Defence Research and Development Canada
Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

GEANT4 application on Radiation Exposure Device scenario at bus stop

Description of the application packages

Chuanlei Liu

Prepared by:
Calian Technologies Ltd.
340 Legget Drive, Suite 101, Ottawa, ON Canada K2K 1Y6

Project Manager: David Waller
Contract Number: PA11036
Contract Scientific Authority: David Waller

The scientific or technical validity of this Contract Report is entirely the responsibility of the contractor and the contents do not necessarily have the approval or endorsement of Defence R&D Canada.
.

Defence R&D Canada – Ottawa

Contract Report
DRDC Ottawa CR 2012-179
October 2012

Canada

GEANT4 application on Radiation Exposure Device scenario at bus stop

Description of the application packages

Chuanlei Liu

Prepared by:

Calian Technologies Ltd.
340 Legget Drive, Suite 101, Ottawa, ON Canada K2K 1Y6

Project Manager: David Waller
Contract Number: PA11036
Contract Scientific Authority: David Waller

The scientific or technical validity of this Contract Report is entirely the responsibility of the contractor and the contents do not necessarily have the approval or endorsement of Defence R&D Canada.

Defence R&D Canada – Ottawa

Contract Report

DRDC Ottawa CR 2012-179

October 2012

Principal Author

Chuanlei Liu

Approved by

J. Tremblay-Lutter
Head Capabilities for Asymmetric and Radiological Defence and Simulation Section

Approved for release by

C. McMillan
DRDC Ottawa Chief Scientist

© Her Majesty the Queen in Right of Canada as represented by the Minister of National Defence, 2012

© Sa Majesté la Reine (en droit du Canada), telle que représentée par le ministre de la Défense nationale, 2012

Abstract

The application of GEANT4 for simulating a Radiation Exposure Device (RED) scenario has been studied. The application tools for simulating the radioactive source, decay processes, radiation transport, interactions with passengers at a bus stop, and all the other supporting classes have been implemented and tested. To assist running jobs and analyzing data, several bash scripts and ROOT macros have been developed for this study. Beyond the RED application, the framework and tools show a promising extensibility to more complex applications.

Résumé

On a étudié l'utilisation de GEANT4 pour la simulation d'un scénario comprenant un appareil d'exposition. Les outils d'application servant à la simulation de la source radioactive, des processus de désintégration, du transport du rayonnement et des interactions avec les passagers à un arrêt d'autobus, de même que toutes les autres classes nécessaires, ont été implémentés et mis à l'essai. Pour faciliter l'exécution des tâches et l'analyse des données, on a composé plusieurs scripts Bash et macros ROOT pour la présente étude. Au-delà de la simulation d'un appareil d'exposition, ces outils et le cadre d'applications offrent une extensibilité prometteuse pour des applications plus complexes.

This page intentionally left blank.

Executive summary

GEANT4 application on Radiation Exposure Device scenario at bus stop

Chuanlei Liu; DRDC Ottawa CR 2012-179; Defence R&D Canada – Ottawa; October 2012.

Background: To improve the ongoing radiological risk assessment work being carried out at DRDC Ottawa, a new radiation simulator, the GEANT4 radiation transport simulation toolkit, has been successfully installed on the CARDS computing cluster as described in report[1]. Using that, a concrete application of GEANT4 has been developed and is described in this report.

Principal results: The application of GEANT4 for simulating a Radiation Exposure Device (RED) scenario at a bus stop has been studied in this report. A framework has been constructed, under which each component of the RED scenario (radioactive source, decay processes, radiation transport, and interactions with passengers at a bus stop) was simulated. The whole application package has been tested and the results are documented in report[2].

Significance of results: This work is the first attempt to apply GEANT4 for consequence (and risk) assessment at DRDC Ottawa. This specific application on a RED scenario not only provided a test ground to verify GEANT4's capability as a tool suitable for use at DRDC Ottawa, but also constructed a complete framework from which extended applications, on more complex RN scenarios, such as radiological dispersal devices can be realized.

Future work: In response to future needs, this framework, like all practical applications, will never be perfectly designed for all possible applications. Therefore, maintenance and development are needed to ensure maximum performance for future needs.

Sommaire

GEANT4 application on Radiation Exposure Device scenario at bus stop

Chuanlei Liu ; DRDC Ottawa CR 2012-179 ; R & D pour la défense Canada – Ottawa ; octobre 2012.

Contexte : Dans le but d'accroître la qualité des travaux continus d'évaluation des risques radiologiques à RDDC Ottawa, on a installé avec succès le nouveau simulateur de rayonnement, soit la trousse de simulation du transport du rayonnement GEANT4, dans la grappe d'ordinateurs CARDS, comme le d´écrit le rapport [1]. On a conçu une application concrète pour GEANT4, qui est d´ecrite dans le présent rapport.

Résultats : Le présent rapport porte sur l´étude de l'application de GEANT4 à la simulation d'un scénario comprenant un appareil d'exposition à un arrêt d'autobus. On a conçu un cadre d'applications qui a permis de simuler chacune des composantes du scénario d'appareil d'exposition (source radioactive, processus de désintégration, transport du rayonnement et interactions avec les passagers à un arrêt d'autobus). L'ensemble du progiciel d'application a été mis à l'essai, et les résultats ont été consignés dans le rapport [2].

Importance : Ces travaux constituent la première tentative d'utilisation de GEANT4 pour l´évaluation des conséquences (et des risques) à RDDC Ottawa. Cette utilisation pour un scénario comprenant un appareil d'exposition a permis non seulement de vérifier la capacité de GEANT4 à servir d'outil approprié pour RDDC Ottawa, mais également de construire un cadre d'applications complet qui permet de réaliser des applications étendues comprenant des scénarios radiologiques-nucléaires plus complexes, par exemple des simulations de dispositifs de dispersion radiologique.

Perspectives : En ce qui a trait aux besoins futurs, ce cadre d'applications, comme pour toute application pratique, ne sera jamais parfaitement adapté à toutes les utilisations possibles. Ainsi, des activités de maintenance et de développement seront nécessaires pour assurer un fonctionnement optimal pour les besoins futurs.

Table of contents

This page intentionally left blank.

