RUSSIAN ACADEMY OF SCIENCES
PETERSBURG NUCLEAR PHYSICS INSTITUTE

Preprint 2373

I. B. Smirnov

SpaceMetricLib, Geometrical Class Library for Detector Modeling in HEP

Gatchina-2000

Abstract

A new geometrical class library has been designed with the use of object-oriented programming. Geometrical objects and concepts have been analyzed and adapted for application in computer modeling the detectors of elementary particles in the C++ programming language. A general-purpose collection of geometrical figures is used for definition of geometrical volumes which allow to describe an experimental setup and to model particle penetration through it. Possible applications are Monte-Carlo simulations and the treatment of experimental data.

1 Introduction

Evolution of large software projects results in migration from programming consisting in algorithmic decomposition, partition of a complicated program into simple procedures, to programming based on structuring and classification of data. This trend is noticeable even at programming on the languages that do not support structures and classes as aggregate types of data. In any programming language, the larger program is written, the more work should be invested in design of allocation of data and methods for access to them. (for example, see the manual of GEANT3[1]). Modern object-oriented languages offer new technology in which the information, represented by data and related functions, is grouped in accordance with its meaning and constitutes aggregates called classes. This technology is the most efficient if the classes are interpreted as not merely the programming notation, but the representation of the abstractions of a subject area. However, this "natural" classification borrowed directly from the subject area is often found to be vague, insufficient, or inappropriate for computer modeling. To adopt it for modeling we need to perform an additional analysis of the subject, to modify the "natural" abstractions, and to identify the additional "artificial" elements which are not used by the experts of the problem domain. This process is called the object-oriented analysis and design and studied in detail in [2].

To prepare an experiment in high energy physics it is often necessary to create the computer model of the detector. The Monte-Carlo program should transport the particles through the detector and simulate its response. To treat the experimental data we also need an assistance from the computer model. In particular, at this stage we often solve the similar problems related to particle transportation or simple propagation of the particle trajectory through the detector. The basis of this model should be a description of the geometrical configuration of the detector, which should be supplemented by the description of materials, measured signals, and other necessary attributes.

Thus, we need a class library providing keeping and arbitrary manipulation with geometrical information. This information is expected to be a description of the detector configuration. The list of main operations includes calculations of the coordinates of any particular point of the detector in any reference system, and the support of the calculation of particle penetration through the setup.

The decision to develop the new geometrical library was motivated by the absence “on the market” of free distributed, portable, and well documented libraries. The design of this package started in 1993 for application in R&D studies. Lately, this library was also applied for the data treatment[3]. Nowadays, the market is full of the geometrical packages. However, most of them are licensed, expensive, platform-dependent, inaccessible in source codes, and expected to have a short life-cycle and a narrow area of application, mainly computer graphics or CAD on the “Wintel” platforms. A modern GEANT4 [4] project of CERN has depended on commercial packages as well, and it seems to be designed only for Monte-Carlo.

The program is coded in C++. We will give some examples of class definitions and function headers. Therefore, the reader of this paper is expected to understand the syntax of C++. The reader should also be familiar with the notations of G. Booch[2]. The class diagrams presented in this paper differ from these conventions only in notation of class itself, for which we use more compact rectangular box with rounded corners. The class methods are denoted by function headers written in C++ syntax as well.

We hope that the analysis presented in this paper may be useful for developing similar programs or at eventual application of ours.

2 Geometrical objects

All objects and figures familiar from geometry: the vector, the point, the straight (or not straight) line, the plane, the circle, the polygon, the cube, and so on, can eventually be utilized in computer modeling. At the numerical calculations every geometrical object will contain the numerical information that determines the object's shape, position, and orientation, or some of these in a certain coordinate system at certain time.

In particular, the point is described solely by its position. On the contrary, the vector, as the representation of the direction and distance between two positions, is not positioned itself. In fact, the point and the vector represent two basic types of information, the position and the direction, which together with scalar sizes allow to describe more complicated figures.

