Geant4 User's Documents
Version: Geant4 5.2 June 2003

Geant4 User's Guide

- For Application Developers -

About the authors

Overview Contents Previous Next

Geant4 User's Guide
For Application Developers

1. 简介

1.1 本手册内容

《The User's Guide for Application Developers》是用户在学习和开发基于 Geant4 的探测器模拟程序时，应该阅读的第一手册。手册主要讲述：

- 向初学者介绍面向对象的 Geant4 探测器模拟工具包，

- 介绍了各种可用工具和他们的使用方法，并且
- 提供了一些对开发、运行模拟程序非常有用的信息

手册只是对工具包的一个概述，而不是详尽的描述。除一些特殊的工具外，没有讨论相关的物理过程。Geant4 中所涉及的相关物理过程在《Physics Reference Manual》中有详细讨论。Geant4 类的功能和设计细节可以在 《User's Guide for Toolkit Developers》中找到，它的全部代码清单在《Software Reference Manual》中给出。

Geant4 是一个用 C＋＋语言编写的、全新的探测器模拟工具包。读者需要有面向对象的 C＋＋语言的基本知识。不需要有早期用 FORTRAN 编写的 Geant 版本的知识。虽然 Geant4 是个相当复杂的软件系统，但对于开发探测器模拟程序来说，只需要了解相对很少的一部分。

1.2 如何使用本手册

第 2 章，”运行 Geant4－一个简单的 example”。 Geant4 非常简单的介绍。讲述了编写和运行一个简单的 Geant4 应用程序的方法。Geant4 的新用户应该首先阅读这一章。强烈推荐在阅读本章的时候在你的计算机上安装并运行一个 Geant4 工具包。当我们在讨论 Geant4 提供的 example 时，我们可以运行这些例子，这将是非常有益的。在安装 Geant4 的时候，请查阅《Installation Guide for Setting up Geant4 in Your Computing Environment》。

第 3 章， "工具包基本组成" 讨论 Geant4 的基本问题，例如类属于哪个功能模块、和物理单位系统。然后讨论了 runs 和事件，他们都是一个模拟过程的基本组成单位。

第 4 章， "探测器定义和响应" 描述了如何创建一个特定材料和形状的探测器，并且将这个探测器放置与电磁场中。同时，还描述了如何使探测器对粒子敏感，如何保存相关的信息。

第 5 章， "粒子跟踪和物理过程" 讨论了粒子如何在介质中输运。Geant4 的粒子跟踪方法与物理过程一起由工具包提供。本章还讨论了粒子在 Geant4 中的定义和实现，最后列出了一个粒子属性清单。

第 6 章， "用户行为(Actions)" 是 Geant4 提供的一些 hooks，通过这些 hooks，用户可以执行一些特殊的、定制的任务。

第 7 章， "应用程序的通讯和控制" 概述了用户可用的、用于控制程序执行的命令。在第 2 章以后，第 6、7 两章对新用户来说是最重要的。

第 8 章， "可视化" 探测器几何、粒子径迹和事件的显示。

第 9 章，"例子" 提供了一些入门和比较高级的模拟程序代码，可以不作任何修改进行编译。这些例子非常适用与学习使用 Geant4 工具包，并可以基于这些例子，开发更加复杂的应用程序。

About the authors

2. 运行 Geant4 – 一个简单的例子

2.1 如何编写 main() 函数

2.1.1 一个 main()函数的例子

main()函数包含的内容将因特定的模拟应用不同而不同，因此必须由用户提供。Geant4 工具包不提供 main() 函数，但提供了一个简单的例子指导用户编写自己的 main()函数。程序清单 2.1.1 是一个模拟程序所必须的 main() 函数的最简单例子。

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

#include "ExN01DetectorConstruction.hh"
#include "ExN01PhysicsList.hh"
```

```
#include "ExN01PrimaryGeneratorAction.hh"

int main()
{
  // 构造缺省 run manager
  G4RunManager* runManager = new G4RunManager;

  // 设置必须的初始化类
  runManager->SetUserInitialization(new ExN01DetectorConstruction);
  runManager->SetUserInitialization(new ExN01PhysicsList);

  // 设置必须的用户行为类
  runManager->SetUserAction(new ExN01PrimaryGeneratorAction);

  // 初始化 G4 内核
  runManager->initialize();

  // 获取指向 UI manager 的指针并设置 verbosities
  G4UImanager* UI = G4UImanager::GetUIpointer();
  UI->ApplyCommand("/run/verbose 1");
  UI->ApplyCommand("/event/verbose 1");
  UI->ApplyCommand("/tracking/verbose 1");

  // 启动一个 run
  int numberOfEvent = 3;
  runManager->BeamOn(numberOfEvent);

  // 任务结束
  delete runManager;
  return 0;
}
```

代码清单 2.1.1

`main()` 函数通过 Geant4 提供的两个类 *G4RunManager* 和 *G4UImanager*，和另外三个类，*ExN01DetectorConstruction*, *ExN01PhysicsList* 和 *ExN01PrimaryGeneratorAction*, 它们是从 Geant4 提供的类派生的。下一节将对这些类进行讨论。

2.1.2 *G4RunManager* 类

`main()` 函数必须做的第一件事是创建一个 *G4RunManager* 类的实例。这是 Geant4 内核中唯一的一个运行管理类，它必须在 `main()`函数中显式的创建。它控制程序的流程并在一个 run

中管理事件循环。当 *G4RunManager* 被创建时，其它的管理类同时被创建。它们在 *G4RunManager* 被删除的时候自动删除。运行管理类同时管理初始化进程，包括在用户初始化类中的方法。我们必传递所有必须的消息给运行控制进程，以建立并运行一个模拟过程，这些消息包括：

1. 探测器将如何构建，
2. 将被模拟的所有粒子和所有物理过程，
3. 在一个事件中的初级粒子将如何产生和
4. 其他模拟必须的消息。

在这个例子中，这些代码是

```
runManager->SetUserInitialization(new ExN01DetectorConstruction);
runManager->SetUserInitialization(new ExN01PhysicsList);
```

分别创建指定探测器几何和物理过程的对象，并传递这些指针给运行管理进程。*ExN01DetectorConstruction* 是一个用户初始化类的例子，它是从 *G4VUserDetectorConstruction* 类派生的。这个类描述整个探测器的结构，包括：

- 探测器的几何形状，
- 在探测器中使用的材料，
- 探测器的敏感区域定义和
- 这些敏感区域的读出方式。

同样 *ExN01PhysicsList* 是由 *G4VUserPhysicsList* 派生，要求用户定义

- 在模拟中将被使用的粒子，
- 这些粒子的截断范围和
- 所有将被模拟的物理过程。

在 `main()` 函数中的下一个指令是

```
runManager->SetUserAction(new ExN01PrimaryGeneratorAction);
```

建立一个粒子发生器的实例并传递指向他的指针给运行管理进程。*ExN01PrimaryGeneratorAction* 是一个用户行为(action)类的例子，它从 *G4VUserPrimaryGeneratorAction* 派生。在该类中，用户必须描述初级事件的初始状态。这个类有一个公有虚方法 `generatePrimaries()` ，它将在每个事件的开始时刻被调用。详细的描述请阅读 2.6 节。注意，Geant4 不提供任何缺省行为用于产生一个初级事件。

下一个指令是

```
runManager->initialize();
```

建立探测器结构，创建物理过程，计算截面并且建立 run。最后

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

运行管理进程开始 3 个顺序执行的事件的第一个 run。`beamOn()` 方法可以在 `main()`函数中被调用任意多次。一旦开始一次 run，探测器结构和物理过程都不可以更改。然而，如 3.4.4 节所述，它们可以在两次 run 之间进行更改。更多关于 *G4RunManager* 的信息可以在 3.4 节找到。

如上所述，其它管理类在运行管理类创建的时候创建。其中一个是用户接口管理类，*G4UImanager* 。在 `main()` 函数中，必须获取指向用户接口管理进程的指针。

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

在本例中，`applyCommand()`被调用了 3 次，发命令给应用程序，让应用程序打印出 run、事件和粒子跟踪的信息。用户能够使用大量可用的用户接口命令对模拟过程进行控制。这些命令可以在 7.1 节找到。

2.1.3 用户初始化（Initialization）和行为（Action） 类

必需的用户类

有三个类必需由用户定义。两个是用户初始化类，另一个是用户行为(action)类。它们必须由 Geant4 提供的抽象基类 *G4VUserDetectorConstruction*, *G4VuserPhysicsList* and *G4VuserPrimaryGeneratorAction* 派生。Geant4 不提供这些类的缺省方法。*G4RunManager* 在调用 `initialize()` 和 `BeamOn()`方法的时候，检查这些必须的类是否存在。

就像在上节中提到的，*G4VUserDetectorConstruction* 要求用户定义探测器，*G4VUserPhysicsList* 要求用户定义物理过程。探测器定义将在 2.2 和 2.3 节讨论。物理定义将在 2.4 和 2.5 节讨论。用户行为(action)类 *G4VuserPrimaryGeneratorAction* 要求定义初级事件状态。初级事件产生将在 2.6 节讨论。

可选用户行为(Action)类

Geant4 提供了 5 个用户 hook 类：

- *G4UserRunAction*
- *G4UserEventAction*
- *G4UserStackingAction*
- *G4UserTrackingAction*
- *G4UserSteppingAction*

在这些类中，有几个虚方法允许用户在模拟程序的各个层次添加其它代码。有关用户初始化和行为(action)类的详细讨论在 第 6 章。

2.1.4. *G4UImanager* 类和发送 UI 用户接口命令

Geant4 提供了一大类叫 **intercoms** 的类。*G4UImanager* 是这个大类（类属）中的管理类。你可以通过调用这些类对象的 set 方法来使用这些类的功能。在代码清单 2.1.1 中，各种 Geant4 管理类的初始化信息被设置。**intercoms** 的机制和使用方法的详细描述，将在下一章进行讨论，并会列出可用的用户接口命令。用户接口命令的发送可以在整个应用程序中进行。

```
#include "G4RunManager.hh"
#include "G4UImanager.hh"
#include "G4UIterminal.hh"

#include "N02VisManager.hh"
#include "N02DetectorConstruction.hh"
#include "N02PhysicsList.hh"
#include "N02PrimaryGeneratorAction.hh"
#include "N02RunAction.hh"
#include "N02EventAction.hh"
#include "N02SteppingAction.hh"

#include "g4templates.hh"

int main(int argc,char** argv)
{
  // 构造缺省的 run manager
  G4RunManager * runManager = new G4RunManager;

  // 设置必须的初始化类
  N02DetectorConstruction* detector = new N02DetectorConstruction;
  runManager->SetUserInitialization(detector);
  runManager->SetUserInitialization(new N02PhysicsList);

  // 可视化 manager
  G4VisManager* visManager = new N02VisManager;
  visManager->initialize();

  // 设置用户行为类
  runManager->SetUserAction(new N02PrimaryGeneratorAction(detector));
  runManager->SetUserAction(new N02RunAction);
  runManager->SetUserAction(new N02EventAction);
  runManager->SetUserAction(new N02SteppingAction);

  // 获取指向 UI manager 的指针
  G4UImanager* UI = G4UImanager::GetUIpointer();
```

```
  if(argc==1)
  // 定义交互式(G)UI 终端
  {
    G4UIsession * session = new G4UIterminal;
    UI->ApplyCommand("/control/execute prerun.g4mac");
    session->sessionStart();
    delete session;
  }
  else
  // 批处理模式
  {
    G4String command = "/control/execute ";
    G4String fileName = argv[1];
    UI->ApplyCommand(command+fileName);
  }

  // 任务结束
  delete visManager;
  delete runManager;

  return 0;
}
```

代码清单 2.1.2
一个使用交互式终端和可视化的 `main()` 函数例子，与代码清单 2.1.1 中不同的部分用蓝色显示

2.1.5 *G4cout* 和 *G4cerr*

虽然在上例中没有包含输出流，但通常它是需要的。*G4cout* 和 *G4cerr* 是由 Geant4 定义的 **iostream** 对象。这些对象的使用，除了输出流被 *G4UImanager* 处理外，其余是与普通的 *cout* 和 *cerr* 完全相同的。因此，输出字符串可能显示在其它的窗口或者存到一个文件中。有关这些输出流的处理在 7.2.4 节中描述。用户应该使用这些对象来代替普通的 *cout* 和 *cerr*。

About the authors

Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Getting Started with Geant4

2.2 如何定义一个探测器的几何形状

2.2.1 基本概念

在 Geant4 中，探测器几何体是由大量几何体组成的。其中，最大的的几何体叫世界(World)。它包含其他所有的几何体。其它的几何体，都被创建并放置在世界(World)中。

每个几何体都通过描述它的形状和物理特性来创建，然后放置到另一个几何体中。

当一个几何体被放置到另一个中时，我们叫前者为子几何体，后者为母几何体。用相对于母体坐标系的坐标指定子体放置的位置。

为了描述几何体的形状，我们使用实体的概念。一个实体是指一个有固定形状和尺寸的几何物体。一个边长 10cm 的立方体和一个半径 30cm、高 75cm 的圆柱体都是具体的实体。

为了描述几何体的全部属性，我们使用逻辑几何体的概念。逻辑几何体包括实体的所有几何特性，和另外的物理特性：几何体的材料；是否包含探测器的敏感单元；磁场；等等。

我们必须描述如何放置几何体。为此，需要建立一个物理几何体，用于放置逻辑体的拷贝。

2.2.2 创建一个简单的几何体

创建一个几何体需要做什么？

- 创建一个实体
- 用这个实体，加上其他属性，创建一个逻辑体。

2.2.3 选择一个实体(solid)

为了创建一个简单的盒子，只需要定义它的名字和它各边的长度。可以在 Novice Example N01 中找到这样的例子。

在探测器定义的代码 `ExN01DetectorConstruction.cc` 中，你可以找到如下定义盒子的代码：

```
G4double expHall_x = 3.0*m;
G4double expHall_y = 1.0*m;
```

```
G4double expHall_z = 1.0*m;

G4Box* experimentalHall_box
  = new G4Box("expHall_box",expHall_x,expHall_y,expHall_z);
```

代码清单 2.2.1
创建一个盒子

这段代码创建了一个叫"expHall_box"的盒子，这个盒子在沿 X 轴方向从-3.0m 到 3m，沿 Y 轴方向从-1m 到 1m，沿 Z 轴方向从-1m 到 1m。

它还使用 *G4Tubs* 创建了一个圆柱体。

```
G4double innerRadiusOfTheTube = 0.*cm;
G4double outerRadiusOfTheTube = 60.*cm;
G4double hightOfTheTube = 50.*cm;
G4double startAngleOfTheTube = 0.*deg;
G4double spanningAngleOfTheTube = 360.*deg;

G4Tubs* tracker_tube
 = new G4Tubs("tracker_tube",
            innerRadiusOfTheTube,
            outerRadiusOfTheTube,
            hightOfTheTube,
            startAngleOfTheTube,
            spanningAngleOfTheTube);
```

代码清单 2.2.2
创建一个圆柱体

这段代码创建了一个叫"tracker_tube"的圆柱体，半径为 60cm，高为 50cm。

2.2.4 创建一个逻辑几何体

为了创建一个逻辑体，你必须首先创建实体和材料。所以，可以使用用上节已经建立的盒子，创建一个用氩气（参看有关材料的章节）填充的简单逻辑体：

```
G4LogicalVolume* experimentalHall_log
  = new G4LogicalVolume(experimentalHall_box,Ar,"expHall_log");
```

这个逻辑体叫"expHall_log"。

同样，我们也可以用圆柱体创建一个填充铝的逻辑体。

```
G4LogicalVolume* tracker_log
  = new G4LogicalVolume(tracker_tube,Al,"tracker_log");
```

名为"tracker_log"

2.2.5 放置一个几何体

如何放置一个几何体？如果已经创建了一个逻辑体，然后你可以决定将这个逻辑体放置到另一个已经存在的几何体中。然后决定将这个几何体的中心放置到另一个几何体的什么位置，并且如何旋转。一旦这些都确定了，那么就可以创建一个物理体了。物理体是一个已经放置的几何体实例，它包含几何体的所有属性。

2.2.6 创建一个物理几何体

通过逻辑体来创建一个物理体。一个物理体是一个已经放置了的逻辑体实例。这个实例必须被放置在另外一个逻辑母几何体之中。为了简化，下面的例子是不旋转的：

```
G4double trackerPos_x = -1.0*meter;
G4double trackerPos_y =  0.0*meter;
G4double trackerPos_z =  0.0*meter;

G4VPhysicalVolume* tracker_phys
  = new G4PVPlacement(0,                      // 不旋转
                  G4ThreeVector(trackerPos_x,trackerPos_y,trackerPos_z),
                                            // 子体在母体中的坐标
                  tracker_log,              // 子逻辑体指针
                  "tracker",                // 物理体名
                  experimentalHall_log,     // 母逻辑体指针
                  false,                    // 无布尔操作
                  0);                       // 物理体的拷贝序号
```

代码清单 2.2.3
一个简单物理体

以上将逻辑体 `tracker_log` 放置在母体 `experimentalHall_log` 的坐标原点，不旋转。生成的物理体叫"tracker"，它的拷贝序号为 0。

一个物理体必须放置在另一个母体之中的规则有一个例外。那就是世界(World)，它是被创建的最大的几何体，包含其他所有几何体。这个几何体很明显不可以被包含于其他任何几何体

中。它必须使用空母体指针通过 *G4PVPlacement* 来创建，它同样必须是不旋转的，必须被放置于全局坐标原点。

通常，最好选用一个简单的实体作为世界。在例 N01，我们使用 experimental hall：

```
G4VPhysicalVolume* experimentalHall_phys
   = new G4PVPlacement(0,                      //不旋转
                       G4ThreeVector(0.,0.,0.), //子体在母体中的坐标
                       experimentalHall_log,    //子逻辑体指针
                       "expHall",               //物理体名
                       0,                      //母逻辑体指针
                       false,                   //无布尔操作
                       0);                     //物理体的拷贝序号
```

代码清单 2.2.4
例 N01 中的世界

2.2.7 坐标系和旋转

在 Geant4 中，与一个已放置的物理体相关的旋转矩阵描述的是这个几何体的坐标系相对于母体坐标系的旋转变换。

一个旋转矩阵的建立通常与 CLHELP 一样，通过实例化一个单位矩阵，然后对这个单位矩阵进行旋转变换。在例 N04 中有这样的示例。

2.3 如何指定探测器的材料

2.3.1 通常情况

在自然界，通常材料(化合物，混合物)是由元素组成的，元素又是由同位素组成的。因此，在 Geant4 中有三个主要的类，每一个类都有一个表作为一个静态数据成员，用于跟踪这些类各自的实例。

G4Element 描述原子属性：

- 原子序数，
- 核子数 ，
- 原子质量，
- 壳层能量，
- 和其他量，如原子截面，等等

G4Material 描述物质的宏观属性：

- 密度，
- 状态，
- 温度，
- 压强，
- 和其他宏观量，如辐射长度，平均自由程，单位长度能损，等等。

G4Material 对工具包中的其它部分是可见的，用于粒子跟踪，几何体，和物理过程。它包含可能是其组成成分的元素和同位素的所有信息，同时，它隐藏这些实现细节。

2.3.2 定义一种简单材料

在下例中，通过指定材料名字，密度，摩尔质量和原子数创建了液 Ar。

```
G4double density = 1.390*g/cm3;
G4double a = 39.95*g/mole;
G4Material* lAr = new G4Material(name="liquidArgon", z=18., a, density);
```

代码清单 2.3.1
创建液 Ar

lAr 为指向该材料的指针，将用于指定一个给定逻辑体的材料：

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

2.3.3 定义一种分子

在下例中，通过指定组成成分的分子中各种原子的数目，创建了水这种材料。

```
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);
```

代码清单 2.3.2
通过定义分子来创建水.

2.3.4 通过质量百分数定义一种混合物

下例中，通过指定各种成分的质量百分数，我们创建了空气这种混合物。

```
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);
```

代码清单 2.3.3
通过定义组分的质量百分数创建空气

2.3.5 打印材料信息

```
G4cout << H2O;                          \\ print a given material
```

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

代码清单 2.3.4
打印材料信息

在/novice/N03/N03DetectorConstruction.cc 中，你可以找到所有创建材料的方法的示例。

2.4 如何指定粒子

G4VuserPhysicsList 是在 2.1 节中所述的用户必要的基类之一。在这个类中，所有将被用于模拟的粒子和物理过程必须被定义。截断范围参数也应该在这个类中定义。

用户必须创建一个由 G4VuserPhysicsList 派生的类并实现下列纯虚方法：

ConstructParticle(): 粒子构造
ConstructProcess(): 物理过程构造并向相关粒子注册这些过程
SetCuts(): 为所有粒子设定一个截断值

本节提供一些 ConstructParticle() 和 SetCuts()方法的简单例子。有关 ConstructProcess() 方法的信息，请看 2.5 节。

2.4.1 粒子定义

Geant4 为用户提供了各种类型的粒子：

- 普通粒子，如电子，质子，gamma
- 短寿命共振粒子，如矢量介子和 delta 重子
- 原子核，如氚核，α 粒子和重离子
- 夸克，底夸克，胶子

每个粒子都由各自的类来描述，这些类都是由 G4ParticleDefinition 派生的。这些粒子主要分为 6 类：

- 轻子，
- 介子，
- 重子，
- 玻色子，
- 短寿命粒子
- 离子，

上述每一类粒子都被定义在 geant4/source/particles 下的相应子目录中。同样，对于每一类粒子都有一个相应的库。

2.4.1.1 G4ParticleDefinition 类

G4ParticleDefinition 中有用于区分每个粒子的属性，如，名字，质量，电荷，自旋，等等。这些属性中，大部分是“只读”的，用户要对这些属性进行更改，就必须重建库。

2.4.1.2 如何存取一个粒子

每个粒子类代表一个独立的粒子，并且，每个类都有一个唯一的静态对象[1][1]。这条规则有一些例外，详细情况请参看 5.3 节 。

例如，G4Electron 代表电子，它的唯一对象是 G4Electron::theElectron。指向这个对象的指针可以通过静态方法 G4Electron::ElectronDefinition()获取。

缺省情况下，Geant4 提供了超过 100 种粒子用于各种物理过程。在通常的应用程序中，用户不需要定义他自己的粒子。

由于粒子是 singleton 粒子类的静态对象，这些对象在 main() 函数执行之前自动被实例化。然而，你必须显式的声明那些将被你的应用程序使用的粒子类，否则，编译程序不能识别哪些类是你需要的，结果将是没有粒子将被实例化。

2.4.1.3 粒子字典

G4ParticleTable 类是一个粒子字典。它提供了各种实用方法，如：

FindParticle(G4String name): 通过名字查找粒子

FindParticle(G4int PDGencoding): 通过 PDG 编码查找粒子

G4ParticleTable 同样定义为一个 singleton 对象，G4ParticleTable::GetParticleTable() 方法提供这个对象的指针。

粒子在构造期间被自动注册。用户不用控制粒子注册。

---

[1][1] 表明粒子类为 singleton 类

2.4.1.4 构造粒子

`ConstructParticle()` 是一个纯虚方法，所有在模拟中需要的粒子的静态成员函数应在这个方法中被调用。这保证了那些粒子的对象将被建立。注意，用户必须定义初级粒子和其它所有可能出现的次级粒子。

例如，假定用户需要一个质子和一个 geantino，geantino 是一个虚粒子，它不与介质发生作用。`ConstructParticle()` 方法的实现如下：

```
void ExN01PhysicsList::ConstructParticle()
{
  G4Proton::ProtonDefinition();
  G4Geantino::GeantinoDefinition();
}
```

代码清单 2.4.1
创建一个质子和一个 geantino.

由于有大量预定义的粒子，用这个方法列出所有的粒子非常麻烦。Geant4 有 6 个粒子大类，对应有 6 个实用类，它们可用于简化粒子构造的过程：

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

在 `ExN05PhysicsList` 中有这样的例子，代码如下。

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

代码清单 2.4.2
构造所有轻子

2.4.2 截断范围

为了避免红外发散，一些电磁过程要求设定一个低阈，在阈值以下，将不产生任何次级粒子。因此，gamma，电子和正电子要求用户设定阈值。阈值应定义为距离，或者截断范围，它将自动转换为对应于不同介质的截断能量。这个阈值应在用户初始化部分用 `G4VUserPhysicsList` 的 `SetCuts()` 方法定义。在 5.4 节中，详细讨论了阈值和粒子跟踪的截断。

2.4.2.1 设置截断

产物的阈值应在 `SetCuts()` 中定义，它是 `G4VUserPhysicsList` 类的纯虚方法。粒子、材料、物理过程的构造应在引用 `SetCuts()`之前。在通常的应用程序中， `G4RunManager` 关心这个顺序。

“唯一截断范围值”的思想是 Geant4 的一个重要特性，它用一个一致的方法处理截断值。对于多数应用来说，用户只要确定一个截断范围值，这个值将以同样的方式用于 gamma，电子，正电子。

在这种情况下，可以使用 `SetCutsWithDefault()` 方法，它由基类 `G4VuserPhysicsList` 提供，它有一个 `defaultCutValue` 成员作为缺省截断范围值。`SetCutsWithDefault()`使用这个值。.

可以对 gamma，电子和正电子设定不同的截断范围值，并且对不同的几何区域设置不同的截断范围值。在这种情况下，必须非常小心，因为 Geant4 进程(特别是能量损失)是遵循“唯一截断范围值”的模式设计的。

```
void ExN04PhysicsList::SetCuts()
{
  //   the G4VUserPhysicsList::SetCutsWithDefault() method sets
  //   the default cut value for all particle types
  SetCutsWithDefault();
}
```

代码清单 2.4.3
使用缺省截断值设定截断值

`defaultCutValue` 在缺省情况下设置为 1.0 mm。当然，你可以在你的 physics list 类中设置新的缺省值。

```
ExN04PhysicsList::ExN04PhysicsList():  G4VUserPhysicsList()
```

```
{
  // default cut value  (1.0mm)
  defaultCutValue = 1.0*mm;
}
```

代码清单 2.4.4
设置缺省截断值

也可以使用在 G4VUserPhysicsList 中的 SetDefaultCutValue() 方法，和 "/run/setCut" 命令来交互的改变缺省截断值。

警告：不要在一个事件循环内改变截断值。但截断值可以在两次 run 之间改变。

SetCuts() 方法的一个实现方法如下：

```
void ExN03PhysicsList::SetCuts()
{
  // set cut values for gamma at first and for e- second and next for e+,
  // because some processes for e+/e- need cut values for gamma
  SetCutValue(cutForGamma, "gamma");
  SetCutValue(cutForElectron, "e-");
  SetCutValue(cutForElectron, "e+");
}
```

代码清单 2.4.5
SetCuts() 方法的实现例子

从 Geant4 5.1 版开始，可以为每个几何区域设置不同的截断值。这个新的功能将在 5.5 节 中进行讨论。

About the authors

2.5 如何指定物理过程

2.5.1 物理过程

物理过程描述粒子如何与物质相互作用。Geant4 提供了 7 个大类的这些过程。

- 电磁相互作用
- 强相互作用
- 输运
- 衰变
- 可见光
- photolepton_hadron, and
- 参数化相互作用

所有这些物理过程的基类都是 *G4VProcess* 。他的虚方法

- `AtRestDoIt`,
- `AlongStepDoIt`，和
- `PostStepDoIt`

和对应的方法

- `AtRestGetPhysicalInteractionLength`,
- `AlongStepGetPhysicalInteractionLength`, 和
- `PostStepGetPhysicalInteractionLength`

在它们的派生类中描述了这些物理过程的行为。这些方法将在 5.2 节讨论。

下面是用于单一物理过程的特殊化基类：

G4VAtRestProcess - 只有 `AtRestDoIt` 的物理过程
G4VContinuousProcess - 只有 `AlongStepDoIt` 的物理过程
G4VDiscreteProcess - 只有 `PostStepDoIt` 的物理过程

其余的 4 个虚类，如 *G4VContinuousDiscreteProcess* 用于复杂的物理过程

2.5.2 物理过程管理

G4ProcessManager 类包含了与粒子相关的一个物理过程列表。它包含了与物理过程顺序相关的信息，同时，在列表中的每一个物理过程都有一个相应的 `DoIt` 方法可用。一个 *G4ProcessManager* 对象与一个粒子相对应，并且与 *G4ParticleDefiniton* 类相关联。

为了使物理过程有效，它们应该向该粒子的 *G4ProcessManager* 对象注册。物理过程的顺序信息是用 `AddProcess()` 和 `SetProcessOrdering()` 方法添加的。`AddAtRestProcess()`, `AddContinuousProcess()` 和 `AddDiscreteProcess()` 方法可以用来注册简单的物理过程。

G4ProcessManager 可以在一个 run 过程中，使用 `ActivateProcess()` 和 `InActivateProcess()`方法来打开或关闭一些物理过程。这些方法只能在物理过程注册之后使用，所以它们不能在用户预初始化部分使用。

G4VUserPhysicsList 类创建 *G4ProcessManager* 对象，并将这些对象与在 `ConstructParticle()`方法中定义的所有粒子相关联。

2.5.3 指定物理过程

G4VUserPhysicsList 是"必要的用户类"的一个基类 (参看 2.1 节)，在这个类中，所有模拟中需要的物理过程和粒子必须被注册。用户必须构造一个从 *G4VUserPhysicsList* 类派生的类，并且实现它的纯虚方法 `ConstructProcess()`。

例如，如果在模拟过程中只有使用了 *G4Geantino* 粒子，那么只要注册输运过程就可以了。`ConstructProcess()` 方法将被实现，如下：

```
void ExN01PhysicsList::ConstructProcess()
{
  // Define transportation process
  AddTransportation();
}
```

代码清单 2.5.1
为 geantino 注册物理过程

在此，`AddTransportation()` 方法在 *G4VUserPhysicsList* 类中提供，用于向所有粒子类注册 *G4Transportation* 类。*G4Transportation* 类 (和/或相关类) 描述粒子在时空中的运动。它是实现粒子跟踪所必需的类。

物理过程应该在 `ConstructProcess()` 方法中创建并向 *G4ProcessManager* 类的每个粒子实例注册。

下面是一个在 *G4VUserPhysicsList*::`AddTransportation()` 方法中注册物理过程的例子。

在 *G4ProcessManager* 中注册其他物理过程和粒子是一个复杂的过程，因为对于某些物理过程来说，物理过程之间的关系是非常重要的。请参考 5.2 节 和例子。

下面是为光子注册电磁作用过程的示例：

```
void MyPhysicsList::ConstructProcess()
{
  // Define transportation process
  AddTransportation();
  // electromagnetic processes
  ConstructEM();
}
void MyPhysicsList::ConstructEM()
{
   //  Get the process manager for gamma
  G4ParticleDefinition* particle = G4Gamma::GammaDefinition();
  G4ProcessManager* pmanager = particle->GetProcessManager();

  // Construct processes for gamma
  G4PhotoElectricEffect * thePhotoElectricEffect = new G4PhotoElectricEffect();
  G4ComptonScattering * theComptonScattering = new G4ComptonScattering();
  G4GammaConversion* theGammaConversion = new G4GammaConversion();

  // Register processes to gamma's process manager
  pmanager->AddDiscreteProcess(thePhotoElectricEffect);
  pmanager->AddDiscreteProcess(theComptonScattering);
  pmanager->AddDiscreteProcess(theGammaConversion);
}
```

代码清单 2.5.2
给一个 gamma 注册物理过程

2.6 如何产生一个初级事件

2.6.1 产生初级事件

G4VuserPrimaryGeneratorAction 是一个用户必需的类，用户需要从它派生自己的类。在用户派生的类中，必须指定如何产生一个初级事件。实际上，初级粒子的产生是由

G4VPrimaryGenerator 类完成的，这将在下一小节中讨论。用户的 *G4VUserPrimaryGeneratorAction* 构造类，只是描述了初级粒子的产生方式。

```
#ifndef ExN01PrimaryGeneratorAction_h
#define ExN01PrimaryGeneratorAction_h 1

#include "G4VUserPrimaryGeneratorAction.hh"

class G4ParticleGun;
class G4Event;

class ExN01PrimaryGeneratorAction : public G4VUserPrimaryGeneratorAction
{
  public:
    ExN01PrimaryGeneratorAction();
    ~ExN01PrimaryGeneratorAction();

  public:
    void generatePrimaries(G4Event* anEvent);

  private:
    G4ParticleGun* particleGun;
};

#endif

#include "ExN01PrimaryGeneratorAction.hh"
#include "G4Event.hh"
#include "G4ParticleGun.hh"
#include "G4ThreeVector.hh"
#include "G4Geantino.hh"
#include "globals.hh"

ExN01PrimaryGeneratorAction::ExN01PrimaryGeneratorAction()
{
  G4int n_particle = 1;
  particleGun = new G4ParticleGun(n_particle);

  particleGun->SetParticleDefinition(G4Geantino::GeantinoDefinition());
  particleGun->SetParticleEnergy(1.0*GeV);
  particleGun->SetParticlePosition(G4ThreeVector(-2.0*m,0.0*m,0.0*m));
}

ExN01PrimaryGeneratorAction::~ExN01PrimaryGeneratorAction()
```

```
{
  delete particleGun;
}

void ExN01PrimaryGeneratorAction::generatePrimaries(G4Event* anEvent)
{
  G4int i = anEvent->get_eventID() % 3;
  switch(i)
  {
    case 0:
      particleGun->SetParticleMomentumDirection(G4ThreeVector(1.0,0.0,0.0));
      break;
    case 1:
      particleGun->SetParticleMomentumDirection(G4ThreeVector(1.0,0.1,0.0));
      break;
    case 2:
      particleGun->SetParticleMomentumDirection(G4ThreeVector(1.0,0.0,0.1));
      break;
  }

  particleGun->generatePrimaryVertex(anEvent);
}
```

代码清单 2.6.1
使用 *G4ParticleGun* 类的 *G4VUserPrimaryGeneratorAction* 例子， *G4ParticleGun* 的使用参考 2.6.2 节。

2.6.1.1 粒子发生器的选择

在用户的 *G4VUserPrimaryGeneratorAction* 类的构造函数中，用户应该初始化初级粒子发生器。如果必要的化，用户需要为这些粒子发生器设置一些初始值。

在代码清单 2.6.1 中， *G4ParticleGun* 是用来作为实际的粒子发生器的。*G4ParticleGun* 中的方法，将在下节中讨论。请注意，那些在用户的 *G4VUserPrimaryGeneratorAction* 构造函数中构造的初级粒子发生器的对象，必须在相应的析构函数中删除。

2.6.1.2 事件的产生

G4VUserPrimaryGeneratorAction 类，有一个纯虚方法 `generatePrimaries()`。这个方法在每一个事件的开始被调用。在这个方法中，用户必须调用具体 *G4VPrimaryGenerator* 类的 `generatePrimaryVertex()` 方法。

用户可以调用一个或多个粒子发生器，也可以多次调用同一个粒子发生器。混合使用几个发生器可以产生一个非常复杂的初级事件。

2.6.2 *G4VPrimaryGenerator*类

Geant4 提供了两个 *G4VPrimaryGenerator* 类。一个是 *G4ParticleGun* 类，将在本节讨论，另一个是 *G4HEPEvtInterface* 类，将在 3.6 节讨论。

2.6.2.1 *G4ParticleGun* 类

G4ParticleGun 是 Geant4 提供的一个粒子发生器类。这个类用一个给定的动量和位置产生初级粒子。它不提供任何的随机模式。*G4ParticleGun* 的构造函数使用一个整型参数用于指定产生多少个完全相同的粒子。通常，用户要求产生一个具有随机能量，动量，和位置的初级粒子。这种随机特性可以通过调用 *G4ParticleGun* 类提供的 set 方法来获得。引用这些方法应在用户的 *G4VUserPrimaryGeneratorAction* 类的 `generatePrimaries()`方法中实现，还应在调用 *G4ParticleGun* 的 `generatePrimaryVertex()`方法之前。Geant4 提供了各种分布的随机数发生器的方法 (参看 3.2 节)。

2.6.2.2 *G4ParticleGun* 类的共有方法

G4ParticleGun 类提供了下列方法，它们都可以在用户构造的 *G4VUserPrimaryGeneratorAction* 类中的 `generatePrimaries()`方法中调用。

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

2.7 如何生成（Make）一个可执行程序

2.7.1 在一个 UNIX 环境下编译例子 1 （ExampleN01）

Geant4 提供的例子放在目录$G4INSTALL/examples 下，这里的$G4INSTALL 是一个环境变量，它被设置为 Geant4 的安装目录 (缺省设置是$HOME/geant4)。下面，我们快速浏览一些在 Geant4 中 GNUmake 的工作机制，并且演示如何编译一个具体的例子， "ExampleN01"。

2.7.1.1 在 Geant4 中 GNUmake 是如何工作的

在 Geant4 中 GNUmake 进程主要由以下的 GNUmake 脚本文件控制 (*.gmk 脚本存放路径为 $G4INSTALL/config)：

architecture.gmk	调用并定义所有特定设置和指定路径的结构，这些信息存放在 $G4INSTALL/config/sys
common.gmk	定义所有用于建立目标文件和库的 GNUmake 通用规则
globlib.gmk	定义所有用于建立复合库的 GNUmake 通用规则
binmake.gmk	定义用于建立可执行文件的 GNUmake 通用规则
GNUmakefile	存放在 Geant4 的每一个目录下，定义了建立一个库，一个库的集合或一个可执行文件的特定指令

Geant4 的内核库缺省存放位置为$G4INSTALL/lib/$G4SYSTEM，这里的$G4SYSTEM 指定了当前使用的系统和编译器。可执行的二进制文件被存放在$G4WORKDIR/bin/$G4SYSTEM，临时文件(目标文件和编译过程中输出的有关文件)存放在$G4WORKDIR/tmp/$G4SYSTEM。 $G4WORKDIR (缺省设置为$G4INSTALL) 应被用户指定为他自己特定的工作目录。

有关如何建立 Geant4 内核库和正确设置 Geant4 环境变量的其他信息，请参考"Installation Guide"。

2.7.1.2 建立可执行文件

Geant4 的例子存放在$G4INSTALL/examples，用户可以在特定例子的子目录下执行 gmake 来进行编译。要建立可执行文件，用户需要先将$G4INSTALL/examples 复制到用户的工作目录 $G4WORKDIR 下，然后：

```
> cd $G4WORKDIR/examples/novice/N01
> gmake
```

结果将在$G4WORKDIR/bin/$G4SYSTEM,中生成"exampleN01"的可执行文件，然后将 $G4WORKDIR/bin/$G4SYSTEM 添加到环境变量 $PATH 中，就可以执行生成的可执行文件了。

2.7.2 在一个 Windows 环境中编译例子 1 （ExampleN01）

在一个居于 Windows/95-98 or Windows/NT 的环境中建立一个 Geant4 可执行文件与在 UNIX 环境下相似，前提是你需要安装 GNUmake， MS-Visual C++ 编译器和其他运行 Geant4 需要的软件 (参看 "Installation Guide")。

2.7.2.1 建立可执行文件

参看 2.7.1 节。

About the authors

Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Getting Started with Geant4

2.8 如何建立一个交互式程序接口（Session ）

2.8.1 简介

"intercoms" 类的角色

"intercoms"这一组类提供了一个可扩展的命令解释器。它是 Geant4 实现用户交互的一个关键机制，使用户不必关心各个大类之间的相互关系。用户可以在 C++程序中直接使用 Geant4 提供的类，以提供最底层的交互特性 ，例如，批处理会话方式。如例子 examples/novice/N01 中所示 ， Geant4 命令和宏是被嵌入到程序代码中的。

使用用户接口控制模拟过程

为了避免大量的编程工作， "intercoms" 提供了抽象类 *G4UIsession* ，同可以捕获交互命令。用户接口和图形用户接口的具体实现在接口(interfaces)这一大类中。这种接口方式使 Geant4 可以使用各种接口工具，允许 Geant 利用目前已经十分完善的图形工具，如 Motif 和 Java，等等。广泛的合作使得不同的小组为 Geant4 命令系统提供了各种接口，目前可用的如下；

1. 字符终端(哑终端和类 tcsh(bash)终端)，这是 Geant4 的缺省用户接口，
2. Xm, Xaw, Win32, 各种使用 Motif，Athena 或 Windows 工具集的上层终端，
3. GAG, 一个完全的图形用户接口和客户/服务器方式的扩展 GainServer，
4. OPACS, 一个与 OPACS 可视化系统一起实现的 OPACS/Wo 工具集管理程序

上述 1 和 2 中的字符终端的完全实现代码可以在 Geant4 的 source/interfaces/basic 目录中找到。至于 3 和 4，有丰富的图形接口功能，它们的前端类实现代码在 Geant4 目录 source/interfaces/GAG 或 OPACS 中。相应的 GUI 工具包可以分别从相关的研究机构的网站上获取。

GAG, GainServer's client GUI Gain	http://erpc1.naruto-u.ac.jp/~geant4
OPACS	http://www.lal.in2p3.fr/OPACS

2.8.2 可用接口类简述

1. *G4UIterminal* 和 *G4UItcsh* 类

这些接口在一个字符终端上建立一个会话。*G4UIterminal* 可以运行在 Geant4 支持的所有平台上，包括在 Windows 上的 *cygwin* ，而 *G4UItcsh* 只运行在 Solaris 和 Linux 上。G4UItcsh 支持与 tcsh(或 bash)方式的用户友好键；

^A	移动光标到当前输入的第一个字符
^B	光标回退一个字符 (←光标)
^D	删除/退出/显示 匹配串
^E	移动光标到末尾
^F	光标前移一个字符 (→光标)
^K	清除光标以后的部分
^N	下一个命令(↓光标)
^P	前次输入的命令（↑ 光标）
TAB	补全命令
DEL	回退并删除一个字符
BS	回退并删除一个字符

另外，支持下列宏替换；

%s	当前应用程序状态
%/	当前工作目录
%h	历史序号

2. *G4UIXm, G4UIXaw* 和 *G4UIWin32* 类

这些接口类是 *G4UIterminal* 分别居于 Motif, Athena 和 WIN32 库实现的版本。*G4UIXm* 使用 Motif XmCommand 工具集，*G4UIXaw* 使用 Athena 对话框工具集，*G4UIWin32* 使用 Windows 的"edit"组件实现命令捕获。如果使用居于 Xt 库或者 WIN32 库的可视化引擎，这些接口将是很有用的。

命令窗是用来处理 Geant4 命令的输入和调用的。在命令行状态下，可以用 TAB 键补全命令。支持"exit, cont, help, ls, cd..."等 shell 命令。用户可以通过 *AddMenu* 和 *AddButton* 方法定制菜单条。

例如：

```
/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 运行在 Unix/Linux 平台上，需要 Motif 的支持。*G4UIXaw* 运行在 Unix 平台上，需要 Athena 工具集的支持，用户友好性方面稍差。*G4UIWin32* 运行在 Windows 平台上。

3. *G4UIGAG* 和 G4UIGainServer 类

它们是 Geant4 的前端类，有各自的图形用户接口，GAG(Geant4 Adaptive GUI) 和 Gain (Geant4 adaptive interface for network)。GAG 必须运行于与 Geant4 应用程序相同的运行平台上；Gain 可以运行于一个安装了 JVM(Java 虚拟机)的远程系统上 (Windows, Linux, 等) ，Geant4 应用程序在 Unix (Linux)系统运行，作为服务器端，它打开一个通讯端口，等待来自 Gain 的连接，Gain 可以被连接到位于不同研究机构的，运行在 Unix(Linux)上的多个 Geant4 服务器端。

对于 GUI 的客户端来说， GAG 和 Gain 是非常相似的。所以，在此只简单介绍一下 GAG 的功能。更详细的信息请参看相关的网页

GAG 是一个图形用户接口工具，用于设置参数和执行命令。它是非常有用的，因为 GAG 反应了 Geant4 这个状态机的内部状态。GAG 是居于服务器/客户模式的，GAG 是服务器端，而 Geant4 应用程序是客户端。然而，GAG 本身什么都没做，它必须调用 Geant4 应用程序。Geant4 的前端类 *G4UIGAG* 必须初始化为与 GAG 通讯。它可以运行于 Linux 和 Windows 2000。

GAG 是用 Java 写的，它的 jar(Java Archive)文档可以从网上找到。在它的网页上还可以了解如何安装并运行 Java 应用程序。

GAG 有下列功能。

- GAG 菜单： 这些菜单用于选择并运行 GEANT4 可执行文件，kill 或退出 GEANT4 进程和退出 GAG。在正常退出或 Geant4 进程死亡之后，GAG 将被自动复位，以接受下一个 Geant4 程序的运行。
- GEANT4 命令树： 在与 Geant4 进程建立管道之后，GAG 将显示与 Windows 文件浏览器相似的命令菜单树。不可用的命令用灰色显示。GAG 不显示命令树根部以下的命令，即只显示命令树的枝叶。直接输入区域可以用来输入那些不可见的命令。命令和命令类的参考信息将显示在当前焦点(focus)位置。GAG 有历史命令功能，用户可以用上次执行时的参数再次执行一个命令，可以编辑这些历史，或者将这些历史存入一个宏文件中
- 命令参数面板： GAG 的参数面板非常友好的部分。它显示了参数名，参数的使用说明，参数的类型(整型，双精度型，布尔型或字符串型)，是否可忽略，缺省值，参数范围和可用参数列表。参数的范围检测是由 intercoms 的类完成的，错误信息将被显示在弹出对话框中。如果一个参数有一个可用参数列表，这个列表框将被自动显示。如果一个命令需要使用一个文件，那么文件选择框将是可用的。
- 日志：日志可以重定向到调用 GAG 的终端 (xterm 或 cygwin 窗口) ，也可以在一次会话过程中被中断，也可以被存储到一个文件中。GAG 用一个弹出式警告工具集显示来自于 Geant4 的警告或错误信息。

4. *G4UIOPACS* 类

- OPACS 是 Geant4 的一个可视化环境： OPACS 是一个居于 X window 和 OpenGL 的可视化环境。它为 HEP (High Energy Physic)环境开发，用于“事件显示”的工具。它用 ANSI C 编写，非常易于移植。目前已经移植到大多数 UNIX 系统，VMS 系统和安装了 X11 的 Window/NT 系统上。如果使用 OPACS 作为可视化环境，只需使用 Geant4 内核就可以了,用户接口和图形直接将由 OPACS 处理。使用这种方法，G4/source/interfaces, G4/source/visualization 这两个模块都是不需要的。
- OPACS 是 Geant4 的一个可视化引擎：*G4UIOPACS* 是 OPACS 的前端类。要使 Geant4 使用 OPACS 作为可视化环境，首先必须安装 OPACS 并遵循一定的规范，有关信息请浏览上面提到的相关网页。用户可以编译 Geant4/source/visualization/tests/test19 来测试 OPACS。

2.8.3 建立接口库

在缺省情况下，建立这些库，不需要外部软件包。它们包括 *libG4UIbasic.a/so* and *libG4UIGAG.a/so* ，这些库包含了 *G4UIterminal*, *G4UItcsh* 和 *G4UIGAG* 。*G4UIGainServer.o* 被包含于 *libG4UIGAG* 中。

要建立 *G4UIXm*, *G4UIXaw* 和 *G4UIWin32* 的相关库，它们各自的环境变量必须被明确的设置。要建立 OPACS 库， 环境变量 **G4UI_BUILD_OPACS_SESSION** 必须被设置

但是，如何环境变量 **G4UI_NONE** 被设置，所有的接口库都不会被建立。

用户接口库的建立流程在"$G4INSTALL/config/G4UI_BUILD.gmk" 中描述，它与外部软件包的相关性在"$G4INSTALL/config/interactivity.gmk" 中被描述。

2.8.4 如何使用交互式接口

要在用户程序中使用指定的接口 (`G4UIxxx` 这里的 `xxx = terminal,Xm, Xaw, Win32, GAG, GainServer, Wo`) ，需要在主程序中加入如下行；

- // to include the class definition in his main program:
- #include "G4Uixxx.hh"
- // to instantiate a session of his choice and start the session
- G4UIsession* session = new G4UIxxx;
- session->SessionStart();
- / /the line next to the "SessionStart" is usually to finish the session
- delete session;

对于一个 tcsh 会话来说，第二行是必需的：

G4UIsession* session = new G4UIterminal(new G4UItcsh);

参看例子"examples/novice/N0x" ，在这些例子中有些使用了终端作为用户会话接口。

另外，使用环境变量选择一个给定的接口。但是，为了方便，其中有一些缺省已被设置。

- *G4UIterminal*, *G4UItcsh*, *G4UIGAG* 和 *G4UIGainServer* 可以直接使用，不需要设置环境变量。这些会话接口不需要使用外部软件包或库（参看"G4UI_BUILD.gmk"），因此，用户不需要设置任何环境变量，也不要重建任何库，就可以实例化这些会话接口中的任何一个。为了用户应用程序的向下兼容，与上述三个环境变量 (**G4UI_USE_TERMINAL**, **G4UI_USE_TCSH** 和 **G4UI_USE_GAG**) 相对应的 C 预编译变量应被设置。如果用户不设置任何环境变量，C 预编译变量 **G4UI_USE_TERMINAL** 缺省将被设置，虽然没有必要使用它。
- 如果要使用 `Xm, Xaw, Win32, Wo` 这些接口，环境变量 **G4UI_USE_XM**, **G4UI_USE_XAW**, **G4UI_USE_WIN32** 或 **G4UI_USE_WO** 必须要设置，"$G4INSTALL/config/interactivity.gmk" 将解析它们与外部软件包的相关性。
- 如果环境变量 **G4UI_NONE** 被设置，将不使用任何外部软件库。同样，为了方便用户，如果环境变量 **G4UI_USE_XXX** 被设置，相应的 C 预编译标志也将被设置。但是，如果环境变量 **G4UI_NONE** 被设置，不需要设置相应的 C 预编译标志。

2.9 如何运行一个程序

2.9.1 简介

Geant4 应用程序能够用下列方式运行

- `purely hard-coded` 批处理模式
- 使用宏命令的批处理模式
- 命令行驱动的交互模式
- 使用图形用户接口的交互模式

最后一种模式将在 2.8 节论述，前三种模式将在本章讨论。

2.9.2 ’Hard-coded’ 批处理模式

下面是一个将运行于批处理模式的应用程序的 main ( )函数例子。

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

  // set mandatory initialization classes
  runManager->SetUserInitialization(new ExN01DetectorConstruction);
  runManager->SetUserInitialization(new ExN01PhysicsList);

  // set mandatory user action class
  runManager->SetUserAction(new ExN01PrimaryGeneratorAction);

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

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

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

代码清单 2.9.1
一个将运行于批处理模式的应用程序的 main ( )函数例子

在运行过程中，即使是事件数也不能更改，用户要改变事件数必须重写 main()函数。

2.9.3 使用宏文件的批处理模式

下面是一个将使用宏文件的批处理模式的应用程序的 main ( )函数例子

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

   // Construct the default run manager
  G4RunManager * runManager = new G4RunManager;

  // set mandatory initialization classes
  runManager->SetUserInitialization(new MyDetectorConstruction);
```

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

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

  // 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);

  delete runManager;
  return 0;
}
```

代码清单 2.9.2
一个将使用宏文件的批处理模式的应用程序的 main ( )函数例子

这个例子将使用如下命令执行：

```
> myProgram  run1.mac
```

这里的 myProgram 是用户可执行文件名， run1.mac 是一个位于当前目录的命令宏文件，典型的宏命令文件如下：

```
#
# Macro file for "myProgram.cc"
#
# 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
```

代码清单 2.9.3
一个典型的命令宏文件

用户可以用不同的运行参数来再次执行应用程序，而不需要重写程序。

另： Geant4 的各个模块的类都有一个冗余标志，用于控制冗余信息的输出。
通常 `verbose=0` 意味这没有信息输出。例如

- /run/verbose 用于设置 RunManager 的冗余标志
- /event/verbose 用于设置 EventManager 的冗余标志
- /tracking/verbose 用于设置 TrackingManager 的冗余标志
- ...等...

2.9.4 命令行驱动的交互模式

下面是一个运行于交互模式的应用程序的 main( )函数，由命令行驱动，它将等待用户命令的输入。

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

  // Construct the default run manager
  G4RunManager * runManager = new G4RunManager;

  // set mandatory initialization classes
  runManager->SetUserInitialization(new MyDetectorConstruction);
  runManager->SetUserInitialization(new MyPhysicsList);

  // visualization manager
  G4VisManager* visManager = new MyVisManager;
  visManager->Initialize();

  // set user action classes
  runManager->SetUserAction(new MyPrimaryGeneratorAction);
  runManager->SetUserAction(new MyRunAction);
  runManager->SetUserAction(new MyEventAction);
  runManager->SetUserAction(new MySteppingAction);

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

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

  // job termination
  delete visManager;
  delete runManager;

  return 0;
}
```

代码清单 2.9.4
一个运行于交互模式的应用程序的 main( )函数，由命令行驱动，它将等待用户命令的输入。

这个例子将由下列命令来执行：

```
  > myProgram
```

这里的 myProgram 是用户可执行文件的名字。

将出现如下的 Geant4 内核提示符：

```
  Idle>
```

然后，用户可以开始输入会话命令。例如：

建立一个空的 scene (缺省是"world")：

```
   Idle> /vis/scene/create
```

添加一个几何体到这个 scene：

```
  Idle> /vis/scene/add/volume
```

为特定图形系统（例如 OGLIX）建立一个 scene handler：

```
  Idle> /vis/sceneHandler/create OGLIX
```

建立一个 viewer：

```
  Idle> /vis/viewer/create
```

显示这个 scene，等：

```
  Idle> /vis/scene/notifyHandlers
  Idle> /run/verbose     0
  Idle> /event/verbose    0
  Idle> /tracking/verbose 1
  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
```

有关机器空闲(idle)状态的描述，请看 3.4.2 节。

对应在调试模式中运行一些事件并显示它们，这个模式是非常有用的。需要注意在 main() 函数中建立的 *VisManager* 和由下列命令选择的可视化系统：

```
/vis/sceneHandler/create OGLIX
```

2.9.5 通常情况

在$G4INSTALL/examples/ 目录下的大多数例子有如下 main()函数，它复合上面提到的第 2、第 3 种模式。因此，这些应用程序可以运行于这两种交互模式。

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

  // Construct the default run manager
  G4RunManager * runManager = new G4RunManager;

  // set mandatory initialization classes
  N03DetectorConstruction* detector = new N03DetectorConstruction;
  runManager->SetUserInitialization(detector);
  runManager->SetUserInitialization(new N03PhysicsList);

#ifdef G4VIS_USE
  // visualization manager
  G4VisManager* visManager = new N03VisManager;
  visManager->Initialize();
#endif

  // set user action classes
  runManager->SetUserAction(new N03PrimaryGeneratorAction(detector));
  runManager->SetUserAction(new N03RunAction);
  runManager->SetUserAction(new N03EventAction);
  runManager->SetUserAction(new N03SteppingAction);

  // get the pointer to the User Interface manager
    G4UImanager* UI = G4UImanager::GetUIpointer();

  if (argc==1)   // Define UI terminal for interactive mode
    {
```

```
    G4UIsession * session = new G4UIterminal;
    UI->ApplyCommand("/control/execute prerunN03.mac");
    session->SessionStart();
    delete session;
   }
  else          // Batch mode
   {
    G4String command = "/control/execute ";
    G4String fileName = argv[1];
    UI->ApplyCommand(command+fileName);
   }

  // job termination
#ifdef G4VIS_USE
  delete visManager;
#endif
  delete runManager;

  return 0;
}
```

代码清单 2.9.5
一个典型的 main() 函数例子

注意，可视化系统部分是由预编译变量 G4VIS_USE 控制的，在交互模式中，一些初始化参数被放在宏文件 prerunN03.mac 中，它将在会话开始前被执行。

```
# Macro file for the initialization phase of "exampleN03.cc"
#
# Sets some default verbose flags
# and initializes the graphics.
#
/control/verbose 2
/control/saveHistory
/run/verbose 2
#
/run/particle/dumpCutValues
#
# Create empty scene ("world" is default)
/vis/scene/create
#
# Add volume to scene
/vis/scene/add/volume
```

```
#
# Create a scene handler for a specific graphics system
# Edit the next line(s) to choose another graphic system
#
#/vis/sceneHandler/create DAWNFILE
/vis/sceneHandler/create OGLIX
#
# Create a viewer
/vis/viewer/create
#
# Draw scene
/vis/scene/notifyHandlers
#
# for drawing the tracks
# if too many tracks cause core dump => storeTrajectory 0
/tracking/storeTrajectory 1
#/vis/scene/include/trajectories
```

代码清单 2.9.6
宏文件 prerunN03.mac

同时，这例子表明用户可以交互的读取并执行一个宏命令文件：

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

2.10 如何可视化探测器和事件

2.10.1 介绍

本节简要介绍如何可视化 Geant4。所有的讨论居于例子 examples/novice/N03。更多的细节在第 8 章 "可视化"中 ， 有关宏文件请参看例子 examples/novice/N03/visTutor/exN03VisX.mac。

2.10.2 可视化引擎

Geant4 可以相应用户的各种可视化要求，使用内建可视化引擎的方法是非常困难的，因此，Geant4 只定义了一个抽象接口。以下提及的“图形系统”是指一个独立于 Geant4 运行的进程，或者是一个与 Geant4 运用程序链接的图形库。Geant4 支持几个具体的图形接口，它们在许多方面是互补的。一个具体图形系统接口被称为“可视化引擎”。

用户根据需要选择合适的可视化引擎，并不需要一定使用可视化引擎。下面，我们假设 Geant4 已经安装了"OpenGL-Xlib 引擎"和"DAWNFILE 引擎"。

为了使用 DAWNFILE 引擎，你需要先安装 Fukui Renderer DAWN，用户可以从 http://geant4.kek.jp/GEANT4/vis 下载。

要使用 OpenGL，需要 OpenGL 库。在多数系统中，缺省情况，OpenGL 库已经被安装。用户需要在编译 Geant4 工具包之前设置环境变量 `G4VIS_BUILD_OPENGLX_DRIVER ＝ 1` ，同时，还需要在编译 Geant4 应用程序的时候设置环境变量 `G4VIS_USE_OPENGLX=1`，设置环境变量 `OGLHOME` ＝OpenGL 安装目录。例如，

```
setenv G4VIS_BUILD_OPENGLX_DRIVER 1
setenv G4VIS_USE_OPENGLX 1
setenv OGLHOME /usr/X11R6
```

某些使用外部软件库的可视化引擎也需要设置环境变量 `G4VIS_BUILD_DRIVERNAME_DRIVER` 和 `G4VIS_USE_DRIVERNAME` 为 `1`。 所有不使用外部软件库的可视化引擎不需要设置，例如，DAWNFILE 和 VRMLFILE 。但是，用户必须编写一个适当的可视化管理类和 `main()` 函数。

对于所有 Geant4 可用的可视化引擎，在编译过程中，GNUmake 将根据 `config/G4VIS_USE.gmk` 自动设置 C 预编译标志 `G4VIS_USE_DRIVERNAME`。 同样，对于那些与 Geant4 库链接的可视化引擎，在工具包安装过程中，GNUmake 将根据 `config/G4VIS_BUILE.gmk` 自动设置 C 预编译标志 `G4VIS_BUILD_DRIVERNAME_DRIVER`。

详情，请阅读文档 `source/visualization/README`。

2.10.3 如何将可视化引擎链接到一个可执行文件

用户可以在 Geant4 应用程序中使用特定的可视化引擎。但是这些可视化引擎必须已经安装在 Geant4 库中，否则你将得到一些警告信息。

要使用可视化引擎，用户必须完成两件事：

1. 用户必须从基类 *G4VisManager* 继承并实现自己的可视化管理类，并且注册使用的可视化引擎，用户还需要实现一个纯虚函数 `void RegisterGraphicsSystems()`。用户可以在 `source/visualization/management` 中找到 *G4VisManager* 的定义。
2. 在 `main()`函数中实例化并初始化这个可视化管理类。

例子 `examples/novice/N03` 中，`ExN03VisManager::RegisterGraphicsSystems()` 例举了注册可视化引擎的过程。

例如， DAWNFILE 和 OpenGL-Xlib 注册过程如下：

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

下面讨论如何编写 main() 函数。

2.10.4 编写一个包含可视化的 main() 函数

现在，我们讨论如何编写一个 Geant4 可视化管理类和 main() 函数。为了可视化 Geant4 应用程序，用户必须在 main()函数中实例化并初始化用户的可视化管理类。典型的可视化的 main() 函数如下：

```
//----- C++ source codes: main() function for visualization
// Use class ExN03VisManager to define Visualization Manager
#ifdef G4VIS_USE
#include "ExN03VisManager.hh"
#endif

.....

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

.....

  // Instantiation and initialization of the Visualization Manager
#ifdef G4VIS_USE
  // visualization manager
  G4VisManager* visManager = new ExN03VisManager;
  visManager->Initialize();
#endif

.....

  // Job termination
#ifdef G4VIS_USE
```

```
  delete visManager;
#endif

.....

  return 0;
}

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

代码清单 2.10.1
典型的可视化的 main( )函数

在可视化管理类的实例化，初始化和删除过程中，我们推荐使用宏命令。注意，在程序结束之前，用户必须删除实例化的可视化管理类。例子 `examples/novice/N03/exampleN03.cc` 是一个完整的可视化的 main( )函数。

2.10.5 Scene, Scene Handler, 和 Viewer

用户可以使用交互式的可视化命令实现几乎所有的 Geant4 可视化功能。在使用这些可视化命令的时候，有必要了解一下"scene", "scene handler"和 "viewer"的概念。"scene"是一个可视化的 3D 原始数据集。"scene handler" 是一个图像数据模式化工具，它对在"scene"中的原始数据进行处理以供后续的可视化。"viewer"根据"scene handler"处理完成的数据生成最终图像。这些概念与它们之间的关系，与 MFC 中“文档”、“模板”、“视”的概念与关系相类似。实际上，一个"scene handler"和一个"viewer"对应于一个可视化引擎。

Geant4 可视化的典型步骤如下：

1. 建立一个 scene handler 和一个 viewer。
2. 建立一个空的 scene。
3. 将原始 3D 数据添加到已建立的 scene 中。
4. 将当前的 scene handler 链接到当前 scene。
5. 设置相机参数、绘图方式（轮廓线/面）等
6. 使 viewer 执行可视化
7. 声明可视化结束并显示图像。

注意，以上的列表并不表示用户必须执行 7 条可视化命令，可以使用复合命令一次执行多个可视化命令。

2.10.6 可视化程序接口例子

在这节中，我们将列举一些典型的 Geant4 可视化会话方式，用户可以通过运行例子 geant4/examples/novice/N03 来测试。用不带参数的方式运行例子"exampleN03"，然后在"Idle>"状态符下执行命令。稍后，我们将讨论这些命令。 (注意，在 Geant4 工具包安装过程中，需要设置相应的环境变量，以使用 OpenGL-Xlib 和 DAWNFILE 图形引擎；在编译例子 "exampleN03"前，必须写安装 Fukui Renderer DAWN。)

2.10.6.1 探测器元件的可视化

下面是使用 OpenGL-Xlib 和 DAWNFILE 引擎进行探测器元件可视化的例子。前者列举了用最少的命令进行探测器元件可视化，后者列举了用户自定义的可视化（用户选择的物理体，坐标轴，文本，等）过程。

```
#################################################
#  vis1.mac:
#    A Sample macro to demonstrate visualization
#    of detector geometry.
#
#  USAGE:
#   Save the commands below as a macro file, say,
#   "vis1.mac", and execute it as follows:
#
#   % $(G4BINDIR)/exampleN03
#   Idle> /control/execute vis1.mac
#################################################

#################################################
# Visualization of detector geometry
#  with the OGLIX (OpenGL Immediate X) driver
#################################################

# Invoke the OGLIX driver:
#  Create a scene handler and a viewer for the OGLIX driver
/vis/open OGLIX

# Visualize the whole detector geometry
#  with the default camera parameters.
#  Command "/vis/drawVolume" visualizes the detector geometry,
#  and command "/vis/viewer/flush" declares the end of visualization.
#  (The command /vis/viewer/flush  can be omitted for the OGLIX
#   and OGLSX drivers.)
#  The default argument of "/vis/drawVolume" is "world".
/vis/drawVolume
/vis/viewer/flush

#########################################
# Visualization with the DAWNFILE driver
```

```
########################################

# Invoke the DAWNFILE driver
#  Create a scene handler and a viewer for the DAWNFILE driver
/vis/open DAWNFILE

# Bird's-eye view of a detector component (Absorber)
#  drawing style: hidden-surface removal
#  viewpoint  : (theta,phi) = (35*deg, 45*deg),
#  zoom factor: 1.1 of the full screen size
#  coordinate axes:
#    x-axis:red,  y-axis:green,  z-axis:blue
#    origin: (0,0,0),  length: 500 mm
#
/vis/viewer/reset
/vis/viewer/set/style        surface
/vis/viewer/zoom            1.1
/vis/viewer/set/viewpointThetaPhi  35 45
/vis/drawVolume             Absorber
/vis/scene/add/axes          0 0 0 500 mm
/vis/scene/add/text          0 0 0 mm  40 -100 -140   Absorber
/vis/viewer/flush

# Bird's-eye view of the whole detector components
#  * "/vis/viewer/set/culling global false" makes the invisible
#   world volume visible.
#   (The invisibility of the world volume is set
#    in ExN03DetectorConstruction.cc.)
/vis/viewer/set/style    wireframe
/vis/viewer/set/culling   global false
/vis/drawVolume
/vis/scene/add/axes      0 0 0 500 mm
/vis/scene/add/text      0 0 0 mm 50 -50 -200   world
/vis/viewer/flush
################### END of vis1.mac ###################
```

2.10.6.2 事件可视化

下面是使用 OpenGL-Xlib 和 DAWNFILE 进行事件(径迹)可视化的例子。前者举例说明了使用最少的可视化命令进行事件可视化，后者举例说明了用户自定义的可视化。注意，正确的描述 run 行为（action）和事件行为（action）类。 (参看下面的例子：
"examples/novice/N03/src/ExN03RunAction.cc",
"examples/novice/N03/src/ExN03EventAction.cc").