1 Introduction

In this report, a new simulation method GEANT4 (GEometry ANd Tracking) [3] and its application to the RED (Radiation Exposure Device) scenario at a bus stop have been described. The primary intention of considering GEANT4 for a RED scenario is because of its extensibility and capability.

GEANT4 is a scientific toolkit for simulating the passage of particles through matter. Its application has been widely extended from high energy physics, to nuclear and accelerator physics, as well as medical and space science. In addition, GEANT4 is capable of simulating different particle beams (radionuclide sources), and has been used for simulating the world's most sophisticated detector systems at the Large Hadron Collider (LHC) and the most complex human phantom system. Furthermore, U.S. government's Defence Threat Reduction Agency supports the use of GEANT4 through its SWORD software which integrates with the Joint Semi-Automated Forces (JSAF) synthetic environment.

The RED study is basic yet practically important towards implementing more complex scenarios. Therefore, we start with this RED study to set up a framework for the GEANT4 application, and later move to more complex applications.

2 The *main* function

The *main* function in *redsim.cc* is responsible for the highest level organization of the whole program's functionality. It is the place where the program initializes toolkit tools, takes command line arguments and starts to execute.

Explicitly speaking, the *main* function is implemented by two toolkit management classes, *G4RunManager* and *G4UImanager*. The *G4RunManager* controls the running flow of the program, from class initialization to run initiation and to event loop control. The other management class *G4UImanager* is responsible for interfacing with user's commands on all levels of the program. Each of these will be explained in the following sections.

2.1 Initialize classes

The *main* function starts with initialization of several GEANT4 mandatory classes, as well as the user defined classes. As seen in Listing 1, after getting a handle (instance), *G4RunManager* starts to initialize classes such as *G4VUserDetectorConstruction*, *G4VUser PhysicsList* and *G4VUserPrimaryGeneratorAction*, which are the three derived classes the user must provide for any GEANT4 application. The remaining GEANT4 classes are defined for different purposes. Note that the *HistManager* class is the user defined class for analysis.

Listing 1: The mandatory initialization of classes in *main* function

```
int main(int argc, char** argv) // argc and argv for vis
{

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

  HistManager* histo = new HistManager;

  // ----------------------------------------
  // -- mandatory initialization classes
  // ----------------------------------------

  G4VUserPhysicsList *physics = new PhysicsList;
  runManager->SetUserInitialization(physics);

  // -- G4User
  RunAction* run_action = new RunAction(histo);
  runManager->SetUserAction(run_action);

  // -- detector
  DetectorConstruction *detector
        = new DetectorConstruction(histo, run_action);
  runManager->SetUserInitialization(detector);

  // -- source
  PrimaryGeneratorAction *gen_action
        = new PrimaryGeneratorAction(detector, histo);
  runManager->SetUserAction(gen_action);

  // -- EventAction
  G4UserEventAction* evt_action = new EventAction(histo, run_action);
  runManager->SetUserAction(evt_action);

  G4UserSteppingAction *step_action = new SteppingAction(histo);
  runManager->SetUserAction(step_action);
```

2.2 Apply arguments

The pre-configuration arguments/parameters can be read into the program at runtime by using the *ApplyCommand* method of *G4UImanager* class and the *execute* command. In this study, certain features of detector simulation and the histogram attributes are controlled by the input parameters in the configuration file *redsim.in*, as shown in Listing 2. More examples of using arguments are also available in Listing 3.

Listing 2: Apply arguments in *main* function

```
int main(int argc, char** argv) // argc and argv for vis
{
  ......

  // -----------------------------------------
  // -- Get pointer to the UI manager to feed in paras
  // -----------------------------------------
  G4UImanager *UI = G4UImanager::GetUIpointer();
  UI->ApplyCommand("/control/execute macros_for_run/redsim.in");

  // -----------------------------------------
  // -- Initialize the Geant4 kernel
  // -----------------------------------------
  detector->Initialize();
  histo->Initialize();

  runManager->Initialize();
```

2.3 Start a run

Various methods are available to start a run: hard-coded start, reading commands from a macro file or interactive mode. More details about these methods can be found in Section 2.9 of the GEANT4 User Guide Manual. Accordingly, the usage of these three methods are implemented in the *main* function and illustrated in Listing 3. The *BeamOn* method is the hard-coded command to execute the program. In the following case as *argc == 1*, the *ui→SessionStart()* is about to start an interactive session, while the last part is to read a macro file or other arguments by using *UI→ApplyCommand()*.

Listing 3: The startup methods in *main* function

```
int main(int argc, char** argv) // argc and argv for vis
{
  ......

  // -----------------------------------------
  // --          Start a run
  // -----------------------------------------
  // -- hard coded method
  //G4int numberOfEvent = 1;
  //runManager->BeamOn(numberOfEvent);

  // -- General case
  if (argc == 1) {// no input --> use interactive mode
    #ifdef G4UI_USE
      #ifdef G4VIS_USE
        G4String command = "/control/execute ";
```

```
17        G4String fileName = "macros_for_run/vis_example.mac"; //argv[1]
18        UI->ApplyCommand(command+fileName);
19      #endif
20      ui->SessionStart();
21      delete ui;
22    #endif
23  }
24  else{
25    G4String command = "/control/execute ";
26
27    // -- find correct passing arguments
28    G4String fileName;
29    G4int resetRunID  = 0;
30
31    for (G4int ar=0; ar<argc; ar++){
32      G4int nextarg = ar + 1;
33      G4String thisarg = argv[ar];
34      if ( thisarg == "-f" && ar+1 <argc ) fileName = argv[nextarg];
35      if ( thisarg == "-r" && ar+1 <argc )
36        resetRunID = std::atoi( argv[ar+1] );
37    }
38
39    run_action->InitializeRunID(resetRunID);
40    if (fileName == "")
41      G4cout <<" .... Please check/specify your arguments to main() !! "
42             << G4endl;
43
44    UI->ApplyCommand(command+fileName);
```

2.4 Save histograms

The last part of the *main* function is to save histograms by calling *writeAndsave()* method of the *HistManager* class. In fact, this step can be done at the end of a run (for example, in the *EndOfRunAction()* function of the *RunAction* class) if only one run is taken every time. More related to booking, filling and saving histograms is described in Section 3.8.

