1 Geometry Kernel Classes

Joe Boudreau

1.1 About the Geometry Kernel Classes.

The geometry kernel classes are provided by the package `GeoModelKernel`. These classes provide a set of geometrical primitives for describing detectors, and a scheme for accessing both the raw geometry of a detector and arbitrary subsystem-specific geometrical services. The scheme provides a means of keeping the geometrical services synched to the raw geometry, while incorporating time-dependent alignments. It also allows one to version the geometry of any subsystem.

The design of these classes reflects the belief that raw geometry is highly constrained by the simulation engines, while the readout geometry is highly subsystem specific and has practically no constraint at all. The description of both types of geometry is normally to be carried out by a subsystem specialist.

This specialist is asked to extend objects called `GeoVDetector Element`, `GeoVDetectorManager`, and `GeoVDetectorFactory`, by writing subclasses describing both the raw and readout geometry of his or her subsystem.

Thus, the simulation engines available today (Geant3 and Geant4), which fortunately have a high degree of conceptual commonality, basically determine the format of the raw geometry. Every subsystem engineer who is responsible for describing a subdetector needs to provide one or more trees of raw geometry for the purpose of simulation. These tasks—creating and accessing raw geometry-- are required methods of the three basic base classes. The geometry kernel classes provide a set of geometrical primitives to support these operations..

In addition, the subsystem engineer has to layer, upon this raw geometry, any detector specific readout services required in simulation, reconstruction, or analysis. This is a very broad task and relies heavily on the creativity and intelligence of the subsystems specialist. The specialist is asked only to provide access to this type of information through the same class (`GeoVDetectorManager`) that accesses the raw geometry.

The set of Geo`VDetectorFactories` are then all called upon during the initialization phase to build both raw and readout geometries. During normal execution, messages to move various pieces of material to new, aligned positions will be routed from the calibration database to the detector managers which must respond by applying new alignment transformations at specific points in the geometry tree designated as alignable. The position of one or more pieces of raw geometry moves about when the alignable transformations are tweaked.

Readout geometry synchronizes itself to raw geometry by holding a pointer to a volume in the raw geometry tree that holds its absolute transformation with respect to world coordinates in cache. The readout geometry should access this information when responding to any queries about, or relying upon, its absolute position.

Thus the detector managers have a dual function: they describe the geometry (potentially misaligned) to the simulation, and they serve as a central store of detector-specific geometrical information which is accessed throughout Athena-based applications in ATLAS.

The interface to this information is largely up to the subsystem engineer. The Geo`VDetectorFactories` for each subsystem are called upon during the initialization of a service (`GeoModelSvc`) to construct geometry through a method called `create()`. They must provide access to the volumes so created, for the purpose of simulation. The `GeoModelSvc` then makes the Geo`VDetectorManagers` available to a variety of different clients.

1.2 Examples

In this section we give a few simple examples of how to use the geometry kernel. First, we illustrate how to get the information into the transient representation—this is the job of the subsystem engineer, for whom this section will be very important. Second, we illustrate how to get the information out of the transient representation—this will be important mostly for the individual who passes the description along to a procedure such as Geant3 or Geant4.. or one of may reconstruction tasks.

1.2.1 Example 1: Getting the data into the transient represention.

In this section we provide and illustrate a simple Geo`VDetectorFactory` subclass called a `ToyDetectorFactory`. This code describes a geometry that has 100 rings contained within a square box. The `ToyDetectorManager` contains two different types of readout elements: `CentralScrutinizers`, and `ForwardScrutinizers`. The header file for `ToyDetectorFactory` looks like this:

```
#include "GeoModelKernel/GeoVDetectorFactory.h"
#include "GeoModelExamples/ToyDetectorManager.h"
class ToyDetectorFactory : public GeoVDetectorFactory  {

    public:

   // Constructor:
   ToyDetectorFactory();

  // Destructor:
  ~ToyDetectorFactory();

  // Creation of geometry:
  virtual void create(GeoPhysVol *world);

  // Access to the results:
  virtual const ToyDetectorManager * getDetectorManager() const;

 private:

  // Illegal operations:
  const ToyDetectorFactory & operator=(const ToyDetectorFactory &right);
  ToyDetectorFactory(const ToyDetectorFactory &right);

  // The manager:
  ToyDetectorManager       *detectorManager;

};
```

From the header file, one can see that the subsystem engineer has created a class called ToyDetectorFactory, that derives from the class GeoVDetectorFactory, which is the base class for all subsystem-specific detector geometry factories. The ToyDetectorFactory is required to provide the following methods (because the base class declares them to be abstract functions):

```
virtual void create(GeoPhysVol *world);
```

Which builds the geometry within a containing physical volume (world volume).

The detector manager is returned from the factory and holds the entire geometry description for the subdetector. The header file for `ToyDetectorManager` is shown below:

```
#include "CLIDSvc/CLASS_DEF.h"
class ToyDetectorManager;
CLASS_DEF(ToyDetectorManager, 9876, 1)
#include "GeoModelKernel/GeoVPhysVol.h"
#include "GeoModelKernel/GeoVDetectorManager.h"
#include "GeoModelExamples/CentralScrutinizer.h"
#include "GeoModelExamples/ForwardScrutinizer.h"

class ToyDetectorManager : public GeoVDetectorManager

  public:
   enum Type {CENTRAL, FORWARD};

   // Constructor
   ToyDetectorManager();

   // Destructor
   ~ToyDetectorManager();

   // Access to raw geometry:
   virtual unsigned int getNumTreeTops() const;

   // Access to raw geometry:
   virtual PVConstLink getTreeTop(unsigned int i) const;

   // Access to readout geometry:
   const ForwardScrutinizer * getForwardScrutinizer(unsigned int i) const;

   // Access to readout geometry:
   const CentralScrutinizer * getCentralScrutinizer(unsigned int i) const;

   // Access to readout geometry:
   unsigned int getNumScrutinizers(Type type) const;

   // Add a Tree top:
   void addTreeTop(PVLink);

   // Add a Central Scrutinizer:
   void addCentralScrutinizer(const CentralScrutinizer *);

   // Add a Forward Scrutinizer:
   void addForwardScrutinizer(const ForwardScrutinizer *);

   private:

   const ToyDetectorManager & operator=(const ToyDetectorManager &right);
   ToyDetectorManager(const ToyDetectorManager &right);


   std::vector<const CentralScrutinizer *> centralScrutinizer;
   std::vector<const ForwardScrutinizer *> forwardScrutinizer;
   std::vector<PVLink> volume;
};
```

One sees from the interface that the manager is essentially a class that permits one to add and retrieve bits of detector description. Two methods, getNumTreeTops() and getTreeTop(unsigned int i), are required and are used to access the number of top-level physical volumes in the system and allow one to access sequentially each top-level physical volume. Physical volumes, essentially, are positioned pieces of material with specific shape and composition. They are explained below in more detail. The raw geometry is organized in a treelikle structure, and the detector managers must provide the top-level branch in the tree. The third method in the toy detector node creates the tree of volumes. We shall see in detail how, shortly.

The last three methods are not required but are provided by the subsystem engineer to describe pieces of readout or other detector-related geometrical information.

```
unsigned int getNumScrutinizers(Type type) const
const ForwardScrutinizer *getForwardScrutinizer(unsigned int i) const
const CentralScrutinizer *getCentralScrutinizer(unsigned int i) const
```

The last three methods give access to readout geometry. The basic pieces of readout geometry in the ToyDetectorManager are called ForwardScrutinizer and CentralScrutinizer. They derive from a base class called GeoVDetectorElement, which stores and provides access to a pointer to a GeoFullVPhysVol (this is a physical volume with an absolute global-to-local coordinate transformation in cache).

What kind of geometrical object are the scrutinizers? They are meant to illustrate pieces of detector with both material and readout properties. For example, in the inner detector, instead of a “Scrutinizer” one would create perhaps a pixel detector, giving the pixel detector the properties of readout pitch along local x and y, number of channels in x and y, and perhaps a multiplexing scheme. The vectors normal to the each side of the pixel detector could be provided through the pixel detectors’s interface—if that is a useful geometrical service for the pixel detector to provide—and could be computed from the full physical volumes absolute global-to-local coordinate transformation information. In the case of a calorimeter, the “Scrutinzers” would be replaced with a class describing a calorimeter module that could describe the peculiar way in which signals were summed within the calorimeter slices. And so forth.

Looking again at the interface to `ToyDetectorManager` and `Factory`: we wish to disable copying and assignment so we make these methods private and leave them unimplemented. We also declare some private member data required to carry out the services described above: a vector to hold the top level physical volumes, and two more to hold the lists of forward and central scrutinizers. Next we shall see how to implement this detector factory.

The implementation of the `ToyDetectorFactory` is shown below. Note how the factory creates both raw geometry and readout geometry and puts it in the manager. In principal, one can tailor the code so that the detector factory itself determines the shape of the whole detector geometry, so that alternate geometries can be constructed simply by creating different types of factories and using them at run time.

The ToyDetectorFactory shown below is a simplified version of actual code that can be found in the Atlas repository. This simplified version does not contain illustration of certain advanced features—namely, access to the material manager, interface to Athena, insertion of the managers within Storegate, and parametrization of volumes using `GeoSerialTransformer` (see section 2.7)–that are present in the full version[i].

```cpp
#include "GeoModelExamples/ToyDetectorFactory.h"
#include "GeoModelExamples/CentralScrutinizer.h"
#include "GeoModelKernel/GeoMaterial.h"
#include "GeoModelKernel/GeoBox.h"
#include "GeoModelKernel/GeoTube.h"
#include "GeoModelKernel/GeoLogVol.h"
#include "GeoModelKernel/GeoNameTag.h"
#include "GeoModelKernel/GeoPhysVol.h"
#include "GeoModelKernel/GeoFullPhysVol.h"
#include "GeoModelKernel/GeoTransform.h"
#include "GeoModelKernel/GeoSerialDenominator.h"
#include "GeoModelKernel/GeoAlignableTransform.h"

ToyDetectorFactory::ToyDetectorFactory()
   :detectorManager(NULL){}

ToyDetectorFactory::~ToyDetectorFactory()
{}
const ToyDetectorManager * ToyDetectorFactory::getDetectorManager() const {
  return detectorManager;
}

//## Other Operations (implementation)
void ToyDetectorFactory::create(GeoPhysVol *world)
{
   detectorManager=new ToyDetectorManager();
  //-----------------------------------------------------------------------------//
  // Get the materials that we shall use  (material manager from Storegate!) //
  // ----------------------------------------------------------------------------//
  const GeoMaterial *air  = materialManager->getMaterial("std::Air");
  const GeoMaterial *poly = materialManager->getMaterial("std::Polystyrene");
  // Next make the box that describes the shape of the toy volume:
  const GeoBox      *toyBox    = new GeoBox(800*cm,800*cm, 1000*cm);
  // Bundle this with a material into a logical volume:
  const GeoLogVol   *toyLog    = new GeoLogVol("ToyLog", toyBox, air);
  // ..And create a physical volume:
  GeoPhysVol        *toyPhys   = new GeoPhysVol(toyLog);
  // Add this to the list of top level physical volumes:
  detectorManager->addTreeTop(toyPhys);
  //   Daughters:
  const GeoTube     *ringTube  = new GeoTube(500*cm, 1000*cm, 5.0*cm);
  // Bundle this with a material into a logical volume:
  const GeoLogVol   *ringLog  = new  GeoLogVol("RingLog", ringTube, air);
  // Make 100 of these within the volume of the toy:
  GeoSerialDenominator       *ringName = new GeoSerialDenominator("RING");
  toyPhys->add(ringName);
  for (int i=0;i<100;i++) {
    GeoFullPhysVol          *ringPhys = new GeoFullPhysVol(ringLog);
    GeoAlignableTransform   *xf = new GeoAlignableTransform(
                                    HepTranslateZ3D((i-50)*20*cm));
    toyPhys->add(xf);
    toyPhys->add(ringPhys);
    detectorManager->addCentralScrutinizer(new CentralScrutinizer(ringPhys));
   }
}
```

1.2.2 Example 2: Getting the data out of the transient representation.

1.3 An Overview of the Geometry Kernel

In this section we give a short overview of all of the pieces of the geometry kernel. These pieces are described in detail in section 2. In this section our goal is to describe the "big picture". The `GeoModel` class tree is shown in below.

```
RCBase----GeoLogVol
      |---GeoMaterial
      |---GeoElement
      |---GeoShape--------GeoShapeSubtraction
      |             |-----GeoShapeIntersection
      |             |-----GeoShapeUnion
      |             |-----GeoShapeShift
      |             |-----GeoBox
      |             |-----GeoCons
      |             |-----GeoPara
      |             |-----GeoPgon
      |             |-----GeoTrap
      |             |-----GeoPCon
      |             |-----GeoTube
      |             |-----GeoTrd
      |
      |--GeoGraphNode-----GeoNameTag
                    |-----GeoSerialDenominator
                    |-----GeoTransform-----------GeoAlignableTransform
                    |-----GeoVPhysVol------------GeoVFullPhysVol-------GeoFullPhysVol
                    |                |-----------GeoPhysVol
                    |
                    |-----GeoSerialTransformer

GeoNodeAction---------GeoCountVolAction
             |---------GeoAccesssVolumeAction
             |---------GeoClearAbsPosAction

GeoVolumeAction-------TemplateVolAction
```

And also:

```
GeoPath, GeoTraversalState, Query<class T>, GeoAbsPosInfo,  GeoXF::Pow
```

Figure 1: The GeoModel Class Tree

Many of the classes in the library represent objects which are reference counted; these all inherit from `RCBase`. Others represent geometrical shapes; these inherit from `GeoShape`. Others represent objects that can be assembled into a geometry graph; these inherit from `GeoGraphNode`.

