

GDML USER'S GUIDE

Version 1.3

Contents

1. Introduction.....	4
2. Getting Started.....	
2.1 Installation.....	5
2.2 Running simple examples.....	5
3. GDML Schema.....	
3.1 General Structure.....	6
3.1.1 Splitting GDML file using ENTITY.....	8
3.1.2 Multiple GDML Files.....	8
3.2 Definitions.....	9
3.2.1 Variables.....	9
3.2.2 Matrices.....	10
3.3 Materials.....	11
3.3.1 Isotopes.....	11
3.3.2 Elements.....	11
3.3.3 Materials.....	12
3.4 Solids.....	13
3.4.1 Box.....	13
3.4.2 Cone Segment.....	13
3.4.3 Ellipsoid.....	13
3.4.4 Elliptical Tube.....	14
3.4.5 Orb.....	14
3.4.6 Paraboloid.....	15
3.4.7 Parallelepiped.....	15
3.4.8 Polycone.....	15
3.4.9 Polyhedra.....	16
3.4.10 Sphere.....	17
3.4.11 Torus Segment.....	17
3.4.12 Trapezoid – x & y varying along z.....	18
3.4.13 General Trapezoid.....	18
3.4.14 Tube with Hyperbolic Profile.....	19
3.4.15 Cut Tube.....	19
3.4.16 Tube Segment.....	20
3.4.17 Twisted Box.....	20
3.4.18 Twisted Trapezoid.....	20
3.4.19 Twisted General Trapezoid.....	21
3.4.20 Twisted Tube Segment.....	21
3.4.21 Xtru.....	22
3.4.22 Arb8.....	23
3.4.23 Tessellated Solid.....	24
3.4.24 Tetrahedra.....	25
3.4.25 Using Loops for solids.....	26
3.5 Structure.....	26
3.5.1 Loops.....	27
3.5.2 GDML Modules.....	30
3.5.3 Volume Auxiliary Information.....	32
3.6 Setup.....	34
3.7 GDML Example File.....	35

4. CPPGDML.....	36
4.1 GDML Reading.....	36
4.1.1 Importing GDML files into Geant4 Geometry.....	36
4.2 GDML Writing.....	37
4.2.1 Exporting Geant4 Geometry.....	37
4.2.2 Exporting ST-Viewer CAD Geometry.....	38
5. User Extensions to GDML.....	40

1. Introduction

The Geometry Description Markup Language (GDML) is a specialized XML-based language designed as an application-independent persistent format for describing the geometries of detectors associated with physics measurements. It serves to implement "geometry trees" which correspond to the hierarchy of volumes a detector geometry can be composed of, and to allow to identify the position of individual solids, as well as to describe the materials they are made of. Being pure XML, GDML can be universally used, and in particular it can be considered as the format for interchanging geometries among different applications.

GDML consists of two elements, an XML definition part containing the set of rules and the list of the legal elements to be used in constructing any GDML document, and the GDML generating and processing code. The structure of the GDML document is defined through a set of XML Schema Definition (XSD) files which we call the GDMLSchema. Any GDML geometry file must be valid with respect to the GDMLSchema. The GDML file itself, can be either written by hand (in case GDML is used as the primary geometry source) or generated automatically (in case GDML is used as an exchange or persistency format) out of the application specific "in-memory" geometry tree using one of the GDML "writers" called by the user application. The GDML reader is responsible for parsing the GDML file and creating the in-memory representation of the geometry tree specific for the user application.

This manual will focus on the Geant4 binding for GDML. The ROOT binding for GDML is integrated within the ROOT framework and therefore the user should refer to the general ROOT manual for any information on importing and/or exporting GDML files to/from the ROOT geometry model. The description of the GDMLSchema in Section 3, however, is application-independent and therefore is relevant for both the Geant4 as well as the ROOT users.