3 Implemented classes

A GEANT4 application such as the RED scenario can require most of tools built in GEANT4 package. These tools consist of classes of event management (EventAction and RunAction), primary particle simulation, detector simulation, physics processes, hit information, user analysis and visualization. In the following several sections, these implemented classes for the RED study are briefly described.

3.1 Event management classes

3.1.1 *EventAction* class

An event is the basic unit of the GEANT4 simulation. The *EventAction* class is defined in the GEANT4 to allow users to take various actions such as initializing variables at the beginning of simulating one event (*EventAction::BeginOfEventAction* function) and summarizing one event at the end of the simulation (*EventAction::EndOfEventAction* function).

There are two issues worthwhile being addressed here. The first one is about the trajectory container. One can obtain all trajectories of an event by using the pointer of the *G4TrajectoryContainer* class. However, this container is not filled by default. Therefore one needs to inform the program to save trajectories during simulation. In this study, one command line argument (*/tracking/storeTrajectory 1*) is issued for activating trajectory saving.

The second issue is about the event tree filling. For these kinds of variables to be filled at event basis, an *evtTree* is created at the initialization stage of the program and will be told to fill these variables in the *EventAction::EndOfEventAction* function. By contrast, the *runTree* is the other ROOT Tree structure to hold variables of interest at a run level.

3.1.2 *RunAction* class

A run consists of a sequence of events. GEANT4 implements a *RunAction* class to allow users to derive their own class, from which various information manipulation can be conducted at the run level. In this study, the *BeginOfRunAction* member function is used to set the run identification number, initialize histograms etc, while the *EndOfRunAction* function performs tasks such as run summary and histogram filling.

3.2 Source simulation: PrimaryGeneratorAction class

A concrete class of primary particle generation is mandatory for GEANT4 to initiate simulations. In our case, the class *PrimaryGeneratorAction*, derived from class *G4VUserPrimaryGeneratorAction*, is the concrete class to fulfil primary particle specification and simulation. The type of primary particle and its properties such as energy spectrum and spacial information can be defined in this concrete class or from the command line arguments. In addition, as the GEANT4 Manual says, an event interface is available to users to allow reading particles from an ASCII file created by other event generators.

The *G4GeneralParticleSource* (GPS) is the tool used to specify the incident particle source in this study. The type and properties of the primary particles are defined with command line arguments rather than the hard-code definition in the *PrimaryGeneratorAction* class. Note that a member function of the *PrimaryGeneratorAction* class called *BookThisSource()* was created to store particle attributes into histograms.

3.3 Passenger simulation

In the RED study, passengers at a bus stop are deemed to be the sensitive detectors/targets. Technically, the full property associated with each passenger is simulated and controlled by four basic steps, which are shape/size designing, positioning arrangement, sensitivity association and the runtime configuration. The methods to realize these steps are implemented in *DetectorConstruction*, *PersonParameterisation* and *DetectorMessenger* classes.

3.3.1 *DetectorConstruction* class

The *DetectorConstruction* class is the class derived from the *G4VUserDetectorConstruction* virtual class and is used to define and customize detector geometry, material and properties.

Listing 4: Part of *Initialize* function in the *DetectorConstruction* class

```
void DetectorConstruction::Initialize()
{
  histManager->SetNumberOfPerson( nperRep );

  // ------------------------------------------------------------
  // -- define world size, in RED, no need to be tuned during the run
  // ------------------------------------------------------------
  SetWorldSize(100.*m, 100.*m, 50.*m); // x, y, z

  // ------------------------------------------------------------
  // -- trash can (or mail box, or as a simple shelter) at the bus stop
  // ------------------------------------------------------------
  canInnerR             = 2*(0.20 - thickness/m)*m; // 0.20*m;
  canOuterR             = 2*0.20*m;  // (0.50 + thickness/m)*m;
  canHeight             = 1.20*m;
  canStartAng           = 0.0*deg;
  canSpanAg             = 360.0*deg;
  if ( canInnerR <=0.1*m ) {
    G4cout<<" reset the radius of trash bin! almost solid now! ";
    return;
  }

  // ------------------------------------------------------------
  // -- trash bin displacement (at origin, ease calculate distance)
  // ------------------------------------------------------------
  G4double CanDispX     = 0.*m;
  G4double CanDispY     = 0.*m;
  // -- z should be on the ground
  G4double CanDispZ     = WorldSizeZ/2.*(-1) + canHeight/2;

  SetDdrDisplacement(CanDispX, CanDispY, CanDispZ);

  // ------------------------------------------------------------
  // -- Bus stop shelter: size and displacement
```

```
35  // ------------------------------------------------------------
36  stopshelter_x          = stopSize.x(); //10.0*m;
37  stopshelter_y          = stopSize.y(); //10.0*m;
38  stopshelter_z          = 50.0*m; // equal to worldSizeZ
39
40  stopshDisplacementX    = 0.*m;
41  stopshDisplacementY    = 0.*m;
42  stopshDisplacementZ    = 0.*m;
```

The *Initialize* function shown in Listing 4 is used to
- Set the number of passengers to be simulated.
- Define the shape and size of the World volume. The World volume, defined as a box in this study, is required in GEANT4 to accommodate all detectors.
- Define the shape, size and displacement of the trash bin to place and shield the source
- Define the shape, size and displacement of the bus stop shelter to accommodate passengers. This additional volume definition is required by GEANT4 in order to have a dedicated mother space to parameterize detectors (passengers).

The other function, *Construct* in this class is the place to apply these pre-defined parameters on geometry and detector building up. In this study, the passenger simulation apparently is the most important part of detector simulation, and therefore a detailed description follows in the remaining part of this section.

Unlike general detector simulation in other fields, the passengers/targets in this study are randomly organized in terms of their sizes and locations in order to reflect a realistic scenario of passengers at the bus stop. This random feature correspondingly leads to two particular considerations when designing the program.