1.3.1 The Detector Store Service and Detector Managers

The detector store service is not part of `GeoModel` per se, but rather an interface from `GeoModel` to Athena and Storegate. It is a Storegate service running within Athena and providing access to all detector information. The service can be accessed in the following way, which is typical of all Storegate services:

```
StoreGateSvc *detStoreSvc;
StatusCode status = service("DetectorStore",detStoreSvc);
```

The service hold several important objects. The first is the world physical volume, the common ancestor of all physical volumes within the system. This object has type `GeoModelExperiment`, which is a Storegate-compatible physical volume. It can be accessed like this:

```
const DataHandle<GeoPhysVol>  world;
StatusCode status = detStoreSvc->retrieve(world,"ATLAS");
```

From there, one may navigate the physical volume tree. The other objects that one can access through the detector store are the detector nodes, which are the master copy of all readout information. For example, for the liquid argon calorimeter, this might look like this:

```
const DataHandle<AbsLARDetectorNode> *laRNode;
StatusCode status = detStoreSvc->retrieve(laRNode,"LAr");
```

The strings used to retrieve detector nodes are assigned subsystems engineers. No catalogue can be published at this time. The detector factories are created by an interface called a tool, which instantiates the detector, and causes it to build its geometry within the world physical volume, and then also records the readout geometry within the detector store. The class `ToyDetectorTool` provides an example. It is in the source tree, under `DetectorDescription/GeoModel/GeoModelExamples`.

1.3.2 Material Geometry

Material geometry consists of a set of classes that bears a large resemblance to the material geometry within some flavour of GEANT. These classes, however, are designed to take a minimal size in memory. This requirement determines the basic data structure used to hold the data for the geometry description. That structure is a graph of nodes consisting of both volumes and their properties. The tree is built directly and accessed in a way that provides users access to volumes and, simulataneously, to the properties accumulated during graph traversal that apply to the volumes. See the *Actions* section, below.

The requirement of minimizing the memory consumption has led us to foresee a system in which objects (as well as classes) in the detector description can be re-used. This is called shared instancing, and is described below. It essentially means that an element, compound, volume, or entire tree of volumes may be referenced by more than one object in the detector description. Shared instancing can make the deletion of objects difficult unless special measures are taken. We have used a technique called reference counting in order to facilitate clean-up and make it less error prone. Using that technique, objects can be created using operator new. The memory is then freed when some action is taken to clean up near the top of the tree. See the section *How Objects are Created and Destroyed*.

Before creating hierarchies of volumes representing positioned pieces of detectors, we need to create lower level primitives, such as elements, materials, and shapes. So, we will discuss these first.

1.3.2.1 Materials

Materials are represented within the geometry kernel class library by the class `GeoMaterial`, and are built up by combining different elements, specifying each element and its fraction-by-mass. Material constants such as the radiation length and the interaction length, as well as constants for ionization energy loss, are available through the interface but do not need to be provided to the constructor. Instead, they are computed from the material's element list.
The class `GeoElement` is used to represent elements. Their constructor requires a name, symbol, and effective Z and A. These properties can also be retrieved from the element.

`GeoMaterial` objects are created by specifying a name and a density. The material is "empty" until elements are added, one by one, using the `add()` method, which is overloaded so that one may provide either elements or prebuilt materials. After all materials are added, the `lock()` method must be called, which prevents further elements or materials from being added.

Material classes, as well as all other classes, use the CLHEP Units wherever applicable. One should normally give units when specifying densities, lengths, volumes, or other quantities in the methods of all of the classes in this library. Therefore, when specifying water, one should use a constructor call like this:

```
GeoMaterial *water = new GeoMaterial("H20", 1.0*gram/cm3);
```

The CLHEP Units are described on the CLHEP web page[ii]. To finish constructing this material, water, one needs to follow the constructor with the following lines:

```
GeoElement *hydrogen = new GeoElement("Hydrogen","H",1.0, 1.010);
```

```
GeoElement *oxygen   = new GeoElement("Oxygen",  "O", 8.0, 16.0);
water->add(hydrogen,0.11);
water->add(oxygen,0.89);
water->lock();
```

The materials are then used to together with shapes to form logical volumes, discussed below.

1.3.2.2 Shapes

Shapes are created using the new operator. Essentially, shapes within this system are required to store and provide access to the geometrical constants that describe their geometrical form. This data is, insofar as possible, to be specified on the constructor. Shapes are extensible and we intend to service requests for extensions, by providing custom shapes to valued customers on request[iii].

Here is how one builds a box:

```
double        length=100*cm, width=200*cm, depth=33*cm;
GeoBox *box = new GeoBox(length, width, depth);
```

Most objects can be constructed along similar lines; exceptions are objects with multiple planes such as polycones and polygons; their interface allows one to add planes successively. For the polycone, for example, the shape is built as follows:

```
double dphi=10*degrees, sphi=40*degrees;
GeoPcon  *polycone=new GeoPcon(dphi,sphi);

double z0=0.0, rmin0=5, rmax0=10.0;
polycone->addPlane(z0,rmin0,rmax0);

double z1=10.0, rmin1=6, rmax1=12.0;
polycone->addPlane(z1,rmin1,rmax1);

double z2=15.0, rmin2=5, rmax2=10.0;
polycone->addPlane(z1,rmin1,rmax1);
```

This creates a polycone whose projection subtends an angle of 10 degrees between 40 degrees anf 50 degrees, with planes at z=0, z=10, and z=15, with minimum and maximum radii there of (5,10), (6, 12), and (5,10).

The shapes can provide their data to a client through their accessors, and in addition support several other operations. Boolean operations on shapes are possible. They can be accomplished through Boolean operators in class `GeoShape`:

```
GeoShape       * donut = new GeoTube();
GeoShape       * hole   = new GeoTube();
const GeoShape & result = (donut->subtract(*hole));
```

The result of a Boolean operation is a shape in a boolean expression tree that can, for example, be decoded when the geometry is declared to GEANT.

Another method that shapes can carry out is to compute their volume. This is useful in the context of mass inventory, in which the mass of the detector model is computed, usually for the purpose of comparing with an actual installed detector.[1] One needs to call the `.volume()` method which is defined for all shape types.

Finally, we mention a type identification scheme for shapes. The scheme relies on two static and two virtual methods which together can be used as follows:

```
// Test if the shape is a box:
if (myShape->typeId()==GeoBox::classTypeId()) {
     .....
}
```

The methods `typeId()` and `classTypeId()` return unsigned integers, making the type identification very fast. Alternately one can use the methods `type()` and `classType()`, which work in the same way, except that these methods return std::strings: “Box”, “Tubs,” “Cons,” etc.

1.3.2.3 Logical Volumes

[1] Presently not implemented on Boolean operations. However this functionality will soon be incorporated. The authors wish to thank Evgueni Tchernaiev for this contribution.

Logical volumes represent, conceptually, a specific manufactured piece that can be placed in one or more locations around the detector. A logical volume is created by specifying a name tag for the volume, a shape, and a material:

```
const GeoLogVol   *myLog  = new  GeoLogVol("MyLogVol",
                                          myShape,
                                          gNitrogen);
```

1.3.2.4 Physical Volumes and the Geometry Graph

Having created elements, materials, shapes, and logical volumes, you are now ready to create and locate placed volumes called physical volumes. Before you start, you will need to know that there are two kinds of these:

- Regular Physical Volumes, designed to be small.
- Full Physical Volumes, designed to hold in cache complete information about how the volume is located with respect to the world volume, its formatted name string and other important information.

There is a common abstract base class for all of these: GeoVPhysVol. In addition both the full physical volumes have another layer of abstraction, GeoVFullPhysVol, in order to allow us to introduce parametrized volumes in the near future. All physical volumes allow access to their children.

The concrete subclasses that you have at your disposition for detector description are called GeoPhysVol and GeoFullPhysVol. Both of these have a method to add either volumes or volume properties.

```
GeoPhysVol *myVol;
myVol->add(aTransformation);
myVol->add(anotherVolume);
```

When you add a transformation, you change the position of the subsequent volume with respect to the parent. If you add no transformation, you will not shift the daughter relative to the parent and commonly will create a daughter which is centered directly in the parent. If you add more than one transformation to the volume before adding a parent, they will be multiplied. The last transformation to be added is applied first to the child. Transformations are discussed next. Like logical volumes, they may be shared.

Like physical volumes, transformations come in two types:

- Regular transformations, designed to be small.
- Alignable transformations, which allow one to add a misalignment to the system. Misaligning a transformation changes the position of all volumes "under" the transformation and clears the absolute location caches of all full physical volumes.

When you create a transformation you must choose the type.

The model of the raw geometry is a tree of nodes, property nodes and volume nodes. The tree can be thought of as as tree of volumes, each one "having" a set of properties (inherited from property nodes throughout the tree). The subsystem engineer judiciously chooses which of the volumes are to contain full, cached, position information – usually, these first-class volumes are to be associated with a detector. He or she also judiciously decides which of the transformations are to be alignable—usually these are the transformations which position something that ultimately has a detector bolted, glued, riveted or otherwise clamped onto a sensitive piece. Then, a Geo`VDetectorFactory` which builds the geometry keeps track of these pointers so that it may connect the important volumes to detector elements and that it may connect the alignable transformations to the alignment database for periodic updating.

Finally, we provide three mechanisms for giving names to volumes:

- Do nothing. The volume will be called "ANON".
- Add a `GeoNameTag` object to the graph before adding a volume. The next volume to be added will be given the `GeoNameTag`'s name.
- Add a `GeoSerialDenominator` object to the graph before adding more volumes. The volumes will be named according to the base name of the `GeoSerialDenominator`, plus given a serial number 0, 1, 2, 3…..

In effect this last method can be thought of as a way of parametrizing the name of the volume.

1.3.2.5 Actions

There are two ways of getting raw geometry information out of the model. Suppose that one has access to a particular physical volume (it could be the "World" physical volume). One can access its children, there names, and their transformations with respect to the parent in the following way:

```
PVConstLink myVol;
for (int c=0; c< myVol->getNChildVols();c++) {
      PVConstLink child = myVol->getChildVol(c);
      HepTransform3D  xf = myVol->getXToChildVol(c);
}
```

One could then iterate in a similar way over the grand children, by using a double loop. Ultimately one would probably to visit all the volumes, whatever their depth in the tree, so probably this would call on some form of recursion. An easy way would be to embed the small sample of code shown above in a recursive subroutine or method. That would be fine, and is conceptually simple. However, within the geometry model's kernel, we have provided an alternate, probably better way to visit the entire tree.

That mechanism involves a `GeoVolumeAction`. A `GeoVolumeAction` is a way (for applications programmers) to obtain recursive behavior without writing any recursive routines. It's a class with a handler routine (`handleVPhysVol`) which is called for each node before (or after) it is called on its children. This can descend to an arbitrary depth in the tree. The `GeoVolumeAction` is an abstract base class and should be subclassed by programmers to suit their needs. Another class `TemplateVolAction` is provided as a template that one can take and modify. To run it, one does this:

```
PVConstLink myVol;
TemplateVolAction tva;
myVol->apply(&tva);
```

The `handleVPhysVol` within the `TemplateVolAction` is where the work is supposed to get done. It will be invoked repeatedly, once for each node in the tree. Within that routine, one can access the physical volume as a subroutine parameter, and information about the transformation and the path to the node through the base class for actions, `GeoVolumeAction`. The action can be designed to run from the bottom up or from the top down.

Incidentally, there is another kind of action in the library called `GeoNodeAction`. `GeoNodeActions` visit all nodes (including naming nodes, transformation nodes, and perhaps other property nodes that may be added later to the model) Since usually an application programmer wants to see volumes and their properties, the `GeoVolumeAction` is more suited to casual users than the `GeoNodeAction`, which is considered mostly internal. However the usage is similar, except that node actions are "exec'd" while volume actions are "applied". Here for example is how we can rewrite the loop over children using volume actions:

```
PVConstLink myVol;
for (int c=0; c< myVol->getNChildVols();c++) {
      GeoAccessVolumeAction av(c);
      myVol->exec(&ac);
      PVConstLink  child = ac.getVolume();
      HepTransform3D  xf = ac.getTransform();
}
```

This, it turns out, will execute faster than the loop shown above, which (internally) will run the action, twice: once, in order to locate the daughter volume and then a second time, to locate its transform.