2. Getting started

2.1 Installation

In order to install the Geant4 binding for GDML one needs to have:

- Geant4 installed (together with the appropriate CLHEP) with all the header files copied to one directory (an option during the Geant4 installation procedure)
- XercesC parser installed (presently GDML uses XercesC 2.7.0)

The GDML source code can be checked out from the GDML2 CVS repository using the anonymous access mode or downloaded in the form of the tarfile from the GDML web page (<http://gdml.web.cern.ch/GDML/gdml.html>).

Before the libraries can be built, you can either:

1. Run `GDMLSetup` from the `CPPGDML` directory. This will ask you various questions to set dependency paths, and enable/disable options, it will then configure the system with the parameters specified using `configure` script also in the `CPPGDML` directory. The setup will then run 'make' and compile and install the system. After running this script you will have a fully installed GDML system.
2. Manually configure, using your own script or from the shell. Please see `./configure - help` for the configure options. Once you have configured you can call `make` etc.

After taking option 1, you will have inside the `CPPGDML` directory, 2 scripts (`MyConfigure.*`). One is an `sh` shell script, the other is a `csh` shell script for your convenience. This contains all the paths and options specified during the setup procedure. You may manually edit the script you wish to use if you need to edit parameters you specified during setup. Alternatively you can run the entire setup procedure again. Please refer to the information on the GDML web page as well as the `README` file in the `CPPGDML` directory for more information on the configuration procedure.

The building of the libraries is done by `make` command issued in the `CPPGDML` directory.

2.2 Running simple examples

The Geant4 binding for GDML comes with two simple examples which demonstrate the writing out (`g4N02gdml`) and the reading in (`g4gogdml`) of GDML files. The directories containing these examples can be found in `CPPGDML/Examples` directory. The examples are built automatically during the general GDML building procedure. In case of using dynamic libraries, one needs to set the `LD_LIBRARY_PATH` variable to point to all the necessary libraries (CLHEP, Geant4, XercesC and CPPGDML) before running the examples. Please refer to the examples `README` files for more information.

3. GDMLSchema

3.1 General structure

The GDML schema is a set of XSD files which define the structure of the GDML document and its legal elements. The general structure of the GDML file can be seen below (with some parts replaced by ‘...’ for brevity).

```
<?xml version="1.0" encoding="UTF-8"?>
<gdml xsi:noNamespaceSchemaLocation="GDMLSchema/gdml.xsd">
  <define>
    ...
    <position name="TrackerinWorldpos" unit="mm" x="0" y="0" z="100"/>
  </define>
  <materials>
    ...
    <element name="Nitrogen" formula="N" Z="7.">
      <atom value="14.01"/>
    </element>
    <material formula=" " name="Air" >
      <D value="1.290" unit="mg/cm3"/>
      <fraction n="0.7" ref="Nitrogen" />
      <fraction n="0.3" ref="Oxygen" />
    </material>
  </materials>
  <solids>
    ...
    <box lunit="mm" name="Tracker" x="50" y="50" z="50"/>
  </solids>
  <structure>
    ...
    <volume name="World" >
      <materialref ref="Air" />
      <solidref ref="world" />
      <physvol>
        <volumeref ref="Tracker" />
        <positionref ref="TrackerinWorldpos"/>
        <rotationref ref="TrackerinWorldrot"/>
      </physvol>
    </volume>
  </structure>
  <setup name="Default" version="1.0" >
    <world ref="World" />
  </setup>
</gdml>
```

One can distinguish there five parts, each holding specific type of data.

The `<define> ... </define>` block contains numerical values of different constants, positions, rotations and scales that will be used later on in the geometry construction.

The `<materials> ...</materials>` block contains definitions of all the materials used in the given geometry. The supported forms are simple materials which are made from one element as well as mixtures. Mixtures can be composed on the basis of fraction of mass or atom count.

The `<solids> ... </solids>` block is the collection of all solid definitions which are used in the given geometry description. The presently supported solids are described in Section 0.

Composite solids made using boolean operation (union, subtraction, intersection) are also supported.

The `<structure> ... </structure>` block contains the actual implementation of the geometry tree together with the assignment of solids and materials. The hierarchy of volumes is defined by specifying the daughter volumes (physvol) positioned inside a volume. Constructions like assembly volumes, reflections, replicas and divisions are possible. A more detailed discussion of the 'structure' block can be found in Section 3.5.

