4_TRICHLOROETHYLENE       1.46     148.1
        6             2
        1             1
       17             3
4      G4_TRIETHYL_PHOSPHATE       1.07      81.2
        6             6
        1            15
        8             4
       15             1
2   G4_TUNGSTEN_HEXAFLUORIDE        2.4     354.4
       74             1
        9             6
2       G4_URANIUM_DICARBIDE      11.28       752
       92             1
        6             2
2     G4_URANIUM_MONOCARBIDE      13.63       862
       92             1
        6             1
2           G4_URANIUM_OXIDE      10.96     720.6
       92             1
        8             2
4                   G4_UREA      1.323      72.8
        6             1
        1             4
        7             2
        8             1
4                 G4_VALINE       1.23      67.7
        6             5
        1            11
        7             1
        8             2
3                  G4_VITON        1.8      98.6
        1      0.009417
        6      0.280555
        9      0.710028
2                  G4_WATER          1        78   H_2O
        1             2
        8             1
2             G4_WATER_VAPOR 0.000756182      71.6   H_2O-Gas
        1             2
        8             1
2                 G4_XYLENE       0.87      61.8
        6             8
        1            10
1               G4_GRAPHITE       2.21        78   Graphite
```

11.6.3 HEP and Nuclear Materials

```
=============================================================
 Ncomp             Name      density(g/cm^3)  I(eV) ChFormula
=============================================================
 1                    G4_lH2     0.0708      21.8
 1                    G4_lN2      0.807        82
 1                    G4_lO2      1.141        95
 1                    G4_lAr      1.396       188
 1                    G4_lBr     3.1028       343
 1                    G4_lKr      2.418       352
 1                    G4_lXe      2.953       482
 3                  G4_PbWO4       8.28         0
        8             4
       82             1
       74             1
 1                G4_Galactic      1e-25      21.8
 1          G4_GRAPHITE_POROUS        1.7        78   Graphite
 3                   G4_LUCITE       1.19        74
        1      0.080538
        6      0.599848
        8      0.319614
 3                    G4_BRASS       8.52         0
       29            62
       30            35
       82             3
 3                   G4_BRONZE       8.82         0
       29            89
       30             9
       82             2
 3          G4_STAINLESS-STEEL          8         0
       26            74
       24            18
       28             8
 3                    G4_CR39       1.32         0
        1            18
        6            12
        8             7
 3               G4_OCTADECANOL      0.812         0
        1            38
        6            18
        8             1
```

11.6.4 Space (ISS) Materials

```
=============================================================
 Ncomp             Name      density(g/cm^3)  I(eV) ChFormula
=============================================================
 4                  G4_KEVLAR       1.44         0
        6            14
        1            10
        8             2
        7             2
 3                  G4_DACRON        1.4         0
        6            10
        1             8
        8             4
 3                 G4_NEOPRENE       1.23         0
        6             4
        1             5
       17             1
```

11.6.5 Bio-Chemical Materials

```
=============================================================
 Ncomp             Name      density(g/cm^3)  I(eV) ChFormula
=============================================================
 4               G4_CYTOSINE       1.55        72
         1             5
         6             4
         7             3
         8             1
 4                G4_THYMINE       1.23        72
         1             6
         6             5
         7             2
         8             2
 4                 G4_URACIL       1.32        72
         1             4
         6             4
         7             2
         8             2
 3             G4_DNA_ADENINE          1        72
         1             4
         6             5
         7             5
 4             G4_DNA_GUANINE          1        72
         1             4
         6             5
         7             5
         8             1
 4            G4_DNA_CYTOSINE          1        72
         1             4
         6             4
         7             3
         8             1
 4             G4_DNA_THYMINE          1        72
         1             5
         6             5
         7             2
         8             2
 4              G4_DNA_URACIL          1        72
         1             3
         6             4
         7             2
         8             2
 4           G4_DNA_ADENOSINE          1        72
         1            10
         6            10
         7             5
         8             4
 4           G4_DNA_GUANOSINE          1        72
         1            10
         6            10
         7             5
         8             5
 4            G4_DNA_CYTIDINE          1        72
         1            10
         6             9
         7             3
         8             5
 4             G4_DNA_URIDINE          1        72
         1             9
         6             9
         7             2
         8             6
 4       G4_DNA_METHYLURIDINE          1        72
         1            11
         6            10
```

```
        7             2
        8             6
2       G4_DNA_MONOPHOSPHATE          1       72
       15             1
        8             3
5                  G4_DNA_A          1       72
        1            10
        6            10
        7             5
        8             7
       15             1
5                  G4_DNA_G          1       72
        1            10
        6            10
        7             5
        8             8
       15             1
5                  G4_DNA_C          1       72
        1            10
        6             9
        7             3
        8             8
       15             1
5                  G4_DNA_U          1       72
        1             9
        6             9
        7             2
        8             9
       15             1
5                 G4_DNA_MU          1       72
        1            11
        6            10
        7             2
        8             9
       15             1
```

BIBLIOGRAPHY

[Booch1994] Grady Booch *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Co. Inc 1994 ISBN: 0-8053-5340-2

[Ellis1990] Margaret Ellis and Bjarne Stroustrup *Annotated C++ Reference Manual (ARM)*. Addison-Wesley Publishing Co. 1990

[Hecht1974] 5. Hecht and A. Zajac *Optics*. Addison-Wesley Publishing Co. 1974 pp. 71-80 and pp. 244-246

[Josuttis1999] Nicolai M. Josuttis *The C++ Standard Library - A Tutorial and Reference.* Addison-Wesley Publishing Co. 1999 ISBN: 0-201-37926-0

[Meyers2001] Scott Meyers *Effective STL.* Addison-Wesley Publishing Co. 2001 ISBN: 0-201-74962-9

[Musser1996] David R. Musser and Atul Saini *STL Tutorial and Reference Guide / C++ Programming with the Standard Template Library*. Addison-Wesley Publishing Co. 1996 ISBN: 0-201-63398-1

[Plauger1995] P.J. Plauger *The Draft Standard C++ Library*. Prentice Hall, Englewood Cliffs 1995

[Chauvie2007] 19. Chauvie, Z. Francis, S. Guatelli, S. Incerti, B. Mascialino, P. Moretto, P. Nieminen and M. G. Pia *Geant4 physics processes for microdosimetry simulation: design foundation and implementation of the first set of models.* To be published in IEEE Trans. Nucl. Sci., Dec. 2007