```
#####################################################
```

```
#  vis2.mac:
#    A Sample macro to demonstrate visualization
#    of events (trajectories).
#
#  USAGE:
#   Save the commands below as a macro file, say,
#   "vis2.mac", and execute it as follows:
#
#   % $(G4BINDIR)/exampleN03
#   Idle> /control/execute vis1.mac
################################################

#########################################################
# Store particle trajectories for visualization
/tracking/storeTrajectory 1
#########################################################

############################################################
# Visualization with the OGLSX (OpenGL Stored X) driver
############################################################

# Invoke the OGLSX driver
#  Create a scene handler and a viewer for the OGLSX driver
/vis/open OGLSX

# Create an empty scene and add detector components to it
/vis/drawVolume

# Add trajectories to the current scene
#  Note: This command is not necessary in, e.g.,
#       examples/novice/N03, since the C++ method DrawTrajectory()
#       is described in the event action.
#/vis/scene/add/trajectories

# Set viewing parameters
/vis/viewer/reset
/vis/viewer/set/viewpointThetaPhi  10 20

# Visualize one it.
/run/beamOn 1

############################################################
# Visualization with the DAWNFILE driver
############################################################

# Invoke the DAWNFILE driver
#  Create a scene handler and a viewer for the DAWNFILE driver
```

```
/vis/open DAWNFILE

# Create an empty scene and add detector components to it
/vis/drawVolume

# Add trajectories to the current scene
#  Note: This command is not necessary in exampleN03,
#       since the C++ method DrawTrajectory() is
#       described in the event action.
#/vis/scene/add/trajectories

# Visualize plural events (bird's eye view)
#  drawing style: wireframe
#  viewpoint  : (theta,phi) = (45*deg, 45*deg)
#  zoom factor: 1.5 x (full screen size)
/vis/viewer/reset
/vis/viewer/set/style  wireframe
/vis/viewer/set/viewpointThetaPhi   45 45
/vis/viewer/zoom       1.5
/run/beamOn           2

# Set the drawing style to "surface"
#  Candidates: wireframe, surface
/vis/viewer/set/style  surface

# Visualize plural events (bird's eye view) again
#  with another drawing style (surface)
/run/beamOn           2

################### END of vis2.mac ###################
```

2.10.7 常用的可视化命令

在本节中，我们将解释出现在上面例子中的每一个可视化命令。

2.10.7.1 `/vis/open` 命令

- **命令**
  `/vis/open [driver_tag_name]`
- **参数**
  一个可用的可视化引擎的名字或模式。
- **功能**
  建立一个可视化引擎，即一个 scene hander 和一个 viewer。

- **例：建立立即模式的 OpenGL-Xlib 引擎**
  ```
  Idle> /vis/open OGLIX
  ```
- **符注**
  列出可用的图形引擎标识名：

  ```
  Idle> help /vis/open
  ```
  或者
  ```
  Idle> help /vis/sceneHandler/create
  ```

  这个命令的结果，举例如下：

  .....
  ```
  Candidates : DAWNFILE OGLIX OGLSX
  ```
  .....

2.10.7.2 `/vis/viewer/` 命令

- **命令**
  ```
  /vis/viewer/reset
  ```
- **功能**
  复位相机参数和绘图样式。

- **命令**
  ```
  /vis/viewer/set/viewpointThetaPhi [<theta>] [<phi>] [<deg|rad>]
  ```
- **参数**
  参数"theta"和"phi"分别是相机的极角和方位角，缺省单位是“度”。
- **功能**
  设置一个在(theta, phi)方向的视点。
- **例：设置(70 deg, 20 deg)方向的视点**
  ```
  Idle> /vis/viewer/set/viewpointThetaPhi 70 20
  ```
- **附注**
  应为每一个 viewer 设置相机参数。它们可以用"`/vis/viewer/reset`"初始化。这个命令属 2.10.5 中的步骤 5。

- **命令**
  ```
  /vis/viewer/zoom [<scale_factor >]
  ```
- **参数**
  刻度因子，这个命令使图像的放大所设置的刻度因子倍。
- **功能**
  放大/缩小 图像。
- **例：放大 1.5 倍**
  ```
  Idle> /vis/viewer/zoom 1.5
  ```

- **附注**
  应为每一个 viewer 设置相机参数。它们可以用"/vis/viewer/reset"初始化。这个命令属 2.10.5 中的步骤 5。

- **命令**
  /vis/viewer/set/style [style_name]
- **参数**
  候选参数是"wireframe" 和 "surface". ("w" 和 "s"为它们的缩写)
- **功能**
  设置绘图样式。
- **例:设置绘图样式为"surface"**
  Idle> /vis/viewer/set/style surface
- **附注**
  应为每个 viewer 设置绘图样式。命令"/vis/viewer/set/style"属 2.10.5 中的步骤 5。

- **命令**
  /vis/viewer/flush
- **功能**
  声明可视化结束并显示图像。
- **附注**
  为了完成可视化，命令"/vis/viewer/flush"应在"/vis/drawVolume", "/vis/specify" 等之后执行 。它属 2.10.5 中的步骤 7。

2.10.7.3 /vis/drawVolume 命令

- **命令**
  /vis/drawVolume [<physical-volume-name>]
- **参数**
  物理体名，缺省是"world", 可忽略。
- **功能**
  建立一个由给定物理体组成的 scene，并请求当前 viewer 绘出相应图像，这个 scene 将成为当前 scene。命令 "/vis/viewer/flush" 应在这个命令之后执行，以声明结束可视化。
- **例:在给定坐标系中可视化整个世界**
  Idle> /vis/drawVolume
  Idle> /vis/scene/add/axes 0 0 0 500 mm
  Idle> /vis/viewer/flush
- **附注**
  命令"/vis/drawVolume"完成 2.10.5 节中的步骤 2、3、4，命令 "/vis/viewer/flush" 应在这个命令之后执行，以声明结束可视化。(步骤 7)

2.10.7.4 `/vis/specify` 命令

- **命令**
  `/vis/specify [logical-volume-name]`
- **参数**
  逻辑体名
- **功能**
  建立一个由给定逻辑体组成的 scene，并请求当前 viewer 绘出相应图像，这个 scene 将成为当前 scene。
- **例(在给定坐标系中可视化一个选定的逻辑体)：**

```
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
```

- **附注**
  命令`/vis/specify` 完成 2.10.5 节中的步骤 2、3、4 和 6，命令 `"/vis/viewer/flush"` 应在这个命令之后执行，以声明结束可视化。(步骤 7)

2.10.7.5 用于事件可视化的命令

- **命令**
  `/vis/scene/add/trajectories`
- **功能**
  这个命令添加径迹到当前 scene。这些添加到 scene 中径迹在事件结束的时候被显示。
- **例:径迹可视化**

```
Idle> /tracking/storeTrajectory 1
Idle> /vis/scene/add/trajectories
Idle> /run/beamOn 10
```

- **附注 1**
  在例子 examples/novice/N03 中，命令`"/vis/scene/add/trajectories"` 不需要通过 (G)UI 来执行，因为在事件的 action 中已经显式的声明了方法 DrawTrajectory()。
- **附注 2**
  为了让使用`/run/beamOn` 可视化的命令工作，run action)和事件 action 应被适当的实现。在例子 examples/novice/N03 中的实现如下：

```
void ExN03RunAction::BeginOfRunAction(const G4Run* aRun)
{
  .....

  if (G4VVisManager::GetConcreteInstance())
  {
    G4UImanager* UI = G4UImanager::GetUIpointer();
    UI->ApplyCommand("/vis/scene/notifyHandlers");
```

```
  }
}

void ExN03RunAction::EndOfRunAction(const G4Run* )
{
  if (G4VVisManager::GetConcreteInstance()) {
     G4UImanager::GetUIpointer()->ApplyCommand("/vis/viewer/update");
  }
}
```

2.10.8 探测器几何体的树型结构可视化

Geant4 支持"tree drivers"，它用于可视化一个探测器几何的树型结构。目前，已实现了两个树型结构引擎

- ASCII tree driver
- GAG tree driver

ASCII tree driver 通过显示物理体名的适当标识，来可视化一个探测器几何的树型结构。GAG tree driver 使用 GAG GUI (http://erpc1.naruto-u.ac.jp/~geant4)完成了相同的功能。稍后将要实现的 XML tree driver，它使用 XML 来显示树型结构。

注意，用户必须要使用这些 tree driver，必须在自己的 Visualization Manager 类中进行注册：

```
RegisterGraphicsSystem (new G4ASCIITree);
RegisterGraphicsSystem (new G4GAGTree  );
```

参看例子 `examples/novice/N03/src/ExN03VisManager.cc`。

2.10.8.1 `/vis/drawTree` 命令

- **命令**
  `/vis/drawTree [<physical_volume_name>] [<driver_name>]`
  候选的 driver 名是"ATree" (ASCII tree，缺省)和 "GAGTree"。
- **功能**
  可视化一个探测器几何树。
- **参数**
  位于树顶层的物理体名，和 tree-driver 名。
- **例：可视化整个几何**

```
Idle> /vis/drawTree ! ATree
.....
"Calorimeter", copy no. 0
```

- "Layer", copy no. -1 (10 replicas)
- "Absorber", copy no. 0
- "Gap", copy no. 0
- .....
- **附注**
  上例中的字符'!'表示“缺省参数”。这是 Geant4 交互式命令的约定。

2.10.8.2 `/vis/XXXTree/verbose` 命令

- **命令**
  `/vis/XXXTree/verbose [<verbosity>]`

  "XXX" 对应于 ASCII tree driver 是 `"ASCIITree"` ，对应于 GAG tree driver 是`"GAG"` 。verbosity 的值为 0（缺省）到 10：
-
- `< 10:` - 不打印重复逻辑体的子几何体。
- - 不重复打印复制的几何体。
- `>= 10:` 打印所有物理体。
- `>= 0:` 打印物理体名。
- `>= 1:` 打印逻辑体名。
- `>= 2:` 打印实体名和类型。
- **功能**
  指定可视化树型结构的详细程度。
- **例:**
-
- Idle> /vis/ASCIITree/verbose 1
- Idle> /vis/drawTree ! ATree
- .....
- "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"
- .....

---

3. 工具包基本组成

About the authors

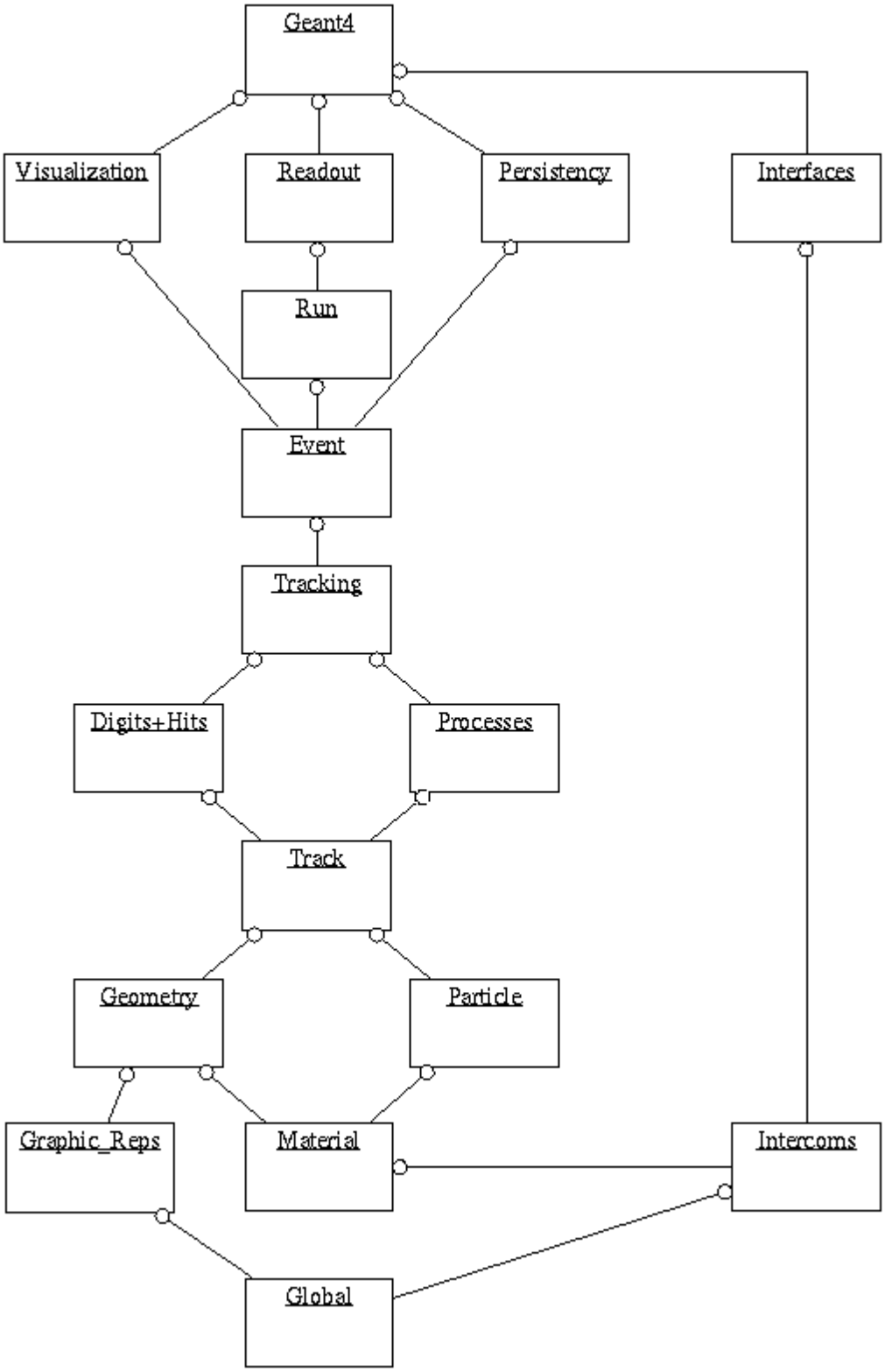
Overview Contents Previous Next

3.1 G4 的各个功能模块和它们的功能

3.1.1 什么是功能模块?

在设计一个像 Geant4 这样的大型软件系统时，有必要把它分为一些小的逻辑单元，这将使整个设计过程容易组织。定义不同的逻辑单元之间尽量相互独立（松耦合），使它们可以并行的开发而没有过多的相互影响。

Grady Booch 在《Object-Oriented Analysis and Design with Applications 》[1]提到，不同模块用于建立不同的逻辑单元，它的定义是"它是一组相互紧耦合的类，但是与其他组之间的关系是松耦合的。"　这意味着一个模块的类包含了那些相互之间有着紧密关系的类，它们与属于其他模块的类之间的关系是弱的，只有一些有限数量的类之间有“使用”关系。功能模块与它们之间的关系用一个模块关系图来表示。下面是 Geant4 中不同功能模块之间的关系图，每个框，表示一个模块，直线表示使用关系，在直线终端的圆，表示该功能模块使用了其他模块中的类。



Geant4 代码文件的组织关系，基本上是遵循模块之间的关系结构。本 *User's Manual* 也是按照这种结构组织的。

在 Geant4 工具包的开发维护过程中，一个软件小组将接受一个模块的所有类的开发和维护任务。

3.1.2 在 Genat4 中的功能模块

下面概述了 Geant4 中各个模块的角色。

1. **Run 和事件（run、event）**

   这些模块的类与事件产生、事件发生器接口，和次级粒子产生有关，它们原则上向粒子跟踪管理类提供被跟踪的粒子。

2. **粒子跟踪和径迹** （tracking、track）

   这些模块的类与粒子的输运过程有关，这是 Geant4 的一个重要部分。它使得 Geant4 应用程序的物理过程可以模拟各种行为，在粒子跟踪的每一步，记录粒子的空间位置，或者时间，或者随空间和时间的分布（和所有这些可能的组合）。

3. **几何体，磁场，和 CAD 接口** （geometry）

   这三个模块的任务是管理探测器的几何定义（实体建模及与 CAD 系统的交互）和实体之间的距离（包括在磁场中）。Geant4 的几何实体建模是基于 ISO STEP [7]标准的，并且与它完全兼容，这保证了 Geant4 能够和 CAD 系统交换几何信息。Geant4 几何的一个关键特性是独立与实体描述，通过那些 G4 实体的抽象接口，使得不同的几何描述，对于粒子跟踪部分来说都是一致的。对于在场中的输运过程，Geant4 已经指定了模拟过程的精度。但是，面向对象的设计使得用户可以替换不同的算法和不同的场，而不影响工具包中的其他部分。

4. **粒子和介质的定义** （particle、material）

   有两个模块是用来管理粒子和材料的定义的。

5. **物理（physics）**

   这个模块管理所有与介质发生作用的物理过程。物理接口的抽象接口允许对每个反应通道和每个作用过程实现多个物理模型，这些物理模型可以通过能量范围、粒子类型、介质材料等，来进行选择。面向对象的数据封装和多态的特性使得截面数据的存取过程是透明的（与截面数据的数据源无关）。正是这些特性，使得用户可以使用同样的方式来处理电磁作用过程和强作用过程。

6. **Hits 和数字化（hits 、digitization）**

   有两个模块用来管理 hits 的构建，以及它在数字化部分的使用。Hits 和数字化的基本设计和实现已经完成，并且在 α 版发行前就提供了几个原型和测试情况。在灵敏感探测器的几何体中， hits collections 表示探测器的逻辑输出。我们已经测试并发布了创建和管理 hits collections 的不同方法,特别是 single hits 和量热器 hits 这两种类型。在任何情况下，已经可以成功地将 hits collections 保存到对像数据管理系统（Object Data Base Management System），也可以从里面读取相应数据。

7. **可视化（Visualization ）**

   这是用来管理实体、径迹和 hits 的可视化，以及与底层图形库的交互。那些基本的、频繁使用的图形功能已经在 α 版的时候实现。面向对象的设计允许我们开发独立的不同的可视化引擎，例如基于 OpenGL 或者 OpenInventor (对于 X11 与 Windows 环境), DAWN, Postscript (使用 DAWN)或者 VRML 的不同引擎。

8. **接口（Interfaces）**

   这个模块处理 GUI 的输出和与外部软件(OODBMS 等)的交互。

---

[1] Grady Booch, Object-Oriented Analysis and Design with Applications The Benjamin/Cummings Publishing Co. Inc, 1994, ISBN 0-8053-5340-2

About the authors

Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Toolkit Fundamentals

3.2 全局类

在"global"这个模块中，包含了那些在 Geant4 工具包中最通用的类，数据类型，结构，和常量，同时，它还定义了与第三方软件库的接口，及与系统相关的类型的转换。

3.2.1 Geant4 类的命名

为了使用统一的命名方式，遵循 Geant4 的一些约定，Geant4 内核的每一个类的名字都以前缀 G4 开头。例如， *G4VHit, G4GeometryManager, G4ProcessVector,* 等。为了便于移植，在 Geant4 中使用 G4 类型（*G4int, G4float, G4double*，等）代替 C 中原有的类型(`int, float, double,` 等)。因为这些基本类型，在不同的编译器和平台中有不同的定义， G4 类型实现了不同给定环境的通用类型。

基本类型

在 Geant4 中的基本类型：

G4int, G4long, G4float, G4double, G4bool, G4complex 和 *G4String*

通常由简单的 `typedef` 和各自在 **CLHEP**, **STL** 或者系统库中的类型定义组成。多数基本类型的定义包含了头文件 `globals.hh`，这个头文件中包含了那些系统头文件和在 Geant4 内核中需要并使用了的全局实用函数。

CLHEP 类在 Geant4 中的类型定义及它们的使用方法

下面的这些类是基于 **CLHEP (Computing Library for High Energy Physics)** 中的相应类定义的。有关的详细文档请参阅《**CLHEP reference guide**》[1] 和《**CLHEP user manual**》[2]。

- *G4ThreeVector, G4RotationMatrix, G4LorentzVector* 和 *G4LorentzRotation*

  向量类：定义了 3 元向量，和它的 3×3 旋转变换矩阵，4 元向量，和对应的 4×4 旋转变换矩阵。

- *G4Plane3D, G4Transform3D, G4Normal3D, G4Point3D*, 和 *G4Vector3D*

  几何类：定义了几何实体和它们在 3D 空间中的变换。

3.2.2 CLHEP 中的 *HEPRandom* 模块

HEPRandom 模块原本就是 Geant4 内核的一个部分，目前作为 **CLHEP** 的一个模块发行。最初的 **CLHEP's** *HepRandom* 模块和 **Math.h++** 中的 **Rogue Wave**【Rouge Wave 是 Rouge Wave 公司提供的 STL】的基础上进行开发的。有关 *HEPRandom* 类的详细情况，请参看《**CLHEP Reference Guide**》[1] 或《**CLHEP User Manual**》[2]。

本手册中所提到的有关信息是从与 *HEPRandom* 一起发行的原始文档[3] 中摘录的。

HEPRandom 模块由实现不同随机引擎和不同随机分布的类组成的。一个随机分布和一个随机引擎组成一个随机数发生器。一个随机分布类可以通过不同的算法，和不同的调用顺序，来定义不同的分布参数和区间。其中一个随机引擎实现了伪随机数发生器的基本算法。

产生随机数的 3 种不同方式：

1. 使用在 *HepRandom* 类中定义的静态随机数发生器：随机数是由不同随机分布类中定义的静态方法 `shoot()` 产生的。静态随机数发生器将使用一个 *HepJamesRandom* 对象作为缺省引擎，用户可以使用在 *HepRandom* 类中定义的静态方法来实例化一个新的引擎，从而设置或者改变随机数发射器的属性。
2. 指定一个随机引擎：使用不同随机分布的类中定义的静态方法 `shoot(*HepRandomEngine)` 产生随机数。用户必须向 shoot 方法传递一个随机引擎的对象。通过调用指定随机引擎的 `flat()` 方法可以旁路相应的随机数发生机制。用户必须注意他所实例化的随机引擎。
3. 实例化一个随机分布：随机数由不同随机分布类中定义的 `fire()` 方法 methods (非静态方法)产生。用户必须通过指针或者应用向随机数发生器的构造函数传递一个随机分布对象，使随机分布与随机引擎相关联。可以使用相应随机引擎的 `flat()` 方法来旁路特定的随机数发生机制。

由于在 Geant4 工具包中只使用了 *HEPRandom* 的静态接口，所以，在本手册中，我们只关心静态发生器。

HEPRandom 随机引擎

HepRandomEngine 类是一个抽象类，它为每个随机引擎定义了一个接口。它实现了 `getSeed()` 和 `getSeeds()` 方法，这两个方法分别返回“初始随机种子”和“初始随机种子数组”(如果存在)。用户可以从 *HepRandomEngine* 派生具体的随机引擎，并添加到 *HepRandom* 中。目前，在 *HepRandom* 中，已经实现了几个不同的随机引擎 。下面是其中的 5 个：

- *HepJamesRandom*

  它实现了在``F.James, Comp. Phys. Comm. 60 (1990) 329"中描述的伪随机数发生的算法。这是静态随机数发生器的缺省随机引擎；如果用户不设置其他的引擎，它将被每个随机分布类所调用。

- *DRand48Engine*

  使用 C 标准库中的系统函数 `drand48()` 和 `srand48()`来实现基本的 `flat()` 分布和设置随机种子。*DRand48Engine* 使用 C 标准库中的系统函数 `seed48()` 来获得随机数发生器的内部状态（3 个 short 型值）， *DRand48Engine* 是在 *HEPRandom* 中定义的唯一的随机引擎，它实际工作在 32 位的精度。这个类型的对象复制是不允许的。

- *RandEngine*

  它是一个简单的随机引擎，使用 C 标准库中的 `rand()` 和 `srand()` 来实现基本分布 `flat()` 和设置随机种子。请注意 `rand()`产生的随机数的谱分布特性，如果要产生一个 高质量的、或者是长周期的随机数，我们不推荐使用这个随机引擎。这个类型的对象复制是不允许的。

- *RanluxEngine*

  RanluxEngine 的算法是从最初在 **MATHLIB HEP** 中，由 Fred James 用 FORTRAN77 实现的版本移植过来的。它通过使用一个乘同余法的随机数发生器进行初始化，该发生器使用了在``F.James, Comp. Phys. Comm. 60 (1990) 329-344"中提到的 L'Ecuyer 常数。这个随机引擎提供了 5 个不同级别质量的随机数，在实例化 *RanluxEngine* 的时候，用户向构造函数可以指定质量级别（缺省是 3）。例如：

  ```
  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
  ...
  ```

  这个类提供了一个 `getLuxury()` 方法来查询随机引擎的质量级别。

  `SetSeed()` 和 `SetSeeds()` 方法用于设置随机种子，它们也可以用来指定质量级别。例如：

```
    // 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*

  RanecuEngine 的算法使用了在 **MATHLIB HEP** 库中用 FORTRAN77 编写的版本。它通过使用一个乘同余法的随机数发生器进行初始化，该发生器使用了在``F.James, Comp. Phys. Comm. 60 (1990) 329-344"中提到的 L'Ecuyer 常数。这个引擎的随机种子处理与同在 *HEPRandom* 中的其他引擎略有不同，它通过一个给定索引号从一个随机种子表中取得随机种子，它的 `getSeed()` 方法同样返回的是随机种子的当前索引号。通过给定当前 `SeedTable` 的索引号，使用 `setSeeds()`方法设置随机种子（如果索引号超过随机种子表的长度，将使用 `[index%size]` 的余数)。例如：

```
// 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.
...
```

  `setSeed()` 方法复位随机引擎的当前状态，使用 *HepRandom* 的静态随机种子表中给定位置的随机种子。

除了 *RanecuEngine* 的内部状态是由一对 long 型值外，其余引擎都有一些更加复杂的状态。目前，这些状态只能通过方法 `saveStatus()`, `restoreStatus()` 和 `showStatus()`来实现，它们可以从 *HepRandom* 中静态的调用。如果用户需要在一个给定的模拟状态下，重新产生一个 run 或在 run 中的一个事件， 那么这些随机数发生器的内部状态是必须的。

RanecuEngine 可能是用于实现这种操作的最佳引擎，它的内部状态可以通过方法 `getSeeds()`/`setSeeds()` (对与在 *HepRandom* 中静态接口是 `getTheSeeds()`/`setTheSeeds()`) 非常容易的存取/复位。

HepRandom 类中的静态接口

HepRandom 是一个 singleton 类【Singleton 类，有的书上译作单实例类，其实例称为单例，指该类只能有一个实例，且在类声明的时候就自动初始化了】，它使用 *HepJamesRandom* 引擎作为伪随机数发生的缺省算法。*HepRandom* 定义了一个私有静态数据成员，`theGenerator` 和一系列用于操作该成员的静态方法。通过 `theGenerator`,用户可以更改当前随机引擎算法，存取随机种子，使用各种定义的随机分布。静态方法 `setTheSeed()` 和 `getTheSeed()` 分别用于设置和获取静态随机数发生器所使用的随机引擎的初始化种子。例如：

```
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'
```

用户在同一时刻只能使用一个随机引擎，但可以在任何时刻进行更改所使用的随机引擎，或者定义一个新的随机引擎（如果尚未定义）并设置它为当前使用的随机引擎。例如：

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

或者简单的将一个已经经过初始化的引擎(该引擎的运行状态已被保存，新的随机数序列将从原先中断的地方开始)设为当前随机引擎。例如：

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

其它在该类中定义的静态方法如下：

- `void setTheSeeds(const G4long* seeds, G4int)`
- `const G4long* getTheSeeds()`

  设置/获取随机数发生器的随机种子数组，对于 *RanecuEngine* 引擎来说，它们同时还设置/获取该随机引擎的当前状态。

- `HepRandomEngine* getTheEngine()`

  获取指向当前使用的随机引擎的指针。

***HEPRandom* 随机分布**

一个随机分布类包括了不同的随机算法，和每个算法的不同调用顺序，用来定义不同的随机分布参数和取值区间；对应不同的分布，它还包含那些用于填充指定长度的随机数组的不同方法。在 *HEPRandom* 中定义了一系列的随机分布类，如下：

- *RandFlat*

  用于产生指定区间内均匀分布的随机数（整型或双精度），该类同时提供了用于产生随机位序列（bits）的方法。

- *RandExponential*

  用于产生给定平均值，指数分布的随机数（缺省平均值为 1）。

- *RandGauss*

  用于产生给定平均值（缺省为 0），或者指定偏差（缺省为 1），高斯（Gauss）分布的随机数。因为每次产生两个高斯分布的随机数，所以，下一次返回的值同样也是本次产生的。

- *RandBreitWigner*

  用于产生符合 Breit-Wigner 分布算法的随机数 (均匀分布 或者 mean^2)。

- *RandPoisson*

  用于产生给定平均值（缺省为 1），符合泊松（Poisson）分布的随机数(算法取自 ``W.H.Press et al., Numerical Recipes in C, Second Edition").

3.2.3 *HEPNumerics* 模块

在 Geant4 中已经实现了一系列的数值算法类。这些算法和方法主要是居于以下书中所推荐的方法实现的：

- 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.

这些类包括：

- *G4ChebyshevApproximation*

  该类为数据成员 fFunction 所指的函数创建了一个车比雪夫(Chebyshev)近似多项式，这个多项式提供了一个评价最小极大值多项式的有效方法。最小极大值多项式是指在所有同阶多项式中，它与原函数相比，有最小的最大偏差。

- *G4DataInterpolation*

  该类提供了数据内插和外推的方法：多项式， 三次样条，…..

- *G4GaussChebyshevQ*
- *G4GaussHermiteQ*
- *G4GaussJacobiQ*
- *G4GaussLaguerreQ*

  这些类实现了高斯－车比雪夫，高斯－埃尔米特（Hermite），高斯－雅各比（Jacobi），高斯－拉格朗日（Laguerre），高斯－勒让德（Legendre）求积分方法。正交多项式的根与对应的权是通过迭代算法（二分法）计算的。

- *G4Integrator*

  该类是一个模板类，包括了一些通用函数(勒让德，幸普森（Simpson），Adaptive Gauss，勒让德， Hermite，雅各比)的积分方法。

- *G4SimpleIntegration*

  该类实现了简单的数值积分方法(梯形、中值、高斯、幸普森、Adaptive Gauss)。

3.2.4 通用管理类

`global' 类同时定义了一系列的实用类，它们用于 Geant4 内核。这些类包括：

- *G4Allocator*

  该类通过页机制快速的为对象从堆中申请空间。它有两个方法 `MallocSingle()`和 `FreeSingle()`分别对应于 `new` 和 `delete` 操作符。

- *G4ReferenceCountedHandle*

该类是一个模板类，它就像是一个用于计数的指针，记录对象在生命周期内被应用的次数。

- *G4FastVector*

  该类是一个模板类，定义了一个指针向量，且并不进行边界检查。

- *G4PhysicsVector*

  该类定义了一个物理向量，它包括在给定能量、动量等范围条件下，一个粒子在物质中的能量损失，反应截面和其他物理量。它作为其他物理向量类的基类，如， 'log' (*G4PhysicsLogVector*) 'linear' (*G4PhysicsLinearVector*), 'free' (*G4PhysicsFreeVector*), 等。

- *G4LPhysicsFreeVector*

  该类实现了用于低能物理反应截面数据的自由向量类，提供了用于查找 能量|动量窗 的方法。

- *G4PhysicsOrderedFreeVector*

  该类是从 *G4PhysicsVector* 派生的，实现了物理有序的自由向量类，同时提供了顺序插入 能量/数值 对的方法，并且也提供了从该向量中取得最大、最小能量和数值的方法。

- *G4Timer*

  该实用类提供了用于测量 用户/系统 进程所消耗时间的方法。
  使用了 `<sys/times.h>` 和 `<unistd.h>` - POSIX.1.规范

- *G4UserLimits*

  该类提供了各种方法，用于在 Geant4 中获取和设置各种限制条件。

- *G4UnitsTable*

  该类实现了在 Geant4 中所使用的单位系统的占位符。

[1] cern.ch/clhep/manual/RefGuide.
[2] cern.ch/clhep/manual/UserGuide.
[3] cern.ch/wwwasd/geant/geant4_public/Random.html.

About the authors

3.3 单位系统

3.3.1 基本单位

Geant4 向用户提供了使用他们所喜欢的单位的可能性。事实上，Geant4 内核在内部使用一个统一的单位集合，它是基于 `HepSystemOfUnits` 的：

```
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)
```

所有其他的单位都是有这些基本单位导出的。

例如：

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

在文件 `source/global/management/include/SystemOfUnits.h` 中，可以找到这些定义，这个文件是 CLHEP 的一部分。

此外，用户可以自由的改变内核所使用的单位系统。

3.3.2 输入数据

避免使用 'hard coded' 数据

你**必须**对你所引入的数据指定单位：

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

事实上，所有的 Geant4 代码都是遵循这些规范，这使得代码与用户选择的单位系统无关。

如果单位没有指定，这些数据将被认为使用 G4 系统内部的隐含单位，我们反对这样的做法，用户在写代码的时候必须清楚 G4 系统的隐含单位，这使得代码的移植性变差。

如果数据集来自于一个数组或外部文件，我们强烈推荐在读取数据的时候设置它们的单位。例如：

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

交互式命令

一些用户接口的内建命令也要求指定单位。例如：

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

如果不指定这些单位，或者指定的单位无效，那么系统将拒绝执行这些命令。

3.3.3 输出数据

用户可以使用他喜欢的单位输出数据，只要将这些数据除以相应的单位：

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

当然， `G4cout << KineticEnergy` 将输出使用系统内部单位的能量值。

用户也可以让 Geant4 来选择使用最合适的单位来输出数据，只要指定用户数据的类别（属于 Length, Time, Energy, 等）。例如

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

`StepSize` 将根据实际的数据输出相应的单位， km, m, mm, fermi, 等。

3.3.4 引入新的单位

如果用户希望引入新的单位，有两个方法：

- 在头文件 `SystemOfUnits.h` 中添加新的单位

```
#include "SystemOfUnits.h"

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

使用这个方法定义复合单位有些困难，最好使用下面的方法：

- 用户可以实例化一个 *G4UnitDefinition* 类的对象

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

例：定义一些速度单位

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

在 *G4UnitsTable* 中， "Speed" 类的单位缺省不存在，但是它将自动被建立。

G4UnitDefinition 类位于 `source/global/management` 目录下。

3.3.5 输出单位列表

用户可以使用静态函数 `G4UnitDefinition::PrintUnitsTable()`输出单位列表，或者使用交互式命令，`/units/list` 。

3.4 Run

3.4.1 *Run* 的基本概念

在 Geant4 中，*Run* 是一个最大的模拟单位，一个 run 由一系列的事件组成，它是 *G4Run* 的一个对象，由 *G4RunManager* 的方法 `beamOn()`启动。在一个 run 过程中，探测器几何，灵敏探测器设置，和模拟中所使用的物理过程，都保持不变。

与 run 有关的表述

G4Run 代表一个 run，它有一个 run 的标识数和在这 run 中所要模拟的事件数。这个标识数可以由用户任意设置，因为在 Geant4 内核中并未用到这个标识。

G4Run 有两个指向 *G4VHitsCollection* 和 *G4VDigiCollection* 表的指针。这两个表，分别与 *sensitive detectors* 和 *digitizer modules* 模拟相关，它们的使用将在 4.4 和 4.5 中讨论。

管理 run 的进程

G4RunManager 管理一个 run 的相关进程。在 *G4RunManager* 的构造函数中，除一些静态管理类外，其余所有的管理类都被构造，这些类将被 *G4RunManager* 的析构函数删除。*G4RunManager* 必须是一个 singleton 对象,用户可以通过静态方法 `getRunManager()` 获取指向这个 singleton 对象的指针。

正如在 2.1 节中提到的，通过使用方法 `SetUserInitialization()`和 `SetUserAction()`，将用户定义的用户初始化类和用户行为类在 Geant4 内核初始化前传递给 *G4RunManager* 。Geant4 内核定义的所有用户类将在第 6 章中概述。

G4RunManager 有几个公用方法：

`initialize()`

Geant4 内核必须的所有初始化都是由这个方法触发的。这些初始化如下：

- 探测器几何、灵敏探测器或数字化模块设置的构成 ，
- 粒子和物理过程的构成，
- 反应截面表的计算。

这个方法必须在第一个 run 执行之前调用。对于第二个 run 和以后的 run，如果某些初始化参数需要改变，那么，它将自动被调用。

`beamOn(G4int numberOfEvent)`

这个方法触发一个实际的模拟过程，即一个事件循环。它有一个整型参数，表示被模拟的事件数。

`getRunManager()`

这个静态方法返回指向 *G4RunManager* singleton 对象的指针。

`getCurrentEvent()`

这个方法返回指向当前模拟的 *G4Event* 对象的指针，它只有在事件模拟过程中是可用的，在这个时候，Geant4 应用程序的状态是*"EventProc"*。如果 Geant4 的状态不是*"EventProc"*，这个方法将返回一个空指针。注意返回值的类型是 `const G4Event *`，因此用户不能更改对象的内容。有关 Geant4 应用程序的状态将在下一小节讨论。

`setNumberOfEventsToBeStored(G4int nPrevious)`

对于模拟多个事件“堆积”的情况，实际就是在同一时刻同时存取多个事件。通过调用这个方法， *G4RunManager* 将保存 `nPrevious` 个 `G4Event` 对象。这个方法必须在 `beamOn()`执行之前调用。

`getPreviousEvent(G4int i_thPrevious)`

通过这个方法，用户可以获取指向 `i_thPrevious G4Event` 对象的指针，返回值是一个指向 const 型对象的指针。在获取 `i_thPrevious` 事件指针之前，必须有在同一次 run 中的 `i_thPrevious` 个事件已经被模拟，否则，将返回空指针。

`abortRun()`

这个方法可以在需要中止一个 run 模拟过程的时候调用，它将安全的中止模拟过程，即使是在一个事件模拟的中间过程。但是，用户需要注意，这个被中止的最后一个事件模拟是不完全的，不可以用来做更多的分析。这个方法在 *GeomClosed* 和 *EventProc* 这两个状态下是可用的。

G4UserRunAction

G4UserRunAction 是一个用户行为类，从它可以派生用户自己的类。这个基类，有两个虚拟方法：

`beginOfRunAction()`

这个方法在 `beamOn()` 方法开始的时候被调用。可假定的使用情况如下：

- 设置一个 run 标识数，
- 登记一个统计图，
- 设置灵敏探测器或数字化模块(e.g., dead channel)的指定运行条件。

`endOfRunAction()` 方法

这个方法在 `beamOn()` 结束的时候调用。典型的使用如下：

- 存储/输出 统计图，
- 操作 run 摘要。

3.4.2 Geant4 状态机

Geant4 被设计为一个状态机，某些方法仅在某个状态是可用的。*G4RunManager* 控制 Geant4 应用程序的状态改变，Geant4 的状态是用枚举量 *G4ApplicationState* 表示的。在一个 Geant4 应用程序的整个生命周期中，一个有 6 个状态。

G4State_PreInit state

这是一个 Geant4 应用程序的开始状态。当应用程序在这种状态是，表明它需要初始化。应用程序在一个 run 之后，改变几何，物理过程，或者截断，也会使程序返回这个状态。

G4State_Init state

在 *G4RunManager* 的方法 `initialize()`被调用的时候，应用程序进入这个状态。在这个状态下，在用户初始化类中定义的方法被调用。

G4State_Idle state

这个状态表明应用程序准备开始模拟。

G4State_GeomClosed state

当 `beamOn()` 被调用之后，应用程序进入到这个状态开始模拟。在这个状态下，用户不可以更改几何，物理过程，截断设置。

G4State_EventProc state

当应用程序处理一个事件的时候，进入这个状态。*G4RunManager* 的 `getCurrentEvent()` 和 `getPreviousEvent()` 方法只有在这个状态下可用。

G4State_Quit state

当 *G4RunManager* 的析构函数被调用的时候，应用程序进入到这个“终结”状态。Geant4 内核的 run Manager 正被删除，应用程序将不能回到任何其他状态。

G4State_Abort state

当 *G4Exception* 发生的时候，应用程序进入到这个“终结”状态并引起一个内核丢弃。用户可以通过实现具体的 *G4VStateDependent* 类，仍然可以利用一个 hook 进行一些“安全”的操作，例如，存储统计图。用户也可以通过交互式命令*/control/suppressAbortion* 选择限制 *G4Exception* 的发生。当例外被限制的时候，用户仍会受到 G4Exception 发出的错误信息，并且不保证在接到这些 G4Exception 错误信息之后的结果是正确的。

G4StateManager 属于 *intercoms* 类。

3.4.3 用于状态改变时的用户 hook

如果用户希望在 Geant4 状态改变时执行一些期望的指令，可以创建一个由 *G4VStateDependent* 派生的类。例如，用户可以在 G4Exception 发生的时候，或者在 Geant4 进入 *Abort* 状态的时候，在内核数据丢弃之前保存用户需要的统计数据。

下面是一个在 Geant4 进入 *Abort* 状态时用户保存统计数据的代码实例。这个类的对象应由用户创建，可以在 *main()*中。这个对象将在它创建的时候自动向 *G4StateManager* 注册。

```
#ifndef UserHookForAbortState_H
#define UserHookForAbortState_H 1

#include "G4VStateDependent.hh"

class UserHookForAbortState : public G4VStateDependent
{
```

```
 public:
  UserHookForAbortState();   // constructor
  ~UserHookForAbortState();  // destructor

  virtual G4bool Notify(G4ApplicationState requiredState);
};

#endif
```

代码清单 3.4.3.1
UserHookForAbortState 的头文件

```
#include "UserHookForAbortState.hh"

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

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

  // Do book keeping here

  return true;
}
```

代码清单 3.4.3.2
UserHookForAbortState 的源文件

3.4.4 定制 run Manager

在 Run Manager 的虚拟方法

G4RunManager 是一个实类，是用于管理 G4 内核的各种功能的完整集合。它是 G4 内核中唯一的管理类，必须在用户应用程序的 main()函数中创建(构造)。用户也可以构造自己的 RunManager ，来代替 G4 提供的 G4RunManager 。但是，用户的 RunManager 必需继承 G4RunManager 。G4RunManager 提供了各种用于管理 G4 内核的虚拟方法，用户在定制的 RunManager 中只需要根据自己的需要重载这些方法就可以了，在基类 G4RunManager 中的其他方法同样可以继续使用。下面是这些虚拟方法：

```
public: virtual void initialize();
```

G4 内核初始化的主入口

```
protected: virtual void initializeGeometry();
```

构造几何

```
protected: virtual void initializePhysics();
```

构造物理过程

```
protected: virtual void initializeCutOff();
```

设置反应产物截断值

```
public: virtual void beamOn(G4int n_event);
```

事件循环主入口

```
protected: virtual G4bool confirmBeamOnCondition();
```

检查事件循环的内核状态

```
protected: virtual void runInitialization();
```

初始化一次 run

```
protected: virtual void BuildPhysicsTables();
```

更新区域信息并调用每个物理过程 建立/更新 物理表(physics tables)

```
protected: virtual void doEventLoop(G4int n_events);
```

处理事件循环

```
protected: virtual G4Event* generateEvent(G4int i_event);
```

生成 *G4Event* 对象

```
protected: virtual void analyzeEvent(G4Event* anEvent);
```

事件的保存分析

```
protected: virtual void runTermination();
```

中止 run

```
public: virtual void defineWorldVolume(G4VPhysicalVolume * worldVol);
```

为 *G4Navigato* 设置 world（世界）几何体 *r*

```
public: virtual void abortRun();
```

放弃运行

定制事件循环

在 `G4RunManager` 中，事件循环是由虚方法 `doEventLoop()`进行处理的。这个方法由一个包括以下一些步骤的 for 循环实现：

1. 创建一个 `G4Event` 对象，并且指定初级粒子顶点和初级粒子。这些工作由虚方法 `generatePrimaryEvent()` 完成。
2. 向 `G4EventManager` 发送一个 `G4Event` 对象，使 *Hits* 和 *trajectories* 与这个 `G4Event` 对象相关联。
3. 针对当前 `G4Event` 对象，记录用户期望的数据。这将由虚方法 `analyzeEvent()` 来完成。

`doEventLoop()` 完成一个事件的整个模拟。然而，常常将上述的三个步骤分开放到相互独立的程序中。例如，如果用户需要检查改变被模拟事件的甄别阈、ADC 门宽和触发条件后的变化，那么，将上述的前两步放在一个程序中，第 3 步放在另一个程序中，将会节约大量的机器时间。前一个程序只需要产生并保存 hit/trajectory 信息，第二个程序读取保存的 `G4Event` 对象并用上述提到的阈值、门宽和触发条件进行数字化（分析），然后改变这些条件并再次进行数字化，而不需要再次运行第一个程序用于产生 `G4Events`。

改变探测器几何

探测器几何定义在用户的 *G4VUserDetectorConstruction* 类中，它可以在两个 run 之间进行更改。有两种不同的情况。

第一中情况是，用户希望删除整个旧的几何结构并建立一个全新的结构。在这种情况下，用户需要向 *RunManager* 传递新的 world 几何体指针。因此，用户应该按如下的方法进行。

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

这种情况是相当少的，对于用户来说，第二种情况更常见。

第二种情况是这样的。假定用户需要移动、旋转探测器中的一个部分，这种情况常发生在束流测试试验中。很明显，用户不需要改变 world 几何体，更切确的说，用户的 world 几何体（对于束流试验中是试验大厅）足够大，可以在里面移动、旋转用户的探测器。对于这种情况，用户可以继续使用所有的探测器几何设置，只需要用相应物理体的 Set 方法更新一下相应的变换向量就可以了。不必重新向 *RunManager* 传递新的 world 几何体指针。

如果用户需要为每一个 run 改变几何设置，可以在 *G4UserRunAction* 类的方法 `beginOfRunAction()` 中实现，它将在每次 run 开始前被调用；也可以派生 `runInitialization()` 方法。请注意，上述的两种情况，用户需要让 *RunManager* 知道"the geometry needs to be closed again"。因此，在下一次 run 以前，用户需要调用

```
runManager->geometryHasBeenModified();
```

在 Geant4 Training kit #2 的 Geant4 tutorial 中给出了一个改变几何设置的实例。

切换（打开/关闭）物理过程

在方法 `initializePhysics()` 中，通过调用 `G4VUserPhysicsList::Construct` 定义用户应用程序中使用的粒子和物理过程。由于 G4 中的粒子是静态对象(详情参看 2.4 和 5.3)，在程序执行过程中，用户不能增加或减少粒子。另外，要在程序执行过程中增加或减少物理过程也是非常困难的，因为注册过程非常复杂，初学者不宜使用(参看 2.5 和 5.2)。这就是为什么假定在 G4 内核初始化过程中方法 `initializePhysics()`立即被调用。

然而，用户可以在两次 run 之间，打开或者关闭在 *G4VUserPhysicsList* 类中定义的物理过程，改变物理过程的参数。

用户可以使用 *G4ProcessManager* 中的方法 `ActivateProcess()` 和 `InActivateProcess()` ，在事件循环以外的任何地方，切换一些物理过程。虽然，即使是在 *EventProc* 状态，都不禁止使用这些方法切换物理过程，但是，要在一个事件循环内部切换物理过程，用户必须十分的小心。

有时候，需要在一个 run 内部改变截断值。用户可以在 *Idle* 状态的时候，改变 *G4VUserPhysicsList* 中的 `defaultCutValue` 。在这种情况下，所有的截面数据表必须在事件循环开始前重新计算。用户应该在改变截断值之后，调用方法 `CutOffHasBeenModified()` 来触发 `InitializeCutOff()` ， `InitializeCutOff()`将调用用户 *PhysicsList* 类中的方法 `SetCuts`。

3.5 事件

3.5.1 事件

G4Event 代表一个事件，它是一个包含所有被模拟事件的输入输出信息的类的实例。这个对象在 *G4RunManager* 中被创建并传递给 *G4EventManager*。可以通过 *G4RunManager* 的方法 `getCurrentEvent()`获取当前正在处理的事件（指针）。

3.5.2 事件的结构

一个 *G4Event* 对象有四类主要的信息。在 *G4Event* 中提供了这些信息的相应 Get 方法。

初级事件顶点和初级粒子
: 详情参看 3.6 节。

径迹
: 粒子的径迹被保存在 G4TrajectoryContainer 类的实例中，指向该对象的指针被保存在 *G4Event* 中。径迹所包含的信息在 5.1.6 中有叙述。

Hits collections
: 由探测器(*sensitive detectors*)所产生的 hits 被保存在 *G4HCofThisEvent* 类的实例中，指向它的指针同样被存储在 *G4Event* 中，详情请看 4.4 节。

Digits collections
: 由 *digitizer modules* 产生的 digits 信息被保存在 *G4DCofThisEvent* 类的实例中，指向该对象的指针也存储在 *G4Event* 中，详情参看 4.5 节。

3.5.3 *G4EventManager* 的要求

G4EventManager 是管理事件对象的管理类，主要完成以下任务：

- 将与当前 *G4Event* 事件对象有关的 *G4PrimaryVertex* 和 *G4PrimaryParticle* 对象转换为 *G4Track* 对象。所有代表初级粒子的 *G4Track* 对象被传递给 *G4StackManager*。
- 从 *G4StackManager* 中弹出一个 *G4Track* 对象并传递给 *G4TrackingManager*。如果径迹(track)在被 *G4TrackingManager* 模拟之后，被标记为“killed”，那么该 *G4Track* 对象将被 *G4EventManager* 删除。

- 如果初级 track 被 *G4TrackingManager* "suspended" 或者 "postponed to next event"，那么这个 track 将被重新压入 *G4StackManager*，由 *G4TrackingManager* 返回的次级 *G4Track* 对象也被压入 *G4StackManager*。
- 当 *G4StackManager* 对出栈请求的响应返回 `NULL` 的时候， *G4EventManager* 将中止当前的处理过程。
- 从 *G4UserEventAction* 类中调用用户自定义的方法 `beginOfEventAction()` 和 `endOfEventAction()`，详情参看 6.2 节。

3.5.4. 栈机制

G4StackManager 共有 3 个栈，分别是 *urgent*, *waiting* 和 *postpone-to-next-event*, 它们都是 *G4TrackStack* 类的实例。确省情况下，所有的 *G4Track* 对象被保存在 *urgent* 栈中，以后进先出的方式工作。在这种情况下，其余两个栈没有被使用。但是，通过用户自定义的 *G4UserStackingAction* 类，tracks 可能会被发送到其余两个栈中。

如果 *G4UserStackingAction* 的方法已经被用户重载，那么， *postpone-to-next-event* 栈和 *waiting* 栈中也可能又 tracks 存在。在一个事件开始的时刻，*G4StackManager* 将检查 *postpone-to-next-event* 栈，看是否有前一事件余留下来的 tracks。如果有的话，它会将这些 tracks 转移到 *urgent* 栈中。 当然，首先需要调用 *G4UserStackingAction* 的方法 `PrepareNewEvent()` ，然后，traks 将可能被用户重新分类并将它们压入 *urgent* 或者 *waiting* 栈，或者重新压入 *postpone-to-next-event* ，等待下一事件的处理。在事件被处理的时候，*G4StackManager* 从 *urgent* 栈中逐个弹出所有的 tracks。接着，*G4UserStackingAction* 中的方法 `NewStage()` 将被调用。在这个方法中，在 *waiting* 栈中的 tracks 可能被发送到 *urgent* 栈，也可能继续保留在 *waiting* 栈中，或者被压入到 *postpone-to-next-event* 栈中，延迟到下一个事件。

有关 *G4UserStackingAction* 中的用户自定义方法的详情，以及它们如何影响 track 栈的管理在 6.2 节中有叙述。

3.6 事件发生器接口

3.6.1 初级事件结构

初级粒子顶点和初级粒子

在通过方法 processOneEvent() 将 *G4Event* 对象发送给 *G4EventManager* 的时候，必须保证这个 *G4Event* 对象已拥有一些初级粒子，这些初级粒子是由用户的 *G4VUserPrimaryGeneratorAction* 类发送给 *G4Event* 对象的。

G4PrimaryParticle 类代表 Geant4 开始模拟一个事件的初级粒子。这个类的实例拥有有关粒子类型及它的动量等信息，相关的位置与时间信息被保存在 *G4PrimaryVertex* 对象中，这样做的好处是，一个或多个 *G4PrimaryParticle* 对象可以共享同一个顶点。初级粒子顶点与初级粒子通过一个链表与 *G4Event* 对象相关联。

G4PrimaryParticle 是 *G4VPrimaryGenerator* 类的一个具体类，需要通过指向 *G4ParticleDefinition* 的一个指针，或者一个表示 P.D.G. 粒子代码的整数，来创建它的实例。对于某些假粒子的情况，如：geantino ，光学光子（optical photon）等,或者一些在 P.D.G. 粒子代码中不存在的奇异的核碎片，*G4PrimaryParticle* 对象应该通过 *G4ParticleDefinition* 指针来构造。另外，那些短寿命的基本粒子，如，弱玻色子，或者夸克/胶子，可以使用 P.D.G. 粒子代码将这些粒子作为 *G4PrimaryParticle* 对象实例化。需要注意的是，Geant4 只模拟那些作为 *G4ParticleDefinition* 对象定义的粒子，其余的初级粒子将会被 *G4EventManager* 忽略。但是，如果要记录初级事件的起始点，定义一些中间(“intermediate”)粒子是非常有用的。

强迫衰变通道

G4PrimaryParticle 对象可以有一系列的次级粒子。如果一个粒子是一个中间粒子，它没有对应的 *G4ParticleDefinition*， 那么，这个粒子将被忽略，而它的次级粒子的定点将被假定为与该粒子的顶点相同。例如，一个 Z 玻色子和它的次级粒子，一个正 μ 介子和一个负 μ 介子，这两个 μ 介子将被认为是从 Z 玻色子的那个顶点出发的。

有些粒子有相应的飞行距离，Geant4 模拟的时候应考虑这些，但有的时候，用户还希望跟踪由事件发生器产生的衰变通道。比较典型的情况是 B 介子。甚至是有相应的 *G4ParticleDefinition*,可以产生次级粒子。Geant4 将跟踪这个粒子，直到它发生衰变，根据粒子的类型，判断是否发生多次散射，电离损失，在磁场中偏转。当粒子发生衰变的时候，它将向“预设”的衰变通道发生衰变，而不是用随机选择的办法选择衰变通道。为了转换粒子的动量和能量，次级粒子将根据初级粒子具体情况进行洛伦茨变换。

3.6.2 初级事件发生器接口

G4HEPEvtInterface

不幸的是，几乎目前所有正在使用的事件发生器都是用 FORTRAN 写的。对于 G4 来说，不希望与任何 FORTRAN 程序或者库进行连接，即使 C++本身允许这样做。在某些情况下，与 FORTRAN 程序包链接非常方便，但是，我们将失去很多 C++面向对象的优点，例如，程序的健壮性。因此，G4 只提供了一个 ASCII 文件接口来实现事件发生器。

G4HEPEvtInterface 是 *G4VPrimaryGenerator* 的一个具体类，它可以在用户的 *G4VUserPrimaryGeneratorAction* 相应实类中使用。*G4HEPEvtInterface* 将读取由事件发生器产生的 ASCII 文件，并生成与 *G4PrimaryVertex* 对象相关联的 *G4PrimaryParticle* 对象。*G4HEPEvtInterface* 将保存在公用块/HEPEVT/ 的信息转换为一个面向对象的数据结构。由于

公用块/HEPEVT/几乎被所有用 FORTRAN 写的事件发生器所使用，所以，*G4HEPEvtInterface* 几乎可以与所有目前在 HEP 领域所使用的事件发生器进行接口。*G4HEPEvtInterface* 的构造函数使用文件名作为参数。代码清单 3.6.1 列出了一个如何使用 *G4HEPEvtInterface* 的实例。注意，事件发生器并不假定提供初级粒子的位置，这个位置必须在调用方法 *GeneratePrimaryVertex()* 之前被设置。

```
#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);
}
```

代码清单 3.6.1
一个 *G4HEPEvtInterface* 接口的实例

ASCII 文件格式

一个将被 *G4HEPEvtInterface* 写的 ASCII 文件应有如下格式。

- 每个初级事件的第一行应为一个整数，表示后续有多少行初级粒子数据。
- 在一个事件中的每一行对应于一个在/HEPEVT/ 公用块中的粒子。每一行有 ISTHEP, IDHEP, JDAHEP(1), JDAHEP(2), PHEP(1), PHEP(2), PHEP(3), PHEP(5)。 有关这些变量的含义请参阅/HEPEVT/ 的手册。

代码清单 3.6.2 显示了一个生成 ASCII 文件的 FORTRAN 代码实例。