Finally, the `<setup> ... </setup>` block serves to specify the top volume of the geometry tree. It is possible to define several ``setups" within one file, allowing to test different subparts (or different configurations included in the same file) of the geometry tree without changing the GDML file.

3.1.1 Splitting GDML file using ENTITY

Any GDML file can be split into several files which are combined together during the parsing by using the ENTITY statement. The files included in such a way are not standalone GDML files, they simply contain parts of the GDML description and in general are meaningful only when put all together.

In order to use such a mechanism, one has to first declare the given ENTITY (just after the <xml ... > tag)

```
<!DOCTYPE gdml [  
<!ENTITY materials SYSTEM "materials.xml">  
>
```

where `materials.xml` is the name of the file to be included and `materials` is the ENTITY name that we have assigned to it. The entity can be then used anywhere inside the GDML description.

```
<gdml ....>  
....  
&materials;  
....  
</gdml>
```

The `&materials` line is then replaced during parsing by the contents of `materials.xml` file.

3.1.2 Multiple GDML files

It is also possible to define the geometry in the form of several standalone GDML files, each one representing a module of the setup we want to describe. Each GDML file would contain only the materials, the solids and the volumes the module is made of. For example, if one wants to describe a detector, instead of creating a single `detector.gdml` file with thousands of volumes and materials, one could create smaller modules called `tracker.gdml`, `calorimeter.gdml`, `beampipe.gdml`, etc, and a small main `detector.gdml` file combining those modules. This approach makes it easier to read and write the geometries and to apply changes. Explanations and details on how GDML modules are defined and put together are given in the Section 3.5.

3.2 Definitions

3.2.1 Variables

Variables are defined in the beginning of the GDML file (in the define section). Once defined, they can be used anywhere inside the file. Their value will be evaluated each time they are used and it can change if they are used as loop variables (see section 3.5.1). A variable can be defined as in the following example:

```
<variable name="x" value="6"/>
```

This simple code defines a variable with an initial value of 6. Of course previously defined variables may be used in variable definitions, as in the following example:

```
<variable name="x" value="6"/>  
<variable name="y" value="x/2"/>
```

In this case variable y is set with an initial value of 3. You can now use your variable wherever you like, for example in a box definition:

```
<box name="my_box" x="x" y="y" z="x+y"/>
```

3.2.2 Matrices

A matrix definition allows you to store values in matrix format. The definition of a matrix is very simple: `<matrix name="name" coldim="number of columns" values="space-separated values" />`. A simple example follows:

```
<matrix name="m" coldim="3" values="0.4 9 126 8.5 7 21 34.6 7 9" />
```

Or you could also write it like this (visually clearer):

```
<matrix name="m" coldim="3" values=" 0.4 9 126
                                8.5 7 21
                                34.6 7 9" />
```

The elements of a matrix can be accessed anywhere after the definition of the matrix itself, just variables. To access them you have to use the following syntax:

```
matrix_name[row_number, column_number]
```

for example:

```
m[3,1]
```

will be evaluated to 34.6. You can also put variables inside the square brackets, like this:

```
m[y,y]
```

So if $y=3$ this will be evaluated to 9. Just be sure that the values of the variables don't exceed the matrix dimensions, otherwise you will get an error when processing the file. A matrix with `coldim="1"` is a vector and its elements may be accessed using only one index as in the following example:

```
<matrix name="m" coldim="1" values="4 9 12 6 7 9" />
```

So,

```
m[4]
```

will be evaluated to 6. Matrices can of course be used inside loops (there is where they are actually most useful). We will see how to do this in section 3.5.1.

