Geometry modelling with `geomtools`

(Software/geomtools/GeometryModellingTutorial – version 0.1)

F. Mauger <`mauger@lpccaen.in2p3.fr`>

2011-12-01

Abstract

In this note, we explain the principles and tools for modelling a virtual geometry setup within `geomtools`.

Contents

1 Introduction

The `geomtools` program library provides some high-level general purpose utility classes and embedded algorithms to build virtual geometry models in a somewhat easy way. It has been designed as the main tool for feeding external programs with a standalone geometry description, using some automated exporting/filtering tools to interface with these third-party programs and libraries. In the fields of experimental high energy and nuclear physics, those programs are typically in charge of :

- the simulation of particle tracks through the virtual geometry model of a detector (GEANT4),
- reconstruction algorithm,
- visualization software,
- various step of the data analysis.

Unfortunately, there is not one single generic and standard tool or even paradigm to achieve such functionnalities. Usually, software libraries use their own geometry modelling scheme, with their own strategy, optimization and various policies that can make them non or hardly interoperable. Particularly, the choice for such a tool can prevent the user of client application to use in parallel another tool with different modeling techniques.
Because of this, we have implemented some geometry modelling interface (GMI) and tools that, we hope, are supposed to allow the making of new interfaces compatible with other geometry modelling implementations. Of course, there is no hope to cover all functionnalities you could dream about. However, the geometry modelling tools in `geomtools` try to address most practical cases in the framework of our experimental activities : detector simulation, visualization, reconstruction and data analysis.
The `geomtools` GMI relies on some standard modelling approach that have intrinsic limitations. Any virtual geometry model managed by `geomtools` uses a hierarchical system to describe the physical 3D-objects in a geometry setup. It deals only with placement of daughter objects in their mother volume at several levels of a possibly large hierarchy. There is no possibility to handle overlapping volumes or complex nested geometries with cyclic mother/daughter relationship. This apporach is shared by the GDML language, the GEANT4 and ROOT libraries which are compatible with this paradigm. So it is expected that `geomtools` will interface naturally with these implementations.

Subversion repository:
`https://nemo.lpc-caen.in2p3.fr/svn/geomtools/`
DocDB reference: `NemoDocDB-doc-1995`
References: see also *Geometry mapping with* `geomtools` (`NemoDocDB-doc-1996`)

2 Basic concepts

2.1 Logical volumes and physical volumes

First of all, the `geomtools` GMI provides an abstract interface that allows the description of the hierarchical relationship between physical objects in a geometry setup.
The key concept is the *logical volume* which describes the fundamental geometry properties of a 3D-object in a way that is independant of its placement in the whole setup. This is illustrated on figure 1 where several *objects* (copies) of a the same *type* are placed within a setup.

It this crucial to distinguish the *logical description* of a chair from its *physical implementations* (placements)) in the virtual world :

- the *logical volume* concept implement the description of intrinsic geoemtry properties (the *type*),
- the *physical volume* concept implement the description of the placement (position/rotation matrix) of an instance of some *logical volume* in the geometry setup (the instantiated *object*).

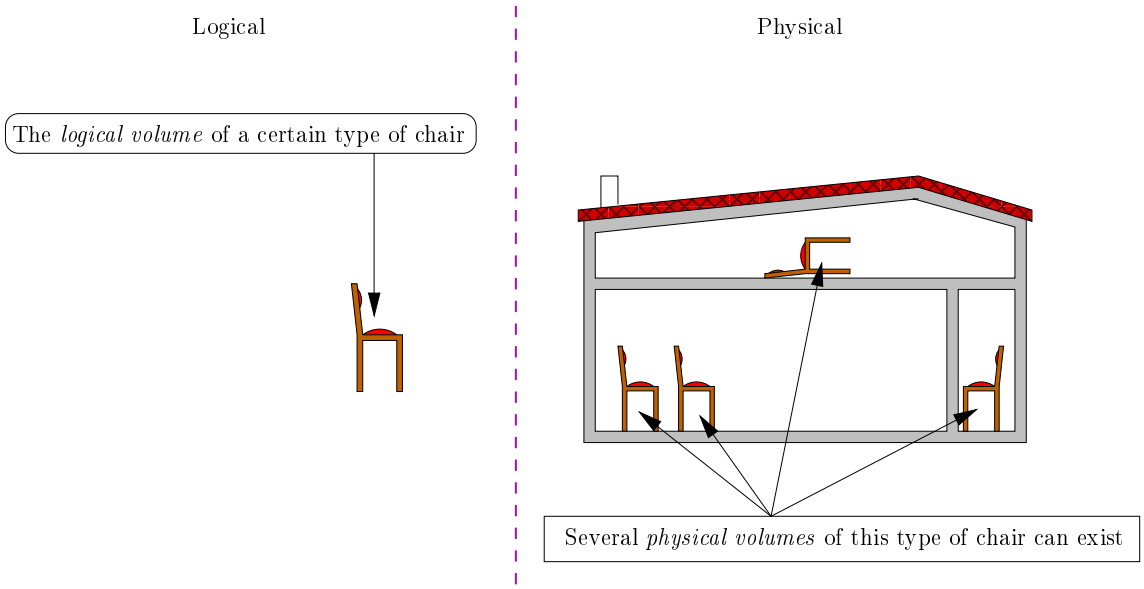

Figure 1: Several copies of a same type of object can be placed in the virtual geometry setup.

So what is a *logical volume* ? Following the approach of the GDML langage and GEANT4 modelling interface, we can list the informations that fully describe an instance of logical volume :

- a unique *name* that will allow to address the *logical volume* object non-ambiguously in a database of many *logical volumes*.

Examples: `"chair"`, `"table"`, `"desk"`, `"bedroom"`, `"bathroom"`, `"city_house"`, `"cottage"`...

- a 3D shape that will define the physical bounds in the 3D virtual space.

  Examples: a *box* of dimensions 3×2×1.3 m^3, a *cylinder* of radius r=25 cm and height h=75 cm...

- An optional list of daughter volumes that are fully contained in the bounding limits (the 3D shape) of the logical volume. This implies that we know where to place these daughter volumes. We thus need to provide not only their own *logical volumes* but also their position and orientation (placement) in the current logical volume that is called the *mother* volume.

Figure 2 shows two different descriptions of some simple volumes. Despite their external envelopes share the same shape and dimensions, the left and right "boxed" logical volumes have significant differences. They do not have the same colour (may be because they are made with different materials) and the blue one contains some daughter box objects while the left one has no daughters. As the blue logical volume contains two tiny red boxes, we must provide the coordinates (position/rotation) for the placement of these daughter volumes. More it is obvious that the red boxes are copies (*physical volumes* that share the same description; we must also provide the description of this third *logical volume*.

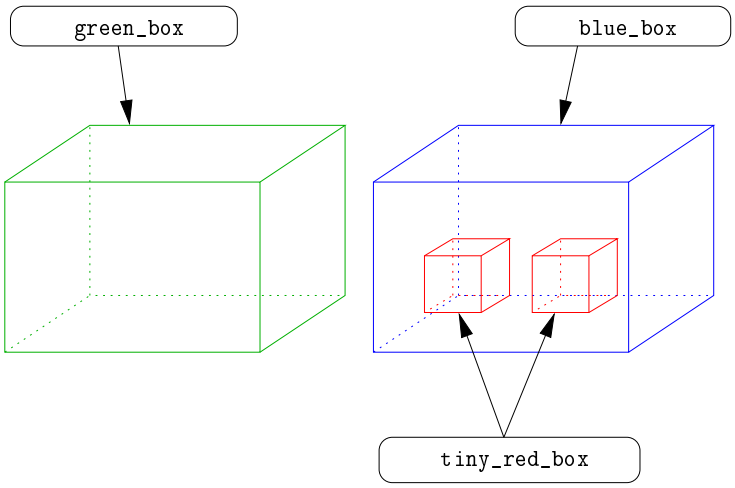

Figure 2: Two different simple logical volumes.

We see here that the full description of a geometry setup will be given by a more or less complex hierarchy of physical volumes nested in logical volumes, in turn nested in physical volume at the parent level and so on...

Typically the geometry description looks like a large tree of an arbitrary depth which depends on the number of nested hierarchy levels:

```
"world.log"(top-level of the hierarchy)
|-- Material: "air"
```

```
|-- Colour: "transparent"
|-- Shape: "box" with x,y,z=(30,20,10) m
`-- Daughters:
    |-- Physical: "house_0.phys"
    |    |-- Position/rotation = (3,-2,0) m / R(Oz, 90°)
    |    `-- Logical: "house.log"
    |        |-- Material: "air"
    |        |-- Colour: "transparent"
    |        |-- Shape: "box" with x,y,z=(30,20,10) m
    |        `-- Daughters:
    |            |-- Physical: "ground_floor.phys"
    |            |   |-- Position/rotation = (1,2,0) m / R(Oz,0°)
    |            |   `-- Logical: "ground_floor.log"
    |            |       |-- Material: "concrete"
    |            |       |-- Colour: "gray"
    |            |       |-- Shape: "box" with x,y,z=(...) m
    |            |       `-- Daughters:
    |            |           |-- Physical: "kitchen.phys"
    |            |           |    |-- Position...
    |            |           |    `-- Logical: "kitchen.log"
    :            :           :
    |-- Physical: "house_1.phys"
    |    |-- Position/rotation = (3,-2,0) m / R(Oz, 90°)
    |    `-- Logical: "house.log"
    |        |-- Material: "air"
    |        |-- Colour: "transparent"
    |        |-- Shape: "box" with x,y,z=(30,20,10) m
    |        `-- Daughters:
    |            |-- Physical: "ground_floor.phys"
    :            :
    |            `-- Physical: "last_floor.phys"
    :
```

Practically, the `geomtools` API provides two classes:

- the `geomtools::logical_volume` class,
- the `geomtools::physical_volume` class.

Basically its API is a clone of what can be found in the GEANT4 program library. This is natural as both APIs follow the GDML geometry modelling approach. However `geomtools` allows to associate arbitrary properties to any logical or physical object. This allows client applications to benefit of some tools for storage and/or fetching high-level meta-data. This mechanism is used to pass visualization informations to some 3D-display program (visibility, colour...), material information for physics simulation programs (GEANT4), directives to the numbering scheme manager (*mapping*)... Some naming conventions are of course needed to ease the extraction of arbitrary informations by topic. This feature relies on the `datatools::utils::properties` container class. By essence, the availability of this functionnality make this mechanism extensible to other applications.

2.2 Geometry models

If the `geomtools` API had proposes only a rewritting of the GEANT4 interface and/or the GDML modelling approach, even with a few more features added in it, it would have been of limited interest. Indeed this API is *based* on a similar interface to GEANT4/GDML, but implements higher level functionnalities.