1.3.2.6 How Objects are Created and Destroyed

We now come to the important topic of how objects in this system are created and destroyed. The geometry kernel uses a technique called reference counting. Reference counting, shortly stated, is a way to perform an automatic garbage collection of nodes that are no longer in use. This is important when describing a large tree of information, much of which is ideally to be shared—used again and again in many places.

You may have noticed, in the section "Example 1: Getting the data into the transient represention.," that many of the objects have been created using operator `new`. You may have also noticed, if you've tried to play around with the kernel classes, that statements which allocate most kernel classes on the stack, such as:

```
GeoBox box(100, 100, 100);
```

are not allowed. Who is going to clean up the memory after all these `new` operations? And why does the compiler disallow allocation on the stack?

Let's look again at Example 1, especially at these lines:

```
const GeoBox       *worldBox    = new GeoBox(1000,1000, 1000);
const GeoLogVol    *worldLog    = new GeoLogVol("WorldLog",
                                     worldBox, gNitrogen);
GeoPhysVol         *worldPhys   = new GeoPhysVol(worldLog);
```

Each of the three objects (worldBox, worldLog, and worldPhys) are created with a reference count. WorldBox's is initially zero, at the time it is created. WorldLog's is also zero when it is created. However, when worldVol is created, the reference count of worldBox increases to one, since now it is referenced somewhere—namely by the logcal volume worldLog. We can diagram this sequence in the following way:

Now, when the physical volume worldPhys is created, the reference count of the logical volume will increase to one—since it is used once by a single physical volume.

Each time a physical volume is positioned within another physical volume, its reference count increases. Suppose we look now at a sub-tree of physical volumes that is used five times. At a run boundary, it may happen that a piece of the tree is torn down. When the first node referencing the physical volume is destroyed, it decreases the volumes reference count, from five to four. When the next node referencing the physical volume is destroyed, the reference count goes from four to three. And so forth.

When the very last node referencing the physical volume is destroyed, this means that the physical volume itself has outlived its usefulness and *should disappear*. And that is what happens. The destruction of objects is carried out automatically when the reference count falls to zero. And in fact, the only way to delete an object is to arrange for all of its references to disappear. This is because the destructor of all reference counted objects is private.

This scheme applies to elements, materials, shapes, logical volumes, physical volumes, full physical volumes,

So far, we have described what happens to an object when it is no longer used by any other node in the tree. However, what about the top of the tree, which has no nodes that refer to it? Since the destructors of our physical volumes are private, how do you arrange to get it to go away?

Reference counts can also be manipulated manually, by using the methods ref() and unref(). The physical volume at the head of the tree, often known as the "world" physical volume, can be referenced manually using this call:

```
worldPhys->ref();  //reference count goes from 0 to 1.
```

Later, you can destroy the world volume and trigger a global collection of garbage by using this call:

```
worldPhys->unref();//reference count goes from 1 to 0.
```

When this happens the world physical volume deletes itself, decreasing the reference counts of it logical volumes and any children. These will then begin dereferencing and possibly deleting their own children, until all the memory has been freed.

Suppose now, that you want to arrange for a node to not be deleted automatically in this fashion—even when nobody references it any more. In order to do this, simply call the ref() method on this object. That way, the reference counts starts at 1 and will not fall to zero until you call unref(), manually.

1.3.3 Detector Specific Geometrical Services

Detector specific geometrical services are known to some as "readout geometry". This consists, first and foremost, of geometrical information that is not declared directly to the tracing engines, G3, for example, or G4. Examples would include: projective towers within a calorimeter, or implant regions within a piece of silicon. Information such as the position of the boundaries of these regions is not required in the simulation of basic physics processes, though it certainly is required in the digitization, and possibly hit-making phase of simulation.

Detector-specific geometrical services can and should include services that derive from the basic raw and readout geometry of the detector. Such services could include point-of-closest-approach calculations, global-to-local coordinate transformations, calculations that compute the total number of radiation lengths within a cell, et cetera. Additionally they could include nearest-neighbor calculations, hopefully in a highly detector specific way which is meaningful in the context of specific algorithms.

We have intended that this kind of service would be provided by the subsystem engineer, or somebody with an intimate knowledge of both the detector geometry and the requirements of hit simulation and/or reconstruction in the detector. This kind of service, ideally, would be spread across at least two classes.

The first place is in the detector element. The detector element (subclass of Geo`VDetectorElement`) has a required association with a piece of material geometry, and has access to that piece. The rest of the interface—all of the geometrical services discussed above, such as the boundaries of implant layers, strip pitches, whatever, can be placed in the detector element.

The second place where detector specific geometrical services may be placed is in the interface to the the detector manager (subclass of Geo`VDetectorManager`), which constructs and manages all raw and readout geometry. This class should provide a fast mechanism for accessing the detector elements that it manages—such as detector-specific, array-based random access. Other services, such as returning the maximum and minimum range of some array index (phi, eta, etc.) may also be appropriate.

So in general the subsystems people have a lot of flexibility, but need to devise an interface to both the detector manager and the detector element that satisfies their needs. The exact layout of these classes is hopefully the object of some design on the part of the engineer, can evolve with experience to involve a larger category of collaborating classes.[2] The basic framework requires only that 1) detector factories create a physical volume tree, 2) they associate readout elements to certain physical volumes, and 3) additional readout information appear in the interface to the detector manager and the detector element.

1.3.4 Alignment

There are two alignment issues we need to address: first, how does the `GeoModel` propagate alignment constants into the geometry description? Second, how is the subsystem engineer supposed to connect the alignment constants to the database so that the geometry changes when the run conditions are updated? The first issue concerns the way that `GeoModel` works, the second issue is mostly a policy question and outside the scope of `GeoModel`, per se.

[2] In certain CDF subdetectors, for example, all questions involving numerical limits to array boundaries were ultimately handled separately by a "numerology" classes, available through the detector node.

GeoModel has a natural way of putting alignment constants into the geometry description and a natural way of getting them out. To put them it, one alters one or more GeoAlignableTransform objects by changing its "Delta", or misalignment, which is a HepTransform3D. The misalignment is then composed with the default transformation.

To get the alignment out of GeoModel, simply query a physical volume for its transformation. All physical volumes have the notion of relative and absolute transformations, both default and (mis)aligned. Full physical volumes cache the absolute transformation, making it immediately available after the first request, while ordinary physical volumes compute it anew each time during tree traversal. In either case, GeoModel methods supply an answer that correctly incorporates the effects of misalignment.

In case of cached transformations, it's worthwhile to describe the mechanism by which the cache is updated. First, when an alignable transform is altered, all parent physical volumes receive a message to clear any caches. These messages are passed onto their daughter volumes, and any physical volume in the geometry tree that contains a cache of absolute transformations is cleared. Then, as soon as some client requests a transformation, it is recomputed recursively, starting from the first parent with valid cache information, and again cached.

A piece of readout geometry (class GeoVDetectorElement) cannot be constructed without a full physical volume. One constructed, it always has access to that volume's transformation. Readout geometry should respond to all queries relevant to absolute spatial positioning by referencing the absolute transformation of the physical volume. See section 2.9 for more details.

Finally, how should the subsystem engineer arrange for the geometry to be updated when run conditions change? The basic suggestion is to use the notification mechanisms of the calibration database. In this scheme, engineer should arrange for the detector manager to receive a message when some relevant database table has changed. The detector manager should then rescan the tables, construct new "Delta's" for each alignable transform under its jurisdiction, and alter those transforms.

When updates occur, readout elements may be required to update any local cache of information that derives, ultimately, from alignment constants. This can be arranged using the same notification mechanism.

For the moment no documentation on the calibration database can be cited. The need for this component is not considered urgent as of this writing.

1.3.5 On Memory Use

Some effort has been spent insuring that the memory used by the a geometry description can be made small, and indeed, it is our belief that using the techniques made possible by this class library a remarkably compact description of the geometry can be achieved. However a compact geometry will not occur automatically. Users need to know what tricks are available, and need to apply them as aggressively as possible.

If aggressive optimization of process size is done, across the board and from the beginning, we think that the GeoModel geometry description could contribute a negligible amount to the overall process size of a typical ATLAS executable.

This goal is worth working towards, for three reasons. First, if the process size is really negligible, then ATLAS executables can instantiate and use the whole geometry description, including even material geometry, at virtually no cost.

Second, it will mean that at GeoModel description could be kept alive even after the whole model has been declared to a simulation engine, such as GEANT3/4.

Third, experience shows that process size becomes unmanageable in large-scale projects unless the memory cost is carefully controlled from the beginning.
Here are some suggestions for how to minimize the size of the geometry description, in memory:

- Share instances of elements, materials, shapes, logical volumes and physical volumes, and even transformations.
- Use full physical volumes and alignable transforms only where necessary.
- Do not give names to physical volumes that represent uninteresting, nondescript pieces of material.
- In case you need to give names to physical volumes, use a serial denominator rather than multiple name tags.
- Parameterize volumes where possible.

The best way of sharing instances of elements and materials is to create them within a dedicated service and access that service, experiment-wide, for any materials that are required to construct the geometry. Logical volumes and shapes should be simple to share if adequate care is taken. Shared instancing and parameterization of physical volumes is limited mostly by the constraints that:

- Physical volumes representing active elements must be "full" and distinct, since they exist to cache an absolute position. This means that they must not be shared, or parameterized, nor live in any branch of a physical volume tree which is shared or parameterized.

Finally, transformations could be shared by creating a bank of common transformations such as common ϕ rotations and reusing them instead of instantiating, say, a 15^{o} rotation hundreds of times. When shared instancing of transformations works, however, parameterization will also usually work and is generally a better solution. Note, parameterizing volumes in GeoModel does not mean that G4 parameterization must be used during simulation. We can and should make this optional.

Not all of the planned optimization tools are available in this release. Notably, parameterization of shapes (as opposed to transformations, only) has not been implemented, and a compressed representation for CLHEP transforms is not available. We foresee adding both of these features to the library at a later date. The first feature will give certain clients more powerful parameterization techniques, such as distortion fields which are needed ultimately by the liquid argon calorimeter software; while the second feature will allow a global reduction in memory cost in a way which is virtually transparent to the users.

2 Geometry Kernel Class Reference

This section describes in more detail the classes in the geometry kernel. In most cases we provide the class interface. In cases where part of the interface is used only internally by the geometry kernel itself and not by other users. In such cases we present a simplified picture of the interfaces.

Detailed descriptions of the geometry kernel classes follow.

2.1 Reference counting

Many objects need to be allocated in the description of a complicated geometry. For efficient use of memory, these should be shared as often as possible. The extensive sharing of objects in a geometry system calls for a way of destroying the objects as soon as they are not used—by any other object that references them. The scheme for doing this is called reference counting. In the geometry kernel, it is achieved by mixing in a abstract base class, `RCBase`:

RCBase

Constructor:

RCBase()

Public Methods:

void ref ()
void unref ()
unsigned int refCount ()

`RCBase` is inherited by many of the classes in the geometry system. See Figure 1. Reference-counted objects can only be created using `operator new`, and cannot be created on the stack. The methods `ref()` and `unref()` can be called to increase or decrease the reference count of an object, When the reference count decreases to zero, the object deletes itself. The accessor `refCount()` returns the current reference count.

2.2 Materials

Two classes are used for describing materials: `GeoElement` and `GeoMaterial`. Elements are declared by specifying a name, chemical symbol, and atomic number and mass; the latter being specified in atomic mass units. Materials are constructed with a name and density and are not ready-for-use until one or more elements have been added, e.g.:

```
water->add(hydrogen,0.11);
water->add(oxygen,0.89);
water->lock();
```

The `add()` method takes an element and its mass fraction, and is overloaded to accept also a material and its mass fraction. The `lock()` method protects the material against further addition of elements, and re-normalizes the mass fractions so that they sum to unity.

The material responds to various queries about its composition, and in addition can return a series of derived quantities of physical interest. The interaction length and radiation length of a material are familiar and are described in the particle data book[iv].

Ionization energy loss in materials follows the Bethe-Bloch formula and is governed by two constants, an overall normalization term and the ionization potential, which can be computed from the atomic number and density of the material; the calculation is also described in the particle data book. The calculation does not include small corrections to the energy loss due to chemical binding of elements. These constants are available through the methods `getDeDxConstant()` and `getDeDxI0()`. The method `getDeDxMin()` returns the minimum ionization energy loss and is derived from the other methods.

Both materials and elements are reference-counted; the reference count of an element is incremented when it added to a material and decremented when a referencing material is destroyed; materials are reference counted when they are used in logical volumes and decremented when the referencing logical volume is destroyed.

GeoElement

Constructor:

GeoElement (const std::string & Name, const std::string & Symbol, double Z, double A)

Public Methods:

std::string getName() const
std::string getSymbol() const
double getZ() const
double getA() const
double getN() const

`GeoElement` has a constructor which takes a name, a chemical symbol, and atomic number, and an atomic weight[3]. The public methods provide access to this information. The `getN()` method returns the effective number of nucleons in the material, Z+A.

