Cookbook for Modeling Indirect Detection Mechanisms with Geant4 - P180F

Alina Kochocki

I. INTRODUCTION

A. Intention of Document

This text is designed as an introduction to the C++ toolkit, Geant4, a software and collection of libraries used primarily to model particle passage through matter. Specifically, this document assumes the reader has no prior experience with Geant4, and possibly limited exposure to C++. The purpose of this work is to introduce those topics of Geant4 which are necessary in modeling, but also highlight important or nuanced concepts of the detection physics.

The text itself is composed of a number of sections, each covering a different aspect of Geant4. These sections should be read in order, as they outline the considerations necessary to simulate a sensitive detector and retrieve important information about the outcome of particle interactions within the modeled environment. Certain topics will be communicated via an example listing which will be related to a sample simulation program referenced throughout this text. Ideally, the brief introduction to C++ presented below, these example listings, as well as the many materials online and the those installed with Geant4, should provide sufficient reference for a new programmer so that the concern is not on the structural aspects or usage of C++, but on producing powerful and well understood simulations with illustrative results.

B. Geant4 Background & History

GEANT, or, GEometry ANd Tracking, generally refers to a series of simulation software dedicated to modeling particle passage and interaction with matter. The first version was developed in 1974 at CERN, intended for use in high energy physics experiments. The original version was written in FORTRAN, a high-performance computing language. Today's modern version, Geant4, relies solely on C++ with an object-oriented design. It is still heavily used within the high energy community (e.g. ATLAS, CMS, DUNE, MINOS, T2K, and various dark matter searches), but also finds applications in space science, the medical field, radiation and nuclear studies.

II. C++ PRIMER

Geant4 is written in C++. While mastery of this language is not the focus or suggestion of this text, it is impossible to follow these tutorials and examples, or to understand the organizational philosophy of Geant4 without becoming familiar with some of these foundational concepts. While an exhaustive C++ tutorial can not be provided here, the basic concepts which are relied on most commonly in writing a simple Geant4 program are introduced.

An example C++ program is shown below:

```
#include <iostream>
int main ()
{ int num_generations_lept=3, num_generations_lept_neutrino=3;
  double elec_m,muon_m; // kg
  elec_m = 9.109e−31;
  muon_m = 1.883e−28;
  float tau_m = 3.17e−27; // kg,
  // float type has a lower maximum of significant figures than double
  bool lepton = true;
  bool boson = false;
  char elec_symbol = ‘e’;
  std::cout << “The mass of an electron is ” << elec_m << ” kg!\n”;
  return 0; }
```

As an example, this listing can be copied and placed into a file titled, ‘example.cc’, and compiled into an executable named ‘example_0’, through the terminal command,

```
$ g++ example.cc −o example_0.
```

The final executable can be run with, ‘$ ./example_0’. The symbol, $, designates a command line or terminal command.

Beginning with the first line, ‘#include’ instructs the compiler to reference the header file, ‘iostream’, for a set of existing input/output functions necessary for printing or providing input to the program, the former being used in this example.

The executed portion of the script is delineated by the function call, ‘int main ()’. Assume that all C++ programs require a main() function, and the execution of a program always begins with a call to main() . Also note the function is typed as an ‘int’, and will return the int value 0, upon successful completion. C++ functions can be thought of as typed by their return value.

Other C++ native types are declared within this script, in some cases the space necessary for such a variable type is allocated to a variable before the variable is a assigned an actual value. In the event a function has no return value, it is typed as ‘void’.

Stylistically, note that each statement ends with a semi-colon, and the body of the function call main() is offset from the rest of the script by curly brackets.

C++ supports object-oriented programming, which Geant4 programs rely on through their use classes. Classes are user-defined data types which have their own associated functions and variables. An object is an instance of a class, in the same way the variable ‘boson’, is an instance of the type, bool. If it was useful to define a new class, lepton, which stores information about lepton spin, charge and mass, this could be done through the use of an implementation file and header file:

```
#ifndef lepton_H
#define lepton_H
class lepton
{private:
  double mass;
  double charge;
 public:
  lepton(double x,double y);
  double getSpin();
  double getMass(); };
#endif
```

Above is the file, ‘lepton.h’, the header file of the class lepton. This file declares the variables, mass and charge as private members of the lepton class, of type double. These variables can not be accessed from outside of the class. There are three accessible public functions declared in this header. Two are typed to return a double value (getMass(), getSpin()), while the function lepton(double x, double y), has no apparent return type. This is a special function which exists in all classes, the class constructor, which carries the same name as the class. The constructor is called when a new instance of the class, an object, is created, an opportunity to pass and introduce defining information about the object. In this case, the constructor takes two doubles as arguments. It is good practice to include a default constructor, with predefined values to initialize the object with. It is sometimes necessary to define a class destructor, which deletes class objects. These will be further discussed in context of Geant. Finally, consider the #ifndef, #define and #endif commands. These are instructions to the compiler to include this header file only once, and should be provided in all header files.

```
#include "lepton.h"
lepton::lepton(double x,double y)
{
  mass = x; // kg
  charge = y; // Coulombs
}
double lepton::getSpin()
{return 0.5;}
double lepton::getMass()
{ return mass;}
```

Above is the implementation, or source file, ‘lepton.cc’. Its header file, ‘lepton.h’, is included within the first line. The constructor definition is provided first. Note, the name of the class, ‘lepton’ is provided before the double semi-colon in defining a function of the class. The constructor expects two doubles, x and y, which it stores in the member variables mass and charge. The typed function getSpin(), returns a preset value, while getMass() returns one of the variables set with the constructor call.

```
#include <iostream>
#include "lepton.h"
int main ()
{ lepton e(9.10938e−31, −1.602e−19);
  double s = e.getSpin();
  std::cout << s <<std::endl;
  double m = e.getMass();
  std::cout << m <<std::endl;
  return 0;}
```

A simple example creating a lepton object, e, and testing some of its member functions is shown, ‘lepton_ex.cpp’. This can be built with:

```
$ g++ lepton_ex.cc lepton.cpp −o lepton
```

assuming the header file, source file, and 'lepton_ex.cpp' sit in the same directory. Generally, header files will be organized under an /include directory, while implementations of these header files or definitions will fall under /src. Geant4 programs will rely on scripting a number of new classes. The organized definition of certain required classes and member methods will be necessary for the simulation processes carried out by the Geant4 software.

Finally, more information on basic programming with C++ and object oriented design can be found here: http://www.cplusplus.com/doc/tutorial/, https://beginnersbook.com/2017/08/cpp-oops-concepts/. The short article on object oriented concepts is especially recommended for its brief explanation of inheritance (used heavily with Geant4).

III. GEANT4 PHILOSOPHY AND USAGE

A. Conceptual Structure of a Geant4 Program

The purpose of Geant4 software is to effectively model the physical processes and interactions of particles with some form of matter. Commonly, it is of interest to model an instrument or detector, replicating its shape, extent, orientation, material, microscopic and macroscopic properties. It is then necessary to consider the physics, or rules of the system. Specifically, any particle (primary or daughter) whose interaction with the system is of interest, must be introduced. After, the physics of the system itself must be provided. When modeling a detector or material's response to a certain physical process, it is not possible or important to fully replicate the physics of our known physical world. You will most likely be interested in the study of a handful of physical processes (say, the production of Cerenkov radiation or scintillation photons, electromagnetic interactions and decay), initiated by some primary particle.

Once the static model has been fully defined, and the relevant particles and physical processes of the simulation provided, testing the response of this framework is performed by launching a primary particle of some type, initial energy, momentum and position, inside of the modeled environment. The Geant4 software evolves the primary particle through the system, evaluating the defined physical processes at each infinitesimal step to determine (with some level of randomness), any changes to the momentum of the particle, energy losses, decays, or the production of new daughter particles (which are evolved similarly). These particle trajectories can be visualized within the model, an important check in confirming proper detector construction, and understanding the production and propagation of different particles within the system.

