HepEmShow a compact EM shower simulation

Mihály Novák
CERN EP-SFT

November 7, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

HepEmShow is an application for simulating electromagnetic (EM) shower in a configurable simplified sampling calorimeter. While the EM shower is simulated by the same algorithm and physics used today for detector simulation in High Energy Physics (HEP), including the Large Hadron Collider (LHC) experiments such as ATLAS or CMS, the entire simulation is kept very lightweight and simple in order to enhance the clarity and transparency of the underlying computing flow and algorithms.

The Geant4 [1][2][3] simulation toolkit is the main workhorse for such simulations today due to the flexibility provided by its carefully design interfaces, powerful geometry description, navigation and comprehensive physics modelling capabilities. While from the applications point of view these properties are advantageous (e.g. making possible to handle even the most complex simulation problems some of which listed above), they might cause considerable difficulties when the goal is to identify and investigate the underlying computing flow and algorithms. This is due to the simple fact that a relatively large fraction of the Geant4 codebase is involved even in a simple simulation application due to the generic internal framework of the toolkit. It already requires substantial effort and time to become familiar enough with the toolkit for application development but even more effort and significant expertise are need when the goal is to identify and understand the underlying computing flow and algorithms. This can cause difficulties or even pose barriers especially when non-simulation experts would like to take some Geant4 based simulation as the application target of their own technology, new techniques, solution, etc.. Moreover, applying the new technology to the complete, final simulation target usually has significant cost and risk (see above) that can be largely reduced by preceding with a feasibility study on a simpler, more compact but realistic simulation, i.e. having the key algorithmic properties identical to the native Geant4 based simulation.

HepEmShow was developed exactly with this goal in mind: to provide a compact, Geant4 like but significantly simpler simulation application with only the required components but all available locally. More details on the main components of the simulation application can be found in *The components of the simulation* section.

CHAPTER

TWO

BUILD AND INSTALL

Will come later but HepEmShow requires G4HepEm that provides its Physics (see more at the *The components of the simulation*). G4HepEm can be built with or without Geant4 though a data file is needed in the latter case. See more later . . .

2.1 Build G4HepEm with or without Geant4 dependence:

2.1.1 With Geant4 (G4HepEm_GEANT4_BUILD=ON default):

```
cmake ../ -DGeant4_DIR=YOUR_GEANT4_INSTALL/lib(lib64)/cmake/Geant4/ -DCMAKE_INSTALL_
↪PREFIX=YOUR_G4HEPEM_INSTALL
```

2.1.2 Without Geant4 (G4HepEm_GEANT4_BUILD=OFF):

```
cmake ../ -DG4HepEm_GEANT4_BUILD=OFF -DCMAKE_INSTALL_PREFIX=YOUR_G4HEPEM_INSTALL
```

2.2 Build HepEmShow with G4HepEm build with or without Geant4:

2.2.1 With a complete, Geant4 dependent G4HepEm build:

```
cmake ../ -DGeant4_DIR=YOUR_GEANT4_INSTALL/lib(lib64)/cmake/Geant4/ -DG4HepEm_DIR=YOUR_
↪G4HEPEM_INSTALL/lib(lib64)/cmake/G4HepEm/ -DCMAKE_BUILD_TYPE=RELEASE
```

2.2.2 With a standalone, Geant4 independent G4HepEm build:

```
cmake ../ -DG4HepEm_DIR=YOUR_G4HEPEM_INSTALL/lib(lib64)/cmake/G4HepEm/ -DCMAKE_BUILD_
↪TYPE=RELEASE
```

2.3 Requirements

2.4 Build and install

CHAPTER

THREE

THE COMPONENTS OF THE SIMULATION

The two main components of the simulation is the *Geometry* and *Physics* providing the necessary information and functionalities fused together to compute the individual simulation steps during the *Event processing* inside the *Stepping loop*, i.e. how far the particle goes in a given simulation step and what happens with it at that post-step point.

The *Geometry* describes the simulation setup, including navigation possibilities or providing information on the **geometrical constraints for the simulation step computation** (e.g. given a global point, how far the nearest volume boundary is or how far the volume boundary to a given direction is, etc.). The *Physics* component is responsible to supply the **physics related constraints for the simulation step computation** (e.g. how far the given particle go till its next interaction of the given type or how far a charge particle goes till it loses all its current energy, etc.). These constraints are required during the *Event processing* at the beginning of each simulation step computation in the *Stepping loop* to decide length of the simulation step. The particle is then moved to the post-step point and the necessary *Geometry* and/or *Physics* related **actions are performed** on the particle. Some information regarding the given simulation step, e.g. energy deposit, might be collected at the end of each simulation step before performing the computation of the next step till the history of the particle is terminated (e.g. e^- lost all its kinetic energy in the last step; γ was absorbed in photoelectric process or destructed by producing e^-/e^+ pair, etc.).

A short description of the main components of the `HepEmShow` simulation application, such as the *Geometry*, *Physics*, *Event processing* and *Stepping loop*, is provided in this section together with *The HepEmShow application main* and some example configuration.

3.1 Geometry

The application `Geometry` is a configurable simplified sampling `calorimeter` that is built up from `N` `layer`-s of an `absorber` and a `gap` as illustrated in Fig. 3.1.

The number of `layer`-s `N`, the thickness of both the `absorber` and `gap` can be set at the construction of the calorimeter (see the *Input arguments* of *The HepEmShow application main*). All thicknesses are measured along the `x`-axis in `mm` units.

Note: The `gap` thickness can be set even to zero in which case the `calorimeter` is built up from the given number of `layer`-s of `absorber` with the given thickness (i.e. a single material calorimeter sliced by the `layer`-s) as illustrated in Fig. 3.2.

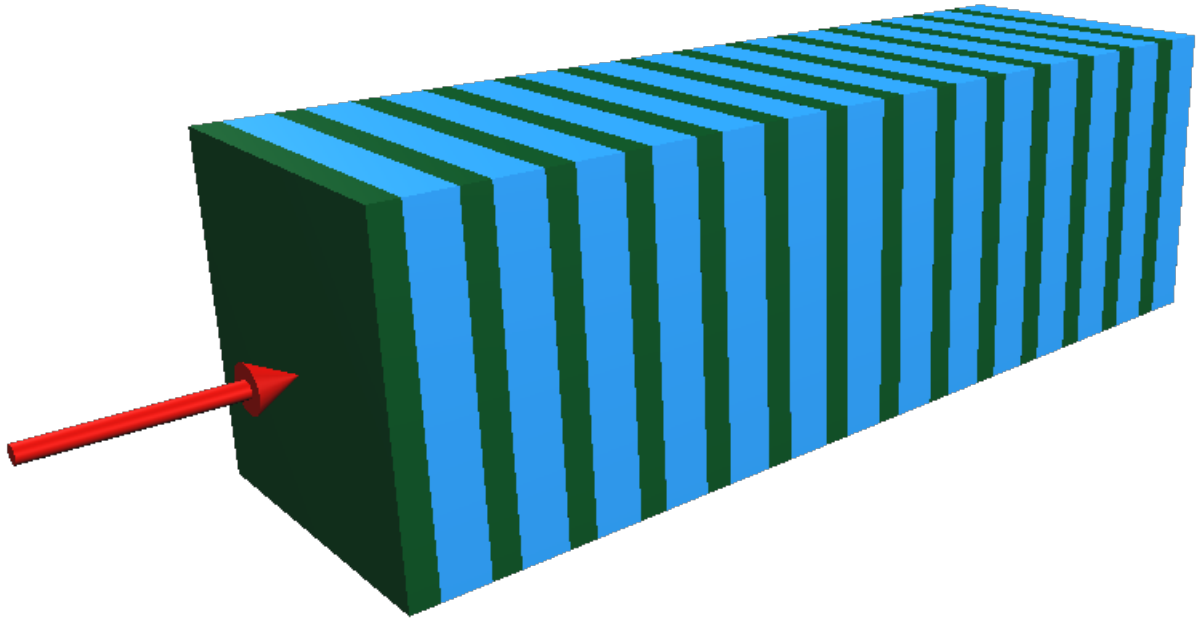

Fig. 3.1: Illustration of the default `absorber` (green) and `gap` (blue) `layer` structured simplified sampling calorimeter configuration (with `N = 14` `layer`-s in this case). The red arrow from left represents the direction of the incoming primary particle beam.