GeoMaterial

Constructor:

GeoMaterial (const std::string & Name, double Density) const

Public Methods:

void add (GeoElement * element, double fraction = 1.0)
void add (GeoMaterial * material, double fraction)
void lock ()
std::string getName () const
double getDensity () const
unsigned int getID() const
unsigned int getNumElements () const
const GeoElement * getElement (unsigned int i) const
double getFraction (int i) const
double getRadLength () const
double getIntLength () const
double getDeDxConstant () const
double getDeDxI0 () const
double getDeDxMin () const

GeoMaterial is a class that describes a material, which is a list of elements. It is created "empty"; subsequently, elements are added one-by-one until the material is locked. When the material is locked, no further editing is possible, and a series of derived quantities (radiation length, interaction length, etc.) is computed for the material.

GeoMaterial (const std::string & Name, double Density) const Constructs the material with a name and a density4.

void add (GeoElement * element, double fraction = 1.0) Adds an element to the material, with a specified mass fraction.

[3] The atomic weight should be specified, in ATLAS, using CLHEP units (such as g/cm3).

void add (GeoMaterial * material, double fraction) Adds a material to the material, with a specified mass fraction. This is useful for combining precombined materials, such as argon + ethane.

void lock () Locks the material against further editing, and computes all derived quanties such as radiation length and interaction length.

std::string getName () const Accesses the name of the material.

double getDensity () const Returns the density of the material[4].

unsigned int getID() const Returns the id of the material. The id is generated automatically by counting instances of materials.

unsigned int getNumElements () const Returns the number of elements in a material.

const GeoElement * getElement (unsigned int i) const Returns a pointer to the ith element.

double getFraction (int i) const Returns the fraction of the ith element.

double getRadLength () const Returns the radiation length of the material, computed from the density, the list of constituents, and their properties.

double getIntLength () const Returns the interaction length of the material, computed from the density the list of constituents, and their properties.

The following methods refer to ionization energy loss, specifically, the following formulation:

$$\frac{dE}{dx} = \frac{K}{\beta^2}(\ln(\frac{2m_e c^2 \beta^2 \gamma^2}{I_0}) - \beta^2)$$

[4] The density is normally specified using CLHEP units. The native unit of mass is the MeV, the native unit for length is the mm. A material with a density of 1 g/cm3 has a density of 1.7 x 10 $^{-22}$, in these units.

double getDeDxConstant () const Returns the constant, *K*, which depends upon the material properties (mostly the density).

double getDeDxI0 () const Returns the effective ionization potential I_0, which is a property of the material.

double getDeDxMin () const Returns an approximation for the ionization of a minimum ionizing particle (βγ=3.4), given by: *K**11.528

2.3 Shapes and Logical Volumes.

The shape classes in the geometry kernel are data structures designed to describe several geometrical primitives. Table 1 describes the different shapes presently provided within the geometry kernel. This set is extensible; one only needs to derive a new shape from the base class and insure that it fits the pattern described below. Shapes are reference-counted objects, as described in1.3.2.6.

GeoBox	Box
GeoCons	Cone Section
GeoPara	Parallelapiped
GeoPcon	Polycone
GeoPgon	Polygon
GeoTrap	Trapezoid (complex)
GeoTrd	Trapezoid (simple)

GeoTube	Tube
GeoTubs	Tube Section

Table 1: Existing geometrical shapes in the geometry kernel.

All shapes provide access to their geometry attributes (height, width, & cetera), and in addition perform several other services:

- They calculate their volume
- They can combine themselves using Boolean operations
- They allow themselves to be identified through a built-in type identification scheme.

The volume calculation is an analytic calculation provided by each subclass.

One or more Boolean operation upon shapes creates a binary expression tree. This tree can be navigated later and the Boolean volumes declared to clients who can cope with them: GEANT4, notably, and certain visualization systems. Several Boolean operations may be combined in a single line of code:

```
GeoShape *A, *B, *C;
const GeoShape & D = (*A).add((*B) .subtract (*C));
```

A shape's reference count is incremented either when the shape is used by a `GeoLogVol`, or in a Boolean expression. In the above example, `D` has been `new`'d, so has an unnamed temporary. The reference count of the temporary is incremented when it is combined with `A` to make `D`. When `D`'s reference count falls to zero, `D` is deleted, and the temporary is deleted.

Shapes can also be shifted about before they are used in a Boolean operation. The operation looks like this:

```
GeoShape *A, *B;
HepTransform3D offset;
const GeoShape & D = A->subtract (B<<offset);
```

The type identification scheme works by comparing the result of a static method with the result of a pure virtual method:

```
GeoShape *A;
if (A->typeId() == GeoBox::GetClassTypeId() ) {

       //
       // It's a box!
       //
}
```

Both methods return a coded integer. When the class returns the same integer as the object, a match has occurred. Alternately one can use the methods `getClassType()` and `type()`, which return meaningful strings like "Box", "Cons"...These are human-readable but slower to compare than unsigned integers.

All `GeoShapes` have the following interface:

GeoShape

Public Methods:

virtual double volume () const
const GeoShapeUnion & add () const
const GeoShapeSubtraction & subtract(const GeoShape & shape) const
const GeoShapeIntersection & intersection(const GeoShape & shape) const
const GeoShapeShift & operator << (const HepTransform3D &) const
virtual const std::string & type () const
virtual ShapeType typeID () const

Static Public Methods

const std::string & getClassType ()
const ShapeType getClassTypeID ()

The classes `GeoShapeShift`, `GeoShapeUnion`, `GeoShapeSubtraction`, and `GeoShapeIntersection` are internal and should be considered for experts. We will not discuss them further.

We now present the interfaces to specific shapes. In general these shapes are by default constructed as symmetrically around the origin as possible.

GeoBox

Constructor:

GeoBox (double XHalfLength, double YHalfLength, double ZHalfLength)

Public Methods:

double getXHalfLength() const
double getYHalfLength() const
double getZHalfLength() const

The constructor fills the box with the x, y, and z half-lengths, and the accessors return those quantities.

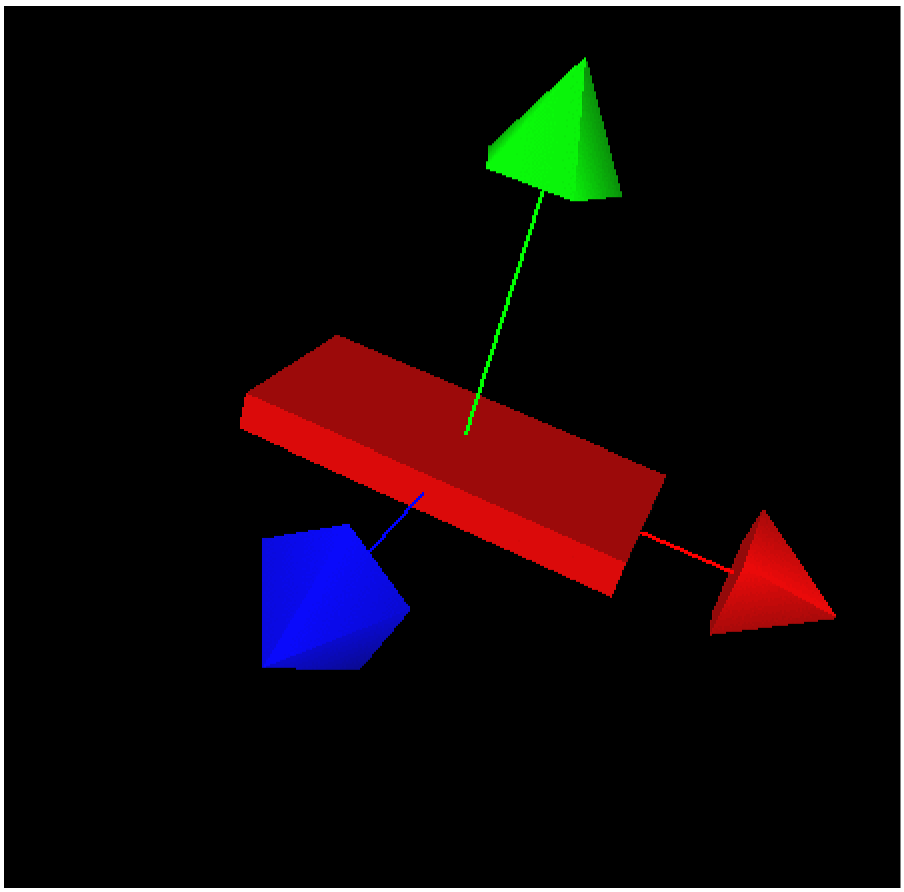

Figure 2: GeoBox object, representing a rectangular prism or "box".

GeoCons

Constructor:

GeoCons (double RMin1, double RMin2,
double RMax1, double RMax2,
double DZ, double SPhi, double DPhi)

Public Methods:

double getRMin1() const
double getRMin2() const
double getRMax1() const
double getRMax2() const
double getDZ() const
double getSPhi() const
double getDPhi() const

A `GeoCons` represents a cone section positioned with its axis along the z direction, and is specified by a starting value of ϕ, a total subtended angle in ϕ, a half-width in z, and minimum and maximum values of radius at both extremities in z. The constructor specifies the values of these constants, and the accessors return them.

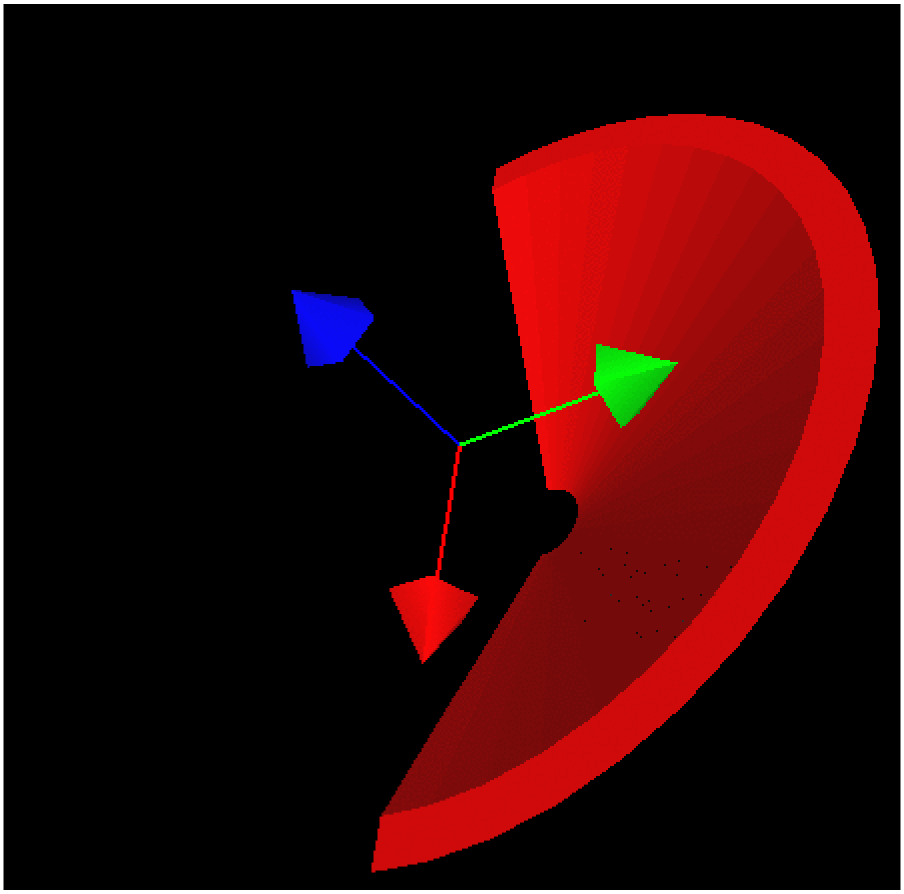

Figure 3: A GeoCons Object, representing a cone section.

GeoPara

Constructor:

**GeoPara (double XHalfLength, double YHalfLength, double ZHalfLength,
double Alpha, double Theta, double Phi)**

Public Methods:

double getXHalfLength() const
double getYHalfLength() const
double getZHalfLength() const
double getTheta() const
double getAlpha() const
double getPhi() const

The `GeoPara` class represents a parallelepiped. Faces at $\pm z$ are parallel to the x-y plane. One edge of each of these faces is parallel to the x-axis, while the other edge makes an angle α with respect to the y-axis. The remaining edge of the parallelepiped is oriented along a vector whose polar angle is θ and whose azimuthal angle is ϕ. Half-lengths in x, *y*, and *z* describe the projections of the sides of the parallelepiped project onto the coordinate axes. The constructor fills these data, while the accessors return them.

Note: Visualization of GeoPara is on the to-do list. If this is a problem for you, please contact the authors, and we will provide you with a quick implementation.

GeoPcon

Constructor:

GeoPcon (double SPhi, double DPhi)

Public Methods:

void addPlane (double ZPlane, double RMinPlane, double RMaxPlane)
double getSPhi() const
double getDPhi() const
unsigned int getNPlanes ()
bool isValid () const
const double & getZPlane (unsigned int i) const
const double & getRMinPlane (unsigned int i) const
const double & getRMaxPlane (unsigned int i) const

`GeoPcon` represents a polycone, which is a union of simple cone sections. The polycone subtends a fixed angle in ϕ (DPhi) beginning at ϕ_0 (SPhi), and is specified at *n* locations in *z*. At each *z* location, the inner and outer radius is given.