```
************************************************************
      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
```

代码清单 3.6.2
一个使用/HEPEVT/ 公用块的 FORTRAN 代码实例

未来新的事件发生器接口

目前已经开始开发面向对象的事件发生器。这些新的事件发生器可以非常容易与基于 G4 的应用程序相链接。另外，我们在 G4 中不用区分初级事件发生器和其他物理过程，未来的发生器可能就是一个从 *G4VProcess* 继承的物理过程。

3.6.3 使用多个事件发生器的事件复合

用户 *G4VUserPrimaryGeneratorAction* 类的具体类可以拥有多个 *G4VPrimaryGenerator* 类的具体类。每个 *G4VPrimaryGenerator* 类的具体类可以在同一事件过程中使用多次。通过使用这些对象，一个事件可以有多个初级事件组成。

一种可能的使用情况如下。在一个事件内部，一个使用最小偏倚事件文档初始化的 *G4HEPEvtInterface* 类对像被存取 20 次，另一个使用信号事件文档初始化的 *G4HEPEvtInterface* 类对像被存取一次。因而，这个事件表示一个典型的与 20 个最小偏倚事件复合的 LHC 信号事件。需要注意的是，事件复合的模拟可以通过合并与多个事件相关的hits和/或 digits 来实现，并且这些事件是可以独立模拟的。在 4.5 节中谈到了多个事件的数字化。

About the authors
Overview Contents Previous Next
Geant4 User's Guide
For Application Developers
Toolkit Fundamentals

3.7 事件偏倚技巧

3.7.1 Scoring，几何重要性采样和权重轮盘赌

G4 提供了使用几何分裂和俄罗斯轮盘赌（也叫几何重要性采样）以及权重轮盘赌的可能性。这些技巧用于节约计算机模拟的机时。G4 提供了一个基本的 Scoring 系统用于监视这些采样。在这一章中，我们假定读者对重要性采样和 G4 的使用都很熟悉。有关详细的文档可以在[1]找到。使用这些采样和记录技巧的各种情况的详细描述，可以在[2]中找到。

Scoring 用于估计与特定区域或表面相关的粒子的数量，应用于给定几何体的所有"cells" (物理体或其副本) 。Scoring 需要用户定制，除它为某些量（例如：径迹进入一个 cell 的数量，进入径迹的平均权重，进入径迹的能量，在 cell 内部的碰撞次数。）提供的标准 scoring。
重要性采样的目的是通过较多的采样比较“重要”的几何区域，较少的采样比较“不重要”的几何区域，来节约计算时间。采用重要性采样后的结果与原来的结果相比较，他们的平均值一定相等，但是对于相同的计算时间，前者的方差将显著减小。
Scoring 的实现与重要性采样的实现是相互独立的。但是两者有一些共同的概念。*Scoring 和重要性采样应用于用户指定的粒子类型。*
关于如何使用 scoring 和重要性采样的例子在 `examples/extended/biasing`。

3.7.1.1 几何体

这里提到的 scoring 和重要性采样应用于用户指定的空间 cells。

一个 **cell** 是一个物理体(如果是副本，需要通过复制序号指定)。这些单元可以定义为两种类型的几何体：

1. **mass geometry**: 试验中将被模拟的几何体，物理过程将应用到这些几何体中。
2. **parallel-geometry**: 这类几何体是根据它们所应用的记录和重要性采样而定义的几何体，它们并不是传统意义上的真实的物理几何体。

根据 mass geometry 或者 parallel geometry, 用户选择相应粒子的重要性进行记录和采样。对于相同的 mass geometry，可以使用多个 parallel geometry，对不同的粒子定义相互独立的，使用不同记录和重要性采样的 parallel geometry，这给用户提供了更大的灵活性。

使用**"parallel" geometries** 的一些限制

- 在 parallel geometry 中定义了一类特殊的输运过程。在当前的版本中，并没有考虑场的存在。因此，parallel geometry 只能用于无场的情况下。
- world 体的 parallel geometry 必须与 world 体的 mass geometry 相重合。
- 经过 parallel geometry 边界的粒子，在边界上，对应的 mass geometry 是真空，这些粒子也应用了偏倚。这一点在今后的版本中可能要优化。
- 记录单元不可以与 world 体存在共同的边界。

3.7.1.2 Changing the Sampling

取样器是跟高层次的工具，它对 G4 的取样过程作一些必要的变化，以便于使用重要性采样和权重轮盘赌。取样器可以应用于 scoring。

Scoring 和方差减小这两种技巧可以任意的组合，应用于特定的粒子类型，和 mass 几何体或者 parallel 几何体。取样器支持所有的组合。

需要为 mass 几何体和 parallel 几何体实现不同的取样器。两者实现了接口 `G4VSampler`，`G4VSampler` 允许准备（prepare）scoring 及方差减小的特定组合，和配置（configure）适当的采样过程。为此，`G4VSampler` 提供相应的方法 `Prepare` 和 `Configure` ：

```
class G4VSampler
{
  public:
   G4VSampler();
   virtual ~G4VSampler();
   virtual void PrepareScoring(G4VScorer *Scorer) = 0;
   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 Configure() = 0;
   virtual void ClearSampling() = 0;
   virtual G4bool IsConfigured() const = 0;
};
```

建立期望组合的方法需要指定以下信息:

- Scoring: 向 `PrepareScoring` 发送一个 `G4VScorer` 消息
- 重要性采样:向 `PrepareImportanceSampling` 发送一个 `G4VIStore` 消息和一个可选的 `G4VImportanceAlgorithm` 消息
- 权重轮盘赌：向 `PrepareWeightRoulett` 发送以下可选消息：
  - *wsurvive*: survival weight
  - *wlimit*: minimal allowed value of weight * source importance / cell importance
  - *isource*: importance of the source cell

一个取样器类的每个对象对应于一种粒子类型。这种粒子类型通过粒子类型名（如，"neutron"）向取样器类的构造函数指定。对于特定的用途，一个取样器的 `Configure()` 方法将为给定的粒子类型建立一些特定的过程，这些过程从 `G4VProcess` 派生；用于在 parallel 几何体中的输运、scoring、重要性采样和权重轮盘赌。当 `Configure()` 被调用的时候，取样器将这些过程安相应正确的顺序放置，使它们与用户使用 `Prepare` 方法调用的顺序无关。

注意

- `Prepare()` 函数每一个只能调用一次。
- 为了配置取样过程， `Configure()` 一定要在 `G4RunManager` 已经被初始化*之后*被调用。

针对 mass 和 parallel 几何体，分别提供了实现 `G4VSampler` 接口的两个类：

- `G4MassGeometrySampler`
- `G4ParallelGeometrySampler`

这两个类的构造函数都使用粒子类型名 (例如， "neutron") 作为参数。其中，`G4ParallelGeometrySampler` 的构造函数另外还需要一个 parallel 几何体的相应物理实体的引用作为参数。

3.7.1.3 Scoring

Scoring 是由一个 framework 提供的，与粒子类型相关。然而，它也可能记录不同类型的粒子于同一个记录里面。Framework 可以方便的使用于定制的 scoring 中。

Scoring 可以应用于 mass 或 parallel 几何体。它通过一个通常叫做 scorer 的对象来实现，这个对象使用一个上面提到的取样器。这个 scorer 接收给定类型的粒子每一步所产生的信息。这些信息由两个对象组成，一个是 G4 内核类 `G4Step` 的对象，另一个是为 scoring 和重要性产样而提供的特定的 `G4GeometryCellStep` 类的对象。`G4GeometryCellStep` 提供粒子径迹的当前 cell 和前一个 cell 的有关信息。

"scorer"类从接口 `G4VScorer` 派生。用户可以创建定制的"scorers"，也可以使用标准 scoring。

在 framework 中涉及的类：

- `G4VScorer`
  从 `G4VScorer` 派生的类可以为 scoring 使用 G4 framework。为了减小程序的耦合性，一个定制的 scoring 必须使用 scoring frame。这个 frame 将提供一个用于发送从 scorer（从 `G4VScorer` 派生）消息的过程，和用于 setup 的类。
  特别地，这种类型的对象被指定一个 `G4VSampler`。为了在同一个 `G4VScorer` 中记录不同粒子类型 ，该对象可以被指定不同的取样器对象。
  接口：
-
- ```
    class G4VScorer
    {
      public:
        G4VScorer();
        virtual ~G4VScorer();
        virtual void Score(const G4Step &step, const G4GeometryCellStep
  &gstep) = 0;
    };
  ```

  成员函数 `Score(const G4Step &step, const G4GeometryCellStep &gstep)` 将为一个特定粒子的每一个"PostStep"调用，并且是在物理过程的"`PostStepDoIt()`"函数调用之前。

- `G4GeometryCellStep`:
-
- ```
    class G4GeometryCellStep
    {
      public:
        G4GeometryCellStep(const G4GeometryCell &preCell,
  ```

```
                        const G4GeometryCell &postCell);
        ~G4GeometryCellStep();
        const G4GeometryCell &GetPreGeometryCell() const;
        const G4GeometryCell &GetPostGeometryCell() const;
        G4bool GetCrossBoundary() const;
        void SetPreGeometryCell(const G4GeometryCell &preCell);
        void SetPostGeometryCell(const G4GeometryCell &postCell);
        void SetCrossBoundary(G4bool b);
      private: ....
    };
```

在 G4GeometryCellStep 中使用的类和函数：

- G4GeometryCell() 将一个"cell"看作是一个有 replica number 的物理体。如果使用了一个用于"parallel"几何体的取样器，那么这个"cell"是在"parallel"几何体中；否则，这个"cell"在 mass 几何体中。
- GetPreGeometryCell() 返回粒子碰到的前一个"cell" ，或者在径迹没有穿越边界的情况下，它等价于 GetPostGeometryCell()。
- GetPostGeometryCell() 指向当前"cell"。
- GetCrossBoundary() 在粒子在几何体中穿越一个边界的情况下，返回 true。

注意

- 当 scoring 应用于一个"parallel"几何体的时候，必须采用特殊的 action，防止记录与 mass 几何体边界的"collisions"。当在该 mass 几何体中使用 scoring 的时候，这将进行不同的处理。

3.7.1.4 重要性产样

重要性采样作用于那些穿过不同"importance cells"的边界的粒子。这个行为依赖于给 cells 指定的重要性值。通常，如果重要性增加或者重要性减小，那么，相应的，这个粒子 history 被分裂或者被使用俄罗斯轮盘赌。根据相应的行为，这个 history 指定的权重将被改变。

这些为重要性采样提供的工具要求用户对问题的物理过程有很好的理解。这是因为用户需要决定哪个粒子必须被重要性采样，定义 cells 并且给相应的 cells 赋相应的重要性值。如果这些没有被正确的指定，那么不能期望获得描述真实实验的结果。

下面描述了怎样使用 importance store 给一个 cell 指定重要性值。

与接口 G4VIStore 有关的"importance store"被用于存储相关 cells 的重要性值。为了进行重要性采样，用户必须创建一个 G4VIStore 类型(例如，G4IStore 类)的对象 。取样器可以指定一个 G4VIStore。用户使用 cells 和相应的重要性值来填充 importance store。

Importance store 必须使用用于重要性采样的几何体的 world 几何体的引用来构造，可以是 mass 几何体或者 parallel 几何体的 world 几何体。Importance stores 从接口 G4VIStore 派生：

```
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;
};
```

由类 `G4VStore` 提供的 importance store 的具体实现。这个类的公有部分：

```
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: .....
};
```

成员函数 `AddImportanceGeometryCell()` 将一个 cell 及其重要性值添加到 importance store。根据一个物理体和一个 replica number，或者根据一个 `G4GeometryCell`，可以获取相应的重要性值。用户必须注意给一个 cell 赋重要性值的阐述。

注意

- 重要性值必须赋给每一个 cell。
- 在 G4 中，world 体的 replica number 是 -1。

不同情况：

- *Cell 不在 importance store 中*
  没有填充某个 cell 到 store 将引起一个异常。
- *重要性值 =0*
  相应粒子的径迹将被 kill。
- *重要性值 > 0*
  正常允许值
- *重要性值<0*
  不允许!

3.7.1.5 重要性采样算法

重要性采样支持使用用户自定义的重要性产样算法。最后，取样器接口 `G4VSampler` 将被指定一个指向接口 `G4VImportanceAlgorithm` 的指针：

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

方法 `Calculate()` 使用以下参数：

- *ipre, ipost*: 分别是前一个 cell 和当前 cell 的重要性值。
- *init_w*: 粒子权重

它返回以下结构：

```
class G4Nsplit_Weight
{
  public:

  G4int fN;
  G4double fW;
};
```

- *fN*: 到退出重要性采样的时候已计算的粒子数目
- *fW*: 粒子权重

用户可以通过提供一个从 `G4VImportanceAlgorithm` 继承的类使用自定义的算法
如果没有自定义算法指定给取样器，缺省的重要性采样算法将被使用。这个算法在 `G4ImportanceAlgorithm` 中实现。

3.7.1.6 权重轮盘赌技巧

如果重要性采样和隐含俘获被同时使用，那么权重轮盘赌 (也叫权重截断)通常被采用。隐含俘获没有在这里描述，需要注意：隐含俘获在每次碰撞发生的时候，以一定的概率，使用减小粒子权重的办法来代替杀死粒子。

与重要性采样一起，粒子权重可以变得很小以致不会显著改变任何结果。所以，跟踪一个非常低权重的粒子意味着浪费计算时间。为了解决这个问题，将使用权重轮盘赌。

权重轮盘赌概念

权重轮盘赌必须考虑当前 cell 的重要性"Ic"，和源所在 cell 的重要性，使用他们的比"R=Is/Ic"。

权重轮盘赌使用一个相对最小权重限和一个相对存活权重。当一个粒子掉到低于权重限的时候，将使用俄罗斯轮盘赌。如果粒子存活，跟踪将继续，并且粒子权重将被设置为存活权重。

权重轮盘赌使用下列参数和他们的缺省值：

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

下列算法被采样：

如果粒子权重"w"小于 R*wlimit:

- 粒子权重将被改变为"ws = wsurvival*R"
- 粒子存活的可能性为"p = w/ws"

[1] Scoring, geometrical importance sampling and weight roulette in Geant4
[2] Use cases for importance biasing and scoring technique

4. 探测器定义和响应

About the authors
Overview Contents Previous Next

4.1 几何

4.1.1 介绍

探测器的定义要求探测器本身几何形状的描述、它们的组成材料及电学特性、还有他们的可视化属性及用户自定义的属性。探测器单元的几何描述关注的是它们的实体模型定义及它们的空间位置，还有就是它们的逻辑关系，例如它们的限定关系。

G4 使用“逻辑体”的概念来管理这些探测器单元属性的描述，使用“物理体”的概念来管理探测器单元空间位置及它们的逻辑关系的描述，使用“实体”的概念来管理探测器单元实体模型的描述。

G4 实体的建模与 STEP 相兼容，通过使用 STEP 的 EXPRESS object definition language 来对实体模型描述进行标准化。STEP 是一个 ISO 标准，用于 CAD 系统之间交换几何数据。

4.1.2 实体 Solids

STEP 标准支持多种实体描述方法，有 Constructive Solid Geometry (CSG) representations 和 Boundary Represented Solid (BREP)。不同的描述适合于不同目的、不同应用、不同的复杂性要求和详细程度。CSG 描述通常有较好的性能和容易使用的特点，但是它不能像 CAD 系统那样生成一些复杂的实体。BREP 描述可以处理更广的拓扑结构、生成最复杂的实体，因此允许用来和 CAD 系统交换模型数据。
所有建立的实体通过使用适当的方法和流操作符可以方便的提供流输出。

4.1.2.1 CSG 实体 Constructed Solid Geometry (CSG) Solids

CSG 实体是直间作为 3D 元素定义的，它们通过尽量少的参数来定义它们的形状、大小。这些 CSG 实体有 Boxes, Tubes and their sections, Cones and their sections, Spheres, Wedges, and Toruses.

要建立一个 box，用户可以使用下面这个构造函数：

```
G4Box(const G4String& pName,
          G4double  pX,
          G4double  pY,
          G4double  pZ)
```

通过提供一个这个 box 的名字和它沿 X，Y 和 Z 轴的半长度：

pX	X 轴方向半长	pY	Y 轴方向半长	pZ	Z 轴方向半长

这将生成一个 box，它在 X 轴方向从-pX 到+pX ，Y 轴方向从-pY 到+pY ，Z 轴方向从-pZ 到+pZ 。

例如要建立一个 2×6×10cm 的 box，将它命名为 BoxA：

```
G4Box* aBox= G4Box("BoxA", 1.0*cm, 3.0*cm, 5.0*cm);
```

类似的要建立一个圆柱或者 tube，用户可以使用下面的构造函数：

```
G4Tubs(const G4String& pName,
           G4double  pRMin,
           G4double  pRMax,
           G4double  pDz,
           G4double  pSPhi,
           G4double  pDPhi)
```

将它命名为 pName， 其余参数的意义是：

pRMin	内半径	pRMax	外半径
pDz	Z 轴方向的半长值	pSPhi	圆周起始位置弧度值
pDPhi	该实体的圆心角弧度值		

类似的，要建立一个圆锥或者 conical section，用户可以使用以下的构造函数：

```
G4Cons(const G4String& pName,
           G4double  pRmin1, G4double pRmax1,
           G4double  pRmin2, G4double pRmax2,
           G4double  pDz,
           G4double  pSPhi, G4double pDPhi)
```

将它命名为 pName， 其余参数是：

pDz	Z 轴方向半长值	pRmin1	在-pDz 位置处的内半径值
pRmin2	在+pDz 位置处的内半径值	pRmax1	在-pDz 位置处的外半径值
pRmax2	在+pDz 位置处的外半径值	pSPhi	圆周起始位置弧度值
pDPhi	该实体的圆心角弧度值		

建立一个平行六面体使用：

```
G4Para(const G4String& pName,
```

```
                    G4double  dx, G4double dy, G4double dz,
                    G4double  alpha, G4double theta, G4double phi)
```

将它命名为 pName ，其余参数是

dx,dy,dz	在 x，y，z 轴方向的半长值
alpha	在-dy 和+dy 处，z-x 平面与 y 轴成的夹角
theta	在 z 轴方向，-dz 和+dz 处的平面中心的连线的极角
phi	在 z 轴方向，-dz 和+dz 处的平面中心的连线的方位角

要建立一个梯形台，可以使用：

```
G4Trd( const G4String& pName,
           G4double  dx1, G4double dx2,
           G4double  dy1, G4double dy2,
           G4double  dz )
```

该实体的名字为 pName ，它的参数是：

dx1	在-dz 处的平面，沿 x 轴方向的半长值
dx2	在+dz 处的平面，沿 x 轴方向的半长值
dy1	在-dz 处的平面，沿 y 轴方向的半长值
dy2	在+dz 处的平面，沿 y 轴方向的半长值
dz	Z 轴方向半长值

要建立一个更为一般的梯形台，可以使用 G4Trap 类。下面是为 Right Angular Wedge 定义的最简构造函数：

```
G4Trap( const G4String& pName,
            G4double  pZ, G4double pY, G4double pX,
            G4double  pLTX )
```

这个实体的名字为 pName ，参数是

pZ	Z 轴方向长度
pY	Y 轴方向长度

pX	宽的侧面 x 轴方向长度
pLTX	窄的侧面 x 轴方向长度 (plTX<=pX)

要了解这些构造函数的完全几何请阅读 Software Reference Manual.

要建立一个球体，可以使用：

```
G4Sphere( const G4String& pName,
             G4double  pRmin, G4double pRmax,
             G4double  pSPhi, G4double pDPhi,
             G4double  pSTheta, G4double pDTheta )
```

该实体的名字为 pName ， 参数是

pRmin	内半径
pRmax	外半径
pSPhi	圆周起始位置弧度值 （内球面？）
pDPhi	该实体的圆心角弧度值 （内球面？）
pSTheta	圆周起始位置弧度值 （外球面？）
pDTheta	该实体的圆心角弧度值 （外球面？）

要建立一个圆环，使用：

```
G4Torus( const G4String& pName,
            G4double  pRmin, G4double pRmax,
            G4double  pRtor, G4double pSPhi, G4double pDPhi )
```

实体名为 pName ， 参数是

pRmin	圆环截面内半径
pRmax	圆环截面外半径
pRtor	圆环半径
pSPhi	圆周起始位置弧度值 (fSPhi+fDPhi<=2PI, fSPhi>-2PI)
pDPhi	该实体的圆心角弧度值

另外， Geant4 Design Documentation 在 Solids Class Diagram 中给出了 CSG 类的完整列表 ，STEP 文档包括每个 CSG 实体的详细的 EXPRESS 描述。

特殊的 CSG 实体

在 G4 中，通过 `G4Polycon` 类实现了 Polycons (PCON) ：

```
  G4Polycone( const G4String& pName,
                 G4double  phiStart,
                 G4double  phiTotal,
                 G4int    numZPlanes,
            const G4double  zPlane[],
            const G4double  rInner[],
            const G4double  rOuter[])

  G4Polycone( const G4String& pName,
                 G4double  phiStart,
                 G4double  phiTotal,
                 G4int    numRZ,
            const G4double  r[],
            const G4double  z[])
```

参数是：

`phiStart`	Phi 的起始角
`phiTotal`	Phi 的总值
`numZPlanes`	Z 平面数
`numRZ`	在 r，z 空间的转角数
`zPlane`	Z 平面位置
`rInner`	到内表面的切线距离（内切圆半径）
`rOuter`	到外表面的切线距离 （内切圆半径）
`r`	转角坐标 r
`z`	转角坐标 z

用 `G4Polyhedra` 实现了 Polyhedra (PGON) ：

```
  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[] )
```

参数是：

phiStart	Phi 的起始角
phiTotal	Phi 的总值
numSide	侧面数
numZPlanes	Z 平面数
numRZ	在 r，z 空间的转角数
zPlane	Z 平面位置
rInner	到内表面的切线距离（内切圆半径）
rOuter	到外表面的切线距离 （内切圆半径）
r	转角坐标 r
z	转角坐标 z

一个 tube 和椭球的相交部分可以使用如下定义：

```
   G4EllipticalTube( const G4String& pName,
                           G4double  Dx,
                           G4double  Dy,
                           G4double  Dz )
```

用 x/y 表示的表面方程是 `1.0 = (x/dx)**2 + (y/dy)**2`

Dx	X 轴方向半长	Dy	Y 轴方向半长	Dz	Z 轴方向半长

一个有双曲侧面的 tube 可以使用如下定义：

```
G4Hype(const G4String& pName,
             G4double  innerRadius,
             G4double  outerRadius,
             G4double  innerStereo,
             G4double  outerStereo,
             G4double  halfLenZ )
```

G4Hype 是由平行于 z 轴，按指定距离 halfLenZ 和最大、最小半径，以 z 轴为中心，围绕 z 轴的曲面形成的。
最小半径为 0 定义了一个内部填充的 Hype，它的内表面是双曲面。
内外双曲表面可以有不同的立体角。立体角为 0 表明是一个柱面。
同样，也可以定义最小(内)半径为 0，内立体角也为 0。

innerRadius	内半径
outerRadius	外半径
innerStereo	内表面所张立体角弧度值
outerStereo	外表面所张立体角弧度值
halfLenZ	与 Z 轴距离

4.1.2.2 使用布尔运算定义的实体

简单的实体可以通过布尔运算被组合起来。例如，我们可以使用 union 运算将一个柱体很一个半球结合起来。

建立这样一个“布尔”实体，要求：

- 两个实体
- 一个布尔运算：合 union, 交 intersection 或者差 subtraction。
- 可选的，用于第二个实体的变换。

用于运算的实体需是 CSG 实体或者其他的“布尔”实体。“布尔”实体的重要用途是它可以用简单、直观的方法描述一些形状特殊的实体， still allowing an efficient geometrical navigation inside them。

注意：这些实体事实上可以是任何类型。然而，为了完全支持使用 STEP，将 G4 实体模型导出到 CAD 系统中，用户被限制只能使用上述的实体子集，不过这个子集几乎包括了所有的常用情况。

注意：在“布尔”实体中，粒子跟踪过程用于 navigating 的开销是正比于组成该实体的子实体的数目（对于目前的实现方法）。所以，用户必须小心，在一些敏感区域的几何描述，不要大量使用不必要的“布尔”实体。

下面是一些最简单的“布尔”实体的例子：

```
G4Box box1("Box #1",20,30,40);
G4Tubs Cylinder1("Cylinder #1",0,50,50,0,2*M_PI);  // r:    0 -> 50
                                                   // z:   -50 -> 50
                                                   // phi:   0 ->  2 pi
G4UnionSolid b1UnionC1("Box+Cylinder", &box1, &Cylinder1);
G4IntersectionSolid b1IntersC1("Box Intersect Cylinder", &box1, &Cylinder1);
G4SubtractionSolid b1minusC1("Box-Cylinder", &box1, &Cylinder1);
```

这里，分别用合，交、差三种运算将一个 box 和一个柱体组成了新的“布尔”实体。

比较有用的情况是，这些实体中的某个实体需要在原来的坐标系中移动。在这种情况下，需要为第二个实体指定坐标（相对于第一个实体），可以通过以下两种方式：

- 通过给定的旋转矩阵和平移向量，将第一个实体的坐标系变换到第二个实体的坐标系。这被叫做被动模式。
- 或者，通过变换，移动第二个实体到相对于它标准始位置的期望的位置。例如，一个 box 的标准位置是它的中心在坐标原点、各边分别平行于坐标轴。这被称作主动模式。

在第一种情况中，首先平移坐标原点，然后再将第一个坐标系相对第二个坐标系进行旋转。

```
G4RotationMatrix yRot45deg;  // Rotates X and Z axes only

yRot45deg.rotateY(M_PI/4.*rad);
G4ThreeVector  translation(0, 0, 50);
G4UnionSolid  box1UnionCyl1Mv("Box1UnionCyl1Moved",
             &box1,&Cylinder1,&yRot45deg,translation);
// 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= yRot45deg;
yRot45deg.invert();
// or else InvRot.yRotate(-M_PI/4.0*rad);
G4Transform3D transform(InvRot,translation);
G4UnionSolid  SameUnion("Box1UnionCyl1Moved2",&box1,&Cylinder1,transform);
```

注意第一个构造函数种传递的参数 G4RotationMatrix 并没有被复制，他是使用引用（指针）进行传递的。因此，一旦使用旋转矩阵建立一个布尔实体，这个旋转矩阵是不能被更改的。

相反， G4Transform3D 是通过值传递提供给相应的构造函数的，它的变换保存在相应的布尔实体中，因此，用户可以更改 G4Transform3D 并且重复使用它。

4.1.2.3 Boundary Represented (BREPS) Solids

BREP 实体是通过它们的边界描述定义的。这些边界可以是平面也可以是二次曲面，这些曲面还可以被裁剪，表面还可以有蚀洞。例如 polygonal, polyconical, 和 hyperboloidal 实体是一些基本的 BREPS 实体。

另外，这些边界表面可以由贝塞尔表面(Bezier surfaces)和 B 样条表面（B-Splines）、或者非均匀有理 B 样条表面 NURBS (Non-Uniform-Rational-B-Splines) surfaces。它们组成一些比较高级得 BREPS 实体。
目前，*Beziers, B-Splines 或 NURBS 表面生成得实现仅仅停留在原型得层次上，并没有实现其全部功能。*

G4 已经定义了一些简单得基本 BREPS 实体，它们可以使用与创建 CSG 实体类似得方法来创建这些实体，下面将简述它们得功能。

然而，大多数 BREPS 实体是通过分别创建不同表面，然后将这些表面绑定到一起形成的。虽然，可以通过直接的编码来完成这些工作，但这是一种潜在的错误倾向。通常，更多的使用一些更为高效的工具去创建这些实体，例如使用 CAD 系统工具。在 CAD 系统内建立的模型可以利用 STEP 标准导出。

所以，BREPS 实体通常是通过使用一个 STEP 读入程序定义的，这个读入程序允许用户导入在 CAD 系统中建立的任何实体模型，就如本手册中 4.1.10 节 所描述的一样。

特殊 BREP 实体 Specific BREP Solids

G4 已经使用 BREPS 定义了一个 polygonal shape 和一个 polyconical shape。Polycone 定义了这样一个形状，它是由一系列的共轴的并沿轴连续分布的圆锥截面组成的。

Polyconical 实体 G4BREPSolidPCone 是通过一系列的内部和外部圆锥面或柱面、和两个垂直于 z 轴的平面组成的。每个圆锥截面根据它在垂直于 z 轴的、两个不同平面上的截面半径来定义的。相应的内、外圆锥截面使用共同的 Z 平面来定义。

```
G4BREPSolidPCone( const G4String& pName,
                        G4double  start_angle,
                        G4double  opening_angle,
                        G4int     num_z_planes,    // sections,
                        G4double  z_start,
                  const G4double  z_values[],
                  const G4double  RMIN[],
                  const G4double  RMAX[]  )
```

圆锥截面并不需要 360 度填充，但是它们有共同的起始角和共同的开口。

start_angle	起始角
opening_angle	开口角
num_z_planes	使用的垂直于 z 轴的平面数
z_start	Z 轴起始值
z_values	每个平面的 z 轴坐标
RMIN	在每个平面上内圆锥的半径
RMAX	在每个平面上外圆锥的半径

Polygonal 实体 G4BREPSolidPolyhedra 是通过内外多边形表面和两个垂直于 z 轴的平面形成的。每个多边形表面表面是通过联接一系列垂直于 z 轴、位于 z 轴不同位置的多边形而形成的。所有这些多边形有同样的边数，并且在同一个 z 轴平面定义相应的内外多边形表面。

同样，polygons 也不需要 360 度填充，但是需要有相同的起始角和开口角。

它的构造函数使用如下参数：

```
G4BREPSolidPolyhedra( const G4String& pName,
                      G4double  start_angle,
                      G4double  opening_angle,
                      G4int     sides,
                      G4int     num_z_planes,
                      G4double  z_start,
                const G4double  z_values[],
                const G4double  RMIN[],
                const G4double  RMAX[]  )
```

除了它的名字之外，其他参数意义如下：

start_angle	起始角
opening_angle	开口角
sides	每个在 x-y 平面内的多边形的边数
num_z_planes	使用的垂直于 z 轴的平面数
z_start	Z 轴起始值
z_values	每个平面的 z 轴坐标

RMIN	在每个平面上内圆锥的半径
RMAX	在每个平面上外圆锥的半径

形状是由多边形的边数确定的，这些边在垂直于 z 轴的平面内。

其他 BREP 实体 Other BREP Solids

其它实体可以通过定义它们的边界表面来创建。创建这些 BREP 实体是相当复杂的。所以它们一般用 CAD 程序来创建，CAD 系统可以建立单个的独立实体，也可以建立有几个实体组合成的复合实体。就如本手册 4.1.10 节所描述的，可以通过使用 STEP 交换文件将这些实体的定义导入到 G4。

4.1.3 逻辑体

逻辑体管理那些与给定实体和材料的探测器单元有关的信息，与它们在探测器中的物理位置无关。

逻辑体知道在其内部包含的物理体是什么，它被定义为这些物理体的唯一母体。一个逻辑体描述了一个没有被指定位置的实体的层次结构，而这些实体（物理体）直接有明确定义的位置关系。创建物理体实际就是放置一个逻辑体的实例，这个层次结构或者说是树状结构可以被重复使用。

一个逻辑体同时也管理与可视化属性相关的信息，和用户定义的，与粒子跟踪、电磁场或截断（通过 G4UserLimits 结口）相关的参数。

缺省情况下，粒子跟踪过程的几何体优化被应用于那些通过逻辑体标识的物理体层次结构上。可以针对一个给定的逻辑体选择不使用几何体优化。但是，这个特性不可以应用于那些由参数化实体组成的物理体，在这这种情况下，优化是始终进行的。

```
G4LogicalVolume( G4VSolid*           pSolid,
                 G4Material*         pMaterial,
                 const G4String&      Name,
                 G4FieldManager*      pFieldMgr=0,
                 G4VSensitiveDetector* pSDetector=0,
                 G4UserLimits*        pULimits=0,
                 G4bool              Optimise=true )
```

最后，逻辑体管理那些与 Envelope 层次结构相关的信息，Envelopes 层次结构是快速 Monte Carlo parameterizations(5.2.6 节) 所要求的。

4.1.3.1 子探测器区域 Sub-detector Regions

在复杂的几何设置中，例如在粒子物理实验中的大型探测器，将整个探测器设置中具有特定功能的部分（子探测器）作为一个特定的逻辑体来处理是非常有用的。在这样的设置中，通过对不同子探测器设置特定的截断值，可以提高模拟的处理速度。这使得可以只在那些需要的区域进行更详细的模拟。

处于这中需要，探测器区域 *Region* 的概念被引入。一旦探测器的最终几何设置被定义，一个区域可以通过以下的方法来建立：

```
G4Region( const G4String&        rName )
```

在这

`rName`	探测器区域的字符串标识符

为了使 `G4Region` 成为一个*根逻辑体*，它必须被指定给一个逻辑体：

```
G4Region* emCalorimeter = new G4Region("EM-Calorimeter");
emCalorimeter->AddRootLogicalVolume(emCalorimeter);
```

一个根逻辑体是第一个逻辑体，位于给定区域的层次结构的最顶层。一旦一个区域被指定给一个根逻辑体，这个信息将自动的传递到整个逻辑体的树状结构中，以便使每个子体可以共享同一个区域。如果传递过程中发现某个子树已经存在根逻辑体，那么在这个子树上的传递过程将被中止。

通过定义一个 `G4ProductionCut` 对象，并赋值，一个特定的*截断值*可以被指定给特定区域。

```
emCalorimeter->SetProductionCuts(emCalCuts);
```

5.4.2 节 描述了如何定义一个截断值。同一个区域可以被赋给多个根逻辑体，也可以从一个已经存在的区域中删除根逻辑体。但是，一个逻辑体只能有一个区域被指定给它。区域将被自动注册存储，在任务结束的时候被删除。一个具有缺省截断值的缺省区域被自动建立并指定给了 world。

4.1.4 物理体 Physical Volumes

物理体指明了那些描述探测器单元的 volumes 的空间位置。在这里，我们可以简单的放置单个拷贝，也可以按照简单线性方程或者用户指定的函数重复放置这些副本。

一个简单的放置涉及一个变换矩阵的定义，这个矩阵将用于指定要放置的逻辑体的位置。重复放置使用一个顺序号，按照一个给定的距离，沿给定的方向，复制一个逻辑体。也可以定义一个参数方程，用于指定一个逻辑体多个副本的位置。这些方法的详细信息将在下面给出。

注意 – 对于在不同的 runs 直接，几何体发生改变。在这种情况下，旧的几何设置需要被显式删除，必须注意删除的顺序（与实际创建几何体时的顺序相反，例如，先删除物理体，然后才是逻辑体）。删除一个逻辑体，并不会自动删除它的子体。
在任务结束的时候，没有必要删除几何设置，因为，系统将自动释放它们的存储空间。在用户自己的应用程序中，必须注意删除那些由用户自己添加的，使用动态内存分配的变换矩阵或旋转矩阵。

4.1.4.1 放置：单位置拷贝 Placements: single positioned copy

在这种情况下，物理体是通过联合一个逻辑体和一个旋转矩阵及一个平移向量而建立的。旋转矩阵表示要放置的逻辑体所在参考系相对于它的母体参考系的旋转。平移向量表示当前逻辑体在母体参考系内的平移。
包含镜像的变换是不允许的。

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 )
```

以下

pRot	相对它母体的旋转
tlate	相对它母体的平移
pName	这个放置的字符串标识符
pCurrentLogical	相关的逻辑体
pMotherLogical	相关的母体
pMany	为今后预留，可以设置为 false
pCopyNo	标识这次放置的整数

必须注意旋转矩阵并没被 G4PVPlacement 复制，所以，在用户使用这个旋转矩阵进行放置之后，不能再一次更改它。当然，同一个旋转矩阵可以被多个逻辑体使用。

目前，布尔运算没有在物理体的层次上实现。所以 pMany 一定是 false。 不过，已经实现了一个布尔运算的替代方法，使用这种方法，可以获得两个实体得合、交、或者差。有关这一点，请参看 4.1.2.2 节。

除 world 外，用户必须为所有的 volumes 指定母体。

还有一种不同的方法可以放置逻辑体。实体自身通过旋转和平移，将它移入母体的坐标系。这种“主动”方法可以使用下面的构造函数来进行：

```
G4PVPlacement(      G4Transform3D     solidTransform,
                  G4LogicalVolume*   pCurrentLogical,
              const G4String&         pName,
                  G4LogicalVolume*   pMotherLogical,
                  G4bool            pMany,
                  G4int             pCopyNo )
```

另外一种指定母体的方法就是指定已经放置的物理体。这种方法可以被用在上述指定放置位置和旋转的方法中，其结果是跟使用逻辑母体的方法完全一样的。

注意一个放置了的逻辑体仍可以表示多个探测器单元。如果它的母逻辑体存在多个副本，那么这种情况就会出现。当然，这些不同的探测器单元属于这个几何体层次结构树上的不同分枝。

4.1.4.2 Repeated volumes

使用用单个物理体描述一个逻辑体在其母体内的多个副本，可以节约内存。这通常要求被放置的物理体在笛卡儿坐标或柱坐标中是严格对称的。Repeated volumes 技巧适用于 CSG 实体。

副本 Replicas

副本是指一类 *Repeated volumes*，它们中的每一个个体都可以被完全识别。用户需要为程序指定副本数目和坐标轴，以便在运行的时候计算每个副本对应的变换矩阵。

```
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	被复制的逻辑体的字符串标识符

pCurrentLogical	相关逻辑体 （被复制的逻辑体）
pMotherLogical	相关母体
pAxis	复制方进行向的轴线
nReplicas	副本数目
width	沿复制轴方向，单个副本的宽度
offset	Possible offset associated to mother offset along the axis of replication

G4PVReplica 描述了 nReplicas 个仅仅在位置上不同的逻辑体，并且完全**填充**容纳它们的母体。因此，如果在一个给定的母体内部有一个已经被放置了的 G4PVReplica ，那么，它一定是这个母体的唯一子体。这些副本对应于那些完全填充母体、并且没有偏移的分割或者切片。在笛卡儿坐标系中，这些切片被认为是垂直于复制轴的。

副本的位置是通过一个线性关系式计算的。复制可以沿着以下这些方向进行：

- *笛卡儿坐标轴 (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.
- *径向轴（柱极坐标）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
- *切向轴（方位角）（柱极坐标）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

对于沿笛卡儿坐标轴的情况，副本坐标系位于每个副本的中心。对于沿径向轴的情况，坐标系与母体相同。对于沿切向轴的情况，新的坐标系进行了旋转，X 轴二等分每个 wedge 所张的角。Z 轴保持与母体的 Z 轴平行。

为了便于可视化，通过那些副本的逻辑体相联关的实体，它们的外形尺寸应与最初的那个逻辑体相同，并且有一定的对称性。
例如，在一个 box 内，沿 X 轴方向的副本，它们因该是具有相同尺寸的另一种 box。 (与母体波形有相同的 Y & Z 尺寸，X 方向的尺寸＝母体 X 方向尺寸/nReplicas)。

副本可以放置在其他副本的内部，但要遵守上述的各种规则。正常放置的逻辑体也可以放置在副本内部，但是不能与母体或者其他副本的边界相交。Parameterised volumes 不可以被放到其他副本内部。正因为上面的那些规则，也不可以在沿径向复制的逻辑体内部放置其它逻辑体。

在粒子跟踪过程中，根据当前“活动的”复制过程，来修改与每个 G4PVReplica 对象相关的平移和旋转。在这个过程中，那些实体没有进行任何改变，结果导致在沿切向和径向复制，及沿复制方向母体的截面不是恒定不变时，将得到错误的参数。

例子:

```
G4PVReplica repX("Linear Array",
                 pRepLogical,
           pContainingMother,
                 kXAxis, 5, 10*mm);

G4PVReplica repR("RSlices",
                 pRepRLogical,
                 pContainingMother,
                 kRho, 5, 10*mm, 0);

G4PVReplica repRZ("RZSlices",
                  pRepRZLogical,
                  &repR,
                  kZAxis, 5, 10*mm);

G4PVReplica repRZPhi("RZPhiSlices",
                     pRepRZPhiLogical,
                     &repRZ,
                     kPhi, 4, M_PI*0.5*rad, 0);
```

Source listing 4.1.1
G4PVReplica 简单 Repeated volumes 的例子

RepX 是宽度 10 毫米的 5 个副本组成的一个系列，放置在由 pContainingMother 指定的母体里面，母体被完全填充。母体的 X 方向长度一定是 5*10 毫米=50 毫米。（举例来说，如果母体是一个相应半长度的 box[25,25,25]，那么副本一定是半长度为 5 的一个 box [25,25,5])
如果那个包含母体的实体是一个 Z 轴半长为 25 毫米，半径为 50 毫米的一个 tube，RepR 把母体 tube 管分为 5 个圆柱体（与 pRepRLogical 相关的实体一定是半径为 10 毫米，Z 方向半长为 25 毫米的一个 tube)； repRZ 把它分为 5 个较短圆柱体（与 pRepRZLogical 相关的实体一定是半径为 10 毫米，Z 方向半长为 5 毫米的一个 tube）；最后， repRZPhi 用 90 度角把它分为 4 个 tube 片段。（与 pRepRZPhiLogical 相关的实体一定是半径为 10 毫米，Z 方向半长为 5 毫米，圆心角为 PI/2 的一个 tube 片段）

其它的逻辑体不能被放置在这些副本之内。由于径向复制的存在，这么做会导致边界相交，这是不允许的。**Parameterised Volumes**

Parameterised Volumes 是一类 *Repeated volumes* ，它们是一个逻辑体的多个拷贝，这些拷贝可以有不同的尺寸，不同的实体类型，或者不同的材料。不管是否存在对称结构，实体的类

型、尺寸、材料和变换矩阵都可以是参数化的，是复制号的函数。用户实现那些期望的参数方程，应用程序在运行的时候将计算并自动刷新与物理体相关的信息。

在 novice 例子 N02 中，有一段建立 parameterised volume 的代码，在下面的两个文件 `ExN02DetectorConstruction.cc` 和 `ExN02ChamberParameterisation.cc`. 中。

要建立一个 parameterised volume，用户首先需要建立它的逻辑体，像下面的 `trackerChamberLV` 。然后必须建立自己的参数化类 (*ExN02ChamberParameterisation*)，并将这个类(`chamberParam`)实例化。下面是具体的例子。

```
//------------------------------
// Tracker segments
//------------------------------
// An example of Parameterised volumes
// dummy values for G4Box -- modified by parameterised volume
G4VSolid * solidChamber =
           new G4Box("chamberBox", 10.*cm, 10.*cm, 10.*cm);

G4LogicalVolume * trackerChamberLV
  = new G4LogicalVolume(solidChamber, Aluminum, "trackerChamberLV");
G4VPVParameterisation * chamberParam
  = new ExN02ChamberParameterisation(
                  6,          //  NoChambers,
                 -240.*cm,     //  Z of centre of first
                 80*cm,       //  Z spacing of centres
                 20*cm,       //  Width Chamber,
                 50*cm,       //  lengthInitial,
                 trackerSize*2.); //  lengthFinal

G4VPhysicalVolume *trackerChamber_phys
  = new G4PVParameterised("TrackerChamber_parameterisedPV",
                 trackerChamberLV, // Its logical volume
                 physiTracker,    // Mother physical volume
                 kZAxis,         // Are placed along this axis
                 6,             // Number of chambers
                 chamberParam);   // The parameterisation
// kZAxis is used only for optimisation in geometrical calculations,
// it specifies the axis along which to apply one-dimensional voxelisation.
// If 3D voxelisation is wished (i.e. along the three cartesian axis, as it
// is applied for normal placements), "kUndefined" should be specified instead.
```

Source listing 4.1.2
一个 Parameterised volumes 的例子。

通常的构造函数是：

```
   G4PVParameterised( const G4String&           pName,
                      G4LogicalVolume*      pCurrentLogical,
                      G4LogicalVolume*      pMotherLogical, // OR
G4VPhysicalVolume*
                  const EAxis               pAxis,
                  const G4int               nReplicas,
                      G4VPVParameterisation* pParam )
```

注意：对于一个 parameterised volume 来说，用户必须指定一个母体。所以 world 体永远都不可能是一个 parameterised volume。母体可以是物理体，也可以是逻辑体。

`pAxis` 指定将应用的粒子跟踪优化算法：如果一个可用的坐标轴（该坐标轴为参数化进行的方向）被指定，那么一个简单的一维体素化(voxelization ,voxelization 实际上就是将空间离散化，有关的概念参考"Volume Graphics" *IEEE Computer, Vol. 26, No. 7* July 1993 pp. 51-64)算法将被使用；如果指定的是"kUndefined"，缺省的三维体素化算法将被使用。对于第二种情况，将产生更多的体素，因此，相应的优化算法也将销耗更多的内存。

与 parameterised volume 相关的参数化机制被定义在相应的参数化（parameterisation） 类和它的方法中。每个参数化类必须有两个方法：

- `ComputeTransformation` 定义一个拷贝放置的位置，
- `ComputeDimensions` 定义一个拷贝的尺寸，及
- 一个用来初始化各个成员变量的构造函数

例子 `ExN02ChamberParameterisation` 定义了一系列不同尺寸的 box：

```
class ExN02ChamberParameterisation : public G4VPVParameterisation
{
  ...
  void ComputeTransformation(const G4int      copyNo,
                         G4VPhysicalVolume *physVol) const;

  void ComputeDimensions(G4Box&            trackerLayer,
                     const G4int           copyNo,
                     const G4VPhysicalVolume *physVol) const;
  ...
}
```

Source listing 4.1.3
一个不同尺寸的参数化 box 的例子。

这些方法完成如下工作：These methods works as follows:

为了将 parameterised volume 实例化，需要使用相应的复制号调用 ComputeTransformation 方法。程序将计算这个拷贝的变换矩阵，并设置相应的物理体使用这个变换：

```
void ExN02ChamberParameterisation::ComputeTransformation
(const G4int copyNo,G4VPhysicalVolume *physVol) const
{
  G4double     Zposition= fStartZ + copyNo * fSpacing;
  G4ThreeVector origin(0,0,Zposition);
  physVol->SetTranslation(origin);
  physVol->SetRotation(0);
}
```

注意：在这里给出的平移和旋转是针对坐标系的（被动模式）。它们不是针对主动模式的，在主动模式中，实体相对于母体坐标系旋转。

类似地， ComputeDimensions 方法被用于设置拷贝地尺寸。

```
void ExN02ChamberParameterisation::ComputeDimensions
(G4Box & trackerChamber, const G4int copyNo,
 const G4VPhysicalVolume * physVol) const
{
  G4double  halfLength= fHalfLengthFirst + (copyNo-1) * fHalfLengthIncr;
  trackerChamber.SetXHalfLength(halfLength);
  trackerChamber.SetYHalfLength(halfLength);
  trackerChamber.SetZHalfLength(fHalfWidth);
}
```

用户必须保证这个方法第一个参数的类型（在这个例子中是 G4Box &) ，对应于用户给定的参数化物理体所对应的逻辑体的对象类型。

更高级的使用允许用户

- 通过建立一个 ComputeSolid 方法改变实体类型，或者
- 通过创建一个 ComputeMaterial 方法改变逻辑体的材料

来进行他所需的参数化过程。

注意：目前多数情况下，不可以给一个 parameterised volume 添加子体。只有在那些 parameterised volumes 的所有实体都具有相同的尺寸时，才可以添加子体。当实体的尺寸或者类型发生变化时，不能再添加相应的子体。

所以， parameterised volumes 的功能，只有在“叶子”volumes（这些 volume 不包含其它的 volumes）的情况下，才能全部发挥。

4.1.5 Touchables:唯一的标识一个 volume（Touchables: Uniquely identifying a volume）

Touchables 介绍

一个 Touchable Volume 用于向一个探测器单元提供一个唯一的标识。它可以有效的用于几何描述，用来替换在 G4 粒子跟踪系统中使用的相应描述，例如一个基于灵敏探测器的读出几何，或者一个用于快速 Monte Carlo 的参数化几何。要建立一个 Touchable Volume，几个技术要被实现：例如，在 G4 中，Touchables 是作为与一个全局参考系中的变换矩阵相关的实体实现的，或者作为一个相对于几何结构树根节点的物理体层来实现的。

一个 touchable，是在探测器描述中，一个具有唯一位置的几何实体。它的抽像基类可以用各种方式实现。每种方式必须提供获取用于 touchable 描述的变换和实体的能力。

Touchable 可以做什么？

所有的 `G4VTouchable` 实现必须响应下面的两个"请求"：

1. `GetTranslation` 和 `GetRotation` ，返回 volume 的变换

另外的功能是与更多的信息有关的实现，它们有一个缺省的实现，这个实现引起一个异常。

Touchable 有几个与物理体有关的功能：

2. `GetSolid` 给出与 touchable 有关的实体。
3. `GetVolume` 给出相应的物理体。
4. `GetReplicaNumber` 如果是 Repeated volumes，给出相应物理体的副本号。

保存 volume 层次的 Touchables 有一个完整的母体栈。因此，为了扩展它的功能，用户可以增加一些状态。这里增加了一个指向这个栈中各层的“指针”和一个用于在这个栈中移动到相应层的成员函数。然后针对某一层，调用上述的成员函数，则将返回该层的有关信息。

层次树的顶端，约定是 world 体。

5. `GetHistoryDepth` 给出这个层次树的深度。
6. `MoveUpHistory( num )` 在 touchable 内移动向层次树的上端移动当前指针，移动 `num` 层。因此，当 `num=1` 时调用该方法，将使内部指针指向当前 volume 的母体。
   警告：这个函数改变 touchable 的状态，而且，如果将它应用到 Pre/Post step touchables ，在粒子跟踪过程中，可能会引起错误。

一个有不同参数的 update 方法可用，所以，在一个 touchable 中的信息可以被刷新：

7. `UpdateYourself` 接受一个物理体指针，启动一次 `NavigationHistory`。

Touchable history 保存了几何数据栈

如在 4.1.3 节和 4.1.4 节所述，一个逻辑体表示没有确定位置的探测器单元，而一个物理体可以表示多个探测器单元。另一方面，touchable 为一个探测器单元提供一个唯一标识。特别地，G4 输运过程和粒子跟踪系统采用 touchables，这些 touchables 就像在 `G4TouchableHistory` 中实现的一样。Touchable 层次结构是指定一个物理体的完整宗系（从该物理体，向上，一直到几何结构树的根）所要求信息的最小集合。这些 touchable volumes 可以在 G4 粒子跟踪的每一步中，被用户在 `G4VUserSteppingAction` 使用。

要建立/存取 `G4TouchableHistory` ，用户必须发消息给 Navigator

```
G4TouchableHistoryHandle CreateTouchableHistoryHandle() const;
```

它将返回一个指向 touchable 的句柄。

在这个类型中，不同于其它 touchables 的方法是：

```
G4int GetHistoryDepth()  const;
G4int MoveUpHistory( G4int num_levels = 1 );
```

第一个方法用于查询当前 volume 在几何树上的深度是多少层。第二个方法请求 touchable 估计它的最深的层。

注意， `MoveUpHistory` 改变一个 touchable 的状态。

4.1.6 创建组合逻辑体

`G4AssemblyVolume` 是一个辅助类，它允许在 3D 空间，用任何方式将几个逻辑体组合在一起。结果就是一个正常的逻辑体的放置，但最终的物理体是多个的。

一个*组合逻辑体* 跟真正的母体不一样，它只是它的子体的包络。在组合逻辑体的放置完成后，它的使命就完成了。产生的物理体是组成这个组合逻辑体的各个逻辑体的独立的拷贝。

在由不同形状组成的复杂元件的空间中，需要建立一个规则的图案，但是又不能通过 replicated volumes 或者 parametrised volumes (参看 figure 4.1.0)。为了避免放在同一一个母体内的物理体“增殖”，必须小心使用 `G4AssemblyVolume` 。

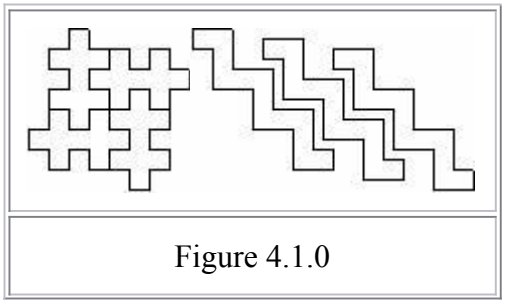


Figure 4.1.0

逻辑体组合的例子。

用它自身的“子体”填充组合逻辑体

相关的逻辑体被描述为一个三元组<logical volume, translation, rotation> (`G4AssemblyTriplet` class)。
根据指定的平移和旋转，相对组合逻辑体的坐标系放置每个相关的逻辑体。

组合逻辑体放置

一个组合逻辑体对象是由一系列的逻辑体组成的；在母逻辑体内部可以放置它的副本。

由于在每此复制过程中组合逻辑体类将生成物理体，用户没有方法给它们指定标识符。因此，通过调用 `MakeImprint(...)`方法建立物理体的时候，有一个内部计数器用于产生唯一的物理体名。
生成的每个物理体的名字有如下格式：

av_**WWW**_impr_**XXX_YYY_ZZZ**

这里：

- **WWW** – 组合逻辑体类的实例号
- **XXX** – 组合逻辑体的复制号
- **YYY** – 被放置的逻辑体的名字
- **ZZZ** – 在组合逻辑体内部的逻辑体的索引号

一个组合逻辑体的析构

在析构过程中，所有在生成的物理体和与这些副本相关的旋转矩阵将被删除。为了能够在不需要的时候，清除那些对象，通过 `MakeImprint()`方法建立的物理体的列表将被保存。这要求用户在整个任务的运行过程中，或者在 `G4Navigator` 的存活时间内，保存这些组合逻辑体对象，逻辑体存储器和物理体存储器可以保存指向由组合逻辑体生成的物理体的指针。
`MakeImprint()` 方法只有使用简单的变换时才有正确的结果，向普通逻辑体的放置一样，反射是不允许的。
在 `G4AssemblyVolume` 析构的时候，所有它生成的物理体和旋转矩阵将被删除，相应的内存将被释放。

例子

这个例子演示如何使用 `G4AssemblyVolume` 类。它实现了一个多层探测器，每层由 4 个平板组成。

在下面的代码中，首先定义了 world 体，然后建立了那个平板的实体和逻辑体，接着，定义了组成那些层的组合逻辑体。
以在一个母体内放置普通的物理体的同样方式，组成各层的组合逻辑体被那些平板所填充。
最后，就像复制这些组合逻辑体一样，那些层被放置在 world 中。（参看源码清单 4.1.4）

```
 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;

   // 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. );
   assemblyDetector->AddPlacedVolume( plateLV, G4Transform3D(Ta,Ra) );

   Ta.setX( -1*caloX/4. ); Ta.setY( caloY/4. ); Ta.setZ( 0. );
   assemblyDetector->AddPlacedVolume( plateLV, G4Transform3D(Ta,Ra) );

   Ta.setX( -1*caloX/4. ); Ta.setY( -1*caloY/4. ); Ta.setZ( 0. );
   assemblyDetector->AddPlacedVolume( plateLV, G4Transform3D(Ta,Ra) );

   Ta.setX( caloX/4. ); Ta.setY( -1*caloY/4. ); Ta.setZ( 0. );
   assemblyDetector->AddPlacedVolume( plateLV, G4Transform3D(Ta,Ra) );

   // 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 );
```

```
    assemblyDetector->MakeImprint( worldLV, G4Transform3D(Tm,Rm) );
  }
}
```

代码清单 4.1.4
使用 G4AssemblyVolume 类的例子

最终生成的探测器如下 figure 4.1.1：

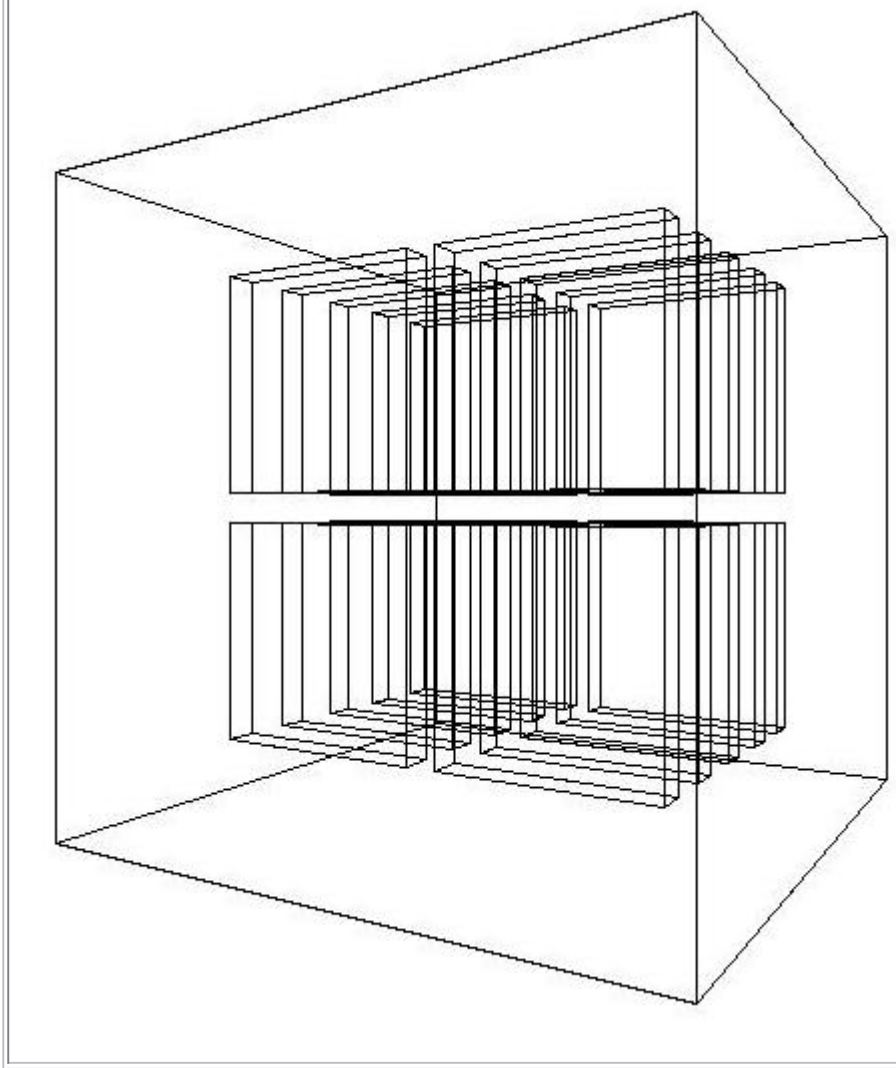


Figure 4.1.1
代码清单 4.1.4 所对应的几何结构

4.1.7 volumes 层次的镜像（Reflecting hierarchies of volumes）

基于 CSG 或特殊实体的放置的层次结构可以通过 G4ReflectionFactory 类和 G4ReflectedSolid 来生成它的镜像，它使一个实体从初始坐标系中变换到新的“镜像”坐标系中。镜像变换常被分解为平移和旋转变换。
factory 是一个 singleton 对象，它提供了如下的方法：

```
G4PhysicalVolumesPair Place(const G4Transform3D&   transform3D,
                        const G4String&       name,
                              G4LogicalVolume* LV,
                              G4LogicalVolume* motherLV,
                              G4bool         isMany,
                              G4int          copyNo)

G4PhysicalVolumesPair Replicate(const G4String&      name,
                                G4LogicalVolume* LV,
                                G4LogicalVolume* motherLV,
                                EAxis          axis,
                                G4int          nofReplicas,
                                G4double        width,
                                G4double        offset=0)
```

方法 Place() 用于逻辑体的放置，评价已传递的变换；在包含镜像变换的情况下，factory 将进行如下操作：

1. 执行变换分解。
2. 建立一个新的镜像实体和逻辑体，如果他们已经存在，那么直接获取它们。
3. 将所有子体进行变换并放置在给定母体中。

如果操作成功，结果将返回一对物理体，这里的第二个物理体是放置在镜像母体内的。
方法 Replicate() 在给定母体内创建副本。如果操作成功，结果返回一对物理体，这里的第二个物理体是在镜像母体内的副本。

