Geometry Description and detector interface

Abstract

This document summarizes a possible path forward for the geometry description for the simulations of EIC detectors. It contains the list of what we believe should be the requirements for a EIC geometry description system. The considerations in this document are probably general enough and can be applied to any geometry system independently on the specific technology choice, the focus however is on the I/O of geometry and on the link to sensitivity information, since these two aspects have been the focus of our discussions in ESC meetings in FY2017 .

Initial considerations

It is safe to assume that in the time-scale for which detector simulations for EIC are needed Geant4 will continue to be the de-facto standard for detector simulations, we should thus consider the paradigms implemented in Geant4 (e.g. hierarchical geometry, concepts of sensitive detectors and hits) as general guidelines for our future works.

There are two main use-cases that drive the development of a geometry module: simulation and reconstructions. It is an obvious requirement that the same geometry description should be used between the two subsystems. How to implement this paradigm is mainly left to the specific choices of experiments, and currently no real detector-independent framework has emerged so far as a widely adopted standard. Many projects have tried to propose such frameworks(among the one mentioned in our meetings are SLIC and DD4hep) with somewhat limited success . It is very important to stress that we do not believe that the lack of wider adoption is not due to inherent quality of the artifacts (that on the contrary is usually quite high), but since no large experiment has adopted these tools as standard the community behind these tools has remained small and fractionated.

Simulation requires the description of geometry in increased level of complexity: from the simplified ideal detectors used for concept studies, to the full detailed simulations of running experiment. The data reconstruction as a general idea requires a more *conceptual* description of the geometry in terms of read-out elements instead of physical placements. In particular the mapping between sensitive geometry element and hits is of crucial importance.

We identified two possible ways of defining the geometry of a detector for simulation.

Geometry implementation via code

The first approach, to write code that uses directly geometry primitives, is usually preferred for smaller applications, e.g. the majority of the examples distributed with Geant4 toolkit create the detector geometry in this way. In case of ROOT based frameworks this approach is quite natural, because you can add to this basic scenario some I/O and scripting capabilities (TGeo classes to describe the geometry in a program, ROOT I/O to write geometry elements, and ROOT scripts, that are programs by themselves, to steer the process).

The benefits of this process is a detailed control of the process by developers: since everything is defined in a procedural program it is relatively easy to implement complex logic-flows with nested loops, conditionals etc. The drawback of this approach is a general lock-in with a given technology. A system based on these technologies tends to be less capable of evolution, because it is harder to adopt new software tools and practices when they become available. As a consequence there is a lack of portability to hardware architectures not supported by the chosen technology. While this is in general a minor problem for a running experiment where some fundamental technology choices are made, this can be a major challenge for an experiment that is in an early R&D phase where it is difficult to predict the computing landscape in the future (e.g. the relative role of large supercomputers centers, commercial cloud solutions and local data-centers).

Geometry implementation via data source

The second approach is to define the data persistency and formats independently of the software artifacts that will use them. In this approach a data model and format to exchange information is agreed-upon and then software products are developed or adapted to adhere to this standard and to provide interfaces to the data. With Geant4 the set of examples located in *example/extended/persistency* show how to write/read geometry GDML, ROOT and ASCII formats.
The benefits of this approach include a larger modularity of the system (developing components separately). A possible drawback of this approach is the need for code duplication in some cases (e.g. a library to read the data source in Geant4, one in ROOT, one in visualization).
If manpower allows, the second approach should be preferred. As modern software best practices show a distinction between data, persistency and control flow allows for *future proof* systems where each component can be developed, validated and replaced separately. This is the approach used with success by industry: replace monolithic systems with much smaller services that cooperate exchanging data and messages in well established formats. e.g. REST APIs and JSON snippets, micro-services architectures, containerization technologies. As an example close to scientific computing we can consider the very popular SciPy software stack: it is composed of largely independent modules that communicate via the exchange of relatively simple data structures (numpy arrays). The details of the implementation are left to the single developers of a given library but it is guaranteed that one can cherry-pick different components from the stack and make them coherently work together (in some cases even the programming language may be different between modules: python, C++, CUDA,...). On the contrary ROOT, is monolithic and one cannot *choose* a single component without using the others.

We recognize that the GDML format is currently the only *de-facto* standard that can be used natively used by Geant4 and ROOT applications. Many other applications that do not build directly on this format do still have converter to at least export to this format.

Definitions

Simulations require a very detailed description of three separate concepts: *solids*, *logical volumes (LV)* and *placements*. LHC experiments, for example have tens of thousands of logical volumes and millions of placements. Geometry is described first in terms of basic shapes (boxes, spheres) and their sizes: the solids. Material and other physical properties (sensitivity, magnetic fields) are attached to the solid to form a logical volume (different LV can have the same associated solid). Logical volumes are then placed in a hierarchical structure to form a physical volume (the same LV can be placed several times in the world, parametrizations allow the change of some aspects of the LV at runtime).