When the polycone is constructed, only ϕ_0 and ϕ are given; then, the information at each *z* location is given, one plane at a time, by using the `addPlane()` method. At least two planes must be given, otherwise the polycone is not valid and methods such as `volume()` will fail and throw an exception. The `isValid()` method can be used to determine whether the polycone has at least two planes.

A polycone (`GeoPcon`) with two planes is equivalent geometrically to a cone section (`GeoCons`).

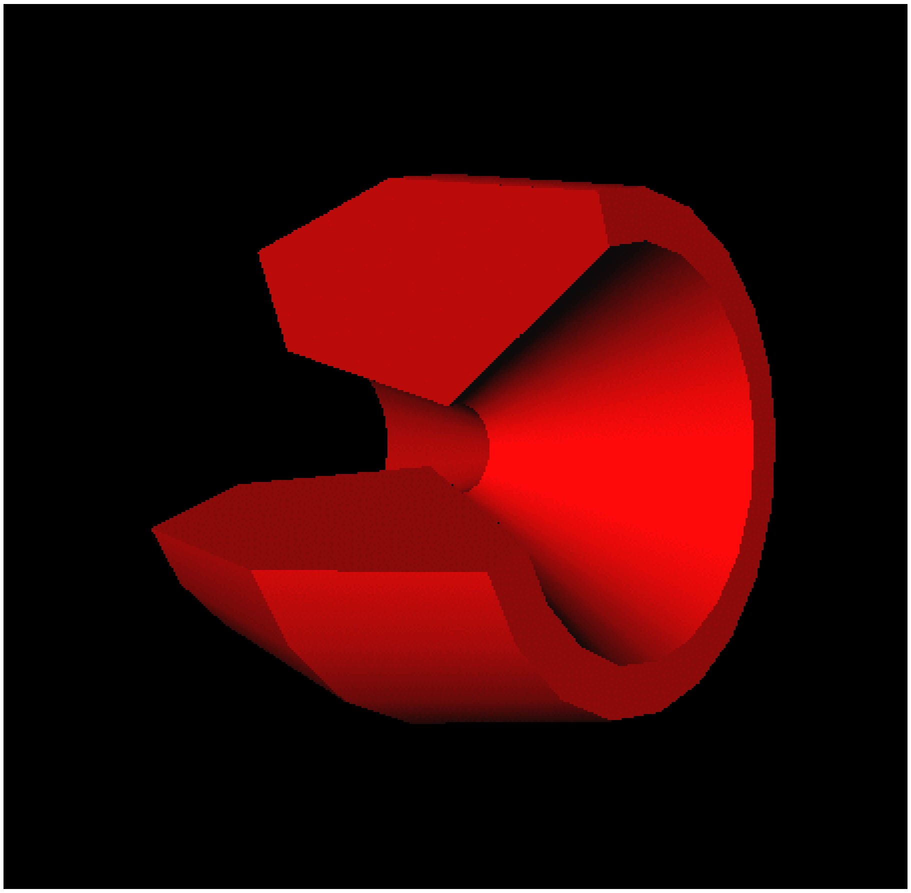

Figure 4: A GeoPCon object, representing a polycone.

GeoPgon

Constructor:

GeoPgon (double SPhi, double DPhi, unsigned int NSides)

Public Methods:

double getSPhi() const
double getDPhi() const
unsigned int NSides() const
unsigned int getNPlanes () const
const double & getZPlane (unsigned int i) const
const double & getRMinPlane (unsigned int i) const
const double & getRMaxPlane (unsigned int i) const
bool isValid () const
void addPlane (double ZPlane, double RMinPlane, double RMaxPlane)

`GeoPgon` is similar to a `GeoPcon` (polycone). Like a `GeoPcon` it subtends a fixed angle in ϕ (dPhi) beginning at ϕ_0 (sPhi), and is further specified by giving inner and outer radii at n locations in z. However the `GeoPgon` object has a polygonal cross section, and the solid angle ϕ is segmented into a fixed number of sides.

The constructor takes ϕ, ϕ_0, and the number of sides in the cross-section as arguments; then, the information at each z location is given, one plane at a time, by using the `addPlane()` method. At least two planes must be given, otherwise the polygon is not valid and methods such as `volume()` will fail and throw an exception. The `isValid()` method can be used to determine whether the polygon has at least two planes.

Note: Visualization of GeoPara is on the to-do list. If this is a problem for you, please contact the authors, and we will provide you with a quick implementation.

GeoTrap

Constructor:

GeoTrap (double ZHalfLength, double Theta, double Phi, double Dydzn, double Dxdyndzn, double Dxdypdzn, double Angleydzn, double Dydzp, double Dxdyndzp, double Dxdypdzp, double Angleydzp)

Public Methods:

double getZHalfLength() const
double getTheta() const
double getPhi() const
double getDydzn() const
double getDxdyndzn() const
double getDxdypdzn() const
double getAngleydzn() const
double getDydzp() const
double getDxdyndzp() const
double getDxdypdzp() const
double getAngleydzp() const

The `GeoTrap` class represents a very general kind of trapezoid. Two faces at $\pm\Delta z$ (`ZHalfLength`) are parallel to each other and to the x-y plane. The centers of the faces are offset by a vector whose polar and azimuthal angles respectively are θ and ϕ. At $-\Delta z$, two edges parallel to the x-axis are offset by $\pm\Delta y_n$ (`Dydzn`) from the face's center, and these two faces have half-lengths of $\Delta x_{\Delta yn\Delta zn}$ (`Dxdyndzn`) and $\Delta x_{\Delta yp\Delta zn}$ (`DxDypdzn`). The face at $+\Delta z$ are similar: two edges parallel to the x-axis are offset by $\pm\Delta y_p$ (`Dydzp`) from the face's center, and these two faces have half-lengths of $\Delta x_{\Delta yn\Delta zp}$ (`Dxdyndzp`) and $\Delta x_{\Delta yp\Delta zp}$ (`DxDypdzp`).

The two edges not parallel to the x-axis make an angle of α_n (Angleydzn) and α_p (Angleydzp) with respect to the y-axis on the bottom face (at $-\Delta z$) and the top face (at $+\Delta z$), respectively).

The constructor fills the `GeoTrap` with these values and the accessors return them.

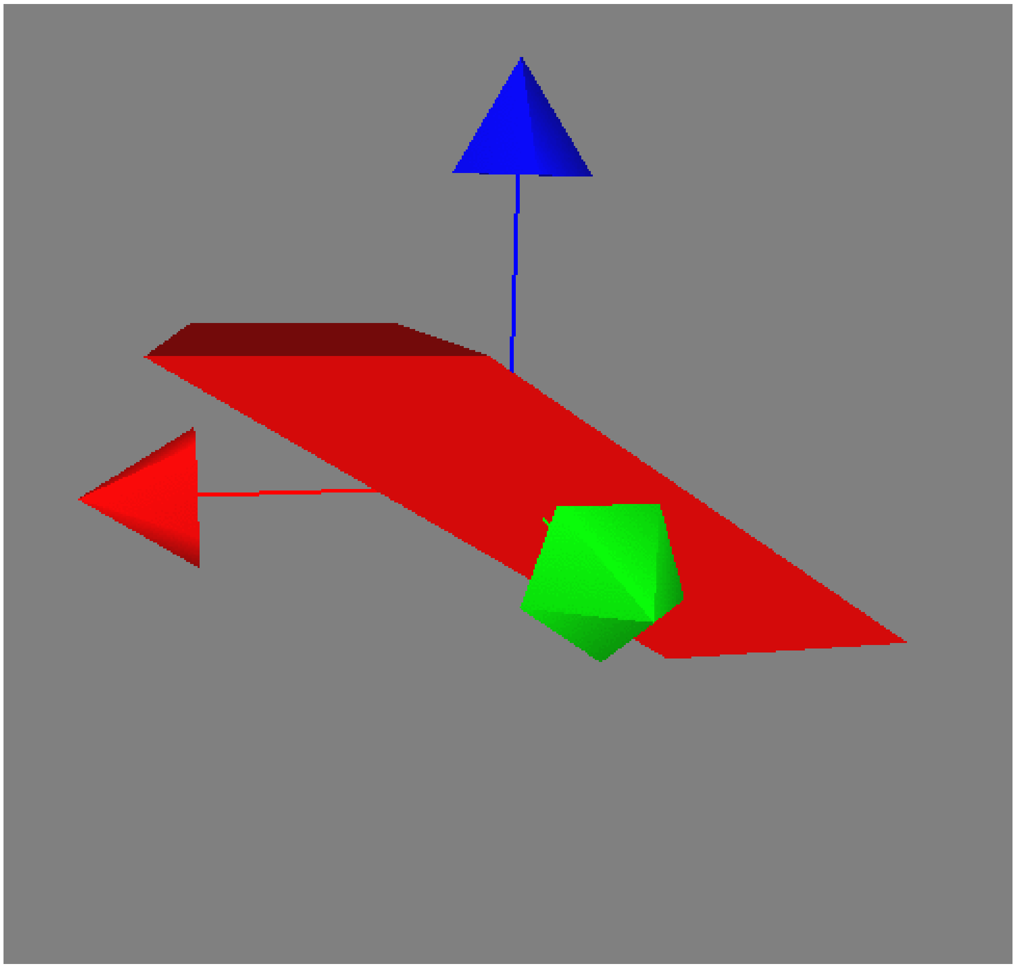

Figure 5: GeoTrap object, representing a very general kind of trapezoid.

GeoTube

Constructor:

GeoTube (double RMin, double RMax, double ZHalfLength)

Public Methods:

double getRMin() const
double getRMax() const
double getZHalfLength() const

The `GeoTube` class represents a tube, specified by an inner radius, an outer radius and a half-length in z. The constructor fills these quantities and the accessors return them.

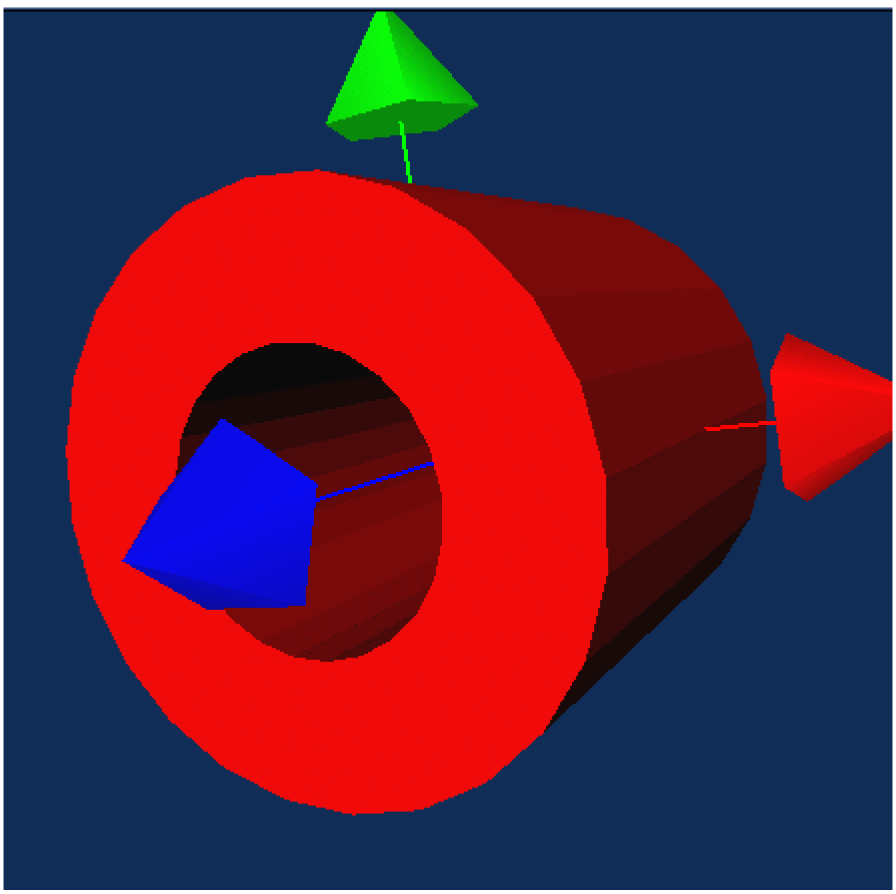

Figure 6: A GeoTube object, representing a tube.

GeoTubs

Constructor:

GeoTubs (double RMin, double RMax, double ZHalfLength, double SPhi, double DPhi)

Public Methods:

double getRMin() const
double getRMax() const
double getZHalfLength() const
double getSPhi() const
double getDPhi() const

A `GeoTubs` is a tube section; a tube that subtends some plane angle (less than 360º) in ϕ. The `GeoTubs` is constructed by providing the inner radius, outer radius, half length is z, as well as the starting ϕ and $\Delta\phi$. Member functions provide access to these quantities.