```
#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

// 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
```

代码清单 4.1.5
使用 `G4ReflectionFactory` 类的例子。

4.1.8 几何 navigator（The geometry navigator）

在粒子跟踪过程中，对整个几何的 navigation 是由类 `G4Navigator` 完成的。Navigator 用于在几何体内定位和计算到几何体边界的距离的。在粒子跟踪过程中，navigator 只定位那些与粒子跟踪有关作用点。
G4Navigator 有几个有用的私有类：

- **G4NavigationHistory** – 存储复合的变换，复制/参数化信息，和相对于当前位置，在层次结构中每一层的 volume 指针。在每一层的 volume 类型也被存储，这些类型描述是否是正常放置的，是否是复制的，或者是否是参数化的。
- **G4NormalNavigation** – 为包含“已放置”物理体，但不使用体素的几何体提供位置和距离计算函数。

- **G4VoxelNavigation** – 为包含“已放置”物理体，同时使用体素的几何体提供位置和距离计算函数。在它内部，有一个体素信息栈。提供一些私有函数用来计算到体素各边界的距离，和到指定方向相邻体素的距离。
- **G4ParameterisedNavigation** –为包含 parameterised volumes，同时使用体素的几何体提供位置和距离的计算。体素信息的维护与 `G4VoxelNavigation` 类似，但是同采用体素，计算将会更简单，因为层次将降至一层（一维优化）。
- **G4ReplicaNavigation** – 为复制体提供位置和距离的计算函数。

另外，navigator 还有一系列用于 exiting/entry 优化的标志。Navigator 不是 singleton 类，这允许将来对设计进行扩展（例如 几何事件偏倚）。就目前来说，不应同时使用两个 navigator 对象。

Navigation 和粒子跟踪

在几何体中进行粒子跟踪所要求的主要函数如下。另外的函数是用于提供返回 volumes 的有效变换，和用于 touchables 的建立。没有函数隐式的要求那些几何体必须是层次结构的。

- **SetWorldVolume()**
  设置层次结构中的第一个逻辑体。它对于坐标原点必须是非旋转、不平移的。
- **LocateGlobalPointAndSetup()**
  定位包含指定全局点的 volume。这涉及到层次间的穿越，要求计算复合变换，测试复制体和 parameterised volumes 等。为了提高效率，本次搜索是与前次相关的，这是推荐的方式。当函数第一次被调用的时候，可以执行“相关”搜索，搜索将从层次结构的根节点开始。搜索也可以使用 `G4TouchableHistory` 。
- **LocateGlobalPointAndUpdateTouchableHandle()**
  首先，像上面的 `LocateGlobalPointAndSetup()`方法一样搜索几何的层次结构。然后，使用 volume found 和 navigation history 刷新 touchable。
- **ComputeStep()**
  计算从指定点，沿指定方向，到相交的边界的距离。在调用 `ComputeStep()`之前，该指定点必须先定位。
  当调用 `ComputeStep()`的时候，将传递一个指定的物理步长。如果可以确定该点到第一个交点的距离等于或大于物理步长，那么将返回 `kInfinity` 。在任何情况下，如果返回的步长大于物理步长，将使用物理步长。
- **SetGeometricallyLimitedStep()**
  通知 navigator 使用上次计算的步长，这将进行 entering/exiting 优化，而且需要在调用 `LocateGlobalPointAndSetup()`之前调用该方法。
- **CreateTouchableHistory()**
  建立一个 `G4TouchableHistory` 对像，调用者有删除该对象的义务。该'touchable' volume 是上次定位操作返回的 volume。该对象包含一个当前 NavigationHistory 的副本，是那些在当前 volume 内部、或者靠近当前 volume 的点可以有效的重定位。

如前所述，navigator 使用那些辅助类进行定位和步长计算。有不同的 navigation 工具对 `G4NavigationHistory` 对象进行操作。
在 `LocateGlobalPointAndSetup()` 中，定位一个点的过程分为三个主要步骤－－优化，确定包含这个点的子树（母体和子体），确定确切包含该点的子体；后两个步骤，首先，在层次结构树上向上扫描，直到包含该点的 volume 被找到，然后，向下扫描，直到包含该点的

确切的 volume 被找到。
在 `ComputeStep()` 中，根据当前 volume 所包含的子体的不同，使用了三种不同的计算方法：

- 包含正常放置的子体，或者没有子体的 volume
- 包含用于表示多个 volumes 的单个 parameterised volume 对像
- volume 是一个副本，并且包含正常放置的子体

4.1.9 一个简单的几何编辑器

GGE 是 G4 图形界面几何编辑器（Geant4 Graphical Geometry Editor）。它是用 JAVA 实现的，是 Momo 的一部分。GGE 是面向那些有一些 C++和 G4 工具包的知识的物理工作者的，使他们可以在图形环境下，方便的构造自己的探测器几何。

GGE 提供如下方法：

1. 构造一个探测器几何，包括 `G4Element`, `G4Material`, `G4Solids`, `G4LogicalVolume`, `G4PVPlacement`,等。
2. 用现有的可视化工具（如，DAWN）流览探测器几何
3. keep the detector object in a persistent way
4. 按照 G4 工具包的标准生成 C++代码
5. 生成 G4 可执行程序

GGE 是由 Java 实现的，使用了 Java 基本类库，Swing-1.0.2。本质上，GGE 是一个参数表的集合，这些参数表包含了构造一个简单的探测器几何所需要的所有相关参数。
有关 GGE 的软件，安装指南和注意，以及其它基于 JAVA 的用户接口工具可以从日本的 Naruto University of Education 的网站 Geant4 GUI and Environments web site 下载。

材料：元素和混合物

GGE 以表格的形式提供了周期表内元素的数据库，用户可以使用这些元素构造新的材料。GGE 还提供了一个预构造的材料数据库，这些数据来自于现有的 PDG 数据。它们可以被加载、使用、编辑，可以保存为 persistent 对像。

Users can also create new materials either from scratch or by combining other materials.

- creating a material from scratch:

Use	Name	A	Z	Density	Unit	State	Temperature	Unit	Pressure	Unit

- 只有在逻辑体内使用的元素和材料才被保存在探测器对象中，并且被用来生成 C++构造函数。使用标记标识那些被使用的材料。
- 该构造函数通过元素化合建立材料，使用 `AddElement` 加入这些元素 。

Use	Name	Elements	Density	Unit	State	Temperature	Unit	Pressure	Unit

- 用鼠标点击 **Elements** 列，将打开一个窗口，用于选择下面的两个方法：
  - 加入一种元素，给定质量分数
  - 加入一种元素，给定原子数。

实体

目前，支持最常用的 CSG 实体 (`G4Box`, `G4Tubs`, `G4Cons`, `G4Trd`) 和特殊的 BREPs 实体(Pcons, Pgons)。这些实体的所有相关参数都可以在一个小窗口内指定。

用户可以使用 DAWN 观查每一个实体。

逻辑体

GGE 可以指定下面这些项：

Name	Solid	Material	VisAttribute

The construction and assignment of appropriate entities for `G4FieldManager` and `G4VSensitiveDetector` are left to the user.

物理体

一个物理体的单个拷贝可以被建立。同样也可以以多种方式建立复制体。首先，用户可以线性平移逻辑体。

Name	LogicalVolume	MotherVolume	Many	X0, Y0, Z0	Direction	StepSize	Unit	CopyNumber

组合平移和旋转，将一个物体重复的按“柱形”图样放置。GGE 同时实现了副本和 parametrised volume 的简单模型。在副本情况下，一个 volume 被分割成新的子体。在 parametrised volumes 情况下，目前已可以创建几种图样 。

C++代码生成：`MyDetectorConstruction.cc`

最后，生成一个 include 文件和一个源文件的源代码表。它们被称作 `MyDetectorConstruction.cc` 和`.hh` 文件。它们实时反应用户的修改。

可视化

在 DAWN 的帮助下，可以查看独立的实体。整个几何的可视化要在 `MyDetectorConstruction.cc` 编译之后才能实现。

4.1.10 从 CAD 系统导入实体模型

我们可以从一个 CAD 系统导入什么？

G4 提供了一个 CAD 系统的接口模块，用于导入按 STEP AP203 协议描述的实体。这些模块有一些已知的限制（参看下节），它们在下一个版本的工具包中将不在支持，将用类似的工具替代。在 G4 内核中的这些模块需要在安装的时候设置环境变量 `G4LIB_BUILD_STEP`，才会被编译，缺省情况是不会被编译的。

被导入的实体模型必须是由兼容 STEP 的 CAD 系统描述的。这些模型可以是由大量元件组成的探测器几何实体，它们有很高的精度和详细的细节。一个实体模型包描述实体的纯几何数据，和在给定参照系中的位置信息。

它不包含与 volume 相关的材料或者层次信息。这些额外的信息很容易添加到从 CAD 模型导入的实体中，可以在探测器描述中可以直接进行物理模拟。

如何导入一个实体模型？

G4 所需要的是来自 CAD 系统的 STEP AP203 文档。模型 CAD 系统（如，Pro/Engineer）可以输出兼容 STEP 的文档。其余的（如，Euclid）使用第三方软件实现这个功能。`STEPinterface` 接口模块中的 `G4AssemblyCreator` 和 `G4Assembly` 类用于读取由 CAD 系统生成的 STEP 文档，并且在 G4 中建立组合几何体。下面的步骤，将为 G4 中的 NIST STEP reader 创建的那些实体，建立或者关联与逻辑体、物理体，材料等有关的信息。有一个简单的例子描述了如何初始化从 STEP 文件 *tracker.stp* 中读取的所有可用的 entities：

```
G4AssemblyCreator MyAC("tracker.stp");
  // Associate a creator to a given STEP file.
MyAC.ReadStepFile();
  // Reads the STEP file.
STEPentity* ent=0;
  // No predefined STEP entity in this example. A dummy pointer is used.
MyAC.CreateG4Geometry(*ent);
  // Generates GEANT4 geometry objects.

void *pl =  MyAC.GetCreatedObject();
  // Retrieve vector of placed entities.
G4Assembly* assembly = new G4Assembly();
  // An assembly is an aggregation of placed entities.
assembly->SetPlacedVector(*(G4PlacedVector*)pl);
  // Initialise the assembly.

G4int solids = assembly->GetNumberOfSolids();
  // Get the total number of solids among all entities.
```

```
  for(G4int c=0; c<solids; c++)
    // Generate logical volumes and placements for each defined solid.
    {
      ps = assembly->GetPlacedSolid(c);
      G4LogicalVolume* lv = new G4LogicalVolume(ps->GetSolid(), Lead, "STEPlog");
      G4RotationMatrix* hr = ps->GetRotation();
      G4ThreeVector* tr = ps->GetTranslation();
      G4VPhysicalVolume* pv = new G4PVPlacement(hr, *tr, ps->GetSolid()-
>GetName(), lv,
                                          experimentalHall_phys, false, c);
    }
```

在 `geant4/examples/extended/geometry/cad` 有一个例子，显示了导入一些 STEP AP203 例子文件的能力。

创建逻辑体和物理体

根据 STEP 几何描述产生逻辑体和物理体，推荐的方法是遵循上面例举的模板，产用相似的 C++模块来实现，这里，每个探测器组件几何在个自独立的 STEP 文件中描述。

已知的限制

- G4 开发小组对 G4 工具包中的 NIST STEP reader 的维护不负责。因此，G4 小组不能保证 NIST STEP reader 模块（它是 NIST STEP 类库的一部分[1]）的改进。
  目前，根据该软件的原始发行文档，它们只能保证基于旧的非 ISO/ANSI 编译器时的结果是正确的。
- 目前，除了使用光线跟踪技术之外，从 CAD 系统导入的模块，在 G4 中实现可视化是不可能的。
- 在许多情况下，由 CAD 系统生成的 AP203 STEP 文档，并不完全兼容 NIST STEP reader。G4 提供了两个 `perl` 脚本用于将它们转换为正确的格式。这些脚本 (`g4step_correct.pl` 和 `g4sort_step_file.pl`) 位于 `geant4/examples/extended/geometry/cad/scripts`；它们用来转换那些产生读取错误的 AP203 STEP 文档。
- 只有有限的一些 STEP `entities` 可以用来转换为 Geant4 BREPS 形状。对那些不支持的 entities，Geant4 STEP 接口将产生一个警告作为标记，并且不生成相应的对象。
- 不支持从 CAD 系统导出的，基于 NURBS 的形状。因此，这些形状将不会被转换。
- 将一个实体模型导出到一个 CAD 系统的特性目前还没有实现。

4.1.11 转换 GEANT 3.21 的几何

Approach

G3toG4 是用于将 GEANT 3.21 几何转换为 G4 几何的工具。它完成以下两步：

1. 用户提供一个包含初始化数据结构的 GEANT 3.21 RZ-文档 (.rz) ， rztog4 读取该文档，并产生一个 *call list* 文件，这个文件包含了如何建立几何的指令。rztog4 是用 FORTRAN 写的。
2. call list 解释器 (G4BuildGeom.cc) 读取这些指令，并且为 G4 在用户的客户代码中建立几何。

Importing converted geometries into Geant4

在 examples/extended/g3tog4 中有两个如何使用 call list 解释器的例子：

1. cltog4 是一个简单的例子，它只是简单的从 G3toG4DetectorConstruction 类中调用 call list 解释器 G4BuildGeom ，建立几何，然后退出。
2. clGeometry 是一个更为完整的例子，作为 G4 入门的例子。它同样调用 call list 解释器，另外，还允许可视化这些几何，进行粒子跟踪。

要建立和编译 G3toG4 库，需要在安装的时候设置环境变量 G4LIB_BUILD_G3TOG4 。缺省情况下， G3toG4 库不会被编译。然后，在 source/g3tog4 目录输入

```
gmake
```

要建立可执行的 rztog4，只要输入

```
gmake bin
```

生成所有的相关程序和库，输入：

```
gmake global
```

删除所有 G3toG4 库，可执行文件和.d 文档，输入

```
gmake clean
```

目前状态

软件包已经经过各种实验几何的检验，如： BaBar, CMS, Atlas, Alice, Zeus, L3, 和 Opal.
下面是目前版本所支持和不支持的特性的列表：

- 支持形状：除 GTRA, CTUB 外的所有 GEANT 3.21 形状。
- 用特殊实体 G4Polycone 和 G4Polyhedra 建立的 PGON, PCON
- 部分支持 GEANT 3.21 的 MANY 特性。
  MANY 被用 G3toG4MANY() 函数转换，必须在 G3toG4BuildTree() 之前处理。(缺省条件下不被调用).
  为了转换 MANY, 用户还必须使用 G4gsbool(G4String volName, G4String manyVolName)为所有重叠的 volume 提供额外的信息。重叠 volume 的子体将被自动转换，并且不因该由 Gsbool 指定。
  限制：具有 MANY 特性的 volume 只能有一个位置；如果需要使用多个位置，那么必须为每个位置都定义(gsvolu())一个新的 volume。
- GSDV* routines for dividing volumes are implemented, using G4PVReplicas, 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 ，通过 G3Division 类，实现了基于简单 CSG 实体的层次镜像。
- Hits 没有被实现。
- 磁场类的使用必须打开。

4.1.12 检测重叠 Volume

重叠 volumes 的问题

经常将一些 volumes 放在另外一些 volumes 内部，目的是为了完全填充后者。当一个在内部的 volume 升出它的母体的时候，它就被认为发生了重叠。Volumes 也经常被放置在同一个 volume 里面，它们之间不相交。当在一个母体内的 volumes 相交的时候，也认为发生了重叠。

检测 volumes 间重叠的问题与实体模型描述负责性密切是相关的。通常，它与描述最负责的实体拓扑逻辑的数学负责性相当。但是，通过近似实体，使用一次或者二次表面近似来检测它们是否相交，可以获得一个可调的精度。

检测重叠：内建命令

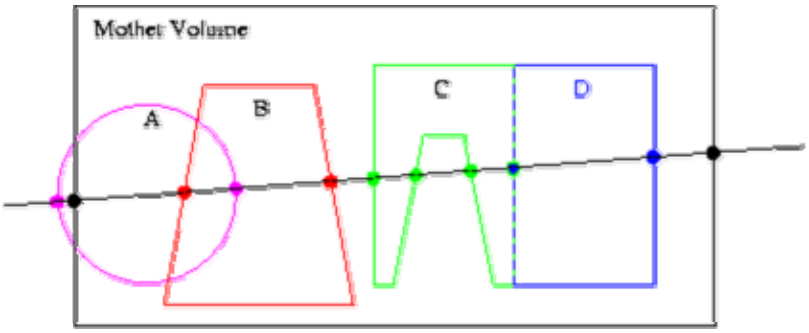
通常，功能最强大的碰撞检测算法是那些 CAD 系统提供的算法，它们根据实体的拓扑形式来处理实体之间的相交。

G4 提供了一些内建命令来为用户定义的几何进行校验测试。：

```
geometry/test/grid_test [recursion_flag]
--> 基于标准栅格线设置，为重叠区域进行几何校验。
    如果递归标志"recursion_flag"被设置为'false' (缺省),
    检测将被限制在几何树的第一级深度；否则，它将递归检测整个几何树。
     后者，将可能花费很长的时间，这与几何的负责程度有关。
geometry/test/cylinder_test [recursion_flag]
--> 根据一个柱形图样发出直线。
    如果递归标志"recursion_flag"被设置为'false' (缺省),
    检测将被限制在几何树的第一级深度；否则，它将递归检测整个几何树。
     后者，将可能花费很长的时间，这与几何的负责程度有关。
geometry/test/line_test [recursion_flag]
--> 根据给定的方向和位置发出一条直线。
    如果递归标志"recursion_flag"被设置为'false' (缺省),
    检测将被限制在几何树的第一级深度；否则，它将递归检测整个几何树。
     后者，将可能花费很长的时间，这与几何的负责程度有关。
geometry/test/position
```

--> 为 line_test 指定位置。
geometry/test/direction
--> 为 line_test 指定方向。
geometry/test/grid_cells
--> 在栅格测试中，在各个方向，X/Y/Z，以指定 cell 数来定义线的分辨率。
新的设置将被应用于 grid_test 命令。
geometry/test/cylinder_geometry
--> 定义柱形几何体的细节，指定：
nPhi – 单位方位角 Phi 的直线数
nZ - Z 方向点数
nRho – 径向点数
新的设置将被应用于 cylinder_test 命令。
geometry/test/cylinder_scaleZ
--> 定义柱形几何体的分辨率，指定 Z 方向点的尺度。
新的设置将被应用于 cylinder_test 命令。
geometry/test/cylinder_scaleRho
--> 定义柱形几何体的分辨率，指定沿径向 Rho 的点的尺度。
新的设置将被应用于 cylinder_test 命令。
geometry/test/recursion_start
--> 为开始递归设置几何树的初始层(缺省值为 0，表示 world)。
新的设置将被应用于任何递归检测。
geometry/test/recursion_depth
--> 为递归设置几何树深度，当到达指定深度后，将使递归停止（缺省深度是整个几何树）。
新的设置将被应用于任何递归检测。

内建的检测程序，使用线性轨迹与实体的交点来检测重叠 volumes。例如，考虑下面的情况：



这里我们有一条直线与一些物理体相交。图中黑色的矩形是母体，它有四个子体：A，B, C，D。图中的那些点表示直线与母体及四个子体的交点。

这个例子中有两个几何错误。首先，实体 A 升出了母体（这个练习，可以在 GEANT3.21 中使用，不允许在 G4 中使用）。可以注意到，它的一个交点 (最左边洋红色的点)位于母体外，而不是在两个黑点之间。

第二个错误是两个子体 A 和 B 发生交叠。这也是很明显的，A 的一个交点（右边的洋红色点）在 B 内部，即在两个红点之间。换句话说，B 的一个交点（左边的红点）在 A 的内部，即在两个洋红点之间。

这两种类型的错误都用一段线段表示，它有一个起点，一个终点，和长度。根据错误的类型，这些点可以在发生错误的 volume 的坐标系，全局坐标系，或者相关子体的坐标系被清晰的识别。

也需要注意，某些错误也可能不能被发现，除非这些错误刚好在直线穿过的路径上。不幸的是，要预言刚好穿过潜在的几何错误的直线是非常困难的。因而，几何测试使用了栅格线，希望至少可以发现那些比较大的几何错误。许多微小的错误可能被忽略。

另一个困难的问题是舍入错误。例如，子体 C 和 D 紧密相邻。可能出现这样的情况，由于舍入的存在，那些相交点中的某个可能刚好进入了另一个子体内部。另外，当一个 volume 紧挨着母体的时候，可能会有交点刚好落到了母体外。

为了避免舍入引起的错误，缺省情况下，使用 0.1 微米的公差。这个公差可以使用运行时命令，由应用程序进行调整：

```
geometry/test/tolerance <new-value>
```

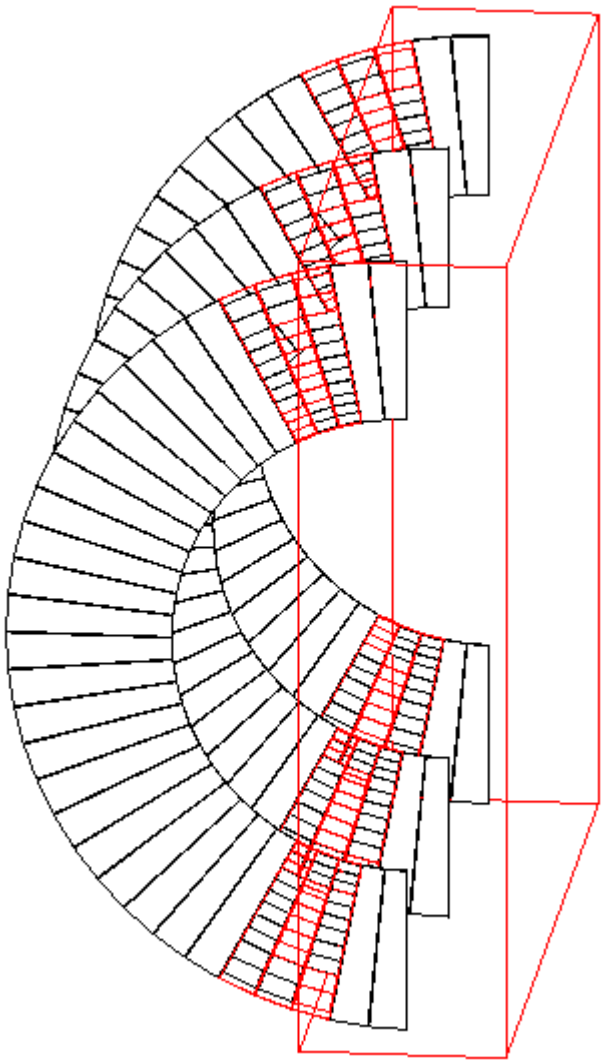
最后，注意在此没有提到的，但可能存在的 A,B,C,D 的子体。为了使代码简单，一次检测只检查一个 volume 的直接子体。因此，要检测那些“孙子”，那些子体必须轮流检测它们自己。为了使这一切变得更加自动化，使用了一个可选的递归算法；它首先检测一个目标 volume，然后进入子体循环并调用自身。

注意！对于一个复杂几何，检测整个 volume 层次结构可能花费大量的时间。

使用可视化引擎：DAVID.

G4 可视化提供了一个强大的调试工具用于检测潜在的物理体相交错误。G4 DAVID 可视化工具[3] 可以自动检测 volumes 之间的交叠，并且将结果用图形表示。图形表示的精度可以被调到精确的表示几何描述。在调试过程中，物理体的表面被自动分解为 3D 多边形，检测产生的多边形是否相交。如果一个多边形和另一个相交，与这些多边形有关的物理体将被用彩色显示（缺省是红色）。下图是一个探测器几何可视化的例子，图中相交的物理体凸出显

式：



目前，由下列实体组成的物理体可以进行这样的调试： `G4Box`, `G4Cons`, `G4Para`, `G4Sphere`, `G4Trd`, `G4Trap`, `G4Tubs`. (其它实体的存在不影响这些实体的调试。)

进行物理体表面的可视化调试，需要使用在可视化类中定义的 DAWNFILE 引擎，和其它两个应用程序包，就是 Fukui Renderer "DAWN" 和可视化相交调试程序"DAVID"。 DAWN [2] 和 DAVID [3] 可以从网上下载。

在 8.6 节中，描述了如何将 DAWNFILE 引擎和 G4 一起编译。

如果 DAWNFILE 引擎，DAWN 和 DAVID 已经可以正常运行，那么，就可以进行物理体表面的可视化相交调试了：

设置环境变量 G4DAWNFILE_VIEWER 为 "david":

```
% setenv G4DAWNFILE_VIEWER   david
```

这个设置使 DAWNFILE 引擎调用 DAVID，而不是调用缺省流览器，DAWN。

运行用户的 G4 可执行程序，调用 DAWNFILE 引擎，执行可视化命令显式用户的探测器几何：

```
Idle> /vis/open DAWNFILE
.....(设置相机等)...
Idle> /vis/drawVolume
Idle> /vis/viewer/update
```

接着在当前目录下产生一个叫"g4.prim"的文件，它描述了探测器几何，DAVID 将读取这个文件。（g4.prim 的格式在下面的网页上有相关描述：http://geant4.kek.jp/~tanaka/DAWN/G4PRIM_FORMAT_24/。）

如果使用 DAVID 检测到物理体表面的相交，它将自动调用 DAWN 显式探测器几何，那些相交的物理体将被凸出显示(参看上面的例子)。

如果没有检测到相交，那么不显式几何，只在控制台显式如下信息：

```
-----------------------------------------------------
!!! Number of intersected volumes : 0 !!!
!!! Congratulations ! \(^o^)/       !!!
-----------------------------------------------------
```

如果用户不希望显式几何，需要预先设置如下环境变量：

```
%  setenv DAVID_NO_VIEW  1
```

要控制与相交计算相关的精度（缺省设置为 9），可以使用 DAWNFILE 图形引擎的环境变量：

```
%  setenv G4DAWNFILE_PRECISION  10
```

在当前目录下生成了一个文件叫"g4david.log"，它描述了相交的详细信息。下面是 g4david.log 的一个例子：

```
.....
!!! INTERSECTED VOLUMES !!!
caloPhys.0: Tubs: line 17
caloPhys.1: Tubs: line 25
.....
```

在本例中，第一栏告诉我们一个物理体的名字为"caloPhys"，它的复制号为"0"，它与另外一个物理体相交，另外一个物理体与它同名，但是复制号为"1"。第二栏显示这

些物理体的形状，它们是由类 G4Tubs 定义的。第三栏显示了相交的物理体在文件 `g4.prim` 中的行号。

如果需要的话，可以凸出显式相交部分，重新显示探测器几何。这些数据被保存当前目录下的"g4david.prim" 中。这个文件可以被 DAWN 用来重复的显式结果，命令如下：

```
% dawn g4david.prim
```

可以将产生的 g4david.prim 文档转换为 VRML 格式的文档，这是非常有用的，用户可以用 WWW 浏览器对 VRML 文档进行交乎式的可视化。文档转换工具可以从下面的网址获得：
http://geant4.kek.jp/~tanaka/DAWN/About_prim2vrml1.html

更多的细节，请参看上面提到的 DAVID 的文档 。

使用几何调试工具 OLAP。

OLAP 是 CERN 在 CMS 实验中开发的一个工具，用来识别探测器几何中的交叠。它被放在特殊工具/例子里面，在目录 `geant4/examples/extended/geometry` 下。它首先在两个相反的方向发射 `geantinos` 粒子，然后校验闯过边界的粒子是否相等。
如果用户的几何可以作为 `G4VUserDetectorConstruction` 的子类进行调试并且用于构造这个工具的 `OlapDetConstr` 类，那么这个工具可以用于任何 G4 几何。工具本身还需要一个 dummy 类 `RandomDetector`。
工具提供了运行时命令来 navigate 几何树。提供了类 UNIX 的命令`/olap/cd` ，用于逻辑体层次的 navigation。逻辑体树的根可以通过字符'/'来存取。Volume 树上的任何节点都可以使用'/'分隔的正则表达式串来存取。如果'/'在字符串开头，那么表达式表示的层次是从根开始的，否则就是从当前选择的逻辑体开始。另一个命令`/olap/goto [regexp]` 可以用来跳转到与表达式`[regexp]`相匹配的第一个逻辑体。can be used to jump to the first logical volume matching the expression `[regexp]`. 每个成功的 navigation 命令 (`/olap/cd`, `olap/goto`)都导致建立一个 `NewWorld`，母体为命令的参数，子体为该母体的直接子体。
`/olap/pwd` 显式当前 `NewWorld` 和母体在整个几何层次树中的位置。
更详细的信息，请阅读该工具提供的 `README` 文档。

4.1.13 使用 GDML 导入 XML 模型

从 G4 4.1 版开始，在 `examples/extended/gdml` 目录下有一个例子，讲述了如何导入基于 **GDML** [4] 的探测器描述。[5]是一个关于如何用 GDML 描述几何的例子，在里面附了各种注解。GDML(Geometry Description Markup Language) [4] 是基于 XML 的一种标识语言，用于探测器几何模型的描述。由于居于“易读”的 XML 描述和结构化的格式，使得可以用来方便的交换几何数据。

---

[1] http://ats.nist.gov/scl

[2] http://geant4.kek.jp/~tanaka/DAWN/About_DAWN.html
[3] http://geant4.kek.jp/~tanaka/DAWN/About_DAVID.html
[4] http://cern.ch/gdml
[5] http://cern.ch/gdml/doc/g4gdml_howto.html

About the authors
Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Detector Definition and Response

4.2 材料

4.2.1 通常情形

在自然界中，普通的材料(化合物和混合物)是由元素组成的，元素又是由同位素组成的，因此，在 G4 中设计了三个主要的类。每个类都有一个表作为静态数据成员，用于登记各个类的实例的创建。

G4Isotope
: 这个类描述原子的属性：原子序数，核子数，摩尔质量，等。

G4Element
: 这个类描述原子的属性：有效原子序数，有效核子数，有效摩尔质量，同位素数目，壳层能量，反应截面，等。

G4Material
: 这个类描述物质的宏观特性：密度，状态，温度，压强，和辐射长度，平均自由程，dE/dx 等宏观量。

G4Material 类对于工具包的其它部分都是可见的，可以用于粒子跟踪，几何，和物理。它包含了所有可能的元素及组成它的同位素的所有信息，同时隐藏它的实现细节。

4.2.2 相关类的介绍

G4Isotope

一个 *G4Isotope* 对像有一个名字，原子序数，核子数，摩尔质量，和在同位素表里的索引。构造函数自动在同位素表内存储"this"同位素，由此得到一个索引号。

G4Element

一个 *G4Element* 对像有一个名字，标识符号，有效原子序数，有效核子数，有效摩尔质量，在元素表里的索引，同位素数目，一个指向这些同位素的指针的向量，一个跟这些同位素相对丰度有关的向量（此处的相对丰度是指单位体积内的原子数）。另外，该类还提供了一些方法，用于逐个添加组成该元素的同位素。

通过提供有效原子序数，有效核子数，有效摩尔质量，用户可以显式的创建 *G4Element* 对像。用户也可以通过声明组成该元素的同位素数目来创建 *G4Element* 对像。构造函数将"new"一个指向 *G4Isotopes* 的指针的向量，和一个用于存储它们的相对丰度的双精度向量。最后，给出用户需要的同位素（已经存在）的地址和丰度，调用添加同位素的方法来添加它们。在最后一个同位素登记之后，系统将自动计算有效原子序数，有效核子数，和有效摩尔质量，并且在元素表中存储"this"元素。

在一个给定的元素中，有些量是常量，可能具有物理意义，也可能没有。它们被作为“派生类的数据成员”来计算和存储。

G4Material

G4Material 对像有一个名字，密度，物理状态，温度和压强（缺省是标准状态），元素数目和一个指向这些组分的指针的向量，一个表示每种组分质量比的向量，一个表示每种组分原子量（分子量）的向量，和一个在材料表中的索引。另外，该类还提供了一些方法，用于逐个相材料中添加它的组分。

如果用户需要显式的建立 *G4Material* 对像，可以直接提供创建该对像需要的最终的有效数字，根据这些数字将建立它的组分。当然，用户也可以通过声明组成材料的组分数目来建立 *G4Material* 对像。构造函数将"new"一个指向 *G4Element* 指针的向量，和一个用于存储质量比的双精度数的向量。在最后一个元素登记后，系统将自动计算单位体积内，各中组分的原子数、总的电子数，并且在材料表中存储"this"材料。用这种方式，我们可以用其它材料和元素组成新的混合物。

需要注意的是，如果用户用每种元素的原子数（分子数）来表示化合物，系统将自动计算各中元素的质量比。在给定的材料里面，有些量是常数，它们可能有物理意义，也可能没有，它们被作为"派生类的数据成员"来计算和存储。

最后注意　　这些类自动判断它们的质量比的和是否正确。这些类有一个数据成员用于保存一个固定的索引，原因是在物理过程中建立了大量的截面数据表和能力表，它们的每一行表示不同的材料（或者元素、或者同位素）。粒子跟踪过程将材料对像的地址传递给物理过程(当前 volume 的材料)。如果这个材料有一个索引，根据这个索引，用户可以直接存取已经建立的截面表。用户可以直接访问正确的行，在根据粒子能力，就可以知道要访问截面表的哪一列。如果没有索引，每次存取截面表或者能量表，都要经过搜索才能找到正确的材料所处的行。更详细的内容在物理过程那一章。

4.2.3 建立一种材料的所有方式

Source listing 4.2.1 例举了定义材料的不同方式。

```
#include <iostream.h>
#include "G4Isotope.hh"
#include "G4Element.hh"
#include "G4Material.hh"
#include "G4UnitsTable.hh"

int main() {

G4String name, symbol;            // a=摩尔质量;
G4double a, z, density;           // z=平均质子数;
G4int iz, n;                     //iz=一种同位素中的质子数;
                                // n=一种同位素中的核子数;

G4int ncomponents, natoms;
G4double abundance, fractionmass;
G4double temperature, pressure;

G4UnitDefinition::BuildUnitsTable();

//
// 定义材料
//

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);

//
// 通过相对丰度，由同位素定义元素
//

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);


cout << *(G4Isotope::GetIsotopeTable()) << endl;

cout << *(G4Element::GetElementTable()) << endl;

//
// 定义简单材料
//

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);

//
// 由元素定义材料。    情形 1: 化学分子
//

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);

//
//由元素定义材料。   情形 2：由质量比混合
//

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);

//
//由元素和/或其它材料定义材料（混合物）
//

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);

//
// 非标准状态下气体的例子
//

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.);

//
// 关于真空？ 真空是一种非常低密度的普通气体
//

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.);

//
// 打印材料表
//

G4cout << *(G4Material::GetMaterialTable()) << endl;

return EXIT_SUCCESS;
}
```

Source listing 4.2.1
例举定义材料的不同方式的例子。

就如后面的例子，一种材料有一种状态：固态（缺省），液态，气态。构造函数将检查材料的密度，如果低于一个给定阈值(10 mg/cm3)，将自动设置为气态。

在气态情况下，用户可以指定温度和压强。缺省是标准状态，它的定义在 `PhysicalConstants.hh` 中。

元素的核子数>=质子数 >= 1.

材料必须有密度，温度，压强这些参数。

4.2.4 材料表

打印一种组分

下面显示如何打印一种组分。

```
G4cout << elU << endl;
G4cout << Air << endl;
```

打印材料表

下面显示如何打印材料表。

```
G4cout << *(G4Material::GetMaterialTable()) << endl;
```

4.3 电磁场

4.3.1 粒子在场中传播

G4 可以描述粒子在各种场中的传播，磁场，电场及电磁场，均匀的或非均匀的。粒子在它们内部的传播可以按用户定义的精度进行。

为了跟踪在一个场中传播的粒子，需要对这个粒子的运动方程进行积分。通常，使用龙格-库塔法，对常微分方程进行积分。然而，对于一些特定情况，它的解析解是已知的。有几种龙格-库塔方法，分别用于不同情况。对于一些特殊情况（例如使用解析方法的均匀场），

可以使用多个不同的方法。另外，当一个近似的解析解已知的时候，就可以使用它进行迭代，直到收敛到要求的精度。目前已经实现了后面的这个方法，它特别适用于近均匀的磁场。

在一个特定的场中，一旦计算粒子传播路径的算法被选定，这个曲线路径将用许多弦来近似。这些弦用于向 Navigator 查询是否径迹穿越了 volume 的边界。有几个参数用于调准积分的精度和后续的几何查询的精度。

用一系列的弦，来近似一条弯曲的径迹，它的准确程度是由一个叫 *chord distance* 或 *miss distance* 的参数来控制的，它表示近似的直线径迹（弦）与实际的曲线径迹之间的最大距离。通过设置这个参数，用户可以控制几何查询的精度。对所有 volume 每一次的查询都保证了在 *miss distance* 精度内。

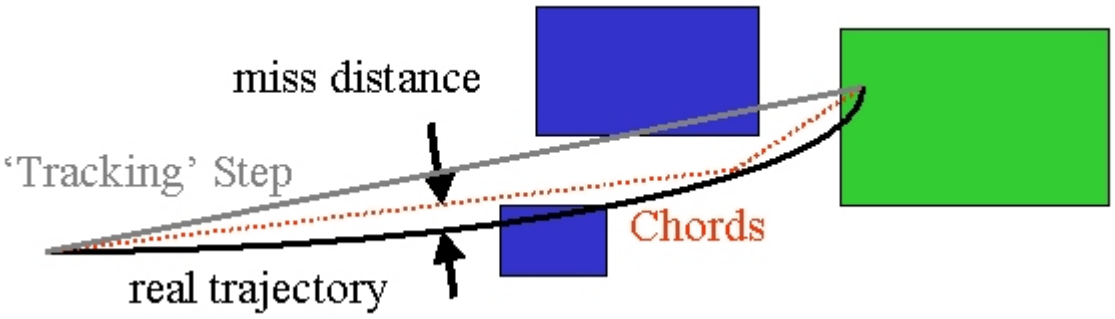


Figure A: 曲线径迹将用弦来近似，这些弦与曲线之间的最大距离必须小于 *miss distance*。

除了 *miss distance* 外，另外还有两个参数用于调准场中粒子跟踪的精度（和性能）。特别是这些参数控制了径迹和 volume 边界相交的精度和其余步积分的精度。正因为如此，它们在粒子跟踪中扮演了一个重要的角色。

delta intersection 参数是计算粒子径迹与 volume 边界相交的精度。如果有一个与边界相交的交点，而且估计精度要比这个参数设置高，那么将使用这个已知的相交值。这个参数非常重要，因为它被用来限制目前算法（用于在场中的边界相交）出现的偏差。这个算法计算近似用的弦与 volume 边界的交点。这样，这个交点始终位于粒子径迹曲线的“内侧”。将这个参数设置为足够小的值，使它远小于可接受的误差，用户可以限制这个偏差带来的影响，例如，将来重建粒子动量的估计。

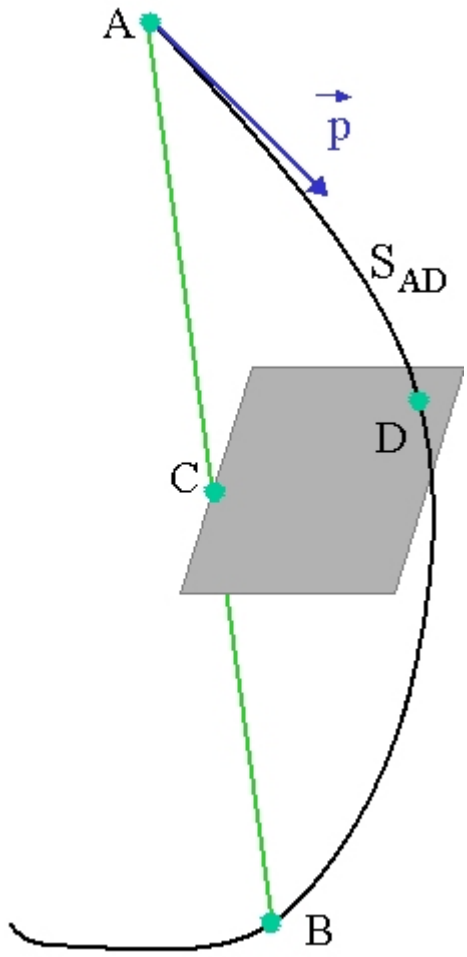


Figure B:　弦截点 C 与曲线交点 D 之间的距离，用来判断 C 点是否能够比较精确的表示曲线 ADB 和 voluem 边界的交点。这里的 CD 太大了，将使用弦 AD 计算新的交点。

delta one step 参数是指'ordinary' step 积分的终点的精度，积分路径与 volume 边界不相交。这个参数是每个物理步终点的估计误差的极限，它可以被看作是统计不确定性，不期望对物理量产生任何系统行为。相反，对于重建粒子径迹的动量，与 *delta intersection* 相关的偏差是与潜在的系统误差有明确关系的。因此，对于进行粒子跟踪的探测器，和将那些交点用于重建粒子径迹动量的情况，要非常严格的限制这些相交参数。

Delta intersection 和 *delta one step* 是 Field Manager 的参数；用户可以根据应用的需要来设置它们。由于可以使用多个 field manager，可以对不同的探测器区域设置不同的值。

注意这两个参数合理的值是强烈相关的：将 *delta intersection* 设置为 1nm，*delta one step* 设置为 100 μm 是没有意义的。除非在这两个参数中，*delta intersection* 比较重要。推荐这两个参数不要相差太大－最好不超过一个量级。

4.3.2 实际问题

为探测器建立一个磁场

为探测器定义一个场的最简单方式有以下步骤：

1. 建立一个场：

```
      G4UniformMagField* magField
        = new G4UniformMagField(G4ThreeVector(0.,0.,fieldValue));
```

5. 将场设置为缺省场：

```
      G4FieldManager* fieldMgr
        = G4TransportationManager::GetTransportationManager()
          ->GetFieldManager();
       fieldMgr->SetDetectorField(magField);
```

11. 建立用于计算径迹的对像：

```
     fieldMgr->CreateChordFinder(magField);
```

用方法 `SetDeltaChord` 改变计算 volume 相交的精度：

```
fieldMgr->GetChordFinder()->SetDeltaChord( G4double newValue);
```

建立一个非磁场

Field 类允许建立一个非磁场。

Source listing 4.3.1 显示了如何为整个探测器定义一个均匀电场。

```
#include "G4EqMagElectricField.hh"
#include "G4UniformElectricField.hh"

...
{
  // 探测器描述代码部分

  G4FieldManager   *pFieldMgr;
  G4MagIntegratorStepper *pStepper;
  G4EqMagElectricField *fEquation = new
   G4EqMagElectricField(&myElectricField);
```

```
   pStepper = new G4ClassicalRK4( fEquation );
   // or    = new G4SimpleHeum(   fEquation );

   // 设置这个作为全局场
   pFieldMgr= G4TransportationManager::GetTransportationManager()->
       GetFieldManager();

   pFieldMgr->SetDetectorField( &myElectricField );
   pChordFinder = new G4ChordFinder( &myElectricField,
                      1.0e-2 * mm,     // 最小步长
                      pStepper);
   pFieldMgr->SetChordFinder( pChordFinder );
 }
```

Source listing 4.3.1
如何为整个探测器定义一个均匀电场。

选择 Stepper

龙格－库塔积分是用来计算场中带电粒子的运动的。有许多通用的 stepper 可供选择，有低阶和高阶的，还有一些专门用来计算纯磁场的 stepper。缺省情况下，G4 使用经典的四阶龙格－库塔 stepper，它是一个通用且健壮（很多猪头叫它鲁棒，太难听了）的 stepper。如果场的一些特性已知，可以使用更低阶、或跟高阶的 stepper，使得用更少的计算时间获得相同质量得结果。

特别是，如果场是从一个 field map 计算得到的，那么推荐使用更低阶的 stepper。场越不光滑，越应该使用低阶的 stepper。低阶的 stepper 包括三阶 stepper G4SimpleHeum, 二阶 G4ImplicitEuler 和 G4SimpleRunge, 以及一阶 G4ExplicitEuler。一阶 stepper 只用于非常粗糙的场。对于稍微光滑的场 (中等光滑), 选择二阶还是三阶的 stepper，需要由试验和误差来决定。为了获得最高的效率，需要仔细研究针对一个特殊的场，使用那种 stepper 是最好的。

G4 提供了一些专门用于纯磁场的 steppers。它们考虑了在一个光滑的场中，在径迹的局部区域，粒子的径迹近似为一条螺旋线。将龙格－库塔法和这个近似组合，形成了一种新的计算方法，使得使用很少的计算时间可以获得很高的精度。

如果需要，用户可以在构造 field manager 时指定 stepper，而不是使用缺省的。在构造时，简单使用

```
G4ChordFinder( G4MagneticField* itsMagField,
            G4double       stepMinimum = 1.0e-2 * mm,
            G4MagIntegratorStepper* pItsStepper = 0 );
```

稍后，改变 stepper 使用

```
pChordFinder->GetIntegrationDriver()
          ->RenewStepperAndAdjust( newStepper );
```

为 volume 层次建立一个场

现在，用户可以建立一个在逻辑体和它所有子体内部都是可见的场。下一个版本将会有这方面的描述。

已知问题

目前，要将 *miss distance* 设置为非常小的值，使近似的径迹被限制到“弯曲程度”(弯曲程度是指近似弦中点的距离)小于这个设置值的范围，使得近似弦线段变多，这将耗费大量的计算时间。对于小曲率径迹(典型的强场、低动量)的情况，这将导致产生大量的计算步（step）。

- 甚至是在没有 volumes 相交的区域 (有些东西需要在未来的开发中得到关注，安全性将用来部分减弱这个限制)
- 特别是在靠近 volume 边界的区域 (在这种情况下，为了发现是否径迹进入了一个 volume，上面 *miss distance* 的设置是必要的。)

当相交计算需要这样的精度时，将花费大量的计算时间，新的开发将要求最小的计算花费。

通过对比，改变相交参数，使得花费较少的计算时间。对于 steps 的一个片断将花费更多计算时间，特别是那些与 volume 边界相交的 step。

4.4 Hits

4.4.1 Hit

hit 是在用户探测器的灵敏区内，粒子发生的物理相互作用的快照。用户可以存储与 *G4Step* 对像相关的各种信息。这些信息可以是

- 每一步的位置和时间
- 粒子的动量和能量
- 每一步的能量沉积
- 几何信息

或者上面这些的任意组合。

G4VHit

G4VHit 是一个表示 hit 的抽像类。用户必须继承这个基类，派生出自己的 hit 类。用户类的成员数据由用户自己选择。

G4VHit 有两个虚方法，`Draw()` 和 `Print()`。用来显示和打印用户的 hits，这些方法应有用户实现。在 8.4 节有如何定义 draw 方法的内容。

G4THitsCollection

G4VHit 是一个抽像类，用户必须从它派生出自己的类。另外一方面，hits 应和 *G4Event* 对像一起存储，这个对像表示当前事件。*G4VHitsCollection* 是一个抽像类，表示一个向量集合，这个集合的元素是一种用户定义的 hits。因此，用户必须为每个 *G4VHit* 的具体类定义一个 *G4VHitsCollection* 的具体类。

G4THitsCollection 是一个从 *G4VHitsCollection* 派生的模板类 ，可以将这个模板类进行实例化，来为特定的 *G4VHit* concrete 类定义 hits collection（*G4VHitsCollection*）的具体类。用于每个事件的每个 hits collection 对像，必须有一个唯一的名字。

G4Event 有一个 *G4HCofThisEvent* 类的对像，它是一个 hits collection 的容器类。Hits collection 是通过它们的指针存储的，这些指针的类型是它们基类的类型。

一个 hit 具体类的例子

Source listing 4.4.1 显示了一个 hit 具体类的例子。

```
#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 G4Allocator<ExN04TrackerHit> ExN04TrackerHitAllocator;

inline void* ExN04TrackerHit::operator new(size_t)
{
  void *aHit;
  aHit = (void *) ExN04TrackerHitAllocator.MallocSingle();
  return aHit;
}

inline void ExN04TrackerHit::operator delete(void *aHit)
{
  ExN04TrackerHitAllocator.FreeSingle((ExN04TrackerHit*) aHit);
}

#endif
```

Source listing 4.4.1
一个 hit 具体类的例子。

G4Allocator 是一个用于从堆为对像中快速分配内存的类，它使用了页机制。有关 *G4Allocator* 的细节，查看 3.2.4。虽然不是一定要使用这个类，但我们推荐使用它，尤其是那些对 C++ 内存分配机制不熟悉的用户，和那些不熟悉其它内存分配工具的用户。

4.4.2 灵敏探测器

G4VSensitiveDetector

G4VSensitiveDetector 是一个描述探测器的抽像基类。一个灵敏探测器原则上要求建立一些 hit 对像，这些对像使用的信息来自于一个粒子径迹上的 steps。*G4VSensitiveDetector* 的 `ProcessHits()` 方法使用 *G4Step* 对像作为输入执行这个任务。在“读出”几何(参看 4.4.3 节)情况下， *G4TouchableHistory* 类的对像可以用作可选输入。

用户的具体探测器类应使用一个唯一的名字进行实例化。这个名字和其它的全局名字一起，用"/"分隔，来区分用户不同的探测器。例如

```
myEMcal = new MyEMcal("/myDet/myCal/myEMcal");
```

这里的 `myEMcal` 是用户探测器名。指向用户灵敏探测器的指针，必须传递给一个或多个 *G4LogicalVolume* 对像，以设置这些逻辑体的灵敏特性。就如 4.4.4 节所述，这个指针同时也应该向 *G4SDManager* 注册。

G4VSensitiveDetector 有三个主要的虚拟方法。

`ProcessHits()`
: 当一个 step 是在一个拥有指向灵敏探测器指针的 *G4LogicalVolume* 内部时，*G4SteppingManager* 将调用这个方法。这个方法的第一个参数是当前 step 的一个 *G4Step* 对像。第二个参数是一个用于``读出几何"的 *G4TouchableHistory* 对像，关于``读出几何"将在下节讲述。在这个方法中，如果当前 step 对用户探测器是非常有意义的，那么，将构造一个甚至多个 *G4VHit* 对像。

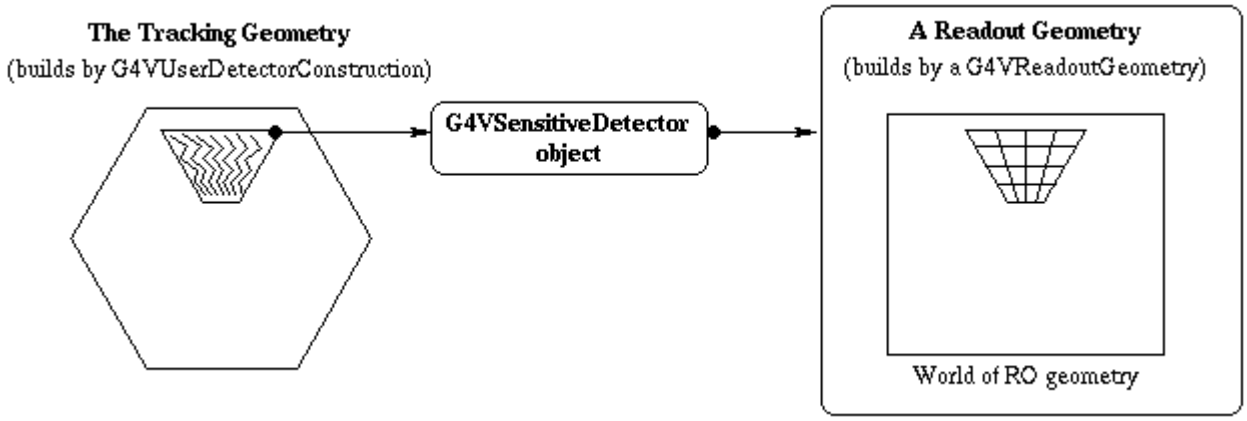
`Initialize()`
: 这个方法在每个事件开始的时候调用。它的参数是一个 *G4HCofThisEvent* 对像。在这个方法中，Hits collections 与 *G4HCofThisEvent* 对像相关，在这个事件内产生的 hits 被存储。在这个方法中，hits collections 和 *G4HCofThisEvent* 对像一起，可以用于``在事件处理期间"数字化。

`EndOfEvent()`
: 这个方法在每个事件结束是调用。它的参数跟前一个方法一样。在用户灵敏探测器中偶尔建立的 hits collections 可以与 *G4HCofThisEvent* 对像关联。

4.4.3 读出几何

在这节中，将解释如何定义一个``读出几何"。读出几何是一个虚拟共生的几何体，它用来获取通道号。

例如，**ATLAS** 折叠状的量热器有一个复杂的几何，然而，他可以被简单的化分为一定 theta，phi 和深度的柱形扇面读出。径迹将在“实际的”几何体中被跟踪，灵敏探测器拥有自己的读出几何，G4 将检查当前 hit 属于那个``读出"单元。



图中显示了在 G4 中是怎样完成这个关联的。通常(参看 4.4.2 节)，用户必须先将一个灵敏探测器与一个粒子跟踪几何中的 volume 关联。然后，用户将 *G4VReadoutGeometry* 对像与灵敏探测器关联。

进行粒子跟踪的时候，在 step 的开始位置(*G4Step* 中 *PreStepPoint* 的位置) ，基类 *G4VReadoutGeometry* ，将在读出几何中向用户灵敏探测器提供 *G4TouchableHistory* .

G4TouchableHistory 是通过 *G4VSensitiveDetector* 的虚方法，使用参数 `ROhist` 传递给用户灵敏探测器的:

```
G4bool processHits(G4Step* aStep, G4TouchableHistory* ROhist);
```

用户将可以使用来自读出几何的 *G4Step* 和 *G4TouchableHistory* 的有关信息。注意关联是通过一个灵敏探测器对像完成的，完全有可能同时存在多个读出几何。

一个虚拟几何设置的定义

用于实现读出几何的基类是 *G4VReadoutGeometry*。这个类有唯一的一个纯虚保护方法：

```
virtual G4VPhysicalVolume* build() = 0;
```

用户必须在自己的具体类中进行重载。用户必须返回的 *G4VPhysicalVolume* 指针读出几何的 physical world.

建立一个读出几何的过程如下：

- 定义一个从 *G4VReadoutGeometry* 继承的类 *MyROGeom* ；
- 在方法 `build()` 中实现读出几何，返回这个几何的 physical world。

指定这个 world 的方式与构造探测器几何时使用的相同：它是一个没有母体的物理体。这个 world 的坐标系与用于粒子跟踪的 world 相同。

在这个几何中，用户必须以在粒子跟踪几何中同样的方式声明灵敏部分：在假定相关的 *G4LogicalVolume* 对像中，设置一个非空的 *G4VSensitiveDetector* 指针。虽然这个灵敏特性不会被使用，但还是需要声明。

事实上，用户同样需要为这个几何中的 volumes 定义材料。这些材料是与粒子跟踪无关的，对于粒子跟踪是不可见的。所以，在这种共生几何体的情况下，可以预见允许用户设置一个空指针。

- 在用户 *G4VUserDetectorConstruction* 具体类的 `construct()` 方法中：
  - 实例化用户的读出几何：
  -
  - ```
    MyROGeom* ROgeom = new MyROGeom("ROName");
    ```
  - 建立:
  -
  - ```
    ROgeom->buildROGeometry();
    ```

    这将调用用户的 `build()` 方法。
  - 实例化灵敏探测器 `MySensitive`，它接受 `ROGeom` 指针作为参数，并且将这个灵敏探测器添加到 *G4SDManager*。像通常一样，将这个灵敏探测器与粒子跟踪几何中的 volumes 关联。
  - 将灵敏探测器关联到读出几何：
  -
  - ```
    MySensitive->SetROgeometry(ROgeom);
    ```

4.4.4 *G4SDManager*

G4SDManager 是一个 singleton 管理类，用于管理灵敏探测器。

灵敏探测器的激活/关闭

用户接口命令 `activate` 和 `inactivate` 可以用来控制用户灵敏探测器。例如：

```
/hits/activate detector_name
/hits/inactivate detector_name
```

这里的 `detector_name` 是探测器名或者类名。例如，如果你的电磁量热计名叫 `/myDet/myCal/myEMcal`，
```

```
    /hits/inactivate myCal
```

这个命令将关闭所有属于 `myCal` 类的所有探测器。

存取 hits collections

有几种存取 hits collections 的情况。

- 数字化
- *G4VUserStackingAction* 中的事件过虑
- ``事件结尾"的简单分析
- 显示/打印 hits

下面是如何存取一个 hits collection 的例子：

```
  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));
```

4.5 数字化

4.5.1 Digi

当一个 step 经过一个灵敏探测器时，将产生一个 hit。因此，灵敏探测器是与相应的 *G4LogicalVolume* 对像相关的。另一方面，digit 使用 hits 的信息或者/和其它 digits，由一个数字化模块建立的。数字化模块与任何 volume 无关，用户必须显式的调用用户 *G4VDigitizerModule* 具体类中的方法 `Digitize()` 。

数字化模块的典型应用：

- 模拟 ADC 和/或 TDC
- 模拟读出模式
- 产生原始数据
- 模拟触发逻辑
- 模拟堆积

G4VDigi

G4VDigi 是一个表示 digit 的抽像基类。用户必须继承这个基类，并派生自己的 digit 类。用户 digit 类的成员数据应由用户指定。*G4VDigi* 有两个虚拟方法，`Draw()` 和 `Print()`。

G4TDigiCollection

G4TDigiCollection 是用于 digits collections 的一个模板类，它是从抽像基类 *G4VDigiCollection* 派生的。*G4Event* 有一个 *G4DCofThisEvent* 对像，它是一个 digits collections 的容器类。*G4VDigi* 和 *G4TDigiCollection* 的使用几乎分别与 *G4VHit* 和 *G4THitsCollection* 一样，后两者在前一节中已经讨论过。

4.5.2 数字化模块

G4VDigitizerModule

G4VDigitizerModule 是一个表示数字化模块的抽像基类。它有一个纯虚方法，`Digitize()`。一个具体的数字化模块必须实现这个虚方法。G4 内核类没有“内建的”代码调用 `Digitize()` 方法。用户必须使用自己的代码调用这个方法。

在 `Digitize()` 方法中，用户构造自己 *G4VDigi* 具体类的对像，并且将它们存储到用户的 *G4TDigiCollection* 具体类对像中。用户的 collection 应和 *G4DCofThisEvent* 对像相关。

G4DigiManager

G4DigiManager 是数字化模块的 singleton 管理类。所有用户的具体数字化模块都应使用它们唯一的名字向 *G4DigiManager* 注册。

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
MyDigitizer * myDM = new MyDigitizer( "/myDet/myCal/myEMdigiMod" );
fDM->AddNewModule(myDM);
```

用户可以使用唯一的模块名存取用户的具体数字化模块。

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
MyDigitizer * myDM = fDM->FindDigitizerModule( "/myDet/myCal/myEMdigiMod" );
```

```
myDM->Digitize();
```

同样，*G4DigiManager* 也有一个使用唯一模块名的方法 *Digitize()* 。

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
MyDigitizer * myDM = fDM->Digitize( "/myDet/myCal/myEMdigiMod" );
```

如何获取 hitsCollection 和/或 digiCollection

G4DigiManager 有下列方法用于存取 hits collections 或 digi collections ，这些被存取的对像，属于当前正在处理的事件或以前的事件。

首先，用户必须获取这些 hits collection 或 digits collection 的 collection ID 号。

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
G4int myHitsCollID = fDM->GetHitsCollectionID( "hits_collection_name" );
G4int myDigiCollID = fDM->GetDigiCollectionID( "digi_collection_name" );
```

然后，用户获取指向具体 *G4THitsCollection* 对像或 *G4TDigiCollection* 对像的指针，这些对像属于当前处理的事件。

```
MyHitsCollection * HC = fDM->GetHitsCollection( myHitsCollID );
MyDigiCollection * DC = fDM->GetDigiCollection( myDigiCollID );
```

如果用户需要存取以前事件的 hits collection 或 digits collection，使用第二个参数。