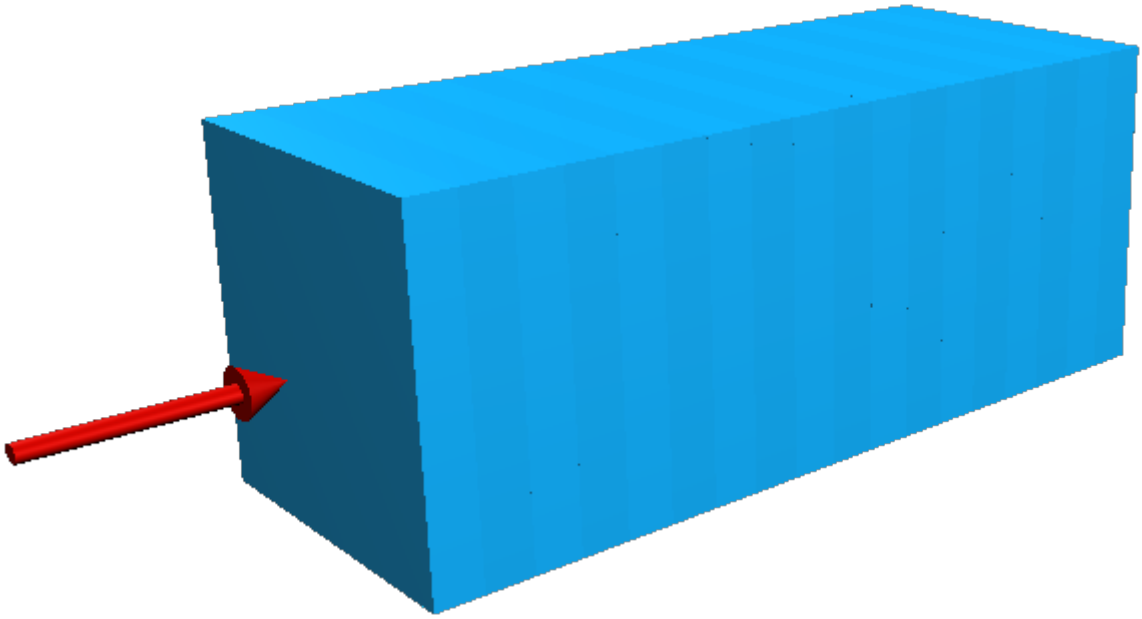

Fig. 3.2: Illustration of the single material calorimeter sliced by the `layer`-s (`N = 18`). As the `gap` thickness is set to `0`, the single `layer` thickness is identical to the `absorber` (blue) thickness while the `calorimeter` thickness is $\times$ `N` of that.

The thickness of the single `layer` as well as the entire `calorimeter` is automatically computed from the given

absorber, gap thicknesses and number of layer-s respectively. The transverse size, i.e. the full extent of the absorber, gap, layer and the calorimeter along the y- and z-axes, can also be set as an input argument (see the *Input arguments* of *The HepEmShow application main*). The calorimeter is places in the center of the world so 5 (*Box* shaped) volumes are used in total to describe the application geometry. The size of the world volume is computed automatically based on the extent of the calorimeter such that it encloses the entire calorimeter with some margin.

Note: The x-coordinate of the left hand side boundary of the calorimeter as well as an appropriate initial x-coordinate of the primary particles (such that they are located mid-way between the calorimeter and the world boundaries along the negative x-axis) are calculated. At the beginning of an event, the *PrimaryGenerator* will generate primary particles at this later position with the [1,0,0] direction vector, i.e. pointing toward to the left hand side boundary of the calorimeter. Each primary particle is then moved to the former position, i.e. on the left hand side boundary of the calorimeter, in their very first step such that they will enter into the calorimeter in the next, second simulation step.

Each of these volumes is filled with a given material specified by the material index of the volume. These indices are the subscripts of the (Geant4 predefined NIST) materials listed in the material name vector of *The HepEmShow-DataGeneration application main*. In the default case, which is the same that was used to generate the G4HepEm data shipped with the application, this material name vector is

Listing 3.1: The material name vector as it is in *The HepEmShow-DataGeneration application main*.

```
// list of Geant4 (NIST) material names
std::vector<std::string> matList {"G4_Galactic", "G4_PbWO4", "G4_lAr"};
```

This corresponds to the default *index-to-material* and eventually to the *material-to-volume* association shown in Table 3.1. A complete list of the predefined NIST materials provided by Geant4 with their composition can be found at the corresponding part of the Geant4 documentation (Book For Application Developers: Geant4 Material Database).

Table 3.1: The default material to index and material to volume association.

Material	Index	Used in Volume
galactic (low density gas)	0	layer calorimeter world
lead tungstate (atolzite)	1	absorber
liquid-argon	2	gap

Note: Changing the material name(s) in this above vector of the *The HepEmShow-DataGeneration application main* (especially at index 1 and/or 2 as the vacuum is always needed to fill the layer, calorimeter and world container volumes), regenerating the data by executing this data generation application, then executing again the HepEmShow application, corresponds to changing the material of the absorber and/or gap volumes of the simulation.

The application geometry also provides a rather simple "navigation" capability (used in the simulation stepping loops) through its *Geometry::CalculateDistanceToOut()* method described in details at the corresponding code documentation.

Attention: Unlike the Geant4 geometry modeller and navigation, that provides generic geometry description and navigation capabilities, the *Geometry* implemented for HepEmShow is specific to the configurable simplified sampling calorimeter described above. Focusing only to this specific geometry modelling and related navigation problem made possible to provide a rather compact, simple and clear implementation of all geometry related functionalities required during the simulation (i.e. in the *SteppingLoop*).

3.2 Physics

Targeting only the simulation of the EM shower inherently leads to a compact simulation as it includes only e^-/e^+ and γ particles with their EM (i.e. without gamma- and lepto-nuclear) interactions. Focusing to the descriptions of these interactions, that ensures sufficient details and accuracy for HEP detector simulations, leads to an even more specific set of interactions and underlying models that the physics component of the simulation needs to provide. This well defined, important but small subset of the very rich physics offered by the Geant4 toolkit, can then be implemented in a very compact form.

The G4HepEm R&D project [4] offers such an implementation with several attractive properties. Separation of data definition, initialisation and run-time functionalities results in a rather small, Geant4 independent, stateless, header based implementation of all physics related run-time functionalities required for such EM shower simulations. Furthermore, all the data, extracted from Geant4 during the initialisation, can be exported/imported into/from a single file making possible to skip the Geant4 dependent initialisation phase in subsequent executions of the application. Therefore, G4HepEm offers the possibility of a Geant4 like but Geant4 independent EM physics component for developing particle transport simulations. Further information on G4HepEm, including the physics interactions included, can be found in the corresponding part of the G4HepEm documentation.

Note: The hepemshow repository includes the pre-generated data file (/data/hepem_data.json) that has been extracted by using the HepEmShow-DataGeneration with the default material configuration settings. Providing this data file makes possible to initialise the G4HepEm data component from this file making HepEmShow independent from Geant4. The HepEmShow-DataGeneration application is also available in the hepemshow repository. This can be used to re-generate the above data file when the goal is to change the default material configuration (see above at the *Geometry* section). However, as the data extraction requires the Geant4 dependent initialisation of G4HepEm, it requires a Geant4 dependent build of G4HepEm. See more details in the Requirements subsection of the Build and Install section.

As mentioned above, the entire physics of the HepEmShow simulation application is provided by G4HepEm [4]. The required, definitions (.hh files) of the G4HepEm run-time functionalities are pulled in by the Physics.hh header while the corresponding implementations (.icc files) are in the Physics.cc. The only missing implementation, that the client needs to provide, is a uniform random number generator that needs to be utilised to complete the implementation of the G4HepEmRandomEngine. This is also done in the Physics.cc file by using the local *URandom* uniform random number generator. More information can be found in the code documentation of the *Physics*.

G4HepEm provides two top level methods, HowFar and Perform in its G4HepEmGammaManager and G4HepEmElectronManager for γ and e^-/e^+ particles respectively:

- HowFar: the physics constrained step length of the given input track, i.e. how far the particle goes e.g. till the next physics interaction takes place or it loses all its kinetic energy or due to any other physics related constraints.
- Perform: performs all necessary physics related actions and updates on the given input track, including the production of secondary tracks in the given physics interaction (if any).