Eventually, it will become necessary to introduce sensitive elements (the true detector components), within the system. These are designated volumes with the ability to take snapshots of a particle's state as it enters the volume (time of entry, momentum and direction, particle energy, and energy deposited within the sensitive detector). These, 'Hits', and the subsequent collection of their associated information provide a powerful method to define the response of the system. This information is especially informative when collected over a statistically significant number of primary events, providing an average response, or measured as a function of varying primary particle attributes. The effective retrieval of this data is the primary goal of these simulations.

B. Necessary Ingredients for a Simple Executable

A Geant4 executable, like all C++ programs, requires a call to the function main(). This program will begin by including three required user-defined classes, and the header file, 'G4RunManager.hh', which refers to a number of functions predefined by Geant, necessary for evolving a primary particle through a defined environment with a set of physical processes. Consider the simple executable below:

```
#include "G4RunManager.hh" //the Geant4 magic making this simulation possible
#include "DetectorConstruction.hh" // a class provided by the user defining physical properties of the model
#include "PhysicsList.hh" // a class provided by the user introducing any particles, and the list of physical
processes to consider in simulation
#include "ActionInitialization.hh" // another user-defined class in which properties of a launched primary
particle are introduced

int main() {
  G4RunManager* runManager = new G4RunManager; // construct the run manager
  runManager->SetUserInitialization(new DetectorConstruction); // set mandatory initialization classes
  runManager->SetUserInitialization(new PhysicsList);
  runManager->SetUserInitialization(new ActionInitialization); // the physics of the system must be provided to
the runManager before instructions for generating a primary particle
  runManager->Initialize(); // initialize Geant4 kernel
  int numberOfEvents = 2;
  runManager->BeamOn(numberOfEvents); // launch one default particle into the environment of the
DetectorConstruction class, with the physics specified in PhysicsList
  delete runManager; // job termination
  return 0; }
```

In its most basic form, the main executable includes three user-defined classes, describing the detector and its physical properties (DetectorConstruction.hh), the particle and physics of the system (PhysicsList.hh), and the default primary particle (ActionInitialization). This information is sufficient to provide to the Geant4 runManager, simulating the result of the primary particle passing through the system. Once the runManager is initialized, the call 'BeamOn(2)', instructs the runManager to fire two primary particles, each a separate event. As simulation of these physical processes relies on some level of randomness, the results of each event will slightly vary. The runManager is then deleted to free up its utilized memory. Zero is returned upon completion.

C. Program Structure

The following documentation and set of example scripts will begin by demonstrating how a rudimentary Geant program is written and organized. At this point, it should be clear that it is necessary to provide a main executable program, as well as source code (.cc or .cpp files) for any required user-classes (a total of four), and corresponding headers (.hh or .h files). The main executable will sit within a directory, at the same level as the sub-directories, /src and /include. Header files will be organized under the /include directory, while their implementations will be stored in /src. Compiling of this executable will be slightly more complicated than those examples given in the 'C++ PRIMER' section, and performed through the construction of a CMakeLists.txt file. The program cmake will utilize this file, building a Makefile, which is then used for compilation.

Once a simple executable has been compiled successfully, visualization with Geant will be introduced through the G4VisManager (from the G4VisExecutive class). This is another set of functions provided by Geant, allowing for visual confirmation of the model or environment defined in DetectorConstruction.cc, as well as the results of individual events.

After the basic setup of the simulation is confirmed, supplying additional commands or directives to the executable will be discussed. In general, commands can be supplied to an instance of G4UIManager (G4UIExecutive class). This allows for user interfacing with the executable program. These commands can be supplied to the G4UIManager through a macro (.mac file), or through an interactive session hosted by the executable.

Last, the SensitiveDetector (SD) class and Hit (SDHit) class will be introduced. A volume registered as an object of the SensitiveDetector class will provide information on its Hits, instances where particles have intersected the detector. The Hit class will organize the necessary user-set information associated with this interaction.

Ultimately, the final program should well model the physical attributes and relevant processes of the system, support visualization and user interfacing, and provide the necessary hit information to fully characterize the result of each event.

IV. DETECTOR CONSTRUCTION

A. Introduction

The first user-defined class passed to the G4RunManager is DetectorConstruction, providing all the static physical properties of the simulation. This class must inherit from the Geant4 class, G4VUserDetectorConstruction, which is a set of predefined rules and functions provided by Geant. DetectorConstruction will have a constructor, destructor, and Construct() function, which necessarily returns the description of the model. As a general rule, user-defined classes with a Geant-specific functionality will inherit from a Geant4-provided class. Additionally, each class will have a constructor, destructor, and at least one mandatory method whose name must match that expected by Geant, in this case, Construct(). A simple header file is provided below:

```
#ifndef DetectorConstruction_h
#define DetectorConstruction_h 1
#include "G4VUserDetectorConstruction.hh"
class G4VPhysicalVolume

class DetectorConstruction : public G4VUserDetectorConstruction {
    public:
        DetectorConstruction();
        virtual ~DetectorConstruction();
        virtual G4VPhysicalVolume* Construct(); };
#endif
```

The necessary Geant header file, G4VUserDetectorConstruction.hh, is included, and the class G4VPhysicalVolume is forward declared. A forward declaration is made for the class of any object explicitly referenced within the header file, in this case, G4VPhysicalVolume. This is a safeguard, letting the compiler know about the existence of such a class. The class DetectorConstruction is declared as public, and inherits from G4VUserDetectorConstruction. The DetectorConstruction class itself has three public methods. The constructor and destructor have no return type, while the Construct() function returns type G4VPhysicalVolume*, a pointer to an object of G4VPhysicalVolume. The following sub-sections will present a number of common ingredients for a Construct() definition, and culminate in the implementation file DetectorConstruction.cpp.

B. Defining Materials

Every portion of your simulated environment must be assigned some representative material (a room filled with air, the specific form of plasitc a slab of scintillator might be composed of, polyvinyl toluene (PVT), a lead brick used for vetoing). While your Geant4 model may not perfectly replicate every component of your real, physical detector, the larger material portions and shapes which are expected to influence the creation and propagation of particles should be considered (don't worry about modeling the plastic insulation on a PMT power cable!). Consider first an example of G4Element (assuming G4Element.hh has been included in the DetectorConstructon.cc file):

```
G4double z, a; // note that Geant provides its own classes for doubles, Strings, ints, etc.
G4String name, symbol;
a = 1.01*g/mole; // and that numerical values can be assigned units
G4Element* elH = new G4Element(name="Hydrogen", symbol="H", z=1.0, a);
a = 16.00*g/mole;
G4Element* elO = new G4Element(name="Oxygen", symbol="O", z=8.0, a);
G4Element* elN = new G4Element("Nitrogen", "N", 7.0, 14.01*g/mole);
```

The class description of G4Element, or any other Geant4 class may be found online. Creating a new object of G4Element requires specifying its name, symbol, atomic number (here, 'z'), and atomic mass ('a'). The values are expected in order with the respective types demonstrated here. The variable 'name' is an object of the class G4String, while 'z' and 'a' are both expected as G4doubles. Instead of naming these variables outside of the G4Element declaration, they can instead be passed directly to the G4Element constructor, and their types will be interpreted according to the G4Element constructor definition.

Now consider how the materials, ice and water, might be defined:

```
G4int ncomponents, natoms;
G4double density = 0.9340*g/cm3;
G4Material* Ice = new G4Material(name="Ice", density, ncomponents=2);
Ice->AddElement(elH, natoms=2);
Ice->AddElement(elO, natoms=1);
G4Material* Ice = new G4Material(name="Water", 1.0*g/cm3, ncomponents=2);
Water->AddElement(elH, natoms=2);
Water->AddElement(elO, natoms=1);
```

The constructor of the class G4Material requires a name, a value defining the material density, and the number of components. The AddElement member function of the object Ice is then called twice to set hydrogen and oxygen as the constituents of Ice. Water is created similarly.