The key idea here is to obtain a very compact and efficient way to describe a geometry setup without entering the guts of the nested logical/physical volume hierarchy and the expertise needed to manipulate and navigate through this hierarchy. More, we would like to implement some tools to:

- automate the construction of a transcient virtual geometry model using a *geometry engine* that uses only a limited set of configuration parameters to build a hierarchy of 3D-volumes of arbitrary complexity,
- benefit of a collection of generic objects that represents very often used geometry patterns : volumes with very familiar shapes (box, cylinder...), stacked volumes, replicated volumes, composite volumes (union, intersection, differences)...,
- automate a simple 3D-rendering for fast debugging and developpement,
- automate the conversion of any geometry model in the format of another API (GDML, GEANT4, ROOT...),
- automated the management of a numbering scheme policy,
- enable some arbitrary meta-data to be attached to any node of the hierarchy tree,
- enable extension with possible new geometry primitives or hard-coded descriptions of new logical volumes if generixity cannot be achieved,
- interoperability between generic components and hard-coded components,
- be human-friendly with ASCII file based configurations,
- hide the complex memory management of the transcient geometry model[1].

Within GDML, part of this features (flexibility, human readable configuration files, some filters to GEANT4 and ROOT...) are addressed through the grammar and syntax of this XML-based language. However, the manipulation of XML files turns to be difficult as the complexity of the geometry system increases. Despite the possibility to use *parametrized logical volumes*, on-the-fly computed positionning is limited. More it does not provide a standalone transcient geometry model : you have to choose the transcient model from the GEANT4 or ROOT library but it is difficult to used both systems in cooperation within the same program or in the context of a third-party application that has its own modelling scheme and approach (your data analysis and event reconstruction for example). However, GDML files are a good interface medium and we will use this technique as the core of the interface with GEANT4.

[1] You may have a look on GEANT4 detector construction and all the pointers users have to play with !

As the handling of many logical/physical volumes is complex and request some programming expertise (memory management, pointers, knowledge of some specific API...), `geomtools` proposes a special concept to automated and hide most parts of this low-level techniques, still providing the user or external application a way to manipulate the concepts of hierarchical geoemtry modelling. A new interface has been implemented : the *geometry model*.

A *geometry model* has the responsability to describe the characteristics of a given *logical volume*, in such a way a logical volume is always instantiated through its associated geometry model. However, a geometry model is for the logical volume what is a class for an object. Thus a geometry model may have parametrization facilities that enable to instantiate different kinds of *logical volumes*; such logical volumes will be rather similar because they are managed/created by the same *geometry driver* and they will *behaves* in the same way, .

The difference between the *geometry model* and the *logical volume* concepts is not obvious for very simple 3D objects (a simple box or cylinder without any daughter volumes): the geometry model for a simple box-shaped volume made of copper will be responsible of the instantiation of the logical volume made of a box shape associated to copper material. The figure 3 represents a `simple_boxed_model` geoemtry model that is able to instantiate a logical volume named `"my_green_box"`. The user (on the left) just has to send a request to the `simple_boxed_model` object that behaves like a *logical volume factory* and instantiates automatically the `"my_box"` logical volume given some specific parameters (w, h, d, colour, material...) that are passed when the user's request is submitted.

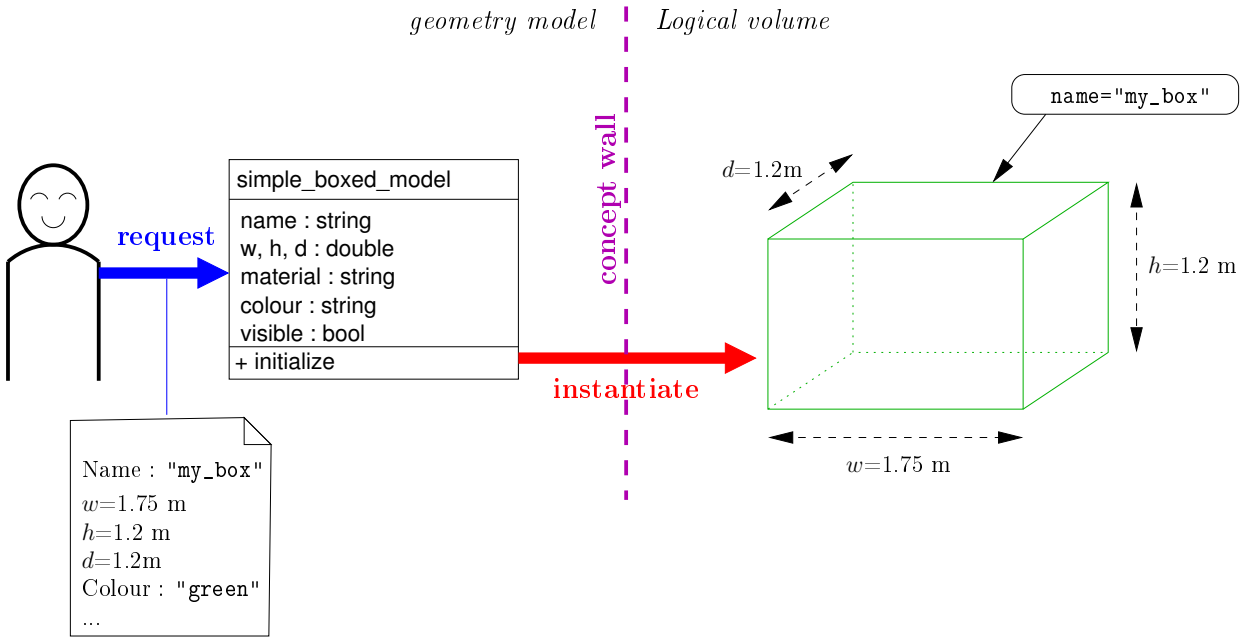

Figure 3: Instantiation of a logical volume through a geoemtry model driver.

Introducing the *geometry model* layer just adds an intermediate step but does not add functionnalities. We could have directly create the logical volume by hand using low-level functionnalities of the API (figure 4).

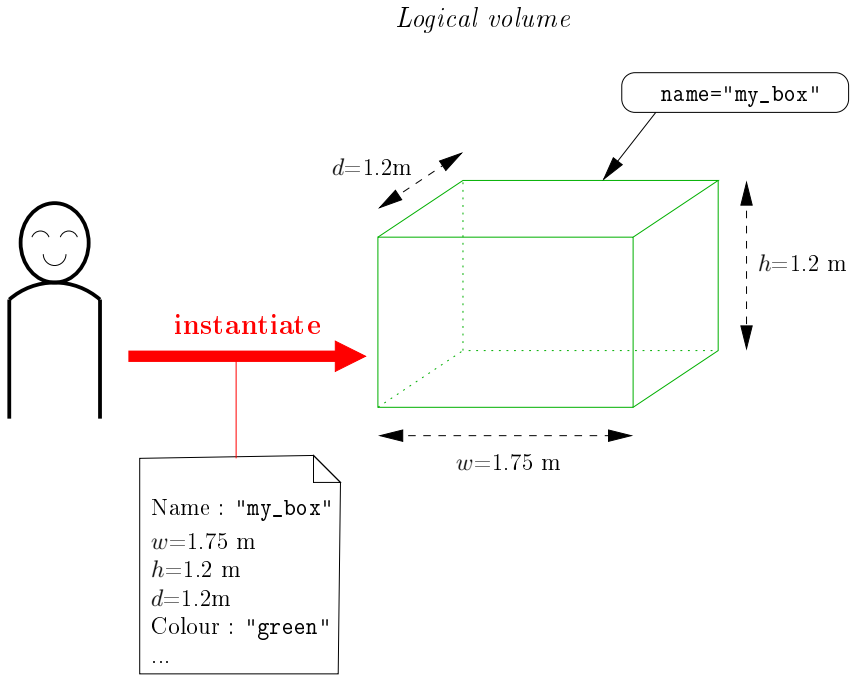

Figure 4: Instantiation of a logical volume through the native API.

However, as soon as we want to manipulate some complex logical volumes, the difference is fondamental. Suppose we want to build a logical volume that stacks three boxes of different dimensions along an arbitrary axis. To get this in GEANT4, you will have to instantiate first the three logical volumes corresponding to each box. Then you will create a logical volume, choose a shape for it (say a box), compute its dimensions from the dimensions of the internal boxes you want to stack, position the daughter boxes (yes we speak about daughter physical volumes) along the given axis, check that there is no overlapping volumes and that daughters are fully contained in the mother box. Such case is illustrated on figure 5.

This is some work ! Each time you will have to stack objects in this way, you will have to reproduce this algorithm, manage the memory associated to the volumes, compute all requested geometry parameters. In real life, it appears that stacking volume is a frequent operation : in fact this is a current *geometry modelling pattern*. This is exactly the moment for a dedicated *geometry model* to enter the scene ! It is possible to implement a generic algorithm of which the task is to perform all the complex operations needed to obtain a set of stacked volumes enclosed in some mother volume. Not only the geometry model for constructing such stacked volumes will use this algorithm, it will also manage al the internals : memory stuff, mother/daughter relationships, conventionnal naming of the internal objects/volumes (figure 6).

Practically, the designer of a virtual geometry setup will never directly manipulate logical and/or physical volumes. He will deal only with geometry models. This approach defers the complex technical part of the modelling to the `geomtools` embedded geometry engine. The user can concentrate only on the building of the fondamental objects that enter the composition of the setup and their relative relationships in terms of hierarchy.

Of course, logical and physical volumes still exist and are used within the internals

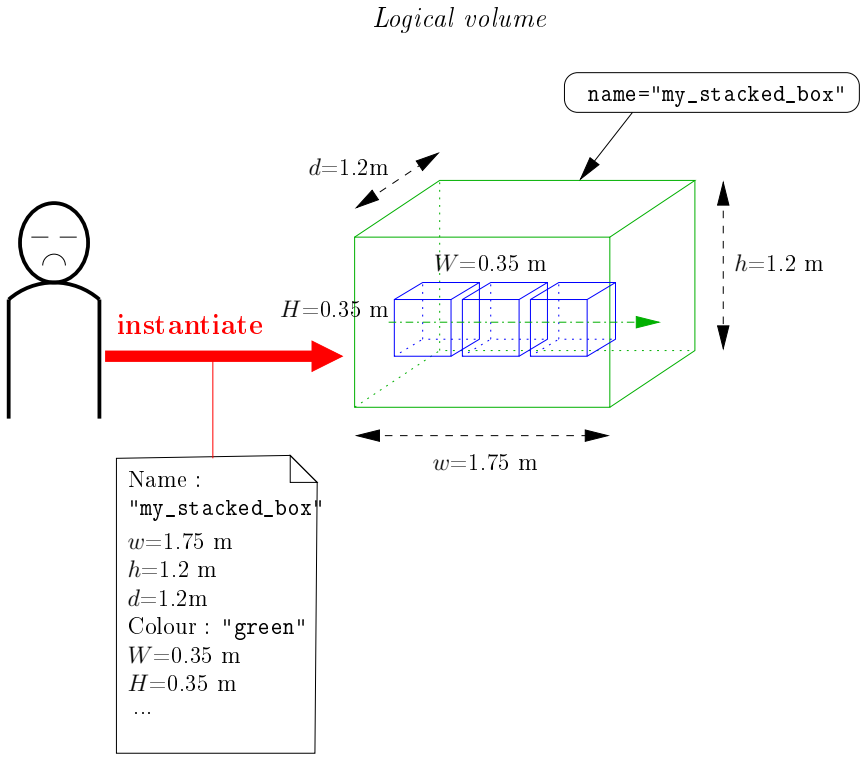

Figure 5: Instantiation of a complex logical volume through the native API.

of the `geomtools` engine. These objects are the core of the hierarchy geometry tree. So it is natural that developpers will still have to handle logical and physical volumes when they design new geometry model/driver classes and propose extension to the library of already available models.