```
MyHitsCollection * HC = fDM->GetHitsCollection( myHitsCollID, n );
MyDigiCollection * DC = fDM->GetDigiCollection( myDigiCollID, n );
```

这里的 n 表示前第 n 个事件的 hits collection 或 digits collection。

About the authors

Overview Contents Previous Next

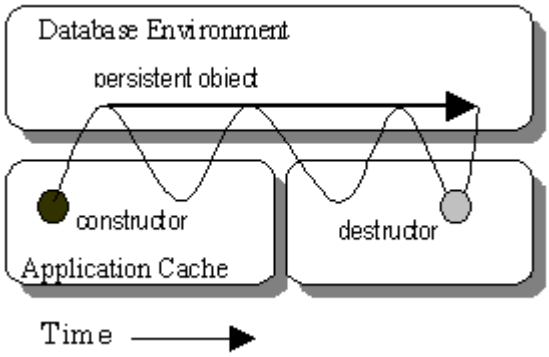
Geant4 User's Guide
For Application Developers
Detector Definition and Response

4.6 对像的持续性

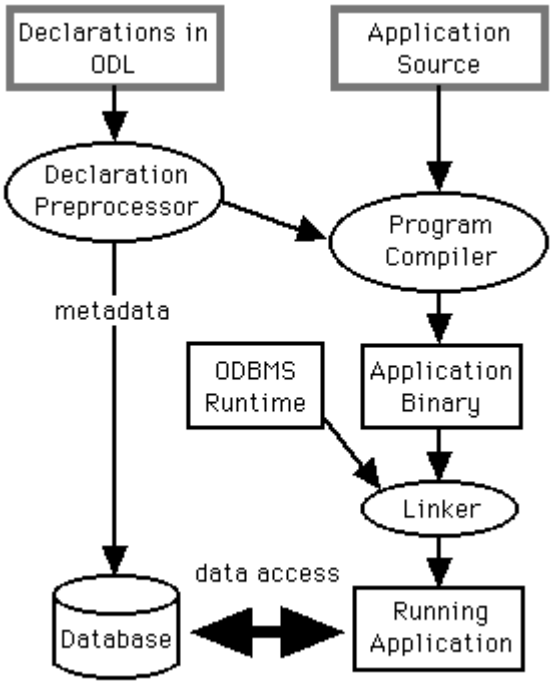
4.6.1 G4 的对像持续性

对像持久性是 G4 提供的一个可选工具类，以便用户可以在使用或者不使用商业对像数据库管理系统(ODBMS)软件包的情况下运行 G4。

当一个普通对像（暂时性对像）在 C++中建立的时候，对像被放到应用程序的堆中，当应用程序中止的时候，对像将停止存活。另一方面，持久性对像可以存活到应用程序进程中止之后，可以被其它经常存取（对于某些情况，是其它机器上的进程）。



C++本身没有能力存取持续性对像。然而，有多种方式获得对像持续性，其中一个是使用商业的对像数据库管理系统。Object Data Management Group (ODMG) 为具有持久能力的对像声明，制定了一个工业标准。类的声明使用 Object Definition Language (ODL)，ODL 的语法几乎与 C++类声明的语法相同。一旦使用 ODL 完成了类的声明，声明的预处理程序将产生 C++ 头文件和一个数据库模式，数据库存取 wrapper code。这些数据库模式定义了那些具有持久能力类的数据成员的格式和数据类型。



4.6.2 G4 对像持续性所需要的 ODBMS 软件包

在 G4 的这个版本中，持续性工具类需要 **Objectivity/DB** 和 **HepODBMS** 。

HepODBMS

HepODBMS 为 HEP 相关的应用程序提供了一个标准 ODBMS 接口，它正作为 CERN RD45 的一部分进行开发。HepODBMS 处理各种对像，包括 histograms,探测器刻度和几何，但重点是处理 HEP 事件数据。在 LHC 运行期间，处理的数据量期望达到每年几个 peta 字节。为了便于将 G4 应用程序从一个 ODBMS 包移植到另外一个，G4 使用 HepODBMS 和 ODMG 标准作为商业 ODBMS 软件包的接口。

Objectivity/DB

Objectivity/DB 是 HepODBMS 选用的一个商业 ODBMS 软件包。它的存储单元有四个逻辑层：

- Federated Database (ooFDObj)
- Database (ooDBObj)
- Container (ooContObj)
- Basic Object (ooObj).

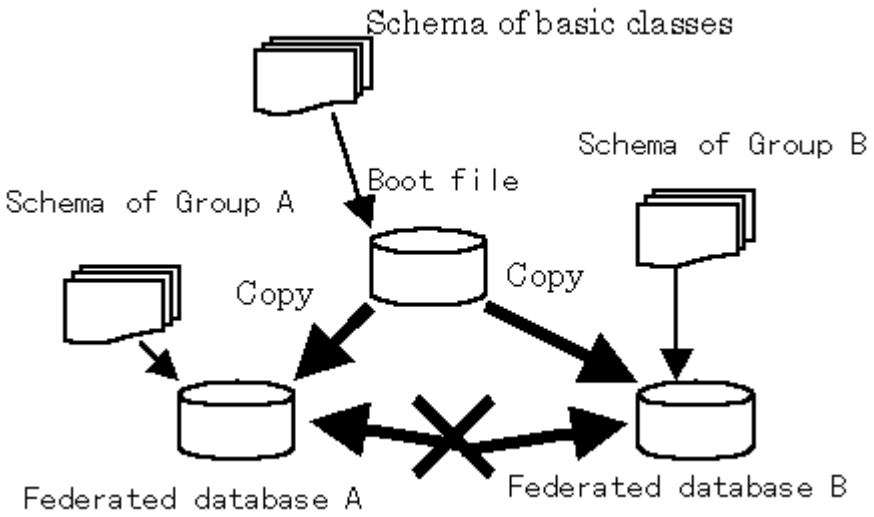
每一层对应与不同的物理存储层。在 Objectivity/DB 中的一个数据库对应于在数据库服务器上的一个物理文件。container 是 basic objects 的逻辑单位，存取 basic object 的过程是一个不

可中断的过程（类似原语的概念）。用户无法在数据库中通过每个对像的物理位置进行直接控制。然而，可以为许多期望同时存取的对像指定一系列指令。

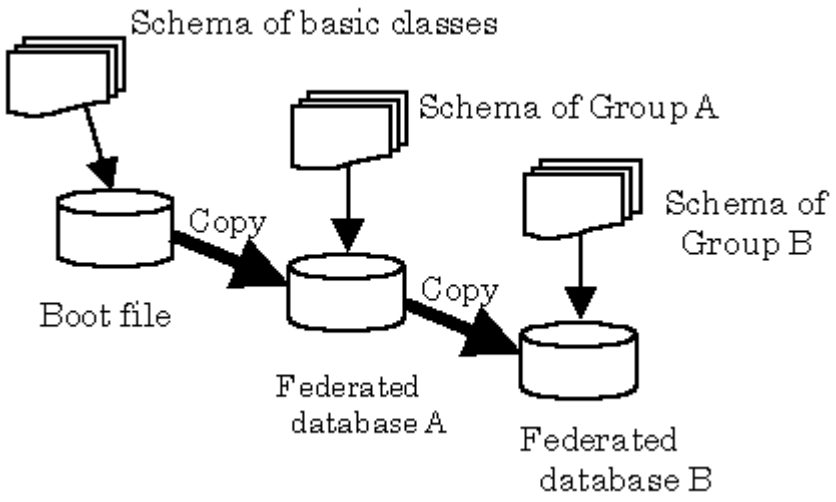
在 Objectivity/DB 中，对像声明语言被称为 DDL (data definition language)。DDL 文档随后由编译器 `ooddlx` 编译生成数据库模式文件，`<class>.hh`, `<class>_ref.hh` 和 `<class>_ddl.cc` 。

一个 federated 数据库定义那些相互关联的数据库文件的配置信息，同时包含数据库模式信息。当程序员改变持续性类中数据成员的数据类型或格式的时候，将引起数据模式的变化，`ooddlx` 编译器将报告一个错误。要排除这个错误，程序员应从暂存区删除这个 federated 数据库，并处理所有的 DDL 文档。（如果这个数据库已经在使用，并且用户希望保护该数据库的内容，程序员应根据数据模式描述提供一个新类声明，并且决定是否在数据存取的时候自动刷新数据模式。）

有时候，将类声明化分到给几个不同的软件小组是合理的。但是，在当前的 Objectivity/DB 中，没有比较方便的方法用于在 federated 数据库之间交换数据模式信息。因此，不同的小组都因该从同一个基本类的 federated 数据库获得完整的数据模式信息，并将它复制给位于自己下游相邻的小组。



- 如果 Group A 和 B 的数据模式是独立编译的，那么它们不能交换模式数据



- Group B 可以使用 Group A 的模式数据

HepODBMS 的基本模式数据库是 HEP_BASE，G4 的基本模式数据库 G4SCHEMA 是建立在 HEP_BASE 之上的。G4 用户因该重 G4SCHEMA 复制它们的基本模式数据。

在程序运行的时候，一个数据库应用程序只能存取一个 federated 数据库。

4.6.3 持续性的例子

`PersistentEx01` 是 G4 持续性的一个例子，位于 `geant4/examples/extended/persistency` 目录下。下面是编译和运行这个例子的步骤。

使用 Objectivity/DB 和 HepODBMS 的环境

首先，必须先得到存取 Objectivity/DB 目录的 ACL。 参看"Registration for Access to LHC++"的许可证测。在 CERN AFS 上，可以为 **Objectivity/DB** 和 **HepODBMS** 设置如下的环境变量：

```
  source
...geant4/examples/extended/persistency/PersistentEx01/g4odbms_setup.csh
```

在例子 `PersistentEx01` 中，用户的持续性对像将被存储到一个叫 `G4EXAMPLE` 的 federated 数据库中，选择则一个具有写权限的目录，例如：

```
cd $HOME
mkdir G4EXAMPLE
setenv G4EXAMPLE_BOOT_DIR $HOME/G4EXAMPLE
setenv G4EXAMPLE_BOOT $HOME/G4EXAMPLE/G4EXAMPLE
```

对于每个 federated 数据库，用户必须指定一个唯一的 federated 数据库 ID(FDID)。向你的本地 Objectivity ``Lock Server"系统管理员资询有关 FDID 的问题。

```
setenv G4EXAMPLE_FDID <nnnnn>
```

这里的<nnnnn> 是一个分配给用户 federated 数据库 G4EXAMPLE 的唯一的数字。在 CERN AFS 上，由于 Objectivity/DB 一些技术上的限制，用户必须启动自己的"Lock Server" 来注册 federated 数据库：

```
oocheckls -notitle `hostname` || \
  oolockserver -notitle -noauto `hostname`::$G4EXAMPLE_BOOT
```

当用户结束运行程序之后，不要忘了 kill 自己的 lock server，输入：

```
ookillls
```

或，

```
ps -ef | grep ools
kill -9 <pid_of_ools>
```

创建 libG4persistency.a 和 G4SCHEMA

现在，用户就可以创建一个 Geant4 持续性库和相关的基本模式文件 (G4SCHEMA)。这个模式文件包含那些持续性对像的数据成员的数据类型和格式。

```
cd ...geant4/source/persistency
gmake         # for granular library setup
gmake global   # for grobal library setup
```

注意，与 **HepODBMS**- 和 **Objectivity**-有关的编译控制命令定义在 GNUmake include 文件里面(*.gmk) ，在 geant4/config 目录下。持续性库将建立在$(G4WORKDIR)/lib 目录下，名叫 libG4persistency.a ， G4SCHEMA 创建在$(G4WORKDIR)/schema 目录下。

建立 federated 数据库和 PersistentEx01 的可执行文件

建立用户本地的 federated 数据库 G4EXAMPLE 和 PersistentEx01 的可执行文件。确认当前目录为 geant4/examples/extended/persistency/PersistentEx01，或者将这个目录下的文档复制到用户的工作目录，然后运行 gmake。

```
cd (..to your working directory..)
cp -r ..geant4/examples/extended/persistency/PersistentEx01 ./
cd PersistentEx01
gmake cleandb    # to create G4EXAMPLE
gmake           # to create PersistentEx01 executable
```

运行 PersistentEx01 并检查持续性对像

添加 geant4/bin 路径到环境变量 path，改变目录到 G4EXAMPLE_BOOT_DIR，然后运行可执行文件：

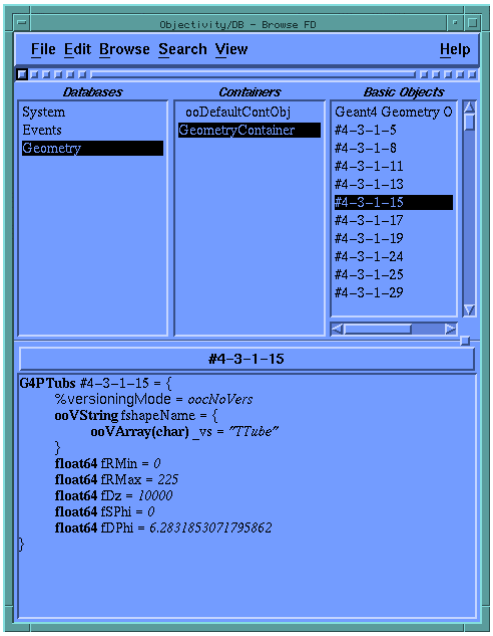
```
  set path = ($path ../geant4/bin)
  cd $G4EXAMPLE_BOOT_DIR
  PersistentEx01 <
geant4/examples/extended/persistency/PersistentEx01/PersistentEx01.in
```

宏文件 PersistentEx01.in 将处理 10 个事件，并存储事件和几何对像到数据库中。

用 ootoolmgr 检查数据库内容。

```
  setenv DISPLAY <...to your local display...>
  cd $G4EXAMPLE_BOOT_DIR
  ootoolmgr -notitle G4EXAMPLE &
```

然后从``Tools"菜单选择``Browse FD" 。下面是流览几何数据库的例子。



4.6.4 生成自己的持续性对像

在当前的 G4 版本中， *G4PersistencyManager* 可以存储两种对像，"events" 和"geometry"。要实现 "hits"的持续性，用户因该从 *G4VHit* 继承并提供自己的持续性类。

要存储事件对像，用户因该在构造 *G4RunManager* 之前构造 *G4PersistencyManager* 。下面是在 main()中通常使用的方式。

```
#include "G4RunManager.hh"
#include "G4PersistencyManager.hh"
 int main()
 {
   G4PersistencyManager * persistencyManager = new G4PersistencyManager;
   G4RunManager * runManager = new G4RunManager;
   ...
 }
```

Source listing 4.6.1
How to store a persistent event object.

在事件 action 结束的时候，如果存在 `G4PersistencyManager::Store( theEvent )`，*G4RunManager* 将调用它。在当前实现的 *G4PersistencyManager* 中，只是在 *G4PEvent* 中设置事件 ID。

要存取几何对像，用户应该在构造完探测器之后，在 user run action 类中调用 `Store( theWorld )`。

```
void MyRunAction::BeginOfRunAction(G4Run* aRun)
{
  aRun->SetRunID(runIDcounter++);
  if(runIDcounter==1)
  {
    G4VPersistencyManager* persM
     = G4VPersistencyManager::GetPersistencyManager();
    if( persM )
    {
      G4VPhysicalVolume* theWorld
       = G4TransportationManager::GetTransportationManager()
         ->GetNavigatorForTracking()->GetWorldVolume();
      persM->Store(theWorld);
    }
  }
}
```

Source listing 4.6.2
How to store a persistent geometry object.

要存储一个 hits 对像，用户应该提供一个具有持续性能力的 hit，和从 *G4VHits* 和 *G4VHitsCollection* 继承的 hit collection 对像。有关创建有持续性能力的 hit 类的细节将在下个版本的 G4 中提供。

下面是如何建立一个持续性类的简要方法。

- 将.hh 文件更名为.ddl 文件
- 从 *HepPersObj* (d_Object) 继承
- 将 C++指针替换为 smart 指针
-
- classname* aPtr --> HepRef(classname) aPtr // in implementation
- --> ooRef(classname) aRef // in data member

- 数据成员使用持续性类型
-
- G4String aString; --> G4PString aString;

- 在.cc 文件中实现数据库存取。

下面是一个关于量热器 hit 和 hits collection 类的例子。

```
#include "G4VHit.hh"
#include "G4ThreeVector.hh"
#include "HepODBMS/odbms/HepODBMS.h"
#include "HepODBMS/clustering/HepClusteringHint.h"

 class MyCalorimeterHit : public HepPersObj, public G4VHit
 {
 public:
   MyCalorimeterHit();
  ~MyCalorimeterHit();
   MyCalorimeterHit(const MyCalorimeterHit& right);
   const MyCalorimeterHit& operator=(const MyCalorimeterHit &right);
   int operator==(const MyCalorimeterHit &right) const;
   ...
   static HepClusteringHint clustering;
   ...
 private:
   G4double edep;
   G4ThreeVector pos;
public:
   ...
};

#include "HepODBMS/odbms/HepODBMS.h"
#include "HepODBMS/clustering/HepClusteringHint.h"
#include "HepODBMS/odbms/HepRefVArray.h"
```

```
#include "G4VHitsCollection.hh"
#include "MyCalorimeterHit.hh"

class G4VSensitiveDetector;

declare(HepRefVArray,MyCalorimeterHit)
typedef HepRefVArray(MyCalorimeterHit) CaloHitsCollection;

class MyCalorimeterHitsCollection : public d_Object, public G4VHitsCollection
{
public:
  MyCalorimeterHitsCollection();
  MyCalorimeterHitsCollection(G4String aName,G4VSensitiveDetector *theSD);
 ~MyCalorimeterHitsCollection();
  static HepClusteringHint clustering;
  ...
private:
  CaloHitsCollection theCollection;
  ...
};
```

Then in `MyEventAction::endOfEventAction()`,

```
void MyEventAction::endOfEventAction()
{
  ...
  G4VPersistencyManager* persM
   = G4VPersistencyManager::GetPersistencyManager();
  if( persM )
  {
    persM->GetDBApp()->startUpdate();
    persM->GetDBApp()->db("Hits");
  }

  const G4Event* evt = fpEventManager->get_const_currentEvent();

  G4HCofThisEvent * HCE = evt->get_HCofThisEvent();

  MyCalorimeterHitsCollection * pTHC1
   = (MyCalorimeterHitsCollection*)(HCE->get_HC(colID1));
  MyCalorimeterHitsCollection * pTHC2
   = (MyCalorimeterHitsCollection*)(HCE->get_HC(colID2));
  ...
```

```
  if( persM )
   persM->GetDBApp()->commit();
  ...
}
```

Source listing 4.6.3
量热器 hit 和 hits collection 类。

About the authors

5. 粒子跟踪和物理过程

5.1 粒子跟踪

5.1.1 基本概念

粒子跟踪方法

All Geant4 processes, including the transportation of particles, are treated generically. 尽管叫"*tracking*"，粒子并不是在 tracking 模块中进行*输运*的。*G4TrackingManager* 是一个接口类，处理 event，track，tracking 模块之间的信息交换。这个类的一个 singleton 实例，处理上层的事件管理对像和下层 tracking 模块中的对像之间的消息传递。事件管理器是 *G4EventManager* 类的一个 singleton 实例。

tracking manager 从事件管理器接收一个 track，进行相应的处理（actions），完成粒子跟踪。*G4TrackingManager* 中集合了那些指向 *G4SteppingManager, G4Trajectory* 和 *G4UserTrackingAction* 的指针。同时，对与 *G4Track* 和 *G4Step* 有一个"使用"的关系。

G4SteppingManager 是一个在粒子跟踪中扮演重要角色的类。它处理在所有 G4 模块中，与粒子输运有关的对像之间的信息传递。(例如，几何，在物质中的相互作用等)。 它的公有方法 `Stepping()` 进行粒子跟踪的分步工作。处理一个 step 的算法如下。

1. 计算粒子在 step 开始是的速度。
2. 每个活动的离散或者连续物理过程，必须根据它所描述的相互作用来假定一个步长。Stepping 使用这些步长中的最小值。
3. 几何 navigator 计算到相邻 volume 的边界距离"newSafety"，如果物理过程中的最小物理步长小于"newSafety"，将使用这个物理步长作为下一步的步长。在这种情况下，不需要在进行其它的几何计算。
4. 如果从物理过程获得的最小物理步长比"newSafety"大，那么，将进行以下的处理：

    - 如果前一步没有被几何所限制，那么回到前一步的开始位置，并且
    - 重新计算到下一个边界的距离。
5. 将最小物理步长和几何步长中的较小者作为步长。
6. 调用所有活动的连续物理过程。注意，只有在所有的调用完成之后，粒子的动能才被更新。粒子动能的改变量，就是这个过程对粒子的贡献的和。
7. 检查 track 是否已经被一个连续物理过程所中止。
8. 在离散物理过程调用之前，更新 track 的属性，包括：
    - 更新当前粒子动能(注意， 'sumEnergyChange'是指在每个连续过程调用完成之后的动能改变的和，而不是物理过程开始前后能量差的和)并且
    - 更新位置和时间。
9. 调用离散物理过程。在过程开始之后
    - 更新当前跟踪的粒子的能量，并且
    - 在 SecondaryList 中存储来自 ParticleChange 的次级粒子。这包括构造"G4Track"对像，设置它们的成员数据。注意，stepping manager 负责删除这些次级粒子。
10. 检查 track，看是否已经被离散物理过程中止。
11. 更新"newSafety"。
12. 如果几何边界限制了 step，这个粒子将继续输运到下个几何体中。
13. 调用用户干预活动 *G4UserSteppingAction*.
14. 处理 hit 信息。
15. 将数据保存到 Trajectory。
16. 更新离散物理过程的的平均自由程。
17. 如果初始粒子仍然存活，那么重置发生离散物理过程的最大作用距离。
18. 到这，已经完成了一个 step。

什么是 step?

G4Step 存储一个 step 的瞬态信息。包括 step 的两个端点，`PreStepPoint` 和 `PostStepPoint`，包括这些点的坐标和包含它们的 volume。*G4Step* 也存储在两个点之间 track 属性的变化。这些属性包括能量和动量，它们在各种活动物理过程调用的时候更新。

什么是 track?

G4Track 保存粒子完成一个 step 之后的最终状态信息。这意味着它在 `AlongStepDoIt` 运行期间，保存了前一个 step 的信息。只有在 `AlongStepDoIt`'s 结束之后，*G4Track* 才拥有最终信息 (例如，最终位置)。同时需要注意，如前所述，*G4Track* 将在每次 `PostStepDoIt` 开始之后更新。

5.1.2 存取 track 和 step 信息

如何获取 track 信息

Track 信息可以通过调用 *G4Track* 类提供的 `Get` 方法来进行存取。详情请看 **Software Reference Manual**。典型信息包括：

- (x,y,z)
- 全局时间 (从事件建立时开始的时间)
- 局部时间 (从 track 建立时开始的时间)
- 本征时间 Proper time
- 动量方向 (单位向量)
- 动能
- 累计的几何径迹长度
- 累计的真实径迹长度
- 指向动态粒子的指针
- 指向物理体的指针
- Track ID 号
- 父 track 的 Track ID 号
- 当前 step 号
- Track 状态
- (x,y,z) track 的开始位置 (顶点位置)
- 在 track 开始位置(顶点位置)的动量方向
- 在 track 开始位置(顶点位置)的动能
- 指向建立当前 track 的物理过程的指针

如何获取 step 信息

Step 和 step-point 信息可以通过调用 *G4Step*/*G4StepPoint* 类的 `Get` 方法来获取。详情请看 **Software Reference Manual**。*G4Step* 中的信息包括：

- 指向 `PreStep` 和 `PostStepPoint` 的指针
- 几何步长 (多次散射校正之前的步长)
- 真实步长 (多次散射校正之后的步长)
- 在 `Pre` 和 `PostStepPoint` 之间，时间/位置的变化量
- 在 `Pre` 和 `PostStepPoint` 之间，动量/能量的变化量。 (注意：要获取在 step 中沉积的能量，不能使用这个'能量变化量。用户只能向下面那样使用“总能量沉积”。)
- 指向的 `G4Track` 指针
- 在 step 期间的总能量沉积－－这是下面这些量的和
    - 由能量损失的物理过程沉积的能量，
    - 那些因低于截断阈值，而并没有产生的次级粒子引起的能量损失

在 *G4StepPoint* (`Pre` 和 `PostStepPoint`) 中的信息包括：

- (x, y, z, t)
- (px, py, pz, Ek)
- 动量方向 (单位向量)
- 指向物理体的指针
- Safety
- Beta, gamma
- 极化
- Step 状态
- 指向定义当前 step 和它的 `DoIt` 类型的物理过程的指针
- 指向定义前一个 step 和它的 `DoIt` 类型的物理过程的指针

- 总的径迹长度
- 全局时间 (从事件建立时开始的时间)
- 局部时间 (从 track 建立时开始的时间)
- 本征时间 Proper time

如何获取"particle change"

用户可以通过调用 *G4ParticleChange* 类的 `Get` 方法存取保存在 particle change 中的信息。典型信息包括 (详情，参看 **Software Reference Manual**)：

- 父粒子的最终动量方向
- 父粒子的最终动能
- 父粒子的最终位置
- 父粒子的最终全局时间
- 父粒子的最终本征时间 proper time
- 父粒子的最终极化
- 父粒子的状态(*G4TrackStatus*)
- 真实步长(这个值被多次散射所使用，将几何步长的变换结果放到真实步长中)
- 局部能量沉积－－可能由以下部分组成
  - 通过能量损失的物理过程沉积的能量，或者
  - 那些因低于截断阈值，而并没有产生的次级粒子引起的能量损失。
- 次级粒子数
- 次级粒子列表 (*G4Track* 列表)

5.1.3 初级粒子处理

初级粒子被当作 *G4Track* 从一个物理过程传递给 tracking 的。 *G4ParticleChange* 为物理过程提供了如下四个方法：

- `AddSecondary( G4Track* aSecondary )`
- `AddSecondary( G4DynamicParticle* aSecondary )`
- `AddSecondary( G4DynamicParticle* aSecondary, G4ThreeVector position )`
- `AddSecondary( G4DynamicParticle* aSecondary, G4double time )`

除了第一个外，其余的方法中，*G4Track* 的构造是在方法内部，使用参数提供的信息来完成的。

5.1.4 用户行为（actions）

在粒子跟踪期间，G4 内核提供了两种 actions 供用户使用：

- 用户粒子跟踪 action，和

- 用户 stepping action。用户可以将自己定义的相互作用代码放在这些 action 方法中。详情请看 **Software Reference Manual**。

5.1.5 冗余信息输出

用户可以打开或关闭用于信息输出标志。通过设置容易信息标志的控制级别，用户可以获得不同详细程度的关于 track/step 的信息。例如

```
G4UImanager* UI = G4UImanager::GetUIpointer();
UI->ApplyCommand("/tracking/verbose 1");
```

5.1.6 Trajectory 和 trajectory point

G4Trajectory 和 G4TrajectoryPoint

G4Trajectory 和 *G4TrajectoryPoint* 是 G4 提供的缺省的具体类，它们分别从基类 *G4VTrajectory* 和 *G4VTrajectoryPoint* 继承。当一个 *G4Track* 从 *G4EventManager* 被传递的时候，*G4TrackingManager* 建立一个 *G4Trajectory* 类对像。*G4Trajectory* 有下列数据成员：

- track 和父 track 的 ID 号
- 粒子名字，电荷和它的 PDG 代码
- *G4TrajectoryPoint* 指针的集合

G4TrajectoryPoint 对应一个沿粒子行进路径的 step point。它的位置表示为一个 *G4ThreeVector*。*G4TrajectoryPoint* 类对像是由 *G4Trajectory* 中的 *AppendStep()*方法建立的，在每一个 step 结尾，由 *G4TrackingManager* 调用这个方法。当 *G4Trajectory* 建立的时候，第一个 point 就被建立了。因此第一个 point 就是初始顶点。

一个 trajectory 的产生过程可以通过 *G4TrackingManager::SetStoreTrajectory(G4bool)*控制。用户接口命令*/tracking/storeTrajectory _bool_* 完成同样的工作。用户可以在自定义的 *G4UserTrackingAction::PreUserTrackingAction()*方法中为每个独立的 track 设置这个标志。

用户不应在簇射过程中为次级粒子建立 trajectories，否则将消耗大量的内存。
所有在一个事件中建立的 trajectories 都被存储在 *G4TrajectoryContainer* 类对像中，且这个对像由 *G4Event* 控制。 用户要显式或者打印在一个事件中产生的径迹，可以在他的 *G4UserEventAction::EndOfEventAction()*中分别调用 *G4VTrajectory* 的方法，*DrawTrajectory()* 或 *ShowTrajectory()* 。几何必须在径迹之前显式。径迹的颜色表示如下。

- 红色：表示负带电粒子
- 绿色：表示中性粒子
- 蓝色：表示正带电粒子

由于 *G4Navigator* 的改进，一个 track 可以沿它的螺旋径迹运行很多圈，只要不穿过几何边界，它的 steps 不会被中断。因而，显式的径迹可以不是圆形的。

自定义 trajectory 和 trajectory point

G4Track 和 *G4Step* 是暂态类。 它们在事件结束的时候是不可用的。因此，用户只能使用 *G4VTrajectory* 和 *G4VTrajectoryPoint* 的具体类来进行事件结束时的分析和实现 G4 的持续性能力。如上所述，G4 提供的缺省类，例如 *G4Trajectory* 和 *G4TrajectoryPoint*，只有很少的几个。用户可以直接从各个基类派生，定制自己的 trajectory 和 trajectory point 类。

用户要定制 trajectory，必须在自己的 *G4UserTrackingAction::PreUserTrackingAction()*中构造他具体的 trajectory 类对像，并且通过方法 *SetTrajectory()*设置指向 *G4TrackingManager* 的指针。在用户的 trajectory 类的 *AppendStep()*方法中，必须建立用户自定义的 trajectory point 对像。这个 *AppendStep()* 方法将被 *G4TrackingManager* 调用。

用户要自定义径迹显式，可以重载他的 trajectory 类中的 *DrawTrajectory()*方法。

About the authors

Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Tracking and Physics

5.2 物理过程

物理过程描述粒子是怎样和物质发生相互作用的。G4 提供了 7 个主要的物理过程模块：

1. 电磁相互作用，
2. 强相互作用，
3. 衰变过程，
4. photolepton-hadron，
5. 光学过程，
6. 参数化过程 和
7. 输运过程

将物理过程进行抽像和一般化是 G4 设计时的一个关键问题。所有的物理过程都使用同样的方式处理。G4 可以让任何用户建立一个物理过程，并将它指定给一种粒子。这种开放的特性允许不同的用户建立一些新的、用于特定领域或特定目的的物理过程。

每个物理过程都有两组方法，`GetPhysicalInteractionLength` (GPIL) 和 `DoIt`，它们在粒子跟踪中扮演重要角色。GPIL 方法给出从当前时空位置到下个时空位置的步长，它根据物理过程的反应截面信息来计算发生反应的概率。在 step 的结尾，调用 `DoIt` 方法。`DoIt` 方法实现了这个反应的细节，改变粒子的能量、动量、方向和位置，如果需要的话，还产生次级粒子。这些变化都被作为 *G4VParticleChange* 对像所记录 (参看 Particle Change).

G4VProcess

G4VProcess 是所有物理过程的基类。每个物理过程必须实现虚方法 *G4VProcess* ，这个方法描述了这个相互作用过程(DoIt) 和决定什么时候发生相互作用 (GPIL)。为了适用于不同的反应类型， *G4VProcess* 提供了三个 `DoIt`：

- `G4VParticleChange* AlongStepDoIt( const G4Track& track, const G4Step& stepData )`

  当通过一个 step 输运一个粒子的时候，将调用这个方法。相应的每个已定义物理过程的 `AlongStepDoIt` 方法，将应用于每个 step，不管哪个物理过程具有最小的步长。每个 track 信息的最终改变都被记录和累计在 *G4Step* 中。在所有的物理过程都被调用完成之后，由于将 `AlongStepDoIt` 应用于 *G4Track*，粒子将发生一些改变，包括粒子重新定位和更新 safety。注意，在启动 `AlongStepDoIt` 之后，如果 step 被几何边界所限制，那么 *G4Track* 对像的终点将位于一个新的 volume 内。为了获取先前那个 voume 的信息，必须对 *G4Step* 进行存取，因为它包含这个 step 的两个端点的信息。

- `G4VParticleChange* PostStepDoIt( const G4Track& track, const G4Step& stepData )`

  只有在与这个方法对应的物理过程产生的步长为最小，或者这个物理过程被强迫发生时，才在这个 step 的终点调用这个方法。与 `AlongStepDoIt` 方法相对应，在启动 `PostStepDoIt` 之后，*G4Track* 将被更新。

- `G4VParticleChange* AtRestDoIt( const G4Track& track, const G4Step& stepData )`

  只有在这个方法对应的物理过程产生的步长最小，或该过程强迫发生时，才调用这个方法来停止这个粒子：

G4VProcess 为上面的每个 `DoIt` 方法提供了一个对应的纯虚 GPIL 方法：

- `G4double PostStepGetPhysicalInteractionLength( const G4Track& track, G4double previousStepSize, G4ForceCondition* condition )`

  这个方法产生一个它对应的物理过程的步长，同时提供一个标志用于强迫发生反应。

- `G4double AlongStepGetPhysicalInteractionLength( const G4Track& track, G4double previousStepSize, G4double currentMinimumStep, G4double& proposedSafety, G4GPILSelection* selection )`

  这个方法产生一个它对应的物理过程的步长。

- `G4double AtRestGetPhysicalInteractionLength( const G4Track& track, G4ForceCondition* condition )`

  这个方法产生一个它对应的物理过程的时间步长，同时提供一个标志用于强迫发生反应。

G4Vprocess 中其它的纯虚方法如下：

- `virtual G4bool IsApplicable(const G4ParticleDefinition&)`

  如果这个物理过程对像可应用于参数给定的粒子类型，返回 true。

- `virtual void BuildPhysicsTable(const G4ParticleDefinition&)`

  无论什么时候，只要截断值发生改变，就需要重建截面表，这时，process manager 将发消息给这个方法。如果物理过程与截面值无关，那么它就不是不需的了。

- `virtual void StartTracking()` 和
- `virtual void EndTracking()`

  在开始和结束跟踪当前 track 时，tracking manager 将发消息给这两个方法。

用于物理过程的其它基类

特殊的物理过程可以从 7 个其它的虚拟基类派生，这些基类本身是从 *G4VProcess* 派生的。它们中的三个类用于简单过程：

G4VRestProcess	只使用 `AtRestDoIt` 方法的物理过程 例如：中子俘获
G4VContinuousProcess	只使用 `AlongStepDoIt` 方法的物理过程 例如：切伦科夫
G4VDiscreteProcess	只使用 `PostStepDoIt` 方法的物理过程 例如：康普顿散射，强子非弹性相互作用

其它的四个类用于一些相对复杂的物理过程：

G4VContinuousDiscreteProcess	同时使用 `AlongStepDoIt` 和 `PostStepDoIt` 方法的过程 例如：输运过程，电离过程 (能量损失和 delta ray【delta ray 是指在物质发生电离时发射的电子】)
G4VRestDiscreteProcess	同时使用 `AtRestDoIt` 和 `PostStepDoIt` 方法的过程 例如：正电子湮灭，衰变 (both in flight and at rest)
G4VRestContinuousProcess	同时使用 `AtRestDoIt` 和 `AlongStepDoIt` 方法的过程
G4VRestContinuousDiscreteProcess	同时使用 `AtRestDoIt`, `AlongStepDoIt` 和 `PostStepDoIt` 方法的过程

Particle change

G4VParticleChange 和它的派生类用于存储 track 的最终状态信息，包括次级 tracks，这些次级 tracks 是由 `DoIt` 方法产生的。*G4VParticleChange* 的实例是唯一一个由物理过程更新信息的对像，它负责更新 step。Stepping manager 收集次级 tracks，还通过 particle change 发送请求来更新 *G4Step*。

G4VParticleChange 是作为一个抽像类引入的。它有一些用于更新 *G4Step* 和处理次级粒子的方法。物理过程可以从 *G4VParticleChange* 派生自己的。它提供了三个纯虚方法，

- `virtual G4Step* UpdateStepForAtRest( G4Step* step )`,
- `virtual G4Step* UpdateStepForAlongStep( G4Step* step )` and
- `virtual G4Step* UpdateStepForPostStep( G4Step* step )`,

它们对应 *G4VProcess* 中的三个 `DoIt` 方法。每个派生的类，都应实现这些方法。

5.2.1 电磁相互作用

这一节中概述了在 G4 中实现的电磁相互作用过程。有关这些物理过程实现的细节，请参看 **Physics Reference Manual**。

5.2.1.1 "标准" 电磁相互作用过程

下面是在 G4 中使用的标准电磁相互作用的概要。

- 光子 相关的过程
  - 康普顿散射 (类名是 *G4ComptonScattering*)
  - Gamma 转换 (也叫电子对生成，类名是 *G4GammaConversion*)
  - 光电效应 (类名是 *G4PhotoElectricEffect*)
- 电子/正电子 相关的过程
  - 韧致辐射(类名 *G4eBremsstrahlung*)
  - 电离和 delta ray 产生 (类名 *G4eIonisation*)
  - 正电子湮灭 (类名 *G4eplusAnnihilation*)
  - 能量损失过程 (类名 *G4eEnergyLoss*) 处理粒子的连续能量损失。这些连续能量损失是由于电离过程和韧致辐射的发生。
  - 同步辐射 (类名 *G4SynchrotronRadiation*)
- 强子 相关的过程
  - 电离 (类名 *G4hIonisation*)
  - 能量损失(类名 *G4hEnergyLoss*)
- 多次散射过程
  名为 *G4MultipleScattering* 的类在某种程度上是一个通用过程/类，它被用于模拟所有带电粒子(例如，用于 e+/e-，muons/带电强子)的多次散射。

  强子的电离/能量损失也可以使用类 *G4PAIonisation*/*G4PAIenergyLoss* 进行模拟。

  电子电离，韧致辐射，正电子湮灭，能量损失，和多次散射同时也在所为的``integral approach"中实现了，它们对应的类名是：

  - *G4IeBremsstrahlung*
  - *G4IeIonisation*
  - *G4IeplusAnnihilation*
  - *G4IeEnergyLoss*

- *G4IMultipleScattering*

5.2.1.2 低能电磁相互作用过程

下面是在 G4 中使用的标准电磁相互作用的概要。更多的信息，请流览 G4 低能电磁物理工作组的 homepage。这个过程的物理信息在 Physics Reference Manual 和其它的 papers 中。

- **光子 相关的过程**
  - 康普顿散射 (类 *G4LowEnergyCompton*)
  - 极化康普顿散射 (类 *G4LowEnergyPolarizedCompton*)
  - 瑞利散射(类 *G4LowEnergyRayleigh*)
  - Gamma 转换 (也叫电子对生成，类 *G4LowEnergyGammaConversion*)
  - 光电效应 (类 *G4LowEnergyPhotoElectric*)
- **电子 相关的过程**
  - 韧致辐射 (类 *G4LowEnergyBremsstrahlung*)
  - 电离和 delta ray 产生 (类 *G4LowEnergyIonisation*)
- **强子和离子 相关的过程**
  - 电离和 delta ray 产生 (类 *G4hLowEnergyIonisation*)

souce listing 5.2.1 是在一个 Physics List 中注册这些物理过程的例子。

```
void LowEnPhysicsList::ConstructEM()
{
  theParticleIterator->reset();

  while( (*theParticleIterator)() ){

    G4ParticleDefinition* particle = theParticleIterator->value();
    G4ProcessManager* pmanager = particle->GetProcessManager();
    G4String particleName = particle->GetParticleName();

    if (particleName == "gamma") {

      theLEPhotoElectric   = new G4LowEnergyPhotoElectric();
      theLECompton        = new G4LowEnergyCompton();
      theLEGammaConversion = new G4LowEnergyGammaConversion();
      theLERayleigh       = new G4LowEnergyRayleigh();

      pmanager->AddDiscreteProcess(theLEPhotoElectric);
      pmanager->AddDiscreteProcess(theLECompton);
      pmanager->AddDiscreteProcess(theLERayleigh);
      pmanager->AddDiscreteProcess(theLEGammaConversion);

    }
```

```
    else if (particleName == "e-") {

      theLEIonisation = new G4LowEnergyIonisation();
      theLEBremsstrahlung = new G4LowEnergyBremsstrahlung();
      theeminusMultipleScattering = new G4MultipleScattering();

      pmanager->AddProcess(theeminusMultipleScattering,-1,1,1);
      pmanager->AddProcess(theLEIonisation,-1,2,2);
      pmanager->AddProcess(theLEBremsstrahlung,-1,-1,3);

    }
    else if (particleName == "e+") {

      theeplusMultipleScattering = new G4MultipleScattering();
      theeplusIonisation = new G4eIonisation();
      theeplusBremsstrahlung = new G4eBremsstrahlung();
      theeplusAnnihilation = new G4eplusAnnihilation();

      pmanager->AddProcess(theeplusMultipleScattering,-1,1,1);
      pmanager->AddProcess(theeplusIonisation,-1,2,2);
      pmanager->AddProcess(theeplusBremsstrahlung,-1,-1,3);
      pmanager->AddProcess(theeplusAnnihilation,0,-1,4);
    }
  }
}
```

Source listing 5.2.1
注册低能电磁相互作用的电子/光子 过程

使用低能电磁相互作用过程的高级例子作为 G4 工具包的一部分一起发行，它们包含在这个文档中

要为光子和电子进行低能电磁作用过程模拟，需要将 **data files** 复制到用户的代码库中。这些文件是与 G4 工具包一起发行的。
用户因该设置环境变量 **G4LEDATA** 为他所复制的这些文件的路径。

就 G4hLowEnergyIonisation 类的公有函数来说，用于强子和离子的低能电磁作用过程有许多选择：
- SetHighEnergyForProtonParametrisation(G4double)
- SetLowEnergyForProtonParametrisation(G4double)
- SetHighEnergyForAntiProtonParametrisation(G4double)
- SetLowEnergyForAntiProtonParametrisation(G4double)
- SetElectronicStoppingPowerModel(const G4ParticleDefinition*,const G4String&)

- SetNuclearStoppingPowerModel(const G4String&)
- SetNuclearStoppingOn()
- SetNuclearStoppingOff()
- SetBarkasOn()
- SetBarkasOff()
- SetFluorescence(const G4bool)
- ActivateAugerElectronProduction(G4bool)
- SetCutForSecondaryPhotons(G4double)
- SetCutForSecondaryElectrons(G4double)

用于 ElectronicStoppingPower 和 NuclearStoppingPower 的模型在 class diagrams 文档中。同样，在 G4LowEnergyIonisation 类中，用于电子的低能电磁作用过程也有一些选择：

- ActivateAuger(G4bool)
- SetCutForLowEnSecPhotons(G4double)
- SetCutForLowEnSecElectrons(G4double)

5.2.1.3 muons 相互作用

下面是 G4 中使用的 muon 相互作用的概要。

- 韧致辐射 (类名 *G4MuBremsstrahlung*)
- 电离和 delta ray/knock on electron 产生 (类名 *G4MuIonisation*)
- 核相互作用 (类名 *G4MuNuclearInteraction*)
- 直接电子对生成 (类名 *G4MuPairProduction*)
- 能量损失 (类名 *G4MuEnergyLoss*),
  这里处理总的连续能量损失。在 muons 的情况下，韧致辐射，电离和电子对生成都对连续能量损失有贡献。

5.2.1.4 ``X 射线产生"过程

下面是 G4 中使用的 X 射线产生过程的概要。

- 切伦科夫过程 (类名 *G4Cerenkov*)
- 跃迁辐射 (类名 *G4TransitionRadiation* 和 *G4ForwardXrayTR*).

在 5.2.1.2 节中列出的低能电磁相互作用过程中也有通过荧光产生 X 射线的。

5.2.2 强相互作用

本节简要介绍在 G4 中实现的强相互作用物理过程。实现的细节请参看 **Physics Reference Manual**。

5.2.2.1 截面处理

截面数据集

每个强相互作用过程对像 (从 *G4HadronicProcess* 派生) ，可以使用一个或者多个"截面数据集"。这里的"数据集" 是指封装了方法和数据的对像，这些方法和数据是用于计算给定物理过程的总截面的。这些方法和数据可以有多种形式，从一个简单的方程，到使用大型数据表的复杂参数化信息。截面数据集，可以从抽像类 *G4VCrossSectionDataSet* 继承，并要求实现下面的方法：

```
G4bool IsApplicable( const G4DynamicParticle*, const G4Element* )
```

如果数据集可以为给定的粒子和材料计算出总的截面，那么这个方法返回 `True` ，否则返回 `False` 。

```
G4double GetCrossSection( const G4DynamicParticle*, const G4Element* )
```

这个方法只有在 `IsApplicable` 返回 `True` 的时候被调用,它返回一个给定粒子和材料的截面，单位是 G4 的缺省单位。

```
void BuildPhysicsTable( const G4ParticleDefinition& )
```

在给定粒子类型的截断值和其它参数发生变化的时候，可以调用这个方法请求重新计算数据集的内部数据库，或者复位它的状态。

```
void DumpPhysicsTable( const G4ParticleDefinition& ) = 0
```

可以调用这个方法，为给定的粒子，请求将这个数据集的内部数据库和/或者其它状态信息打印到标准输出流。

截面数据存储器 Cross Section Data Store

截面数据集，用于物理过程计算物理反应发生的长度(Physical Interaction Length)。一个给定的截面数据集，可以用于某个能量范围，或者仅用于计算一个特殊类型的粒子的截面。G4 提供了 *G4CrossSectionDataStore* 给用户，用于为一个物理过程指定一系列的数据集，并且指定数据集的优先级，以便针对不同的能量范围、粒子、和材料，使用适当的截面数据集。它实现了如下公有方法：

```
G4CrossSectionDataStore()
~G4CrossSectionDataStore()
```

构造函数和析构函数，及

```
G4double GetCrossSection( const G4DynamicParticle*, const G4Element* )
```

对于给定的粒子和材料，这个方法返回一个截面数据，这个数据是由 data store 对像中列出的截面数据集中的一个所提供的。如果不存在已知的数据集，将 throw 一个 G4Exception 异常，并返回 DBL_MIN 。否则，针对给定的粒子和材料，调用 IsApplicable 方法，以数据集列表的相反顺序，查询每个数据集。复合条件的第一个数据集将被 GetCrossSection 方法请求返回一个截面数据。如果没有数据集复合条件，将 throw 一个 G4Exception 异常，并返回 DBL_MIN。

```
void AddDataSet( G4VCrossSectionDataSet* aDataSet )
```

这个方法将给定截面数据集添加到 data store 的数据集列表末尾。该列表具有 LIFO (后进先出)优先级，意味着后添加到列表中的数据集，比先添加的有更高的优先级。换句话说，在 data store 中，当收到一个 GetCrossSection 请求后，以列表相反的方向进行 IsApplicable 查询，从最后加入列表的数据集，一直到第一个加入的数据集，复合条件的第一个数据将用来计算截面。

```
void BuildPhysicsTable( const G4ParticleDefinition& aParticleType )
```

这个方法用来向 data store 指示给定粒子类型的截断值或者其它参数已经发生了变化。作为相应，data store 将调用每个数据集的 BuildPhysicsTable。

```
void DumpPhysicsTable( const G4ParticleDefinition& )
```

这个方法可以用来请求 data store 调用它的每个数据集的 DumpPhysicsTable 方法。

缺省截面

缺省总截面数据和计算已经被封装在 singleton 类 *G4HadronCrossSections* 中。每个强相互作用过程： *G4HadronInelasticProcess*, *G4HadronElasticProcess*, *G4HadronFissionProcess*, 和 *G4HadronCaptureProcess*, 都有一个截面数据 data store 和一个缺省的寂寞数据集。这些数据集对像实际上仅仅是个调用 singelton 类 *G4HadronCrossSections* 的外壳，计算截面的实际工作由 *G4HadronCrossSections* 完成。

用户可以部分或者全部重载缺省的截面数据。最后，基类 *G4HadronicProcess* 提供了一个``get"方法：

```
G4CrossSectionDataStore* GetCrossSectionDataStore()
```

对于每个物理过程来说，data store 是公有的。按照下面的框架，用户的截面数据集可以被添加到 data store 中来：

```
G4Hadron...Process aProcess(...)

MyCrossSectionDataSet myDataSet(...)

aProcess.GetCrossSectionDataStore()->AddDataSet( &MyDataSet )
```

通过 `IsApplicable` 方法的指示，加的数据集将重载缺省的截面数据。

除了这个``get"方法之外， *G4HadronicProcess* 也提供了方法

```
void SetCrossSectionDataStore( G4CrossSectionDataStore* )
```

这个方法允许用户用一个新的 data store 完全替换缺省的 data store。

因该注意，一个物理过程不发送它本身的任何信息给它相关的 data store 对像 (因而，也不会发送给 data set 对像)。因此，假定每个数据集是用来为一种且仅为一种类型的物理过程计算截面的。当然，就像上面提到的 *G4HadronCrossSections* 类的情况，这并不禁止不同的数据集共享公用的数据和/或计算方法。确实， *G4VCrossSectionDataSet* 只是指定了一个物理过程和它们的数据集之间的抽像接口，允许用户实现任何的低层结构。

当使用 **GHEISHA** （Gamma **Hadron** Electron **Interaction** SHower code，它所实现的物理过程请参看 H. Fesefeldt, Report PITHA-85/02 (1985), RWTH Aachen）的时候，数据集 *G4HadronCrossSections* 的当前实现为弹性和非弹性散射，辐射俘获和裂变重用了总截面数据，提供截面数据用于给定粒子在给定材料中个自的平均自由程。

用于低能中子输运的截面数据

用于低能中子输运的截面数据放在一系列的文件中，它们在初始时刻由相应的数据集类读入。使用文件系统在不同的文件中存取数据。通过用户设置的环境变量 `NeutronHPCrossSections` 存取截面数据目录结构的``根"目录 。不同物理过程的总截面的类，就是用于低能中子输运的截面数据集类，有 *G4NeutronHPElasticData*, *G4NeutronHPCaptureData*, *G4NeutronHPFissionData*, 和 *G4NeutronHPInelasticData*。对于低能中子总截面的详细描述，可以由用户进行注册，就像上面用相应物理过程的 data store 描述中子相互作用一样。

因该注意，使用这些总反应截面数据类并不意味着也使用 neutron_hp 中的模型。这完全取决于用户是否期望使用它。

5.2.2.2 静止的强子 Hadrons at rest

已实现的"Hadron at Rest"过程的清单

下面的物理过程类已经被实现：

- pi- 吸收(类名 *G4PionMinusAbsorptionAtRest* 或 *G4PiMinusAbsorptionAtRest*)
- kaon- 吸收 (类名 *G4KaonMinusAbsorptionAtRest* 或 *G4KaonMinusAbsorption*)
- 中子俘获 (类名 *G4NeutronCaptureAtRest*)
- 反质子湮灭 (类名 *G4AntiProtonAnnihilationAtRest*)

- 反中子湮灭 (类名 *G4AntiNeutronAnnihilationAtRest*)
- mu- 俘获 (类名 *G4MuonMinusCaptureAtRest*)

注意，上面最后一个物理过程严格的说并不是一个``hadron at rest"过程，因为 mu- 并不是强子。虽然如此，它还是因为选择的实现模型的缘故，与清单中其它的过程共享了很多特性。对应 kaon 和 pion 吸收有两个可相互替换的实现，它们的不同只是与发射粒子谱的快成分有关。G4PiMinusAbsorptionAtRest, 和 G4KaonMinusAbsorptionAtRest 更注重描述快成分。

G4 接口的实现 Implementation Interface to Geant4

所有这些类都是从抽像类 *G4VRestProcess*.派生的。处理构造和析构函数外，已经为上面的 6 个过程实现了下列抽像类的公有方法 ：

- `AtRestGetPhysicalInteractionLength( const G4Track&, G4ForceCondition* )`
  这个方法返回反应发生前所历经的时间。上面清单中的所有过程，除 muon 俘获外，都返回 0。对应 muon 俘获过程返回 muon 俘获的寿命。
- `AtRestDoIt( const G4Track&, const G4Step& )`
  这个方法产生由这些物理过程产生的次级粒子
- `IsApplicable( const G4ParticleDefinition& )`
  这个方法返回一个检查结果，查看一个给定的粒子是否可能发生这个物理过程。

如何使用 hadron at rest 物理过程的例子

例如，在 G4 系统中，包括一个 pi 粒子的``hadron at rest" 过程，可以用以下方式完成：

- 建立一个物理过程：
-
-
```
        theProcess = new G4PionMinusAbsorptionAtRest();
```

- 向粒子的 process manager 注册这个物理过程:
-
-
```
        theParticleDef = G4PionMinus::PionMinus();
        G4ProcessManager* pman = theParticleDef->GetProcessManager();
        pman->AddRestProcess( theProcess );
```

5.2.2.3 飞行的强子 Hadrons in flight

用户需要什么样的物理过程?

对应在运动中的强子，存在四个物理过程类。Table 5.2.2.3 显示了每个物理过程和与它相关的粒子。

G4HadronElasticProcess	pi+, pi-, K^+, K^0_S, K^0_L, K^-, p, p-bar, n, n-bar, lambda, lambda-bar, Si

	gma^{+}, 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	所有粒子
G4CaptureProcess	n, n-bar

Table 5.2.2.3
强相互作用过程和相关粒子。

如何注册物理模型

要为一个粒子注册一个非弹性碰撞过程的模型，例如是一个质子，首先要获得指向这个粒子的 process mananger 的指针：

```
G4ParticleDefinition *theProton = G4Proton::ProtonDefinition();
G4ProcessManager *theProtonProcMan = theProton->GetProcessManager();
```

创建这个粒子非弹性碰撞过程的一个实例：

```
G4ProtonInelasticProcess *theProtonIEProc = new G4ProtonInelasticProcess();
```

创建这个模型的的实例，它用来决定次级粒子的生成和计算这些粒子的动量：

```
G4LEProtonInelastic *theProtonIE = new G4LEProtonInelastic();
```

向这个粒子的非弹性碰撞过程注册这个模型

```
theProtonIEProc->RegisterMe( theProtonIE );
```

最后，将这个粒子的非弹性碰撞过程加入到离散物理过程列表中：

```
theProtonProcMan->AddDiscreteProcess( theProtonIEProc );
```

上例中，这个粒子的非弹性碰撞过程类 *G4ProtonInelasticProcess* 是从 *G4HadronicInelasticProcess* 类派生的，只是简单的定义了过程的名字，并调用了 *G4HadronicInelasticProcess* 的构造函数。所有特定粒子的非弹性碰撞过程都是从 *G4HadronicInelasticProcess* 类派生的，这个类调用 `PostStepDoIt` 函数，从 *G4HadronicProcess* 的函数 `GeneralPostStepDoIt` 返回 particle change 对像。这个类同时获取平均自由程，建立 physics table，并且获取微观截面。*G4HadronicInelasticProcess* 类是从 *G4HadronicProcess* 类派生的，后者是最顶层的强相互作用过程类。*G4HadronicProcess* 类是重 *G4VDiscreteProcess* 类派生的。弹性碰撞，非弹性碰撞，俘获，裂变等物理过程都是从

G4HadronicProcess 类派生的。这个纯虚类，还提供了 energy range manager 对像和存取方法 `RegisterMe`。

source listing 5.2.2 中是一个质子的非弹性碰撞作用模型的例子，这里的 `G4LEProtonInelastic.hh` 是 include 文件的名字：

```
 ------------------------------ include file ---------------------------------
-------

#include "G4InelasticInteraction.hh"
 class G4LEProtonInelastic : public G4InelasticInteraction
 {
 public:
   G4LEProtonInelastic() : G4InelasticInteraction()
   {
     SetMinEnergy( 0.0 );
     SetMaxEnergy( 25.*GeV );
   }
   ~G4LEProtonInelastic() { }
   G4ParticleChange *ApplyYourself( const G4Track &aTrack,
                                G4Nucleus &targetNucleus );
 private:
   void CascadeAndCalculateMomenta( required arguments );
 };

 ------------------------------ source file ---------------------------------
------

 #include "G4LEProtonInelastic.hh"
 G4ParticleChange *
  G4LEProton Inelastic::ApplyYourself( const G4Track &aTrack,
                                  G4Nucleus &targetNucleus )
 {
   theParticleChange.Initialize( aTrack );
   const G4DynamicParticle *incidentParticle = aTrack.GetDynamicParticle();
   // create the target particle
   G4DynamicParticle *targetParticle = targetNucleus.ReturnTargetParticle();
   CascadeAndCalculateMomenta( required arguments )
   { ... }
   return &theParticleChange;
 }
```

Source listing 5.2.2
一个质子非弹性作用模型类的粒子。

`CascadeAndCalculateMomenta` 是这个模型的主要部分，应有模型创建者提供。它决定了在反应中生成那些次级粒子，并为这些粒子计算动量，最后将这些信息放到 *ParticleChange* 对像中返回。

G4LEProtonInelastic 类是从 *G4InelasticInteraction* 类派生的，后者是一个纯抽像基类，因为它的虚函数 `ApplyYourself` 并未被定义。*G4InelasticInteraction* 本身是从抽像基类 *G4HadronicInteraction* 派生的。后者是所有物理模型类的基类，它为不同的模型选择能量范围，提供一些实用类。*G4HadronicInteraction* 类提供了 `Set/GetMinEnergy` 和 `Set/GetMaxEnergy` 方法用来确定模型使用的最小和最大能量范围。可以为一个特定的元素、材料，或者其它通用情况（非特定元素、材料）指定能量范围：

```
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 )
```

存在那些模型，那些是缺省的模型

在 G4 中，任何模型都可以与其它模型一起运行，并不需要实现任何特殊的接口，不同模型作用的范围可以在初始化的时候进行控制。这样，一些高度特殊的模型(只应用于一种材料和粒子，并且只适用与一个非常严格的能量范围)可以跟其它通用模型一起，以一致的方式，在同一个应用程序中使用。

每个模型有它固有的适用范围，为一个模拟选择正确模型依懒于使用情况。因此，并没有``缺省"的模型。但是，为各种目的指定各种物理模型集合的 Physics lists 将在适当的时候提供。

我们已经实现了三种类型的强子簇射模型，它们是参数驱动模型（parametrisation driven models），数据驱动模型（ data driven models），和理论驱动模型（ and theory driven models）。

- 参数驱动模型　用于所有与将要静止，并与原子核发生作用的粒子有关的物理过程。对于飞行中的粒子，有两个模型集用于非弹性散射；低能，和高能模型。两个模型集最初都居于 Geant3.21 的 **GHEISHA** 工具包，对于核激发，核内级联过程和蒸发等基本的作用过程，使用了最初的处理方法。这些模型在子目录 `hadronics/models/low_energy` 和 `hadronics/models/high_energy` 下。低能模型的目标是针对能量低于 20 GeV 的情况；高能模型覆盖了从 20 GeV 到 O(TeV)的能量范围。裂变，俘获和相干弹性散射也是用参数化模型实现的。
- 数据驱动模型　用于低能中子在物质中的输运，位于子目录 `hadronics/models/neutron_hp` 中。是根据 **ENDF/B-VI** 的数据格式进行建模的，这个标准数据格式中的所有已发行版本都已实现。使用的数据集是从遵从这些标准格式

的数据库中选择的。使用文件系统对用于不同同位素，不同反应通道的截面数据进行存取。这些模型覆盖的能量范围从 thermal energies 一直到 20 MeV。
- 理论启动模型　在第一个实现版本中用于非弹性散射，覆盖整个 LHC 实验的能量范围。它们在子目录 `hadronics/models/generator` 中。目前的理论暗示了在高能情况下使用 parton 弦模型，在中能情况下，使用 intra-nuclear transport 模型，以及对于退激情况使用统计分裂模型（statistical break-up models）。

5.2.3 粒子衰变过程

这节简要介绍 G4 中使用的衰变过程。有关粒子衰变过程的实现细节，请参看 **Physics Reference Manual**。

5.2.3.1 粒子衰变过程类

G4 提供了一个 *G4Decay* 类，用于``静止(at rest)"和``飞行(in flight)"的粒子衰变。*G4Decay* 可以应用于除以下粒子外的所有粒子：

无质量的粒子	`G4ParticleDefinition::thePDGMass <= 0`
具有``负的" 寿命的粒子	`G4ParticleDefinition::thePDGLifeTime < 0`
短寿命粒子	`G4ParticleDefinition::fShortLivedFlag = True`

用户可以使用 `G4ParticleDefinition::SetPDGStable()` 和 *G4ProcessManager* 提供的方法 `ActivateProcess()` 与 `InActivateProcess()`来为某种粒子打开/关闭相应的粒子衰变。

除了在 *G4DynamicParticle* 中定义的 `PreAssignedDecayProperTime` 之外，*G4Decay* 根据粒子的寿命来估计步长（对于 `AtRest` 的粒子是时间步长）。

G4Decay 类没有定义粒子的衰变模型。G4 提供了两种方法来确定衰变模型：

- 在 *G4DecayTable* 中使用 *G4DecayChannel*
- 使用 *G4DynamicParticle* 的 `thePreAssignedDecayProducts`

G4Decay 类只负责计算 `PhysicalInteractionLength` 和产生衰变产物，这些衰变产物是由 *G4VDecayChannel* 或事件发生器建立的。有关衰变模型的确定请参看下面的信息。

多个粒子可以共享一个 *G4Decay* 对像。在 Source listing 5.2.3 中，显示了在 *PhysicsList* (参看 2.5.3 节)中的 `ConstructPhysics` 方法内部进行粒子衰变过程的注册。

```
#include "G4Decay.hh"
void ExN02PhysicsList::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);
    }
  }
}
```

Source listing 5.2.3
在 *PhysicsList* 中的 `ConstructPhysics` 方法内部进行粒子衰变过程的注册。

5.2.3.2 衰变表 Decay table

每个粒子都有它自己的 *G4DecayTable*，这个 *G4DecayTable* 保存了与粒子衰变模式有关的信息。每种衰变模式，和它的分支比，对应与一个从 *G4VDecayChannel* 类派生的``衰变通道(decay channel)"类的对像 。缺省的衰变模式在粒子类的构造函数中建立。例如，neutral pion 的衰变表中有 *G4PhaseSpaceDecayChannel* 和 *G4DalitzDecayChannel* ，实现如下：