These two top level methods are utilised in the *Stepping loop* during the computation of the individual simulation step. HowFar is invoked at the pre-step point, i.e. at the step limit evaluation, while Perform is utilised at the post-step point of each individual simulation step computation inside the *SteppingLoop::GammaStepper()* and *SteppingLoop::ElectronStepper()* methods.

Attention: Unlike the *Geometry* of the application, the *Physics* is fully generic as G4HepEm provides an application independent, generic EM physics component similarly to the corresponding native Geant4 implementation. However, the Geant4 dependent initialisation phase of G4HepEm, i.e. the data extraction, has been eliminated from HepEmShow by separating it to the additional HepEmShow-DataGeneration application in order to make HepEmShow independent from Geant4. As a consequence, the corresponding generated data file is specific to a

given material configuration and needs to be re-generated whenever one would like to change that material configuration as discussed above.

3.3 Event processing

3.3.1 Event loop

3.3.2 Stepping loop

3.4 The `HepEmShow` application main

3.4.1 Input arguments

3.5 The `HepEmShow-DataGeneration` application main

3.6 Example configurations

CHAPTER

FOUR

CODE DOCUMENTATION

4.1 Code documentation of the `Simulation` part of the `HepEmShow` application.

4.1.1 The main of the `HepEmShow` application

The main funtion of the `HepEmShow` application.

Author
M. Novak

Date
July 2023

The main of the `HepEmShow` application is responsible for setting up the environment, lunch the simulation and write the results. This is done by:

- reading the input arguments provided at the execution of the application into an *InputParameters* object. (Note, these arguments provide configuration options).
- loading the `G4HepEm` data and parameters from file into a `G4HepEmState` (see the note below)
- constructing a `G4HepEmTLData` (also required by `G4HepEm` and encapsulates the random number generator and some track buffers) with its random number generator (utilising the local *URandom* generator)
- constructing and setting up the application *Geometry* according to the provided configuration input arguments (in *InputParameters*)
- constructing and setting up a *Results* structure that will be used to collect some data during the simulation
- constructing and setting up the *PrimaryGenerator* of the application according to the provided configuration input arguments (in *InputParameters*)
- the *EventLoop::ProcessEvents* method is invoked then to **perform the simulation**
- the simulation results are witten to file (and to the standard output) by invoking *WriteResults()* (from the *Results*)

Note: The `G4HepEm` data file is either the one included in the `HepEmShow` repository (under `hepemshow/data/`) or generated by the auxiliary `HepEmShow-DataGeneration` application. In the former case, the data file contains all data

that G4HepEm needs for the simulation for the 3 default ({"G4_Galactic", "G4_PbWO4", "G4_lAr"}) materials, i.e. those used in the default *Geometry* configuration.

Functions

int **main**(int argc, char *argv[])

The main function of the HepEmShow simulation application (see more in the description).

4.1.2 The Geometry description related code documentation

The *Geometry* of this application is built up from 5 *Box* objects (by default, i.e. at non-zero gap thickness): an absorber and a gap building up a layer that is repeated N times along the x-axis building up and filling in the calorimeter that is centered at the origin and placed in the world.

class **Geometry**

Geometry description for this simple simulation setup.

Author
M. Novak

Date
July 2023

The simulation setup is a **configurable simplified sampling calorimeter** built up from N layers of an absorber and a gap (both by default). The number of layers N, the thickness of both the absorber and gap along the x-axes can be set and changed dynamically.

- layer:
  - number : fNumLayers
  - set/get : *SetNumLayers(int)*/*GetNumLayers()*
  - thickness: fLayerThick (calculated automatically from the absorber and gap thicknesses)
- absorber:
  - thickness: fAbsThick
  - set/get : *SetAbsThick(double)*/*GetAbsThick()*
  - material : lead tungstate/atolzite ("G4_PbWO4") with material index = 1 (by default)
- gap:
  - thickness: fGapThick
  - set/get : *SetGapThick(double)*/*GetGapThick()*
  - material : liquid argon ("G4_lAr") with material index = 2 (by default)

All thickness measured along the x axes while the yz extent is the same both for the absorber and gap determined by the fCaloSizeYZ which can be set dynamically by SetCaloSizeYZ(val)/GetCaloSizeYZ().

Note: The default length unit is [mm] so all thicknesses and sizes are assumed to be give in [mm] units.

Note: The `gap` thickness can be set even to zero in which case the `calorimeter` is built up from the given number of `layer`s of `absorber` with the given thickness (i.e. a single material calorimeter sliced by the `layer`s).

Note: The material indices are determined by the order of the corresponding `Geant4` (predefined NIST) material names listed in the material name vector of the data extraction application (i.e. in `DataGeneration.cc`). This application is used beforehand to extract the material (and cuts) dependent data required during the simulation. The default vector in `DataGeneration.cc`, that was used to extract the provided data files, is

```
// list of Geant4 (NIST) material names
std::vector<std::string> matList {"G4_Galactic", "G4_PbWO4", "G4_lAr"};
```

hence the above `material - index` mapping. Changing the material name(s) in this above vector (especially at index 1 and/or 2 as the vacuum is always needed to fill the container volumes like the `layer`, `calorimeter` or the `world` volumes), regenerating the data, then executing the same `Simulation` application, corresponds to changing the mateiral (with the given index) in the simulation. A list of the available predefined Geant4 NIST material names can be found in the corresponding part of the Geant4 Documentation.

A single layer is composed from the above `absorber` and `gap` while the entire `calorimeter` is built up form the given number of identical `layer`s shifted along the `x` axes. The `calorimeter` center, i.e. the [0,0,0] position of the corresponding *Box* shape local coordinate, is at the global origin (i.e. no translation nor rotation is applied). The entire calorimeter is placed inside the `world` that is the limit of our simulation universe. The `layer`, `calorimeter` and `world` is filled with vacuum (very low density hydrogen), so only the `absorber` and the `gap` have non-vacuum like materials.

The shape of all objects (`absorber`, `gap`, `layer`, `calorimeter`, `world`) is *Box*. A box object is constructed for each in the constructor by setting the appropriate name and material index fields. Their proper sizes are calculated and updated automatically whenever one of the above setters, affecting any of the thicknesses or sizes, is invoked.

The geometry can also provide an apropriate initial `x` position for the primary particles locating in between the `world` and the `calorimeter` on the left hand side (*GetPrimaryXposition()*). The `x` position, where the `calorimeter` starts on the left hand side, can also be obtained (*GetCaloStartXposition()*). These method will always give an appropriate value as the corresponding data are also updated dynamically whenever any of the thicknesses or sizes are modified.

The geometry also provides a very simple "navigation" through its *CalculateDistanceToOut(double*, double*, Box**, int*, int*)* method that determines:

- the (deepest) volume/box in which the given global `position` is located
- the index of the layer (only if the point is located inside the `calorimeter`)
- and the index of the absorber: 0 for the `absorber` and 1 for the `gap` (but only in case the point is inside the calorimeter)

At the end, it returns:

- a large (1E+20 [mm]) value whenever the `position` and `direction` is such that the particle is about leaving the `calorimeter` (i.e. going to vacuum then would hit the boundary of the `world` at the end of that step)

- otherwise: the distance to the boundary of the volume in which the given `position` was located: from the given `position` along the given `direction`.

Note, that this distance is computed by using the corresponding box/volume method (namely *Box::DistanceToOut(double*,double*) const*) and the `box` object in which the given `position` was located:

- distance to out is 0 if a given `position` is outside of that volume/box
- volume boundaries, closer to `position` than half of the *Box::kCarTolerance* (= 1E-9 [mm]), are ignored when the direction is pointing out, i.e. these positions are considered to be outside, leading to 0 distance to the volume boundary from inside. (see more on `inside`, `surface` and `outside` at the *Box* documentation)

A small (e.g. 1E-6 [mm]) push along the current direction might be applied in the steppers when the distance to out was found to be 0.

This is because it's assumed, that the 0 distance is due to the employed simple location calculation, that ignors the tolerance, instead of using an apropriate navigator. Namely, the point was located in volume A. but it's actually on its `surface`: closer than *Box::kCarTolerance*/2 to its boundary. Morover, the direction is pointing toward to the next, volume B., that is just on the other side of this boundary. This correctly gives 0 distance to the boundary of volume A as actually the given step will be done in volume B. Therefore, the small push is to overcome the volume boundary, i.e. to push the point to be on the other side of the boundary between volume A and B. Then, the point is calulated to be in volume B now when relocating and as the direction is pointing inside volume B, the expected distance to the next boundary of volume B is computed.