A *model factory* class has been implemented as the main engine responsible of the construction of a geometry hierarchy. Given some geometry configuration files, it automates the allocation of requested geometry models, checks (partially) the coherence of the system, generates the associated logical and physical volumes and makes a full transcient geometry model available to the user. The class is named `geomtools::model_factory`.

More all generic and primitive geometry models available from the library benefit of an automated class registration mechanism based on some internal lookup table. The instantiation of geometry models is thus completely transparent, as well as memory management issues. This mechanism also extends to the geometry model classes implemented by developpers to handle special cases where no combination of the existing pre-registred models can cover users' needs. This makes the system rather generic and extensible.

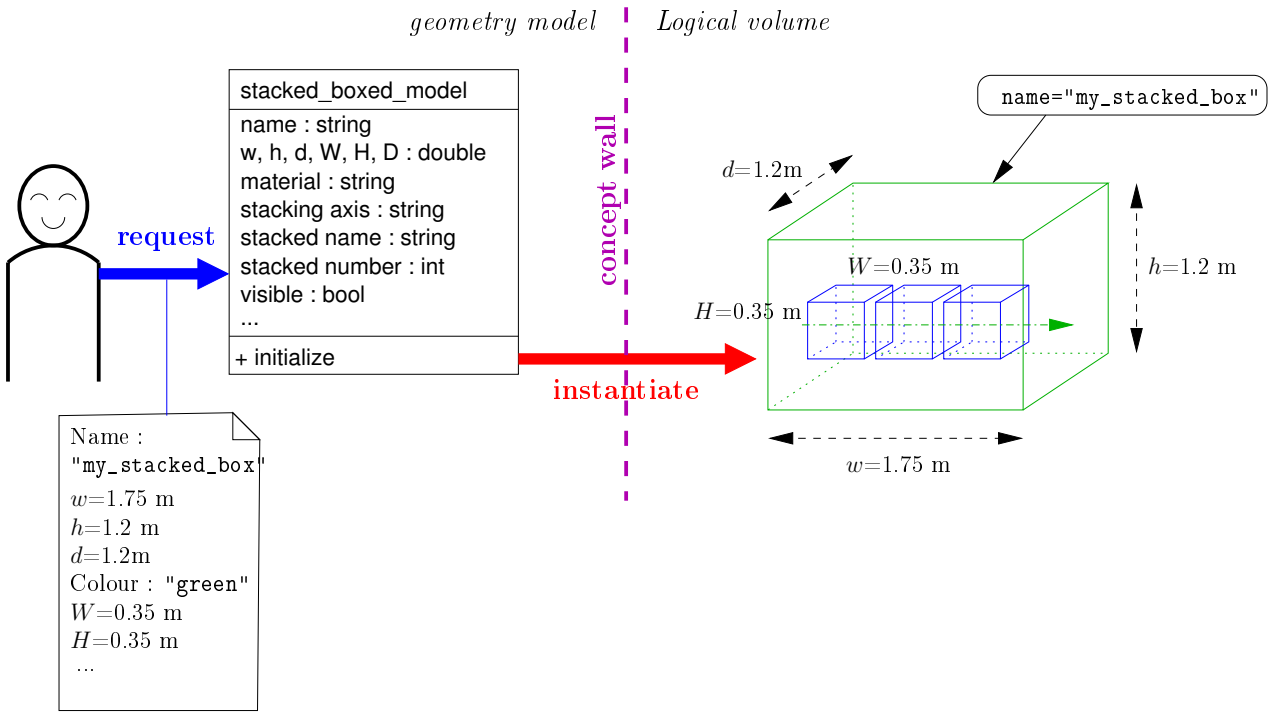

Figure 6: Instantiation of a complex logical volume through a smart geometry model driver.

3 Use cases

Now it is time to play with geometry models and the geometry factory. We will start by a very simple case and will introduce more and more complexity progressively.

3.1 A box in a virtual world

We want to achieve the construction of the virtual setup shown on figure 3.1. We have here a simple top level large green box (the *world*) with given dimensions and auxiliary properties (color, material...). This *world* box contains a single smaller red box with its own geometry and auxiliary properties. The placement of the red box inside the mother green box is determined by the position of the center O' of the red box with respect to the center O of the green box. The orientation of the red box inside the mother green box can be determined by the Euler angles associated to the rotation matrix that transform the $(0xyz)$ world reference frame axis into the $(0'x'y'z')$ daughter frame axis.

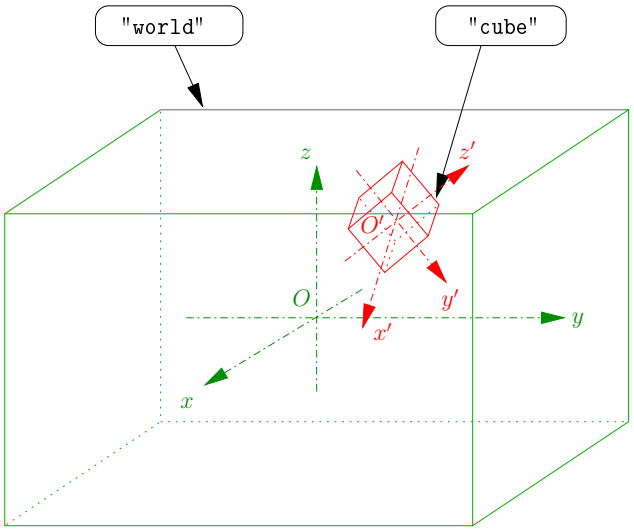

Figure 7: A very simple world.

With `geomtools`, the easiest way to construct this virtual geometry setup is to use two available geometry model classes :

- the `geomtools::simple_world_model` class is designed to model a top-level box with arbitrary dimensions. It can contain one and only one daughter volume at any position and orientation: the `"setup"` volume. This daughter volume can be modeled by any other geometry model available at run-time in the library.
- the `geomtools::simple_shaped_model` class is designed to address very usual cases : volume with a simple shape like box, cylinder, sphere.... Optionally it can contain some daughters volumes.

Both `geomtools::simple_world_model` and `geomtools::simple_shaped_model` are registered by default in the `geomtools`' *model factory*.

What we need now is to provide some configuration file with all the directives that reflect the layout seen on figure 3.1. The file will use by convention the `.geom` extension, note however it is not mandatory. The format is the ASCII encoding of the `datatools::utils::multi_properties` class. The principle is to provide one section of properties per geometry model described in the setup. Here we will have two section : one for the top-level world model, the second for the daughter box in it. Note that blank lines are ignored as well as line starting with the # character. There is an exception with special meta-comments starting with `#@` that are part of the syntax of the `datatools::utils::multi_properties` and embedded `datatools::utils::properties` objects. This meta-comments are `#@description` and `#@config` (see the example below).

We first create a file named "`simple_world_1.geom`" and we write its header as shown on sample 1.

```
1 @description List of geometry models and their relationships in a simple world
2 #@key_label   "name"
3 #@meta_label  "type"
```

Sample 1: The header of the "`simple_world_1.geom`" file.

This meta information, stored as meta-comments, inform the parser for the `datatools::utils::multi_properties` object that each section will have a main key labeled with `name` and an additional tag labeled `type`. The *name* will be used as the primary key to access each geometry model from an internal look-up table. The *type* is a character string that identifies the unique geometry model class that must be used to instantiate the corresponding geometry model and its associated logical volume; it is thus used by the internal model factory.
Now we are done with the logistics of *multi_properties* and *factory* objects, we can describe the two models we need. Here again there is an important constraint. As the small red box is to be inserted in the lard world green box. The geometry model of the green box will *depend on* the existence of the small red box. On the other side, in this hierarchy, the small red box does not need to know anything about the large green box. In principle, we could have chosen to put it in another cylindrical purple universe. So to reflect this mother/daughter dependency, we **must** first declare the geometry model for the small red box. Then only we will provide the definition of the model that will instantiate the green world.
The small red box use a very simple shape. More, as a terminal leaf of the hierarchy, it contains no daughter. This simple case is addressed by the `simple_shaped_model` provided by the `geomtools` API.
The file sample 2 shows the section for the *small red box*.
Note that the mandatory parameters are:

- `shape_type`, `x`, `y`, `z` : for a full geometry description of the volume,
- `material.ref` for this is a crucial physical property for client application.
  Remark: this behavior should be change in the future to support, for debugging purpose, a default material when this property is missing in the file.

```
[name="small_red_box" type="geomtools::simple_shaped_model"]

#@config the list of parameters that define a small red box

#@description the shape of the volume
shape_type        : string =  "box"   # must be supported by the model class

#@description the unit for all length dimensions (see above)
length_unit       : string = "cm"     # note: we use CLHEP unit system
x                 : real   = 20.0     # dimensions of the shape;
y                 : real   = 25.0     #  must be coherent with the
z                 : real   = 15.0     #  type above (box)

#@description the name of the material the box is made of
material.ref      : string  = "__default__" # must be known by an external agent

#@description optional visibility flag
visibility.hidden : boolean = 0        #  hidden flag

#@description optional visibility information
visibility.color  : string  = "red"    #  display color
```

Sample 2: The *small red box* section of the "`simple_world_1.geom`" file.