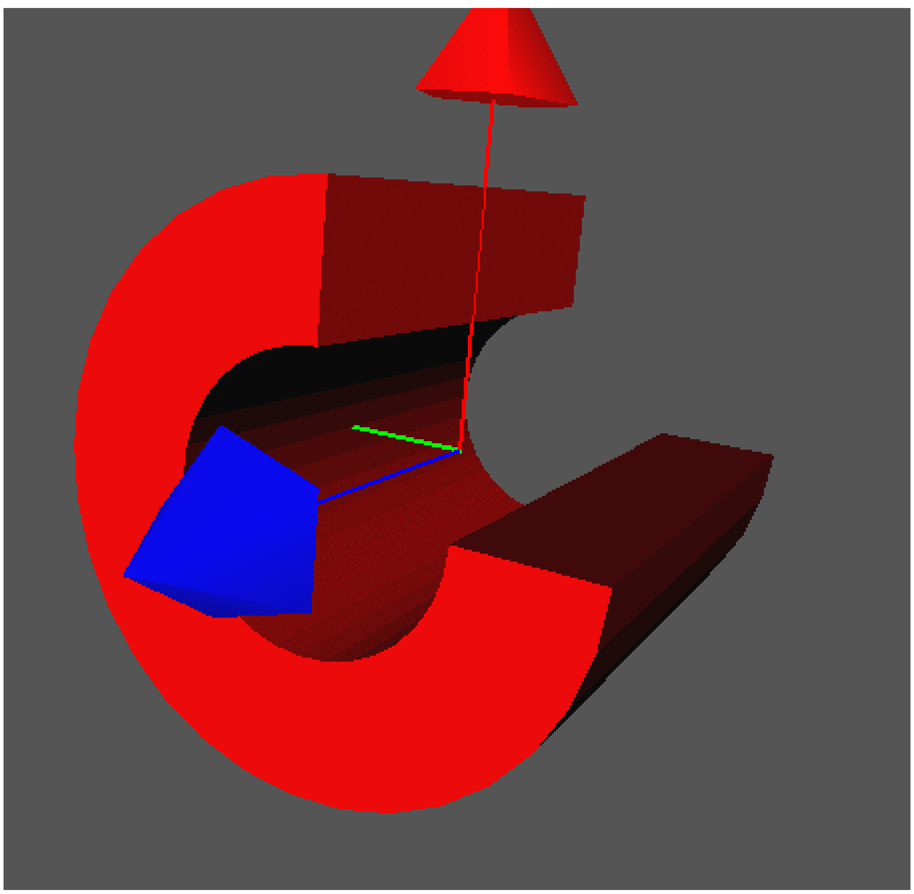

Figure 7: A GeoTubs object, representing a tube section.

GeoTrd

Constructor:

GeoTrd (double XHalfLength1, double XHalfLength2,
double YHalfLength1, double YHalfLength2,
double ZHalfLength)

Public Properties:

double getXHalfLength1() const
double getXHalfLength2() const
double getYHalfLength1() const
double getYHalfLength2() const
double getYHalfLength() const

A `GeoTrd` is a simple trapezoid. Two faces at $\pm\Delta z$ are parallel to each other and to the *x-y* plane, and each centered on the *z*-axis. At $-\Delta z$ ($+\Delta z$), the half-length of the edges parallel to the *x*-axis is `XHalfLength1`(`XHalfLength2`) and the half-length of the edges parallel to the *y*-axis is `YHalfLength1`(`YHalfLength2`). The constructor fills the object with these values and the accessors return them.

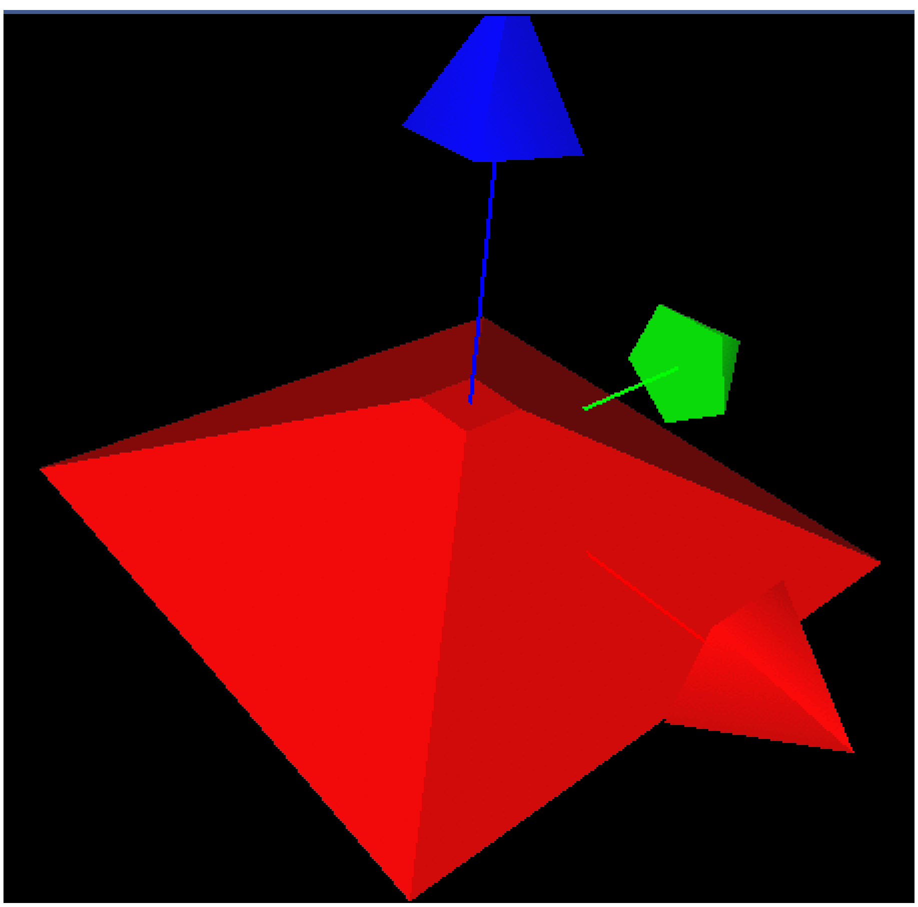

Figure 8: A GeoTrd object, representing a simple trapezoid.

GeoLogVol

Constructor:

GeoLogVol (const std::string & Name, const GeoShape * Shape, const GeoMaterial * Material)

Public Methods:

const std::string & getName () const
const GeoShape * getShape () const
const GeoMaterial * getMaterial () const

A `GeoLogVol` is an agglomeration of a name, a shape, and a material. These constituents are provided as arguments to the constructor, which increments the reference count of both the material and the shape. These reference counts are decremented when the `GeoLogVol` is destroyed.

The `GeoLogVol` provides const access to its constituents through the three access methods list above.

2.4 Physical Volumes

Physical volumes are objects that have a single logical volume and list of daughters. These daughters can have several types:

- Physical volume.
- Physical volume property nodes, such as name tags, or transformations.
- Parametrizations of physical volumes.

These types of daughters are referred to collectively as `GeoGraphNodes`. Physical volumes and graph nodes are the building blocks of the *geometry graph*. The geometry graph is a specification of a *physical volume tree*, which, when traversed at a later time, appears to consist of physical volumes within other physical volumes.

Unlike other geometry modelers, in GeoModel physical volumes live within other physical volumes and not within logical volumes. This simplifies tree traversal.

```
RCBase
|
|--GeoGraphNode-----------GeoNameTag
                    |-----GeoSerialDenominator
                    |-----GeoTransform-----------GeoAlignableTransform
                    |-----GeoVPhysVol------------GeoVFullPhysVol-------GeoFullPhysVol
                    |                |-----------GeoPhysVol
                    |
                    |-----GeoSerialTransformer
```

Figure 9: Class Tree for Graph Nodes. These objects are assembled into a tree by placing them within physical volumes.

In Figure 9, one sees that physical volumes have two main types, `GeoPhysVol` and `GeoFullPhysVol`. These in turn each have an interface class, `GeoVPhysVol` and `GeoVFullPhysVol`[5]. The user needs to be concerned only with the classes `GeoPhysVol` and `GeoFullPhysVol`. The former is generally used for nondescript pieces of detector geometry whose absolute position in space is not accessed often; while the latter is used typically for active detector components whose absolution position in space is used frequently within reconstruction, or digitization or hit creation.

The complete interfaces (including the inherited part) of `GeoPhysVol` is here:

[5] The reason for the interface classes is to provide a hook for virtual physical volumes which use recipes to generate children that do not actually exist permanently in memory. So far this has not been necessary to achieve parameterization, but we do not for now rule out the need for an interface class.

GeoPhysVol

Constructor

GeoPhysVol (const GeoLogVol * LogVol)

Public Methods

void add (GeoGraphNode * graphNode)

Public Methods from GeoVPhysVol

bool isShared () const
Query<unsigned int> indexOf (PVConstLink daughter) const
PVConstLink getParent () const
const GeoLogVol * getLogVol () const
unsigned int getNChildVols () const
PVConstLink getChildVol (unsigned int index) const
HepTransform3D getXToChildVol (unsigned int index) const
HepTransform3D getDefXToChildVol (unsigned int index) const
std::string getNameOfChildVol (unsigned int i) const
Query<unsigned int> getIdOfChildVol() const
void apply (GeoVolumeAction * action) const

Public Methods from GeoGraphNode

void exec (GeoNodeAction * action)

GeoPhysVol (const GeoLogVol * LogVol) Construct the physical volume from the logical volume. The reference count of the logical volume is incremented, and will be decremented again when the physical volume is destroyed.

void add (GeoGraphNode * graphNode) Add a graph node to the physical volume. The reference count of the graph node is incremented.

bool isShared () const Accessor to determine whether the physical volume is shared; i.e, used more than once in the geometry description.

Query<unsigned int> indexOf (PVConstLink daughter) const Accessor to determine the index of a daughter physical volume, in other words, in position within the list of daughters. The result only counts physical volumes as daughters, not their properties. The class `Query<T>` is described in the Appendix.

PVConstLink getParent () const Returns the parent physical volume. In case the parent is not unique (i.e., if the physical volume is shared), return NULL.

const GeoLogVol * getLogVol () const Returns the logical volume.

unsigned int getNChildVols () const Returns the number of child volumes. This includes only physical volumes and does not count property nodes. It does, however, include virtual physical volumes from a parametrization (See section 2.7).

PVConstLink getChildVol (unsigned int index) const Returns the specified child volume.
HepTransform3D getXToChildVol (unsigned int index) const Returns the transformation to the specified child volume. The transformation is relative to this object; it is *not* an absolute transformation to a global coordinate system.

HepTransform3D getDefXToChildVol (unsigned int index) const Returns the default transformation of the specified child volume, relative to this object.

std::string getNameOfChildVol (unsigned int i) const Returns the name of the child volume, relative to this object.

Query<unsigned int> getIdOfChildVol(unsigned int i) const Returns the identifier of the child volume. The class `Query<T>` is described in the appendix.

void apply (GeoVolumeAction * action) const Applies a volume action to the volume. This action normally recursively visits each daughter volume. See section 2.8 for a full discussion.

void exec (GeoNodeAction * action) Applies a node action to the volume. This action normally recursively visits each graph node in the geometry graph and includes handler functions even for property nodes.

GeoFullPhysVol

Constructor:

GeoFullPhysVol(const GeoLogVol *LogVol)

Public Methods:

void add (GeoGraphNode * graphNode)

Public Methods from GeoVFullPhysVol

const HepTransform3D & getAbsoluteTransform () const
void clearPositionInfo ()
const HepTransform3D & getDefAbsoluteTransform () const
const std::string & getAbsoluteName () const
Query<unsigned int> getId () const

Public Methods from GeoVPhysVol

bool isShared () const
Query<unsigned int> indexOf (PVConstLink daughter) const
PVConstLink getParent () const
const GeoLogVol * getLogVol () const
unsigned int getNChildVols () const
PVConstLink getChildVol (unsigned int index) const
HepTransform3D getXToChildVol (unsigned int index) const
HepTransform3D getDefXToChildVol (unsigned int index) const
std::string getNameOfChildVol (unsigned int i) const
Query<unsigned int> getIdOfChildVol(unsigned int i) const
void apply (GeoVolumeAction * action) const

Public Methods from GeoGraphNode

void exec (GeoNodeAction * action)

`GeoFullPhysVol` has a method for caching important information like absolute transformation of the piece with respect to the global coordinates, default transformation, and the name. Most of the interface comes from `GeoVPhysVol` and has already been described, the rest of the interface is described in more detail here:

GeoFullPhysVol (const GeoLogVol * LogVol) Construct the physical volume from the logical volume. The reference count of the logical volume is incremented, and will be decremented again when the physical volume is destroyed.

void add (GeoGraphNode * graphNode) Add a graph node to the physical volume. The reference count of the graph node is incremented.

const HepTransform3D & getAbsoluteTransform () const Returns the absolute transform with respect to global coordinate system.

const HepTransform3D & getDefAbsoluteTransform () const Retuns the default absolute transform with respect to the global coordinate system.

void clearPositionInfo () Clears the position information cache. Users do not normally ever need to do this. The cache is automatically cleared by the geometry system whenever a change in alignment is detected.

const std::string & getAbsoluteName () const Returns the absolute name of the volume. This is a "/" separated string of names whose substrings represent physical volumes along the path from the world physical volume down to this volume.