3.3 Materials

3.3.1 Isotopes

Isotopes can be defined using a very simple construct. `<isotope name=its_name Z=atomic_number N=num_of_nucleons>` The following shows an example:

```
<isotope name="U235" Z="92" N="235">
  <atom type="A" value="235.01">
</isotope>
<isotope name="U238" Z="92" N="238">
  <atom type="A" value="235.03">
</isotope>
```

3.3.2 Elements

Elements can be defined using this very self explanatory construct:

```
<element Z="8" formula="O" name="Oxygen" >
  <atom value="16" />
</element>
```

Instead of an atomic number being used on a child node of the element declaration, a fraction of an isotope can be used using `<fraction ref=isotope_name n=fraction />`. This is implemented as the following shows:

```
<element name="enriched_uranium" >
  <fraction ref="U235" n="0.9" />
  <fraction ref="U238" n="0.1" />
</element>
```

3.3.3 Materials

Materials can be defined in three ways. The material can be defined directly from an element, created from previously defined elements or materials on number of atoms, and finally created as a fractional mixture of previously defined elements or materials.

```
<material name="Al" Z="13.0" >  
  <D value="2.70" />  
  <atom value="26.98" />  
</material>
```

```
<material name="Water" formula="H2O">  
  <D value="1.0" />  
  <composite n="2" ref="Hydrogen" />  
  <composite n="1" ref="Oxygen" />  
</material>
```

```
<material formula="air" name="Air" >  
  <D value="0.0012899999999999999" />  
  <fraction n="0.7" ref="Nitrogen" />  
  <fraction n="0.3" ref="Oxygen" />  
</material>
```

3.4 Solids

3.4.1 Box

The GDML box is formed using the three standard dimensions: x , y and z . The box therefore has 4 attributes, the other being name which is standard for any `SolidType` element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a box made from the above dimensions:

```
<box name = "thebox" x= "30" y= "30" z= "30" lunit= "mm" />
```

3.4.2 Cone Segment

The GDML Cone Segment is formed using 7 dimensions.

<i>rmin1</i>	inner radius at base of cone
<i>rmax1</i>	outer radius at base of cone
<i>rmin2</i>	inner radius at top of cone
<i>rmax2</i>	outer radius at top of cone
<i>z</i>	height of cone segment
<i>startphi</i>	start angle of the segment
<i>deltaphi</i>	angle of the segment

The cone segment therefore has 8 attributes, the other being name which is standard for any `SolidType` element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a cone segment made from the above dimensions:

```
<cone name = "thecone" rmin1="10" rmax1="15" rmin2="15" rmax2="25" z="30" startphi="1" deltaphi="4" aunit="rad" lunit= "mm" />
```

3.4.3 Ellipsoid

The GDML Ellipsoid is formed using 3 dimensions.

<i>ax</i>	x semi axis
<i>by</i>	y semi axis
<i>cz</i>	z semi axis

There are 2 optional dimensions that can cut the ellipsoid. *zcut1* will remove the specified value from the lower z axis effectively cutting a part of the bottom of the shape off. *zcut2* performs the same action but acts on the upper part of the z axis.

The ellipsoid therefore has 4 attributes, the other being name which is standard for any `SolidType` element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a ellipsoid made from the above dimensions:

```
<ellipsoid name = "theellipsoid" ax="10" by="15" cz="20" zcut2="4" lunit= "mm"/>
```

3.4.4 Elliptical Tube

The GDML Elliptical Tube is formed using 3 dimensions:

ax x semi axis
by y semi axis
cz z semi axis

The elliptical tube therefore has 4 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of an elliptical tube made from the above dimensions:

```
<eltube name = "theeltube" ax="10" by="15" cz="20" lunit= "mm" />
```

3.4.5 Orb

The GDML Orb is formed using only 1 dimension:

r radius

The orb is a simplified GDML Sphere with only the radius specified. The orb therefore has 2 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of an orb made from the above dimensions:

```
<orb name = "theorb" r="10" lunit= "mm"/>
```

3.4.6 Paraboloid

NB: Currently not supported by Geant4

The GDML Paraboloid is formed using 3 dimensions:

rlo radius at -z
rhi radius at +z
dz z length

The paraboloid therefore has 4 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of an paraboloid made from the above dimensions:

```
<paraboloid name = "theparab" rlo="10" rhi="15" dz="20" aunit="rad" lunit= "mm" />
```