The first one is to declare local variables in the *Construct* function, rather than as private members in *DetectorConstruction*, to initialize the body size. This is because each individual size will be eventually changed in the parameterization process. The shape of passengers is built as a cylinder for simplicity, and the material to fill in each passenger is pure water. The second consideration is to use an instance of the *G4VPVParameterisation* class to carry out the actual parameterization process, as shown in Listing 5. More explanations on these two issues are given in the following *PersonParameterisation* class.

Listing 5: The parameterization code in the *Construct* function

```
1  G4VPVParameterisation* personPara = new PersonParameterisation(
2                                  ......);
3
4  // -- To place each replica (if Case three is using)
5  per_phys = new G4PVParameterised(
6                                  .......);
```

In the end of the *Construct* function, the parameterized passengers are assigned to be sensitive by calling *SetSensitiveDetector(apersonSD)* function by the *logical* volume of pas-

sengers. The argument in *SetSensitiveDetector(apersonSD)* is an object of the user defined class *personSD*. The *peronSD* class is used to record any interesting information about hits. The assignment of each passenger as a sensitive detector is given in Listing 6

Listing 6: Assigning passengers as sensitive in the *Construct* function

```
// ------------------------------------------
// -- add sensitive detectors here
// ------------------------------------------
// -- Get a handle of SDManager
G4SDManager* SDman    = G4SDManager::GetSDMpointer();

// -- Instantiate a new SD with name of "personName"
G4String personName   = "SD/personID";
PersonSD* apersonSD   =
              new PersonSD(personName, histManager, runAction);

// -- register the newly created SD into SDManager
SDman->AddNewDetector( apersonSD );

// -- assign a logical volume to this SD
per_log->SetSensitiveDetector( apersonSD );
```

However, note that it is not necessary to set the passengers as *sensitive* detectors to collect signal/hit information, because all this information can be alternatively obtained in the *tracking* or *stepping* action classes. More details about *peronSD* class can be found in section 3.5.

3.3.2 *PersonParameterisation* class

The *PersonParameterisation* class contains variable members and methods to realize the random parameterization of passenger in terms of physical and positioning features.

Listing 7: The first part of the *PersonParameterisation* constructor

```
// --
CheckCapacity();
// --
histManager->de_person_n = fNoPerson;
```

The *CheckCapacity* function shown above in Listing 7 is used to check the personnel capacity (*noPerson*) given a specific size of mother volume (*motherV*). This check is intended to ensure a low passenger density so that the positioning overlap of passengers is avoided and manageable by the program. In the case of an over load, the number of people will be forcedly changed to the maximum capacity defined by this function, along with a warning message (or exit action: not implemented yet). Therefore the variable *de_person_n* in the

ROOT files always records the actual number of people simulated since it is filled *after* the *CheckCapacity* function.

Listing 8: The second part of the *PersonParameterisation* constructor

```
// --
CLHEP::HepRandom::setTheSeed((unsigned)time(0));
```

The code in Listing 8 is to set the random seed according to the local computer time. If nothing was specified here, the initial random seed would always be the same for different *runs*, which would end up with same passenger simulation for all *runs*.

The mechanism to parameterize a detector is implemented in two pre-defined methods, the *ComputerTransformation* and *ComputerDimensions* functions, derived from the base class *G4VPVParameterisation*. The former is to transform a physical volume identified by its copy number to its mother volume, while the latter is to set the physical size of that copy. These two methods must be created in case of the parameterization is needed, and the arguments of these two methods are fixed.

Listing 9: The third part of the *PersonParameterisation* constructor

```
// --
physPerson    = new G4ThreeVector[fNoPerson];
solidPerson   = new G4ThreeVector[fNoPerson];

ComputeTransformation(
      const G4int copyNo, G4VPhysicalVolume* physVol) const
{......}

ComputeDimensions( G4Tubs &iPerson,
      const G4int copyNo, const G4VPhysicalVolume* physVol) const
{......}
```

These two methods can be called many times during a run, depending on how active each copy of the detector is. For example, as long as one passenger is hit, these two methods will be called to compute its location and size, according to its copy number. This kind of mechanism however gives rise to trouble when the user would like the random parameterization being realized *inside* these two methods because these detector features are not expected to change time to time after its initialization. Therefore, one work-around is to pre-compute these features in the *PersonParameterisation* constructor, and later pass them into these two methods. This is the reason why two *ThreeVector* variables, *physPerson* and *solidPerson* are defined in Listing 9 (the first consideration mentioned in the previous section).

Listing 10: The map container in the *PersonParameterisation* constructor

```
map<double, double> peopleLocations;
```

The map defined above in Listing 10 stores a pair of double precision variables for the passenger positions on the x and y axes. The reason for having a map container is to have only the closest *parameterised* passengers (by using lower_bound and upper_bound functions of map container) available to compute the distance to the current parameterised passenger. This is needed to check the position overlap (more checks are needed).

Listing 11: Parameterizing passenger size in the *PersonParameterisation* constructor

```
  // -- start to compute parameterization values
  for (G4int i=0; i<fNoPerson; i++){
    // ------------------------------------------------------
    // -- solid (shape/size) parameterization
    // ------------------------------------------------------
    G4double iInnRad = 0.*m;
    G4double iOutRad = GausRandom( detector->GetPersonWidth() );
    G4double iHeight = GausRandom( detector->GetPersonHeight());

    // -- save this solid for later use
    G4ThreeVector Vsolid(iInnRad, iOutRad, iHeight);
    solidPerson[i] = Vsolid;

    FillPhysicalHist(i, Vsolid);
```

The piece of code in Listing 11 is to start to parameterize the physical size of each passenger. As mentioned before, the shape of each passenger is taken as a solid cylinder (the inner radius is zero). The width (*iOutRad* variable) is computed randomly from a Gaussian distribution (*GausRandom* function) configured in the user input file, same as the height (*iHeight* variable). The newly constructed passenger size is afterwards stored in the *solidPerson* vector and written out to Ntuples by calling the *FillPhysicalHist* function. The positioning parameterization is in principle configured the same way as the size parameterization.

The conditional statement below in Listing 12 is used to decide between running in a mode with or without inter-person shielding effects.

Listing 12: Inter-personal shielding effect in the *PersonParameterisation* constructor

```
if ( detector->GetInterShield() ) {
} else {
}
```

3.3.3 *DetectorMessenger* class

The messenger interface currently defined in terms of detector construction is shown below in Listing 13. The corresponding commands, their meaning and the default values are given in Table 1.