Visibility parameters (`visibility.hidden`, `visibility.color`) are optional. However, because there are used by the Gnuplot based fast visualization program provided in `geomtools`, we recommend to use them. They will also be passed to the GEANT4 Open-GL visualization driver.
Believe it or not, these few lines will generate all the software machinery to handle the 3D-box object, set its dimension, add some properties in it, and allocate the associated logical volume. This is transparent to the user. At the end of the processing of these lines by the geometry model factory, a new object named "`small_red_box`" is inserted in a dynamic internal database for further usage. This object now just has to wait to be used by some other (mother) geometry model. Be patient, the *world* is coming !
So what about the *world* volume ? As mentioned above, the `geomtools::simple_world_model` has been implemented in this purpose : hosting a single 3D object in a boxed universe. Let's write the *world* section ! The file sample 3 stores the directives that must be written **after** the "`small_red_box`" section :
It can be seen here that this model needs more information to be properly described. Not only it requires the dimensions, material and visibility parameters, but it also needs some geometry informations for the placement of the setup volume it contains. The configuration properties are rather self explanatory.
The C++ program 1 illustrates a minimal use of the `geomtools::model_factory` class to construct the virtual geometry model that corresponds to the directives stored in the "`simple_world_1.geom`" file. Once the factory object has loaded the file and has been locked, the transcient geometry hierarchy model is built and the program prints its structure on the terminal.
The following C++ program 2 shows how to pass the geometry hierarchy model built by the *geometry model factory* to some special driver : a Gnuplot renderer (visualization)

```
[name="world" type="geomtools::simple_world_model"]

#@config the list of parameters that define the top-level large green world box

#@description the name of the material the box is made of
material.ref    : string = "vacuum"       # must be known by an external agent

#@description the name of the model that is contained in the world volume
setup.model     : string = "small_red_box" # we use here the name of the
                                           # only available model

#@description the unit for all angular dimensions (see below)
angle_unit      : string = "degree" # CLHEP system

#@description  ZYZ Euler first angle
setup.phi       : real   = 80.0

#@description  ZYZ Euler second angle
setup.theta     : real   = 60.0

#@description  ZYZ Euler third angle
setup.delta     : real   = 30.0

#@description the unit for all length dimensions (see below)
length_unit     : string = "m"  # CLHEP system

#@description x position of the setup in the world coordinate system
setup.x         : real   = 0.4

#@description y position of the setup in the world coordinate system
setup.y         : real   = 0.5

#@description z position of the setup in the world coordinate system
setup.z         : real   = 0.6

#@description dimension of the world box on the x axis
world.x         : real   = 1.5 # must be larger enough to fully enclose the setup

#@description dimension of the world box on the y axis
world.y         : real   = 1.5 # must be larger enough to fully enclose the setup

#@description dimension of the world box on the z axis
world.z         : real   = 1.5 # must be larger enough to fully enclose the setup

#@description optional visibility flag
visibility.hidden : boolean = 0        #  hidden flag

#@description optional visibility information
visibility.color  : string  = "green"    #  display color

#@description optional visibility information for daughters
visibility.daughters.hidden : boolean = 0 # only the 'setup' daughter is affected
```

Sample 3: The *world* section of the "`simple_world_1.geom`" file.

and a GDML filter. Figure 8 shows the Gnuplot based 3D-display originated from the program. We have been able to achieve the goal in figure . Sample 4 shows the contents of the GDML file that is generated by the GDML export driver object.

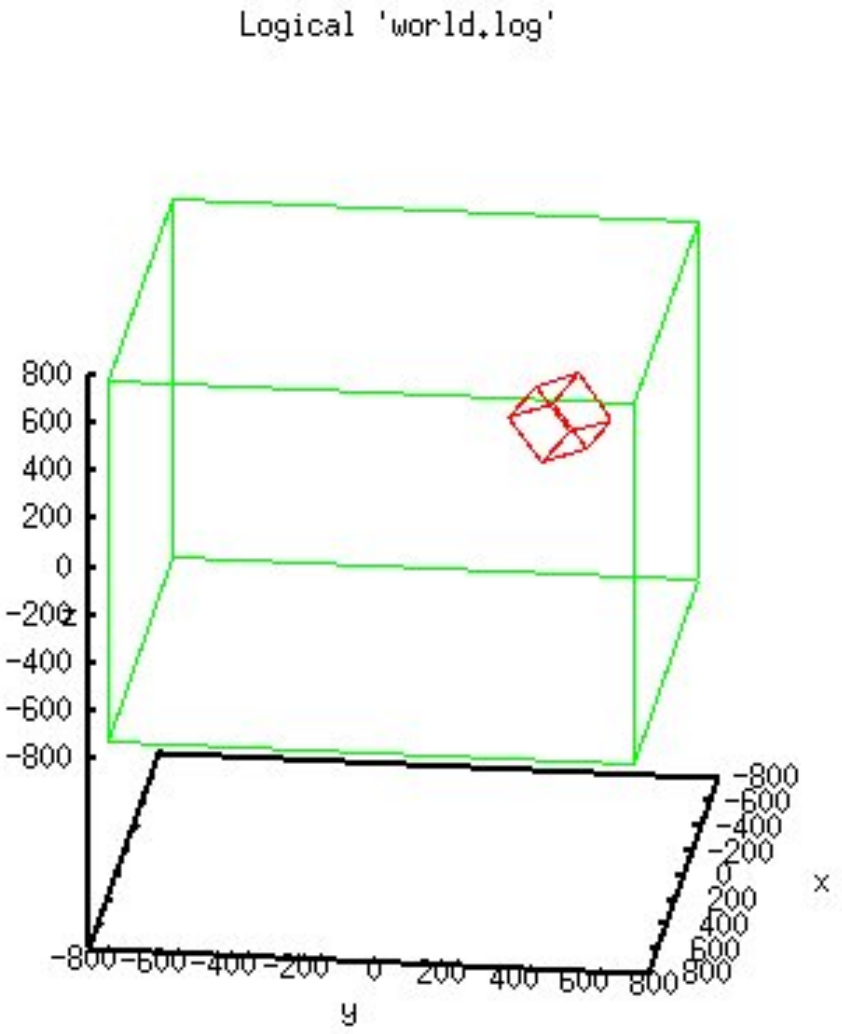

Figure 8: The Gnuplot display of the simple virtual world constructed from the file `simple_world_1.geom` built by the 2 program.

```
// -*- mode: c++ ; -*-
// simple_world_1.cxx

#include <cstdlib>
#include <iostream>
#include <exception>

#include <geomtools/model_factory.h>

int main (void)
{
  int error_code = EXIT_SUCCESS;
  try
    {
      // Declare a GID manager :
      geomtools::model_factory the_model_factory;

      // Load the configuration file for geometry models :
      the_model_factory.load ("simple_world_1.geom");

      // Lock the model factory and trigger the construction :
      the_model_factory.lock ();

      // Dump the model factory :
      the_model_factory.tree_dump (std::clog);

    }
  catch (exception & x)
    {
      std::cerr << "error: " << x.what () << std::endl;
      error_code = EXIT_FAILURE;
    }
   return error_code;
}

// end of simple_world_1.cxx
```

Program 1: A program for the construction of the virtual geometry setup described in the "`simple_world_1.geom`" file.

```
// -*- mode: c++ ; -*-
// simple_world_2.cxx

#include <cstdlib>
#include <iostream>
#include <exception>

#include <geomtools/model_factory.h>
#include <geomtools/gnuplot_drawer.h>
#include <geomtools/gdml_export.h>
#include <geomtools/placement.h>

int main (void)
{
  int error_code = EXIT_SUCCESS;
  try
    {
      // Declare a GID manager :
      geomtools::model_factory the_model_factory;
      the_model_factory.load ("simple_world_1.geom");
      the_model_factory.lock ();

      // Declare a Gnuplot renderer :
      geomtools::gnuplot_drawer GPD;
      GPD.set_view (geomtools::gnuplot_drawer::VIEW_3D);
      GPD.set_mode (geomtools::gnuplot_drawer::MODE_WIRED);
      geomtools::placement reference_placement;
      reference_placement.set (0 * CLHEP::m, 0 * CLHEP::m, 0 * CLHEP::m,
                               0 * CLHEP::degree, 0 * CLHEP::degree, 0);
      GPD.draw (the_model_factory,
                "world",
                reference_placement,
                geomtools::gnuplot_drawer::DISPLAY_LEVEL_NO_LIMIT);

      // Declare a GDML export driver :
      geomtools::gdml_export GDML;
      GDML.export_gdml ("simple_world_2.gdml", the_model_factory, "world");

    }
  catch (exception & x)
    {
      std::cerr << "error: " << x.what () << std::endl;
      error_code = EXIT_FAILURE;
    }
   return error_code;
}

// end of simple_world_2.cxx
```

Program 2: A program that extends the functionnalities of program 1 and display a simple 3D view of the geometry. It produces also a GDML file that describes the geometry setup.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<gdml xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="http://service-spi.web.cern.ch/service-spi/app/releases/GDML/schema/gdml.xsd"  >

<define>

<position name="world.log.setup.phys.pos" x="400" y="500" z="500" unit="mm" />

<rotation name="world.log.setup.phys.rot" x="59.6187448575295" y="-8.64916510528758" z="-84.9616312267025" unit="deg" />

</define>


<materials>

<material name="__default__" Z="1" >
  <D value="1" unit="g/cm3" />
  <atom value="1" />
</material >

<material name="__unknown__" Z="1" >
  <D value="1" unit="g/cm3" />
  <atom value="1" />
</material >

<material name="vacuum" Z="1" >
  <D value="1e-15" unit="g/cm3" />
  <atom value="1" />
</material >

</materials>


<solids>

<box name="world.log.solid" x="1500" y="1500" z="1500" lunit="mm" />

<box name="small_red_box.log.solid" x="200" y="250" z="150" lunit="mm" />

</solids>


<structure>

<volume name="small_red_box.log" >
  <materialref ref="__default__" />
  <solidref    ref="small_red_box.log.solid" />

  <auxiliary auxtype="material.ref"  auxvalue="__default__" />
  <auxiliary auxtype="visibility.color"  auxvalue="red" />
  <auxiliary auxtype="visibility.hidden"  auxvalue="0" />

</volume>

<volume name="world.log" >
  <materialref ref="vacuum" />
  <solidref    ref="world.log.solid" />

  <physvol>
    <volumeref   ref="small_red_box.log" />
    <positionref ref="world.log.setup.phys.pos" />
    <rotationref ref="world.log.setup.phys.rot" />
  </physvol>


  <auxiliary auxtype="material.ref"  auxvalue="vacuum" />
  <auxiliary auxtype="visibility.color"  auxvalue="green" />
  <auxiliary auxtype="visibility.daughters.hidden"  auxvalue="0" />
  <auxiliary auxtype="visibility.hidden"  auxvalue="0" />

</volume>

</structure>


<setup name="Setup" version="1.0" >
  <world ref="world.log" />
</setup>


</gdml>

```

Sample 4: The GDML file generated by the program 2 from the setup described in the "`simple_world_1.geom`" file. Here a default list of materials is added by the driver. In a practical case, a list of materials is inserted by an external software agent.

3.2 A virtual world with two identical boxes in it

We now address a somewhat more complex setup : we want two identical small red boxes to be placed within the world volume at different (no overlapping) positions and orientations.

We thus create a new "`setup_2.geom`" file to describe this new geometry layout. Here the description of the "`small_red_box`" geometry model is unchanged with regards to the previous setup (sample 2). But now, as we want to place two objects within the *world* volume, we cannot use the former `geomtools::simple_world_model` geoemtry model. In place, we will use a `geomtools::simple_shaped_model` model (with a "`box`" shape) but we will add some directives in the "`world`" section to inform the *world* volume that it contains two boxes. To achieve this we use special directives prefixed by `internal_item`.
The file sample 5 shows the directives that must be written in the "`world`" section of file "`setup_2.geom`" in order to describe the new world volume :

```
[name="world" type="geomtools::simple_shaped_model"]