The dependency of the numerical information on the coordinate system is convenient to determine as the transformation of the object at passing from the current system to another one. The transformation occurs also at the shift or rotation of the object with respect to the current system. The transformations are the intrinsic feature of any geometrical figure. They allow to distinguish the geometrical objects from the others, the scalars in respect of geometry. This common feature exhibited by different classes manifests the inheritance of a common super-class. This class is called here `absref`, an abridgment of <u>abs</u>tract <u>ref</u>erence, fig. 1. All geometrical figures are sub-classes of the class `absref`. This class has virtual functions `up()` , `down()`(conversion to another coordinate system), `turn()`, and `shift()`. The functions `up()` and `down()` are called with the pointer to an "upper" coordinate system. They convert the object to or from the "upper" system respectively (in detail this will be explained later). The function `turn()` receives the vector around which it should rotate the object counter-clockwise, and the angle of rotation in radians. The rotation is performed around the center of the object's current coordinate system. The type `vfloat` mentioned in the list of arguments in fig. 1 is our alias of `double float` or `float` types and is an abridgment of "vector's float". The function `shift()` performs the translation by the distance and in the direction given by its argument.

The simplest objects derived immediately from the `absref` are the point (class `point`) and the vector (class `vec`). The vector is represented by three Cartesian components. The point is also

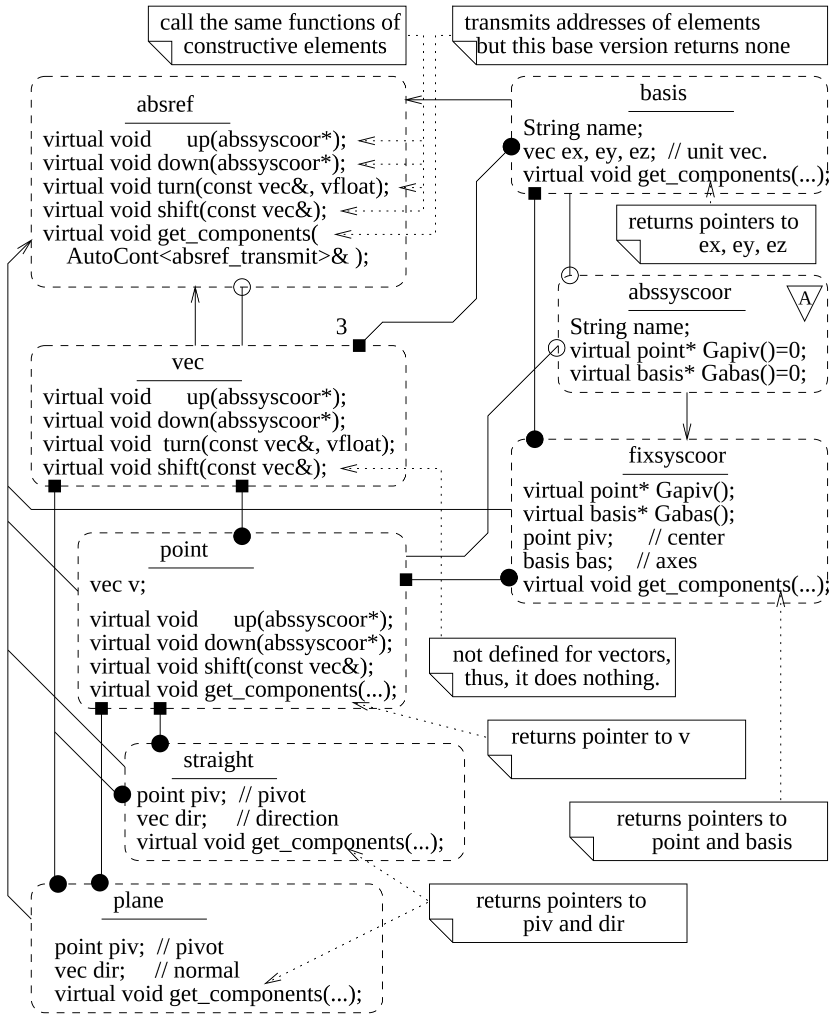

Figure 1: The class diagram of geometrical primitives.

determined by three coordinates, but has different features. Therefore, none of these types is the sub-class of the other. The point is defined as a class with element of class `vec`.