Public Functions

Geometry()

Constructor: sets the default configuration, creates the *Box* objects for all components.

~Geometry()

Destructor: deletes the *Box* objects that represents the volume of the components.

inline void **SetNumLayers**(int nlayers)

Sets the number of layers the entire calorimeter should be built up.

param nlayers
[in] Number of layers (must be > 0) requested (all parameters are recalculated).

inline int **GetNumLayers**() const

Gives the number of layers the calorimeter is built up.

return
number of layers.

inline double **GetCaloThick**() const

Gives the thickness of the `calorimeter` (i.e.

full size along the x-axis).

return
thickness of the `calorimeter` in [mm] units.

inline void **SetAbsThick**(double thickness)

Sets the required absorber thickness (i.e.

full size along the x-axis).

param thickness
[in] Required thickness of the absorber in [mm].

inline double **GetAbsThick**() const

Gives the thickness of the absorber (i.e.

full size along the x-axis).

return
thickness of the absorber in [mm] units.

inline void **SetGapThick**(double thickness)

Sets the required gap thickness (i.e.

full size along the x-axis).

Note, that the gap thickness can also be set to zero. The calorimeter is built up from a single material layers, i.e. a block of material sliced along the x-axis.

param thickness
[in] Required thickness of the gap in [mm] (can be set to 0).

inline double **GetGapThick**() const

Gives the thickness of the gap (i.e.

full size along the x-axis).

return
thickness of the gap in [mm] units.

inline void **SetCaloSizeYZ**(double val)

Sets the transverse size (i.e.

full size along the yz-axes).

Note, this also determines yz sizes of the corresponding the absorber, gap and layer volumes/shapes.

param val
[in] Required full transvers size of the calorimeter in [mm].

inline double **GetCaloSizeYZ**() const

Gives the transverse size of the calorimeter (i.e.

full size along the yz-axis).

return
full transverse size of the calorimeter volume/shape in [mm] units.

inline double **GetPrimaryXposition**() const

Provides the x-coordinate of the mid-position between the world and calorimeter boundaries on the left hand side.

Note that this is only for the primary generator, the primary tracks should be inside the calorimeter or its boundary but pointing inside (see more at *CalculateDistanceToOut()*)

return
the x-coordinate of the mid-point between the world and calorimeter boundaries on the left.

inline double **GetCaloStartXposition**() const

Provides the x-coordinate on the `calorimeter` boundary on the left hand side.

Note this is the initial x-coordinate of each primary track while their direction should point toward the calorimeter (i.e. having positive x-coordiante).

return
the x-coordinate of the `calorimeter` boundary on the left hand side.

double **CalculateDistanceToOut**(double *r, double *v, *Box* **currentVolume, int *indxLayer, int *indxAbs)

Locates a point in the geometry and calculates the distance till the next boundary.

This method is supposed to be called at the pre-step point of the simulation step with the global pre-step ponit coordinates and actual direction in order to:

- determine the volume in which this simulation step will be done (and more importantly, the material as everything depends on that (at least))
- the distance to the boundary of that volume along the given diretion (as the material might change on the other side of that boundary)

The pre-step point is supposed to be inside the `calorimeter` volume, i.e. either

- `inside`: deeper than *Box::kCarTolerance*/2 form any of its boundaries
- on `surface`: closer to a boundary than *Box::kCarTolerance*/2 While the distance to the `calorimeter` volume boundary is > zero in the first case, this depends on the actual direction in the second case:
- zero: when the direction is pointing outside of that boundary, i.e. particle is about leaving the volume
- positive: as the particle is about moving in the volume otherwise These are true for all (and not only for the `calorimeter`) volumes!

During the simulation, each primary track starts from the `calorimeter` volume boundary with a direction that is pointing inside, i.e. ensured to be inside (on `surface` but pointing in). All tracks are terminated when the particle is about leaving the `calorimeter`, i.e. the particle is on `surface` and pointing out (as the step would be done in the vaccum ending on the boundary of our `world`). Therefore, all step pints, and secondary tracks created at some of these points, are also ensured to be inside the `calorimeter` (as defined above).

In order to achive the above, this method returns with a large (1E+20 [mm]) distance to boundary whenever the particle is about leaving the `calorimeter`. The point (the step) is located to be in the `world` volume (`layer` and `absorber` indices are set to -1). Otherwise, the point is located, i.e. the deepest volume inside the `calorimeter` in which the point is located, is determined, the `layer` and `absorber` indices are set.

However, this is done based on a simply computation of the `layer` index (based on its thickness) then the same within the layer. In other words, this is done without considering the tolerance or the direction (unlike in Geant4, having a robust but complex navigator for this). Therefore, it might be the case that the point is calculated to be inside a given volume but actually it's on the `surface` while moving out. The corresponding simulation step should actually be performed in the next volume (that is just on the other side of that boundary). This is detected during the simulation step computation, as this method returns zero distance in this case, and:

- a small push of 1E-6 [mm] is applied along the current direction (just to push the point to the other side of the boundary)
- this method is called again with the new position: calculated to be in the good volume now

The input position, given in global coordinates, always transformed to the local system of the given volume. This is an identity transformation for the `calorimeter` (as not translated nor rotated). Then for the `layer`, the translation vector (having non-zero only its x-component) is determined based on the first layer x-position (where the `calorimeter` starts), the thickness of the `layer` and the current x-position of the point. After transforming the point to `layer` local coordinates, the position is transformed further either to `absorber` or `gap` local coorinates depending on which the point was calculated to be in. At the end, the input position vector contains the position of the point in the local system of the volume/shape that the pint was calculated to be located. Therefore, this local coorinates can be used later directly in any shape (*Box*) methods, e.g. for computing the safety.

param r
[inout] pointer to a 3D array that stores the x, y and z coorinates of the position in global coordiantes at input. These will be updated to be local coordinates in the system of the volume (*Box*) in which the point was located in.

param v
[in] pointer to a 3D array that stores the x, y and z coorinates of the current normalised direction vector

param currentVolume
[inout] address of a pointer to a *Box* object that can be anything at input, while at output the corresponding pointer is set to the `Box` object that represents the volume in which the given point was calculated to be located

param indxLayer
[inout] pointer to an integer that can be anything at input, while at output it will be the indx of tha `layer` in which the given point was calculated to be located (-1 when the particle is leaving the `calorimeter` or no layers)

param indxAbs
[inout] pointer to an integer that can be anything at input, while at output it will be 0 or 1 that corresponds to the `absorber` and `gap` depending on which the given point was calculated to be located (-1 when the particle is leaving the `calorimeter` or no layers)

return
the distance, from the given position along the given direction, to the boundary of the volume in which the given point was calculated to be located. It might be zero (the step actually shouldn't be done in the located volume) or 1E+20 [mm] (the particle about leaving the `calorimeter`).

Private Functions

void **`UpdateParameters`**()

Privite method that clculates the apropriate positions and volume/shape sizes whever any related parameters is updated.

Private Members

int **fNumLayers**

Number of layers the calorimeter is built up.

Can be set (even to zero: calorimeter is just a single volume/box)

double **fAbsThick**

`Absorber` thickness measured along the `x`-axis in [mm] (can be set)

double **fGapThick**

`Gap` thickness measured along the `x`-axis in [mm] (can be set; even to zero: single material `calorimeter` sliced by layers along the `x`-axis)

double **fLayerThick**

`Layer` thickness measured along the `x` axes in [mm].

Computed automatically (whenever the `absorber` or `gap` thickness is updated

double **fCaloThick**

The thickness of the entire `calorimeter` measured along the `x` axes in [mm] Computed automatically whenever the `layer` thickness (i.e.

absorber and/or gap thickness) or number of layer is updated.

double **fCaloSizeYZ**

The transverse size (i.e.

full size along the `yz` axes) of the `calorimeter` in [mm] units (same for `asborber`, `gap` and `layer` volumes/shapes)

double **fCaloStartX**

The `x`-coordinate of the `calorimeter` boundary on the left hand size.