While water is a molecule with a chemical definition, you can also define materials through a mixture of elements:

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

Air, a mixture of nitrogen and oxygen, can be specified similarly to Ice and Water, where the proportions of elements are provided as percentages.

Any element or material can be modeled through the above definitions. However, Geant4 provides a convenience library of predefined materials, accessible through the G4NistManager:

```
auto nistManager = G4NistManager::Instance();
nistManager->FindOrBuildMaterial("G4_GLASS_LEAD");
auto Glass = G4Material::GetMaterial("G4_GLASS_LEAD");
```

In this example, nistManager is an instance of G4NistManager ('auto' recognizes the class of the returned object in Instance(), and automatically types 'nistManager' as a pointer to a 'G4NistManager' object). The object 'Glass' now represents a material with the attributes of lead glass (name, density, composition). See: http://geant4-userdoc.web.cern.ch/geant4-userdoc/UsersGuides/ForApplicationDeveloper/html/Appendix/materialNames.html, for elements, compounds and nuclear materials provided by Geant4.

Either through new definitions of elements, or references to the materials supplied by Geant, each component of your model will require a specified G4Material.

C. Defining Material Properties

While a material alone carries the intrinsic properties associated with its density and the atomic structures of its elements, there is another class of properties which can optionally be associated with a material. Generally, these properties can be thought to define how optical (wave-like) photons are produced and propagate (Cerenkov radiation, scintillation production, scattering and absorption rely on their definitions).

Consider the simple example of Cerenkov radiation, where photons ranging in wavelength from UV to blue are produced as a conical wave-front from a charged particle traveling faster than the speed of light (in some medium). The intensity of radiation is given as a function of photon wavelength, λ, and distance traveled, x:

$$\frac{dN}{dx} = 2\pi\alpha\left(1 - \frac{1}{\beta^2 n^2(\lambda)}\right)\left(\frac{1}{\lambda_2} - \frac{1}{\lambda_1}\right) \tag{1}$$

N is the number of Cerenkov photons emitted between wavelengths λ_1 and λ_2, α the fine structure constant ($e^2/\hbar c$), and β is the ratio of primary charged particle speed to the speed of light. Regardless of whether this equation is intuitive at first glance, note that the number of photons between relevant frequencies λ_1 and λ_2, is actually a function of photon wavelength (or momentum/energy), through the wavelength-dependent (momentum/energy) index of refraction, $n(\lambda)$. In many other applications, n, is assumed constant. However, the percentage level variations in n over the relevant values of λ for Cerenkov radiation should be associated with the material for proper (or any) modeling of Cerenkov. At the very least, a single value should be provided, as no default index is supplied. In Geant4, these values would be supplied as an array, or table, corresponding to different photon energy:

```
G4double photonEnergy[] =
            { 2.034*eV, 2.068*eV, 2.103*eV, 2.139*eV,
              2.177*eV, 2.216*eV, 2.256*eV, 2.298*eV,
              2.341*eV, 2.386*eV, 2.433*eV, 2.481*eV,
              2.532*eV, 2.585*eV, 2.640*eV, 2.697*eV,
              2.757*eV, 2.820*eV, 2.885*eV, 2.954*eV,
              3.026*eV, 3.102*eV, 3.181*eV, 3.265*eV,
              3.353*eV, 3.446*eV, 3.545*eV, 3.649*eV,
              3.760*eV, 3.877*eV, 4.002*eV, 4.136*eV };
  const G4int nEntries = sizeof(photonEnergy)/sizeof(G4double);
  G4double refractiveIndex1[] =
            { 1.3435, 1.344, 1.3445, 1.345, 1.3455,
              1.346, 1.3465, 1.347, 1.3475, 1.348,
              1.3485, 1.3492, 1.35, 1.3505, 1.351,
              1.3518, 1.3522, 1.3530, 1.3535, 1.354,
              1.3545, 1.355, 1.3555, 1.356, 1.3568,
              1.3572, 1.358, 1.3585, 1.359, 1.3595,
              1.36, 1.3608}; // Water
```

Above is a specific example for water. A table of photon energies is provided, with a corresponding table (note same number of entries), of refractive indexes. Optical properties like index of refraction are generally established through experiment. Resources for constructing these tables of empirical measurements will be provided at the end of this section.

Now consider the process of scintillation. A scintillating material absorbs the energy deposited by an incoming, ionizing particle, and emits it in the form of scintillation photons. The subsequent photons are released over an extended period of time. In general, the expectation of the number of released photons, N, is a function of two timescales; τ_s parameterizing the slow component of scintillation, and τ_f describing the fast component:

$$N = A \exp\left(-\frac{t}{\tau_f}\right) + B \exp\left(-\frac{t}{\tau_s}\right). \tag{2}$$

Here, A and B are constants setting the relative contributions of either mode of scintillation. Often, the fast scintillation component alone is a sufficient description of scintillation yield, but this can't always be assumed. In general, the intensities of light production for either mode will need to be specified as PDFs in tables as a function of scintillation photon energy.

Additional parameters describing the scintillation are also necessary. The scintillation yield is a value representing the characteristic light yield per unit energy of a material. The yield ratio specifies the fractional amount of scintillation photons attributed to the fast component to the amount of total scintillation. Resolution scale controls the broadness of the statistical distribution describing number of photons produced (doping and impurities within materials add greater variance to the scintillation response). There are a number parameters defining this scintillation process. This information is interpreted by the Geant4 runManager by organizing it within a

G4MaterialPropertiesTable, which is then associated with whatever material is being modeled as a scintillator:

```
G4double scintilFast[] =
            { 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00,
              1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00,
              1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00,
              1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00,
              1.00, 1.00, 1.00, 1.00 }; // equal amount of scintillation expected at any given energy
 assert(sizeof(scintilFast) == sizeof(photonEnergy));
 G4double scintilSlow[] =
            { 0.01, 1.00, 2.00, 3.00, 4.00, 5.00, 6.00,
              7.00, 8.00, 9.00, 8.00, 7.00, 6.00, 4.00,
              3.00, 2.00, 1.00, 0.01, 1.00, 2.00, 3.00,
              4.00, 5.00, 6.00, 7.00, 8.00, 9.00, 8.00,
              7.00, 6.00, 5.00, 4.00 }; // amount of scintillation varies as a function of energy
 assert(sizeof(scintilSlow) == sizeof(photonEnergy));
 G4MaterialPropertiesTable* myMPT1 = new G4MaterialPropertiesTable();
 myMPT1->AddProperty("RINDEX", photonEnergy, refractiveIndex1, nEntries)
       ->SetSpline(true);
 myMPT1->AddProperty("FASTCOMPONENT", photonEnergy, scintilFast, nEntries)
       ->SetSpline(true);
 myMPT1->AddProperty("SLOWCOMPONENT", photonEnergy, scintilSlow, nEntries)
       ->SetSpline(true);
 myMPT1->AddConstProperty("SCINTILLATIONYIELD", 50./MeV);
 myMPT1->AddConstProperty("RESOLUTIONSCALE", 1.0);
 myMPT1->AddConstProperty("FASTTIMECONSTANT", 1.*ns);
 myMPT1->AddConstProperty("SLOWTIMECONSTANT", 10.*ns);
 myMPT1->AddConstProperty("YIELDRATIO", 0.8);
 myMPT1->DumpTable();
 Water->SetMaterialPropertiesTable(myMPT1);
 Water->GetIonisation()->SetBirksConstant(0.126*mm/MeV);
```

Here, we have set two arrays, 'scintilFast' and 'scintilSlow', to represent the intensities of each component at a given photon energy. A new G4MaterialPropertiesTable object is made, and used to store this information, the refractive index of water, any relevant constant describing the material specific properties of the scintillator, and the decay timescales. Then, the G4MaterialPropertiesTable of Water is set to this object. Finally, note the variable describing

Birk's constant is set. Birk's equation describes the light yield per unit path length, and is material-specific.