#@config the list of parameters that define the top-level large green world box

shape_type   : string = "box"    # must be supported by the model class
length_unit  : string = "m"      # note: we use CLHEP unit system
x            : real   = 1.5      # dimensions of the shape;
y            : real   = 1.5      #   must be coherent with the
z            : real   = 1.5      #   type above (box)
material.ref : string = "vacuum" # must be known by an external agent

#@description the names of two internal objects placed in the world volume
internal_item.labels : string[2] = "box1" "box2" # we must use here some
                                                 # distinct names

#@description the model of the first box
internal_item.model.box1     : string  = "small_red_box" # we must use here some
                                                         # a known model name

#@description the placement of the first box in the mother reference frame
internal_item.placement.box1 : string  = "0.4 0.5 0.6 (m) @ 80.0 60.0 30.0 (degree)"
  # note here the syntax for the rotation using ZYZ Euler angle

#@description the model of the second box
internal_item.model.box2     : string  = "small_red_box" # we must use here some
                                                         # a known model name

#@description the placement of the second box in the mother reference frame
internal_item.placement.box2 : string  = "0.2 -0.3 -0.6 (m) / z 30.0 (degree)"
  # note here the syntax for the simple rotation along the z axis

visibility.hidden           : boolean = 0       # hidden flag
```

Sample 5: The *world* section of the "`setup_2.geom`" file.

Using the `setup_construct_and_view.cxx` program (see program 3), we obtain the

display in figure 9.

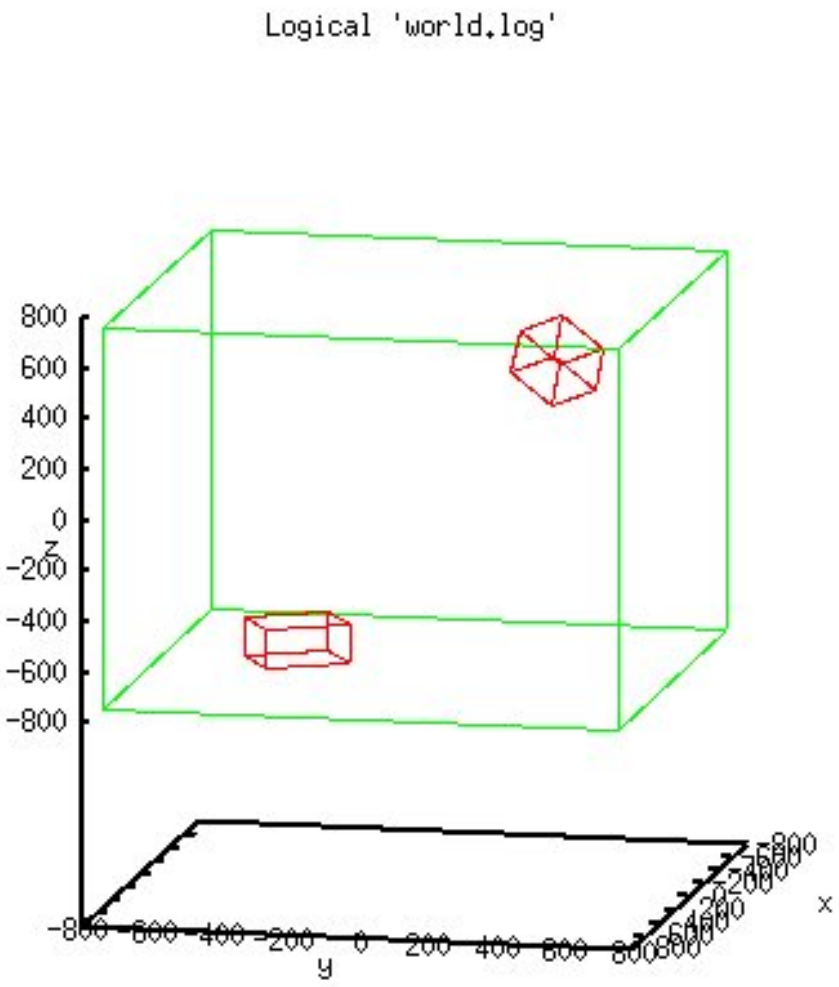

Figure 9: The Gnuplot display of the virtual world constructed from the file `setup_2.geom` built by the 3 program.

```
// Example: setup_construct_and_view "setup_2.geom" "world"

#include <cstdlib>
#include <iostream>
#include <stdexcept>

#include <geomtools/model_factory.h>
#include <geomtools/gnuplot_drawer.h>
#include <geomtools/placement.h>

int main (int argc_, char ** argv_)
{
  int error_code = EXIT_SUCCESS;
  try
    {
      if (argc_ < 2)
        {
          throw std::logic_error ("Missing .geom filename !");
        }
      string geom_file = argv_[1];
      string model_name = "world";
      if (argc_ >= 3)
        {
          model_name = argv_[2];
        }

      // Declare a GID manager :
      geomtools::model_factory the_model_factory;
      the_model_factory.load (geom_file);
      the_model_factory.lock ();

      // Declare a Gnuplot renderer :
      geomtools::gnuplot_drawer GPD;
      GPD.set_view (geomtools::gnuplot_drawer::VIEW_3D);
      GPD.set_mode (geomtools::gnuplot_drawer::MODE_WIRED);
      geomtools::placement reference_placement;
      reference_placement.set (0 * CLHEP::m, 0 * CLHEP::m, 0 * CLHEP::m,
                               0 * CLHEP::degree, 0 * CLHEP::degree, 0);
      GPD.draw (the_model_factory,
                model_name,
                reference_placement,
                geomtools::gnuplot_drawer::DISPLAY_LEVEL_NO_LIMIT);

    }
  catch (exception & x)
    {
      std::cerr << "error: " << x.what () << std::endl;
      error_code = EXIT_FAILURE;
    }
   return error_code;
}

// end of setup_construct_and_view.cxx
```

Program 3: The `setup_construct_and_view.cxx` program.

3.3 A virtual world with several objects of different types...

Adding more boxes is trivial, we just have to extend the "`world`" section's `internal_item.labels` list of labels with additional names and provide the corresponding `internal_item.model.XXX` and `internal_item.placement.XXX` rules. But we can also get a mix of different kind of objects as daughters of the *world* volume.

In a new "`setup_3.geom`" file, let's introduce, after the well known "`small_red_box`" section, a new geometry model that represents a long blue cylinder. This is done with the file sample 6 that shows the parameters of a new section for the cylindric object.

```
[name="long_blue_cylinder" type="geomtools::simple_shaped_model"]
#@config the list of parameters that define a long blue cylinder
shape_type        : string = "cylinder"
length_unit       : string = "cm"
r                 : real   = 10.0
z                 : real   = 85.0
material.ref      : string  = "__default__" # must be known by an external agent
visibility.hidden : boolean = 0        #  hidden flag
visibility.color  : string  = "blue"    #  display color
```

Sample 6: The *long blue cylinder* section of the "`setup_3.geom`" file.

The "`world`" section now writes like in sample 7.

```
[name="world" type="geomtools::simple_shaped_model"]
#@config the list of parameters that define the top-level large green world box
shape_type   : string = "box"    # must be supported by the model class
length_unit  : string = "m"      # note: we use CLHEP unit system
x            : real   = 1.5      # dimensions of the shape;
y            : real   = 1.5      #   must be coherent with the
z            : real   = 1.5      #   type above (box)
material.ref : string = "vacuum" # must be known by an external agent

internal_item.labels : string[3] = "box1" "box2" "cyl1"
internal_item.model.box1     : string  = "small_red_box"
internal_item.placement.box1 : string  = "0.4 0.5 0.6 (m) @ 80.0 60.0 30.0 (degree)"
internal_item.model.box2     : string  = "small_red_box"
internal_item.placement.box2 : string  = "0.2 -0.3 -0.6 (m) / z 30.0 (degree)"
internal_item.model.cyl1     : string  = "long_blue_cylinder"
internal_item.placement.cyl1 : string  = "-0.4 -0.3 0.3 (m) @ 30.0 25.0 (degree)"

visibility.hidden           : boolean = 0       # hidden flag
visibility.color            : string  = "green" # display color
visibility.daughters.hidden : boolean = 0       # only for the daughter boxes
```

Sample 7: The *world* section of the "`setup_3.geom`" file.

Now the `setup_construct_and_view.cxx` program (program source 3) displays the figure 10. Easy isn't it ?

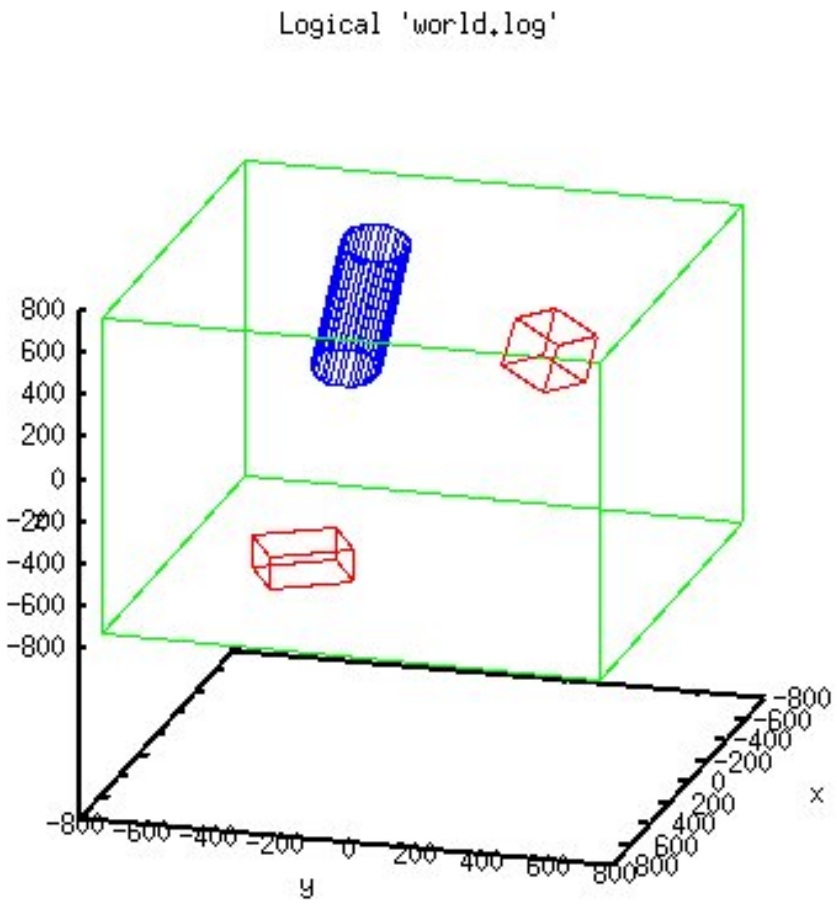

Figure 10: The Gnuplot display of the virtual world constructed from the file `setup_3.geom` built by the 3 program.

3.4 A setup with more hierarchy levels

This section describes a more complex setup. Now we will implement a world volume that contains not only a long blue cylinder but also a huge magenta cube that contains in turn three small red boxes with arbitrary placements.

File samples 8 and 9 show the associated sections of a new file "`setup_4.geom`". The display of this three-level hierarchy setup can be sen on figure 11.

There is no limit to the number of levels we can handle with such mecanism. The `internal_item.XXX` rule can be used to nest more and more daughter volumes at higher depth.