```
// 建立一个衰变通道
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);
```

在 G4 中定义的衰变模式和分支比的清单在 5.3.2 节。

5.2.3.3 由事件发生器事先指定的衰变模式

重味粒子(heavy flavor particles)的衰变是非常复杂的，例如，B 介子，有许多不同的衰变模式和衰变机制。重粒子衰变的许多模式有不同的事件发生器给出。要使用 *G4VDecayChannel* 定义重粒子的所有衰变模式是不可能的。换句话说，重粒子的衰变不能通过 G4 衰变过程来

定义，只能通过事件发生器和外部工具包来定义。所以，G4 提供了两种方式来实现这个目的，`pre-assigned decay mode` 和 `external decayer`.

后一种实现方式，使用 *G4VExtDecayer* 类与一个外部软件包进行接口，用这个软件包来定义一个粒子的衰变模式。如果将一个 *G4VExtDecayer* 联接到 *G4Decay*，那么次级粒子将由外部衰变处理程序产生。

在前一种方式中，通过一个事件发生器和一个拥有衰变信息的初级事件来模拟重粒子的衰变的。*G4VPrimaryGenerator* 自动将这些次级粒子作为 *G4DynamicParticle* 的成员 PreAssignedDecayProducts 链接到父粒子。*G4Decay* 采用这些事先指定的次级粒子来代替请求 *G4VDecayChannel* 生成衰变产物。

另外，用户可以为一个特定的 track，在它的静止参考系内，使用 *G4PrimaryParticle::SetProperTime()*方法指定 `pre-assigned` 衰变时间(即，衰变是由本征时间定义的)。*G4VPrimaryGenerator* 设置 *G4DynamicParticle* 的成员 PreAssignedDecayProperTime。*G4Decay* 使用这个衰变时间来代替这种粒子的寿命。

5.2.4 Photolepton-hadron processes

To be delivered.

5.2.5 光学光子过程

当一个光子的波长大到可以与原子间的空隙相比的时候，就应当考虑它的*光学特性*。在 G4 中，光学光子作为一个粒子类来处理的，它区别于相对比较高能量的 *gamma* 光子。因为这个理论描述在能量较高的时候不成立，所以在光学光子与 gamma 粒子之间，并没有一个光滑的过渡。

G4 处理光学光子过程的模块包括了在介质边界的折射和反射，部分吸收和瑞利散射。这个产生光学光子的物理过程，包括切伦科夫效应，跃迁辐射和闪烁(scintillation)。G4 产生的光学光子不遵从能量守恒，它们的能量不能计入一个事件的能量平衡的。

介质的光学特性是实现这些类型的物理过程的关键，它们作为条目存贮在 `G4MaterialPropertiesTable` 。这个 `G4MaterialPropertiesTable` 在问题中被链接到 `G4Material`。这些特性可以是常数，也可以是跟波长相关的一个函数。这个表是 `G4Material` 类的一个私有数据成员。`G4MaterialPropertiesTable` 是作为一个散列目录实现的，在这个表中的每个条目，都是由一个*值*和一个*键*组成的。键用来快速有效的读取对应的值。在目录中的所有值，或者是 `G4double` 的实例，或者是 `G4MaterialPropertyVector` 的实例，而所有键都是 `G4String` 类型。

G4MaterialPropertyVector 是由 G4MPVEntry 的实例组成的。G4MPVEntry 是一对数字，在表示光学特性的时候，是光子动量和对应的属性值。G4MaterialPropertyVector 是作为一个 G4std::vector 实现的，使用了如 $MPVEntry_1 < MPVEntry_2 == photon\_momentum_1 < photon\_momentum_2$ 的排序操作。 在所有 G4MaterialPropertyVectors 中的结果是以升序排列的光子动量。用户可以使用 G4MaterialPropertiesTable 提供的方法，向材料添加许多材料(光学)特性。source listing 5.2.4 中就是这方面的一个例子。

```
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};
MPT -> AddProperty("ABSLENGTH",ppckov,absorption,NUMENTRIES};

scintillator -> SetMaterialPropertiesTable(MPT);
```

Source listing 5.2.4
将光学特性添加到一个 4MaterialPropertiesTable，并链接到一个 G4Material。

5.2.5.1 在 processes/electromagnetic/xrays 中- 切伦科夫光子的产生

当一个带电粒子介质中运动，且它的速度大于在该介质中光的群速时，将发生切伦科夫光辐射。发射的光子形成一个光锥，它的张角与粒子减速运动时的瞬时速度有关。同时，随着速度的降低，发射光子的频率降低，光子数量减少。当粒子的速度降到比介质中的光速还低的时候，辐射停止，光锥塌缩为零。这个过程产生的光子有一个固有的偏振方向，它们垂直于光锥的表面。

在类 G4Cerenkov 的方法 AlongStepDoIt 中，切伦科夫光的发射，偏振，谱，通量服从一些已知的公式，它存在两个固有的局限。第一个出现在用 step-wise 方式模拟的时候，第二个出现在要求用数值积分计算每一步(step)切伦科夫光的平均光子数 时。这个过程使用一个 G4PhysicsTable ，它包含了用于加速这个计算的增量积分。

切伦科夫光发射的时间和位置是根据带电粒子在每一步开始时刻的已知量来计算的。每一步(step)都被假定是直线，即使有磁场的存在。用户可以使用方法 SetMaxNumPhotonsPerStep(const G4int NumPhotons)，来指定在一步中产生的最大(平均)

切伦科夫光子数，从而限制步长。由于光子的产生服从泊松分布，实际产生的数目将是不同的。在当前的实现中，光子产额密度是沿粒子的各个径迹段(step)平均分布的，即使在一个 step 中，它的速度已经急剧降低。

通常，在一个 step 中就会产生大量的次级粒子 (在水中约 300/cm)，迫使在使用 GEANT3.21 中的发法，暂停初级粒子跟踪，直到所有的次级粒子已被跟踪完成。尽管事实上 G4 使用了动态内存分配，而不存在像 GEANT3.21 那样，需要使用大量固定的 ZEBRA 存储器的限制，然而 GEANT4 还是提供了公有方法 `SetTrackSecondariesFirst`，用于实现类似的功能。source listing 5.2.5 是注册切伦科夫过程的一个例子。

```
#include "G4Cerenkov.hh"

void ExptPhysicsList::ConstructOp(){

  G4Cerenkov*   theCerenkovProcess = new G4Cerenkov("Cerenkov");

  G4int MaxNumPhotons = 300;

  theCerenkovProcess->SetTrackSecondariesFirst(true);
  theCerenkovProcess->SetMaxNumPhotonsPerStep(MaxNumPhotons);

  theParticleIterator->reset();
  while( (*theParticleIterator)() ){
    G4ParticleDefinition* particle = theParticleIterator->value();
    G4ProcessManager* pmanager = particle->GetProcessManager();
    G4String particleName = particle->GetParticleName();
    if (theCerenkovProcess->IsApplicable(*particle)) {
      pmanager->AddContinuousProcess(theCerenkovProcess);
    }
  }
}
```

Source listing 5.2.5
在 `PhysicsList` 中注册切伦科夫过程。

5.2.5.2 在 `processes/electromagnetic/xrays` 中- 闪烁光子的产生

每种闪烁材料都会发射特征光 `SCINTILLATIONYIELD`，而且都有一个本征分辨率 `ResolutionScale`，这个分辨率通常会使产生的光子的统计分布展宽。宽的本征分辨率是由于杂质引起的。比较典型的情况是一些掺杂的晶体，如 NaI(Tl) 和 CsI(Tl)。另一方面，当法诺因子起作用的时候，本征分辨率也可以变窄。在一个 step 内，实际发射的光子数目总是围绕着平均值涨落，这个涨落的宽度是 `ResolutionScale*sqrt(MeanNumberOfPhotons)`。平均

发射的光子 MeanNumberOfPhotons，与局部能量沉积有一个线性关系，但是对于 minimum ionizing 和 non-minimum ionizing 粒子，结果会有所不同。

不同的闪烁体有不同的发射谱和不同的衰减时间。在 G4 中，闪烁体可以有一个快成分和一个快成分。其中，快成分相对于总闪烁光的强度由 YIELDRATIO 确定。可以为每种闪烁材料指定这些经验参数，从而模拟它们的闪烁发光。在用户的 DetectorConstruction 类中，可以为闪烁材料指定一个相对谱分布，这个分布是光子能量的函数。source listing 5.2.6 就是这个问题的例子。

```
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);
```

Source listing 5.2.6
在 DetectorConstruction 中指定闪烁特性。

当闪烁体的发光跟粒子的类型有关时，用户可以为它们定义不同的闪烁过程。在 source listing 5.2.7 中，显示了如何用 ScintillationYieldFactor 在用户的 PhysicsList 中为材料指定发光强度。在那些因粒子类型不同而使快慢成分发生变化的情况中，可以为每一个闪烁过程调用方法 SetScintillationExcitationRatio(参看高级例子 underground_physics)，它将改写从 G4MaterialPropertiesTable 获得的 YieldRatio。

```
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);
     }
  }
}
```

Source listing 5.2.7
在 PhysicsList 中闪烁过程的实现。

在每一步(step)中，将根据能量损失，产生一个服从高斯分布的光子数目。如果分辨率为 1.0，将产生一个围绕平均光产额的统计涨落，这个平均光产额由 AddConstProperty("SCINTILLATIONYIELD")设置，当分布率大于 1 的时候，涨落将展宽。如果分辨率为 0，表明没有涨落。每个光子的频率是从经验谱中采样得到的。光子沿粒子径迹段(step)均匀产生，并且在 4π角内性均匀发射，这些光子有随机的线偏振方向和与闪烁成分相关的时间特性。

5.2.5.3 processes/optical 中的光子跟踪

吸收

G4OpAbsorption 实现了光学光子的部分吸收，它仅仅 kill 了那些光子。这个程序要求用户使用公有方法 AddProperty 中的 ABSLENGTH 作为属性键，来使用吸收长度的经验数据填充相应的 G4MaterialPropertiesTable 。吸收长度是指光子在被介质吸收之前所传播的平均距离；也就是由方法 GetMeanFreePath 返回的平均自由程。

瑞利散射

在瑞利散射中的微分截面，σ/ω ，正比于 $\cos^2(\theta)$，这里的θ 是新的偏振方向与原来的偏振方向之间的夹角。瑞利散射过程 G4OpRayleigh 采样这个角，然后计算散射后光子的方向。光子新的方向必须与这个光子新的偏振方向垂直，而且与原来的偏振方向和新的偏振方向都在

同一平面内。因此，这个过程跟光子的偏振方向(自旋)有关。光子的偏振数据是类 G4DynamicParticle 的一个数据成员。

一个在产生的时候没有指定偏振方向的光子是不能进行瑞利散射的。用户可以通过 G4PrimaryParticle 类的方法 SetPolarization，或者间接的通过 G4ParticleGun 类的方法 SetParticlePolarization 指定光子的偏振方向。切伦科夫过程产生的光学光子，在产生的时候有一个固有的垂直于光锥表面的偏振方向。闪烁光子有一个垂直于它们传播方向的偏振方向。

这个过程要求用户使用瑞利散射长度数据来填充一个 G4MaterialPropertiesTable，要不就使用方法 RayleighAttenuationLengthGenerator。 瑞利散射衰减长度是指光子在介质中发射瑞利散射之前所传播的平均距离，它就是由 GetMeanFreePath 方法返回的平均自由程。类 G4OpRayleigh 提供了一个方法 RayleighAttenuationLengthGenerator ，它用来计算服从 Einstein-Smoluchowski 方程的介质的衰减系数，这个系数依懒于温度和介质的等温压缩率。当瑞利散射衰减长度不能从测量得到，但可以使用上述的材料常数进行计算时，这个 generator 方法是很方便的。

边界过程

[E. Hecht and A. Zajac, Optics, Addison-Wesley Publishing Co., pp. 71-80 and pp. 244-246, 1974.]

对于两种介电材料的界面是完全光滑的简单情况，用户只需要在它们个自的 G4MaterialPropertiesTable 中提供它们的折射系数。对于其它情况，光学边界过程的设计依懒于 *surfaces* 的概念。相关的信息在两个类中。一个类在材料模块中，它保存了有关 surface 本身的物理特性的信息。第二个类在几何模块中，它拥有一个指向相关的物理体和逻辑体的指针，并且与物理类相关联。第二中类型的 surface 对像被保存在一个相关表中，并且可以用指定与 surface 相接触的物理体的两个有序对来读取，也可以通过被这个 surface 完全包围的逻辑体来读取。前者叫边界(*border surface*)，后者被称为表面(*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. skin surface 只能有一个，而且所有被它所包围的 volume 的侧面只有一种相同的光学特性。border surface 是物理体的一个有序对，所有从原理上将，用户可以为同一界面的两个方向选择不同的光学特性。如果光学边界过程要使用一个 border surface，那么对应的这两个 volumes 必须是用 G4PVPlacement 放置的。这个有序组合可以存在于任何地方。当不需要使用 surface 概念，并且在两个介电材料间有一个完全光滑的介面的时候，唯一相关的特性就是折射系数，它与材料保存在一起，这时就不存在如何放置这些 volumes 的问题了。

physical surface 对像指定了那个边界过程模型将用于模拟与 surface 的相互作用。另外，physical surface 可以有一个包含它自己所有材料的材料属性表 。这个表允许所有的镜面常数与波长相关。在表面被着色或者包裹(并未涂覆)的情况下，这个表可以包含这个薄层的折射系数。这可以用来模拟介质和表面层相交时的边界效应，和在薄层远侧的朗伯反射。这发生在边界过程本身，不调用 G4Navigator。各种表面加工特性，如 *polished* 或 *ground* 和 *front painted* 或 *back painted*，它们的组合枚举了可以被模拟的不同情形。

当一个光子到达一个介质边界的时候，它的行为依懒于连接这这个边界的两种材料的特性。介质边界可以由两种介电材料或一种介电材料和一种金属形成。在两种介电材料的情况下，光子根据它的波长、入射角和边界两边的折射系数，可以经历内部的全反射，折射或者反射。

此外，反射和透射概率是与光子的线偏振态有关的。在一种介电材料和一种金属形成界面的情况，光子可以被金属吸收，或者反射回介电材料中。如果光子被吸收，那么按照金属的光电子效率，光子可能被删除。

如麦克斯韦方程所表达的，菲涅耳反射和折射通过它们相关的发生概率联系在一起。因此，这两个过程，还有全反射，它们都不能看作是独立的物理过程，不值得实现个自独立类。虽然如此，G4 还是在一些可行的地方，通过将代码放到不同的方法中，对过程独立化的抽像进行了一些尝试。

类 `G4OpBoundaryProcess` 的实现使用了 UNIFIED 模型 [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.]. 它适用于介电材料－介电材料界面，并且试图提供一个真实的模拟，处理表面加工和反射涂层的所有方面。这个表面可以被假定是光滑的，并由一个用给定反射系数，表示镜面反射的金属化涂层覆盖，或者用发生朗伯反射的漫反射材料涂覆。这些表面可以与其它的元件有光学接触，也可以没有。对于非常粗糙的表面，有可能出现在发生反射之后，同一个光子反过来又在同一表面上发生折射，因而，在物理过程本身内部，在边界上发生多次相互作用是可能的，并且不需要 `G4Navigator` 进行重新定位。

UNIFIED 模型规定了一个不同反射机制的范围。specular lobe 常数表示在一个微面元法线附近的反射概率。接着，specular spike 常数表示在平均表面法线附近的反射概率。diffuse lobe 常数表示内部朗伯反射的概率，最后，back-scatter spike 常数表示在深槽中近多次反射导致背向散射的最终结果的概率。这 4 个概率的和必须为 1， diffuse lobe 常数是被隐含的。读者可以查询参考书目了解这个模型的详情。

```
G4VPhysicalVolume* volume1;
G4VPhysicalVolume* volume2;

G4OpSurface* 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);
```

Source listing 5.2.8
由 *G4OpticalSurface* 定义介电材料—介电材料的表面特性。

这个过程最初在 GEANT3.21 中的实现 可以通过 GLISUR 方法标志来使用。 [GEANT Detector Description and Simulation Tool, Application Software Group, Computing and Networks Division, CERN, PHYS260-6 tp 260-7.].

```
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);
```

Source listing 5.2.9
由 *G4OpticalSurface* 定义介电材料金属表面特性。

如果用户不指定模型和表面加工特性，程序缺省使用 GLISUR 模型和 *polished* 表面加工。在介电材料－金属界面的情况下，或者当指定 GLISUR 模型的时候，可选择的表面加工只有 *polished* 和 *ground*。

5.2.6 参数化过程

在这节中，讲述了如何使用 G4 的参数化，或者“快速模拟”工具。例子在 **examples/novice/N05** 。

5.2.6.1 一般性:

G4 的参数化工具允许用户更加容易的完成一个给定 volume 和给定粒子的详细粒子跟踪，

用户将要进行参数化的 volume 被称为 ***envelope***。一个 envelope 可以有一个几何子结构，它的子体，和孙子体(……)中的所有点都在这个 envelope 中。

Envelopes 通常对应于一些子探测器的 volumes：电磁量热器，径迹室等。使用 G4，可以通过覆盖一个 parallel 几何体或者”ghost”几何体来定义 envelopes，参看 5.2.6.8 节。

在 G4 参数化中有三个主要特性，用户必须指定：

- 用户的 parameterisation 可以用于哪些粒子类型；
- 用户的 parameterisation 可以用于哪些 dynamics conditions，并且必须被 triggered;
- The parameterisation properly said: 在什么地方删除初级粒子，在什么地方建立次级例子，等，还有，在什么地方计算探测器响应。

GEANT4 将在 envelope 的 volume 中，在每一步开始的时候，向用户的 parameterisations 代码发送消息。首先将向当前粒子请求可用的 parameterisations，如果，有且仅有一个 parameterisation 希望 issue a trigger，那么调用这个粒子的 parameterisation code properly said 。在这种情况下，tracking 将**不应用物理过程**到这一步中的粒子。不过，将会调用 UserSteppingAction。

Parameterisations 看上去就像一个"user stepping action"，但是更加高级，因为：

- 只有在用户绑定的 envelope 中，可以向用户的 parameterisation 代码发送消息；

- 在 envelope 中的任何位置，可以向用户的 parameterisation 发送代码，即使这个 track 位于它的子体中；
- 用户可以用非常灵活的方式将 parameterisations 绑定到 envelopes。
- GEANT4 将为用户的 parameterisation 代码提供有关 envelope 的信息。

5.2.6.2 parameterisation 元件一览：

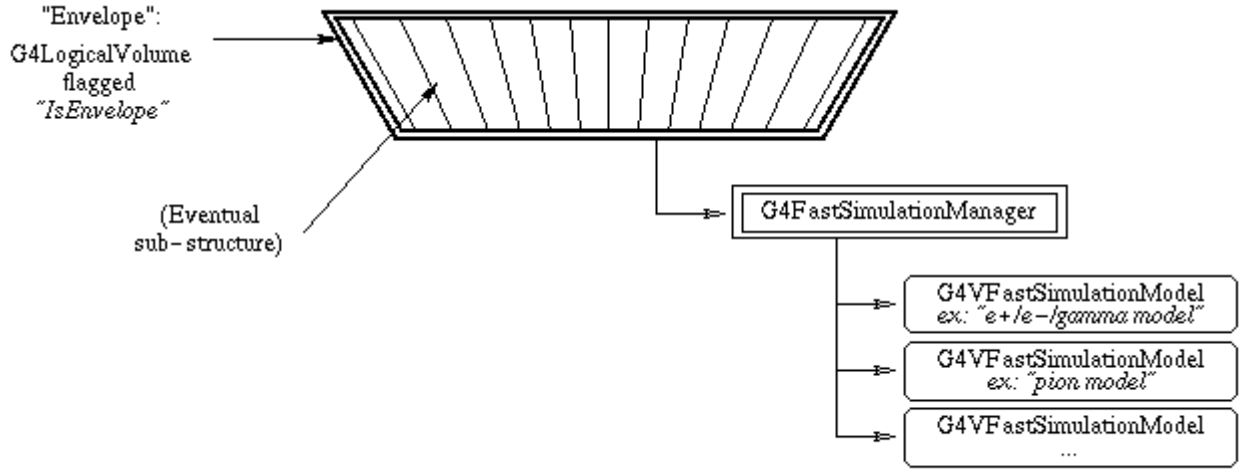
允许用户实现和控制 parameterisations 的 GEANT4 元件：

- **`G4VFastSimulationModel`** 这是用于实现 parameterisations 的抽像类 。用户必须从它继承并实现自己的具体 parameterisation 模型。

- **`G4FastSimulationManager`** G4VFastSimulationModel 对像是通过一个 G4FastSimulationManager 联接到 envelope 的。这个 G4FastSimulationManager 对像将管理一系列的模型，并在粒子跟踪的时候发消息给它们。

- **`Envelope`** 在 G4 中的一个 envelope 是是一个 **`G4LogicalVolume`** 对像，它只是简单的通过设置标志表明它是一个 envelope。 通过设置一个指向 parameterisation 的 G4FastSimulationManager 指针，将它绑定到 envelope。

  下图显式了 G4VFastSimulationModel 和 G4FastSimulationManager 对像是如何绑定到 envelope 的：



- **`G4FastSimulationManagerProcess`** 这是一个 G4VProcess。它提供 tracking 和 parameterisation 之间的接口。用户必须在它期望参数化的粒子的物理过程列表中对它进行设置。(可以预见有一个自动的方式用于为适当的粒子设置这个过程)

- **`G4GlobalFastSimulationManager`** 这是一个 singleton 类，它提供了对 G4FastSimulationManager 对像和一些 ghost 工具的管理。

5.2.6.3 `G4VFastSimulationModel` 抽像类：

- **构造函数：**

  G4VFastSimulationModel 类有两个构造函数。第二个构造函数允许用户快速的"getting started".

  - **`G4VFastSimulationModel(const G4String& aName)`:** 这里的 aName 用于标识 parameterisation 模型。

  - **`G4VFastSimulationModel(const G4String& aName, G4LogicalVolume*, G4bool IsUnique=false)`:** 除了模型名外，这个构造函数还接受一个 G4LogicalVolume 指针。这个 volume 将自动成为 envelope，并且如果需要的话，那个必需的 G4FastSimulationManager 对像将被建立。如果这个模型已经存在，它将被添加到这个 manager。然而，这个 *G4VFastSimulationModel 对像不将保存在构造函数中给出的 envelope 的 track*。
    布尔参数是用于优化目的的：如果用户指定 G4LogicalVolume envelope 只被放置了一次，用户可以将这个参数设置为"true" (在这里可以预见实现一个自动机制。)

- **虚方法:**

  G4VFastSimulationModel 有三个纯虚方法，用户必须在自己的具体类中进行重载：

  - **`G4bool IsApplicable(const G4ParticleDefinition&)`:** 在用户的实现中，当用户的模型可以被传递给这个方法的 G4ParticleDefinition 所接受时，返回 "true"。G4ParticleDefinition 所有固有的粒子信息 (质量，电荷，自旋，名字…)。

    在用户希望实现一个用于精确的粒子类型的模型时，由于效率的原因，推荐使用相应粒子类的静态指针
    例如，在一个只用于 *gamma*s 的模型中， IsApplicable() 方法应采用的形式：

```
o
o     #include "G4Gamma.hh"
o     G4bool MyGammaModel::IsApplicable(const G4ParticleDefinition&
partDef)
o     {
o       return &partDef == G4Gamma::GammaDefinition();
o     }
```

  - **`G4bool ModelTrigger(const G4FastTrack&)`:** You have to return "true" when the dynamics conditions to trigger your parameterisation are fulfiled.
    G4FastTrack 提供用户对当前 G4Track 进行存取，存取 envelope 相关的特性 (G4LogicalVolume, G4VSolid, G4AffineTransform references between the global and the envelope local coordinates systems) 和在 envelope 坐标系内的位置，动量。通过这些量和 G4VSolid 的方法，用户可以方便的检查目前离 envelope 边界有多远。

  - **`void DoIt(const G4FastTrack&, G4FastStep&)`:** Your parameterisation properly said. G4FastTrack 引用提供了输入信息。粒子经过参数化之后的最终状

态必须通过 G4FastStep 引用返回。在参数化调用完成之后，将对这个最终状态进行 tracking。

5.2.6.4 `G4FastSimulationManager` 类:

(在 5.2.6.8 节，将讲述 G4FastSimulationManager 有关使用 ghost volumes 的功能。)

- **构造函数：**
  - **`G4FastSimulationManager(G4LogicalVolume *anEnvelope, G4bool IsUnique=false)`**：这个是唯一的一个构造函数。在这个函数内，用户通过给定 G4LogicalVolume 指针指定 envelope。G4FastSimulationManager 对像将自身绑定到这个 envelope，并且通知这个 G4LogicalVolume 成为一个 envelope。如果用户知道这个 volume 只被放置一次，用户可以设置 IsUnique 为"true"，从而进行一些优化。(这里我们可以预见存在一种自动化的机制。)
    注意，如果用户为自己的模型选择使用 G4VFastSimulationModel(const G4String&, G4LogicalVolume*, G4bool)这个构造函数，那么，将使用这个模型的构造函数给定的 G4LogicalVolume*和 G4bool 值创建 G4FastSimulationManager。
- **G4VFastSimulationModel 对像管理：**
  - **`void AddFastSimulationModel(G4VFastSimulationModel*)`**
  - **`RemoveFastSimulationModel(G4VFastSimulationModel*)`**
    这两个方法提供了那些常用的管理功能。
- **和 G4FastSimulationManagerProcess 的接口：**

  这请参阅 User's Guide for Toolkit Developers (section 3.9.6)

5.2.6.5 "Envelope":

`G4LogicalVolume` 类是在 4.1.3 节中论述了的。这里只关注跟 envelope 有关的方面。

- **打开 envelope 模式:**

  将一个 G4LogicalVolume 设置为一个 envelope 的过程是透明的：当一个 G4FastSimulationManager 对像被建立的时候，它使用在构造函数中给定的 G4LogicalVolume 指针调用 G4LogicalVolume 的方法：

  `void BecomeEnvelopeForFastSimulation(G4FastSimulationManager*)`

- **关闭 envelope 模式:**

如果用户需要关闭一个 G4LogicalVolume 的 envelope 模式，用户必须调用：

`void ClearEnvelopeForFastSimulation()`.

- **局限性：**

  使用 envelope 定义的当前实现，存在一些小的限制：

  一个被放置在一个 envelope 层次树中的 G4LogicalVolume，不能在放置到这个 envelope 的其它地方。

  这个限制的原因是，由于效率的原因，为了实现在 tracking 时的快速检查(在这种情况下，事实上不会发出警告信息！)，一个 envelope 的 G4FastSimulationManager 指针被递归传递给它子体的 G4logicalVolume。

- **Ghost Envelopes:**

  Ghost envelopes 是定义在一个 parallel 几何体中的 envelopes。像 tracking 几何体中的 envelopes 一样，用户必须给它们设置一个 G4FastSimulationManager 对像和其它的 G4VFastSimulationModel 对像。

  我们将在 5.2.6.8 节中解释如何建立和使用 ghost envelopes。

5.2.6.6 `G4FastSimulationManagerProcess`:

G4VProcess 作为 tracking 和 parameterisation 之间的一个接口。在 tracking 的时候，它与当前 volume 可能存在的 G4FastSimulationManager 一起工作，允许那些模型触发。如果 manager 不存在，或者没有模型发出触发信号，tracking 将正常的进行。

在现存的实现中，用户必须在用户参数化的粒子的 G4ProcessManager 中设置这个 process，以使能用户的 parameterisation。

(由于 G4 拥有所有必要的信息，所以，可以预见一个在相关粒子的 G4ProcessManager 中放置 process 的方式。)

这些(物理)过程的顺序是：

```
[n-3] ...
[n-2] Multiple Scattering
[n-1] G4FastSimulationManagerProcess
[ n ] G4Transportation
```

这个顺序在用户使用 ghost 几何体时是重要的，因为 G4FastSimulationManagerProcess 将提供在 ghost world 中的 navigation，以限制在 ghost 边界上的 step。

如果用户为一个粒子使用了 ghost 几何体，G4FastSimulationManager 必须作为一个连续离散

(continuous and discrete)过程被添加到这个粒子的过程列表中。如果用户不使用 ghosts，可以作为一个离散过程添加。
下面的代码给所有的粒子设置 G4FastSimulationManagerProcess，这个 process 是作为连续离散过程被设置的：

```
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);
    }
}
```

5.2.6.7 `G4GlobalFastSimulationManager` singleton 类:

这个类是一个 singleton 类。用户可以通过以下方式对它存取：

```
#include "G4GlobalFastSimulationManager.hh"
...
...
G4GlobalFastSimulationManager* globalFSM;
globalFSM = G4GlobalFastSimulationManager::getGlobalFastSimulationManager();
...
...
```

目前，如果用户使用 ghost 几何体，主要需要使用 GlobalFastSimulationManager。

5.2.6.8 使用 ghost 几何体的 Parameterisation：

在一些情况下，tracking 几何体的 volumes 不允许用来定义 envelopes。这可能发生在几何体是来自 CAD 系统的情况。由于这样的几何体是没有层次结构的，用户需要使用一个 parallel 几何体来定义 envelopes。
注意这种情况，用户希望进行带电 pions 的参数化，通过将探测器的电磁和强子量热器组织在一起的 volume 定义 evelope。用户可能不希望这个 envelope 对电子是可见的，这意味着这些 ghost 几何体必须按粒子 flavour 来进行编组。

因为通过 G4FastSimulationManagerProcess 在 ghost 几何中提供了一个 navigation，所以，对那些对 ghosts 敏感的粒子来说，使用 ghost 几何意味着在参数化机制方面有更多的开销。但

是值得注意的是，由于用户在 ghost world 中只放置比较少的 volumes，有关几何体的计算开销因该是相当少的。

在目前的实现中，用户不需要显式的建立 ghost 几何体，G4GlobalFastSimulationManager 将完成这个工作。首先，建立一个 world 的空的"clone"，这个 world 是用于 tracking 的，是由用户的具体类 G4VUserDetectorConstruction 的方法 construct()提供的。用户将在 G4FastSimulationManager 对像中提供 envelope 相对 ghost world 坐标的位置。通过与一个 ghost envelope 相关的 G4FastSimulationManager 中保存的一个非空的位置列表，可以识别这个 ghost envelope。
G4GlobalFastSimulationManager 将使用这些位置信息和那些被连接到 G4FastSimulationManager 对像的模型的 IsApplicable()方法，建立与 flavour 相关的 ghost 几何体。
然后，在一个粒子 tracking 开始的时候，如果存在 ghost world，那么将选用适当的 ghost world。

下面是用户建立一个 ghost envelope 必须提供的步骤：

1. 建立一个 envelope (即一个 G4LogicalVolume)： myGhostEnvelope;
2. 建立一个 G4FastSimulationManager 对像，myGhostFSManager, 使用 myGhostEnvelope 作为它构造函数的参数；
3. 通过为每一个 envelope 调用 G4FastSimulationManager 的方法，将 myGhostEnvelope 的所有位置信息传递给 G4FastSimulationManager：
4.

```
AddGhostPlacement(G4RotationMatrix*, const G4ThreeVector&);
```

   或者:

```
AddGhostPlacement(G4Transform3D*);
```

   这里 3D 变换的旋转矩阵，平移向量描述了相对与 ghost world 坐标的相对位置。

5. 建立用户的 G4VFastSimulationModel 对像，并将它们添加到 myGhostFSManager.
   用户模型的 IsApplicable()方法将通过 G4GlobalFastSimulationManager 被使用，用于为一中给定的粒子类型建立相应的 ghost 几何体。
6. 调用 G4GlobalFastSimulationManager 的方法：
7.
8.
9.

```
G4GlobalFastSimulationManager::getGlobalFastSimulationManager()->
    CloseFastSimulation();
```

这个最终的调用将使 G4GlobalFastSimulationManager 建立与粒子 flavour 相关的 ghost 几何体。这个调用过程必须在 RunManager 关闭几何体之前完成。(可以预见，在将来，run manager 将调用 CloseFastSimulation()，与关闭几何体进行适当的同步。)

G4 为 ghosts 几何体提供了可视化工具。在 CloseFastSimulation()启动之后，可以在交互式接口中请求显式 ghosts。基本的命令是：

- /vis/draw/Ghosts particle_name
  显式与指定粒子相关的 ghost 几何体。
- /vis/draw/Ghosts
  显示所有的 ghost 几何体。

5.2.7 输运过程

To be delivered by J. Apostolakis (John.Apostolakis@cern.ch)

5.3 粒子

5.3.1 常用概念

有三个层次的类用于在描述 G4 中使用的粒子。
G4ParticleDefinition 定义一个粒子
G4DynamicParticle 描述一个与材料相互作用的粒子
G4Track 描述一个在时空中传播的粒子

G4ParticleDefinition 集中了反应一个粒子属性的信息，例如，名字，质量，自旋，寿命，和衰变模式。*G4DynamicParticle* 集中了描述粒子的动力学信息，例如能量，动量，极化，和本征时间，还有``particle definition"信息。*G4Track* 包含了在一个探测器模拟中需要的所有信息，例如，时间，位置，和 step，还有``dynamic particle"信息。

G4Track 有在 G4 中用于 tracking 的所必要有信息。包括位置，时间，和 step，还有动力学信息。*G4Track* 的细节在 5.1 节中描述过。

除了以上三个类之外，还有 *G4ParticleWithCuts* 类也扮演了一个重要角色。它提供了将截断值转换为不同材料的能量阈的功能。

5.3.2 一个粒子的定义

有大量的基本粒子和原子核。Geant4 提供了 *G4ParticleDefinition* 类来表示粒子。各种粒子，例如电子，质子，和 gamma，它们都从 *G4ParticleDefinition* 派生出个自的类。

用户不需要在 Geant4 构造一个类，用于所有类型的粒子。 在 G4 中，有超过 100 种已经定义的类。哪种粒子因该被包括，如何实现它们，是根据以下的准则确定的。(当然，用户可以定义他想定义的任何类。请参看 **User's Guide: For ToolKit Developers**)

5.3.2.1 在 Geant4 的粒子列表（Particle list）

这个列表包括了所有在 G4 中的粒子，用户可以查看这些粒子的属性，例如

- PDG 代码
- 质量 和 宽度
- 自旋，同位旋和宇称
- 寿命和衰变模式
- 组成夸克

这里是在 G4 中的粒子的一个列表。这个列表是用 G4 自动产生的，所以列出的值跟用户在 G4 应用程序中使用的是相同的。(只要用户没有改变源码)

种类

- gluon / quarks / di-quarks
- leptons
- mesons
- baryons
- ions
- others

5.3.2.2 粒子分类

a. 在 G4 种应被跟踪的基本粒子
   所有可以飞行一个有限长度，能与探测器中的材料发生反应的粒子都属这一类。另外，一些非常短寿命的粒子也包括在里面。
   1. 稳定粒子
      稳定就意味着粒子不会衰变，或者在探测器中衰变的几率很小，例如，gamma，电子，质子，和中子。
   2. 长寿命粒子 ($>10^{-14}$sec)
      可以传播一个有限长度的粒子，例如， muon，带电 pions。

3. 需要在 G4 中衰变的短寿命粒子
   例如，π^0，η
4. K^0 系统
   K^0 立即"衰变"为 K^0_S 或 K^0_L，然后 K^0_S/K^0_L 在根据自身的寿命和衰变模式进行衰变。
5. 光学光子
   从模拟的角度来看，Gammas 和光学光子是有区别的，虽然它们是同种粒子(具有不同能量的光子)。例如，光学光子被用于切伦科夫光和闪烁光。
6. geantinos/带电的 geantinos
   Geantinos 和带电的 geantinos 是用于模拟的虚粒子，它们不与材料发生作用，只进行输运。

b. 原子核
   任何种类的原子核都可以在 G4 中使用，例如 alpha(He-4)，uranium-238 和 carbon-14 的激发态。从模拟角度来看，在 G4 中的原子和被分成了两组。
   1. 轻原子核
      在模拟中经常使用的轻原子核，例如，alpha，deuteron， He3，triton。
   2. 重原子核
      除在前面那组中定义了的原子核。

   注意，G4ParticleDefinition 描述原子核状态， G4DynamicParticle 与一些原子核一起描述原子状态。带 2 个单位正电荷的 alpha 粒子跟不带电的氦原子使用同一个 G4Alpha 的“粒子定义(particle definition)”，但是，应当给它们指定不同的 G4DynamicParticle 对像。(在下面可以找到这方面的细节)。

c. 短寿命粒子
   非常短寿命的粒子立即进行衰变，它们从来都不会在探测器几何体内被跟踪。这些粒子通常只用在物理过程的内部，来实现一些反应模型。*G4VShortLivedParticle* 是这些粒子的基类。与这类粒子相关的所有类可以在 `particles` 目录下的 `shortlived` 子目录找到。
   1. 夸克/di-夸克
      例如，所有的 6 中夸克。
   2. 胶子
   3. 断寿命的重子激发态
      例如，自旋为 3/2 的重子和反重子
   4. 断寿命的介子激发态
      例如，自旋唯一的矢量玻色子

5.3.2.3 粒子的实现

Singleton: 种类 a, b-1

这些粒子常用于 G4 的粒子跟踪。对于这些种类的粒子，每种粒子都定义了一个独立的类。每个类的对像是唯一的，并且是一个静态对像 (所谓的 singleton)。用户可以使用这些类的静态方法获取指向这些对像的指针。

在飞行过程种产生: 种类 b-2

在探测器中传播的离子，应该被跟踪。但是，可能由于强相互作用过程的离子数目是非常巨大的，以致需要使用动态的方法来实现。每个离子对应于一个 *G4Ions* 对像，它将在飞行过程中，在 `G4ParticleTable::GetIon()`方法中被建立。

由物理过程动态产生： 种类 c
在这类粒子中的粒子类型缺省情况下是不会生成的，只能由物理过程或者用户直接请求来生成。每个短寿命粒子对应于一个从 *G4VshortLivedParticle* 派生的类，它将在``初始化(initialization phase)"。

5.3.2.4 *G4ParticleDefinition*

G4ParticleDefinition 类有一些``只读"属性用来刻画独立的粒子，例如，名字，质量，电荷，自旋，等。这些属性是在每个粒子初始化期间设置的。获取这些属性的方法列于 Table 5.3.1。

`G4String GetParticleName()`	particle name
`G4double GetPDGMass()`	mass
`G4double GetPDGWidth()`	decay width
`G4double GetPDGCharge()`	electric charge
`G4double GetPDGSpin()`	spin
`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 5.3.1
获取粒子属性的方法。

Table 5.3.2 显式了用于获取有关衰变模式和粒子寿命信息的 *G4ParticleDefinition* 的方法。

`G4bool GetPDGStable()`	稳定标志
`G4double GetPDGLifeTime()`	寿命
`G4DecayTable* GetDecayTable()`	衰变表

Table 5.3.2
获取粒子衰变模式和寿命的方法。

用户可以修改这些属性，但是注意，在上面那个表中列出的那些属性只能通过重建库的方式来修改。

G4ParticleDefinition 提供了用于设置和/或者获取截断值的方法，在 Table 5.3.3。

然而，这些方法只能提供设置和获取截断值的功能。如下所述，根据这些截断值来计算能量截断范围是在 *G4ParticleWithCuts* 类中实现的。

另外，每个粒子都有自己的 *G4ProcessManger* 对像，用于管理一系列可用于这个粒子的过程。

5.3.3 Dynamic particle

G4DynamicParticle 类有一些粒子的运动学信息，这些信息用于描述物理过程的动力学。在 *G4DynamicParticle* 中的这些属性在 Table 5.3.4 中。

`G4double theDynamicalMass`	Dynamic 质量
`G4ThreeVector theMomentumDirection`	标准化的动量矢量
`G4ParticleDefinition* theParticleDefinition`	粒子定义

`G4ThreeVector thePolarization`	极化矢量
`G4double theKineticEnergy`	动能
`G4double theProperTime`	本征时间
`G4ElectronOccupancy* theElectronOccupancy`	粒子的电子轨道

Table 5.3.4
用于设置/获取截断值的方法。

在此，dynamic 质量是指 dynamic particle 的质量。对于大多数情况，它与在 *G4ParticleDefinition* 类中定义的质量是相同的。(即，由方法 `GetPDGMass()`给定的质量值)。然而，有两个例外。

- 共振粒子
- 离子

在质心系中，共振粒子有大的质量宽度和大的总衰变产物能量，而且事件之间可能是有差别的。（共振态的衰变末态各粒子的质量和是可以大于共振态粒子的。）

就粒子而言，*G4ParticleDefintion* 定义一个原子核， *G4DynamicParticle* 定义一个原子。*G4ElectronOccupancy* 描述轨道电子状态。所以，由于电子质量(和它们的结合能)的原因，daynamic 质量可能与 PDG 质量不同。

在事件发生器中给出重味粒子的衰变产物。在这种情况下，*G4VPrimaryGenerator* 在 `*thePreAssignedDecayProducts` 中设置这个信息。另外，通过使用 `PreAssignedDecayProperTime`，可以将粒子的衰变时间设置为任意时间。

About the authors

Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Tracking and Physics

5.4 产物阈值与截断值

5.4.1 大致情况

用户必须面对两个矛盾的要求。根据自身的实际能力产生次级粒子是每个独立 **process** 的责任。另一方面，只有 G4 内核（即，tracking）可以保证模拟的完全一致性。

在 G4 中遵循的一般原则是：

1. 产生次级粒子的每个 **process** 有它本身内在的一些限制。
2. 所有产生（和接受）的粒子都将被跟踪，一直到 **zero range**。（zero range 是指粒子静止，此时粒子的射程为 0）
3. 每个粒子在射程范围内，都有一个建议的截断值(对于不同材料，它将转换为不同的能量值)，这个值是通过方法 `SetCut()`定义的。(参看 2.4.2 节).

第 1 和 2 点意味着跟粒子相关的截断值是次级粒子(推荐)的产物阈值

5.4.2 设置产物阈值(`SetCut` 方法)

就像前面提到的，每一种粒子都有建议的产物阈值。有一些物理过程不使用这个阈值 (例如，衰变)，然而其它的过程将使用这个值作为它们内在限制的缺省值 (例如，电离和韧致辐射)。

有关如何设置这个产物阈值，参看 2.4.2 节 。

5.4.3 使用截断

每个过程的 `DoIt` 方法可以产生次级粒子。可能发生下面两种情况：

- 一个过程设置它自身的内在限制大于或者等于推荐的产物阈值。OK. Nothing has to be done (nothing can be done !).
- 一个过程设置它自身的内在限制小于推荐的产物阈值(for instance 0).

  次级粒子列表通过一个 *ParticleChange* 对像被发送给 *SteppingManager*。

在这些粒子被拷贝到临时堆栈用于以后的 tracking 前，将根据后面马上要提到的安全机制，判断是否将那些低于产物阈值的粒子删除。

- *ParticleDefinition* (or *ParticleWithCuts*) 有一个布尔数据成员：`ApplyCut`.
- `ApplyCut` 关闭的情况：不考虑粒子的初始能量，将所有的次级粒子压栈 (以后进行粒子跟踪)。G4 内核注重物理上的完美，忽略了整体的一致性和效率。只要物理过程知道如何正确处理产生的粒子，能量守恒将被遵守。
- `ApplyCut` 打开的情况：*TrackingManager* 将按产物阈值和 safety 来检查每个次级粒子的射程 range。如果 `range > min(cut,safety)`，那么这个粒子将被压栈。

- 如果不是，检查这个过程是否设置了``good for tracking"标志，是则压栈 (参看 5.4.4 节关于 `GoodForTracking` 标志的解释)。
- 如果不是，在 `localEnergyDeposit` 中回收它的动能，并设置 `tkin=0`。
- 然后在 *ProcessManager* 中检查是否 *ProcessAtRest* 的向量是非空的。如果是，将这个粒子压栈，用于以后进行``Action At Rest"处理。如果不是，丢弃这个次级粒子。

同过这个复杂机制，用户获得了一个期望的全局截断值，除了能量守恒，还遵守边界约束 (safety)和物理过程的期望(通过``good for tracking")。

5.4.4 为什么要产生低于阈值的次级粒子？

一个物理过程有可能有一些很好的理由去产生低于推荐阈值的次级粒子：

- 将次级粒子的射程与几何量（如 safety）相比，可以使用户了解产生的粒子（甚至是低于阈值的）到达探测器灵敏部分的可能性；
- 另一个例子是 gamma 转换：即使在零能量的时候，正电子也在产生，然后湮灭。

使用前面一节(恰好当 `ApplyCut` 打开的时候)提到的过滤器对 `GoodForTracking` 标志进行过滤，将通过的次级粒子发送给``Stepping Manager"。

5.4.5 射程截断 还是 能量截断?

射程截断使能量在正确的空间位置，在一个近似给定的距离内释放。相反，能量截断意味着跟材料相关的能量沉积的精度。

5.4.6 总结

总之，G4 没有 tracking 截断；只有在产物的射程阈值。所有产生和接受的粒子都被跟踪到 zero range。

必须清楚，G4 提供的完整的一致性不能超越物理过程产生低于推荐阈值粒子的实际能力。

换句话说，一个物理过程可以产生低于推荐阈值的次级粒子，并且，通过查询几何，或者通过了解什么时候发生质能转换，来确认什么时候不得不产生低于阈值得粒子。

5.4.7 特殊的 tracking 截断

基于优化的原因，用户可能需要在给定的 volumes 内截断特定的粒子类型。这由用户选择，可以作用于 tracking 期间的粒子。

用户必须只将这些特殊的截断应用到期望 volumes 内的期望粒子，不应用到其它粒子，从而引入过多的计算开销。

方法如下：

- 特殊的用户截断被注册在 *UserLimits* 类中 (或者它的子类)，这个类是与逻辑体类相关的。

  当前缺省清单：

  - 最大允许的步长
  - 最大总径迹长度
  - 最大总飞行时间
  - 最小动能
  - 最小保持射程

  用户可以仅为期望的逻辑体实例化一个 *UserLimits* 对像，并进行关联。

  第一项 (最大步长)是由 G4 内核自动考虑的，而其它项必须由用户处理，如下面所解释的。

  例子 (参看 novice/N02)：在 Tracker 区域，为了强迫步长不超过 Tracker 的 1/10，可以在 `DetectorConstruction::Construct()`中加入以下代码：

```
G4double maxStep = 0.1*TrackerLength;
logicTracker->SetUserLimits(new G4UserLimits(maxStep));
```

  G4UserLimits 类在 `source/global/management`

- 关于其它的截断，用户必须定义专用的 process(es)。然后只在期望的粒子的 process manager 中为这些粒子注册这个 process (或者它的子类)。用户可以在他这个 process 的 `DoIt` 中应用他的截断，以后，他就可以通过 *G4Track* 存取逻辑体和 *UserLimits*。

  在软件库里面提供了一个这样的 process 的例子(叫做 *UserSpecialCuts*)，但是没有被嵌入到任何粒子的任何 process manager 中。

  例子：中子。在中子飞行一个给定的时间后(或者在磁场中的一个带电粒子历经了一个给定的总径迹长度之后... 等 ...)，用户可能需要放弃对这些中子的 tracking。

  粒子(参看 novice/N02)：在 Tracker 区域，为了使中子的总飞行时间不超过 10 毫秒，需要在 `DetectorConstruction::Construct()`中放置下面的代码：

```
G4double maxTime = 10*ms;
```

```
        logicTracker->SetUserLimits(new
G4UserLimits(DBL_MAX,DBL_MAX,maxTime));
```

并且在 `N02PhysicsList` 中放置以下代码：

```
        G4ProcessManager* pmanager = G4Neutron::Neutron-
>GetProcessManager();
        pmanager->AddProcess(new G4UserSpecialCuts(),-1,-1,1);
```

(缺省的 *G4UserSpecialCuts* 类在 `source/processes/transportation` 中。)

About the authors

Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Tracking and Physics

5.5 分区域截断

5.5.1 常用概念

从 G4 5.1 版开始，区域(region)的概念已经被定义，用在几何描述中。关于区域和如何使用区域的细节在 4.1.3.1 小节中。例如，假定一个用户定义了三个区域，分别对应一个探测器的 tracking volume，量热器，和体结构。出于性能的原因，用户也许对发生在不敏感的体结构中的电磁簇射的详细过程不感兴趣，但希望在 tracking 区域保持最高可能的精度。在这样的使用情况下，G4 允许用户为不同的几何区域设置不同的产物阈值 ("截断")。这个能力被称作"分区域截断"，它是 G4 5.1 版提供的一个新特性。这些产物阈值的常用概念在前一节中已经讲过。

请注意，这个新特性，只计划用于这些用户

1. 正在模拟最复杂的几何体，例如一个 LHC 探测器，和
2. 对在物质中的电磁簇射过程有丰富的经验。

我们强烈推荐对使用这个新特性后的结果和使用单一产物阈值的结果进行比较。为单独的区域设置完全不同的截断值，可能破坏模拟的一致和综合的精度。因此，因该基于与使用单一阈值的结果进行比较，仔细地优化截断值。

5.5.2 缺省区域

world 缺省是作为一个区域处理地。一个 *G4Region* 对像自动地被分配给 world，并且被当作"缺省区域"。这个区域的产物阈值是缺省的，它在 *UserPhysicsList* 中被定义。除非用户为其它区域定义不同的截断值，否则，在缺省区域中的截断值将被用于整个几何。

请注意，缺省区域和它的缺省产物截断是由 *G4RunManager* 自动建立并设置的。**不允许**用户给 world 设置一个区域，**也不允许**给缺省区域指定其它的产物截断。

5.5.3 为一个区域指定产物截断值

在用户 physics list 的方法 `SetCuts()` 中，用户必须首先定义缺省截断值。然后，用户必须为指定区域，使用期望的截断值，建立并初始化一个 *G4ProductionCuts* 对像。接着，这个对像必须被分配给一个区域对像，区域对像可以从 *G4RegionStore* 通过名字进行存取。下面是一个 `SetCuts()`代码的例子。

```
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);
}
```

Source listing 5.5.1
为一个区域设置产物截断

About the authors

Overview Contents Previous Next

Geant4 User's Documents
Geant4 User's Guide
For Application Developers

6. 用户行为(Actions)

6.1 必要的用户行为（Actions）和初始化

Geant4 有三个虚类，用户为了实现模拟必须重载它们的方法。它们要求用户定义探测器，指定将要使用的物理过程，并且描述如何初始化将要产生的粒子。

G4VUserDetectorConstruction

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

  public:
    virtual G4VPhysicalVolume* Construct() = 0;
};
```

Source listing 6.1.1
G4VUserDetectorConstruction

G4VUserPhysicsList

这是一个用来构造粒子和物理过程的抽像类。用户必须从它派生出一个具体类，并且实现三个虚方法：

- `ConstructParticle()` 用于实例化每个被请求的粒子类型，
- `ConstructPhysics()` 用于实例化期望的物理过程，并向适当的粒子的 process manager 注册每个过程。
- `SetCuts(G4double aValue)` 为在粒子表(particle talbe)中的所有粒子设置一个射程截断值，它将引起 physics table 的重建。

当被调用的时候，*G4VUserPhysicsList* 的方法 `Construct()`首先调用 `ConstructParticle()`，然后调用 `ConstructProcess()`。为了保证粒子输运的进行，方法 `ConstructProcess()`必须始终调用方法 `AddTransportation()`， `AddTransportation()` 绝对不能被重载。

为实现上述的虚方法，*G4VUserPhysicsList* 提供了几个使用方法。在 source listing 6.1.2 的类声明中，列出了这些方法和注释。

```
class G4VUserPhysicsList
{
  public:
    G4VUserPhysicsList();
    virtual ~G4VUserPhysicsList();

  public:  // with description
    // By calling the "Construct" method,
    // particles and processes are created
    void Construct();


  protected: // with description
   // These two methods of  ConstructParticle() and ConstructProcess()
   // will be invoked in the Construct() method.

   // each particle type will be instantiated
   virtual void ConstructParticle() = 0;

   // each physics process will be instantiated and
   // registered to the process manager of each particle type
   virtual void ConstructProcess() = 0;

  protected: // with description
   //  User must invoke this method in his ConstructProcess()
   //  implementation in order to insures particle transportation.
   //  !! Caution: this class must not be overriden !!
   void AddTransportation();

  ////////////////////////////////////////////////////////////////
```

```
public: // with description
 //  "SetCuts" method sets a cut value for all particle types
 //   in the particle table
 virtual void SetCuts() = 0;

public:  // with description
 //  set/get the default cut value
 //  Calling SetDefaultCutValue causes re-calcuration of cut values
 //  and physics tables just before the next event loop
 void    SetDefaultCutValue(G4double newCutValue);
 G4double GetDefaultCutValue() const;

////////////////////////////////////////////////////////////////
public: // with description
  // Invoke BuildPhysicsTable for all processes for all particles
  // In case of "Retrieve" flag is ON, PhysicsTable will be
  // retrieved from files
  void BuildPhysicsTable();

 // do BuildPhysicsTable for specified particle type
  void BuildPhysicsTable(G4ParticleDefinition* );

   // Store PhysicsTable together with both material and cut value
  // information in files under the specified directory.
  //  (return true if files are sucessfully created)
  G4bool  StorePhysicsTable(const G4String& directory = ".");

  // Return true if "Retrieve" flag is ON.
  // (i.e. PhysicsTable will be retrieved from files)
  G4bool  IsPhysicsTableRetrieved() const;
  G4bool  IsStoredInAscii() const;

  // Get directory path for physics table files.
  const G4String& GetPhysicsTableDirectory() const;

  // Set "Retrieve" flag
  // Directory path can be set together.
  // Null string (default) means directory is not changed
  // from the current value
  void   SetPhysicsTableRetrieved(const G4String& directory = "");
  void   SetStoredInAscii();

  // Reset "Retrieve" flag
  void   ResetPhysicsTableRetrieved();
  void   ResetStoredInAscii();
```

```
 ////////////////////////////////////////////////////////////////////////
  public: // with description
    // Print out the List of registered particles types
    void DumpList() const;

  public: // with description
    // Request to print out information of cut values
    // Printing will be performed when all tables are made
    void DumpCutValuesTable(G4int nParticles=3);

    // The following method actually trigger the print-out requested
    // by the above method. This method must be invoked by RunManager
    // at the proper moment.
    void DumpCutValuesTableIfRequested();

  public: // with description
    void  SetVerboseLevel(G4int value);
    G4int GetVerboseLevel() const;
    // set/get controle flag for output message
    //  0: Silent
    //  1: Warning message
    //  2: More

  ////////////////////////////////////////////////////////////////////////
  public: // with description
   //  "SetCutsWithDefault" method sets the default cut value
   //   for all particles for the default region.
   void SetCutsWithDefault();

   // Following are utility methods for SetCuts

   // SetCutValue sets a cut value for a particle type for the default region
   void SetCutValue(G4double aCut, const G4String& pname);

   // SetCutValue sets a cut value for a particle type for a region
   void SetCutValue(G4double aCut, const G4String& pname, const G4String& rname);

   // Invoke SetCuts for specified particle for a region
   // If the pointer to the region is NULL, the default region is used
   // In case of "Retrieve" flag is ON,
   // Cut values will be retrieved from files
   void SetParticleCuts(G4double cut,G4ParticleDefinition* particle,G4Region*
region=0);
```

```
   // Invoke SetCuts for all particles in a region
   void SetCutsForRegion(G4double aCut, const G4String& rname);

   // Following are utility methods are obsolete
   void ResetCuts();

////////////////////////////////////////////////////////////////
  public:
   // Get/SetApplyCuts gets/sets the flag for ApplyCuts
   void SetApplyCuts(G4bool value, const G4String& name);
   G4bool GetApplyCuts(const G4String& name) const;

////////////////////////////////////////////////////////////////////////////
  protected:
    // do BuildPhysicsTable for make the integral schema
    void BuildIntegralPhysicsTable(G4VProcess* ,G4ParticleDefinition*  );


  protected:
    // Retrieve PhysicsTable from files for proccess belongng the particle.
    // Normal BuildPhysics procedure of processes will be invoked,
    // if it fails (in case of Process's RetrievePhysicsTable returns false)
    virtual void  RetrievePhysicsTable(G4ParticleDefinition* ,
                                const G4String& directory,
                                G4bool        ascii = false);

   ////////////////////////////////////////////////////////////
  protected:
    // adds new ProcessManager to all particles in the Particle Table
    //   this routine is used in Construct()
    void InitializeProcessManager();

  public: // with description
    // remove and delete ProcessManagers for all particles in tha Particle Table
    //    this routine is invoked from RunManager
    void RemoveProcessManager();

  public: // with description
    // add process manager for particles created on-the-fly
    void AddProcessManager(G4ParticleDefinition* newParticle,
                        G4ProcessManager*    newManager = 0 );

};
```

Source listing 6.1.2
G4VUserPhysicsList

G4VUserPrimaryGeneratorAction

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

  public:
    virtual void GeneratePrimaries(G4Event* anEvent) = 0;
};
```

Source listing 6.1.3
G4VUserPrimaryGeneratorAction

About the authors
Overview Contents Previous Next
Geant4 User's Guide
For Application Developers
User Actions

6.2 可选的用户行为（Actions）

有 5 个虚类，用于增强在不同阶段对模拟的控制，用户必须重载这些类的方法。每个 action 类的每个方法都有一个空的缺省实现，允许用户继承并实现期望的类和方法。用户 action 类的对像，必须向 G4RunManager 注册。

G4UserRunAction

这个类有三个虚方法，它们由 G4RunManager 为每个 run 调用：

GenerateRun()

这个方法在 BeamOn 开始的时候被调用。用户可以继承类 G4Run 并建立他自己的具体类，用于存储跟 run 有关的一些信息，这个 GenerateRun() 就是实例化这个具体类的地方。这个方法也是为一个特殊的 run 设置那些影响 physics table (such as production thresholds)的变量的地方，因为 GenerateRun() 是在 physics table 计算之前被调用的。

BeginOfRunAction()

这个方法在进入事件循环之前被调用。这个方法的典型使用是为一个特殊的 run 初始化和/或预订 histograms。这个方法在 physics tables 计算之后被调用。

EndOfRunAction()

这个方法在 run 处理的结尾被调用。它一般用于对运行完成的 run 进行一些简单分析。

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

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

Source listing 6.2.1
G4UserRunAction

G4UserEventAction

这个类有两个虚方法，G4EventManager 为每个事件调用它们：

beginOfEventAction()

这个方法在将初级粒子转换为 G4Track 对像之前被调用。这个方法的典型使用是为一个特殊的事件初始化和/或预订 histograms。

endOfEventAction()

这个方法在事件处理的结尾被调用。它一般用于对处理完的事件进行一些简单的分析。

```
class G4UserEventAction
{
  public:
     G4UserEventAction() {;}
     virtual ~G4UserEventAction() {;}
```

```
        virtual void BeginOfEventAction(const G4Event*);
        virtual void EndOfEventAction(const G4Event*);
    protected:
        G4EventManager* fpEventManager;
};
```

Source listing 6.2.2
G4UserEventAction

G4UserStackingAction

这个类有三个虚方法，ClassifyNewTrack， NewStage 和 PrepareNewEvent， 为了控制各种 track 的栈机制，用户可以对它们进行重载。ExampleN04 可能是理解怎样这个类的一个很好的粒子。
无论什么时候，只要一个新的 G4Track 对像被 G4EventManager“压入”堆栈，G4StackManager 都要调用 ClassifyNewTrack() 。ClassifyNewTrack() 返回一个枚举对像 G4ClassificationOfNewTrack，它的值表明 track 将被发送到哪个栈。这个值应该由用户确定。G4ClassificationOfNewTrack 有四个可能的值：

fUrgent – track 被放置在 *urgent* 栈
fWaiting - track 被放置在 *waiting* 栈，并且直到 *urgent* 栈为空的时候才会被模拟
fPostpone - track 被推迟（postpone）到下个事件
fKill – track 被立即删除，并为保存到任何栈。

这些基于 track 起点生成的值，可以用下面的方法获取：

```
G4int parent_ID = aTrack->get_parentID();
```

这里
parent_ID = 0 表明是一个初级粒子
parent_ID > 0 表明是一个次级粒子 indicates a secondary particle
parent_ID < 0 表明是一个来自前个事件的被推迟的粒子。

当 *urgent* 栈为空，且 *waiting* 栈包含至少一个对像的时候，将调用 NewStage()。这里，用户可以通过调用 stackManager->ReClassify()方法，将在 *waiting* 栈中的所有 traks kill 掉，或者重新分配到不同的栈，stackManager->ReClassify()方法将调用 ClassifyNewTrack() 方法。如果没有使用任何用户 action，在栈中的所有 tracks 将被转移到栈。用户也可以决定通过调用 stackManager->clear()来中止当前事件，尽管在 *waiting* 栈中可能存在一些 tracks。只有从 G4UserStackingAction 类中调用这些方法才是可用并且安全的。中止事件的全局方法是

```
G4UImanager * UImanager = G4UImanager::GetUIpointer();
UImanager->ApplyCommand("/event/abort");
```

PrepareNewEvent() 在每个事件开始的时候被调用。因为，在这个时刻，还没有初级粒子被转换为 tracks，所以， *urgent* 栈和 *waiting* 栈是空的。但是，可能在 *postponed-to-next-event*

栈中存在一些 tracks；G4 将为每个 track 调用 `ClassifyNewTrack()`方法，这些 track 将被分配到合适的栈。

```
#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();
//
//------------------------------------------------------------------

};
```

Source listing 6.2.3
`G4UserStackingAction`

G4UserTrackingAction

```
//---------------------------------------------------------------
//
// G4UserTrackingAction.hh
//
```

```
// Description:
//   This class represents actions taken place by the user at each
//   end of stepping.
//
//--------------------------------------------------------------

//////////////////////////
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;

};
```

Source listing 6.2.4
G4UserTrackingAction

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;

};
```

Source listing 6.2.5
G4UserSteppingAction

About the authors

Overview Contents Previous Next

Geant4 User's Documents
Geant4 User's Guide
For Application Developers

7. 运用程序的通迅和控制

7.1 内建命令

Geant4 有各种内建的用户接口命令，每一个大概对应一个模块。用户可以通过以下方式使用这些命令

- 通过一个图形用户接口(G)UI 交互式的使用，

- 在宏文件中通过/control/execute <command>方式使用，
- 在 C++ 代码内，通过 G4UImanager 的方法 ApplyCommand 使用。

注意
每个命令的有效性，参数的范围，关于个别命令的候选参数值，它们都会因用户应用程序的不同实现而不同，甚至有可能在用户任务执行期间发生动态变化。

内建命令清单

7.2 用户接口——定义新的命令

7.2.1 *G4UImessenger*

G4UImessenger 是一个消息发送器的基类，它将命令传递给目标类对像。用户的具体消息发送器应有下面的功能。

- 在用户消息发送器中构造用户命令。
- 在用户消息发送器中析构用户命令。

这些要求意味着用户消息发送器因该将所有指向用户命令对像的指针作为自己的数据成员保存。

对于大多数常用的命令，用户可以使用 *G4UIcommand* 的派生类。这些派生类对不同类型的命令有个自的转换方法，它们使用户更加简单容易的实现用户消息发送器的方法 `SetNewValue()` 和 `GetCurrentValue()` 。

对应使用各种参数的复杂命令，用户可以使用 *G4UIcommand* 基类，自己构造 *G4UIparameter* 对像。用户不需要删除 *G4UIparameter* 对像。

在用户消息发送器的方法 `SetNewValue()` 和 `GetCurrentValue()` 中，用户可以比较 *G4UIcommand* 指针，这个指针是由用户命令指针在这些方法的参数中给定的。用户消息发送器保存了那些指向命令的指针。因此，用户必须要通过命令名来进行比较。请记住，在用户使用 *G4UIcommand* 派生类的场合，用户应该用这些派生类的类型存储这些指针，以便用户可以根据它们的类型使用派生类中定义的方法，而不需要进行强制转换。

G4UImanager/G4UIcommand/G4UIparameter 拥有非常强大的类型和范围检查例程。强烈推荐用户设置参数的取值范围。对于数值(`int` 或 `double`)的情况，可以通过一个 *G4String* 使

用 C++ 的表示方法给定范围，例如，"X > 0 && X < 10"。对于字符串类型的参数，用户可以设置一个候选参数列表。请参考下面描述的细节。

GetCurrentValue() 在用户请求相应命令之后，在 SetNewValue()启动之前被调用。这个 GetCurrentValue()方法只有在下列情形下被调用

- 至少该命令的一个参数有取值范围
- 至少该命令的一个参数有一个候选参数列表
- 至少一个参数的值被忽略，并且这个参数是作为可忽略和 currentValueAsDefault 定义的。

对于前两种情况，如果需要的话，用户可以重置参数取值范围和候选参数列表，但是，只有在参数取值范围和候选参数列表动态变化的情况下，才需要``重置"这些参数。

命令可以是``状态敏感的"，即，该命令只在某种 *G4ApplicationState*(s)状态下被接受。例如，在 G4 正在处理其它的事件时(``G4State_EventProc" state)，不应接受/run/beamOn 命令。用户可以通过方法 AvailableForStates()设置命令的可用状态。

7.2.2 *G4UIcommand* 和它的派生类

所有派生类可用的方法

这些方法被定义在 *G4UIcommand* 基类中，它们应在派生类中使用。

- void SetGuidance(char*)

  定义一个提示行。用户可以在需要给出充分提示的地方任意调用这个方法。请注意，第一行将用作命令提示的标题。

- void availableForStates(G4ApplicationState s1,...)