Additional processes undergone by photons include bulk absorption, Rayleigh scattering, and wavelength-shifting. If necessary, more information on implementing these processes, as well as determining the proper values for modeling scintillation processes in organic scintillator will be given in lecture or elsewhere.

D. Defining a Model

Once the materials and their respective properties have been introduced, the shape, material and position of any model component can be defined. To fully specify a volume with these physical attributes, we begin by creating an object of some basic shape provided by Geant4. Consider a cube the size of a small room which we intend to fill with water. We define the shape, or 'Solid':

```
G4double world_hx = 3.0*m;
G4double world_hy = 3.0*m;
G4double world_hz = 3.0*m;
G4Box* worldBox = new G4Box("World", world_hx, world_hy, world_hz);
```

The four arguments of the 'G4Box' constructor include the name, and three spatial dimensions of the box. Now, to specify the material associated with 'worldBox':

```
G4LogicalVolume* worldLV = new G4LogicalVolume(worldBox, Water, "World");
```

This new object, 'worldLV', is a 'Logical Volume'. The Solid or basic shape is provided by including 'worldBox', as the first argument of the G4LogicalVolume constructor. Then the material and name of the Logical Volume is given. We would now like to position and orient this Logical Volume within our model:

```
G4PVPlacement* worldPV = new G4PVPlacement(0, G4ThreeVector(), worldLV, "World", 0, false, 0, TRUE);
```

This new object, 'worldPV', is a Physical Volume, taking the Logical Volume, 'worldLV', as an argument. Referencing the input arguments from left to right, '0' designates no rotation matrix is supplied to specify orientation, 'G4ThreeVector()' sets the position of the Physical Volume center at the origin, the respective Logical Volume is ‘worldLV’, the Physical Volume is named ‘World’, there is no set Mother Volume (0), the volume has not been parameterised (false), this is the zeroth copy (0), and the runManager should check for volume overlaps with this volume at execution.

This three tiered method for declaring an object’s shape, material and position is followed for all new volumes. These distinctions may seem gratuitous, but having multi-leveled declarations allows for Solids of different materials and placements without re-declaring many objects of the same initial shape.

Now, note this first volume has a special property. Consider the concept of a preexisting ‘Mother Volume’, and new ‘Daughter Volume’ to be placed. The Mother Volume position and orientation defines its own respective coordinate system with origin at its center. The Daughter Volume is placed with respect to these axes, fully inside the Mother Volume, and can not overlap with other such Daughter Volumes. The material and shape of the Daughter Volume supersedes that previously filled by the Mother. At the top of this inheritance tree is the ‘World Volume’, which contains all other volumes within the model, and has no Mother Volume. A simulation requires at minimum this World Volume. Here is an example introducing one smaller Daughter Volume within our ‘World’:

```
G4double Rmax = 0.3*m;
G4double Rmin = 0.0*m;
G4double StartPhi = 0.0;
G4double DeltaPhi = 2.0*3.14159265358979323846;
G4double StartTheta = 0.0;
G4double DeltaTheta = 3.14159265358979323846;
G4Sphere* PMT = new G4Sphere(“PMT”, Rmin, Rmax, StartPhi, DeltaPhi, StartTheta, DeltaTheta);
G4LogicalVolume* PMTLV = new G4LogicalVolume(PMT, Water, “PMT”);
G4PVPlacement* PMTPV = new G4PVPlacement(0, G4ThreeVector(1.0*m,1.0*m,1.0*m),PMTShellLV, “PMT”,
 worldLV, false, 0, TRUE);
```

E. Defining Model Interfaces

The final topic of Detector Construction (what will arguably be the most involved class in terms of developer input), is defining the physical properties of material interfaces. Modeling these interfaces will reuse concepts of Materials, MaterialPropertiesTables and the abstraction used in fully defining Physical Volumes.

Consider the two most common kinds of interface, a dielectric with a metal, and a dielectric with dielectric. An example of a dielectric interfaced with a metal would be a volume of air in direct contact with a slab of gold. When particles propagating through air meet the metal slab, they will either be absorbed or reflected. When two dielectrics are paired (water and air), particles intersecting the interface from either side will have some probability of transmission, refraction, or reflection. The only exception is when the geometry of the intersection allows for total internal reflection, in which case reflection always occurs.

Proper handling of these boundary physics relies on the concept of Surfaces (which mimic the declaration structure of Volumes). Defining a Surface will always begin with defining an object of type G4OpticalSurface, which stores physical properties about the interface. Defining a new Surface between two dielectrics may begin with:

```
G4OpticalSurface* opWaterSurface = new G4OpticalSurface(“WaterSurface”);
opWaterSurface−>SetType(dielectric_dielectric);
opWaterSurface−>SetFinish(polished);
opWaterSurface−>SetModel(glisur);
```

The types (dielectric_dielectric, dielectric_metal) have already been discussed. There are many options for setting the finish of the surface (polished, ground, polishedfrontpaint, polishedbackpaint, etc), which can be further referenced through online materials. The model associated with the surface is the model used to simulate boundary physics at the surface (glisur or unified). Essentially, particles interacting with the surface of a solid reflect or refract as a function of the normal vector of their interaction point. The normals describing a polished surface are inherited from the Solid paired to the surface. However, a ground surface is modeled as a collection of microfacets with randomized normal vectors. The choice in distribution of these randomized normal vectors with respect to the polished surface normal is the topic of models glisur and unified.

Once the G4OpticalSurface is defined, these physical properties can be associated with the interface between two volumes with a G4LogicalBorderSurface:

```
G4LogicalBorderSurface* WaterSurface = new G4LogicalBorderSurface(“WaterSurface”, worldPV, PMTPV,
OpWaterSurface);
```

Here, WaterSurface, is defined through its name, “WaterSurface”, the Physical Volumes of the two interfacing surfaces, and the Optical Surface it represents, OpWaterSurface. Note that these properties will only translate to the in-contact portions of the two Physical Volumes. Alternately, if a Logical Volume can be entirely described by a single set of surface properties, these can be set through the G4LogicalSkinSurface class:

```
G4LogicalSkinSurface* WaterSurface = new G4LogicalSkinSurface("WaterSurface", PMTLV, OpWaterSurface);
```

This time, the name of a single Logical Volume, 'PMTLV', is specified.

Again, a G4OpticalSurface can be defined with extra properties allowing for proper modeling of any optical boundary physics. These include (as a function of photon energy), refractive index, reflectivity, transmission efficiency, the probability of back-scatter (reflection in the opposite direction of incidence), and the probability of specular reflection with respect to a specified normal. These concepts and proper selection of representative empirical measurements may be discussed further in lecture.

```
G4MaterialPropertiesTable* myST1 = new G4MaterialPropertiesTable();
const G4int num = 2;
G4double ephoton[num] = {2.034*eV, 4.136*eV};
G4double refractiveIndex[num] = {1.35, 1.40};
G4double specularLobe[num] = {0.3, 0.3};
G4double specularSpike[num] = {0.2, 0.2};
G4double backScatter[num] = {0.2, 0.2};
G4double reflectivity[num] = {0.3, 0.5};
G4double efficiency[num] = {0.8, 1.0};
myST1->AddProperty("RINDEX", ephoton, refractiveIndex, num);
myST1->AddProperty("SPECULARLOBECONSTANT", ephoton, specularLobe, num);
myST1->AddProperty("SPECULARSPIKECONSTANT", ephoton, specularSpike, num);
myST1->AddProperty("BACKSCATTERCONSTANT", ephoton, backScatter, num);
myST1->AddProperty("REFLECTIVITY", ephoton, reflectivity, num);
myST1->AddProperty("EFFICIENCY", ephoton, efficiency, num);
myST1->DumpTable();
opWaterSurface->SetMaterialPropertiesTable(myST1);
```

Above, a new G4MaterialPropertiesTable is defined, and filled with values corresponding to the optical boundary properties. Here, only two energies are specified, a rough minimum and maximum for the expected optical photon energy. Optical properties corresponding to energies between these values will be interpolated during execution. This table is then supplied to 'opWaterSuface' as its MaterialPropertiesTable.