Calculated automatically (whenever the related parameters are updated)

double **fPrimaryXPosition**

The `x`-coordinate of the mid-point between the `calorimeter` and `world` boundaries on the left hand size.

Calculated automatically (whenever the related parameters are updated)

Box ***fBoxWorld**

Pointer to the *Box* shape representing the `world` volume.

Box ***fBoxCalo**

Pointer to the *Box* shape representing the `calorimeter` volume.

Box ***fBoxLayer**

Pointer to the *Box* shape representing the `layer` volume.

Box ***fBoxAbs**

Pointer to the *Box* shape representing the `absorber` volume.

Box ***fBoxGap**

Pointer to the *Box* shape representing the `gap` volume.

class **Box**

A simplified version of the G4Box shape.

Author
M. Novak

Date
July 2023

This is a simple version of the G4Box shape to describe geometry objects and use them in the simulation to calculate distance to their boundary from a point inside. Note, that the calculations include a tolerance, e.g. a point is on the surface if closer to a boundary than 1/2 tolerance (kCarTolerance). The two most important methods, used during this simplified simulation, are:

- DistanceToOut(position, direction): distance to boundary from a local position (inside the box) along the given direction. The boundary is ignored if the position is closer to it than 1/2 tolerance (i.e. point is on the surface). The distance to boundary is zero in this case whenever the direction is pointing outisde (i.e. the particle is moving away/out).
- DistanceToOut(postition): this is the `safety`, i.e. the distance to the nearest boundary from the given local point inside (zero if on the surface or outside).

This version of the *Box* stores an index to the material that fills the shape (therefore closer to the Geant4 `logical` volume concept than to a shape).

Box shapes are constructed for each geometry object in the *Geometry* and the above methods are utilised during the simulation step computation in the `GammaStepper` and `ElectronStepper` (in some case indirectly by calling `Geometry::DistanceToOut` that first locates the point, i.e. finds the *Box* object which the given global point is located in).

NOTE: a point given in local coordiantes can locate

- `inside` : if deeper inside than `kCarTolerance/2` from any boundary
- `surface` : if within `kCarTolerance/2` from any boundary (in- or outside)
- `outside` : if further away than `kCarTolerance/2` from any boundary outside

NOTE: distance to volume boundary from a point along a given direction is zero when the point is not `inside` and the direction is pointing away. Therefore, a point located on the surface gives distance to boundary:

- zero : if the direction is pointing outside of that boundary
- non-zero: if the direction is pointing inside of that boundary

Public Functions

Box(const std::string &name, int indxMat, double pX, double pY, double pZ)

Constructor.

param name
[in] Name of this volume.

param indxMat
[in] Index of the material this volume is filled with.

param pX
[in] Half length of the box along the x-axis.

param pY
[in] Half length of the box along the y-axis.

param pZ
[in] Half length of the box along the z-axis.

inline **~Box**()

Destructor (nothig to do)

inline const std::string &**GetName**() const

Get the name of this volume.

inline void **SetMaterialIndx**(int indx)

Set the material this volume is filled with.

param indx
[in] Index of the material.

inline int **GetMaterialIndx**() const

Get the material this volume is filled with.

return
Index of the material.

void **SetHalfLength**(double val, int idx)

Set the half length of the box along the given axis.

param val
[in] Half length in [mm] units.

param idx
[in] Encodes the axis along the half length is given (idx=0 –> x; idx=1 –> y; idx=2 –> z).

double **GetHalfLength**(int idx) const

Get the half length of the box along the given axis.

param idx
[in] Encodes the axis along the half length is required (idx=0 –> x; idx=1 –> y; idx=2 –> z).

return
Half length of this box along the required axis.

double **DistanceToOut**(double *r, double *v) const

Calculates distance to the volume boundary from inside along the given direction.

Returns the distance along the normalised direction vector `v` to the volume boundary, from the given point `p` inside or on the surface of the box. Intersections with surfaces, when the point is within half tolerance (`kCarTolerance/2`) from a surface, is ignored.

param r
[in] 3D position of the point in local coordinates

param v
[in] 3D normalised direction

return
Distance to the surface boundary from inside (see above).

double **DistanceToOut**(double *r) const

Calculates the distance to the nearest boundary of a shape from inside (safety).

While the above considers the direction, this finds the nearest boundary.

param r
[in] 3D position of the point in local coordinates

return
Distance to the nearest surface boundary from inside (see above).

Private Members

const std::string **fName**

Name of this volume.

int **fMaterialIndx**

Index of the material this volume is filled with.

double **fDx**

Half length of the box along the x-axis.

double **fDy**

Half length of the box along the y-axis.

double **fDz**

Half length of the box along the z-axis.

const double **kCarTolerance** = 1.0E-9

Value of the tolerance in [mm].

double **fDelta**

Half of the above tolerance.

4.1.3 The Physics code documentation

class **Physics**

The entire physics of the simulation is provided by G4HepEm [1] and pulled-in to the HepEmShow application by the Physics.hh and Physics.cc files.

Author
M. Novak

Date
July 2023

The Physics.hh header file includes the G4HepEmRun headers that give the complete set of run-time functioinalities required for the EM physics modelling. The corresponding implementations are pulled-in all together in thePhysics.cc implentation file.

The only ingredient of G4HepEmRun, that a client application (such as HepEmShow), needs to provide is an implementation of a uniform random number generator. An object from such genertor must be plugged-in to the G4HepEmRandomEngine by implementing the two missing G4HepEmRandomEngine::flat() and G4HepEmRandomEngine::flatArray(const int, double *) methods. This is also done in the Physics.cc implementation file that completes the implementation of G4HepEmRun.

URandom is the uniform random number generator implemented in HepEmShow based on the 64-bit verson of the Mersenne Twister generator provided by c++11. An obejct of this is utilised in the Physics.cc file to complete the implementation of the G4HepEmRandomEngine as mentioned above. Then the actual uniform random number generator and the random engine objects are constructed (and set to the G4HepEmTLData obejct) in the HepEmShow.cc main function of the application.

G4HepEm implements two top level methods, HowFar and Perform, in its G4HepEmGammaManager and G4HepEmElectronManager:

- to provide the information on HowFar a given input γ or e^-/e^+ track goes according to their physics related constraints (e.g. till their next physics interaction takes place or other physics related constraints).
- to Perform all necessary physics related updates on the given input γ or e^-/e^+ track, including the production of secondary tracks in the given physics interaction (if any).

The first (HowFar) is invoked at the pre-step point while the second (Perform) is at the post-step point of each individual simulation step computation inside the *SteppingLoop::GammaStepper()* and *SteppingLoop::ElectronStepper()*.

class **URandom**

A uniform random number generator.

Author
M. Novak

Date
July 2023

This is the uniform random number generator, i.e. the only thing that is need to make the G4HepEm physics implementation complete (see more at the *Physics* documentation). This random number generator relies on the c++11 implementation of the 64-bit Mersenne Twister engine. The *URandom::flat()* method can be used to provide uniform random numbers on the $(0, 1)$. An object from this class is constructed in the HepEmShow main and set to be used in the G4HepEmRandomEngine.

Note: This random number generator can be replaced with anything that can provide uniform fandom numbers on $(0, 1)$. One need to modify the corresponding implementations in *Physics* (namely, one line in the G4HepEmRandomEngine::flat() and G4HepEmRandomEngine::flatArray() implementations in Physics.cc) and replace the *URandom* object construction in the HepEmShow main.

Public Functions

URandom(int seed = 123)
CTR.

param seed
seed of the random number generator.

~URandom()
DTR.

double **flat**()
Method to provide uniform random numbers on $(0, 1)$.

Public Members

std::mt19937_64 **fEngine**
c++11 implementation of the 64-bit Mersenne Twister engine

std::uniform_real_distribution<double> ***fDist**
uniform distribution: utilises the above random engine to provide random numbers on $(0, 1)$

4.1.4 The PrimaryGenerator code documentation

class **PrimaryGenerator**
Generates primary particles for an event.

Author
M. Novak

Date
July 2023

This is a simple primary particle generator. The kinetic energy, position, direction and the particle type (through its charge) can be configured. Note, that we simulate only e^-/e^+ and γ particles with -1, +1 and 0 charge respectively.

The *GenerateOne()* method is invoked at the beginning of each event. This generates one primary particle/track by setting the properties of the provided `G4HepEmTrack` based on the stored configuration.