The simple figures are utilized for description of more complicated shapes. At the transformation of the composite object the constructive parts should typically undergo the similar transformation. This observation allows to automate the transformations. The transforming functions of the class `absref` can perform the transformation by calling itself for each constructive element. They obtain the addresses of the elements via the call of another virtual function called here `get_components()`. This function is expected to be overloaded by the derived classes. We will not describe here the argument of this function, but just say that it allows to transmit the necessary addresses of the components. The base version `absref::get_components()` returns nothing. Therefore, if this function is not overloaded, and neither are the transforming ones, the sub-class will ignore the transformations. If all transforming functions are overloaded, as it is in the class `vec`, the function `get_components()` does not affect the object. The class `point` is the simplest composite class consisting of the only constructive element, the vector. The function `point::get_components()` transmits the address of this vector and its rotation is therefore performed by `vec::turn()` called by `absref::turn()`. Nevertheless, the other transformations of `point` can not be done by the automatic way. To preserve the integrity of composite figures the positions should vary by different way if compared with the directions. Therefore the other three transforming functions are reloaded in the `point`.

A class `basis` consists of three unit vectors perpendicular to each other, fig. 1. The vectors constitute the ordinary right-hand system, therefore $\vec{e}_z = \vec{e}_x \times \vec{e}_y$. The vectors are meant numerically described in a parent basis. The latter basis is not referred in the current one. The function `basis::get_components()` returns the addresses of all three vectors and this provides the automatic transformations.

The basis and the point together determine the coordinate system. They are assumed to be numerically defined in a parent coordinate system. This latter system is not referred. These features are gathered in the abstract class `abssyscoor`. This class contains the name of the system and pure functions `Gapiv()` and `Gabas()`, which return the addresses of the central point and the basis respectively. The derived class can either contain them or just know where they are. The class `fixsyscoor` contains the point and the basis as its elements. The latter class is also derived from the class `absref` and therefore is a geometrical object modified at the transformations.

The transforming functions of `absref`, `up()` and `down()`, have one argument, `abssyscoor*`. For both functions it denotes the "upper" system numerically measured in the "lower" system. The object itself is expected to be described in the lower system before calling `up()` and in the upper one before calling `down()`.

In some other libraries, see for example [5], one can meet the definitions of the transformations through specific matrices defined in "transforming" classes. Although the latter approach appears to be nice in theory, it is less convenient for object-oriented computer modeling. The transformation is not an object, it is a procedure. Nevertheless, it is quite clear that the both approaches are equivalent in mathematical procedures and results.

There are two objects defined by the point and the unit vector: the straight line (class `straight`) and the plane (class `plane`). The circle has an additional parameter determining its radius. Although this way of describing is the most convenient, the redundancy of the parameters entails difficulties at the recognition of identical objects, as they could be determined via the different components. All the functions from this library are clever enough to recognize them.

A range of more complicated objects is defined via points, vectors, and straight lines.

Sometimes it is necessary not only to recognize identity, but to investigate or to confirm other relationships between the objects occurring because of their specific shape, position, or orientation.

It may be allocation of one object in another, parallelism, crossing, and so on. Functioning of some algorithms depends on whether the computer can correctly recognize these relations in the objects. However this ability can be hampered not only by the redundancy of the parameters, but also by the numerical inaccuracy inherent to computer calculations. Since the floating-point numbers in the computer are rounded and restricted by the order of magnitude, one or a few last bits of mantissa of any floating-point number can be inaccurate. Therefore every function which recognizes anything that can be compromised by the numerical errors is supplied by an additional argument determining the expected or required precision. The meaning of the precision is different and is defined for each function separately. Typically it means the largest distance or the angle between something when the answer is still affirmative.