F. Example Implementation File

The volumes and properties discussed in the previous sections are implemented in 'Detector-Construction.cc', provided with this document.

V. PHYSICAL PROCESSES AND PARTICLES

A. Introduction

This section will focus on construction of the user-defined class, PhysicsList, which necessarily inherits from the provided Geant4 class, G4VUserPhysicsList.hh. This class names all relevant particles used within the simulation, and any physical process. The structure of the class's header file and implementations will be presented. However, as many models may reuse very similar sets of particles and processes, different lists of grouped particles and processes provided directly by Geant are often used as opposed to freshly constructing in a new user-defined class. As an example, an experiment measuring secondary cosmic rays might be well described assuming the existence of gamma rays, electrons, muons, positrons, optical photons, in combination with the many optical processes and boundary processes already discussed, decay and electromagnetic interactions, as well as transportation processes. Below is a stripped example header:

```
#ifndef PhysicsList_h
#define PhysicsList_h 1
#include "globals.hh"
#include "G4VUserPhysicsList.hh"
class G4Cerenkov;
class G4Scintillation;
//class G4OpAbsorption;
class PhysicsList : public G4VUserPhysicsList
{ public:
    PhysicsList();
    virtual ~PhysicsList();
  public:
    virtual void ConstructParticle();
    virtual void ConstructProcess();
    virtual void SetCuts();
    void ConstructDecay();
    void ConstructEM();
    void ConstructOp();
  private:
    static G4ThreadLocal G4int fVerboseLevel;
    static G4ThreadLocal G4Cerenkov* fCerenkovProcess;
    static G4ThreadLocal G4Scintillation* fScintillationProcess;
#endif
```

Again, it is common practice to use predefined lists of physical processes and particles. Here, we walk through the different methods and their purposes so the physics 'known' to the simulation can be easily referenced from these files and understood.

As with the DetectorConstruction class, the PhysicsList class also must inherit from a predefined Geant4 class, G4VUserPhysicsList.hh.

The different classes with predefined physical processes are then declared, 'class G4Scintillation;'. Some of the relevant physics has not been included, and should be added in. Specifically, the classes G4OpAbsorption, G4OpRayleigh, G4OpMieHG, G4OpBoundaryProcess should be introduced (forward declared), and should also have static members.

Finally, the class definition for the new class, PhysicsList is made. Public methods include the class constructor, destructor, a method to construct or declare the particles to be used within the simulation (required by Geant4), a method to construct the physical processes (required by Geant4), and a method to set thresholds on particle production (Geant can't track each particle forever: required). As there are many physical processes which can be roughly grouped by type (decay processes, electromagnetic and optical), these processes are declared in separate methods, which are each called by ConstructProcess(). The first private object is an int value used to set the level of verbosity (printed commentary), when certain methods are called. The private objects corresponding to different physical processes are pointers to such classes provided by Geant4, to be used in the implementation file.

B. Relevant Particles

In each PhysicsList class, the relevant particles must be provided before the physical processes. This is performed through a call to ConstructParticle(), which must be implemented (with this name), for successful simulation.

Geant4 provides six categories of particles:

1. lepton

2. meson
3. baryon
4. boson
5. shortlived (resonant particles, vector mesons, delta baryons, etc.)
6. ion (including heavy nuclei)

Each particle (e.g. the electron, muon, and tau of the lepton class), is defined in a sub-directory under geant4/source/particles. Much of the content describing particles (within G4ParticleDefinition) can not be edited, nor should be.

Accessing a particle definition (in the case of the G4Electron class), can be performed through creating an electron object, G4Electron::theInstance. The pointer to this object can be accessed through G4Electron::ElectronDefinition() or G4Electron::Definition().

As reference, Geant4 provides a dictionary of particles, G4ParticleTable. Particles can be located either through name or PDG (Particle Data Group) encoding:

```
FindParticle(G4String name);
FindParticle(G4int PDGencoding);
```

Additionally, users have the ability to create certain particles (heavy nuclei), through the following definition:

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

However, the many predefined particles of Geant4 will almost always suffice.

Again for reference, consider a simulation in which only two particles (electron and proton) are used. The ConstructParticle() method should consist of calls to all required particles:

```
void PhysicsList::ConstructParticle()
  {
    G4Proton::ProtonDefinition();
    G4Electron::ElectronDefinition();
  }
```

As the actual number of particles referenced in a typical simulation is much higher, Geant4 provides six utility classes, each of which make calls to all particle definitions within one of the six particle categories introduced. This makes it very easy to import all possible (defined in Geant4), particles which might be used by the physical processes to be defined. These constructors are below:

1. G4BosonConstructor
2. G4LeptonConstructor
3. G4MesonConstructor
4. G4BaryonConstructor
5. G4ShortlivedConstructor
6. G4IonConstructor

An example method, ConstructParticle(), is given below with the necessary header files.

```
#include "G4BosonConstructor.hh"
void PhysicsList::ConstructParticle()
{
  G4BosonConstructor bConstructor;
  bConstructor.ConstructParticle();
}
```

Here, only the bosons provided by Geant4 have been constructed. Clearly, a simulation consisting of only bosons would not be very interesting. Make sure to construct the other relevant particles in your own PhysicsList implementation.

C. Relevant Physical Processes

Similar to the construction of particles, Geant4 expects physics processes to be introduced through a specific method in the PhysicsList class, ConstructProcess().

There are seven essential categories of processes:

1. electromagnetic
2. hadronic

3. transportation
4. decay
5. optical
6. photolepton_hadron
7. parameterisation

The electromagnetic, optical, decay and transportation processes will be used heavily in modeling the indirect detection of cosmic-rays through scintillation or Cerenkov radiation.

Each process derives from the G4VProcess class, with methods AtRestDoIt, AlongStepDoIt, PostStepDoIt. The 'Step' is the increment (in length and time) over which the simulation is evolved. A primary particle progresses through the detector in increments, at each step having some probability of undergoing a number of physical processes. These probabilities are influenced by the type of particle, its state, and the physical properties of the detector. Methods used in this physical modeling process are evaluated at the beginning, middle, or completion of each step. In general, Geant4 users must register physical processes on a particle-by-particle basis with the G4ProcessManager, providing additional information about the call time and order of processes. This information will be provided, and is also illustrated in the many PhysicsList classes included as examples with your Geant4 install. An example listing to illustrate this registration is below:

```
void PhysicsList::ConstructProcess()
{
  AddTransportation();
  ConstructEM();
}

void PhysicsList::ConstructEM()
{
    G4ParticleDefinition* particle = G4Gamma::GammaDefinition();
    G4ProcessManager* pmanager = particle->GetProcessManager();
    pmanager->AddDiscreteProcess(new G4GammaConversion());
    pmanager->AddDiscreteProcess(new G4ComptonScattering());
    pmanager->AddDiscreteProcess(new G4PhotoElectricEffect());

    G4ParticleDefinition* particle = G4Positron::PositronDefinition();
    G4ProcessManager∗ pmanager = particle−>GetProcessManager();
    pmanager−>AddProcess(new G4eMultipleScattering(),−1, 1, 1);
    pmanager−>AddProcess(new G4eIonisation(), −1, 2, 2);
    pmanager−>AddProcess(new G4eBremsstrahlung(), −1, 3, 3);
    pmanager−>AddProcess(new G4eplusAnnihilation(), 0,−1, 4);
}
```

Here, an example ConstructProcess() function is shown, with a short ConstructEM() definition. The body of the ConstructEM() could exist within ConstructProcess(), but is separated for readability. The AddTransportation() class is provided in the G4VUserPhysicsList class to register transportation processes with all particles; no further inline definition for AddTransportation() is required. Physical processes are registered with a gamma particle (high energy, particle-like photon, as opposed to a wave-like optical photon), and positron. Normally, many more particles would be registered with processes. A pointer to the particle definition is called, and its G4ProcessManager is instantiated. Discrete processes (only performed after each Step), are easily registered for the gamma particle. Continuous (occurring only during the Step), and AtRest (occuring only after Step completion), processes are also registered easily. Sets of processes requiring call ordering are added with AddProcess(), and require additional arguments. Understanding and properly adding AddProcess() arguments is beyond the scope of this document. Properly registered physical processes will be provided.