Public Functions

PrimaryGenerator()

Constructor.

inline **~PrimaryGenerator**()

Destructor (nothing to do).

void **GenerateOne**(G4HepEmTrack &primTrack)

Generates one primary particle into the provided track.

The *GenerateOne()* method is invoked at the beginning of each event. This generates one primary particle/track by setting the properties of the provided `G4HepEmTrack` based on the stored configuration.

param primTrack
[inout] a track to fill in the primary particle properties

inline void **SetKinEnergy**(double ekin)

Sets kinetic energy of the primary particle.

param ekin
[in] kinetic energy of the primary particle in [MeV] units.

inline double **GetKinEnergy**() const

Privides the kinetic energy of the primary particle.

return
kinetic energy in [MeV] units.

void **SetPosition**(double *pos)

Sets the position of the primary particle.

param pos
[in] pointer to a 3D (global) position vector (legth is in [mm])

void **SetPosition**(double x, double y, double z)

Sets the position of the primary particle.

param x
[in] x-coordinate of the positon vector.

param y
[in] y-coordinate of the positon vector.

param z
[in] z-coordinate of the positon vector.

inline const double *GetPosition() const

Provides the 3D position vector of the primary particle.

return
pointer to a 3D array that stores the x, y and z-coordinates of the position vector.

void SetDirection(double *dir)

Sets the direction of the primary particle.

param dir
[in] pointer to a 3D normalised direction vector

void SetDirection(double x, double y, double z)

Sets the direction of the primary particle.

param x
[in] x-coordinate of the normalised direction vector.

param y
[in] y-coordinate of the normalised direction vector.

param z
[in] z-coordinate of the normalised direction vector.

inline const double *GetDirection() const

Provides the 3D normalised direction vector of the primary particle.

return
pointer to a 3D array that stores the x, y and z-coordinates of the normalised direction.

inline void SetCharge(double ch)

Sets the charge of the primary particle that also determines its type.

param ch
[in] the charge in e+ change units: -1 e-; 0 gamma; +1 e+.

inline double GetCharge() const

Privides the charge of the primary particle.

return
charge: -1 e-; 0 gamma; +1 e+.

Private Members

double **fKinEnergy**

Kinetic energy of the primary particle in [MeV] units.

double **fPosition**[3]

Position of the primary particles in (global) coordinates (legth is in [mm]).

double **fDirection**[3]

Normalised direction of the primary particles.

double **fCharge**

Charge of the primary particle in units of e+ charge: -1 e-; 0 gamma; +1 e+.

4.1.5 Auxiliary code documentation

Collecting data during the simulation

A collection of data that are recorded during the simulation.

Author
M. Novak

Date
July 2023

The following data is recorded during the simulation (mean is per event):

- mean values in the individual layers of the calorimeter for energy deposit, neutral (gamma) and charged (electron/positron) particle simulation steps
- mean number of energy deposited in the `absorber` and `gap`
- mean number of secondary gamma, electron and positrons produced
- mean number of neutral (gamma) and charged (electron/positron)

Quantities, recorded in the individual layers are stored in histograms and written to files at the end of the simulation while the others are reported in the screen. An example looks like

```
--- Results::WriteResults -----------------------------------

Absorber: mean Edep = 6722.95 [MeV] and  Std-dev = 309.636 [MeV]
Gap     : mean Edep = 2571.75 [MeV] and  Std-dev = 118.507 [MeV]

Mean number of gamma       4457.043
Mean number of e-          7957.899
Mean number of e+          428.922

Mean number of e-/e+ steps 36097
Mean number of gamma steps 40436.2
-------------------------------------------------------------
```

Functions

void **WriteResults**(struct *Results* &res, int numEvents = 1)

Writes the final results of the simulation.

Writes the 3 histrograms (mean energy deposit, γ and e^-/e^+ steps per-layer) into files while all the other collected data to the screen.

struct **ResultsPerEvent**

#include <Results.hh> Data that needs to be accumulated during one `event` (the scope is one event):

- at the beginning of an `event`: usually reset (to zero)
- at the end of an `event`: usually written to the run scope data (see the *`Results`* below)

Public Members

double **fEdepAbs** = {0.0}

energy deposit in the absorber during one event

double **fEdepGap** = {0.0}

energy deposit in the gap during one event

double **fNumSecGamma** = {0.0}

number of seconday γ particles generated during one event

double **fNumSecElectron** = {0.0}

number of seconday e^- particles generated during one event

double **fNumSecPositron** = {0.0}

number of seconday e^+ particles generated during one event

double **fNumStepsGamma** = {0.0}

number of γ simulation steps during one event

double **fNumStepsElPos** = {0.0}

number of e^-/e^+ simulation steps during one event

struct **Results**

#include <Results.hh> Data that are collected during the entire `run` of the simulation:

- at the beginning of the `run`: need to be initialised
- at the end of an `run`: written out (to file or to the std output) Mean quantities are computed over the simulated events.

Public Members

Hist **fEdepPerLayer**
mean energy deposit per-layer histogram

Hist **fGammaTrackLenghtPerLayer**
mean number of γ steps per-layer histogram

Hist **fElPosTrackLenghtPerLayer**
mean number of e^-/e^+ steps per-layer histogram

double **fEdepAbs** = {0.0}
mean energy deposit in the `absorber`

double **fEdepAbs2** = {0.0}
mean of the squared energy deposit in the `absorber`

double **fEdepGap** = {0.0}
mean energy deposit in the `gap`

double **fEdepGap2** = {0.0}
mean of the squared energy deposit in the `gap`

double **fNumSecGamma** = {0.0}
mean number of the produced secondary γ particles

double **fNumSecGamma2** = {0.0}
mean of the squared number of produced secondary γ particles

double **fNumSecElectron** = {0.0}
mean number of the produced secondary e^- particles

double **fNumSecElectron2** = {0.0}
mean of the squared number of produced secondary e^- particles

double **fNumSecPositron** = {0.0}
mean number of the produced secondary e^+ particles

double **fNumSecPositron2** = {0.0}
mean of the squared number of produced secondary e^+ particles

double **fNumStepsGamma** = {0.0}
mean number of γ steps in the entire calorimeter

double **fNumStepsGamma2** = {0.0}
mean of the squared number of γ steps in the entire calorimeter

double **fNumStepsElPos** = {0.0}

mean number of e^-/e^+ steps in the entire calorimeter

double **fNumStepsElPos2** = {0.0}

mean of the squared number of e^-/e^+ steps in the entire calorimeter

ResultsPerEvent **fPerEventRes**

data structure to accumulate results during a single event

class **Hist**

A simple histogram only to collect some data during the simulation.

Author
M. Novak

Date
July 2023

Public Functions

Hist(const std::string &filename, double min, double max, int numbin)

Constructor.

param filename
String to be used as file name when writing into file.

param min
Minimum bin value.

param max
Maximum bin value.

param numbin
Number of bins required between `min` and `max`.

Hist(const std::string &filename, double min, double max, double delta)

Constructor.

param filename
String to be used as file name when writing into file.

param min
Minimum bin value.

param max
Maximum bin value.

param delta
Required width of a bin.

`Hist`()

Default constructor.

inline `~Hist`()

Destructor.

void `Initialize`()

Auxiliary method to setup the initial state of the histogram.

void `ReSet`(const std::string &filename, double min, double max, int numbins)

Method to modify the properties of the histogram.

param filename
The new name of the filename.

param min
The new minimum bin value.

param max
The new maximum bin value.

param numbins
The new number of bins required between `min` and `max`.

void `Fill`(double x)

Method to populate the histogram with data: the corresponding bin content is increased by 1.

param x
Value to add.

void `Fill`(double x, double w)

Method to populate the histogram with data and a weight: the corresponding bin content is increased by the weight.

param x
Value to add.

param w
The corresponding weight.

void `Scale`(double sc)

Method to scale all bin content by a constant.

param sc
Scaling factor.

inline int `GetNumBins`() const

Method to provide the number of bins.

return
Number of bins.