3 Volumes

An experimental installation, its details, as well as any other material object, solid, liquid, gas, or vacuum occupy some volumes. The volume can have any form, it can be torn to stand-alone pieces or be infinite in some directions. Whatever shape the volume has, it always displays one feature: it is totally isolated from the external world. If this very world exists (this is not guaranteed, since the volume can occupy all space), there exists a surface that isolates the volume from it and represents the volume boundary. One can not bypass the surface while traveling from an inner point of the isolated region to the outside space. This entails two consequences important for computer modeling. First of all, this means that a point can be located either inside the volume, or outside it. Second, if a trajectory along which the point is moving exits or enters the volume, we can measure the distance to the intersection with its boundary. Measuring this distance is important, because the motion at computer modeling should be impeded at the volume boundaries for necessary arrangements.

The volume surface divides the space to three logically different parts: the inner region (the volume), the outside space, and the surface itself. The surface itself does not represent any material but indicates the border between two. The particle moving through the installation and crossing the surface will interact with the material of the solid for which the particle is headed. This leads to the following convention regarding the moving point: starting from the moment when the point meets the surface the point is considered to be located already in the sub-space that it enters. This convention provides automatic re-allocation of the point to the next volume without an external control or prompting.

To suppress the side effects of the numerical inaccuracy inherent to computer calculations, the fact of location on the surface should be confirmed with some allowance, the allowed distance from the point to the surface. Since the solids can have different scale in the same program, the allowance may also be different and should characterize each volume separately. This precision is convenient to determine as an attribute of each volume. It may be calculated at the volume initialization as a function of its geometrical dimensions.

When the point meets the surface and is re-allocated to the next volume, it can be physically positioned still in front of the surface at the little allowed distance from it. Propagating the trajectory ahead we should encounter first on this surface and enter to the next volume again. The second entering to the same volume can confuse the program. Therefore the functions propagating the trajectories should know the current point location, recognize the "ghost" intersections and ignore them.

The union, intersection, or subtraction of several volumes is also the volume. These actions are known as boolean operations[1, 4]. Especially important operation is the intersection, because it allows to assemble the complicated volumes from several simple surfaces like plane, sphere and so on.

The complicated installation is naturally to describe by a hierarchical structure of volumes. The examples of this approach are demonstrated by [1, 4]. The volume enclosed in another one represents either an empty space inside the solid or another solid embedded in the first one. There may be many embedded sub-volumes in the single mother volume, but none of them can project beyond the mother one. In the program [1] the sub-volumes can nevertheless overlap each other. Such sub-volumes usually have no relation with the real objects, but they artificially delimit regions with simple shapes for auxiliary purposes. Since this leads to complications and confusions, this is not allowed in our program. The treelike hierarchical structure of volumes begins from the root volume that is expected to cover all studied region. For the trajectory propagation the volume should provide the following information. For any point in the volume, either root or descendant, there can exist the hierarchy of sub-volumes in which the point is located at the same time. For any trajectory and given point on it there exists the shortest distance from the point to the border of the given volume or its sub-volume.

The volume shape is frequently much easier to define in a certain coordinate system "glued" to its particular point and direction. This point is typically the central point, for example, the center of the rectangular box. To make use of the volume this "local" system should be positioned (numerically described) with respect to the current (external) system. If we have the hierarchical multi-level volume structure, the shape of each volume is expected to be described in its local system, which should be allocated with respect to the system of the previous level. If the installation consists of many similar volumes, the user can reduce the memory consumption if he determines the shape once, and refers to this object as many times as necessary, each time associating it with a different coordinate system. This approach is consistently developed in GEANT [1, 4]. However, the possibility to describe the volume in the external system should also be preserved. The memory consumption at assembling a "granular" installation of the simplest forms like sphere can make their description in the external system even more preferable. Moreover, the user should be aware that the volume description in the local system is not an ordinary geometrical figure, because its numerical description is given in the coordinate system other than the current (external) one. Any transformation of the volume described in this way will not affect this description, but will correct the numerical representation of the coordinate system.