  如果用户命令只是对于 G4 内核的某种状态才是可用的，那么需要通过这个方法指定这些状态。当前可用的状态是，G4State_PreInit, G4State_Init, G4State_Idle, G4State_GeomClosed, 和 G4State_EventProc。有关这些状态的意义，请参考 3.4.2 节。请注意 Pause 状态已经从 *G4ApplicationState* 中被删除了。

- void SetRange(char* range)

  定义参数的一个取值范围。使用 C++的表示方法，例如"x > 0 && x < 10"，其中的变量名通过方法 SetParameterName()定义。对于 *G4ThreeVector* 的情况，用户可以设置参数之间的关系，例如，"x > y"。

G4UIdirectory

这是一个用于定义路径的 *G4UIcommand* 派生类。

- `G4UIdirectory(char* directoryPath)`

  构造函数。参数是路径（绝对路径），必须以`/'开始和结尾。

G4UIcmdWithoutParameter

这是一个用于不带参数的命令的派生类 *G4UIcommand*。

- `G4UIcmdWithoutParameter(char* commandPath,G4UImessenger* theMessenger)`

  构造函数。参数是（绝对路径）命令名和指向用户消息发送器的指针。

G4UIcmdWithABool

这是带一个布尔型参数的 *G4UIcommand* 派生类。

- `G4UIcmdWithABool(char* commandpath,G4UImanager* theMessenger)`

  构造函数。参数是（绝对路径）命令名和指向用户消息发送器的指针。

- `void SetParameterName(char* paramName,G4bool omittable)`

  定义布尔参数的名字和可忽略标志。如果可忽略标志为真，用户应该使用下一个方法定义缺省值。

- `void SetDefaultValue(G4bool defVal)`

  定义布尔参数缺省值。

- `G4bool GetNewBoolValue(G4String paramString)`

  将用户消息发送器的方法 `SetNewValue()`提供的参数 *G4String* 转换为布尔型。

- `G4String convertToString(G4bool currVal)`

  将布尔型值转换为 *G4String* ，这个 *G4String* 应由用户消息发送器的 `GetCurrentValue()`方法返回。

G4UIcmdWithAnInteger

这是带一个整型参数的 *G4UIcommand* 派生类。

- `G4UIcmdWithAnInteger(char* commandpath,G4UImanager* theMessenger)`

  构造函数。参数是（绝对路径）命令名和指向用户消息发送器的指针。

- `void SetParameterName(char* paramName,G4bool omittable)`

  定义整型参数的名字和设置可忽略标志。如果可忽略标志为真，那么用户需要使用下面的方法定义缺省值。

- `void SetDefaultValue(G4int defVal)`

  定义整型参数缺省值。

- `G4int GetNewIntValue(G4String paramString)`

  通过用户消息发送器的方法`SetNewValue()`,将 *G4String* 参数值转换为整型值。

- `G4String convertToString(G4int currVal)`

  将当前整型值转换为 *G4String* ， 这个 *G4String* 应由用户消息发送器的`GetCurrentValue()`方法返回。

G4UIcmdWithADouble

这是带一个双精度型参数的 *G4UIcommand* 派生类。

- `G4UIcmdWithADouble(char* commandpath,G4UImanager* theMessenger)`

  构造函数。参数是（绝对路径）命令名和指向用户消息发送器的指针。

- `void SetParameterName(char* paramName,G4bool omittable)`

  定义双精度型参数的名字和设置可忽略标志。如果可忽略标志为真，那么用户需要使用下面的方法来定义缺省值。

- `void SetDefaultValue(G4double defVal)`

  定义双精度型参数的缺省值。

- `G4double GetNewDoubleValue(G4String paramString)`

  将用户消息发送器的`SetNewValue()`方法提供的 *G4String* 参数值转换为双精度型。

- `G4String convertToString(G4double currVal)`

  将当前双精度值转换为 *G4String* ， 这个 *G4String* 应由用户消息发送器的`GetCurrentValue()`方法返回。

G4UIcmdWithAString

这是带一个字符串型参数的 *G4UIcommand* 派生类。

- `G4UIcmdWithAString(char* commandpath,G4UImanager* theMessenger)`

  构造函数。参数是（绝对路径）命令名和指向用户消息发送器的指针。

- `void SetParameterName(char* paramName,G4bool omittable)`

定义字符串型参数的名字和设置可忽略标志。如果可忽略标志为真，那么用户需要使用下面的方法来定义缺省值。

- `void SetDefaultValue(char* defVal)`

  定义字符串型参数缺省值。

- `void SetCandidates(char* candidateList)`

  定义一个候选参数列表。在这个列表中的每个候选参数应该用一个空格分隔。在指定了这个候选参数列表后，如果用户提供的字符串在这个列表中不存在，那么将被拒绝接受。

G4UIcmdWith3Vector

这是带一个三维矢量参数的 *G4UIcommand* 派生类。

- `G4UIcmdWith3Vector(char* commandpath,G4UImanager* theMessenger)`

  构造函数。参数是（绝对路径）命令名和指向用户消息发送器的指针。

- `void SetParameterName`
  `(char* paramNamX,char* paramNamY,char* paramNamZ,G4bool omittable)`

  定义这个三维矢量每个部分的名字和设置可忽略标志。如果可忽略标志为真，用户可以使用下面的方法定义缺省值。

- `void SetDefaultValue(G4ThreeVector defVal)`

  定义这个三维矢量的缺省值。

- `G4ThreeVector GetNew3VectorValue(G4String paramString)`

  将用户消息发送器的方法 `SetNewValue()`提供的 *G4String* 参数值转换为一个 *G4ThreeVector*。

- `G4String convertToString(G4ThreeVector currVal)`

  将当前三维向量转换为 *G4String*，这个 *G4String* 因该由用户消息发送器的 `GetCurrentValue()`方法返回。

G4UIcmdWithADoubleAndUnit

这是带一个双精度型参数和它的单位的 *G4UIcommand* 派生类

- `G4UIcmdWithADoubleAndUnit(char* commandpath,G4UImanager*`
  `theMessenger)`

  构造函数。参数是（绝对路径）命令名和指向用户消息发送器的指针。

- `void SetParameterName(char* paramName,G4bool omittable)`

定义这个双精度参数的名字和设置可忽略标志。如果可忽略标志为真，用户可以使用下面的方法定义缺省值。

- `void SetDefaultValue(G4double defVal)`

  定义这个双精度参数的缺省值。

- `void SetUnitCategory(char* unitCategory)`

  定义可接受的单位类型。

- `void SetDefaultUnit(char* defUnit)`

  定义缺省单位。请使用这个方法或者 `SetUnitCategory()`方法。

- `G4double GetNewDoubleValue(G4String paramString)`

  将用户消息发送器的方法 `SetNewValue(`提供的 *G4String* 参数值转换为双精度型。请注意，返回值是这个双精度值与指定单位值的乘积。

- `G4double GetNewDoubleRawValue(G4String paramString)`

  将用户消息发送器的方法 `SetNewValue()`提供的 *G4String* 参数值转换为双精度型，但是不与指定单位值相乘。

- `G4double GetNewUnitValue(G4String paramString)`

  将由用户消息发送器的方法 `SetNewValue()`提供的 *G4String* 单位值转换为双精度型。

- `G4String convertToString(G4bool currVal,char* unitName)`

  将双精度值转换为 *G4String*，这个 *G4String* 应由用户消息发送器的 `GetCurrentValue()`方法返回。这个双精度值将与指定的单位值相除，并且转换为一个字符串。指定的单位将被添加到这个字符串中。

G4UIcmdWith3VectorAndUnit

这是带一个三维矢量参数和它的单位的 *G4UIcommand* 派生类。

- `G4UIcmdWith3VectorAndUnit(char* commandpath,G4UImanager* theMessenger)`

  构造函数。参数是（绝对路径）命令名和指向用户消息发送器的指针。

- `void SetParameterName (char* paramNamX,char* paramNamY,char* paramNamZ,G4bool omittable)`

定义这个三维矢量每个部分的名字和设置可忽略标志。如果可忽略标志为真，用户可以使用下面的方法定义缺省值。

- `void SetDefaultValue(G4ThreeVector defVal)`

  定义这个三维矢量的缺省值。

- `void SetUnitCategory(char* unitCategory)`

  定义可接受的单位类型。

- `void SetDefaultUnit(char* defUnit)`

  定义缺省单位。请使用这个方法或者 `SetUnitCategory()` 方法。

- `G4ThreeVector GetNew3VectorValue(G4String paramString)`

  将用户消息发送器的方法 `SetNewValue()`给定的 *G4String* 参数值转换为一个 *G4ThreeVector*。请注意，这个返回值是这个参数值与给定单位的乘积。

- `G4ThreeVector GetNew3VectorRawValue(G4String paramString)`

  将用户消息发送器的方法 `SetNewValue()`给定的参数值转换为一个三维矢量，但结果不与给定单位值相乘。

- `G4double GetNewUnitValue(G4String paramString)`

  将用户消息发送器的方法 `SetNewValue()`给定的 *G4String* 单位值转换为一个双精度值。

- `G4String convertToString(G4ThreeVector currVal,char* unitName)`

  将当前三维矢量转换为一个 *G4String*，这个 *G4String* 应由用户消息发送器的方法 `GetCurrentValue()`返回。这个三维矢量值将与给定的单位值相除，结果转换为一个字符串。给定单位将被添加到这个字符串中。

Additional comments on the `SetParameterName()` method

用户可以为前面提到的每一个 `SetParameterName()`方法增加一个 `G4bool` 类型的附加参数。这个附加参数名为 `currentAsDefaultFlag`，且这个参数的缺省值为 `false`。如果用户给这个附加的参数赋值为 `true`,将使用目标类的当前值对参数的缺省值进行重载。

7.2.3 一个消息发送器的例子

这是 *G4ParticleGunMessenger* 的例子，它继承了 *G4UIcommand*。

```
#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
```

Source listing 7.2.1

一个 G4ParticleGunMessenger.hh 例子。

```
#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;
}
```

Source listing 7.2.2
一个 G4ParticleGunMessenger.cc 的例子。

7.2.4 如何控制 *G4cout/G4cerr* 的输出

G4 使用 *G4cout* 和 *G4cerr* 替换 *cout* 和 *cerr*。来自的 *G4cout/G4cerr* 输出流将被 *G4UImanager* 处理，*G4UImanager* 允许应用程序员对输出流进行控制。因此，输出字符串可以在另一个窗口显示，也可以存储到文件。这可以用如下方式完成：

1. 从 *G4UIsession* 派生一个类，并实现两个方法：
2.
3. `G4int ReceiveG4cout(G4String coutString);`
4. `G4int ReceiveG4cerr(G4String cerrString);`

这些方法分别接收 *G4cout* 和 *G4cerr* 的字符串流。这个字符串可以按指定要求处理。下面的代码显示了如何制作输出流的一个日志文件。：

```
ostream logFile;
logFile.open("MyLogFile");
G4int MySession::ReceiveG4cout(G4String coutString)
{
  logFile << coutString << flush;
  return 0;
}
```

5. 使用 `G4UImanager::SetCoutDestination(session)`设置 *G4cout/G4cerr* 的目标（会话）。

比较典型地，从 *G4UIsession* 或它派生类（如，*G4UIGAG/G4UIteminal*）的构造函数中调用这个方法。这个方法设置 *G4cout/G4cerr* 的目标为一个会话。例如，当下面的代码出现在 *G4UIterminal* 的构造函数中时，方法 `SetCoutDestination(this)`告诉 *UImanager* ，*G4UIterminal* 的 this 实例接收由 *G4cout* 产生的输出流。

```
G4UIterminal::G4UIterminal()
```

```
{
  UI = G4UImanager::GetUIpointer();
  UI->SetCoutDestination(this);
  //  ...
}
```

类似的，`UI->SetCoutDestination(NULL)` 必须被添加到这个类的析构函数中。

6. 编写或者修改主程序。修改 `exampleN01` 以产生一个日志文件，就像上面第 1 步中将的一样派生一个类，并且在主程序中加入以下代码行：

```
#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;
}
```

注意：如果一个类的实例打算作为静态实例来使用，那么在这个类的构造函数中不应该使用 *G4cout/G4cerr* 。这个限制来自与 C++语言的规范。详情请看下面的文档。

M.A.Ellis, B.Stroustrup. ``Annotated C++ Reference Manual", Section 3.4
P.J.Plauger, ``The Draft Standard C++ Library"

About the authors

Overview Contents Previous Next

Geant4 User's Documents
Geant4 User's Guide
For Application Developers

8. 可视化

8.1 可视化介绍

在这一节中，将描述如何进行 Geant4 可视化，即，可视化探测器元件，粒子径迹，tracking steps，hits，文本（字符串），等。

G4 可视化有许多不同的要求。例如，

1. 对观察连续事件有非常快的相应
2. 用于文档的高质量输出
3. 用于演示的生动的特殊效果
4. 用于探测器元件的几何调试和物理调试的，灵活的 camera（视点）控制。
5. 用于属性编辑和相关数据反馈的交互式图形对像拾取
6. 物理体碰撞的增强显示
7. 通过 Internet 进行远程可视化
8. 与图形用户接口协同工作

Geant4 可视化可以响应所有的这些要求，但是仅仅使用内建的可视化程序对所有的这些要求进行响应是非常困难的。因此，G4 支持一个可以与许多种图形系统接口的抽像接口。这里的图形系统，是指独立于 G4 独立允许的一个运用程序，或者是与 G4 一起编译的一个图形库。Geant4 可视化也支持各种图形系统的具体接口。这些图形接口是相互补充的。一个图形系统的具体接口被称作"可视化引擎"。

可视化进程是由"Visualization Manager"控制的，这个"Visualization Manager"用一个用户类 *MyVisManager* 来进行描述，它是定义在可视化模块中的 *G4VisManager* 类的子类。Visualization Manager 接受用户的可视化请求，处理这些请求并将这些处理后的要求传递给抽像接口，即，当前选择的可视化引擎。

Geant4 应用程序开发人员应该知道下列用于可视化的事情。

1. 可以被可视化的对像
2. 如何使用可视化属性，例如，颜色
3. 如何选择编译和执行时的可视化引擎
4. 如何使用内建的可视化命令
5. 如何使用 Visualization Manager 的显示方法，等

这些问题将在后续的章节中讲述。每个引擎的特性和注意点简要的在 8.6 节"**可视化引擎**"中讲述，细节在那里提到的网页上可以找到。一些高级的和/或与可视化引擎相关的主题在 8.10 节 "**其它**"中描述。同时，请参看宏文件 `examples/novice/N03/visTutor/exN03VisX.mac`。

Next section
Back to contents
Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Visualization

8.2 什么可以被可视化？

在 Geant4 可视化中，用户可以可视化被模拟的数据，如：

- 探测器元件
  - 物理体的一个层次结构
  - 一个物理体，逻辑体，或者实体
- 粒子径迹和 tracking steps
- 粒子在探测器元件中的 hits

和其它用户定义的对像，如：

- 折线，它是一系列连续的线段
- 标记，标记任何一个 3D 位置
- 文本，即，用于描述，注释，或者作为标题的字符串
- Eye guides，如，坐标轴

用户可以使用可视化命令，或者通过在用户的 C++程序中调用可视化函数，对所有这些东西进行可视化。

Next section
Back to contents
Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Visualization

8.3 可视化属性

可视化属性，是一系列可以被可视化的对像相关的信息。这些信息只用于可视化，并且不包括几何信息，例如，形状，位置，和方向。一个可视化属性的典型粒子是"颜色"。例如，在一个 box 的可视化中，Visualization Manager 必须指定它的颜色。如果将被可视化的对像没有被指定一系列的可视化属性，那么将自动使用适当的缺省值。

一系列的可视化属性被保存在一个 *G4VisAttributes* 类的实例中，这个类是在 `graphics_reps` 模块中定义的。下面，将逐个解释 *G4VisAttributes* 的主要字段。

8.3.1 可见性

可见性是一个布尔型标志，用于控制传递给 Visualization Manager 用于可视化的对像的可见性。可见性是通过下面的存取函数进行设置的：

```
void G4VisAttributes::SetVisibility (G4bool visibility);
```

如果用户将这个参数设为 `false`，并且 culling 被激活（参看下面），那么将跳过这个对像的可视化。可见性的缺省值是 `true`。

注意，当前的 culling 策略影响一个对像的可见性，它可以通过可视化命令进行调节。

缺省情况，下面的公有静态常量数据成员被定义

```
    static const G4VisAttributes Invisible;
```

在这，可见性被设置为 `false`。它可以以如下方式被引用，`G4VisAttributes::Invisible`，例如：

```
    experimentalHall_logical -> SetVisAttributes (G4VisAttributes::Invisible);
```

8.3.2 颜色

G4VisAttributes 类将它的颜色项作为一个 *G4Colour*(一个等价的类名是 *G4Color*)对像保存。

G4Colour 类有四个字段，它们代表颜色的 RGBA (红，绿，蓝，和 alpha)成分。每种成分的取值为 0 到 1。如果构造函数使用了一个无用的值，小与 0 或者大于 1 的值，这个值将分别被自动设置为 0 或 1。Alpha 是不透明度，目前没有使用。用户可以使用它的缺省值 1，它意味着 *G4Colour* 实例的"不透明"。

为 *G4Colour* 的构造函数指定红，绿，蓝成分，初始化 *G4Colour* 对像，

```
    G4Colour::G4Colour ( G4double r = 1.0,
                         G4double g = 1.0,
                         G4double b = 1.0,
                         G4double a = 1.0);
                               // 0<=red, green, blue <= 1.0
```

每种成分的缺省值是 1.0。也就是说，缺省颜色是"白色" (不透明)。

例如，常用的颜色可以使用如下的值初始化：

```
    G4Colour  white   ()            ;  //
    G4Colour  white   (1.0, 1.0, 1.0) ;  //
    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
```

在一个 *G4Colour* 对像初始化之后，用户可以使用下面的存取函数存取它的成分。：

```
G4double G4Colour::GetRed   () const ; // Get the red   component.
G4double G4Colour::GetGreen () const ; // Get the green component.
G4double G4Colour::GetBlue  () const ; // Get the blue  component.
```

使用下面的存取函数将一个 *G4Colour* 对像传递给一个 *G4VisAttributes* 对像：

```
//----- Set functions of G4VisAttributes.
void G4VisAttributes::SetColour (const G4Colour& colour);
void G4VisAttributes::SetColor (const G4Color& color );
```

用户也可以之间设置 RGBA 成分：

```
//----- 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.);
```

下面使用 *G4Colour* 作为构造函数的参数也是可以接受的。

```
//----- Constructor of G4VisAttributes
G4VisAttributes::G4VisAttributes (const G4Colour& colour);
```

注意，为一个 *G4VisAttributes* 对像指定的颜色并不总是与可视化中显示的真实颜色相同。因为真实的颜色还受阴影和光照效果等的影响。通过用户选择的可视化引擎和图形系统，将这些效果综合起来，显示出真实的颜色。

8.3.3 强制线形框架（wireframe）和强制实体（solid ）显示方式

就像后面将要看到得，用户可以通过各种选项选择"显示方式"。例如，用户可以选择用户探测器以“线形框架”或者以“表面”的形式进行可视化。在前一种方式中，只有用户探测器的骨架会被显示，并且探测器看上去是透明的。在后一种方式中，用户探测器看上去是不透明的，而且是有阴影效果的。

强制线形框架和强制实体，这两种显示方式使混合线形框架和表面可视化成为可能(如果用户选择的图形系统支持这种可视化)。例如，用户可以只将探测器的外壳显示为"线条" (透明的)，以便观查内部细节。

使用下面的存取函数设置强制线形框架的显示方式：

```
void G4VisAttributes::SetForceWireframe (G4bool force);
```

如果用户给定的参数使 `true` ，被指定使用这些显示属性的对像将始终与线形框架的方式显示，即使用户已经请求以表面的方式显示。强制线形框架的缺省值是 `false`。

类似地，强制实体显示，即，强制那些对像始终使用表面进行可视化，使用下面的函数设置：

```
void G4VisAttributes::SetForceSolid (G4bool force);
```

强制实体方式的缺省值也是 `false`。

8.3.4 *G4VisAttributes* 的构造函数

下面是 *G4VisAttributes* 的构造函数

```
//----- Constructors of class G4VisAttributes
G4VisAttributes (void);
G4VisAttributes (G4bool visibility);
G4VisAttributes (const G4Colour& colour);
G4VisAttributes (G4bool visibility, const G4Colour& colour);
```

8.3.5 如何给一个逻辑体指定 *G4VisAttributes*

在构造用户的探测器元件时，用户可以给每个“逻辑体”指定可视化属性，以便随后对它们进行可视化(如果用户没有进行这项工作，图形系统将使用一些缺省值。)。用户不可以让一个实体（例如，*G4Box*）也拥有可视化属性；这是因为一个实体只能拥有几何信息。目前，用户尚不能使物理体拥有可视化属性，但是，有计划设计一种有效利用内存的方式来实现它；然而，用户可以使用一个临时指定的可视化属性对一个实体或物理体进行可视化。

G4LogicalVolume 类拥有一个 *G4VisAttributes* 的指针。使用下面的存取函数对这个字段进行设置和引用：

```
//----- 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;
```

下面是用于指定可视化属性的 C++源码的例子，用青色和强制线形框架方式对一个逻辑体进行可视化：

```
//----- 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
```

注意，可视化属性的生存期至少与被指定这些可视化属性的对像的生存期一样长；用户有责任保证这一点，并且当它们不在需要的时候删除它们(或者让它们在任务结束的时候自动删除)。

8.3.6 提供可被拾取的信息

从 5.0 版开始，Geant4 允许通过 *G4AttValue* 对像提供可拾取的信息。可以被存储的这种信息是 track 的动量或者沉积在一个 hit 中的能量。*G4AttValue* 对像只包含数值；for the long description and other sharable information the *G4AttValue* object refers to a *G4AttDef* object. 它们基于 HepRep 标准，http://heprep.freehep.org/。Geant4 也提供了一个 *G4AttDefStore*。

Geant4 也提供了一些在 *G4Trajectory* 和 *G4SmoothTrajectory* 中使用这个工具的缺省例子。*G4Trajectory::CreateAttValues* 显式对像如何建立 *G4AttValue* 对像，*G4Trajectory::GetAttDefs* 显示如何将来相应的 *G4AttDef* 对像和使用 *G4AttDefStore*。注意，CreateAttValues 的用户必须保证会删除它们；这是一种允许在要求的时候创建 *G4Trajectory* 对像的方式，this is a way of allowing creation on demand and leaving the *G4Trajectory* object, for example, free of such objects in memory。在 *G4VTrajectory.hh* 中的注释解释了更多的细节，通过浏览它所使用的两个方法也许可以获得额外的知识，这两个方法叫做 *G4VTrajectory::DrawTrajectory* 和 *G4VTrajectory::ShowTrajectory*。

从那些地方，用户可以看到如何为自己的 hits 实现同样的功能。基类的 no-action 方法，CreateAttValues 和 GetAttDefs 因该在用户具体类中重载。在 *G4VHit.hh* 的细节中解释了这方面的细节。

另外，用户可以给一个 *G4VisAttributes* 对像自由添加一个 *G4std::vector** 和一个 *G4std::vector** ，这个 *G4VisAttributes* 对像可以被一个 *G4LogicalVolume* 对像使用。

在写这个手册的时候，只有两个图形系统可以显示这种信息，这两个系统是使用 DAWN browser/renderer 的 DAWN(FILE)和使用 WIRED viewer 的 HepRepXML，但是，这个列表将会扩展。决不能使用这个在一个 *G4LogicalVolume* 对像中通过 *G4VisAttributes* 指针提供的信息。

Next section
Back to contents
Overview Contents Previous Next Geant4 User's Guide

8.4 折线，标记和文字

折线，标记和文字是在 graphics_reps 模块中定义的，只用于可视化。在这里将解释它们的定义和用法。

8.4.1 折线

折线是一系列连续的线段。它是在 graphics_reps 模块中定义的一个类 *G4Polyline* 。折线常用来对 tracking steps，粒子径迹，坐标轴，和其它用户定义的由线段组成的对像，进行可视化。

G4Polyline 是作为一个 *G4Point3D* 对像（顶点位置）列表定义的。这些顶点，用方法 push_back()放到一个 *G4Polyline* 对像中。

例如，Source listing 8.4.1 中定义了一个长 5cm 的红色的 x 轴。

```
 //----- C++ source codes: An example of defining a line segment
// Instantiate an empty 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
```

Source listing 8.4.1
定义一个长 5cm 的红色的 x 轴 。

8.4.2 标记（Marker）

这里将解释如何在 G4 可视化中使用 3D 标识。

什么是标记？

标记可以在 3D 空间中的任何位置设置记号。它们常用于在探测器元件中对粒子的 hits 进行可视化。标记是一个具有形状(正方形，圆，等)，颜色，的 2 维元件，它具有始终面对着视点(camera)，和可能以像素定义的大小等其它特殊属性。这里的"大小"是指"整体大小"，例如，圆的直径和正方形的边 (不过定义了直径和半径的存取函数，以避免结果的不确定性)。

构造标记的用户应决定是否将它以一个给定的大小显示。这个大小是通过设定 world 大小以 world 坐标表示的。另一种方式，用户也可以设定屏幕大小，用标记的屏幕大小来显示它。最后，用户也可以决定不设定任何大小；在这种情况下，系统将根据缺省标记中指定的大小来显示标记，这个缺省标记是在 *G4ViewParameters* 类中指定的。

作为缺省情况，在 G4 可视化中支持"正方形"和"圆"。前者用类 *G4Square* 进行描述，后者用类 *G4Circle* 进行描述：

标记类型	**Class Name**
圆	*G4Circle*
正方形	*G4Square*

这些类是从类 *G4VMarker* 继承的。它们有如下的构造函数：

```
//----- Constructors of G4Circle and G4Square
G4Circle::G4Circle (const G4Point3D& pos );
G4Square::G4Square (const G4Point3D& pos);
```

类 *G4VMarker* 的存取函数概述如下。

标记的存取函数

Source listing 8.4.2 显示从基类 *G4VMarker* 派生的存取函数。

```
//----- 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};
```

Source listing 8.4.2
从基类 *G4VMarker* 继承的存取函数。

Source listing 8.4.3 显示了定义非常小的红圈的 C++ 源码，这个红圈是一个直径为一个相素的点。这样的点常用于 hit 的可视化。

```
//----- 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
```

Source listing 8.4.3
定义一个非常小红圈的 C++ 源码。

8.4.3 文字（Text）

文字，就是字符串，它用来可视化各种描述信息，如，粒子名，能量，坐标名等。文字是用类 *G4Text* 来表示的。下面是改类的构造函数：

```
//----- Constructors of G4Text
G4Text (const G4String& text);
G4Text (const G4String& text, const G4Point3D& pos);
```

这里的参数 `text` 是将要被可视化的文字(字符串)，`pos` 是 text 进行可视化的位置。

注意，类 *G4Text* 也继承了类 *G4VMarker*。文字的大小被称作"字体大小(font size)"，也就是文字的高度。所有上面提到的为类 *G4VMarker* 定义的存取函数在这里都可以使用。另外，下面的存取函数也是可用的：

```
//----- 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;
```

方法 `SetText()` 定义将要被可视化的文字， `GetText()` 返回 `SetText()`定义的文字。方法 `SetOffset()` 用屏幕坐标定义 x (水平)和 y (垂直)偏移量。缺省情况，两个偏移都为 0，文字的起始位置是在构造函数或者方法 `G4VMarker:SetPosition()`中指定的 3D 位置。文字偏移量的单位应跟它所使用的大小的单位相同，要么是 world 大小单位，要么是屏幕大小单位。

Source listing 8.4.4 显示了以下面的性质定义文字的 C++源码：

- 文字: "Welcome to Geant4 Visualization"
- 位置: (0.,0.,0.) 以 world 坐标表示
- 水平偏移: 10 pixels
- 垂直偏移: -20 pixels
- 颜色: blue (缺省)

```
//----- 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
```

Source listing 8.4.4
定义文本的例子。

Next section
Back to contents
Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Visualization

8.5 生成一个可视化的可执行程序

在这里将要解释如何编写 `main()` 函数并且生成可执行程序，实现用户选择的可视化引擎。为了使用户的 G4 可执行程序可以进行可视化，用户必须将它与所实现的可视化引擎一起编译。有许多可视化引擎中供用户选择，不过一次不需要使用所有的引擎。

8.5.1 可用的可视化引擎

用户可以根据自己的需要，安装相应的可视化引擎。按照用户不同的可视化目标，可以选择一个或多个将在编译过程中实现的可视化引擎。每个引擎的特性和一些注意点在 8.6 节 "**Visualization Drivers**"中作了简单描述，它们的细节可以在里面提到的网页上找到。一些高级和/或与引擎相关的主题在 8.10 节"**More About visualization**"中也作了叙述。

Table 8.1 以子母顺序列出了可用的可视化引擎。为了使这些可视化引擎工作，首先必须按照它们对应的图形系统。Table 8.1 概述了可用的可视化引擎和它们要求的图形系统，以及适用的操作系统平台。

可视化引擎	要求的 3D 图形系统	操作系统平台
DAWNFILE	Fukui Renderer DAWN	UNIX, Windows
DAWN-Network	Fukui Renderer DAWN	UNIX

HepRepFile	WIRED event display	UNIX, Windows
OPACS	OPACS, OpenGL	UNIX, Windows with X environments
OpenGL-Xlib	OpenGL	UNIX with Xlib
OpenGL-Motif	OpenGL	UNIX with Motif
OpenGL-Win32	OpenGL	Windows
OpenInventor-X	OpenInventor, OpenGL	UNIX with Xlib or Motif
OpenInventor-Win32	OpenInventor, OpenGL	Windows
RayTracer	(JPEG viewer)	UNIX, Windows
VRMLFILE	(VRML viewer)	UNIX, Windows
VRML-Network	(VRML viewer)	UNIX

Table 8.1
所有可用的可视化引擎，与子母顺序排序

除非环境变量 G4VIS_NONE 设置为"1"，否则，那些不依懒外部库的可视化引擎将在 G4 库安装的时候自动编译到 G4 库内。(这里“G4 库的安装”是指“通过编译生成 G4 库”)这些被自动安装的可视化引擎是：DAWNFILE, HepRep-File, RayTracer, 和 VRMLFILE。

对于其它的引擎在缺省时是不会被安装的，如 OPACS，OpenGL 和 OpenInventor。另外，DAWN-Network 引擎和 VRML-Network 引擎缺省也不安装，因为它们要求被安装机器的网络设置 。要安装它们，应在安装 G4 库之前，设置环境变量 "G4VIS_BUILD_DRIVERNAME_DRIVER"为"1"：

```
setenv G4VIS_BUILD_DAWN_DRIVER        1  # DAWN-Network driver
setenv G4VIS_BUILD_OPACS_DRIVER       1  # OPACS driver
setenv G4VIS_BUILD_OPENGLX_DRIVER     1  # OpenGL-Xlib driver
setenv G4VIS_BUILD_OPENGLXM_DRIVER    1  # OpenGL-Motif driver
setenv G4VIS_BUILD_OIX_DRIVER         1  # OpenInventor-Xlib driver
setenv G4VIS_BUILD_VRML_DRIVER        1  # VRML-Network
```

如果环境变量 G4VIS_NONE 不设置为"1"，这些环境变量中的任何设置都会引起设置一个同名的 C 预处理标志；同样，C 预处理标志 G4VIS_BUILD 被设置为“1”(查看 config/G4VIS_BUILD.gmk)，将使被选的可视化引擎在 G4 库安装的时候一起被安装到 G4 库内。

8.5.2 如何在一个可执行程序中实现可视化引擎

用户可以在自己的 G4 可执行程序中实现并使用自己希望的可视化引擎。这些引擎必须是失效已经安装的 G4 库内的。如果用户请求一个不可用的引擎，将会得到一个警告信息。

为了实现可视化引擎，用户在编译自己的 G4 可执行程序前，必须完成下列工作：

1. 继承在中定义 `source/visualization/management` 的基类 *G4VisManager*，并编写自己的 visualization manager，然后注册用户所选择的可视化引擎。要求用户实现一个纯虚函数 `void RegisterGraphicsSystems()`。
2. 在 `main()`函数中实例化 visualization manager 并初始化。
3. 如果用户所选的可视化引擎依懒于外部库，那么需要设置环境变量 `"G4VIS_USE_DRIVERNAME"` 为"1"。

对于第 1 点，例如，下面的程序注册 DAWNFILE 和 OpenGL-Xlib 可视化引擎：

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

更多细节参看 `examples/novice/N03/src/ExN03VisManager.cc` 。

对于第 2 点，在下一节中将解释如何编写 `main()` 函数。

对应第 3 点，缺省情况下，用户可以使用 DAWNFILE, HepRep, RayTracer, VRMLFILE 可视化引擎。另外，用户可以选择 DAWN-Network, OPACS, OpenGL-Xlib, OpenGL-Motif, OpenInventor, 和 VRML-Network 引擎，它们中的每一个都可以通过设置适当的环境变量进行选择：

```
setenv G4VIS_USE_DAWN        1
setenv G4VIS_USE_OPACS       1
setenv G4VIS_USE_OPENGLX     1
setenv G4VIS_USE_OPENGLXM    1
setenv G4VIS_USE_OIX         1
setenv G4VIS_USE_VRML        1
```

(当然，在它们编译的时候被选择的可是化引擎必须是已被安装到 G4 库内的)如果没有设置 G4VIS_NONE 为“1”，这些设置将引擎设置一个同名的 C 预编译标志。

同样，如果环境变量 G4VIS_NONE 没有设置为“1”，GNUmakefile `config/G4VIS_USE.gmk` 将自动设置 C 预编译标志 G4VIS_USE。这个标志可以在 `main()` 函数中使用。

对应用户所选择的可视化引擎和图形系统，可能必须设置附加的环境变量。例如，OpenGL 引擎可能要求设置 `OGLHOME` ，这个环境变量表示 OpenGL 库的安装位置。有关细节，参看 8.6 节 "**Visualization Drivers**" 和该节中提到的网页。同时请看 `geant4/source/visualization/README`.

一个 set-up 文件的例子

下面是一个`.cshrc` 文件的一部分，这个文件用于在 Linux 平台上创建可视化的 G4 可执行程序。更多细节参看文件 `source/visualization/README` 。同时请看 `config/G4VIS_BUILD.gmk` 和 `config/G4VIS_USE.gmk`。

```
#################################################################
# Main Environmental Variables for GEANT4 with Visualization #
#################################################################

### Platform
setenv G4SYSTEM Linux-g++

### CLHEP root directory
setenv CLHEP_BASE_DIR /usr/local

### OpenGL root directory
setenv OGLHOME /usr/X11R6

### G4VIS_BUILD
###   Incorporation of OpenGL-Xlib and OpenGL-Motif drivers
###   into Geant4 libraries.
setenv G4VIS_BUILD_OPENGLX_DRIVER 1
setenv G4VIS_BUILD_OPENGLXM_DRIVER 1

### G4VIS_USE
###   Incorporation of OpenGL-Xlib and OpenGL-Motif drivers
###   into Geant4 executables.
setenv G4VIS_USE_OPENGLX      1
setenv G4VIS_USE_OPENGLXM     1

### Viewer for DAWNFILE driver
### Default value is "dawn".  You can change it to, say,
### "david" for volume overlapping tests
# setenv G4DAWNFILE_VIEWER david
```

```
### Viewer for VRMLFILE drivers
setenv G4VRMLFILE_VIEWER vrmlview

########## end
```

Source listing 8.5.1
一个用于 Linux 平台的.cshrc setup 文件的一部分。

8.5.3 如何编写 main() 函数

现在，解释如何为 G4 可视化编写 main() 函数。

为了用户的 Geant4 可执行程序可以进行可视化，用户必须在 main()函数中实例化自己的 Visualization Manager，并初始化。Visualization Manager 的内核是类 *G4VisManager*，这个类在 G4 工具包的可视化模块中定义。这个类要求用户实现一个纯虚函数 RegisterGraphicsSystems(), 用户可以通过继承 *G4VisManager*，编写自己的类 *MyVisManager*，来完成这个工作。在 RegisterGraphicsSystems()的实现中，描述了注册备选可视化引擎的步骤。

用户可以使用在 MyVisManager.hh 和 MyVisManager.cc 中定义的类 *MyVisManager* 的实现，这两个文件在 visualization/management/include 目录下。不过，用户可以根据自己的要求，编写自己的派生类并实现它的方法 RegisterGraphicsSystems()。

Source listing 8.5.2 显示了在 main() 函数中进行的实例化和初始化 。

```
//----- C++ source codes: Instantiation and initialization of G4VisManager

.....
// Your Visualization Manager
#include "MyVisManager.hh"
.....

// Instantiation and initialization of the Visualization Manager
#ifdef G4VIS_USE
G4VisManager* visManager = new MyVisManager;
visManager -> initialize ();
#endif

.....
#ifdef G4VIS_USE
delete visManager;
#endif
```

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

Source listing 8.5.2
在 main() 函数中实例化 G4VisManager，并初始化。

还有一种方法是，用户实现一个空的 RegisterGraphicsSystems() 函数，而在 main()函数中直接注册希望使用的可视化引擎。参看 Source listing 8.5.3.

```
//----- C++ source codes: How to register a visualization driver directly
//                     in main() function

.....
G4VisManager* visManager = new MyVisManager;
visManager -> RegisterGraphicsSystem (new MyGraphicsSystem);
.....
delete visManager

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

Source listing 8.5.3
另一种在 main() 函数注册可视化引擎的方式。

用户**不要忘记**删除已实例化的 Visualization Manager。注意一个用于 G4 可视化的图形系统也许是可以独立运行的。在这中情况下，*G4VisManager* 的析构函数必须终止该图形系统并且/或者关闭连接。

对于 Visualization Manager 的实例化，初始化，和删除，推荐使用宏。在用户使用在 config 目录下的 GNUmakefile 编译 G4 可执行程序的时候，如果没有设置环境变量 G4VIS_NONE，那么宏 G4VIS_USE 将被自动定义。

Source listing 8.5.4 显示了用于 G4 可视化的 main() 函数的例子。

```
//----- C++ source codes: An example of main() for visualization
.....
#include "MyVisManager.hh"
.....

int main()
{
```

```
    // 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 MyVisManager;
    visManager -> initialize ();
#endif

    // Event loop
    // Define (G)UI terminal
    G4UIsession * session = new G4UIterminal
    session->sessionStart();

    delete session;
    delete runManager;

#ifdef G4VIS_USE
    delete visManager;
#endif

    return 0;
}

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

Source listing 8.5.4
用于 G4 可视化的 `main()` 函数的例子。

另外需要注意的是，在初始化 Visualization Manager 的时候，可以显示一些关于可视化引擎的有用信息。用户可以通过设置 verborsity 标志为正整数值，来实现这些冗余信息的输出。例如，在用户的 `main()`函数中，加入如下代码：

```
...
G4VisManager* visManager = new MyVisManager ();
```

```
visManager -> SetVerboseLevel (1);
visManager -> Initialize ();
...
```

Next section
Back to contents
Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Visualization

8.6 可视化引擎

缺省情况下，许多种可视化引擎是可用的。用户可以在 G4 可执行程序编译的时候，按照用户的可视化要求，选择一个或者多个引擎。每个引擎的特性和注意点将在本节论述。更多的细节请查看相应的网页。一些高级和/或与可视化引擎相关的特性在 8.10 节"**More About Visualization**"中也有论述。

如果用户快速的了解 G4 可视化，那么可以跳过本节，但是，在用户真正开始使用 G4 可视化前，强烈推荐用户阅读本节内容。

8.6.1 DAWN 可视化引擎

DAWN 可视化引擎是 G4 与 Fukui Renderer DAWN 的接口，后者是由 Satoshi Tanaka, Minato Kawaguti (Fukui University)等人开发的。它是一个矢量化的 3D PostScript 处理程序，非常适合产生那些用于报告和文档的高质量输出。它对探测器几何的准确调试也是很有帮助的。支持探测器模拟的远程可视化，离线重复可视化，剖面观察，和其它许多有用的特性。当执行可视化的时候，系统将自动调用一个 DAWN 进程作为 G4 的协处理进程，通过文件或者 TCP/IP socket 实现进程间的通迅，传递 3D 数据。

当使用 DAWN 引擎进行 Geant4 可视化时，可视化的结果被自动保存在当前目录下，一个名叫 `g4.eps` 的文件中。这个文件描述了可视化结果的矢量化(Encapsulated) PostScript 数据。

有两种 DAWN 引擎，DAWNFILE 引擎和 DAWN-Network 引擎。通常，推荐使用 DAWNFILE 引擎，因为在不使用网络的情况下它更快更安全。

DAWNFILE 引擎通过一个中间文件将 3D 数据发给 DAWN，这个文件在当前目录下，名为 `g4.prim` 。这个文件可以用于在 Geant4 停止运行之后，通过手动调用 DAWN，再次对这些3D 数据进行可视化：

```
% dawn g4.prim
```

DAWN-Network 引擎几乎与 DAWNFILE 引擎相同，除了

- 3D 数据经 TCP/IP socket(缺省)或者指定管道(pipe)传送给 DAWN，以及，

- 它可应用与远程可视化 (细节参看 8.10 节 "**其它**").

如果用户没有为主机进行网络设置，那么需要设置环境变量 `G4DAWN_NAMED_PIPE` 为"1"，例如， `% setenv G4DAWN_NAMED_PIPE 1`。这个设置将使系统从缺省的 socket 联接切换到同一个主机的指定管道。DAWN-Network 引擎也保存 3D 数据到当前目录下的文件 `g4.prim` 。

更多信息：

- Fukui Renderer DAWN:
  http://geant4.kek.jp/GEANT4/vis/DAWN/About_DAWN.html
- DAWNFILE 引擎：
  http://geant4.kek.jp/GEANT4/vis/GEANT4/DAWNFILE_driver.html
- DAWN-Network 引擎：
  http://geant4.kek.jp/GEANT4/vis/GEANT4/DAWNNET_driver.html
- 定制 DAWN 和 DAWN 引擎的环境变量：
  http://geant4.kek.jp/GEANT4/vis/DAWN/DAWN_ENV.html
  http://geant4.kek.jp/GEANT4/vis/GEANT4/g4vis_on_linux.html
- DAWN 格式(g4.prim 格式)手册：
  http://geant4.kek.jp/GEANT4/vis/DAWN/G4PRIM_FORMAT_24/
- Geant4 Fukui University Group Home Page:
  http://geant4.kek.jp/GEANT4/vis/

8.6.2 HepRepFile 引擎

HepRepFile 引擎建立一个 HepRep 文件，这个文件适合于用 WIRED 事件显示客户程序进行观察。HepRep 图形格式的更多细节在 http://heprep.freehep.org 可以找到。这个引擎产生的 HepRep 版本是 1。每次调用/vis/viewer/update 重写文件 G4HepRep.xml。使用 WIRED Event Display 查看这个文件。WIRED Event Display 的网址是 http://www.slac.stanford.edu/BFROOT/www/Computing/Graphics/Wired/。因为 WIRED 可以读取压缩格式和非压缩格式的 xml 文件，所以用户可以压缩 xml 文档以节约磁盘空间。用 gzip 压缩后，文件的大小大约只有它原始大小的 5%。

更多信息：

- WIRED 主页：
  http://www.slac.stanford.edu/BFROOT/www/Computing/Graphics/Wired/.
- HepRep 图形格式：
  http://heprep.freehep.org

8.6.3 OPACS 引擎

OPACS 是一个基于 X windows 和 OpenGL 的可视化环境。它是主要由 Guy Barrand 在 LAL (Orsay, France)进行开发的。它是用 ANSIC 编写的，有很高的可移植性(UNIX, NT/X11, VMS)。

这个可视化环境同时还有一个 widget manager：就是被 G4 GUI senssion 所使用的 Wo 软件包。Wo permits one to interactively create interpreted Xt widget hierarchies. Widget callbacks are also interpreted.缺省条件下，使用 Bourne shell 语法解释器"osh" ，其余的如 G4 命令解释器，可以向 Wo 进行声明。

Xo 工具包括用作 G4 可视化引擎的 XoCamera 3D viewer。这个工具用来察看被 Go 图形包处理过的场景(scenes)。Go 使用 OpenGL 来进行可视化。Xo/Go 小组提供了交乎式的场景操作，拾取工具，可以非常方便的生成 PostScript, GIF, VRML, DAWN 格式的文件。

OPACS 的长处是它在用户接口和图形直接的灵活性和一致性。同时请参看有关接口模块的章节，那里有一些 Wo G4UI session 的论述。

更多信息：

- http://www.lal.in2p3.fr/OPACS

8.6.4 OpenGL 引擎

这些引擎是由 John Allison 和 Andrew Walkden (University of Manchester)开发的。它是与事实上的 3D 标准图形库 OpenGL 的接口。它非常适合于实时快速可视化和演示。快速可视化是通过硬件加速、重用存储在显示列表中的形状，等方法来实现的。同时支持 NURBS 可视化。

目前已经完成几个版本的 OpenGL 引擎。缺省情况下，用于 Xlib, Motif 和 Win32 平台的版本是可用的。每个版本都有两个模式：即时模式和存储模式。前者在数据量上没有任何限制，而后者用于可视化大量重复性数据的时候比较快，非常适合于动画。

更多消息(OpenGL 和 Mesa):

- http://www.opengl.org/
- http://www.mesa3d.org

8.6.5 OpenInventor 引擎

这个引擎是基于 Joe Boudreau (Pittsburgh University)最初的 the "Hepvis class library" ，由 Jeff Kallenbach (FNAL)开发完成的。OpenInventor 引擎和 Hepvis 类库是基于用于科学可视化的 OpenInventor 技术。它们由很好的可扩展性，如，被选中对像的属性编辑。用它们可以实现虚拟现实可视化。

用户可以将一个已可视化的 3D 场景保存为一个 OpenInventor 格式的文件，用于以后再次可视化这个场景。

OpenInventor 引擎可以使用在 LAL 开发的免费 Inventor 内核"SoFree"，这个内核于 Hepvis 类库协同工作。

更多信息 (OpenInventor 引擎， Hepvis 和 SoFree)：

- http://www-pat.fnal.gov/graphics/HEPVis/www
- http://www.lal.in2p3.fr/Inventor

更多信息 (OpenInventor):

- http://www.sgi.com/software/inventor.html
- Josie Wernecke, "The Inventor Mentor", Addison Weslay (ISBN 0-201-62495-8)
- Josie Wernecke, "The Inventor Toolmaker", Addison Weslay (ISBN 0-201-62493-1)
- "The Open Inventor C++ Reference Manual", Addison Weslay (ISBN 0-201-62491-5)

8.6.6 *RayTracer* 引擎

这个引擎是由 Makoto Asai 和 Minamimoto (Hirosihma Instutute of Technology)开发的。它使用 G4 粒子跟踪的例程进行射线跟踪的可视化。因此，它可以用来进行调试粒子跟踪的例程。它非常适合于为报告输出照片质量的图像，和探测器几何的直观调试。

更多信息：

- http://hitds1.cc.it-hiroshima.ac.jp/Geant4/g4ray/whatisg4r_J.html

8.6.7 VRML 引擎

这些引擎是由 Satoshi Tanaka 和 Yasuhide Sawada (Fukui University)开发的。它们产生描述了一些 3D 场景的 VRML 文件，这些场景将使用一个合适的 VRML viewer，在本地或者远程主机上进行可视化。它实现了使用 WWW 浏览器的虚拟现实可视化。有许多优秀的 VRML viewer 允许用户进行很多交互式的操作，如探测器的旋转，在探测器内部或者粒子簇射中漫游，详细探测器几何的交换式检查，等。

有两种 VRML 引擎：VRMLFILE 引擎，和 VRML-Network 引擎。通常，推荐使用 VRMLFILE 引擎，因为它在不使用网络的情况下更快更安全。

VRMLFILE 引擎通过在当前目录下，名叫 `g4.wrl` 的中间文件发送 3D 数据给用户的 VRML viewer，这个 VRML viewer 必须与 G4 在同一主机上，且正在运行。`g4.wrl` 可以用户以后的再次可视化。在进行可视化的适合，用户因该通过环境变量 `G4VRML_VIEWER` ，事先指定 VRML viewer 的名字。例如，

```
% setenv G4VRML_VIEWER  "netscape"
```

这个环境变量缺省是 `NONE`，表示只产生 `g4.wrl` 文件，不调用任何 viewer。

VRML-Network 引擎是用于远程可视化的。详情请看 8.10 节 "**其它**" 和下面的网页。

更多信息 (VRML 引擎):

- http://geant4.kek.jp/GEANT4/vis/GEANT4/VRML_file_driver.html
- http://geant4.kek.jp/GEANT4/vis/GEANT4/VRML_net_driver.html

VRML 的示例文件：

- http://geant4.kek.jp/GEANT4/vis/GEANT4/VRML2_FIG/

更多信息(VRML 语言和浏览器):

- http://www.vrmlsite.com/

Next section
Back to contents
Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Visualization

8.7 交互式可视化

在一个交互式可视化会话过程中常用的可视化命令将在本节进行描述。为了简化，这里假定 G4 可执行文件使用的是 DAWNFILE 和 OpenGL-Xlib 引擎。有关建立可视化的可执行程序的细节请参看 8.5 节 。

8.7.1 场景 Scene,场景处理器 scene handler，和 浏览器 viewer

用户可以使用交互式的可视化命令实现几乎所有的 Geant4 可视化功能。在使用这些可视化命令的时候，有必要了解一下"scene", "scene handler"和 "viewer"的概念。"scene"是一个可视化的 3D 原始数据集。"scene handler" 是一个图像数据模式化工具，它对在"scene"中的原始数据进行处理以供后续的可视化。"viewer"根据"scene handler"处理完成的数据生成最终图像。这些概念与它们之间的关系，与 MFC 中“文档”、“模板”、“视”的概念与关系相类似。实际上，一个"scene handler"和一个"viewer"对应于一个可视化引擎。

Geant4 可视化的典型步骤如下：

1. 建立一个 scene handler 和一个 viewer。
2. 建立一个空的 scene。
3. 将原始 3D 数据添加到已建立的 scene 中。
4. 将当前的 scene handler 链接到当前 scene。
5. 设置相机参数、绘图方式（轮廓线/面）等
6. 使 viewer 执行可视化
7. 声明可视化结束并显示图像。

注意，以上的列表并不表示用户必须执行 7 条可视化命令，可以使用复合命令一次执行多个可视化命令。

8.7.2 调用可视化引擎： /vis/open 命令

命令"/vis/open" 调用一个可视化引擎，这对应上面的第 1 步。

- **命令**
  /vis/open [driver_tag_name]
- **参数**
  一个可用的可视化引擎的名字或模式。
- **功能**
  功能
  建立一个可视化引擎，即一个 scene hander 和一个 viewer。
- **例：建立立即模式的 OpenGL-Xlib 引擎**
  Idle> /vis/open OGLIX
- **符注**
  列出可用的图形引擎标识名：

  Idle> help /vis/open
  或者
  Idle> help /vis/sceneHandler/create

  这个命令的结果，举例如下：
  .....
  Candidates : DAWNFILE OGLIX OGLSX
  .....
-

8.7.3 基本 camera 工作：/vis/viewer/ 命令

在"/vis/viewer/"这个命令路径下的命令 设置当前 viewer 的 camera 参数和绘图样式 ，对应于上面的第 5 步。注意，因该为每个 viewer 设置 camera 参数和绘图样式。它们可以用户命令"/vis/viewer/reset"进行初始化，设置为缺省值。

- **命令**
  /vis/viewer/reset
- **功能**
  复位相机参数和绘图样式。

- **命令**
  `/vis/viewer/set/viewpointThetaPhi [<theta>] [<phi>] [<deg|rad>]`
- **参数**
  参数"theta"和"phi"分别是相机的极角和方位角，缺省单位是“度”。
- **功能**
  设置一个在(theta, phi)方向的视点。
- **例：设置(70 deg, 20 deg)方向的视点**
  `Idle> /vis/viewer/set/viewpointThetaPhi 70 20`
- **附注**
  应为每一个 viewer 设置相机参数。它们可以用"`/vis/viewer/reset`"初始化。

- **命令**
  `/vis/viewer/zoom [<scale_factor >]`
- **参数**
  刻度因子，这个命令使图像的放大所设置的刻度因子倍。
- **功能**
  放大/缩小 图像。
- **例：放大 1.5 倍**
  `Idle> /vis/viewer/zoom 1.5`
- **附注**
  应为每一个 viewer 设置相机参数。它们可以用"`/vis/viewer/reset`"初始化。

- **命令**
  `/vis/viewer/set/style [style_name]`
- **参数**
  候选参数是"wireframe" 和 "surface". ("w" 和 "s"为它们的缩写)
- **功能**
  设置绘图样式。
- **例:设置绘图样式为"surface"**
  `Idle> /vis/viewer/set/style surface`
- **附注**
  应为每个 viewer 设置绘图样式。绘图样式是用命令 "`/vis/viewer/reset`"初始化的。

- **命令**
  `/vis/viewer/flush`
- **功能**
  声明可视化结束并显示图像。

- **附注**
  为了完成可视化，命令"`/vis/viewer/flush`"应在"/vis/drawVolume", "/vis/specify" 等之后执行 。它属于上面的第 7 步。

8.7.4 一个物理体的可视化 `/vis/drawVolume` 命令

命令"/vis/drawVolume"可视化一个物理体，完成上面的的步骤 2、3、4，命令 "/vis/viewer/flush" 应在这个命令之后执行，以声明结束可视化。(步骤 7)

- **命令**
  `/vis/drawVolume [<physical-volume-name>]`
- **参数**
  物理体名，缺省是"world", 可忽略。
- **功能**
  建立一个由给定物理体组成的 scene，并请求当前 viewer 绘出相应图像，这个 scene 将成为当前 scene。命令 "/vis/viewer/flush" 应在这个命令之后执行，以声明结束可视化。
- **例:在给定坐标系中可视化整个世界**

```
Idle> /vis/drawVolume
Idle> /vis/scene/add/axes 0 0 0 500 mm
Idle> /vis/viewer/flush
```

8.7.5 一个逻辑体的可视化 `/vis/specify` 命令

命令"/vis/specify"可视化一个逻辑体，完成上面的步骤 2、3、4 和 6，命令 "/vis/viewer/flush" 应在这个命令之后执行，以声明结束可视化。(步骤 7)

- **命令**
  `/vis/specify [logical-volume-name]`
- **参数**
  逻辑体名
- **功能**
  建立一个由给定逻辑体组成的 scene 并请求当前 viewer 绘出相应图像，这个 scene 将成为当前 scene。
- **例(在给定坐标系中可视化一个选定的逻辑体)：**

```
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
```

8.7.6 径迹可视化：`/vis/scene/add/trajectories` 命令

命令 "`/vis/scene/add/trajectories`" 添加径迹到当前 scene。注意用户必须事先通过执行"`/tracking/storeTrajectory 1`"，存储相应径迹。使用命令"`/run/beamOn`"执行可视化。

- **命令**
  `/vis/scene/add/trajectories`
- **功能**
  这个命令添加径迹到当前 scene。这些添加到 scene 中径迹在事件结束的时候被显示。
- **例:径迹可视化**
  ```
  Idle> /tracking/storeTrajectory 1
  Idle> /vis/scene/add/trajectories
  Idle> /run/beamOn 10
  ```
- **Additional note 1**
  在例子 examples/novice/N03 中，不需要执行命令"`/vis/scene/add/trajectories`"，因为在事件 action 中已经显示的描述了方法 `G4Trajectory::DrawTrajectory()`。因此这个命令不需要通过 (G)UI 执行。
- **Additional note 2**
  为了让使用`/run/beamOn` 可视化的命令工作，run action 和事件 action 应被适当的实现。在例子 examples/novice/N03 中的实现如下：
-
```
      void ExN03RunAction::BeginOfRunAction(const G4Run* aRun)
      {
        .....

        if (G4VVisManager::GetConcreteInstance())
        {
          G4UImanager* UI = G4UImanager::GetUIpointer();
          UI->ApplyCommand("/vis/scene/notifyHandlers");
        }
      }

      void ExN03RunAction::EndOfRunAction(const G4Run* )
      {
        if (G4VVisManager::GetConcreteInstance()) {
           G4UImanager::GetUIpointer()->ApplyCommand("/vis/viewer/update");
        }
      }
```

8.7.7 如何将一个可视化了的 views 保存为 PostScript 文件

大多数可视化引擎提供将可视化了的 view 保存为 PostScript 文件(或者 Encapsulated PostScript (EPS)文件)的方法。

与 Fukui Renderer DAWN 协同工作的 DAWNFILE 引擎，使用"analytical hidden-line/surface removal"产生"矢量化的" PostScript 数据，所以，它非常适合与用于报告，文档，和几何调试的高质量输出。在 DAWNFILE 引擎的缺省始终下，当每次执行可视化时，将自动产生名叫 "g4_00.eps, g4_01.eps, g4_02.eps,..." 的 EPS 文件，然后，自动调用一个叫"gv"的 PostScript viewer 可视化这些产生的 EPS 文件。

对于大数据量的情况，可能需要一些时间用于产生矢量化的 PostScript 数据。在这种情况下，因该先用一个更快的可视化引擎对这个 3D scene 进行可视化，作为预览，然后再用 DAWNFILE 引擎进行最终的可视化。例如，下面的例子，先使用 OpenGL-Xlib(即时模式)引擎可视化整个探测器，然后用 DAWNFILE 引擎产生一个 EPS 文件 g4_XX.eps 来保存这些可视化了的 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
```

这是一个很好的例子，它显示了这些可视化引擎是相互补充的。

在 OPACS 引擎中，可以在它的 Xo viewer 的右键弹出式菜单中选择"PostScript"选项，以产生一个 EPS 文件（out.ps），作为一个可视化了的 view 的硬拷贝。产生的这个 EPS 文件在当前目录。

在 OpenInventor 引擎中，用户可以简单在它们的 GUI 点击"Print"按钮，生成一个 PostScript 文件，同样，这个文件也是一个可视化了的 view 的硬拷贝。

OpenGL-Motif 引擎也有一个生成 PostScript 文件的菜单。它产生矢量化和光栅化的两种 PostScript 数据。在产生矢量化 PostScript 数据时，将各种形状的面元化分为小的近似三角形后，基于绘图程序的算法，那些隐藏表面将被删除。

8.7.8 Culling

"Culling" 是指跳过一个 3D scene 的部分可视化。对于避免可视化了的 views 的复杂性，保持 3D scene 的透明特性，和快速可视化等情况，culling 将是非常有用的。

Geant4 可视化支持下面的 3 中 culling：

- 不可见物理体的 Culling
- 低密度物理体的 Culling
- 被其它物理体覆盖了的物理体的 Culling

为了打开上述的一种或多种 culling，也就是激活 culling，需要打开全局 culling 标志。

Table 8.3 概述缺省的 culling 策略。

Culling Type	**缺省值**
全局 global	ON
不可见 invisible	ON
低密度 low density	OFF
覆盖子体 covered daughter	OFF

Table 8.3
缺省 culling 策略。

低密度 culling 的缺省密度阈值是 0.01 g/cm^3。

缺省 culling 测量可以使用下面的交互式可视化命令进行修改。(下面的参数 `flag` 取值为 `true` 或者 `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
```

8.7.9 剖面观察

切片 sectioning

"Sectioning" 是指在给定平面附近截取一个 3D scene 的薄片。目前，OpenGL 引擎支持这个功能。切片是在执行可视化之前，通过设置一个切片平面来实现的。这个切片平面可以用下面的命令设置，

```
/vis/viewer/set/sectionPlane x y z units nx ny nz
```

这里的矢量 (x,y,z) 定义了在切片平面上的一个点，矢量 (nx,ny,nz)定义了这个切片平面的法向量。例如，下面的命令设置切片平面为在 x = 2 cm 处的一个 yz 平面：

```
Idle> /vis/viewer/set/sectionPlane  2.0  0.0  0.0  cm  1.0  0.0  0.0
```

切除 Cutting away

"Cutting away"是指从一个 3D 空间删除一个平面一侧的空间。目前，DAWNFILE 引擎支持这个功能。按照下面的方法：

- 用 DAWNFILE 引擎进行可视化，以产生描述整个 3D scence 的文件 `g4.prim`。
- 使用应用程序"DAWNCUT"读取上面生成的文件，以制作一个 cutting away 的 view。

细节请看下面的网页：http://geant4.kek.jp/GEANT4/vis/DAWN/About_DAWNCUT.html.