Listing 13: Runtime parameters

```
    // -- Number of person to be simulated
    G4UIcmdWithAnInteger*       perNumCmd;

    // -- if the src to be shielded
    G4UIcmdWithABool*           srcSldCmd;

    // -- if the inter-person shield effect exists
    G4UIcmdWithABool*           interSldCmd;

    // -- if shielded with iron, what is thickness
    G4UIcmdWithADoubleAndUnit*  thicknessCmd;

    // -- Three dimension of the bus stop
    G4UIcmdWith3VectorAndUnit*  stopSizeCmd;

    // -- Three parameters for height (1/2 height, width and cutoff )
    G4UIcmdWith3Vector*         perHeightCmd;

    // -- Three parameters for width
    G4UIcmdWith3Vector*         perWidthCmd;
```

The configuration parameters defined in Table 1 turn out to be very useful and flexible in simulating different scenarios. For example, one can easily activate or disable the source or inter-person shield effect to study the impact. The study of the transverse distance impact can also be easily simulated by adjusting the size of the person height, from the unit area case to body trunk ($\sim$ 0.8*m), or even to the whole body size.

Table 1: Parameters used to configure detector construction.

parameter name	parameter type	default values	parameter meaning
/det/noperson	G4int	200 (30)	No. of passengers to be simulated in case of with (without) inter-person shielding effect.
/det/srcshield	G4bool	true	Is the source shielded?
/det/thickness	G4double	0.4 cm	If srcshield is true, how thick is the shielding material (iron bin)?
/det/intershield	G4bool	true	Is the inter-person shielding effect simulated?
/det/stopsize	G4ThreeVector	in meter	The 3-dimensional size of bus stop
/det/perheight	G4ThreeVector	0.4 0.1 2	The half height is 0.4 cm, Gaussian width is 0.4*0.1 cm and cutoff value is 0.4*0.1*2 away from 0.4
/det/perwidth	G4ThreeVector	0.2 0.05 2	Similar meaning as perheight

3.4 Physics processes: PhysicsList class

The user can decide and assign a specific physics process which a particle will undergo as well as the order of processes in case of multiple choices. The standard processes such as electromagnetic, hadron and transport processes are available in GEANT4 and one just needs to add these to a particle. For the RED or other radionuclide studies, the important process is the radioactive decay, which is not a standard process in HEP. Therefore, one needs to get the external libraries (PhotonEvaporation and RadioactiveDecay) and set the radionuclide to undergo this particular process. In principle, the RadioactiveDecay library take cares of the first step of decay of the radionuclide, and the PhotoEvaporation take cares of the cascade decay afterwards.

The way associating the RadioactiveDecay with a heavy nuclide physics process is implemented in the *ConstructRadDecay* function, as shown in Listing 14.

Listing 14: Associating RadioactiveDecay with a heavy Ion

```
#include "G4RadioactiveDecayPhysics.hh"
#include "G4RadioactiveDecay.hh"
#include "G4IonTable.hh"

void PhysicsList::ConstructRadDecay()
{
  // -- Add radiation Process
  G4RadioactiveDecay* therad = new G4RadioactiveDecay();
  therad->SetHLThreshold(-1.*s);
  theParticleIterator->reset();

  const G4IonTable* theIonTable =
        G4ParticleTable::GetParticleTable()->GetIonTable();

  for (G4int i=0; i<theIonTable->Entries(); i++){
    G4String particleName =
        theIonTable->GetParticle(i)->GetParticleName();
    if (particleName == "GenericIon"){
        G4ProcessManager* pmanager =
                theIonTable->GetParticle(i)->GetProcessManager();
        pmanager->AddProcess(therad);
```

3.5 Hit information

As explained in Section 3.3, simulated passengers are assigned to be sensitive. Here sensitive means one can derive its own concrete class from *G4VHit* so as to collect information of hits, and afterwards store them into a template class of hit collection for later use. These operations are implemented in *PersonHit* class. The other class *PersonSD*, a sensitive detector representative, is created primarily for constructing hit objects using the information from steps along a particle track.

Different from the *SteppingAction* class described below, these two hit classes are used to store information coming solely from the interaction with the human body while the *SteppingAction* class is for general information of tracks.

3.5.1 *PersonHit* class

As shown in the *PersonHit* class, information like *particle name*, *track ID*, *energy deposit* and so on are defined to characterize a hit. A template class *personHitsCollection* is defined in the header file of *PersonHit* class, as shown in Listing 15. As its name implies, the *personHitsCollection* is used as a hit object container. The instantiation of this class is done in the initialization step of *PersonSD* class, shown in Listing 16 and is filled with concrete hits in the *ProcessHits* method of *PersonSD* class, as shown in Listing 17.

Listing 15: HitsCollection definition

```
typedef G4THitsCollection<PersonHit>  personHitsCollection;
```

Listing 16: HitsCollection instantiation

```
void PersonSD::Initialize(G4HCofThisEvent* HCE)
{
 ......
 hitCollection =
    new personHitsCollection(SensitiveDetectorName, collectionName[0]);
 ......
}
```

Listing 17: HitsCollection instantiation

```
G4bool PersonSD::ProcessHits(G4Step* aStep, G4TouchableHistory*)
{
 ......
 hitCollection->insert( newHit );
 ......
}
```

3.5.2 *PersonSD* class

The *PersonSD* class was created to represent a sensitive detector. This class is mainly used to construct hit objects in the *ProcessHits* method, store these hits as shown above in Lists 16 and 17, and invoke user actions at the end of an event to manage hit collection.

The *BookTheseHits* method is called at the end of an event to accumulate hit information stored in *hitCollection* class, as shown in Listing 18. The energy deposit quantity, used for dose rate calculations, is computed in this step.