```
41 [name="huge_magenta_cube" type="geomtools::simple_shaped_model"]
42 #@config the list of parameters that define a small red box
43 shape_type        : string = "box"
44 length_unit       : string = "cm"
45 x                 : real   = 100.0
46 y                 : real   = 100.0
47 z                 : real   = 100.0
48 material.ref      : string  = "__default__"
49 internal_item.labels : string[3] = "box1" "box2" "box3"
50 internal_item.model.box1     : string  = "small_red_box"
51 internal_item.placement.box1 : string  = "0.35   0.35 -0.25 (m) / z 5.0 (degree)"
52 internal_item.model.box2     : string  = "small_red_box"
53 internal_item.placement.box2 : string  = "0.10  -0.10  0. (m)   / z 15.0 (degree)"
54 internal_item.model.box3     : string  = "small_red_box"
55 internal_item.placement.box3 : string  = "-0.30 -0.10  0.25 (m) / z 25.0 (degree)"
56 visibility.hidden_envelope   : boolean = 0
57 visibility.hidden            : boolean = 0
58 visibility.color             : string  = "magenta"
```

Sample 8: The *huge magenta cube* section of the "`setup_4.geom`" file.

```
[name="world" type="geomtools::simple_shaped_model"]
#@config the list of parameters that define the top-level large green world box
shape_type    : string = "box"    # must be supported by the model class
length_unit   : string = "m"      # note: we use CLHEP unit system
x             : real   = 1.5      # dimensions of the shape;
y             : real   = 1.5      #   must be coherent with the
z             : real   = 1.5      #   type above (box)
material.ref  : string = "vacuum" # must be known by an external agent

internal_item.labels : string[2] =  "cyl1" "cube"
internal_item.model.cyl1     : string  = "long_blue_cylinder"
internal_item.placement.cyl1 : string  = "-0.55 -0.5 0.1 (m) @ 0.0 10.0 (degree)"
internal_item.model.cube     : string  = "huge_magenta_cube"
internal_item.placement.cube : string  = "0.1 0.1 0 (m) "

visibility.hidden           : boolean = 0       # hidden flag
visibility.color            : string  = "green" # display color
visibility.daughters.hidden : boolean = 0       # only for the daughter boxes
```

Sample 9: The *world* section of the "`setup_4.geom`" file.

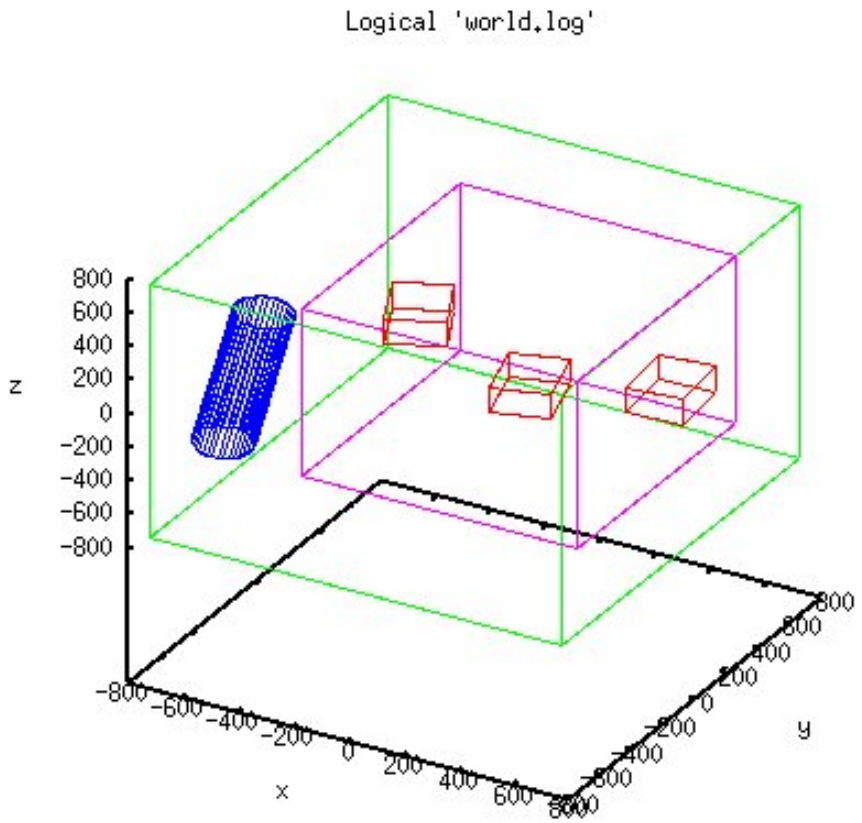

Figure 11: The Gnuplot display of the virtual world constructed from the file `setup_4.geom`.

3.5 Smart geometry models

We have seen in the previous section the basics to build a nested geometry. Up to now, we only have placed some volumes in some mother volumes thanks to the *internal item* technique. However the placement of all this daughter objects was arbitrary. We were obliged to explicitely give the positions and rotation angles of each daughter object with respect to the coordinate system of their mother volume.

In practical situations, we often face the case of the placement of objects – in a mother volume – following some symmetrical or regular patterns. This can be seen on figure 12 with several objects built from the same model and placed regularly along a given symmetry axis.

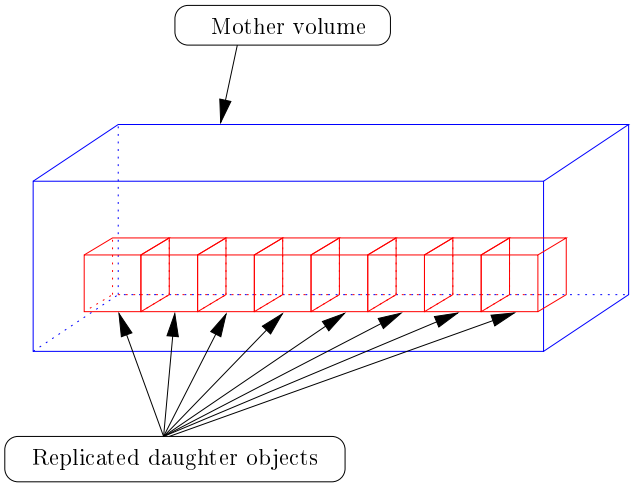

Figure 12: A mother volume with some replicated daughter volumes.

The `geomtools` library provides a few useful models that allow to automatically placed some volumes inside a mother volume using special patterns :

3.5.1 The `rotated_boxed_model` driver

This geometry model enables to create a wrapper model that operates a rotation on a given box shaped model. The rotated daughter box is included in a mother box of which dimensions can be automatically computed if special rotation angles are used.

The sample 10 shows the syntax used to define a `rotated_boxed_model` driver using the Oz axis with an arbitrary rotation angle. The rotation operates on a predefined box-shape model. As the rotation operates on the Oz axis, the height of the mother volume equals the height of the daughter volume. However the width (`x`) and depth (`y`) of the mother box must be provided by the user in such a way the mother box fully enclose the daughter volume.

Another possibility is shown with sample 11. In this case a *special* rotation angle around the $0x$ axis is used (can be 0, 90, 180 or 270 °). The mother volume's dimensions are automatically computed so there is no need to pass the transverse dimensions `y` and `z`.

```
[name="rotated_small_red_box" type="geomtools::rotated_boxed_model"]
#@config the list of parameters that define a rotated box
rotated.axis  : string = "z"       # possible axis : "x" "y" "z"
angle_unit    : string = "degree"
rotated.angle : real   = 35.0
length_unit   : string = "cm"
x             : real   = 50.0
y             : real   = 50.0
rotated.model : string = "small_red_box"
rotated.label : string = "twisty"
visibility.hidden_envelope  : boolean = 0
visibility.daughters.hidden : boolean = 0
visibility.color            : string  = "magenta"
```

Sample 10: The syntax for a *rotated box model* section.

```
[name="special_rotated_small_red_box" type="geomtools::rotated_boxed_model"]
#@config the list of parameters that define a rotated box
rotated.axis  : string = "x"
angle_unit    : string = "degree"
rotated.special_angle : string = "90" # possible values : "0" "90" "180" "270"
length_unit   : string = "mm"
rotated.model : string = "small_red_box"
rotated.label : string = "twisty"
visibility.hidden_envelope  : boolean = 0
visibility.daughters.hidden : boolean = 0
```

Sample 11: The syntax for a *rotated box model* using a special angle.

The sample 12 shows a *world* volume that contains three boxes with two of them rotated with these mechanisms. The box in magenta materializes the mother volume of the central rotated box (in red inside). The figure displays the setup.

3.5.2 The `replicated_boxed_model` driver

This geometry model enables to build a box-shaped mother volumes that contains an arbitrary numbers of adjacent (no space between them) daugther copies of one unique box shaped geometry model along one of its main axis (x, y or z).

The sample 13 shows the syntax used to define a `replicated_boxed_model` driver using the Ox axis to place 4 copies of a box shape model. The mother volume's dimensions are automatically computed.

The sample 14 replicate the above replicated model on the Oy axis. It results in a grid layout of boxex.

The sample 15 shows the *world* volume that contains this grid mother volume. The figure displays the setup.

```
[name="world" type="geomtools::simple_shaped_model"]
#@config the list of parameters that define the top-level large green world box
shape_type   : string = "box"    # must be supported by the model class
length_unit  : string = "m"      # note: we use CLHEP unit system
x            : real   = 1.5      # dimensions of the shape;
y            : real   = 1.5      #   must be coherent with the
z            : real   = 1.5      #   type above (box)
material.ref : string = "vacuum" # must be known by an external agent
internal_item.labels         : string[3] =  "box1" "box2" "box3"
internal_item.model.box1     : string    = "small_red_box"
internal_item.placement.box1 : string    = " 0.5 0.0 0.0 (m)"
internal_item.model.box2     : string    = "rotated_small_red_box"
internal_item.placement.box2 : string    = " 0.0 0.0 0.0 (m)"
internal_item.model.box3     : string    = "special_rotated_small_red_box"
internal_item.placement.box3 : string    = "-0.5 0.0 0.0 (m)"
visibility.hidden            : boolean   = 0
visibility.color             : string    = "green"
visibility.daughters.hidden  : boolean   = 0
```

Sample 12: The section of the *world* volume with rotated boxed models.