Providing input arguments to the HepEmShow application

To see all configuration option, run the application as:

```
./HepEmShow --help

=== Usage: HepEmShow [OPTIONS]

    -l  --number-of-layers      (number of layers in the calorimeter)          -␣
↪default: 50
    -a  --absorber-thickness    (in [mm] units)                                -␣
↪default: 2.3
    -g  --gap-thickness         (in [mm] units)                                -␣
↪default: 5.7
    -t  --transverse-size       (of the calorimeter in [mm] units)             -␣
↪default: 400
    -p  --primary-particle      (possible particle names: e-, e+ and gamma)    -␣
↪default: e-
    -e  --primary-energy        (in [MeV] units)                               -␣
↪default: 10 000
    -n  --number-of-events      (number of primary events to simulate)         -␣
↪default: 1000
    -s  --random-seed                                                          -␣
↪default: 1234
    -d  --g4hepem-data-file     (the pre-generated data file with its path)    -␣
↪default: ../data/hepem_data
    -v  --run-verbosity         (verbosity of run infomation: nothing when 0)  -␣
↪default: 1
    -h  --help
```

struct **InputParameters**

A data structure that encapsulates all the possible input arguments of the HepEmShow application.

Author
M. Novak

Date
Aug 2023

Public Functions

inline **InputParameters**()

CTR with default values: default geometry, primary and event configuirations (see below) with pre-generated data files expected at `../data/hepem_data` relative to the HepEmShow executable.

Public Members

Geometry **fGeometry**

the geometry related configuration

PrimaryAndEvents **fPrimaryAndEvents**

the primary partcile and events related configuration

std::string **fG4HepEmDataFile**

the pre-generated data file (with path)

int **fRunVerbosity**

level of printout verbosity duing setting up: nothing when < 1.

struct **Geometry**

The geometry related input arguments.

Public Functions

inline **Geometry**()

CTR with default values: 50 layers of 2.3 mm absorber and 5.7 mm gap with 400 mm transvers size.

Public Members

int **fNumLayers**

number of layers in the calorimeter

double **fThicknessAbsorber**

absorber thickness along X in [mm]

double **fThicknessGap**

gap thickness along X in [mm]

double **fThicknessCalo**

calorimeter thickness along X [mm] ONLY if number of layers is zero

double **fSizeTransverse**

calorimeter full size along YZ in [mm]

struct **PrimaryAndEvents**

The primary partcile and events related input arguments.

Public Functions

inline **PrimaryAndEvents**()

CTR with default values: simulate 1000 events, starting with an electron of 10 GeV each (do not report progress).

Public Members

std::string **fParticleName**

primary particle name: {"e-", "e+" or "gamma"}

double **fParticleEnergy**

primary particle energy in [MeV]

int **fNumEvents**

number of events to simulate (each will start with a single primary)

double **fRandomSeed**

seed for the random number generator

Event loop, stepping loops and the track stack

class **EventLoop**

Event loop for simulating the required number of primary tracks/events.

Author
M. Novak

Date
July 2023

The *EventLoop::ProcessEvents()* method is responsible to generate track(s) for the required number of events and simulate the histories of all primary and their secondary tracks.

Public Static Functions

static void **ProcessEvents**(G4HepEmTLData &theTLData, G4HepEmState &theState, *PrimaryGenerator* &thePrimaryGenerator, *Geometry* &theGeometry, *Results* &theResult, int numEventToSimulate, int verbosity)

Generates and simulates the required number of events.

Events, i.e. primary track(s) are generated by using the input *PrimaryGenerator*. At the beginning of each event, the *PrimaryGenerator* is used to generate the primary track(s) that belong to the actual event. Note, that we have only one primary track per-event at the moment. The generated primary track(s) is inserted/pushed into the *TrackStack* as the very first track and the simulation of the event starts. During the simulation of the event:

- one track is popped from the stack and the appropriate *SteppingLoop* is called to simulate its entire history ina step-by-step way

- at the end of each simulation step, secondary tracks that are created in that step in the related physics interaction (if any), are inserted/pushed into the *TrackStack* Simulation of the event is completed when the *TrackStack* becomes empty. See the implementation for more details.

In order to be able to collect some infomation during the event processing, the *BeginOfEventAction()*/*EndOfEventAction()* methods are invoked before/after each event processing while the *BeginOfTrackingAction()*/*EndOfTrackingAction()* methods are invoked before/after tracking each new track.

param theTLData
a G4HepEm specific (thread local) object primarily used to obtain all physics related information from G4HepEm needed to compute a simulation step

param theState
a G4HepEm specific object that stores pointers to the top level G4HepEm data structure and parameters that are used by G4HepEm to provide all physics related infomation needed to compute a simulation step

param thePrimaryGenerator
the primary generator that is used to generate primary track(s) at the beginning of each event (only one primary track per event in our case now)

param theGeometry
the geometry of the application in which the input track history is simulated

param theResult
the data structure that holds all the infomation needs to be collected during the simulation.

param numEventToSimulate
number of events required to be simulated

param verbosity
to control the verbosity of printouts reporting progress and state of the event processing

Private Static Functions

static void **BeginOfEventAction**(*Results* &theResult, int eventID, const G4HepEmTrack &thePrimaryTrack)

Method invoked at the beginning of each event by passing the (single) primary track of the event.

static void **EndOfEventAction**(*Results* &theResult, int eventID)

Method invoked at the end of each event.

static void **BeginOfTrackingAction**(*Results* &theResult, G4HepEmTrack &theTrack)

Method invoked before start tracking of a new track (provided as input argument).

static void **EndOfTrackingAction**(*Results* &theResult, G4HepEmTrack &theTrack)

Method invoked after terminating tracking of a track (provided as input argument).

class **SteppingLoop**

Stepping loops for simulating e^-, e^+ and γ particle histories.

Author
M. Novak

Date
July 2023

The stepping loops can calculate a given γ or e^-/e^+ particle simulation history from their initial state till the end in a step-by-step way (by the *SteppingLoop::GammaStepper*(G4HepEmTLData&, G4HepEmState&, *TrackStack*&, *Geometry*&, *Results*&, int) and *SteppingLoop::ElectronStepper*(G4HepEmTLData&, G4HepEmState&, *TrackStack*&, *Geometry*&, *Results*&, int) respectively). At each step:

- the actual step length is calculated (accounting both the geometrical and the physics related constraints)
- the track is moved to its post-step position
- all physics related actions, happening along and/or at the post-step point, are performed on the track
- secondary tracks, generated in the given step by a physics interaction (if any), are insterted into the track stack (by calling the *SteppingLoop::StackSecondaries*(G4HepEmTLData&, *TrackStack*&, G4HepEmTrack&) method)
- information (e.g. energy deposit) might be collected at the end of each simulation step (by calling the *SteppingLoop::SteppingAction*(*Results*&, const G4HepEmTrack&, const Box*, double, int, int, int, int) method)

A bit more details:

A simulation history is terminated when:

- the particle kinetic energy becomes zero (e.g. an e^- lost all its kinetic energy along its last step)
- the particle participated in a destructive interaction (e.g. photoelectric absorption of a γ photon or conversion to e^-/e^+ pairs)
- the particle leaves the calorimeter (in a normal Geant4 simulation the the history is terminated when the particle leaves the world)

The physics related step length constraints as well as the actions (including the secondary track production) are provided by the G4HepEm implementation of the EM physics simulation.

G4HepEm implements two top level methods, in its Gamma and Electron managers:

- to provide the information on HowFar a given input γ or e^-/e^+ track goes according to their physics related constraints (e.g. till their next physics interaction takes place or due to other physics related constraints).
- to Perform all necessary physics related updates on the given input γ or e^-/e^+ track and produce all secondary tracks in the given physics interaction (if any).

The first (HowFar) is invoked at the pre-step point while the second (Perform) is at the post-step point of each individual simulation step computation inside the *SteppingLoop::GammaStepper()* and *SteppingLoop::ElectronStepper()*.

In G4HepEm it's the G4HepEmTLData (thread local data) that is used in the top level, two sided communication between the consumer and G4HepEm. It encapsulates the (primary and secondary) tracks and the random number generator dedicated for one particular thread. Its **primary** Gamma/Electron track field is used to store the actual state of the γ or e^-/e^+ track that is under tracking. The step limit, imposed by all physics related constraints on the actual track, is calculated at each pre-step point by calling the bove-mentioned HowFar top level method provided by G4HepEm. Then the Perform method needs to be invoked at the post-step point that performs all necessary physics related updates on the input **primary** track while produces all **scondary** tracks related to the given physics interaction (if any). The secondary tracks are delivered back to the caller in the appropriate **seondary** track fields of the G4HepEmTLData object.