Query<unsigned int> getId () const Returns an integer identifier for labeling purposes. The class `Query<T>` is described in the appendix.

2.5 Transformations

Transformations (class `GeoTransform` and `GeoAlignableTransform`) are graph nodes that can be inserted into the geometry graph prior to the insertion of a physical volume. Transformations in the geometry graph are interpreted as affecting the position of the physical volume that follows. They do not affect the position of subsequent physical volumes.

No transformations are inserted before a physical volume, the physical volume remains at the center of its mother volume.

If one transformation is inserted, the physical volume is moved relative to the mother volume according to the transformation (which includes both a rotation and a translation).

If more than one transformation is inserted, then the composition of all transformations is applied to the physical volume. The *last* transformation to be added (the one closest to the physical volume) is applied *first*. Instances of `GeoTransform` and `GeoAlignableTransform` may be shared within a geometry graph.

The two classes both provide access to a default transformation and to an actual transformation. The major difference between them is that these transformations may differ for the `GeoAlignableTransform` but not for the `GeoTransform`. In addition, the misalignments may be set or cleared for `GeoAlignableTransform`.

The full interface for these classes is shown here:

GeoTransform

Constructors:

GeoTransform (const HepTransform3D & Transform)

Public Methods:

HepTransform3D getTransform () const
HepTransform3D getDefTransform () const
const HepTransform3D * getDelta () const

Public Methods from GeoGraphNode

void exec (GeoNodeAction * action)

GeoAlignableTransform

Constructors:

GeoAlignableTransform (const HepTransform3D & Transform)

Public Methods

void setDelta (const HepTransform3D & Delta)
void clearDelta ()

Public Methods from GeoTransform:

HepTransform3D getTransform () const
HepTransform3D getDefTransform () const
const HepTransform3D * getDelta () const

Public Methods from GeoGraphNode

void exec (GeoNodeAction * action)

2.6 Name Tags, Identifier Tags and Serial Denominators.

To minimize the use of memory, name information is not stored within physical volumes. In the case of a very large and complicated geometry, the need to denominate many millions of volumes is deemed largely unnecessary. However, certain pieces of the geometry are important and do deserve to be named. For these pieces, we have created two types of objects: name tags, and serial denominators.

Each physical volume is associated with a name, and with an absolution name. The name of the object is a simple string; while the absolute name is a "/" separated string that looks like a unix directory: /WORLD/INNERDET/SCT/BARRELSUPPORT, for example. When a physical volume is placed into a tree, by default it's name is "ANONYMOUS".

The default name can be modified by placing a name tag into the geometry graph immediately prior to a physical volume. A name tag is applied only to the physical volume that is inserted after the name tag into the geometry graph; it does not affect subsequent physical volumes. The class representing a name tag is `GeoNameTag`:

GeoNameTag

Constructors:

GeoNameTag (const std::string & BaseName)

Public Methods:

const std::string & getName() const

Public Methods from GeoGraphNode

void exec (GeoNodeAction * action)

The interface is simple, consisting only of a constructor taking a name-string, and an accessor which retrieves it.

Another way of modifying the default name is through an object that automatically generates name strings for all subsequent volumes added to a specific physical volume. This object is called a `GeoSerialDenominator`:

GeoSerialDenominator

Constructors:

GeoSerialDenominator (const std::string & BaseName)

Public Methods:

const std::string & getBaseName() const

Public Methods from GeoGraphNode

void exec (GeoNodeAction * action)

Its interface is also simple: one constructor, and one accessor which retrieves the base name. If the base name is "BASE", then the serial denominator generates names like "BASE0", "BASE1", "BASE2", etc. This continues until the last child, or until a `GeoNameTag` is encountered, or until a new `GeoSerialDenominator` is encountered.

The generation of names using `GeoNameTag` and `GeoSerialDenominator` applies also to the virtual volumes when parameterization is used. If a name tag is used with parameterization, then all of the virtual physical volumes of the parameterization are given the name tag. In case a serial denominator is used, each parameterized volume is given a name consisting of the base name plus an additional serial number.

In many cases it is useful to assign an identifier, or "serial number", to physical volumes. One use case is to enable a simulation engine such as GEANT to make a correspondence between a piece of geometry designated as sensitive, and the readout element corresponding to that piece. For such cases we provide a `GeoIdentifierTag`, similar to a `GeoNameTag`. It provides a way of labeling physical volumes with an unsigned int. `GeoIdentifierTag`s, however, apply only to the physical volume that immediately follows the tag in the geometry graph. Its interface follows:

GeoIdentifierTag

Constructors:

GeoIdentifierTag (unsigned int id)

Public Methods:

const std::string & getIdentifier() const

Public Methods from GeoGraphNode

void exec (GeoNodeAction * action)

2.7 Parameterization

A principle goal in the design of the geometry kernel has been to limit memory usage. A powerful way of doing this is to parameterize the volumes rather than to create them, say, inside of a single, double, or other multiple loop.

Parameterizations are mathematical recipes for creating volumes. There are three main ingredients to these recipes:

- `GENFUNCTIONS`, which are mathematical function-objects; they allow one to perform function arithmetic in the same way that one performs floating point arithmetic.
- `TRANSFUNCTIONS`, which, together with `GENFUNCTIONS` and `HepTransform3D`, allow one to expand and parametrize elements of the Euclidean group (i.e., rigid body transformations).
- `GeoSerialTransformer`, a kind of `GeoGraphNode` that allows a particular physical volume to be placed according to a `TRANSFUNCTION` expansion of a rigid body transformation.

An example will demonstrate how this is useful. (The example is taken from the ATLAS CVS repository, from the GeoModel description of the liquid argon barrel

```
#include "CLHEP/GenericFunctions/AbsFunction.hh"
#include "CLHEP/GenericFunctions/Variable.hh"
#include "GeoModelKernel/GeoXF.h"
#include "GeoModelKernel/GeoSerialTransformer.h"

using namespace Genfun;
using namespace GeoXF;

// Define some constants:

GeoPhysVol  *pV, *world;
int          N;
double       c1, c2, r, z;

// Construct a simple linear function of a variable i:

Variable      i;
GENFUNCTION   g =  c1+c2*i;

// Parameterize a transformation using this function:

TRANSFUNCTION xf =      Pow(HepRotateZ3D(1),g)*
                        HepTranslateX3D(r)*
                        HepTranslateZ3D(z);

// Use the parametrized transformation to place a volume N times:

GeoSerialTransformer *st=new GeoSerialTransformer(pV, &xf, N);
world->add(st);
```

Figure 10: Example code for parameterizing volumes using GeoSerialTransformer.

calorimeter.)

The file `AbsFunction.hh`, from CLHEP defines the interface to `GENFUNCTIONS` and must be included. In addition, if specific functions such as trig functions, higher transcendental functions or physics-specific functions are required, header files for these function-objects should be included from the same area.

The headers `XF.h` and `GeoSerialTransformer.h` are needed for the classes TRANSFUNCTION and `GeoSerialTransformer`, respectively.

GENFUNCTIONS and TRANSFUNCTIONS both live within namespaces, which we access with the `using` statements near the top of the example. After defining the variables used in this example, we construct a simple linear function of an input variable i:

```
Variable      i;
GENFUNCTION   g =  c1+c2*i;
```

This example is very simple, but shows already how we can use these classes to build symbolic expressions. A variety of functions lives already within CLHEP. The set is user-extensible, and the extension procedure is amply described within the GenericFunctions package documentation. Addition, subtraction, multiplication, division, composition, and direct product operations are all valid.

The next step, in which TRANSFUNCTIONS are constructed, parametrizes the rigid body transformation. The TRANSFUNCTION, `xf`, has a function call operator that can be used to evaluate a particular rigid body transformation as a function of an input argument, like this:

```
HepTransform3D tx = xf(j);
```

The expansion of the TRANSFUNCTION is as follows. Let X_i (i = 1, 2, … N) represent any transformation, either a rotation, a translation, or even some combination of these. The rotations may be about a coordinate axis, or even some other axis. Furthermore, let us denote by $f_i(x)$ (where i =1, 2… N) a function of a single variable. Then, the expansion of an arbitrary function is:

$$T(x) = X_1^{f1(x)} * X_2^{f2(x)} * X_3^{f3(x)\ldots\ldots} X_n^{fn(x)}$$

In this expression, $T(x)$ is the resulting transformation, which is now a function of the single input parameter, x. The expansion is both simple, and completely general. A single term in this expansion (for example $X_2^{f2(x)}$), will be referred to as an *exponentiated transformation*. It is implemented in terms of the class `Pow`, which has the following constructor:

```
Pow(const HepTransform3D &,  GENFUNCTION f);
```

Exponentiated transformations are simple TRANSFUNCTIONs, and can be composed to make other TRANSFUNCTIONS. The TRANSFUNCTION interface also allows one to compose fixed transformations with exponentiated transformations.

The interface to GENFUNCTION and TRANSFUNCTION provide all the necessary operations on these types of object. The interfaces to these types of objects are not simple to read, so we will not attempt to explain them in this document. Instead, one should assume that all well-defined mathematical properties that apply to functions are properties of GENFUNCTIONS, and all mathematical properties that apply to parameterizations of elements of the Euclidean group are properties of TRANSFUNCTIONS.

Once one has a a TRANSFUNCTION in hand, it can be used together with a GeoSerialTransformer object to repeatedly place a physical volume. To do this, use the following constructor:

```
GeoSerialTransformer(const GeoPhysVol *pVol,
                     TRANSFUNCTION xf,
                     unsigned int N);
```

In this constructor, `pVol` is the volume to be repeatedly placed, `xf` is the TRANSFUNCTION that specifies how to place it, and N is the desired number of copies.

The `GeoSerialTransformer` can then be added to the geometry graph. During any subsequent volume traversal, the geometry graph will appear to contain multiple physical volumes at different locations. However, only the memory of a single physical volume and a TRANSFUNCTION is actually allocated.

During node traversal (using `GeoNodeActions`) one can recover, from the geometry graph, the actual recipe for generating volumes. This is sometimes useful; for example, in case one wishes to create a GEANT4 parameterization from a GeoModel parameterization.

[One further word about parameterizations is in order: parameterizations, as they are usually understood, allows for the shape or composition of an object to vary as a function of copy number. This is presently not a part of the geometry kernel.

However, we intend to include this in subsequent releases. The basic design strategy is to start with a concrete shape class, such as `GeoBox`, and to use this a basis of a new class for parameterizing boxes. In the new class—call it `GeoBoxParameterization`—we replace all of the floating point member data with GENFUNCTION member data, and all of the floating point constructor arguments with GENFUNCTION constructor arguments. In this way we create a very flexible recipe for generating a box.

The same technique can be used to vary an objects composition as a function of copy number.]

GeoSerialTransformer

Constructors:

GeoSerialTransformer (const GeoVPhysVol * volume, const GeoXF::Function * func, unsigned int copies)

Public Methods:

HepTransform3D getTransform (int i) const
unsigned int getNCopies() const
PVConstLink getVolume() const;

Public Methods from GeoGraphNode

void exec (GeoNodeAction * action)

This interface provides a constructor for the serial transformer and access to all of its constituents.

2.8 Actions

The principle way of accessing the physical volume tree is through *actions* which facilitate a recursive tree traversal and accumulate certain volatile information, such as a volume's absolute position, during the traversal. An action is applied to a node in the tree, which applies it in turn to its children. One type of action, `GeoNodeAction`, is passed to all children. Another type, `GeoVolumeAction`, is only passed only to daughter physical volumes (this includes "virtual" physical volumes that come from parameterizations). Casual users should consider subclassing `GeoVolumeAction` in order to recurse through all of the physical volumes in the geometry graph. "Power users" will occasionally need to access the geometry graph directly, in order to visit nodes such as `GeoSerialTransformers`. Such users should consider subclassing `GeoNodeAction`. This section describes the two types of existing actions in more detail.

2.8.1 Volume Actions and associated classes

`GeoVolumeAction` is a base class specifically designed for user subclassing, and all users are encouraged to use it for data access. Its interface is here:

GeoVolumeAction

Enumerated Types:

enum GeoVolumeAction::Type {TOP_DOWN, BOTTOM_UP}

Constructor:

GeoVolumeAction (Type type = TOP_DOWN)

Public Methods:

void handleVPhysVol (const GeoVPhysVol *)
void terminate ()
const GeoTraversalState * getState () const

In order to subclass this, follow this checklist:

1) Write a class inheriting from `GeoVolumeAction`. You should decide whether your class should walk through the volumes from the top down, which is the default, or from the bottom up. If you want a bottom up tree traversal you need to initialize the base class, accordingly, in the constructor for your new class.
2) Override the `handleVPhysVol` method in order to do something with each volume you encounter during geometry graph traversal. You may, of course, add member data or additional methods in order to carry out the action, or to access results.
3) If you wish for the action to hit every node in the tree, you don't need to do anything special. If you wish it to terminate early, call the `terminate()` method from with your `handleVPhysVol()` . Geometry graph traversal will immediately terminate.
4) The action keeps track of its traversal state, accumulating such information as absolute transformation to the current node, and the path to the current node. The state can be retrieved from the `getState()` method. This may be called from with your `handleVPhysVol()` method in order to, say, find out where you are in global coordinates.