8.7.10 探测器几何树的可视化

Geant4 支持"tree drivers"，它用于可视化一个探测器几何树。目前，已实现了两个树型结构引擎

- ASCII tree 引擎
- GAG tree 引擎

ASCII tree driver 通过显示物理体名的适当标识来可视化一个探测器几何的树型结构。GAG tree driver 使用 GAG GUI (http://erpc1.naruto-u.ac.jp/~geant4)完成了相同的功能。稍后将要实现的 XML tree driver，它使用 XML 来显示树型结构。

注意，用户必须要使用这些 tree driver，必须在自己的 Visualization Manager 类中进行注册：

```
RegisterGraphicsSystem (new G4ASCIITree);
RegisterGraphicsSystem (new G4GAGTree  );
```

参看例子 `examples/novice/N03/src/ExN03VisManager.cc`。

- **命令**
  `/vis/drawTree [<physical_volume_name>] [<driver_name>]`
  候选的 driver 名是"ATree" (ASCII tree，缺省)和 "GAGTree"。
- **功能**
  可视化一个探测器几何树。
- **参数**
  位于树顶层的物理体名，和 tree-driver 名。
- **例：可视化整个几何**
-
-
```
Idle> /vis/drawTree ! ATree
.....
"Calorimeter", copy no. 0
   "Layer", copy no. -1 (10 replicas)
      "Absorber", copy no. 0
         "Gap", copy no. 0
.....
```
- **附注**
  上例中的字符'!'表示“缺省参数”。这是 Geant4 交互式命令的约定。

- **命令**
  `/vis/XXXTree/verbose [<verbosity>]`

  "XXX" 对应于 ASCII tree driver 是 `"ASCIITree"` ，对应于 GAG tree driver 是`"GAG"` 。verbosity 的值为 0（缺省）到 10：
-
- `< 10: -` 不打印重复逻辑体的子几何体。
- `-` 不重复打印复制的几何体。
- `>= 10:` 打印所有物理体。
- `>= 0:` 打印物理体名。
- `>= 1:` 打印逻辑体名。
- `>= 2:` 打印实体名和类型。
- **功能**
  指定可视化树型结构的详细程度。
- **例子：**
-
```
Idle> /vis/ASCIITree/verbose 1
Idle> /vis/drawTree ! ATree
.....
"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"
- .....

8.7.11 Tutorial 宏

下面是在 examples/novice/N03/visTutor/目录下的一些 tutorial 宏：

- exN03Vis0.mac:
  一个最简单的宏，用于演示探测器几何和事件的可视化
- exN03Vis1.mac:
  一个基本的宏，用于探测器几何可视化
- exN03Vis2.mac:
  一个基本的宏，用于事件可视化
- exN03Vis3.mac:
  一个基本的宏，用于演示绘图样式
- exN03Vis4.mac:
  一个可视化逻辑体的例子
- exN03Vis5.mac:
  一个基本宏，用于演示 OPACS/Xo 引擎
- exN03Vis6.mac:
  一个基本宏，用于演示 OpenInventor 引擎
- exN03Vis7.mac:
  一个基本宏，用于演示 VRMLFILE 引擎
- exN03Vis8.mac:
  一个宏，用于演示使用 DAWNFILE 引擎生成 PostScript 文件的"成批"可视化
- exN03Vis9.mac:
  一个宏，用于演示使用 DAWNFILE 引擎建立一个"多页" PostScript 文件
- exN03Tree0.mac:
  一个宏，用于演示 ASCII tree.
- exN03Tree1.mac:
  一个宏，用于演示 GAG tree.

8.7.12 其它

一些高级的高级的和与可视化引擎相关的主题在 8.10 节中论述。

8.8 非交互式可视化

当一个 G4 模拟程序正在运行的时候，可以不需要用户得干预进行可视化。可以从用户 action 类(例如，*G4UserRunAction* 和 *G4UserEventAction*)的方法中调用 Visualization Manager 的方法，来实现可视化。在本节中将论述 *G4VVisManager* 类的方法，并给出使用这些方法的例子，这个 *G4VVisManager* 类是 `intercoms` 模块的一部分。

8.8.1 *G4VVisManager* 类

Visualization Manager 是由类 *G4VisManager* 和 *MyVisManager* 实现的。参看 8.5 节 **"生成一个可视化的可执行程序"**。为了是用户的 G4 应用程序在存在可视化模块和不存在可视化模块的情况下都能进行编译，用户不应在 `main()`函数之后的 C++代码中直接使用这些类。相反，用户因该使用在 `intercoms` 模块中定义的它们的抽像基类 *G4VVisManager*。

指向类 *MyVisManager* 的具体实例的指针，就是具体的 Visualization Manager，可以用下面的方法获取：

```
//----- Getting a pointer to the concrete Visualization Manager instance
G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
```

如果 G4 可视化没有就绪，那么方法 `G4VVisManager::GetConcreteInstance()` 返回 `NULL` 。因而用户的 C++ 源码应该作如下保护：

```
//----- How to protect your C++ source codes in visualization
if (pVVisManager) {
    ....
    pVVisManager ->Draw (...);
    ....
}
```

8.8.2 探测器元件的可视化

如果用户已经使用逻辑体构造了探测器元件，它们的可视化属性也已经指定，用户进行探测器元件可视化的工作就基本就绪了。剩下的工作就是用 `ApplyCommand()`方法，在自己的 C++ 代码中描述适当的可视化命令。

例如，下面是用于可视化探测器元件的 C++ 源码：

```
//----- 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
```

在以上例子中，用户也应在合适的位置描述`/vis/open` 命令，或者在程序运行的时候从(G)UI 执行这个命令。

8.8.3 径迹可视化

为了可视化径迹，用户可以使用在 tracking 模块中定义的方法 `void G4Trajectory::DrawTrajectory()` 。在这个方法的实现中，使用了下面这个 *G4VVisManager* 的绘图方法：

```
   //----- A drawing method of G4Polyline
   virtual void G4VVisManager::Draw (const G4Polyline&, ...) ;
```

这个方法具体的实现在类 *G4VisManager* 中描述。

在一个事件结束的时候，一系列的径迹可以被作为一个 *G4Trajectory* 对像的列表进行存储。因此，用户可以在每个事件结束的时候可视化这些径迹，例如，可以通过实现如下的 `MyEventAction::EndOfEventAction()`方法来进行可视化：

```
//----- 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; iGetTrajectoryContainer()))[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
```

8.8.4 hits 的可视化

G4 使用类 *G4Square* 或 *G4Circle*,，或者其它用户自定义的 *G4VMarker* 的派生类，来对 Hits 进行可视化。缺省情况，不支持用于 hits 的 drawing 方法，需要按照抽像基类 *G4VHit* 和 *G4VHitsCollection* 的虚方法 `G4VHit::Draw()`和 `G4VHitsCollection::DrawAllHits()`来实现。这些方法在 `digits+hits` 模块中定义为空函数 。用户可以使用下面的类 *G4VVisManager* 的 drawing 方法来重载这些方法，以实现 hits 的可视化：

```
//----- Drawing methods of G4Square and G4Circle
virtual void G4VVisManager::Draw (const G4Circle&, ...) ;
virtual void G4VVisManager::Draw (const G4Square&, ...) ;
```

在类 *G4VisManager* 中，描述了 `Draw()`方法的具体实现。

例如，由继承 *G4VHit* 类的 *MyTrackerHits* 实现 `G4VHits::Draw()`的重载：

```
//----- 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
```

例如，由继承 *G4VHitsCollection* 类的 *MyTrackerHitsCollection* 实现 `G4VHitsCollection::DrawAllHits()` 的重载：

```
//----- 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
```

例如，用户通过实现下面的 `MyEventAction::EndOfEventAction()`方法，就可以在每个事件结束的时候对 hits 和 trajectories 进行可视化了：

```
void MyEventAction::EndOfEventAction()
{
  const G4Event* evt = fpEventManager->get_const_currentEvent();

  G4SDManager * SDman = G4SDManager::get_SDMpointer();
  G4String colNam;
  G4int trackerCollID = SDman->get_collectionID(colNam="TrackerCollection");
  G4int calorimeterCollID = SDman->get_collectionID(colNam="CalCollection");

  G4TrajectoryContainer * trajectoryContainer = evt->get_trajectoryContainer();
  G4int n_trajectories = 0;
  if(trajectoryContainer)
  { n_trajectories = trajectoryContainer->entries(); }
```

```
  G4HCofThisEvent * HCE = evt->get_HCofThisEvent();
  G4int n_hitCollection = 0;
  if(HCE)
  { n_hitCollection = HCE->get_capacity(); }

  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->get_trajectoryContainer()))[i]->DrawTrajectory();
    }

    // Construct 3D data for hits
    MyTrackerHitsCollection* THC
      = (MyTrackerHitsCollection*)(HCE->get_HC(trackerCollID));
    if(THC) THC->DrawAllHits();
    MyCalorimeterHitsCollection* CHC
      = (MyCalorimeterHitsCollection*)(HCE->get_HC(calorimeterCollID));
    if(CHC) CHC->DrawAllHits();

    // Declare end of visualization
    G4UImanager::GetUIpointer()->ApplyCommand("/vis/viewer/update");

  }

}

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

用户可以在探测到 hit 的地方，用高亮度的色彩，对相应的那个物理体进行重新可视化，而不需要对整个探测器元件集进行可视化。通过直接调用一个物理体的 drawing 方法来实现这个目的。这个方法如下：

```
//----- Drawing methods of a physical volume
virtual void Draw (const G4VPhysicalVolume&, ...) ;
```

这个方法在用户的 MyXXXHit::Draw()方法中被调用，后者使用 markers 描述 hits 的可视化。下面是这个 MyXXXHit::Draw()方法的例子：

```
//----- 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
```

8.8.5 文字的可视化

在 Geant4 可视化中，文字，也就是字符串，是通过类 *G4Text* 进行描述的。*G4Text* 像 *G4Square* 和 *G4Circle* 一样，继承了 *G4VMarker* 。因此，实现文字可视化的方法与实现 hits 可视化相同。对应的 *G4VVisManager* 的 drawing 方法是：

```
//----- Drawing methods of G4Text
virtual void G4VVisManager::Draw (const G4Text&, ...);
```

在类 *G4VisManager* 中描述了这个方法的具体实现。

8.8.6 折线和 tracking steps 的可视化

折线就是一些连续线段的集合，它们使用类 *G4Polyline* 进行描述。类 *G4VVisManager* 提供了供 *G4Polyline* 使用的的 darwing 方法：

```
 //----- A drawing method of G4Polyline
```

```
virtual void G4VVisManager::Draw (const G4Polyline&, ...) ;
```

在类 *G4VisManager* 中描述了这个方法的具体实现。

使用这个方法对 *G4Polyline* 进行可视化的 C++源码如下：

```
//----- 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
```

基于上述 *G4Polyline* 的可视化，可以实现对 tracking steps 的可视化。用户可以在每个 step，通过继承 *G4UserSteppingAction*，实现一个合适的 *MySteppingAction* 类，同时在 Run Manager 的帮助下，自动对 tracking step 进行可视化。

首先，用户必须实现一个方法 `MySteppingAction::UserSteppingAction()`。这个方法的典型实现如下：

```
//----- 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
```

接着，为了使上面的 C++代码正常工作，用户必须在 `main()`函数中给 Run Manager 传递的 *MySteppingAction* 信息 ：

```
//----- 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->set_userAction(new MySteppingAction);
   ...
}

//----- end of C++ source code
```

从此，用户就可以使用各种可视化属性在每一步(step)自动对 tracking steps 进行可视化了，例如，颜色。

与 tracking 一样，通过使用类 *G4Polyline* 和它在类 *G4VVisManager* 中定义的 drawing 方法，用户可以对任何由线段组成的 3D 对像，进行可视化。参看例子，命令`/vis/scene/add/axes`的实现。

Next section
Back to contents
Overview Contents Previous Next Geant4 User's Guide

8.9 内建可视化命令

Geant4 有各种内建的可视化命令。文件 `geant4/source/visualization/README.built_in_commands` 中使最新的有关内建的概要和它们的使用方法。

Next
Back to contents
Overview Contents Previous Next

8.10 其它

8.10.1 远程可视化

G4 是可以考虑用来进行远程可视化的，可以在 Geant4 主机以外的机器上进行可视化。有些可视化引擎支持这个特性。

通常，进行可视化的主机是用户的本地主机，而运行 G4 的主机是一台远程主机，例如是用 `telnet` 命令登录的主机。这使得进行分布式处理 G4 可视化，避免通过网络传送大量的可视化数据到用户的显示终端。

下面，假定用户在本地主机进行 G4 可视化，而 G4 进程运行在一台远程主机上。

使用 VRML-Network 引擎进行远程可视化

这节描述如何使用 VRML-Network 引擎进行 G4 远程可视化。

为了使用 VRML-Network 引擎进行远程可视化，用户必须事先在本地主机安装下列软件：

1. 一个 VRML viewer
2. Java 应用程序 `g4vrmlview`.

对应第 2 项的 Java 应用程序 `g4vrmlview` 是 G4 工具包的一部分，放在下面的位置：

source/visualization/VRML/g4vrmlview/

g4vrmlview 的安装指导在同一目录下的 README 文件中，也可以在下面的网页上找到。在以下的讨论中，假定用户已经正确安装了这个软件。

下列步骤实现了用本地 VRML browser 对远程 Geant4 可视化进行显示：

1. 使用一个 VRML viewer 名作为参数，在本地主机调用 g4vrmlview：

   ```
   Local_Host> java g4vrmlview  VRML_viewer_name
   ```

   例如，如果用户希望使用 Netscape 流览器作为用户的 VRML viewer，那么应用下面的方式执行 g4vrmlview：

   ```
   Local_Host> java g4vrmlview  netscape
   ```

   当然，应该正确设置 VRML viewer 的命令路径。

2. 登录运行 Geant4 可执行程序的远程主机。
3. 在远程主机按如下方式设置一个环境变量：

   ```
   Remote_Host> setenv G4VRML_HOST_NAME local_host_name
   ```

   例如，如果用户在一台名为"arkoop.kek.jp"的本地主机工作，那么设置这个环境变量如下：

   ```
   Remote_Host> setenv G4VRML_HOST_NAME arkoop.kek.jp
   ```

   这个设置告诉运行在远程主机的 Geant4 进程，将在什么地方进行 G4 可视化，也就是在什么地方显示可视化结果。

4. 通过 VRML-Network 引擎调用一个 Geant4 进程并进行可视化，例如 ：
5.
6. ```
   Idle> /vis/open VRML2
   ```
7. ```
   Idle> /vis/drawVolume
   ```
8. ```
   Idle> /vis/viewer/update
   ```

   执行第 4 步的时候， 3D scene 数据从远程主机 ，使用 VRML 格式发送到本地主机，接着，g4vrmlview 进程调用在第 3 步中指定的 VRML viewer 对这些 VRML 数据进行可视化。这些被传送的 VRML 数据被保存在本地主机的当前目录下，文件名为 g4.wrl。

   更多信息：

   - http://geant4.kek.jp/~tanaka/GEANT4/VRML_net_driver.html
   ```

使用 DAWN-Network 引擎的远程可视化

本小节将描述如何使用 DAWN-Network 引擎进行远程 Geant4 可视化。为了实现这个目标，用户必须先在本地主机安装 Fukui Renderer DAWN。更多信息请看 8.6 节 可视化引擎 。

下列这些步骤实现了用 DAWN 对远程 Geant4 可视化的显示。

- 在本地主机使用"-G"选项调用 DAWN：

  ```
  Local_Host> dawn -G
  ```

  这是使用网络联接模式调用 DAWN。

- 登录到运行 G4 可执行程序的远程主机。
- 在这台远程主机上按如下方式设置一个环境变量：

  ```
  Remote_Host> setenv G4DAWN_HOST_NAME local_host_name
  ```

  例如，如果用户在一台名叫"arkoop.kek.jp"的本地主机工作，那么设置这个环境变量如下：

  ```
  Remote_Host> setenv G4DAWN_HOST_NAME arkoop.kek.jp
  ```

  这个设置告诉在远程主机运行的 Geant4 进程，将在哪里进行 G4 可视化，也就是在哪里显示可视化的结果。

- 通过 DAWN-Network 引擎调用一个 Geant4 进程并进行可视化。例如：
-

  ```
  Idle> /vis/open DAWN
  Idle> /vis/drawVolume
  Idle> /vis/viewer/flush
  ```

  在执行步骤 4 的时候，3D scene 数据以 DAWN 格式从远程主机发送的本地主机，本地的 DAWN 将对这些数据进行可视化。这些被传送的数据被保存在本地主机的当前目录下，文件名为 `g4.prim`。

  更多信息：

  - http://geant4.kek.jp/~tanaka/DAWN/About_DAWN.html
  - http://geant4.kek.jp/~tanaka/DAWN/G4PRIM_FORMAT_24/

8.10.2 一个探测器几何树的可视化

更多信息：

- http://geant4.kek.jp/GEANT4/vis/GEANT4/GENOVA/visgenova.htm

---

8.10.3 对于进行 G4 可视化演示的一些提示

用户可能会在一些场合演示 Geant4 可视化，如，讨论会，会议。对应一个演示，需要注意下面的一些要求：

快的响应

用户应该使用一台速度快一点的机器，它的显卡带有 OpenGL 命令的硬件加速。接着就是，使用 OpenGL，OpenInventor 等引擎进行 G4 可视化，这样可以获得更快的速度。

Views 的交互式操作

OpenGL-Motif 引擎，OpenInventor 引擎，和 OPACS 引擎都支持它们自己的图形用户接口(GUI)用于 views 的交互式操作。许多 VRML viewers 也支持这个特性。

增强效果

例如，一些 OpenInventor viewers 支持"stereoscopic" 效果，在这种效果下，当观察者戴上特殊的眼镜时，可以体验生动的 3D 景象。用户还可以使用 PostScript viewer 的功能，对一个生成的矢量化 PostScript 图像的一小部分进行放大，以清晰的显示细节，这是非常有效的。

离线演示 G4 可视化可能是不错的选择。用户可以事先将那些已经可视化了的 views 保存为 OpenInventor 文件， VRML 文件， PostScript 文件，等，然后，在演示的时候使用适当的 viewers 将这些文件进行可视化。

About the authors

9. 例子

---

1. 入门例子
2. 高级例子

9.1 入门例子

Geant4 工具包包含了几个例子，它们演示了建立一个自定义的模拟程序所需要的类的实现。提供了 6 个"入门(novice)"例子，范围覆盖了从一个非交互式粒子和一个简单探测器的模拟，到在一个复杂探测器中电磁和强相互作用过程的模拟。用户可以基于这些粒子，进一步开发更具体的应用程序。一些“扩展(extended)”例子实现了实际高能物理探测器的模拟，这些例子需要使用一些除 G4 库以外的库文件。"高级(advanced)" 例子覆盖的情况对 G4 工具包本身的开发非常有帮助。

这些例子可以不作任何修改就进行编译并运行。它们中的大多数都可以运行在交互和批处理这两种模式下，其中批处理模式使用输入宏文件(`*.in`)，并提供参考输出文件(`*.out`)。这些例子通常作为 G4 工具集官方版本的校验或测试部分运行。

9.1.1 入门例子概述

这里提供了 6 个入门例子的简单描述和它们源码的链接。

`ExampleN01` (在下面的描述)

- 必要的用户类
- 演示 G4 内核如何工作

`ExampleN02` (在下面的描述)

- 在均匀磁场中的简单 tracker 几何
- 电磁作用过程

`ExampleN03` (在下面的描述)

- 简单量热器几何
- 电磁作用过程
- 各种材料

`ExampleN04` (在下面的描述)

- 使用一个读出几何的简单对撞机（collider）探测器
- 所有 ``ordinary'' 过程
- PYTHIA 初级事件
- 通过堆栈进行事件过滤

`ExampleN05` (在下面的描述)

- 简单 BaBar 量热器
- 电磁簇射的参数化

`ExampleN06` (在下面的描述)

- 光学光子过程

Tables 9.1.1 和 9.1.2 显示了入门级例子的``item charts''。

	`ExampleN01`	`ExampleN02`	`ExampleN03`
comments	用于 geantino 输运的最小集合	固定靶径迹室几何	在量热器中的电磁簇射
Run	用于用户编码方式批处理的 `main()`	用于交互模式的 `main()`	用于交互模式的 `main()`
			SetCut and Process On/Off
Event	事件发生器选择 (parti	事件发生器选择 (p	事件发生器选择 (particleGun)

	cleGun)	articleGun)	在 *UserEventAction* 中的``end of event"简单分析
Tracking	在程序中设定冗余信息级别	选择次级例子	选择径迹
Geometry	几何定义 (CSG)	几何定义(包括参数化 volume)	几何定义 (包括副本)
		均匀磁场	均匀磁场
Hits/Digi	-	径迹室类型的 hits	量热器类型的 hits
PIIM	最小粒子集合	EM 例子集合	EM 粒子集合
	单元素材料	混合物和化合物	混合物和化合物
Physics	输运过程	电磁物理过程	电磁物理过程
Vis	-	探测器和径迹显示	探测器和径迹显示
		径迹室类型的 hits 显示	
(G)UI	-	GUI 选择	GUI 选择
Global	-	-	-

Table 9.1.1
例子 `N01`，`N02` 和 `N03` 的``item chart"

	`ExampleN04`	`ExampleN05`	`ExampleN06`
comments	简单对撞机几何	参数化簇射的例子	光学光子的例子
Run	用于交互模式的 `main()`	用于交互模式的 `main()`	用于交互模式的 `main()`
Event	事件发生器选择 (HEPEvtInterface)	事件发生器选择 (HEPEvtInterface)	事件发生器选择 (particleGun)
	堆栈控制		

Tracking	选择径迹	-	-
	选择次级粒子		
Geometry	几何定义 (包括参数化/副本)	用于簇射参数化的 Ghost volume	几何定义 (使用旋转的 BREP)
	非均匀磁场		
Hits/Digi	Tracker/calorimeter/counter types	用于簇射参数化的灵敏探测器	-
	读出几何		
PIIM	所有粒子集	EM 粒子集	EM 粒子集
	混合物和化合物	混合物和化合物	混合物和化合物
Physics	所有物理过程	参数化簇射过程	光学光子过程
Vis	探测器和 hit 显示	探测器和 hit 显示	-
	量热器类型 hits 的显示	-	-
(G)UI	定义用户命令	定义用户命令	定义用户命令
Global	-	-	随机数引擎

Table 9.1.2
例子 `N04`，`N05` 和 `N06` 的``item chart" 。

9.1.2 例子 N01

基本概念
用于 geantino 输运的最小集合

例子中的类

`main()` (源文件)

- 用户编码的批处理方式
- *G4RunManager* 的构造和删除

- 设置 *G4RunManager*, *G4EventManager* 和 *G4TrackingManager* 的冗余信息级别
- 构造和设置必要的用户类
- 用户编码的 `beamOn()`
- 用户编码的 UI 命令请求

ExN01DetectorConstruction

(头文件) (源文件)

- 从 G4VUserDetectorConstruction 派生
- 单元素材料的定义
- CSG 实体
- 不使用旋转的 *G4PVPlacement*

ExN01PhysicsList

(头文件 s) (源文件)

- 从 *G4VUserPhysicsList* 派生
- geantino 的定义
- 输运过程的指定

ExN01PrimaryGeneratorAction

(头文件) (源文件)

- 从 *G4VPrimaryGeneratorAction* 派生
- *G4ParticleGun* 的构造
- 使用 particle gun 产生初级事件

9.1.3 例子 N02

基本概念

探测器：固定靶类型
物理过程：电磁
Hits：径迹室类型的 hits

例子中的类

`main()` (源文件)

- 用于交互模式 (和使用宏文件的批处理模式)的 `main()`
- (G)UI session 和 *VisManager* 的构造
- 随机数引擎
- *G4RunManager* 的创建和删除
- 创建和设置必要的用户类

ExN02DetectorConstruction

(头文件) (源文件)

- 从 *G4VUserDetectorConstruction* 派生
- 单元素材料，混合物和化合物材料的定义
- CSG 实体
- 均匀磁场： 创建 *ExN02MagneticField*
- 物理体
  - 使用旋转和不是用旋转的 *G4Placement* volumes 。
  - 不使用旋转的 *G4PVParameterised* volumes

ExN02MagneticField

(头文件) (源文件)

- 从 *G4MagneticField* 派生
- 均匀磁场. *ExN02MagneticField*

ExN02PhysicsList

(头文件) (源文件)

- 从 *G4VUserPhysicsList* 派生
- 定义 geantinos，电子，正电子，gammas
- 使用输运过程和“标准”电磁相互作用过程
- 交互特性：交互的选择物理过程(=> messenger 类)

ExN02PrimaryGeneratorAction

(头文件) (源文件)

- 从 *G4VPrimaryGeneratorAction* 派生
- 构造 *G4ParticleGun*
- 使用 particle gun 产生初级事件

xN02RunAction

(头文件) (源文件)

- 从 *G4VUserRunAction* 派生
- 显示探测器

ExN02EventAction

(头文件) (源文件)

- 从 *G4VUserEventAction* 派生
- 打印事件信息

ExN02TrackerSD

(头文件) (源文件)

- 从 *G4VSensitiveDetector* 派生
- 产生径迹室类型的 hit

ExN02TrackerHit

(头文件) (源文件)

- 从 *G4VHit* 派生
- 显示 hit 点

ExN02VisManager

(头文件) (源文件)

- 从 *G4VisManager* 派生
- 实现虚方法的 Visualization Manager 实例

9.1.4 例子 N03

基本概念

可视化电磁相互作用过程
交互特性：建立 messenger 类
Gun：随机发射例子。
Tracking：收集能量沉积，总径迹长度

例子中的类

`main()` (源文件)

- 用于交互式和使用宏文件的批处理模式的 *`main()`*
- *G4RunManager* 的创建和删除
- 创建和设置必要的用户类
- 几何体的自动初始化，和通过一个宏文件进行可视化

ExN03DetectorConstruction

(头文件) (源文件)

- 从 *G4VUserDetectorConstruction* 派生
- 单元素材料和混合物的定义
- CSG 实体
- 不使用旋转的 *G4PVPlacement*
- 交互特性：改变探测器尺寸，材料，磁场。 (=>messenger 类)
- 可视化

ExN03PhysicsList

(头文件) (源文件)

- 从 *G4VUserPhysicsList* 派生
- 定义 geantinos, gamma，轻子，轻介子，重子和离子
- 输运过程，标准电磁过程，衰变
- 交互特性：*SetCut*, process on/off. (=> messenger 类)

ExN03PrimaryGeneratorAction

(头文件) (源文件)

- 从 *G4VPrimaryGeneratorAction* 派生
- 构造 *G4ParticleGun*
- 使用 particle gun 产生初级事件
- 交互特性：随机发射粒子。 (=> messenger 类)

ExN03RunAction

(头文件) (源文件)

- 从 *G4VUserRunAction* 派生
- 显示探测器 and tracks
- 交互特性： *SetCut*, process on/off.
- 交互特性：改变探测器尺寸，材料，磁场。

ExN03EventAction

(头文件) (源文件)

- 从 *G4VUserEventAction* 派生
- 存储径迹
- 打印事件结束信息 (能量沉积，等)

ExN03SteppingAction

(头文件) (源文件)

- 从 *G4VUserSteppingAction* 派生
- 收据能量沉积，等信息。

ExN03VisManager

(头文件) (源文件)

- 从 *G4VisManager* 派生
- 实现虚方法的 *Visualization Manager* 实例

9.1.5 例子 N04

基本概念

简单对撞机实验几何

Full hits/digits/trigger

例子中的类

`main()` (源文件)

- *ExN04RunManager* 的创建和删除
- 创建 (G)UI session 和 *VisManager*
- 创建和设置 user 类

ExN04DetectorConstruction

(头文件) (源文件)

- 从 *G4VUserDetectorConstruction* 派生
- 创建 *ExN04MagneticField*
- 定义混合物和化合物材料
- 与材料相关的截断
- 使用参数化/副本的简单对撞机几何
- 径迹室/muon –参数化
- 量热器—副本

ExN04TrackerParametrisation

(头文件) (源文件)

- 从 *G4VPVParametrisation* 派生
- 参数化尺寸

ExN04CalorimeterParametrisation

(头文件) (源文件)

- 从 *G4VPVParametrisation* 派生
- 参数化 位置/旋转

ExN04MagneticField

(头文件) (源文件)

- 从 *G4MagneticField* 派生
- 螺线管和螺线管形的场

ExN04TrackerSD

(头文件) (源文件)

- 从 *G4VSensitiveDetector* 派生
- 产生径迹室类型的 hit

ExN04TrackerHit

(头文件) (源文件)

- 从 *G4VHit* 派生
- 显示 hit 点

ExN04CalorimeterSD

(头文件) (源文件)

- 从 *G4VSensitiveDetector* 派生
- 产生量热器类型的 hit

ExN04CalorimeterHit

(头文件) (源文件)

- 从 *G4VHit* 派生
- 使用可变的颜色绘制物理体

ExN04MuonSD

(头文件) (源文件)

- 从 *G4VSensitiveDetector* 派生
- 产生闪烁体类型的 hit

ExN04MuonHit

(头文件) (源文件)

- 从 *G4VHit* 派生
- 使用可变的颜色绘制物理体

ExN04PhysicsList

(头文件) (源文件)

- 从 *G4VUserPhysicsList* 派生
- 定义所有粒子
- 指定所有物理过程

ExN04PrimaryGeneratorAction

(头文件) (源文件)

- 从 *G4VPrimaryGeneratorAction* 派生
- 构造 *G4HEPEvtInterface*
- 使用 PYTHIA event 产生初级事件

ExN04EventAction

(头文件) (源文件)

- 存储初始种子

ExN04StackingAction

(头文件) (源文件)

- 从 *G4UserStackingAction* 派生
- ``stage'' 控制和优先级控制
- 事件中止

ExN04StackingActionMessenger

(头文件) (源文件)

- 从 *G4UImessenger* 派生
- 定义中止条件

ExN04TrackingAction

(头文件) (源文件)

- 从 *G4UserTrackingAction* 派生
- 选择径迹
- 选择次级粒子

9.1.6 Example N05

基本概念

参数化簇射的使用：
* 定义一个电磁簇射模型
* 指定一个逻辑体
* (当就绪的时候定义一个 ghost volume)
交互特性：建立 messengers 类
Hits/Digi：filled from detailed and parameterised simulation (calorimeter type hits ?)

例子中的类

`main()` (源文件)

- 用于交互模式的 `main()`
- *G4RunManager* 的创建和删除
- 创建和设置必要的用户类
- 创建 *G4GlobalFastSimulationmanager*
- 创建一个 *G4FastSimulationManager* 用于给一个逻辑体(envelope)指定一个快速模拟模型
- (定义用于参数化的 ghost volume)
- 创建电磁簇射快速模拟模型

ExN05EMShowerModel

(头文件) (源文件)

- 从 *G4VFastSimulationModel* 派生
- 在灵敏探测器内的能量沉积

ExN05PionShowerModel

(头文件) (源文件)

- 从 *G4VFastSimulationModel* 派生
- 在灵敏探测器内的能量沉积

ExN05DetectorConstruction

(头文件) (源文件)

- 从 G4VUserDetectorConstruction 派生
- 定义单元素材料和混合物
- CSG 实体
- *G4PVPlacement*

ExN05PhysicsList

(头文件) (源文件)

- 从 *G4VUserPhysicsList* 派生
- 指定 *G4FastSimulationManagerProcess*

ExN05PrimaryGeneratorAction

(头文件) (源文件)

- 从 *G4VPrimaryGeneratorAction* 派生
- 构造 *G4ParticleGun*
- 使用 particle gun 产生初级事件

ExN05RunAction

(头文件) (源文件)

- 从 *G4VUserRunAction* 派生
- 显示探测器
- (参数化的激活/关闭?)

ExN05EventAction

(头文件) (源文件)

- 从 *G4VUserEventAction* 派生
- 打印时间信息

9.1.7 例子 N06

基本概念

交互特性：建立 messenger 类。
Event ：Gun，向切伦科夫辐射体和闪烁体发射带电粒子。
PIIM：使用光学和闪烁特性的材料/混合物。
几何体：用光学材料填充，拥有表面属性的几何体。
物理过程：定义和初始化光学过程。
Tracking：产生切伦科夫辐射，收集沉积的能量用于产生闪烁。
Hits/Digi ：用作探测器的 PMT。
可视化：几何体，光学光子径迹。

例子中的类

`main()` (源文件)

- 用于交互式和使用宏文件的批处理模式的 *`main()`*
- 随机数引擎
- *G4RunManager* 的创建和删除
- 创建和设置必要的用户类
- 用户编码的 `beamOn`

ExN06DetectorConstruction

(头文件) (源文件)

- 从 *G4VUserDetectorConstruction* 派生
- 单元素材料和混合物的定义
- 生成材料属性表(Material Properties Table)，并给材料添加
- CSG 和 BREP 实体
- 使用旋转的 *G4PVPlacement*
- 表面(surfaces)定义
- 生成材料属性表(Material Properties Table)，并给表面添加
- 可视化

ExN06PhysicsList

(头文件) (源文件)

- 从 G4VUserPhysicsList 派生
- 定义 gamma，轻子和光学光子
- 输运过程，“标准”电磁过程，衰减，切伦科夫效应，闪烁效应，“标准”光学和边界过程
- 修改/增大 光学过程参数

ExN06PrimaryGeneratorAction

(头文件) (源文件)

- 从 *G4VPrimaryGeneratorAction* 派生
- 构造 *G4ParticleGun*
- 使用 particle gun 产生初级事件

xN06RunAction

(头文件) (源文件)

- 从 *G4VUserRunAction* 派生
- 绘制探测器

9.2 高级例子

9.2.1 高级例子

Geant4 高级例子举例说明了在一些典型实验环境中，G4 的实际应用。它们中大多数显示了分析工具(如 histograms, ntuples 和 plotting)，各种可视化特性和高级用户接口工具的使用。可以在这里预览这些高级例子的共同特性。
这些高级例子包括：

- **浅部治疗**，举例说明了一个典型的医用物理学应用
- **gamma 射线望远镜**，举例说明了一个灵活配置的典型的 gamma 射线望远镜的运用
- **x 射线望远镜**，举例说明了一个在典型的 X 射线望远镜中，背景辐射研究的应用
- **x 射线荧光**，举例说明了 X 射线荧光和 PIXE（质子诱发 X 射线）的发射
- **地下物理**，举例说明一个用于暗物质寻找的地下探测器

其它高级例子包含在 Geant4 医用扩展例子内：

- DICOM，举例说明与 DICOM 图形格式接口的 G4 接口的初级版本。这个例子是由 G4 用户开发的，他们是：Louis Archambault, Luc Beaulieu, Vincent Hubert-Tremblay (Centre Hospitalier Universitaire de Quebec (CHUQ), Université Laval, Québec (QC) Canada).

更多关于在这些例子中使用的分析工具的文档在： AIDA (Abstract Interfaces for Data Analysis) 和 Anaphe/Lizard， JAS 和 OpenScientist.

这些高级例子与低能电磁物理工作组联合开发的。

10. 附录

10.1 Geant4 程序编译提示

本节举例说明并验证在 G4 工具包编译过程中缺省使用的一些选项。同时也是用户的安装指南，用来避免和更正可能发生在一些编译器上的问题。这里的解决办法是移植 Geant4 代码过程中获得的经验，是针对一些特定的操作系统和编译器版本的。

众所周知，每个编译器都采用自己的内部技术来生成目标代码，它们根据与系统体系结构有关的几个因素，对目标代码进行一些优化。目前 C++编译器的一个特性是，在编译/链接期间对模板实例进行处理。一些 C++编译器需要存储临时模板实例文件(目标文件或者临时源码文件)，这些文件被保存在一个"模板库"，或者可以来自编译命令的唯一确定的目录，也可以不是直接来自编译命令 (可能来自 C++编译器的旧的基于 C 语言前端的实现)【注：最初的 C++实现，是通过预编译将 C++代码转换为 C，然后由 C 编译器进行编译的】。

在 Geant4 中，向模板库的路径由环境变量$G4TREP 指定，它是固定的，缺省指向 `$G4WORKDIR/tmp/$G4SYSTEM/g4.ptrepository/`，这里的`$G4SYSTEM` 是当前使用的 系统结构/编译器 ，`$G4WORKDIR` 是 G4 的用户工作目录。

缺省情况建立一个副模板库`$G4TREP/exec` ，当生成可执行文件时发生主模板库类名（建立G4库时需要使用这个主模板库）这个冲突，可以使用这个副模板库隔离主模板库。可以使用环境变量(或者在与将要编译的 test/example 的 GNUmakefile 文件中)`G4EXEC_BUILD` 激活这个副模板库；一旦被激活，这个副模板库将变为可读写模板库，而主模板库则成为只读模板库。

在这些 G4 库被安装之后，强烈推荐将安装目录($G4INSTALL)和工作目录($G4WORKDIR)分开，以便对模板库的安装区域不作更改。

使用一个模板库有一个好的建议，就是在建立 G4 库的时候使用一个 single 模板库 (通过环境变量`$G4TREP` 指定)；然后当编译任何例子或应用程序时，使用一个副模板库 (`$G4TREP/exec`, 与`$G4EXEC_BUILD` 标志一起确定)。
不时的清理副模板库是个很好的习惯。

10.1.3 Sun

操作系统: SunOS
编译器 r: CC

缺省优化标准是-O2。这个编译器使用一个模板库来处理模板实例，因此，请注意上面提出的建议。
自从 5.0 版的编译器开始，支持本地 STL，并要求与 ISO/ANSI 兼容。

10.1.4 Unix/Linux - g++

操作系统: Linux
编译器: GNU/gcc

对应 2.95.2 或者更高版本的 gcc 编译器，需要使用严格的 ISO/ANSI 兼容性 (使用-ansi -pedantic 编译器标志)，同时代码使用高冗余诊断进行编译(-Wall 标志)。

10.1.5 PC - MS Visual C++

操作系统: MS/Windows
编译器: MS-VC++

参看 Installation Guide 第 3 节。

About the authors
Overview Contents Previous Next
Geant4 User's Guide
For Application Developers
Appendix

10.2 数据分析接口

Geant4 是独立于任何数据分析工具包。G4 工具包没有任何用于数据分析工具包的引擎，并且在 G4 中使用数据分析工具包也不需要任何引擎。进行数据分析的代码应与用于数据分析[1]的抽像接口 AIDA 一起编译。

因此，用户可以使用自己喜欢的数据分析。

10.2.1 JAS

使用 JAVA Analysis Studio tool [2]，请参考有关数据分析的 JAS 文档。

10.2.2 Lizard

使用 Lizard AIDA Interactive Analysis Environment [3]，请参考有关数据分析的 Lizard 文档。

10.2.3 Open Scientist Lab

使用 Lab Analysis plug-in for the OnX package [4]，请参考有关数据分析的 Open Scientist Lab 文档。

10.2.4 例子

在 Geant4 中的例子显示了如何使用用于数据分析的 AIDA 兼容工具，这些例子在目录 `geant4/examples/extended/analysis` 和 `geant4/examples/advanced` 中。

---

[1] http://aida.freehep.org

[2] http://jas.freehep.org/documentation.htm

[3] http://cern.ch/anaphe/Lizard/documentation.html

[4] http://www.lal.in2p3.fr/OpenScientist

About the authors

Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Appendix

10.3 CLHEP 基本类库

CLHEP [1] 代表用于高能物理(HEP)的类库，它包含了许多专用于物理和高能物理的基本类。

详情查看 CLHEP Reference Guide [2]和 User Guide [3]。

CLHEP 的起源和目前状况

CLHEP 是开始于 1992 年的一个库，实际上源自于用 C++写的 MC 事件发生器 MC++，它包含了 MC++常用的基本类。从那时起，许多作者将一些类添加到这个库中，有些部分是在 Geant4 合作项目中的开发人员完成的。

Geant4 和 CLHEP

Geant4 项目对正在进行的 CLHEP 开发作了很多贡献。随机数包，物理单位和常数，以及一些数值和几何类是源自于 Geant4 的。

Geant4 同样得益于 CLHEP 的开发。除了已经提到的用于随机数和数值的类外，还有一些类，它们用于点，矢量，和平面以及它们在 3D 空间中的变换，和洛伦茨矢量及变换。不过，在 G4 中这些类有自己的名字，它们只是使用 typedefs 将 CLHEP 类重新定义了以下，例如 G4ThreeVector。

---

[1] cern.ch/clhep

[2] cern.ch/clhep/manual/RefGuide

[3] cern.ch/clhep/manual/UserGuide

About the authors

10.4 C++ 标准模板库

概述

标准模板库(STL)使用一个包含了通用算法和数据结构的通用目的库。它是 C++标准库的一部分。目前，大多数编译器提供商在它们的产品中都包括一个 STL，同样也有一些商业版本可供使用。

关于 STL 的好书【中文书可以看候捷翻译的一系列有关 STL 的书，部分书在网上有 pdf】

Nicolai M. Josuttis: The C++ Standard Library. A Tutorial and Reference, Addison-Wesley, 1999, ISBN 0-201-37926-0.

David R. Musser, Atul Saini: STL Tutorial and Reference Guide / C++ Programming with the Standard Template Library, Addison-Wesley, 1996, ISBN 0-201-63398-1.

可用在线资原包括 A Modest STL tutorial 和 SGI 实现参考：

- Mumit's STL Newbie Guide 是一个 FAQ，包含了 STL 编程的实用细节。
- SGI STL homepage，这是本地 egcs STL 实现的基础。

G4 中的 STL

从 0.1 版开始，Geant4 就支持 STL。从 1.0 版开始，使用 G4 要求必须有 STL。
可以预见 G4 支持的所有平台上的 STL *本地* 实现

About the authors

Overview Contents Previous Next

Geant4 User's Guide
For Application Developers
Appendix

10.5 Makefiles 和 Geant4 环境变量

本节描述在 G4 中如何实现 GNUmake 框架，并向用户和安装人员提供了一些最重要环境变量的快速参考。

10.5.1 在 G4 中的 GNUmake 系统

就如本手册 2.7.1.1 节中描述的，主要由下列 GNUmake 脚本文件(放置在`$G4INSTALL/config`的`*.gmk` 脚本)控制了 G4 中的 GNUmake 处理：

- `architecture.gmk`：定义所有指定设置和路径的结构。系统设置被保存在`$G4INSTALL/config/sys`中的独立文件中。
- `common.gmk`：定义所有用于建立目标文件和库通用 GNUmake 规则。
- `globlib.gmk`：定义所有用于建立复合库的通用 GNUmake 规则。
- `binmake.gmk`：定义用于建立可执行程序的通用 GNUmake 规则。
- `GNUmake scripts`：被放置在 G4 的每个目录下，定义建立一个库(或者一组子库)或者可执行文件的专用指令。

为了建立一个库 (或者一组子库)或者一个可执行程序，用户必须显示的更改当前目录为用户希望的目录，从该目录调用"gmake"命令("gmake global" 用于建立一个复合库)。下面实一些基本命令或者说是 GNUmake "targets"，用户可以调用它们来建立库或可执行程序：

- gmake
  这个命令将开始一个编译过程，用于建立内核库或例子程序的库。内核库将是离散的，也就是如果一个模块是一个复合模块，那么这个命令将建立所有相关的子库，而不是复合库。在这种情况下，在$G4INSTALL/source 顶层的 GNUmakefile 也将为每个库建立一个相互关系映射文件 libname.map，以便在 bin 阶段自动建立链接顺序。这个映射文件将被放在$G4LIB/$G4SYSTEM
- gmake global
  它将开始一个编译过程，为每个模块建立单一的复合内核库。如果随后又执行 "gmake"，那么将同时安装'离散'库和'复合'库(注意：这将使用更多的磁盘空间。缺省情况，在链接的时候将使用复合库，除非指定了 G4LIB_USE_GRANULAR)。
- gmake bin 或者 gmake (仅用于 examples/)
  它将开始一个编译过程，建立一个可执行程序。这个命令将隐式的建立与例子相关的库，并且链接最终的可执行程序。假定所有的内核库已经生成，并且放置在 $G4INSTALL。
  在使用离散库的情况下，链接顺序是自动控制的，并且产生链接表。

lib/ bin/ 和 tmp/ 目录

环境变量$G4INSTALL 指定 G4 工具包的的安装路径，内核库将被放在$G4INSTALL/lib。用户设置环境变量$G4WORKDIR ，指定用户的工作目录；临时文件 (G4 安装过程中产生的目标文件和数据)将放置在$G4WORKDIR/tmp。二进制文件将放置在$G4WORKDIR/bin。 路径 $G4WORKDIR/bin/$G4SYSTEM 应添加到用户环境变量$PATH 中。

10.5.2 环境变量

这里是一个最重要环境变量的表和一些简单的使用说明，它们是在 GNUmake 框架中定义的。**建议不要重载或设置(显示的或者不小心)在这里列出、并用*标识的环境变量。它们已经被设置并在缺省的安装过程中被使用！**

- 系统设置

  $CLHEP_BASE_DIR
  指定在用户系统中 CLHEP 工具包的路径。

  $G4SYSTEM
  定义系统体系结构和当前使用的编译器。
  注意： 如果安装过程使用了 Configure 脚本，那么这个变量将被自动设置。这将产生一个正确的设置，用于使用 shell 脚本 env.[c]sh 进行环境设置。

- 安装路径

$G4INSTALL
定义 G4 工具包将要安装的路径。它应有系统安装程序进行设置。缺省情况，假定 G4 安装目录为$HOME，则设置为$HOME/geant4。

$G4BASE **(*)**
定义指向源代码的路径。用于为-I 和 -L 指令定义$CPPFLAGS 和$LDFLAGS 。它必须被设置为$G4INSTALL/src。

$G4WORKDIR
为 G4 的用户工作目录定义路径。假定用户的 G4 工作路径在$HOME 目录下，则缺省设置为$HOME/geant4。

$G4INCLUDE
定义头文件路径，这些头文件可以是使用 gmake includes 命令进行安装是产生的镜像(缺省设置为$G4INSTALL/include)。

$G4BIN,
$G4BINDIR **(*)**
系统用于指定可执行程序存储的位置。缺省情况分别设置为$G4WORKDIR/bin 和 $G4BIN/$G4SYSTEM 。路径$G4WORKDIR/bin/$G4SYSTEM 应被添加到用户环境的$PATH 中。用户可以重载$G4BIN 。

$G4TMP,
$G4TMPDIR **(*)**
系统用于指定临时文件存储的位置，这些文件是用户的应用程序或测试程序在编译过程中产生的。缺省情况分别设置为$G4WORKDIR/tmp 和$G4TMP/$G4SYSTEM 。用户可以重载$G4TMP 。

$G4LIB,
$G4LIBDIR **(*)**
系统用于指定库文件存储的位置。缺省情况分别设置为$G4INSTALL/lib 和 $G4LIB/$G4SYSTEM 。用户可以重载$G4LIB 。

- Build 指定

$G4TARGET
指定将要编译的应用程序/例子的目标(定义 main()函数的源文件名)。这个变量是为在 $G4INSTALL/examples 目录下的例子自动设定的。

$G4EXEC_BUILD
如果要使用一个副模板库，或者在编译用户应用程序/例子的时候要处理模板实例，那么需要指定这个标志。对于 G4 的内部模块测试，这个变量已经在相关的 GNUmakefile 中设置了。然而，对于在$G4INSTALL/examples 中的例子不需要使用这个标志，因为在例子中的类名已经被更改，并且是互不相同的。这个标志只用于那些使用模板库(参看本手册的附录 A.2)的编译器，副模板库被设置为$G4TREP/exec 。

$G4DEBUG
指定编译代码(库文件或者例子)的时候在目标代码中包含符号信息，以便用于调试。

生成的目标代码大小将显著增大。缺省情况下，代码是在优化模式下编译的 (设置 `$G4OPTIMISE`)。

$G4NO_OPTIMISE
指定不使用优化进行代码编译 (库文件或例子)。

$G4NO_STD_NAMESPACE **(*)**
在 G4 库中避免使用 *std* namespace。

$G4NO_STD_EXCEPTIONS **(*)**
在 G4 中避免丢弃异常。

$G4_NO_VERBOSE
缺省情况下(设置`$G4VERBOSE` 标志)，Geant4 代码是在高冗余模式下编译的。为了获得更好的性能，可以通过定义`$G4_NO_VERBOSE` 忽略冗余信息。

$G4LIB_BUILD_SHARED
如果要建立一个共享的内核库(缺省将使用这些库)，那么需要设定这个标志。如果不设置，缺省将建立一个静态文档库。

$G4LIB_BUILD_STATIC
如果除了共享库外(同时设置`$G4LIB_BUILD_SHARED`).，还要建立一个静态文档库，那么需要指定这个标志。

$G4LIB_USE_GRANULAR
如果在“离散”库和“复合”库都安装的情况下，要在链接的时候强制使用“离散”库，而不使用“复合”库，那么需要设置这个标志。如果安装的“复合”库，G4 将缺省选择“复合”库。

- UI 指定

这里只列出了打多数于用户接口引擎相关的标志。更多的细节描述在用户手册的第 2 章。

G4UI_USE_TERMINAL
在将要编译的应用程序中(缺省)，指定使用哑终端接口。

G4UI_BUILD_XM_SESSION, G4UI_BUILD_XAW_SESSION
指定在内核库中包含基于 *XM* 或 *XAW* Motif 的用户接口

G4UI_USE_XM, G4UI_USE_XAW
指定在将要编译的应用程序中使用 *XM* 或 *XAW* 接口。

G4UI_BUILD_WO_SESSION, G4UI_USE_WO
指定使用与 OPACS 工具相关的 *WO* 用户接口。

G4UI_BUILD_WIN32_SESSION
指定内核库包含用于 Windows 系统的 WIN32 终端接口。

G4UI_USE_WIN32
指定在将要编译的应用程序使用 WIN32 接口，这个程序将在 Window 平台上运行。

G4UI_NONE
如果设置这个标志，那么没有任何 UI sessions 和任何 UI 库被建立。当用户运行一个纯批处理任务时，或者在用户程序框架中有自己的 UI 系统时，可以设置这个标志。

- 可视化指定

这里只列出了与可视化引擎最相关的标志。更详细的描述在本手册的第 2 章。

$G4VIS_BUILD_OPENGLX_DRIVER
指定建立包含带 X11 扩展的 OpenGL 可视化引擎的内核库。要求设置`$OGLHOME` (OpenGL 的安装路径)。

$G4VIS_USE_OPENGLX
指定在将要建立的应用程序中使用带 X11 扩展的 OpenGL 引擎。

$G4VIS_BUILD_OPENGLXM_DRIVER
指定建立包含带 XM 扩展的 OpenGL 可视化引擎的内核库。要求设置`$OGLHOME`(OpenGL 安装路径)。

$G4VIS_USE_OPENGLXM
指定在将要建立的应用程序中使用带 XM 扩展的 OpenGL 引擎。

$G4VIS_BUILD_OI_DRIVER
指定建立包含 OpenInventor 可视化引擎的内核库。要求设置`$OIHOME` 和`$HEPVISDIR` (`OpenInventor/HepVis` 安装路径)。

$G4VIS_USE_OI
指定在将要建立的应用程序中使用 OpenInventor 可视化引擎。

$G4VIS_BUILD_OIX_DRIVER
指定建立用于 X11 版 OpenInventor 的引擎。

$G4VIS_USE_OIX
指定使用 X11 版 OpenInventor 的引擎。

$G4VIS_BUILD_OIWIN32_DRIVER
指定建立用于 X11 版 OpenInventor 的引擎，这个引擎将运行于 Windows 系统。

$G4VIS_USE_OIWIN32
指定使用 X11 版 OpenInventor 的引擎，这个引擎将运行于 Windows 系统。

$G4VIS_BUILD_OPACS_DRIVER
指定建立包含 OPACS 可视化引擎的内核库。要求设置$OPACSHOME(OPACS 的安装路径)。

$G4VIS_USE_OPACS
指定在将要建立的应用程序中使用 OpenInventor 引擎。

$G4VIS_BUILD_DAWN_DRIVER
指定建立包含 DAWN 可视化引擎的内核库。

$G4VIS_USE_DAWN
指定在将要建立的应用程序中使用 DAWN 引擎

$G4DAWN_HOST_NAME
为使用 DAWN-network 引擎指定主机名。

$G4VIS_NONE
如果设置了这个变量，将不建立任何可视化引擎。

- `g3tog4` 模块

  $G4LIB_BUILD_G3TOG4
  如果设置了这个变量，将触发 `g3tog4` 模块的编译，这个模块用于将一些简单的 G3 几何描述转换为 G4 描述。缺省情况下，不设置这个标志，不建立这个模块的库。设置这个标志将同时隐式的设置下面的标志。

  $G4USE_G3TOG4
  指定使用 `g3tog4` 模块，假定相关的库都已经被安装。

- `STEP` 模块

  $G4LIB_BUILD_STEP
  如果设置了这个标志，将触发 `STEP` reader 和 interface 模块的编译，这些模块用于从 CAD 系统导入简单的 STEP AP203 几何模型。缺省情况下，不设置这个标志，不建立这些模块的库。设置这个标志将同时隐式的设置下面的标志。
  注意：这些模块只有用于旧的非 ISO/ANSI 编译器时，才保证产生正确的结果。

  $G4USE_STEP
  指定使用这些 `STEP` 模块，假定相关的库都已经被安装。

- 数据分析工具指定

  $G4ANALYSIS_USE
  为应用程序指定激活用于数据分析的适当环境，该应用程序包含基于 *AIDA* 的数据分析代码。还要求设置其它的一些变量 (`$G4ANALYSIS_AIDA_CONFIG_CFLAGS`，`$G4ANALYSIS_AIDA_CONFIG_LIBS`)用于定义 AIDA 的配置选项("`aida-config --cflags`"和"`aida-config --libs`")。有关细节参看特定分析工具的安装指令。

- 物理数据的目录

  $NeutronHPCrossSections
  为中子散射过程设置的外部数据路径。

$G4LEDATA
为低能电磁过程设置的外部数据路径。

$G4LEVELGAMMADATA
为 Photon Evaporation 设置的数据路径。

$G4RADIOACTIVEDATA
为放射性衰变设置的数据路径。

10.5.3 G4 与外部库进行链接

Geant4 GNUmake 结构允许使用外部软件包(或者用户自定义)包，对库链接表进行扩展，这些软件包是产生最终可执行的用户应用程序所必须的。

10.5.3.1 添加*不*使用 G4 的外部库

在用户应用程序的 GNUmakefile 中，在包含 binmake.gmk 之前，使用-L...-l... 或者使用绝对路径名，在 EXTRALIBS 中指定附加库，例如：

```
EXTRALIBS := -L<your-path>/lib -l<myExtraLib>
```

或

```
EXTRALIBS := <your-path>/lib/lib<myExtraLib>.a
```

用户也可以指定 EXTRA_LINK_DEPENDENCIES，这个指定的值将变添加到目标可执行程序的依赖关系中，用户也可以为它指定编译规则，例如：

```
EXTRA_LINK_DEPENDENCIES := <your-path>/lib/lib<myExtraLib>.a

<your-path>/lib/lib<myExtraLib>.a:
     cd <your-path>/lib; $(MAKE)
```

注意，通常需要用户为外部库的头文件增加 CPPFLAGS ，例如：

```
CPPFLAGS+=-I<your-path>/include
```

参看 table 10.5.1.

```
# --------------------------------------------------------------------
# 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)
```

Table 10.5.1
一个自定义 GNUmakefile 的例子，用于使用外部模块的应用程序或例子，这些外部模块不被绑定到 Geant4。

10.5.3.2 添加使用 G4 的外部库

除了上述变量外，还要在 EXTRALIBSSOURCEDIRS 中指定一系列目录，这些目录的 src/子目录中包含了源文件。因此，用户的 GNUmakefile 可能包含：

```
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)
```

参看 Table 10.5.2.

```
# --------------------------------------------------------------
# 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)
```

Table 10.5.2
一个自定义 GNUmakefile 的例子，用于使用外部模块的应用程序或例子，这些外部模块被绑定到 Geant4。

10.6 使用 MS Visual C++ 编译 Geant4

可以使用 MS Visual Studio C++的 C++编译器和 Cygwin 工具集对 Geant4 进行编译。在安装手册中有详细的指南。与依赖 `make` 和其它 Unix 工具的编译系统一样，Makefiles 和 Geant4 环境变量中的内容同样适用与使用 MS Visual C++进行编译。

不支持在 MS Visual Studio 下的直接编译，也就是说，用户不需要提供 workspace 文件 (`.dsw`) 和工程文件 (`.dsp`).

不过，建立的可执行程序可以使用 MS Visual Studio 的调试工具进行调试。在第一次调试一个指定的可执行程序时，用户也许必须帮助调试程序查找源文件路径。

10.7 开发和调试工具

1. UNIX

虽然属于本用户手册的范围，但在本节中还是要提供一系列值得了解的参考材料、开发工具和环境，这应该对用户使用 C++进行开发是很有帮助的。这里只列出了相当有限的一个列表。

- 在 Linux 系统上的 KDevelop 开发环境 [1]。
- The GNU Data Display Debugger (DDD) [2].
- SUN Forte C++ environment (former Workshop) [3].
- Microsoft Visual Studio 开发环境 [4].
- Parasoft Insure++实时调试程序和内存检查工具 [5]
- Parasoft Code Wizard 源代码分析器[6].
- Rational Rose CASE 工具 [7].
- Together ControlCenter 开发环境[8].
- 用于软件分析测试的 Logiscope 工具[9].
- Rational Unified Process (RUP)软件工程工具[10]

---

[1] http://www.kdevelop.org

[2] http://www.gnu.org/software/ddd

[3] http://www.sun.com/forte/cplusplus

[4] http://msdn.microsoft.com/vstudio

[5] http://www.parasoft.com/products/insure

[6] http://www.parasoft.com/products/wizard

[7] http://www.rational.com/products/rose

[8] http://www.togethersoft.com/products/controlcenter

[9] http://www.telelogic.com/products/logiscope

[10] http://www.rational.com/products/rup

About the authors