```
[name="row" type="geomtools::replicated_boxed_model"]
replicated.axis              : string = "x"
replicated.number_of_items   : integer = 4
replicated.model             : string = "small_red_box"
replicated.label             : string = "box"
visibility.hidden_envelope   : boolean = 1
visibility.daughters.hidden : boolean = 0
```

Sample 13: The syntax for a *replicated box model* section.

3.5.3 The `surrounded_boxed_model` driver

This model is used when one wants to assemble objects on some part or all of the faces of a central box shape model.

The sample 16 shows the syntax used to define a `surrounded_boxed_model` driver with three different volumes placed on three faces of a central box. The mother volume's dimensions are automatically computed. The figure displays the setup.

Some boolean options are available to force the centering of the assembly within the mother volume. There is one option per x, y or z axis: `surrounded.centered_x`, `surrounded.centered_y` and `surrounded.centered_z`. The `surrounded.bottom_model`, `surrounded.top_model`, `surrounded.back_model`, `surrounded.front_model`, `surrounded.left_mod` and `surrounded.right_model` parameters are all optional and allow to specify the name of a model to be assembled on the corresponding face of the central model.

3.5.4 The `stacked_model` driver

The `stacked_model` driver is responsible to stack several objects along an arbitrary axis (x, y or z). A mother volume is automatically computed to enclose the full set of stacked daughter volumes. Stacked objects must fulfill the *stackable object* interface;

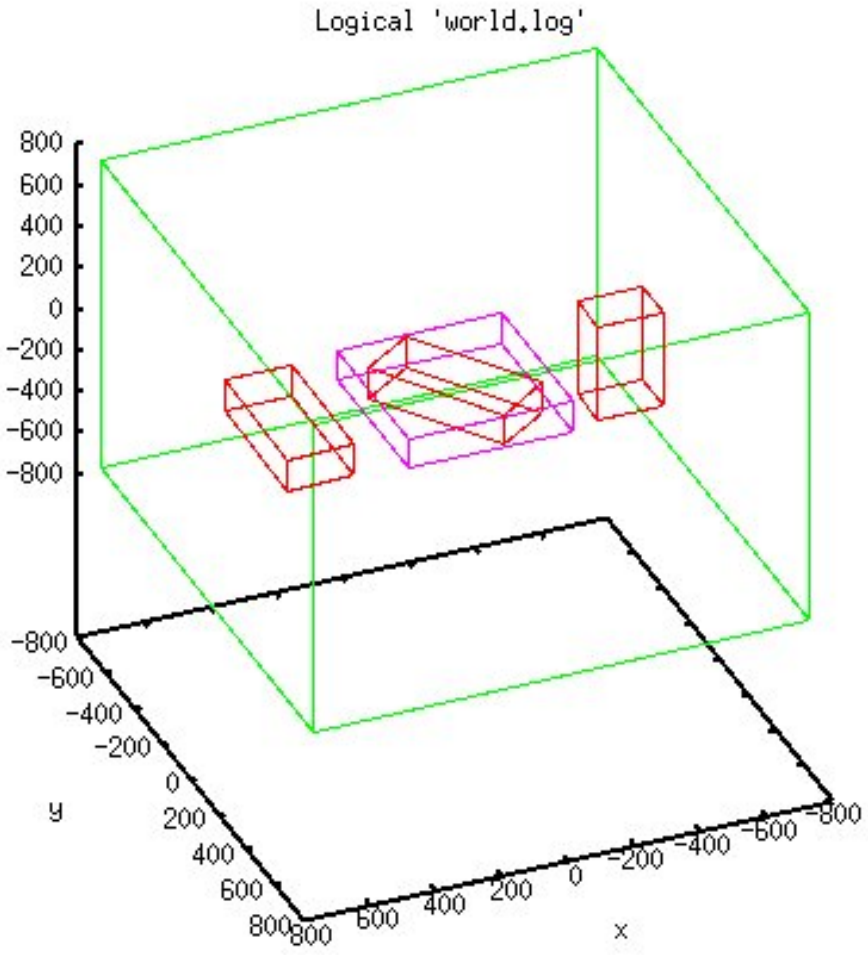

Figure 13: The Gnuplot display of the virtual world with a first non rotated box (left), a box rotated by some arbitrary angle (center) and a box rotated by some special angle (right).

boxes, cylinders, tubes as well as other basic shapes do.

The sample 17 shows the syntax used to define a `stacked_model` driver with four different volumes placed along the $0z$ axis. The mother volume's dimensions are automatically computed. The figure displays the setup.

The dimensions of the mother box can be enforced rather than automatically computed through the optional `x`, `y`, `z` real parameters (associated with the `length_unit` string property). In this case, the dimensions must ensure that the daughter volumes are fully contained in the mother box.

Some additional rules can be added to allow complex stacking as shown on figure 17. This case can occur when some volume has some concave shape along the stacking axis. A stacked neighbour convex volume with smaller dimension(s) can interpenetrate (at least partially) the concave region of the former shape. By default, *stackable objects* has some default bounds (large purple arrow on the figure) that allow only stacking algorithm to assemble neighbour volume by such bounds. With the `stacked_model`

```
[name="grid" type="geomtools::replicated_boxed_model"]
replicated.axis             : string = "y"
replicated.number_of_items  : integer = 3
replicated.model            : string = "row"
replicated.label            : string = "column"
visibility.hidden_envelope  : boolean = 1
visibility.daughters.hidden : boolean = 0
```

Sample 14: The syntax for another *replicated box model*.

```
[name="world" type="geomtools::simple_world_model"]
#@config the list of parameters that define the top-level large green world box
material.ref    : string = "vacuum"
setup.model     : string = "grid"
length_unit     : string = "m"
setup.x         : real   = 0.0
setup.y         : real   = 0.0
setup.z         : real   = 0.0
angle_unit      : string = "degree"
setup.phi       : real   = 30.0
setup.theta     : real   = 25.0
setup.delta     : real   = 20.0
world.x         : real   = 1.5
world.y         : real   = 1.5
world.z         : real   = 1.5
visibility.hidden           : boolean = 0
visibility.color            : string  = "green"
visibility.daughters.hidden : boolean = 0
```

Sample 15: The section of the *world* volume with replicated boxed models.

driver, it is possible to redefine new bounds for a volume along the stacking axis and thus allow some penetrating placement of other volumes. For each stacked model, it is possible to use the `stacked.limit_min_X` and `stacked.limit_max_X` real rules. As seen of figure 18, these bounds can be set arbitrarily and the choice depends on what layout the user want to achieve. The sample 18 shows a typical syntax for these options and figure shows the setup.

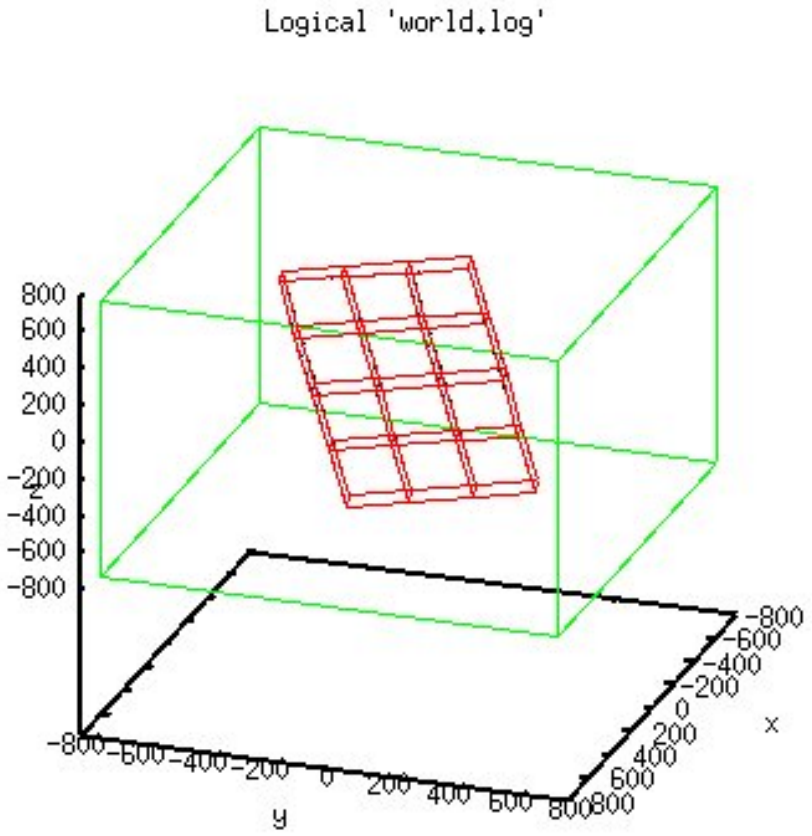

Figure 14: The Gnuplot display of the virtual world with a grid made from a two levels replicated model.

```
[name="surrounded_box" type="geomtools::surrounded_boxed_model"]
material.ref              : string  = "__default__"
surrounded.model          : string  = "huge_red_box"
surrounded.label          : string  = "the_central_box"
surrounded.centered_x     : boolean = 0
surrounded.centered_y     : boolean = 0
surrounded.centered_z     : boolean = 1
surrounded.bottom_model   : string  = "blue_cylinder"
surrounded.top_model      : string  = "magenta_tube"
surrounded.front_model    : string  = "small_orange_box"
visibility.hidden           : boolean   = 0
visibility.daughters.hidden : boolean = 0
```

Sample 16: The syntax for a *surrounded box model* section.

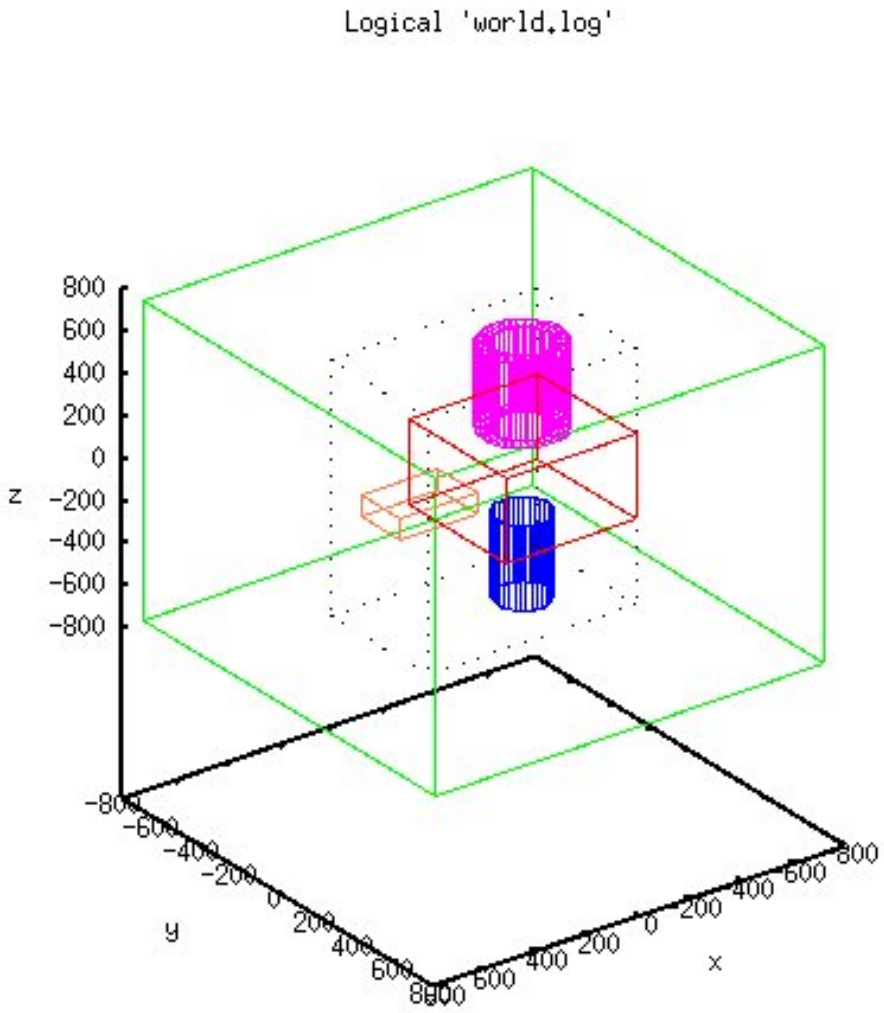

Figure 15: The Gnuplot display of the virtual world with a box surrounded by other volumes and wrapped in its mother envelope.