Finally, as reference, there is a helper function provided by Geant4, G4PhysicsListHelper, which is aware of process type and proper ordering. An example usage is below (replacing the need for specification with the G4ProcessManager):

```
void MyPhysicsList::ConstructEM()
{
  G4PhysicsListHelper∗ ph = G4PhysicsListHelper::GetPhysicsListHelper(); // Pointer to G4PhysicsListHelper
  G4ParticleDefinition∗ particle = G4Positron::PositronDefinition(); // Get particle definition
  ph−>RegisterProcess(new G4GammaConversion(), particle); // Register processes
  ph−>RegisterProcess(new G4ComptonScattering(), particle);
  ph−>RegisterProcess(new G4PhotoElectricEffect(), particle);
}
```

This example registers only the positron. It may be of interest to try replacing some of the registration performed by G4ProcessManager with G4PhysicsListHelper, after a working example is completed. This would ideally preserve the physics of the simulation, but provide an opportunity to better understand the PhysicsList source code.

D. Stripped Implementation File

A partially completed implementation file, corresponding to the completed PhysicsList header file discussed in this section should be provided with this document. Process registry is provided within a stripped PhysicsList implementation file. The SetCuts() method is also provided.

VI. TESTING THE PHYSICS OF YOUR SYSTEM: 'GENERATE ACTION'

A. Introduction

The third mandatory user-defined class of our main executable is the ActionInitialization class, which inherits from G4VUserActionInitialization. The ActionInitialization class is essentially a way to manage user actions during different stages of simulation, and organizes those other classes corresponding to such actions. Consider the simple header file and inline code below:

```
#ifndef ActionInitialization_h
#define ActionInitialization_h 1
#include "G4VUserActionInitialization.hh"
class ActionInitialization : public G4VUserActionInitialization
{ public:
    ActionInitialization();
    virtual ~ActionInitialization();
    virtual void Build() const;
    virtual void BuildForMaster() const; };
#endif
```

```
#include "ActionInitialization.hh"
#include "PrimaryGeneratorAction.hh"
ActionInitialization::ActionInitialization()
 : G4VUserActionInitialization(){}
ActionInitialization::~ActionInitialization() {}
void ActionInitialization::BuildForMaster() const {}
void ActionInitialization::Build() const
{ SetUserAction(new PrimaryGeneratorAction()); }
```

This class has a constructor, destructor, and two required, named functions, BuildForMaster() and Build(). Build() should be supplied any user actions, and requires only a single argument in this basic example, the PrimaryGeneratorAction(). BuildForMaster() currently does nothing, but should still be defined. Feel free to re-purpose the above scripts.

B. The Primary Generator Action

PrimaryGeneratorAction is the fourth required user-defined class. Geant4 provides no default primary particle, so the initial particle injected into the system must have its type, initial position, momentum, energy and entrance time defined by the user. This occurs within the PrimaryGeneratorAction class, which must inherit from the Geant4 class, G4VUserPrimaryGeneratorAction. As usual, this class will have a constructor, destructor, and one named method expected by the Geant4 runManager at execution, GeneratePrimaries(). Consider the example header below:

```
#ifndef PrimaryGeneratorAction_h
#define PrimaryGeneratorAction_h 1
#include "G4VUserPrimaryGeneratorAction.hh"
#include "globals.hh"
class G4ParticleGun;
class G4Event;
class PrimaryGeneratorAction : public G4VUserPrimaryGeneratorAction
{
  public:
    PrimaryGeneratorAction();
    virtual ~PrimaryGeneratorAction();
  public:
    virtual void GeneratePrimaries(G4Event*);
  private:
    G4ParticleGun* fParticleGun;
};
#endif
```

Any necessary classes are declared. The PrimaryGeneratorAction class which inherits from G4VUserPrimaryGeneratorAction is defined with the expected constructor, destructor and GeneratePrimaries() method (a function of a single event). The G4ParticleGun object used within the class is designated as private. An example of the corresponding inline definitions are below:

```
#include "PrimaryGeneratorAction.hh"
#include "Randomize.hh"
#include "G4Event.hh"
#include "G4ParticleGun.hh"
#include "G4ParticleTable.hh"
#include "G4ParticleDefinition.hh"
#include "G4SystemOfUnits.hh"
PrimaryGeneratorAction::PrimaryGeneratorAction()
 : G4VUserPrimaryGeneratorAction(),
   fParticleGun(0)
{
  G4int n_particle = 1;
  fParticleGun = new G4ParticleGun(n_particle);
  G4ParticleTable* particleTable = G4ParticleTable::GetParticleTable();
  G4ParticleDefinition* particle = particleTable->FindParticle("gamma");
  fParticleGun->SetParticleDefinition(particle);
  fParticleGun->SetParticleTime(0.0*ns);
  fParticleGun->SetParticlePosition(G4ThreeVector(-2.0*m,-2.0*m,-2.0*m));
  fParticleGun->SetParticleMomentumDirection(G4ThreeVector(-1.0,-1.0,-1.0));
  fParticleGun->SetParticleEnergy(1.0*eV);
}
PrimaryGeneratorAction::~PrimaryGeneratorAction()
{
  delete fParticleGun;
}
void PrimaryGeneratorAction::GeneratePrimaries(G4Event* anEvent)
{
  fParticleGun->GeneratePrimaryVertex(anEvent);
}
```

The constructor for this class defines a number of user-set attributes for a primary particle. Specifically, the object fParticleGun is a new instance of G4ParticleGun, for a single event (n_particle = 1). fParticleGun stores information about the primary, including type, launch time, position, direction and energy. The call to GeneratePrimaries(), occurring when a primary is generated with a call to BeamOn(), launches a particle with these default values. Finally, the destructor deletes fParticleGun, freeing the memory stored with it.

Choice of these primary particle values will generally depend on DetectorConstruction and the specific physical processes a user might be interested in testing. Feel free to re-purpose these provided scripts.

VII. INTERACTING WITH A SIMPLE EXAMPLE

A. Compiling and Testing

At this point, enough information has been provided to define header and implementation files for the four necessary user-defined classes (inheriting from a specific Geant4 library), each with at least one named method required by the G4RunManager at execution. Additionally, the C++ file containing the main() method should initialize a G4RunManager with the necessary classes, and make a call to the BeamOn() method, to fire a predefined primary particle.