These general considerations have resulted in the design presented in fig. 2.

The volumes are described by two base classes, class `absvol` and class `manip_absvol`. The class `absvol`(abstract volume) describes the volume shape and provides access to the included sub-volumes. The class `manip_absvol` allocates the shape in a proper place by definition of the "local" coordinate system. The "local" coordinate system may not be defined at all if the volume can be numerically represented in the external system. The typical user's class representing the solid should inherit the attributes of these both classes.

The class `absvol` offers a few virtual functions expressing the principal volume features listed in the discussion above. Actual shapes should be described by its sub-classes. The class `manip_absvol` has similar methods, but they are redefined by such a way that the calls of the actual methods of the class `absvol` are preceded by conversion of all their parameters to the local system and are followed by the opposite conversion of the results. These methods have similar names, but to avoid confusion they are preceded by characters `m_`. The class `manip_absvol` contains neither the object of class `absvol` nor the "local" coordinate system but provides and uses itself generic access to them via the virtual functions. The function `Gavol()` returns address of `absvol`, and the function `Gasc()` returns address of `abssyscoor`. The default version of the latter function shown in fig. 2 returns zero pointer, which means that the system is the same. The sub-class `sh_manip_absvol` contains the "local" coordinate system and its own version of `Gasc()` returns the system's address. This sub-class is more convenient for ordinary applications.

There are four functions expressing the principal volume features. They denote the investigation whether the point is inside the volume (`check_point_inside(...)`), finding the hierarchy of the sub-volumes containing the point (`find_embed_vol(...)`), and two