```
[name="stacked_box" type="geomtools::stacked_model"]
material.ref : string = "__default__"

stacked.axis            : string = "z"
stacked.number_of_items : integer = 4
stacked.model_0     : string  = "blue_cylinder"
stacked.label_0     : string  = "stacked_0"
stacked.model_1     : string  = "huge_red_box"
stacked.label_1     : string  = "stacked_1"
stacked.model_2     : string  = "magenta_tube"
stacked.label_2     : string  = "stacked_2"
stacked.model_3     : string  = "orange_cylinder"
```

Sample 17: The syntax for a *stacked model* section.

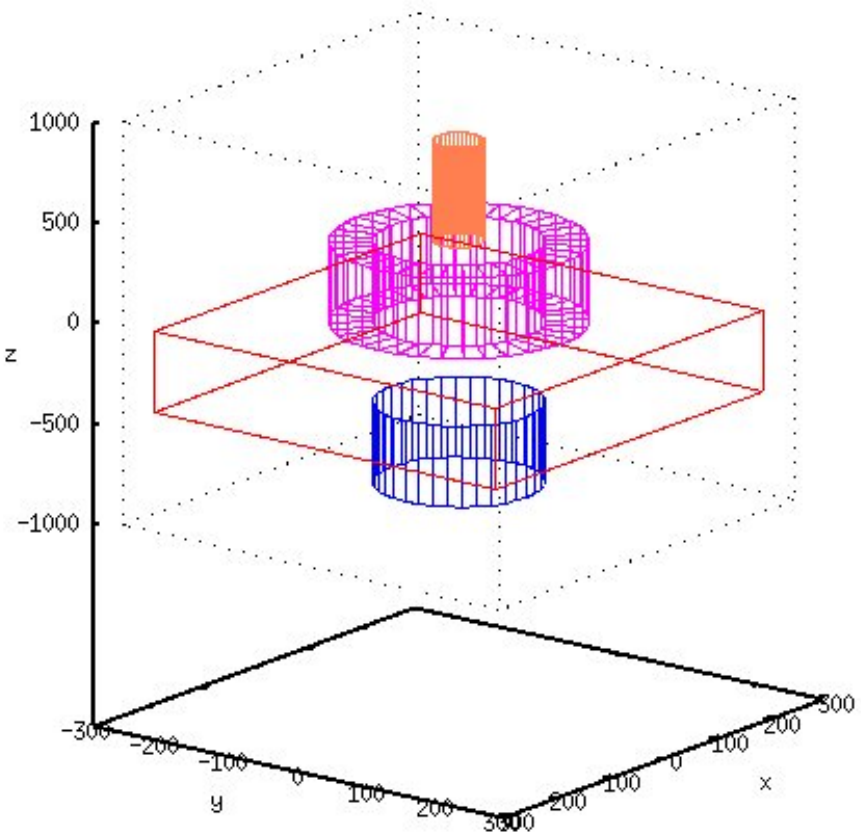

Figure 16: The Gnuplot display of the virtual world with a stack of four different volumes along the z axis and wrapped in its mother envelope.

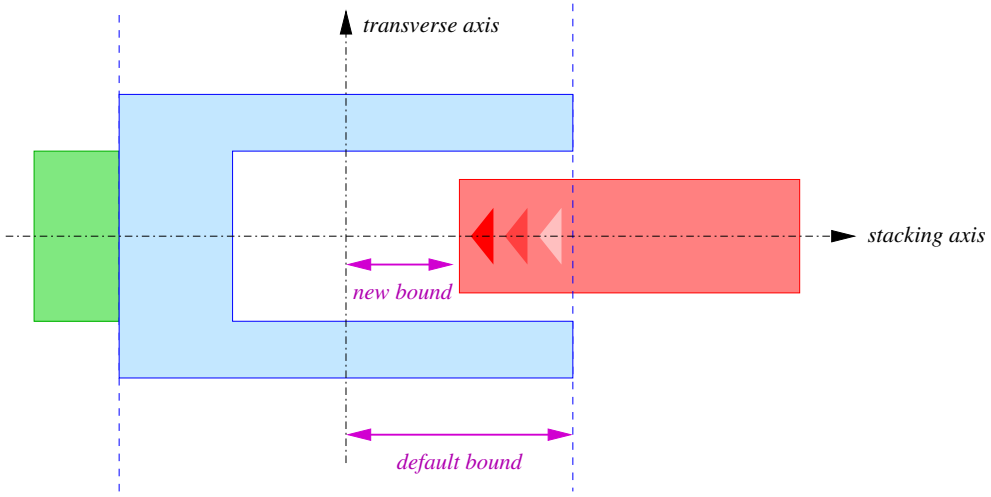

Figure 17: Special stacking with interpenetrating volumes.

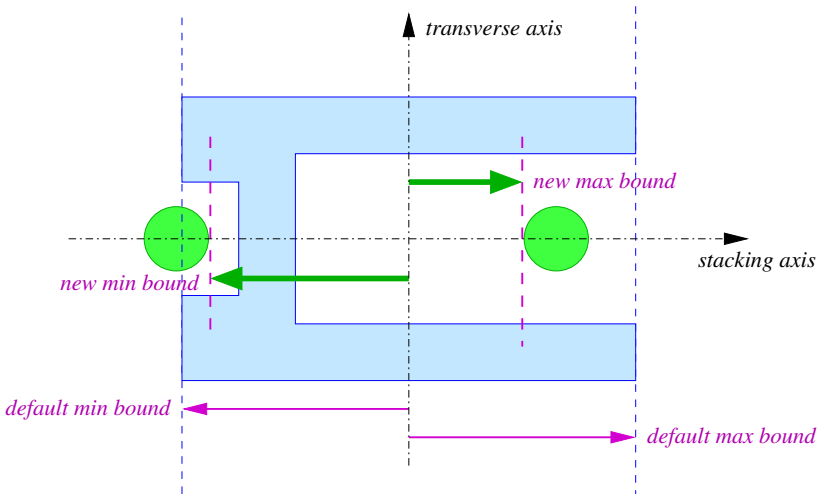

Figure 18: Setting new stacking bounds to a volume. Purple arrows show the new bounds applied to the volume. The green spheres indicates the extreme positions allowed by these stacking informations.

```
[name="stacked_box" type="geomtools::stacked_model"]
material.ref : string = "__default__"

length_unit         : string = "cm"
stacked.axis            : string = "z"
stacked.number_of_items : integer = 4
stacked.model_0     : string  = "blue_cylinder"
stacked.label_0     : string  = "stacked_0"
stacked.model_1     : string  = "huge_red_box"
stacked.label_1     : string  = "stacked_1"
stacked.model_2     : string  = "magenta_tube"
stacked.label_2     : string  = "stacked_2"
stacked.limit_max_2 : real    = -15.        # a negative value is possible
stacked.model_3     : string  = "orange_cylinder"
stacked.label_3     : string  = "stacked_3"

visibility.hidden : boolean   = 0
visibility.color  : string    = "grey"
```

Sample 18: The syntax for a *stacked model* section with penetrating volumes.

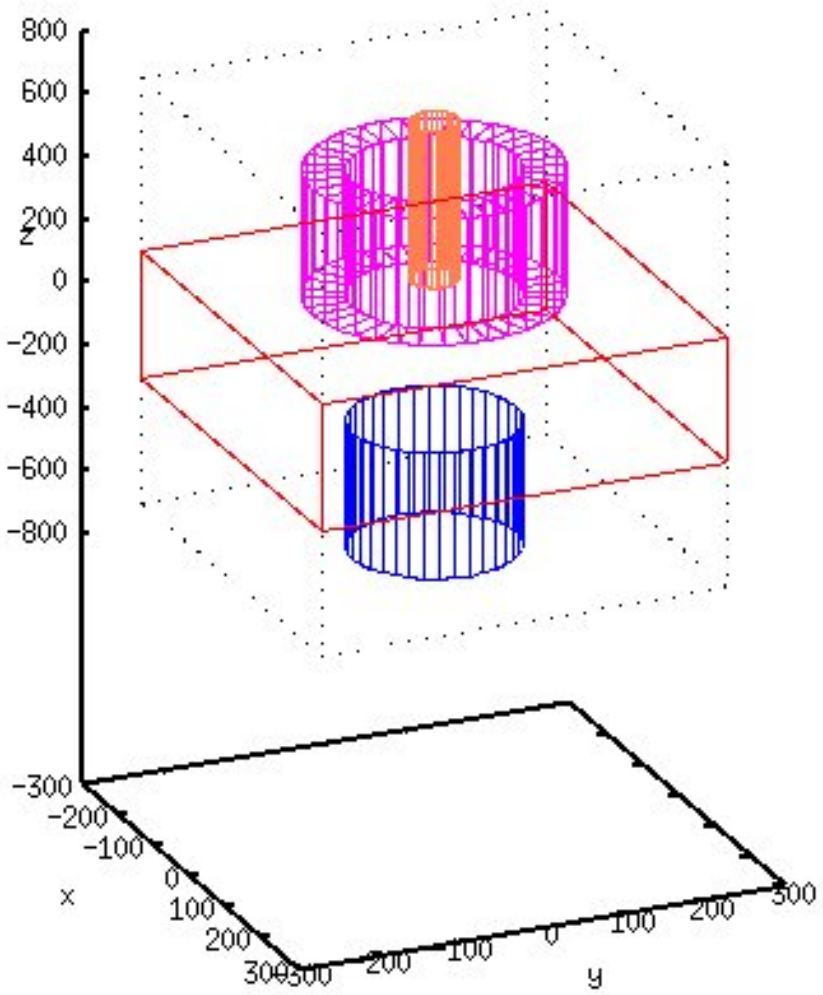

Figure 19: The Gnuplot display of stacked volumes with penetrating placement.

4 Conclusion

To be done.