One last file is recommended to compile this executable. Consider the following .txt file, CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.6 FATAL_ERROR)
project(Example)
option(WITH_GEANT4_UIVIS "Build example with Geant4 UI and Vis drivers" ON)
if(WITH_GEANT4_UIVIS)
  find_package(Geant4 REQUIRED ui_all vis_all)
else()
  find_package(Geant4 REQUIRED)
endif()
include(${Geant4_USE_FILE})
include_directories(${PROJECT_SOURCE_DIR}/include
                    ${Geant4_INCLUDE_DIR})
file(GLOB sources ${PROJECT_SOURCE_DIR}/src/ *.cc)
file(GLOB headers ${PROJECT_SOURCE_DIR}/include/ *.hh)
add_executable(Example Example.cc ${sources} ${headers})
target_link_libraries(Example ${Geant4_LIBRARIES} )
install(TARGETS Example DESTINATION bin)
```

Place the contents of this file, CMakeLists.txt, within the file containing the C++ executable, /src and /header. This file provides instructions for the creation of a Makefile, used in compilation. Note that the implementation file extension must be '.cc', and the header extension '.hh' for the files to be properly included. Also note that the script builds with the UI and Vis drivers supplied by Geant4. These provide user interfacing and visualization, and will be introduced shortly.

The first step in compiling is to move to or create a new directory outside of the directory containing the C++ executable, /src and /header. Inside this new directory, /build_Example, the following command is given:

```
$ cmake −DCMAKE_PREFIX_PATH=/path/to/geant4−install path/to/build_Example
```

Here, /path/to/geant4-install is the full path to the location of the geant4-install directory. The actual name of the geant4-install directory will differ, including a postfix with the specific Geant4 install version. 'path/to/build_Example' is the full path to the new directory you intend to build in.

After this command completes successfully, and a Makefile is produced, make:

```
$ make −jN
```

Here, N is less than or equal to the number of cores on your machine. If you are unsure, run with N = 1.

Assuming the build completes successfully, the simulation can be tested by running './Example', within the build directory. If the optical photon processes and necessary properties have been added properly, and the primary particle is selected to interact with the modeled environment, the progress of the run should be printed. Unfortunately, this is not very useful for visualization, interacting with or controlling the simulation and its output at run-time. Adding in these capabilities will be the topic of the next sub-sections.

B. Visualization

Visualisation with Geant4 is a powerful tool, allowing a user to see both their modeled detector, and the paths of all (or a selection) of particles from one or multiple runs. These capabilities are enabled through the Geant4 class, G4VisExecutive. An instance of G4VisManager will register the geometry of the simulation, the results of each run, and any user provided commands, to construct an interpretable visualisation code to be read by an independent software. Essentially, a readable 3D model of the result of each primary event is saved.

If the executable has been compiled with Vis drivers ON, visualisation capabilities can be added by editing the main executable. At the beginning of the script, make sure to include the header "G4VisExecutive.hh". Next, a pointer to an instance of G4VisManager is initialized. Specifically, the following lines are included after the runManager is initialized:

```
G4VisManager* visManager = new G4VisExecutive;
visManager->Initialize();
```

At the end of the main() call, the visManager must be deleted, as well.

In the next section, the result of an event is translated to a file for visualisation through use of UI (user interface) commands.

C. Interactive Usage Commands

How can a user interact with the executable (and simulation) at run-time? Something as simple as adjusting the amount of of i/o produced from a run might necessitate this type of action. More complex requests, like changing the properties of the primary particle after compilation, may also be handled in this fashion. These and similar operations are facilitated through use of the G4UIExecutive class.

Implementation of this class is done in the main executable. First, make sure to include the G4UIExecutive class and G4UIManager class at the beginning of the script. The following lines could be included after the initialization of the VisManager:

```
G4UIExecutive* ui = nullptr;
if (argc == 1) ui = new G4UIExecutive(argc,argv);
G4UImanager* UImanager = G4UImanager::GetUIpointer();
if (ui) {
  // interactive mode
  ui->SessionStart();
  delete ui;
}
else {
  // batch mode
  G4String command = "/control/execute ";
  G4String fileName = argv[1];
  UImanager->ApplyCommand(command+fileName);
 }
```

In the above listing, an object of class G4UIExecutive is initialized. The values ‘argc’ and ‘argv’ are special variables known to C++, referring to the number of arguments the executable is run with, and the arguments themselves (it is okay to give these typed values new names, but ‘argc’ and ‘argv’ are standard). These values must be defined as arguments of the main() function, specifically, ‘main(int argc, char** argv)’. If no arguments are passed to the executable, the default action is to start an interactive session. In this case, commands can be given to the executable directly from the terminal, during run-time. Alternately, if a ‘macro file’, or list of commands is passed as an argument to the executable, these will be executed in ‘batch mode’.

At this point, the executable should be recompiled. Once this has completed successfully, the program can be run, and a visualization file produced. The specific extension of this file will depend on what options Geant was built with. Commonly, Macintosh and Linux systems can produce a ‘.wrl’ file, which is viewed with the command ‘$ open name.wrl’, where ‘name’ corresponds to the number of existing events with corresponding ‘.wrl’ files in the executable directory. A simple test of these commands is try opening and writing to a (.wrl) file, through the following macro file (or, providing each line as a direct command in an interactive session):

```
/control/verbose 2 #setting verbosity level of control output
/run/verbose 2 #setting verbosity level of run output
/run/initialize
/vis/open VRML2FILE #opening a fresh VRML2FILE (of file extension .wrl)
/vis/verbose errors
/vis/drawVolume #automatically drawing world volume, daughter volumes
/vis/viewer/set/style wireframe #change visual representation
/vis/viewer/zoom 1.5
/vis/scene/add/trajectories smooth
/vis/scene/endOfEventAction accumulate
/vis/verbose warnings
/gun/particle mu+ # set primary particle type
/gun/energy 1 GeV # set primary particle energy
/run/beamOn 1 # firing one particle, should automatically write to the VRML2FILE
```

When opened, the resulting visualization file should clearly contain the World and Daughter Physical Volumes as defined in DetectorConstruction, and the paths of any primary and secondary particles. If this is not the case, try troubleshooting before proceeding.

Commands are roughly organized into directories by the type(s) of Geant4 classes they govern.

When inside of an interactive session, try typing '$ ls' to see a list of directories. To see the contents of an individual directory, type '$ ls /directory/'. Further discussion of commands can be found here: http://geant4.in2p3.fr/2007/prog/PaulGueye/Visualization2-Gueye.pdf.

Ultimately, there are three basic approaches for controlling Geant4 event simulations. The first, used up until this section, relies fully on the information compiled into the executable (calling BeamOn() through the runManager). The second uses the UIManager to set up an interface between the user and executable, allowing new commands to be provided in real-time. Finally, the batch mode allows the user to specify a pre-made file of commands to be executed. In general, the first method will be the least powerful in generating useful results. The second is ideal for debugging or performing basic tests of the simulation, and the third is likely the best option for harvesting statistically significant results.

VIII. EXTRACTING HIT RESULTS WITH SENSITIVE MATERIALS

A. Introducing Hits

At this point, the basics in defining a working simulation have been introduced. However, as in a real experiment, there is usually some designated sensitive device which observes the results of these interactions, and provides interpretable information for further analysis. In general, a model might have one or multiple sensitive detectors. When a (daughter or primary) particle crosses into such a volume, the state of the particle during this transportation interaction is organized as a 'Hit'. A Hit is another user-defined class (inheriting from G4VHit), which might include information like the position and momentum of the particle as it crosses into the sensitive material, the energy it deposits during its entrance, or its time of entrance. Specifically, each increment in space a particle traverses can be described as a Step (of the G4Step class). Each Step holds information about the differential distance traveled, differential energy lost, the time differential, the two defining G4StepPoints at the beginning and end of the step, etc. When a particle enters into a sensitive detector, these members of G4Step are accessed and organized into a Hit. As usual, the Hit class will have three required methods, methods for setting and getting Hit parameter values, and private member data. The Hit class header (SDHit.hh in the following example), also must introduce an object to store Hits per Event, an instance of G4THitsCollection:

```
#ifndef SDHit_h
#define SDHit_h 1
#include "G4VHit.hh"
#include "G4THitsCollection.hh"
#include "G4Allocator.hh"
#include "G4ThreeVector.hh"
#include "tls.hh"

class SDHit : public G4VHit
{
  public:
    // constructor with default arguments
    SDHit(const SDHit&); // constructor provided with arguments
    // destructor

    const SDHit& operator=(const SDHit&); // operators for handling the G4THitsCollection
    G4bool operator==(const SDHit&) const;
    inline void* operator new(size_t); // methods for operators
    inline void operator delete(void*);
    // optional virtual method for visualizing Hits, Draw()
    // optional virtual method for printing Hit results, Print()
    void SetTrackID (G4int track) { fTrackID = track; };
    // method for setting Hit position
    G4int GetTrackID() const { return fTrackID; };
    // method for getting Hit Position
  private:
      G4int fTrackID;
      // private member for Hit position
};
typedef G4THitsCollection<SDHit> HitsCollection;
extern G4ThreadLocal G4Allocator<SDHit>* HitAllocator;
inline void* SDHit::operator new(size_t)
{ if(!HitAllocator)
      HitAllocator = new G4Allocator<SDHit>;
  return (void *) HitAllocator->MallocSingle();
}
inline void SDHit::operator delete(void *sdhit)
{HitAllocator->FreeSingle((SDHit*) sdhit);}
#endif
```

Two definitions for the constructor are declared, a destructor, methods to set or update Hit parameters (here, track ID, or particle identifier, and particle position) and to retrieve these values, are named. The values themselves are also named and typed (private members). Some of the required and optional method declarations are missing, and would need to be added back in for use. Finally, there is a block of code introducing a G4THitsCollection object (to store Hits), and setting rules for how memory is to be allocated. Below is the corresponding implementation file:

```
#include "SDHit.hh"
#include "G4UnitsTable.hh"
#include <iomanip>
G4ThreadLocal G4Allocator<Hit>* HitAllocator=0;
SDHit::SDHit()
 : G4VHit(),
   fTrackID(-1),
   // fPos(G4ThreeVector())
{}
SDHit::~SDHit() {}
SDHit::SDHit(const SDHit& right)
  : G4VHit()
{
  fTrackID = right.fTrackID;
  // fPos = right.fPos;
}
const SDHit& SDHit::operator=(const SDHit& right)
{
  fTrackID = right.fTrackID;
  // fPos = right.fPos;
  return *this;
}
```

The first constructor (passed with no arguments), sets up the framework for a Hit, but does not set any physical values. When the constructor is called with an argument (of type Hit), the values are unpacked into the predefined member variables, fTrackID and dPos. Also consider how the Hit time or particle momentum might be implemented as member variables.

```
#include "G4VVisManager.hh"
#include "G4Circle.hh"
```

```
#include "G4Colour.hh"
#include "G4VisAttributes.hh"
G4bool SDHit::operator==(const SDHit& right) const
{ return ( this == &right ) ? true : false; }
void SDHit::Draw()
{
  G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
  if(pVVisManager)
  {
    G4Circle circle(fPos);
    circle.SetScreenSize(4.);
    circle.SetFillStyle(G4Circle::filled);
    G4Colour colour(1.,0.,0.);
    G4VisAttributes attribs(colour);
    circle.SetVisAttributes(attribs);
    pVVisManager->Draw(circle);
  }
}
```

Note the optional Draw() method above, which would allow the user to include spheres representative of Hit locations when the model is drawn. Another optional method, Print() may also be defined, but is not shown.

Versions of these scripts will be provided with this document for reference.

B. Specifying a Detector Volume

To extract Hits from an actual simulation, a specific logical volume must be designated as a 'Sensitive Detector' (within the DetectorConstruction class). To do this, a SensitiveDetector (SD) class must first be defined by the user. This class and its member functions will be organized into its own header and source file. As usual, this user-defined class inherits from a provided Geant class, G4VSensitiveDetector. It will have a public constructor, destructor, mandatory method for processing Hits, and an optional methods for initializing a collection of such hits:

```
#ifndef SD_h
#define SD_h 1
#include "G4VSensitiveDetector.hh"
#include "SDHit.hh"
```

```
#include <vector>
class G4Step;
class G4HCofThisEvent;
class SD : public G4VSensitiveDetector
{
  public:
    SD(const G4String& name,
                const G4String& hitsCollectionName);
    virtual ~SD();
    virtual void Initialize(G4HCofThisEvent* hitCollection);
    virtual G4bool ProcessHits(G4Step* step, G4TouchableHistory* history);
  private:
    SDHitsCollection* fHitsCollection;
};
#endif
```

The constructor for this file takes a HitsCollection as an argument:

```
SD::SD(const G4String& name, const G4String& hitsCollectionName) : G4VSensitiveDetector(name),
fHitsCollection(NULL)
{
  collectionName.insert(hitsCollectionName);
}
```

The destructor for this class takes no arguments and performs no actions. The Initialize() method is called at the beginning of each event with the relevant HitsCollection object. This allows the specific HitsCollection to be assigned to a particular G4Event, becoming the hit collection of the event (G4HCofThisEvent). This collection can be accessed later through the G4Event object. An example implementation is below:

```
void SD::Initialize(G4HCofThisEvent* hce)
{
 fHitsCollection = new HitsCollection(SensitiveDetectorName, collectionName[0]);
 G4int hcID = G4SDManager::GetSDMpointer()->GetCollectionID(collectionName[0]);
 hce->AddHitsCollection( hcID, fHitsCollection );
}
```

Finally, the mandatory ProcessHits() takes a G4Step object as argument, with the positional/geometric history of the particle, and checks whether any energy has been deposited when crossing into the Detector. If this is true, a new Hit object is created with track ID and position set, and is added in to the collection of Hits.

```
G4bool SD::ProcessHits(G4Step* aStep, G4TouchableHistory*)
{
  G4double edep = aStep->GetTotalEnergyDeposit();
  if (edep==0.) return false; // check energy deposit
  Hit* newHit = new Hit();
  newHit->SetTrackID (aStep->GetTrack()->GetTrackID());
  newHit->SetPos (aStep->GetPostStepPoint()->GetPosition());
  fHitsCollection->insert( newHit );
  return true;
}
```

While not listed in entirety here, it is recommended to the developer to reference the class descriptions for G4Step, and G4StepPoint online to understand the information accessible from these objects.

Finally, at least one Logical Volume from DetectorConstruction must be registered as the Detector. This is demonstrated for an example Logical Volume, “PMT”, below:

```
G4String PMT_Detector_name = “Example/PMT_Detector”;
SD* PMT_SD = new SD(PMT_Detector_name, “HitsCollection”);
G4SDManager::GetSDMpointer()->AddNewDetector(PMT_SD);
SetSensitiveDetector(“PMT”, PMT_SD, true);
```

After the necessary edits have been made, the executable should again be recompiled and tested.

C. Ideas on Retrieving Hit Information Practically

At this point, you should have a working simulation that well mimics the physics of your real, physical model, the ability to visualize any results, interact and control the simulation through use of macro files and the interactive mode, at least one Detector, and the ability to create and access Hit information. How do we save these results for analysis outside of Geant? Here are four suggestions, in order of sophistication:

1. Remove all verbose output, except for printing any important parameter value (e.g. all hit timestamps for a single event). When the executable is run, pipe this output into a new text file, where these results can be parsed later on.

2. Within a class source file or method, use C++ to open and write to a new text file. During execution, append to this file any relevant information. Consider how the name of this file could be changed at execution for different events.

3. Include the necessary ROOT libraries for generating histograms. Create any relevant histogram for the event, and save to a structured ROOT file.

4. Introduce a user-defined EventAction class (G4UserEventAction), with optional method 'EndOfEventAction(const G4Event* event)'. This method can access the final HitsCollection associated with the argument Event. Any of the above methods could then be implemented to retrieve data for analysis. Register this user-action with the ActionInitialization class (similarly to PrimaryActionGenerator).

Ultimately, it will be up to the user to use the functionality of Geant to produce useful results. These may include the distribution of photon arrival times at a PMT cathode, the response of the system to different primary particle species, how incidence angle changes the final pulse shape, the effect of altered PMT or scintillator geometry, and other possibilities.

Many of these questions can be answered easily with the worked example presented in this document, while others will rely on the ingenuity of the user... implementing the discussed classes in new ways, and investigating possible additional routes to better model and test the physics of some real, physical experiment.