We might provide more details on how a γ and e^-/e^+ simulation step is computed but you might find some information by inspecting the implementations of the top level HowFar and Perform G4HepEm methods in the corresponding G4HepEmGammaManager/G4HepEmElectronManager.

Public Static Functions

static void **GammaStepper**(G4HepEmTLData &theTLData, G4HepEmState &theState, *TrackStack* &theTrackStack, *Geometry* &theGeometry, *Results* &theResult, int eventID)

Stepping loop for simulating the entire history of a γ track.

The initial state of the γ track is provided in the `G4HepEmGammaTrack` field of `theTLData` input argument by the caller. The history is simulated then till the end, the state of the γ track is updated while secondary tracks, produced in the physics interactions, are pushed to `theTrackStack` (if any) and the required simulation results are collected/updated into `theResult` structure after each individual simulation step.

param theTLData
a `G4HepEm` specific (thread local) object primarily used to obtain all physics related information from `G4HepEm` needed to compute a simulation step

param theState
a `G4HepEm` specific object that stores pointers to the top level `G4HepEm` data structure and parameters that are used by `G4HepEm` to provide all physics related infomation needed to compute a simulation step

param theTrackStack
the track stack that is used to store the secondary tracks produced while simulating the entire history o fthe input γ track

param theGeometry
the geometry of the application in which the input track history is simulated

param theResult
the data structure that holds all the infomation needs to be collected during the simulation. It might be updated after each simulation step by calling the `SteppingAction` method.

param eventID
ID of the currently simulated event, i.e. the one to which the given input γ track belongs to

static void **ElectronStepper**(G4HepEmTLData &theTLData, G4HepEmState &theState, *TrackStack* &theTrackStack, *Geometry* &theGeometry, *Results* &theResult, int eventID)

Stepping loop for simulating the entire history of a e^-/e^+ track.

The initial state of the e^-/e^+ track is provided in the `G4HepEmGammaTrack` field of `theTLData` input argument by the caller. The history is simulated then till the end, the state of the e^-/e^+ track is updated while secondary tracks, produced in the physics interactions, are pushed to `theTrackStack` (if any) and the required simulation results are collected/updated into `theResult` structure after each individual simulation step.

param theTLData
a `G4HepEm` specific (thread local) object primarily used to obtain all physics related information from `G4HepEm` needed to compute a simulation step

param theState
a `G4HepEm` specific object that stores pointers to the top level `G4HepEm` data structure and parameters that are used by `G4HepEm` to provide all physics related infomation needed to compute a simulation step

param theTrackStack
the track stack that is used to store the secondary tracks produced while simulating the entire history o fthe input e^-/e^+ track

param theGeometry
the geometry of the application in which the input track history is simulated

param theResult
the data structure that holds all the infomation needs to be collected during the simulation. It might be updated after each simulation step by calling the `SteppingAction` method.

param eventID
ID of the currently simulated event, i.e. the one to which the given input e^-/e^+ track belongs to

Private Static Functions

static void **StackSecondaries**(G4HepEmTLData &theTLData, *TrackStack* &theTrackStack, G4HepEmTrack &thePrimary)

Auxiliary method that pushes the secondary track(s), produced by physics interactions at the post-step point (if any), into the track stack.

param theTLData
the `G4HepEm` specific (thread local) object that is used by `G4HepEm` to deliver the secondary tracks to the caller after calling the its `Perform` top level method

param theTrackStack
the track stack that is used to store the secondary tracks produced while simulating the entire history of the input track in the steppers

param thePrimary
the primary track, in its post interaction state (after calling `G4HepEm` top level `Perform` method), i.e. the one that underwent the physics interaction

static void **SteppingAction**(*Results* &theResult, const G4HepEmTrack &theTrack, const *Box* *currentVolume, double currentPhysStepLength, int indxLayer, int indxAbsorber, int eventID, int stepID)

This method is called at the end of each simulation steps to collect some data during the simulation.

This method provides the possibility of collecting some data after each simulation steps (e.g. energy deposit or length of the step). Among the `Geant4` user actions this corresponds to the `G4UserSteppingAction`

param theResult
the data structure that holds all the infomation needs to be collected during the simulation (some fields might be updated)

param theTrack
the primary track, in its post interaction state, i.e. at the end of the step

param currentVolume
pointer to the volume (`absorber`/`gap`) in which the simulation step was done

param currentPhysStepLength
real (physical) length of the step

param indxLayer
index of the layer in which the step was done

param indxAbsorber
indicates if the step was done in the `absorber` (0) or in the `gap` (1)

param eventID
ID of the event to which the particle under tracking belongs to

param stepID
ID of this step that was just performed, i.e. number of steps cmpleted so far with with the current track

class **TrackStack**

A simple track-stack to handle both primary and secondary particle tracks.

Author
M. Novak

Date
July 2023

This stack holds tracks (all belonging to the same event), that are still to be tracking (i.e. still need to call/instert to the appropriate *SteppingLoop*):

- at the begining of each event, (a) primary track is inserted into the stack (inside the *EventLoop::ProcessEvents()*) as the very first track (NOTE: assumed to have only one primary per event for simplicity)
- all secondaries, generated by the entire simulation of the event, are inserted when created (inside the appropriate *SteppingLoop*), i.e. pushed and later popped for tracking
- the event is completed when the track-stack becomes empty again

A new event can be started then.

Public Functions

TrackStack()

CTR.

inline **~TrackStack**()

DTR.

int **PopInto**(G4HepEmTrack &track)

Pops a secondary track from the stack and writes to the input address.

This method is called from *EventLoop::ProcessEvents()* before start tracking a new track. It returns with the original index of the popped track or -1 when the track is actually empty, i.e. no more track to pop.

param track
[inout] the address of the G4HepEmTrack where the next track should be popped, i.e. copied.

return
returns with the original index of the popped track or -1 if the there are no more tracks in the track

int **GetTypeOfNextTrack**()

Can provide the type of the next track.

Returns with an integer that encodes the type of the next track in the stack, i.e. the type of the track that will be popped when calling *PopInto()* next time.

return
an integer indicating the type of the next track:

- -1 in case of e^-
- 0 in case of γ
- +1 in case of e^+
- -999 if the stack is empty

G4HepEmTrack &**Insert**()

Returns a reference to a secondary track that can be used to push a new track into the stack.

This method is called whenever a new track needs to be inseted into the stack. The provided reference can be used to fill the track infomation (the referenced track is re-set).

return
A reference to a `G4HepEmTrack` that can be used to add a new track to the stack (by filling in its filed).

void **Copy**(G4HepEmTrack &from, G4HepEmTrack &to)

Copying the content of the `from` to the `to` track.

inline int **GetNextTrackID**()

Returns with the next track ID (track ID is incremented whenever this method is invoked).

inline void **ReSetTrackID**()

Resets the track ID to zero.

Private Members

int **fSize**

current capacity of the track stack

int **fCurIndx**

number of tracks used from the capacity

int **fCurrentTrackID**

current track ID

std::vector<G4HepEmTrack> **fTrackVect**

the stack as a vector of tracks

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Sea Agostinelli, John Allison, K al Amako, John Apostolakis, H Araujo, Pedro Arce, Makoto Asai, D Axen, Swagato Banerjee, GJNI Barrand, and others. Geant4—a simulation toolkit. *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, 2003.

[2] John Allison, Katsuya Amako, JEA Apostolakis, HAAH Araujo, P Arce Dubois, MAAM Asai, GABG Barrand, RACR Capra, SACS Chauvie, RACR Chytracek, and others. Geant4 developments and applications. *IEEE Transactions on nuclear science*, 53(1):270–278, 2006.

[3] John Allison, Katsuya Amako, John Apostolakis, Pedro Arce, Makoto Asai, Tsukasa Aso, Enrico Bagli, A Bagulya, S Banerjee, GJNI Barrand, and others. Recent developments in geant4. *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 835:186–225, 2016.

[4] Mihaly Novak, Jonas Hahnfeld, and Ben Morgan. mnovak42/g4hepem: The G4HepEm R&D Project. 2022. accessed on 7 September 2023, https://github.com/mnovak42/g4hepem https://g4hepem.readthedocs.io/en/latest/.

INDEX

M

P

R

S

T

U

W