Listing 18: Accumulate hit information for an event

```
// ----------------------------------------
void PersonSD::BookTheseHits()
{
 ......
  // ----------------------------------------
  // -- to accumulate hits per person in this event
  // ----------------------------------------
  //
  if ( iperson_tot_nhits > 0 ) {
    for ( G4int i=0; i<iperson_tot_nhits; i++ ) {
      G4int person_id              = (*hitCollection)[i]->GetPersonID();

      iperson_tot_nstep[person_id] ++;
      iperson_tot_edep[person_id]  += (*hitCollection)[i]->GetEdep();
      iperson_tot_steplen[person_id]+=
                (*hitCollection)[i]->GetStepLen()/cm;
 ......
}
```

3.6 Stepping information

GEANT4 provides two interface classes to access particle information at a specific moment of track evolution. These two classes are *G4TrackingManager* and *G4SteppingManager*, which are closely related to each other. The corresponding user action classes, *G4UserSteppingAction* and *G4UserTrackingAction*, are available to users for specific information access or particular quantity computations. Users can use either instance of these two classes (derived class) to handle particle and stepping information. In this study, the *SteppingAction* and *SteppingMessenger* classes are implemented to meet our needs.

3.6.1 *SteppingAction* class

The *SteppingAction* class is instantiated only once before initializing the GEANT4 kernel. Therefore, initialization of variables declared in *SteppingAction* class should be done somewhere else if variables need to be reset event-by-event. One way is to create a member function, for example *BeginOfEvent*, and call this function in the *BeginOfEvent* function in the *EventAction* class. By doing this, all data members inside *BeginOfEvent* of *SteppingAction* class will be initialized on event-by-event basis. Or in the current implementation, variables can be reset when both track ID and Step ID equal to 1 (implying a new event just starts.), as shown in Listing 19.

Listing 19: Initializing variables at event base in *SteppingAction*

```
 if ( fTrk_id == 1 && fTrk_step_id == 1 )
 {
```

```
    trk_index   = 0;  histManager->SetTrkNo(0);
    decay_index= 0;  histManager->SetDecayNo(0);
```

The messenger variable *debugLevel* defined below is used to control the level of the debug information printout. In principle, each class can implement its own debug control variable as being done here. The debug messenger *debugmsg* is created to format the printout of debug message. The *debugmsg* function is defined at the beginning of the code, as Listing 21 shows.

Listing 20: debug message in *SteppingAction*

```
if ( debugLevel > 0)
{
      debugmsg(theLifetime);
```

Listing 21: Predefined *debugcode* in *SteppingAction*

```
#define debugmsg(x) G4cout<< " \t...In SteppingAction, "
                           << #x <<" : " << x <<G4endl;
```

Additionally in *SteppingAction*, several methods have been defined to process specific particle information. As listed in Listing 22, the process type (Photoelectric, Compton, etc.) will be returned after calling *GetParentProcType* or *GetDaughterProcType*. The number of decay products of a particle, especially the radioactive nuclide in this study, can be obtained by calling *GetNoOfDaughters*. For a certain radionuclide, the activity per gram is also easily accessible. For debugging purpose, the decay chain of the radionuclide can be specifically printed out by calling the *PrintDecayChain* function. Even detailed information can be available by invoking *UserSteppingInfo*.

Listing 22: Methods defined in *SteppingAction*

```
    G4int GetParentProcType(const G4Step*);
    G4int GetDaughterProcType(const G4Step*);
    G4int GetNoOfDaughters( const G4Step*);
    G4double GetActivityPerGram(const G4Step*);

    void PrintDecayChain(const G4Step*);
    void UserSteppingInfo(const G4Step*);
```

The other important issue addressed in the *SteppingAction* class is to store the information of interest into Ntuples. Three types of particles are classified based on whether or not the particle hits passengers, or if it is the initial heavy nuclide or non-intermidate light particles. The way to separate these type of particles and how to fill Ntuples are shown in Listing 23

Listing 23: Information booking in *SteppingAction*

```
  // --------------------------------------
```

```
  // -- book these trks if found in the sensitive detector
  // ------------------------------------
  if ( fTrack->GetNextVolume()->GetLogicalVolume()
                              ->GetSensitiveDetector() != 0 )
  {
    // -- should at before trk_index++ step.
    BookThisTrack(fStep, trk_index);
    trk_index++;
  } else {
    // -- book these light particles (nProton <=3 )
    if ( fTrk_step_id == 1 && fTrk_pdg_chg <= 3.
                           && GetParentProcType(fStep) != 2 )
    {
      BookThisTrack(fStep, trk_index);
      trk_index++;
    }

    // -- book these nucleus (nProton > 3)
    if (fTrk_pdg_chg > 3. && GetDaughterProcType(fStep) != 2 )
    {
      if (debugLevel >0 ) PrintDecayChain(fStep);
      BookDecayProduct(fStep);
      BookThisTrack(fStep, trk_index);
      trk_index++;
    }
  }
```

All selected particles/tracks will be saved by calling *BookThisTrack* function, and the dedicated function *BookDecayProduct* will save these products of radionuclide decay.

3.6.2 *SteppingMessenger* class

As described in the previous section, the variable *debugLevel* corresponds to the user command */step/debug*, controlling the level of debug information printing. In table 2, the usage and meaning of this variable is described.

Table 2: Parameters used to control debug information printing in *SteppingAction*.

parameter name	parameter type	value range	parameter meaning
/step/debug	G4int	$>= 0$	value 0 means no specific debug info printing; value 4 means every info user designs to check

3.7 User Interface: Messenger classes

This section corresponds to the Communication and Control chapter in the GEANT4 Manual [4], mainly describing the built-in User Interface (UI). For the mechanism and syntax of the command, please refer to the GEANT4 manual. For this study, the implemented command and the meaning of UI are described in each individual class.

3.8 User Analysis classes

The User Analysis class is usually needed for the purpose of manipulating data computation, extraction and saving to Ntuples. In this study, two particular classes, *HistManager* and *HistManagerMessenger*, are implemented for these purposes.

3.8.1 *HistManager* class

The *Initialize* function defined in the *HistManager* class contains the declaration of ROOT histograms and Trees. These objects are created to keep the information about the radioactive source, detector simulation, hit/track information and the user-derived quantities such as the dose rate. Note that, in the *HistManager* header file, there are several *SetValue* and *GetValue* methods to manipulate member variables in the *HistManager* class. For simplicity, many data members are defined as public type so that these can be called by the instance of the class. However, as easily understood, public member bears the risk to be modified unexpectedly.

