

A System for Managing the CLEO III Detector Geometry

Christopher D. Jones
University of Florida

Abstract

A description of the detector's geometry is needed for several software tasks, e.g. Monte Carlo generation, track finding, and event displays. This description must also be capable of handling change over time to account for alignment shifts in the detector. We discuss the DetectorGeometry package, which is an object oriented detector geometry description that is compatible with the GEANT3 and GEANT4 geometry models. In addition to describing the placement of bulk materials, this system is also used to determine the position of "sensors", e.g. drift chamber wires, calorimeter crystals, and silicon strips.

Introduction

CLEO III needs a description of the detector's geometry for several software tasks, e.g. Monte Carlo generation, track finding, and event displays. This description must also be capable of handling change over time to account for alignment shifts in the detector.

In this paper we will start by discussing the overall design of the DetectorGeometry package and then move to a more detailed description of the package. We will then explain how we handle alignment changes and then conclude with how we plan to actually use this package.

Design Overview

Our primary goal in developing the DetectorGeometry package was to provide one 'master' description of the detector that all software routines use as a reference. To be a usable reference the package must be able to deal with alignment changes, which includes notifying the appropriate routines when an alignment change has occurred. A secondary goal was to make it easy to generate GEANT3 (and possibly GEANT4) geometries from the detector geometry description.

In the design we separate a sub-detector's geometry description into two parts: *material placement* and *sensors*. The material placement describes the position of bulk materials through out the detector (e.g. drift chamber endplates). To properly describe a sub-detector, we need both a base and an aligned material placement description. The base geometry is the 'nominal' description of the sub-detector that is used as the reference for all subsequent sub-detector alignments. The base geometry description is valid through the lifetime of the experiment. The aligned geometry description is based on the structure found in the base geometry but includes the known alignments. For example, if the base geometry has a drift chamber composed of an inner and outer shell and two endplates, then the aligned description also has all four of those pieces but

includes the known rotation of the entire drift chamber. The aligned geometry is updated each time a new alignment is measured for that sub-detector. The sensor geometry description handles the position of the ‘sensitive’ elements of the detector: wires, silicon strips, cathode pads, etc. The sensors use the aligned geometry to set the lab coordinates of the different sensor elements and therefore the sensors must be updated each time the aligned geometry is updated.

All sub-detectors’ material placement descriptions can be constructed using a general-purpose geometry description package. Unfortunately we were unable to devise a general-purpose description to describe all the different types of sensors used by the different sub-detectors. Therefore each sub-detector has its own method of dealing with sensors. The rest of this paper will only deal with the general-purpose material placement package, which we call DetectorGeometry, and we will not discuss the sub-detector specific sensor packages.

DetectorGeometry Package

The DetectorGeometry package is a group of C++ classes that are used to build a description of the placement of materials in the detector. The design is loosely based on the GEANT3 geometry model in which one creates an abstract volume and then places copies of that volume inside other volumes. Our model supports hierarchical descriptions (i.e. Drift Chamber is made up of two endplates, an inner and outer shell) and it is easy to share objects within the hierarchy. For example, the Drift Chamber endplates can share the same material description (i.e. aluminum) or the base geometry and aligned geometry for the Drift Chamber can share the same endplates shape description.