A GeoVolumeAction upon a tree of physical volumes is initiated when the following line of code is invoked:

```
GeoPhysVol                  *vol;
MyGeoVolumeAction            action;
vol->apply(&action);
```

Your handleVPhysVol routine can obtain information about the current node from two sources: first, from the physical volume itself, which is passed as an argument to the routine, and secondly from the state (class GeoTraversalState) which is available from the getState() method. We examine that next.

GeoTraversalState

Const Public Methods:

const HepTransform3D & getTransform () const
const std::string & getName () const
Query<unsigned int> getId() const
const HepTransform3D & getDefTransform () const
const std::string & getAbsoluteName () const
const HepTransform3D & getDefAbsoluteTransform () const
const HepTransform3D & getAbsoluteTransform () const
const GeoNodePath * getPath () const

The interface is simple and self-explanatory; only the getPath() method and the information it returns needs further explanation. The path (class GeoNodePath) is an ordered stack of nodes that shows how the current node was reached. It's interface is shown here:

GeoNodePath

Public Methods:

unsigned int getLength ()
const GeoVPhysVol * getItem(unsigned int i)
const GeoVPhysVol * getHead ()
const GeoVPhysVol * getTail ()

The head node, or node from which the action was initiated, and tail node, or last node in the path, are available. The total length of the path can be retrieved, as well as arbitrary item along the path.

TemplateVolAction

TemplateVolAction is a class which has been added only as an illustration for how to write volume actions. The header file is shown in Figure 11, while the source file is shown in Figure 12.

```
#ifndef TemplateVolAction_h
#define TemplateVolAction_h 1

class TemplateVolAction:public GeoVolumeAction
{

public:

  // Constructor
  TemplateVolAction ();

  // Destructor
  ~TemplateVolAction ();

  // Action routine
  virtual void handleVPhysVol (const GeoVPhysVol *);

};
#endif
```

Figure 11: TemplateVolAction header file. This is an example of how to write actions that traverse the physical volume tree.

```
#include "GeoModelKernel/TemplateVolAction.h"

TemplateVolAction::TemplateVolAction ()
:GeoVolumeAction (GeoVolumeAction::TOP_DOWN)
{
}

TemplateVolAction::~TemplateVolAction ()
{
}

void TemplateVolAction::handleVPhysVol (const GeoVPhysVol
*)
{
  // Your procedure here.  This one does nothing…
}
```

Figure 12: TemplateVolAction source file. This is an example of how to write actions that traverse the physical volume tree.

2.8.2 Node Actions

`GeoNodeActions` do more than traverse the tree of physical volumes. These actions stop and execute on every graph node in the geometry graph, not the physical volumes represented within or generated by the geometry graph. Three kinds of node actions are used internally by the kernel:

- `GeoCountVolAction`,
- `GeoAccessVolumeAction`,
- `GeoClearAbsPosAction`.

`GeoCountVolAction` counts the number of physical volumes below some node, down do a depth. GeoAccessVolumeAction is for retrieving a particular volume. GeoClearAbsPosAction can be used to invalidate a cache of absolute position information below some node. The first two have potential usefulness outside of the geometry kernel itself, and will be discussed below.

The following methods are available on all `GeoNodeActions`, and control the depth limit:

- `void setDepthLimit(int limit);`
- `Query<unsigned int> getDepthLimit() const;`
- `void clearDepthLimit();`

Specific `GeoNodeActions` may set specific defaults for their depth limits. See the documentation on these actions.

GeoCountVolAction

Constructor:

GeoCountVolAction ()

Public Methods:

const unsigned int getCount() const

When this action is executed upon a node it counts daughter volumes. The count includes only physical volumes. It contains virtual, or parameterized volumes as well as actual volumes.

The depth limit for this action by default is 1: only the volumes directly beneath the top volume are counted. The final count does not include the top volume, only the children. The depth limit can be reset or cleared using the methods `clearDepthLimit()` or `setDepthLimit()` from the base class. Here is a typical use case:

```
GeoPhysVol *world;
GeoCountVolAction cv;
cv->setDepthLimit(2);
world->exec(&cv);
int count = cv.getCount ();
```

GeoAccessVolumeAction

Constructor:

GeoAccessVolumeAction (unsigned int Index)

Public Methods:

PVConstLink getVolume () const
const HepTransform3D & getTransform () const
const HepTransform3D & getDefTransform () const
const std::string & getName () const
Query<unsigned int> getId() const

GeoAccessVolumeAction is used to retrieve physical volume and some of its properties from within the geometry tree.

The constructor needs to provide the index of the child volume which is sought. The public methods can be used once the action has executed and return a link to the daughter volume, the transform to the daughter, the default transform to the daughter, and the name of the daughter. A typical use case is shown here:

```
int index
GeoFullPhysVol *vol;
GeoAccessVolumeAction av (index);
vol->exec (&av);
std::string name = av->getName();
```

Besides the name, the transformations (default and misaligned), can be retrieved along with the volume. The action executes to a depth of 1; i.e. it is used only for accessing direct descendents.

An simpler alternative to accessing daughter volumes in this way is to use the methods in GeoVPhysVol and subclasses to retrieve information about their daughters. This is simpler, and uses the action internally. Using the action directly, however, can be faster if accessing different kinds of information at the same time since the volume location can be performed just once.

2.8.3 For Power Users: How to Make Your Own GeoNodeAction

The basic structure of the GeoModel is: a geometry graph which emulates a physical volume tree. The graph is constructed by adding a combination of physical volumes, transformations, name tags, serial denominators and serial transformers into physical volumes. The information is usually retrieved by scanning the physical volume tree as opposed to the geometry graph. The class GeoVolumeAction exists in order to be subclassed by users for this purpose.

However in certain more rare cases users will need to navigate the geometry graph directly. For example, when declaring the geometry to the simulation, one may wish to translate GeoModel parametrizations from `GeoSerialTransformer` nodes into `G4Parametrizations`. In that case one is interested in accessing the parametrization directly and not the virtual volumes that they generate. These “power users” can access all nodes in the geometry tree by subclassing `GeoNodeAction`. In this subsection we examine how to do this.

GeoNodeAction has the following interface:

GeoNodeAction

Constructor:

GeoNodeAction()

Public Methods:

GeoNodePath * getPath () const
Query<unsigned int> getDepthLimit () const
void terminate ()
bool shouldTerminate () const
void setDepthLimit (unsigned int limit)
void clearDepthLimit ()

Virtual Public Methods to be overridden in subclasses:

void handleNode (const GeoGraphNode *)
void handleTransform (const GeoTransform *)
void handlePhysVol (const GeoPhysVol *)
void handleFullPhysVol (const GeoFullPhysVol *)
void handleNameTag (const GeoNameTag *)
void handleIdentifierTag(const GeoIdentifierTag *)
void handleSerialDenominator (const GeoSerialDenominator *)
void handleSerialTransformer (const GeoSerialTransformer *)

When subclassing the `GeoNodeAction`, the principle task is to write a handle method for each type of `GeoGraphNode` object that you wish to visit. These methods are shown in the above table under the rubric "Virtual Public Methods to be overridden in subclasses". They are called in sequence each time a specific kind of graph node is encountered, from top to bottom.

In addition, the action can specify its own depth by calling the methods `setDepthLimit()` and `clearDepthLimit()`. In addition the action can be terminated by any time by calling the `terminate()` method; usually this should be done within one of the handler methods. The path (class `GeoNodePath`, discussed in section 2.8.1) can also be accessed from anywhere within the class, most notably within the handler methods.

2.9 Base classes for subsystem description

The geometry kernel contains three base classes for subsystem description. These provide require a minimum amount of functionality from subclasses—just enough to function within a reasonable framework. The classes are: GeoVDetectorElement, and abstract base class for a separately-alignable piece of a detector subsystem, GeoVDetectorManager, an abstract base class for an algorithm which builds the geometry, including the detector elements and other pieces of nondescript support material, and and GeoVDetectorManager, which is stored in the transient detector store and provides access to the geometry, both material geometry and readout geometry. Both classes should be extended by subsystems people to provide a bona fide readout geometry interface. The interfaces to these classes are shown here:

GeoVDetectorElement

Constructor:

GeoVDetectorElement (const GeoVFullPhysVol * fullPhysVol)

Public Methods:

const GeoVFullPhysVol * getMaterialGeom () const

Virtual Public Methods to be overridden by subclasses:

Identifier identify() const;

GeoVDetectorElement (const GeoVFullPhysVol * fullPhysVol) The constructor for a GeoVDetectorElement requires a full physical volume, which should be unique and which should live under a unique branch. This is how all detector elements know where they are.

const GeoVFullPhysVol * getMaterialGeom () const Provides access to the full physical volume. This is mostly for subclasses to use, when they compute derived position information.

Identifier identify() const Provides an identifier for the volume. [Note that the coupling to the actual identification scheme is loose because this method is pure virtual and the Identifier class is forward-declared].

GeoVDetectorFactory

Constructor:

GeoVDetectorFactor()

Virtual Public Methods to be overridden by subclasses:

void create (GeoPhysVol * world)
const GeoVDetectorManager *getDetectorManager() const;

`GeoVDetectorFactories` create geometry and map it into readout geometry. For the simulation, access to the raw geometry is required. The interface to GeoVDetectorFactory contains two pure virtual functions which must be implemented in the subclass.

void create (GeoPhysVol * world) Creates the material and readout geometry, and attaches the raw geometry to the physical volume tree, under the world volume, which is passed in from the calling routine.

const GeoVDetectorManager *getDetectorManager() const Returns a detector manager, which contains all of the constructed geometry. It is permissible to return a pointer to a subclass of `GeoVDetectorManager` (the so-called covariant return type mechanism). This is proper C++ and the correct way of avoiding dynamic casting.

GeoVDetectorManager

Constructor:

GeoVDetectorManager()

Virtual Public Methods to be overridden by subclasses:

unsigned int getNumTreeTops () const
PVConstLink getTreeTop (unsigned int i) const

`GeoVDetectorManagers` are hold the results of geometry construction and are stored in the transient detector store. For the simulation, access to the raw geometry is required. Experience has shown that subsystems cannot be cleanly implemented in terms of a single, top-level volume; instead several volumes must exist at the top level, for topological reasons, hence the iterative access to top level volumes, or tree-tops. The interface to GeoVDetectorManager contains several pure virtual functions which must be implemented in the subclass.

unsigned int getNumTreeTops () const Returns the number of top-level physical volumes in the raw geometry for the subsystem.

PVConstLink getTreeTop (unsigned int i) const Returns the ith top-level physical volume for the subsystem.

3 Appendix A: The Query<T> template class.

The template class Query<T> is designed as the return type of a query that can fail. One place we use it within this library is to return the index of a daughter volume, in other words, its position within a child list. If the volume is not found within the daughter list, the query fails. The failure could be handled in several ways, one of which would be to return a value of -999. If the user were to blithely use this value without checking it first, the program could likely crash immediately or misbehave later.

The class Query<unsigned int> fixes ths problem. Because it has a constructor taking unsigned int as a single argument and a cast operator, it can freely convert itself to and from an unsigned int. So you can write:

```
GeoPhysVol *parent, *child:
unsigned int c = parent->getIndex(child);
```

If the function succeeds, it returns Query<unsigned int> and a conversion takes place. But, if it fails, the conversion itself throws an exception, in this case std::range_error.

One does not need to handle the exception in order to proceed past the failed query, a better way is to check the return value, and this can be done by recoding the example as follows:

```
GeoPhysVol *parent, *child:
Query<unsigned int> c = parent->getIndex(child);
if (c.isValid()) {
     // Now use c like an ordinary unsigned integer:
     unsigned int d = c+1;
}
```

Now, this kind of checking is safer, but can always be skipped if the operation is guaranteed to succeed, for example, if the programmer knows the child is in the daughter list (because she put it there). The class Query<T> is based on the class Fallible<T> from reference[v].

[i] http://atlassw1.phy.bnl.gov/lxr/source/atlas/DetectorDescription/GeoModel/GeoModelExamples/

[ii] CLHEP, A Class Library for High Energy Physics. http://wwwinfo.cern.ch/asd/lhc++/clhep/ See the user's guide and specifically the section on CLHEP units.

[iii] Requests should be sent to boudreau@pitt.edu.

[iv] K. Hagiwara et al., Physical Review D**66**, 100001 (2002)

[v] JJ.Barton and LR.Nackmann, Scientific and Engineering C++, 1994, Addison Wesley