In the *HistManager* constructor as shown in Listing 24, two ROOT Tree objects *evtTree* and *runTree* have been created and initialized. The reason to have these two Tree structures is because of the different tree filling stages. All contents in *evtTree* will be booked and saved on the event basis, while the *runTree* is on the run basis. The *hList* instance here is defined to register all histograms, branches and trees so that the collection of these objects can be written out to the Ntuple at once by using *hList*. The *addToList* function shown in Listing 25 is designed for registration purpose, and *hList→Write()* is to write the list into the ROOT file.

Listing 24: HistManager constructor

```
1   evtTree = new TTree("evtTree","Event Tree");
2   runTree = new TTree("runTree","Run Tree");
3   hList   = new TList();       // list of histograms/branches to store
```

Listing 25: The *addToList* function

```
1 void HistManager::addToList(TObject* add_this)
2 {
3   if ( add_this->IsA()->InheritsFrom(TH1::Class()) ) {
```

```
    hList->Add( (TH1*)add_this );
  }
  else if ( add_this->IsA()->InheritsFrom(TH2::Class()) ) {
    hList->Add( (TH2*)add_this );
  }
  else if ( add_this->IsA()->InheritsFrom(TTree::Class()) ) {
    hList->Add( (TTree*)add_this );
  } else {
    G4cout<<"  Error: the TObject to be book is NOT defined! "<< G4endl;
  }
}
```

The relevant function in the *HistManager* class is the *writeAndsave* method as shown in Listing 26, in which the name of the ROOT file has been finalized and the ROOT objects collected by *hList* are ready to write out to the ROOT file. The ROOT file closes in the end. Note that this method is called at the end of the *main* function.

Listing 26: The *writeAndsave* function

```
void HistManager::writeAndsave()
{
  NtupleName += ".root";
  char* theName = new char[NtupleName.length() + 1 ];

  strcpy(theName, NtupleName.c_str());
  fName = new TFile(theName, "RECREATE");
  hList->Write();
  fName->Close();
}
```

3.8.2 *HistManagerMessenger* class

The *HistManagerMessenger* class was created mainly to control the content and size of the ROOT file by reducing the ROOT Tree branches. The dimension of the Tree branches can also be configured at runtime by using command line arguments. The description of these parameters are given in Table 3, along with other features of parameters.

Table 3: Parameters used to configure Histogram attributes.

parameter name	parameter type	default values	parameter meaning
/hist/lightwgt	G4bool	true	if choping the size of the ROOT file
/hist/noperson	G4int	200	the maximum number of person to be booked (not yet functioned)
/hist/notrk	G4int	200	the maximum number of tracks to be booked (not yet functioned)

4 Integration of ROOT into GEANT4

As one likes to work on a platform where ROOT has been successfully installed, two different ways can be explored to run *root*. One is in an interactive mode by typing *root*, while the second way is to integrate ROOT into the executable of the other program. In either case, the ROOT environment variables must be set up properly. You might have something similar to what Listing 27 shows to set up these environment variables.

Listing 27: Set environment variables for ROOT

```
export ROOTSYS=/usr/local/ROOT/root
export PATH=$ROOTSYS/bin:$PATH
export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
```

In order to integrate ROOT into the GEANT4 framework, the following lines have to be placed in the GNUmakefile, which is used to compile or recompile GEANT4. Basically, all header files should be linked and the ROOT library is configured platform-independently if using *root-config*.

Listing 28: Include ROOT lib and head file in makefile

```
  CPPFLAGS += -I$(ROOTSYS)/include
  EXTRALIBS = $(shell root-config --glibs)
```

Then it is straightforward to use ROOT objects by declaring corresponding header files beforehand. A good example of how to use ROOT objects can be found in *HistManager.hh*.

5 Summary and future plans

A concrete application of the GEANT4 simulation toolkit has been implemented for a Radiation Exposure Device scenario at a bus stop. The structure of the application, as well as the implemented classes are described in this report. The runtime configuration of many relevant parameters makes it easier to check different simulation scenarios without re-compilation. The bash scripts and ROOT macro associated with this work are also helpful to make jobs run semi-automatically.

To go beyond this initial RED study, more complex radiation exposure scenarios can be simulated using GEANT4 and the framework developed here.

References

[1] C. Liu, Installation of GEANT4 toolkit at CARDS computer cluster, **DRDC Ottawa CR 2012-178**, (2012)

[2] C. Liu and D. Waller, GEANT4 simulation on Radiation Exposure Device scenario at bus stop, **DRDC Ottawa TM 2012-180**, (2012)

[3] S. Agostinelli et al., Nuclear Instruments and Methods in Physics Research, **A 506**, 250-303, (2002)

[4] Geant4 User's Guide for Application Developers, Version 9.4 (2011)

[5] Online Nuclear Calculations: http://www.radprocalculator.com/

[6] Evaluated Nuclear Structure Data File and Decay Data: http://wwwndc.jaea.go.jp/nucldata/ensdf.html/

Annex A: Bash scripts and ROOT macros

This appendix describes all scripts and ROOT macros which are currently developed and used in both this study and the following RED analysis work [2]. The functionality of these scripts and macros are briefly explained in the following sections, with an emphasis on the key components to realize the functionality. A overview of these scripts and macros is given in Listing 29.

Listing 29: List of scripts and ROOT macros

```
00-setup-geant4.sh                    drawRad.C
0-a-compile_and_run                   GNUmakefile
0-b-run_many_jobs                     include
0-c-submit-jobs.sh                    log.txt
1-b-merge_ntuples_simple_way          macros_for_plot
1-c-loop-events                       macros_for_run
2-b-comb_plots_for_reports            null
2-b-draw_plots_for_reports            README
2-b-stability_plots_for_reports       redsim.cc
3-extract-combine-plots-for-reports   results
3-extract-plots-for-reports           RunDetail.log
combine.C                             src
currentEvent.rndm                     stability.C
```

A.1 Compilation and run: *scripts with prefix 0-**

A.1.1 Set up GEANT4 environment: *00-setup-geant4.sh*

This is the first thing to do if working with GEANT4, and the *00-setup-geant4.sh* script is written for GEANT4 environment setup purpose.