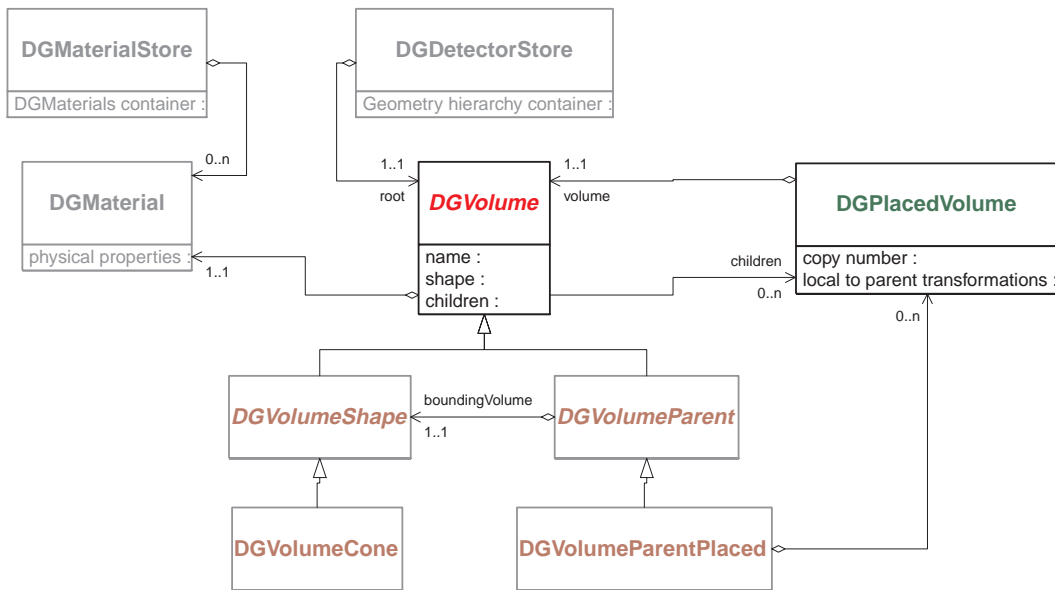


Figure 1: A simplified DetectorGeometry class diagram

Figure 1 shows a simplified class diagram of the DetectorGeometry package. (See [1] for a complete description.) *DGVolume* is the base class which is used to represent a physical object, e.g. a Cesium Iodide crystal. A *DGVolume* has a name and has a reference to its material properties (which are encapsulated in the *DGMaterial* class). In

addition, a *DGVolume* can have children which allows the hierarchical description of a detector. The *DGVolumeShape* class inherits from *DGVolume* and is the base class for a single physical volume, i.e. a *DGVolumeShape* has no children. Classes that inherit from *DGVolumeShape* define a particular geometry shape (e.g. cone) where the shapes have been chosen to be compatible with the GEANT3 shapes. The *DGVolumeParent* class inherits from *DGVolume* and is the base class for all logical volumes, i.e. volumes that contain other volumes. A *DGVolumeParent* has a bounding volume (which is a *DGVolumeShape*) that contains all the children volumes. A *DGVolumeParentPlaced* is a type of *DGVolumeParent* which holds a list of *DGPlacedVolumes*. A *DGPlacedVolume* describes a volume that has been ‘placed’ inside a parent volume. This placed volume has a reference to a *DGVolume*, a copy number, and knows the local to parent coordinate system transformation. In fact, *DGPlacedVolume* holds two transformations: the local nominal to parent and the local alignment to local nominal transformations. The second transformation makes it easier to handle alignment changes.

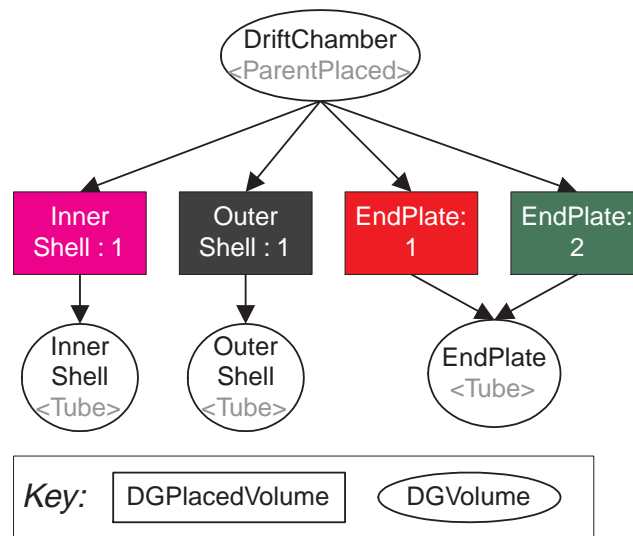


Figure 2: Example of simple drift chamber described by DetectorGeometry classes

Figure 2 shows an example of a simple Drift Chamber represented by DetectorGeometry classes. The root of the hierarchy is a *DGVolumeParentPlaced* with four children. Each of the four *DGPlacedVolumes* references a *DGVolume* where each shell has its own *DGVolumeTube*, but the two endplates share a *DGVolumeTube*.