In particular it is very important to note that the role of sensitivity is to link a geometry element to a specific algorithm. As a general concept Sensitive Detectors (SD) are not C++-objects encapsulating data but instead they are algorithms that transform data to derived data (transforming a G4Step into a user-defined Hit). In real-world applications this distinction is actually blurred: SDs become "support" structures to easily locate a hit in space.
Another important distinction to make is that digitization is outside the scope of this discussion. Digitization is the further transformation of hits to digits, i.e. data objects that resemble the output of the detector (e.g. adding noise, time-response). In general it is a bad design to include digitization in the SD (more generally in the detector simulation): digitization is usually very specific to a given detector, difficult to share and port between experiments.

Use-cases

- **Full-simulations**. Full simulations usually contain detailed descriptions of the detector geometry. We have mainly discussed how to attach sensitivity information to the description and how to make available the relevant information to the other components of the pipeline (i.e. reconstruction, analysis and visualization). In Geant4 applications users are responsible for creating the code for allocating and filling hits (in SensitiveDetectors class) and for writing hits in output files. The same SD can be associated to any number of logical volumes and a single SD can create more than one hits collection. The SD elements and associated collections are identified by names (strings) while there is no general rule enforced by the toolkit to identify a single hit in a collection, however it is a general accepted practice to identify hits by one or more indexes (e.g. the calorimeter cell or tracker strip number). Thus a hit is generally uniquely identified by the triplet: "SDName" (that in general also identifies the LV associated to it), "HitsCollection", "HitID#".
  In general two types of hits exist: calorimetric and tracking hits. The latter are a collection of all energy deposits associated with sensitive detectors, each step that produces signal in the LV is transformed in a separate hit. The former is an object that accumulates the energy in a given geometric element (a cell). The second is used for calorimeters because the number of steps to deal with may be extremely large and it is unpractical to save all hits. This distinction between tracking and calorimetry is probably an oversimplification. Specifically for NP experiments TPC hits (technically a tracker) are indeed a mixture of the two with multiple energy deposits being collected in a single hit.
- **Reconstruction and analysis**. In this context the reconstruction and analysis inputs consist of the geometry information and hits collections. There are two main differences with respect the simulation use-case. Depending on the detector or application, the geometry description may be simplified in reconstruction, because not all the details may be necessary, however what is mandatory is that the hits can be associated to volumes. In simple cases the names of the SD, collections and hits IDs are enough to do this mapping.. It is very important that the geometry information can be accessed outside of the simulation system of choice, ideally the hits collection should contain enough information to be able to self describe their position in the geometry tree, without the need to use any of the code used in the simulation.

Requirements

1. The geometry information **should be the same in both simulation and reconstruction.**
2. Fast simulation systems should, as much as possible, **be able to use the common exchange format**.
3. The geometry system **should allow to include misalignment** and more general condition data.
4. Geometry description format should be **independent of a specific software technology**.
5. Geometry description **should be modular**. It should be possible to specify different geometry components in isolation with ideally zero dependency between different modules (detectors).
6. Geometry description **should allow to specify logical information** (sensitivity, B-Fields) in addition the solids, material and placements. In particular sensitivity is recognized as a critical issue.
7. It **should be possible to make the geometry description persistent**. Different equivalent output format should be supported (e.g. ROOT files, GDML files) and it should always be possible to translate one format into another in a simple manner.
8. Hits output files produced in a simulation job should be as much as possible self-describing, in particular **it should be possible to locate hits in space** without the need to run the simulation job. A *self-describing* format for the hits would be ideal, but in case this is not possible , the additional libraries to manipulate hits should not depend on the simulation stack used to produce the hits.
9. It should be possible to **change sensitivity attributes without changing other static aspects of the geometry**.
10. Geometry exchange format should **allow clients to use a subset of the features clearly stating which are the optional ones.** We should support existing interested frameworks (e.g. EicRoot, GEMC, Fun4ALL, SLIC,…), without discouraging other R&D activities (e.g. DD4hep). Since it is difficult to support all use-cases, the minimal set of mandatory elements to support should be clearly specified and what to do with non-supported one should be stated (e.g. ignore visualization attributes if not needed).
11. **Some support for import from CAD** should be foreseen.
12. Geometry information **should have support for versioning.**

We recognize that experiments in different levels of maturity may have additional requirements, as such this list of requirement should be considered as the baseline for EIC detector-geometry exchange format and may evolve with time and experimentation.

List of technologies identified so far

- GDML (http://gdml.web.cern.ch/GDML/). Pure XML description of the detector geometry. Supported by Geant4 and ROOT.
- GEMC (https://gemc.jlab.org/gemc/html/index.html). More of an application than a geometry module. Includes many other functionalities. Support many possible sources. Used at JLAB for CLAS12 and EIC studies.
- DD4Hep (http://aidasoft.web.cern.ch/DD4hep). Developed for LC efforts. A standard (XML) and a software product.
- AgML (https://drupal.star.bnl.gov/STAR/comp/simu/geometry0/agml-tutorials) XML descripion of geometry supporting loops, variables, constants, data structures, branches and hits. Started for Geant3. Planned support for Geant4.
- FairROOT (https://fairroot.gsi.de/) . Developed at GSI for Fair experiments. Very similar to ALICE approach. Geometry input in several formats is possible, including ROOT TGeo and GDML. A FairRoot clone (EicRoot), which is being used at BNL for EIC studies, has a number of features facilitating a tight coupling between simulation and reconstruction passes of a typical detector modeling process implemented already, though not in an easily portable way.