A.1.2 Compile and run: *0-a-compile_and_run*

The script *0-a-compile_and_run* is created for auto-compilation purpose after code modification, and then running a sainty check by calling configuration file *./macros_for_run/vis_plain.mac*. The simulation result will be analyzed by calling macro *plot.C* under *macros_for_plot* directory. In the end, a postscript file called *sanity_check.ps* will be open for review.

A.1.3 Launch many runs at one machine: *0-b-run_many_jobs*

After a success compilation of the application, one might want to generate bunch of events, rather than the small amount of test events produced in the previous section. For this purpose, the script *0-b-run_many_jobs* was created to launch as many events and runs as the user wants. With this script, the user can additionally tune many parameters such as the run

ID and number of runs to be launched, the radionuclide ID, the name tags of the generated ROOT files, choice of input configuration files, if using a specific CPU core, if running in parallel with commands “parallel” or “mpirun”.

A.1.4 Submit jobs to the cluster: *0-c-submit-jobs.sh*

To fully utilize the local computing resource at cluster, one can submit jobs into all slave nodes available. The submission step can be realized by using either “mpirun”, “screen” or “nohup” command. Currently in this work, command “nohup” is used for submission. Two portable functions are available in this script, one is used for slave node accessibility check while the other for CPU core check.

A.2 Managing ROOT files: *scripts with prefix 1-**

As job completes, simulation results are stored into many ROOT files. Regarding the method of analyzing multiple ROOT files, two different ways are described in the following.

A.2.1 Merge ROOT files into one: *1-b-merge_ntuples_simple_way*

This method is the simple way to merge ROOT files with histograms by using ROOT predefined macro *hadd.C* and command *MergeRootfile*. Note that this merging step also works well for ROOT files with Trees, however the dependencies between Branches of the Tree might be lost after merging. Therefore, the proper way to handle ROOT files with Trees should be like something like what described in the next section.

A.2.2 Chain ROOT files and do event loop: *1-c-loop-events*

This method is not implemented yet, however, the basic idea is like this: By using *Add* function of TChain class of ROOT, one can make a collection of Tree objects from different ROOT files. After that, one can start loop events as what is illustrated in web (http://root.cern.ch/drupal/content/ use-chains-list-files). One might also want to use the Tree method *MakeClass* or *MakeSelector* to generate a skeleton analysis class to proceed event loop.

A.3 Analyze results and produce plots: *scripts with prefix 2-* and 3-**

The scripts with prefix “2-*” are used to produce postscript files to store all histograms, while these with prefix “3-*” for extract individual histogram from those postscript files for the report inclusion.

A.3.1 Result for default scenario: *2-b-draw_plots_for_reports*

This is the script for generating all histograms except the comparison ones. Basically, one can pass a ROOT file to this script, and then produce a postscript file containing much information about one simulation scenario. The "drawRad.C" ROOT macro is used in this step.

A.3.2 Comparison results between scenarios: *2-b-comb_plots_for_reports*

This is the script to perform the comparison between different simulation scenarios, and then plot each individual result together into one plot for comparison purpose. The ROOT macro "combine.C" is used in this step.

A.3.3 Stability results for all runs: *2-b-stability_plots_for_reports*

This script is used to perform run-by-run stability check on the dose rate result.

A.3.4 Extract plots for the report: *3-extract-plots-for-reports*

This script is written for automatic plot extraction for the report use. It basically extracts each specific plot from a postscript file, and then pass it into the source code of the report latex file. The realization of the extraction is done with Linux command "psselect".

A.3.5 Extract the comparison plots for the report: *3-extract-combine-plots-for-reports*

The script works the same way as the script described in the previous section, however is used to extract the comparison results for the report.

This page intentionally left blank.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Calian Technologies Ltd.
340 Legget Drive, Suite 101, Ottawa, ON Canada K2K 1Y6

2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.)

UNCLASSIFIED
(NON_CONTROLLED GOODS)
DMC:A
REVIEW: GCEC June 2010

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

GEANT4 application on Radiation Exposure Device scenario at bus stop: Description of the application packages

4. AUTHORS (Last name, followed by initials – ranks, titles, etc. not to be used.)

Liu, C.

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total containing information. Include Annexes, Appendices, etc.)	6b. NO. OF REFS (Total cited in document.)
October 2012	38	6

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Contract Report

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence R&D Canada – Ottawa
3701 Carling Avenue, Ottawa, Ontario, Canada K1A 0Z4

9a. PROJECT NO. (The applicable research and development project number under which the document was written. Please specify whether project or grant.)	9b. GRANT OR CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
	PA11036

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC Ottawa CR 2012-179	

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

(X) Unlimited distribution
( ) Defence departments and defence contractors; further distribution only as approved
( ) Defence departments and Canadian defence contractors; further distribution only as approved
( ) Government departments and agencies; further distribution only as approved
( ) Defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11)) is possible, a wider announcement audience may be selected.)

Unlimited

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

The application of GEANT4 for simulating a Radiation Exposure Device (RED) scenario has been studied. The application tools for simulating the radioactive source, decay processes, radiation transport, interactions with passengers at a bus stop, and all the other supporting classes have been implemented and tested. To assist running jobs and analyzing data, several bash scripts and ROOT macros have been developed for this study. Beyond the RED application, the framework and tools show a promising extensibility to more complex applications.

On a étudié l'utilisation de GEANT4 pour la simulation d'un scénario comprenant un appareil d'exposition. Les outils d'application servant à la simulation de la source radioactive, des processus de désintégration, du transport du rayonnement et des interactions avec les passagers à un arrêt d'autobus, de même que toutes les autres classes nécessaires, ont été implémentés et mis à l'essai. Pour faciliter l'exécution des tâches et l'analyse des données, on a composé plusieurs scripts Bash et macros ROOT pour la présente étude. Au-delà de la simulation d'un appareil d'exposition, ces outils et le cadre d'applications offrent une extensibilité prometteuse pour des applications plus complexes.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

GEANT4 application
Radiation Exposure Device scenario
Radionuclide decay
Radiation transport
Passenger simulation
ROOT analysis tool
bash script and ROOT macro