A *DGPlacedVolume* only knows how to transform from its local coordinate system to its parent’s coordinate system. The more useful local to lab transformation is handled by the *DGLabVolume* class. *DGLabVolumes* are created by specifying a ‘path’ from the root volume to the appropriate *DGPlacedVolume*. For example, the path to the drift chamber endplate #2 might be

‘root’ -> ‘drift chamber parent volume’:1 -> ‘drift chamber endplate’:2

The *DGLabVolume* has two transformations: a nominal local to lab and an aligned local to lab. The following code fragment demonstrates how these are accessed.

```

//detectorStore is a DGDetectorStore containing the drift chamber
//There is only one DGPlacedVolume called 'drift chamber endplate':2 so let
// detectorStore find the path to it
DGPath pathToEndplate2 = detectorStore.pathTo( "drift chamber endplate", 2 );
DGLabVolumePtr pLabEndplate2 = detectorStore.labVolume( pathToEndplate2 );

//use matrix math to find the displacement of the center caused by the alignment
HepPoint3D origin(0,0,0);
HepVector3D displacement = pLabEndplate2->localToLabAligned()*origin -
                           pLabEndplate2->localToLabNominal()*origin;

```

To make memory management easier, we use reference counting smart pointers to keep track of multiple references to the same object (e.g. sharing the *EndPlate DGVolumeTube* between two *DGPlacedVolumes*). They are 'reference counting' because they keep track of how many pointers are referring to this object. When the number of reference goes to 0 (i.e. no pointers are referring to the object) then the object is deleted. They are called 'smart pointers' because they use the operations * and -> just like a pointer. We actually have two types of reference counting smart pointers: Const and Non-Const. The Const pointers do not allow you to change the object it refers to. These types of smart pointers have 'Const' in their name, e.g. *DGConstVolumePtr*. The Non-Const pointers allow you to change the object and do not have 'Const' in their name, e.g. *DGVolumePtr*. The Non-Const pointers inherit from their 'Const' counterparts, because anywhere you can use a Const pointer, you should also be able to use a Non-Const pointer.

The detector hierarchy holds smart pointers to the base class *DGVolume* but sometimes you need to know the actual class type, e.g. is this *DGVolume* actually a *DGVolumeParent*. The templated class *DGVolumeType<>* is used to safely convert a *DGVolumePtr* into a specific *DGVolume* type. The code snippet below shows how this is accomplished.

```

DGVolumePtr pTube = new DGVolumeTube(...);
// attempt to convert pTube from a DGVolume to a DGVolumeTube
DGVolumeType<DGVolumeTube> pTest1= pTube;
// did the conversion succeed?
If( pTest1.isValid() ) {
    //can now treat pTest1 as if it were a pointer to a DGVolumeTube
    ...
}
// attempt to convert pTube into a DGVolumeParent
DGVolumeType<DGVolumeParent> pTest2 = pTube;
If( pTest2.isValid() ) {
    //This will never be true since DGVolumeTube is NOT a DGVolumeParent
}

```

Because we have two different types of smart pointers for *DGVolume* (Const and Non-Const) we must have two types of converts, *DGVolumeTypePtr<>* for *DGVolumePtr* and *DGConstVolumeTypePtr<>* for *DGConstVolumePtr*.