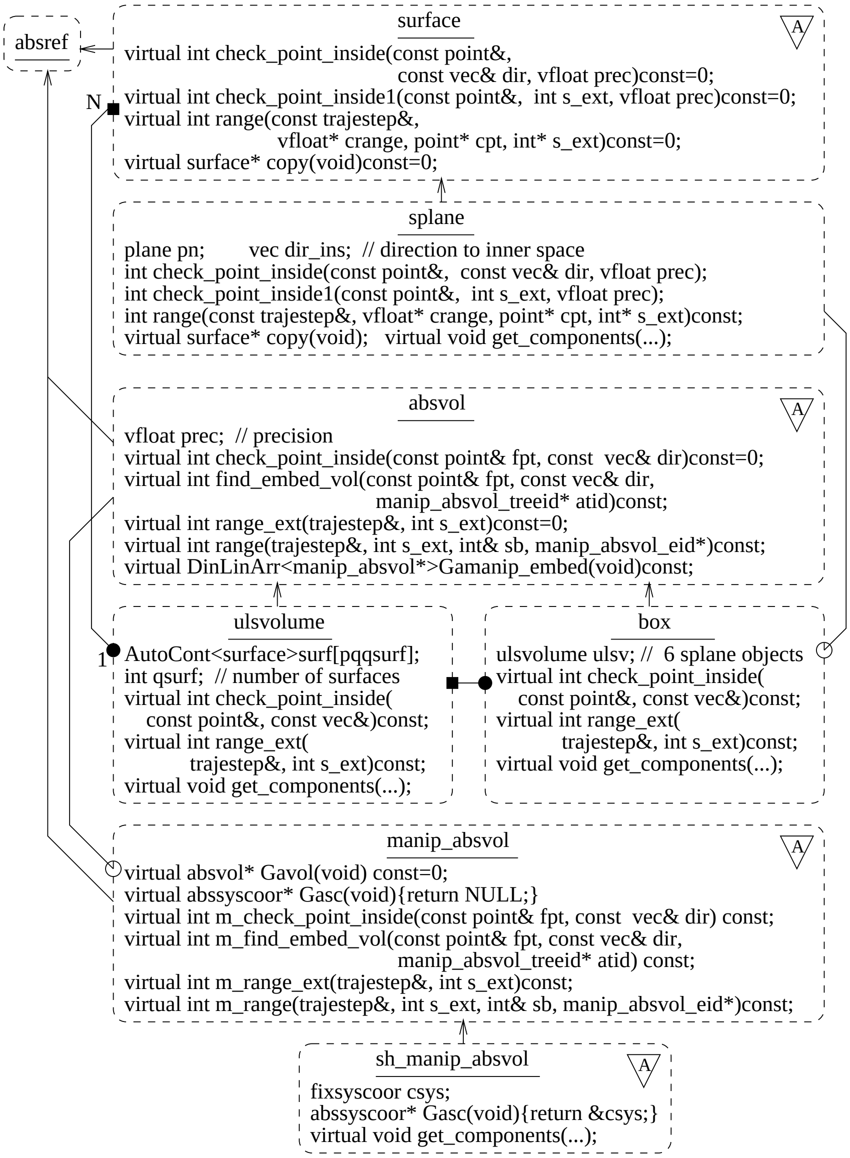

Figure 2: The class diagram of geometrical volumes.

range functions, one taking into account the sub-volumes (`range(...)`) and the other ignoring them (`range_ext(...)`). In fact, only two of four functions depend directly on the volume shape and therefore are declared pure in the abstract class `absvol`. These two functions do not take into account the sub-volumes. The standard logic of sub-volumes is removed to the other couple which is determined via the calls of the first one. To gain access to the sub-volumes the latter functions use the fifth virtual function `Gamanip_embed(void)`, which returns no sub-volumes in the base version and should be overloaded by the user if the sub-volumes exist.

Now we consider some auxiliary classes and variables which are crucial for understanding of the class diagram presented in fig. 2.

Two range functions use class `trajestep` that determines the current position, the trajectory direction, the curvature (in current version the point can move by the straight line or by the circle), the maximum possible step, the end point and some other auxiliary things. This is an ordinary geometrical figure in respect of inheritance of the class `absref` and the transformations. At the beginning the object of this class is initialized with the largest possible step, which is usually determined by required precision of extrapolation of the curved trajectory or by other reasons. All the range functions (except those of the class surface, see later) reduce the maximum possible step and correct the end point if they encounter on the nearer border. Since we will not describe the range functions in detail, there is no necessity to describe the class `trajestep` either.

The class `manip_absvol_eid`, which is not described either, contains the address of the object of the class `manip_absvol`, and its number in the external mother volume. The class `manip_absvol_treeid` has an array of `manip_absvol_eid` and the number of initialized elements in it. It gives the route from the root volume to the deepest one in which the point resides.

Two other parameters common for the range functions are `int s_ext` and `int& sb`. The first parameter `int s_ext` is an abbreviation of "sign of exiting" and it determines whether the point is expected to exit the volume and therefore is still in it (1), or expected to enter and now is outside the volume (0). Although this can be independently found by `trajestep`, this may be already known before the call of the range functions. The return parameter `sb`, the abbreviation of "sign of boundary", has the following meaning:

- sb=0: there is no boundary found along the allowed step.
- sb=1: crossing the boundary of this volume
- sb=2: crossing the boundary of an included sub-volume.

In the last case the parameter `manip_absvol_eid*` should contain the pointer to the included volume.

The principal volume functions are determined as follows:

- The function `int check_point_inside(...)` checks if the point is inside the volume. If it is inside, the function returns 1. As it was explained before, it takes into account not only the position of the point, but also the direction of motion.
- The function `int find_embed_vol(...)` calculates the hierarchy of embedded volumes in which the current point resides. Its return value is 1 if the point is in the given volume and 0 otherwise. This function calls `int check_point_inside()` for each relevant volume.
- The function `int range_ext(trajestep& fts, int s_ext)` (abridgment of "range to exit or enter") calculates the distance along the trajectory defined by `trajestep& fts` to the border of the given volume. If the intersection occurs at the distance less than the maximum step determined in `fts`, the function reduces the maximum step and returns 1.
- The function `int range(...)` is similar to `range_ext()`, however it takes into account the sub-volumes of given volume (of course, only if the point is inside the given volume; otherwise there is no difference from `range_ext()`) and finds the nearest

intersection with the boundaries of given or any sub-volume. This function calls range_ext() for each relevant volume.

The auxiliary function Gamanip_embed(void) returns the list of pointers to manipulators of volumes, class manip_absvol, enclosed into the current volume. The template class DinLinArr< T > is the local definition of 1-dimensional array with variable size. The default version of this function returns no sub-volumes. Thus, the volume contains no sub-volumes by default.

To construct any specific shape one needs to define the sub-class of class absvol and to overload the virtual functions check_point_inside() and range_ext(). On the first place we need the rectangular box and other polyhedrons, however the shapes with curved sides are also expected. To assemble the volume of several surfaces one can define the volumes isolated by every surface separately and then define their intersection. However, one would need an extra range function, because when the point is outside the whole volume, not all intersections of the trajectory with the surfaces are real. Moreover, such surfaces can not have the embedded volumes. Therefore, we define a specific abstract surface class that bears this very name, the class surface. The range function of this class receives the trajectory const trajestep& and calculates all intersections with the surface till the maximum allowed step but left the step unchanged. It returns the array of the points were the intersection occurs (point* cpt), the range to them (vfloat* crange), and the array of the signatures of exiting or entering the inner space at these points (int* s_ext). This class also has extra version of check_point_inside that uses s_ext instead of vector determining the direction of motion. The function copy() creates new instance allocating it in free memory and returning its address. This is the standard way to preserve inheritance.

All this information is treated in the functions of class ulsvolume, which is the sub-class of absvol, and it assembles the volume of several surfaces. This class contains the surfaces but it controls them via the pointers and allows inheritance. The pointer to each surface is kept in the array of templates AutoCont<surface>. AutoCont<T>

is an original version of the container class keeping object `T` in free memory and providing its correct copying. `pqqsurf` is a constant value. Thus, each instance of `ulsvolume` has individual copies of its surfaces. This is equivalent to containing the surfaces as the elements of `ulsvolume`. (Of course, the class `ulsvolume` can not contain the instance of the class `surface`, since the latter is the abstract class and can not have instances. Nevertheless, we have marked the relation as belonging of `surface` to `ulsvolume`, since this ever logic is implemented in the template `AutoCont<T>`. Since this template stores not the object `T` but the pointer to it, the template can handle the sub-classes of `T`. Therefore, the class `ulsvolume` actually contains the sub-classes of `surface`.)

The most useful real sub-class of the surface is the plane, class `splane`, which divides space on two semi-spaces. The inner one is pointed by `vec dir_ins`.

The classes `splane`, `absvol`, and `ulsvolume` are used for definition of rectangular box, class `box`. This approach makes possible simple determination of many other shapes.

This system of classes offers all necessary geometrical tools to perform the trajectory drawing. Indeed, simulation of the trajectory may be expressed by the following algorithm:

1. Finding in which volume the initial point is located. If it is outside the root volume, exit simulation.

2. Assign `s_life=1` (sign of life is one of the attributes of the moving point).

3. Go to item 6.

4. If `s_life` is not equal to 1, exit the simulation.

5. Making the step: changing the current position to the next.

6. Calculate the next anticipated position depending on whether or not the current point (the former next) has been obtained as an intersection with any volume boundary. If not, calculate the shortest step to the border of this volume or an

included sub-volume. If yes, initialize the next position with the same coordinates and determine the new volume in which it is located (if the particle is outside the root volume, make `s_life=0`).