Handling Alignment Changes

We use two descriptions of a detector: base and aligned. Both of these descriptions use *DGPlacedVolumes* to set the position of *DGVolumes*. The *DGPlacedVolume* class holds two transformations: local nominal to parent coordinate and local measured alignment to local nominal. The first transformation places the volume into the nominal location within the parent volume. The second transformation handles alignment shifts away from the nominal. The base description is constructed with all *DGPlacedVolumes* using the identity transformation for the aligned to nominal transformation.

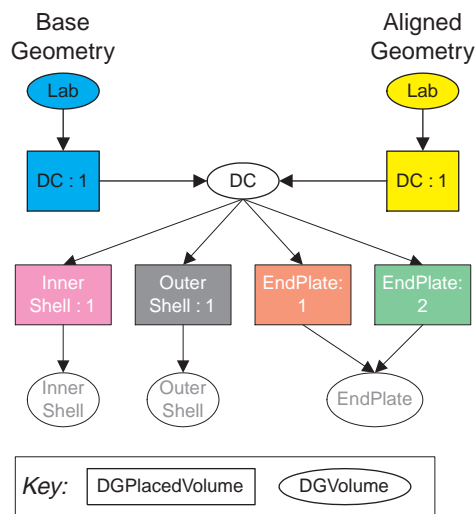


Figure 3: Sharing of objects between base and aligned geometry

The aligned description is constructed by first ‘copying’ the base description’s hierarchy. When copying, objects that will not change can be shared between the base and the aligned descriptions. Once the copy is complete, we set the local aligned to local nominal transformation of the aligned description. An example of sharing objects between base and aligned descriptions is shown in Figure 3. In the figure, only the position of the Drift Chamber (DC) mother volume is allowed to be ‘misaligned’. Since only the Drift Chamber mother volume’s *DGPlacedVolume* needs to be modified, the base and aligned descriptions can share the remaining hierarchy that comes after that placed volume.

We only need to build a new aligned description if the measured alignment data has changed and someone has requested the aligned description. The new CLEO III data access system allows us to handle both these conditions [2].

Checking and Usage

It is nearly impossible to check the correctness of a geometry description just by looking at the code or by printing out the lab coordinates of the various detector pieces. Therefore we developed an interactive geometry display program based on the Spectator Framework [3]. The program is a module that runs within the CLEO III data access system and can display the base or aligned geometry of any sub-detector. Using the geometry viewer, we were able to quickly and easily catch errors in the sub-detector descriptions.

The DetectorGeometry package is a general-purpose package for describing the placement of material in space. It is not optimized for any task, e.g. it has no built-in capability for calculating track intersections. We plan to use the detector description to build new structures that are optimized for specific tasks.

Monte Carlo: get parameters for GEANT3 build routine calls

Tracking: build structures optimized for intersection calculations

Event Display: build objects that can draw themselves

The CLEO III data access system makes sure that the proper algorithms are run when data on which they depend changes. For example, if there is a new aligned detector description then the system calls the routine that builds new track intersection structures.

Conclusion

As we expected, object oriented programming allows us to build geometry structures in a very natural way, i.e. a hierarchical structure made from abstract classes. We also found that a general-purpose package gives us the flexibility to deal with change. For example, we can use the DetectorGeometry package to describe the CLEO II, II.5, and III detectors. We also found that the only viable way to check the correctness of a geometry is to visualize it. Therefore a geometry visualization application must be available from the very beginning of a project. And finally, we are hopeful that using a general structure as a blueprint for constructing specialized structures will allow us to maintain one description of the detector but also be able to use optimized classes for our particular tasks.

Acknowledgements

This work is funded by the Department of Energy, grant DEFG0586ER40272.

References

- [1] www.phys.ufl.edu/~cdj/geometry/DetectorGeometry.html
- [2] Lohner, M., Jones, C.D., Patton, S.J., Avery, P.R., CHEP98 #194 "The CLEO III Data Access Framework"
- [3] Jones, C.D., Avery, P.R., CHEP98 #208 "Spectator: A Reusable Framework for Information Displays"