7. Possible shortening the next step due to other reasons, physics interactions, etc.

8. Go to item 4.

This algorithm leads to definition of general particle class as containing, in particular, the current position, the next anticipated position , and their `sb` parameters (signatures of crossing with the boundary). The constructor of this class executes steps 1,2,6, and 7. Each step is performed by sequence of the steps 5, 6, and 7 and consists of moving the particle to the next position and calculation of the following anticipated one. Our geometrical system provides all necessary tools to program this procedure.

4 Example of application

To illustrate how this system can be applied we describe a hodoscope consisting of several scintillating counters. Each counter has the rectangular box shape and generates a signal characterized by an amplitude. To describe the counter's characteristics, shape, and position at once one can declare the following class (only the essential lines are given) :

```
class StintCounterVol:
        public sh_manip_absvol, // position of counter
        public box              // shape of counter
{public:
  int amplitude;        // signal of counter
  virtual absvol* Gavol(void) const
    {return const_cast<StintCounterVol*>(this); }
protected:
  virtual void get_components
```

```
               (AutoCont<absref_transmit>& aref_tran)
    {   // to modify the fixsyscoor at transformations:
      sh_manip_absvol::get_components(aref_tran);
    }
public:
   ...
};
```

The function Gavol is overloaded to return this object. Since two base classes are geometrical classes, the function get_components should also be overloaded to resolve the uncertainty. Call of the version of this function for class sh_manip_absvol results in modification of the coordinate system at the transformations.

The counters are the elements of the hodoscope. The hodoscope is also assumed to be a box of larger sizes and it should include the boxes of counters as embedded volumes. The other not geometrical counter's information should be freely acceptable by referring to counters as the hodoscope elements. The "natural" model is easily implemented with our class library:

```
const int qscount=10;
class HodoscopeVol:
        public sh_manip_absvol, // position of hodoscope
        public box              // shape of hodoscope
{public:
  StintCounterVol scount[qscount];   // counters
  virtual absvol* Gavol(void) const
    {return const_cast<HodoscopeVol*>(this); }
protected:
  virtual void get_components
                  (AutoCont<absref_transmit>& aref_tran)
    {   // to modify the fixsyscoor at transformations:
      sh_manip_absvol::get_components(aref_tran);
    }
public:
  // return array with addresses of counters
```

```
  // saved in array scount:
  virtual DinLinArr< manip_absvol * >
                  Gamanip_embed(void) const ;
  ...
};
```

The function `absvol::Gamanip_embed(void)` is overloaded by version providing pointers to the component counters. This is enough to provide all functionality of the package. If one adds the material features, one could run Monte-Carlo simulations. If one adds more experimental information, he could use this system for data treatment.

5 Summary

3-dimensional geometry offers a range of well-known concepts. Being adapted for computer modeling they allow to design and to code the convenient class library. The library is applicable for large variety of computing problems, but oriented toward the problems related to track propagation and particle penetration through experimental installations.

References

[1] GEANT, Detector Description and Simulation Tool, CERN Program Library, W5013, 1994.

[2] *Grady Booch.* Object-Oriented Analysis and Design with Applications, second edition. Addison-Wesley Publishing Comp., Russian edition: “Binom”, “Nevsky Dialekt”.

[3] *I.B. Smirnov.* Track Reconstruction for Forward Spectrometer of SPES4-π Experiment. Preprint PNPI-2345, 2000, p. 35.

[4] GEANT4, http://wwwinfo.cern.ch/asd/geant4/geant4.html

[5] CLHEP - A Class Library for High Energy Physics, http://wwwinfo.cern.ch/asd/lhc++/clhep/