3.4.7 Parallelepiped

The GDML Parallelepiped is formed using 6 dimensions:

x length of x
y length of y
z length of z
alpha angle between x and z planes
theta polar angle of the line joining the centres of the faces at -z & +z in z
phi azimuthal angle of the line joining the centres of faces at -z & +z in z

The Parallelepiped therefore has 7 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a parallelepiped made from the above dimensions:

```
<para name = "thepara" x="10" y="10" z="10" alpha="1" theta="1" phi="1" aunit="rad" lunit= "mm"/>
```

3.4.8 Polycone

The GDML Polycone is described using the following information:

startphi start angle of the segment – if not given 0.0 is defaulted
deltaphi angle of the segment
zplane see below

The zplane element is required to be used at least once. You can use it as many times as you like. The zplane is formed using the 3 attributes shown above:

rmin inner radius of cone at this point – if not given 0.0 is defaulted.
rmax outer radius of cone at this point
z z coordinate of the plane

The Polycone therefore has a minimum of 7 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a polycone made from the above dimensions:

```
<polycone name = "thepolycone" startphi="1" deltaphi="4" aunit="rad" lunit= "mm">  
  <zplane rmin="1" rmax="9" z="10" />  
  <zplane rmin="3" rmax="5" z="12" />  
</polycone>
```

3.4.9 Polyhedra

The GDML Polyhedra is described using the following information:

startphi start angle of the segment
deltaphi angle of the segment
numsides number of sides
zplane *see below*

The zplane element is required to be used at least once. You can use it as many times as you like. The zplane is formed using the 3 attributes shown above:

rmin radius of inner tangent circle
rmax radius of outer tangent circle
z z coordinate of the plane

The Polyhedra therefore has a minimum of 6 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a polyhedra made from the above dimensions:

```
<polyhedra    name="thepolyhedra"    startphi="0"    deltaphi="360"    numsides="8"  
  aunit="degree" lunit="mm">  
  <zplane rmin="4" rmax="8" z="0" />  
  <zplane rmin="4" rmax="8" z="10" />  
</polyhedra>
```


3.4.10 Sphere

The GDML Sphere is formed using 6 dimensions:

<i>rmin</i>	inner radius – if not given 0.0 is defaulted
<i>rmax</i>	outer radius
<i>startphi</i>	starting angle of the segment – if not given 0.0 is defaulted
<i>deltaphi</i>	delta angle of the segment
<i>starttheta</i>	starting angle of the segment – if not given 0.0 is defaulted
<i>deltatheta</i>	delta angle of the segment

The Sphere therefore has a minimum of 7 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a sphere made from the above dimensions:

```
<sphere name = "thesphere" rmin="1" rmax="4" deltaphi="1" deltatheta="1" aunit="rad"
lunit= "mm"/>
```

3.4.11 Torus Segment

The GDML Torus Segment is formed using 5 dimensions:

<i>rmin</i>	inside radius of segment
<i>rmax</i>	outside radius of segment
<i>rtor</i>	swept radius of torus
<i>startphi</i>	starting phi position angle of segment
<i>deltaphi</i>	delta angle of segment

The Torus segment therefore has a minimum of 6 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a torus segment made from the above dimensions:

```
<torus name = "thetorus" rmin="1" rmax="4" rtor="2" deltaphi="3" startphi="1" aunit="rad"
lunit= "mm"/>
```

3.4.12 Trapezoid – x & y varying along z

The GDML Trapezoid is formed using 5 dimensions:

<i>x1</i>	x length at -z
<i>x2</i>	x length at +z
<i>y1</i>	y length at -z
<i>y2</i>	y length at +z
<i>z</i>	z length

The Trapezoid therefore has a minimum of 6 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a Trapezoid made from the above dimensions:

```
<trd name = "thetrd" x1="9" x2="8" y1="6" y2="5" z="10" lunit= "mm"/>
```

3.4.13 General Trapezoid

The GDML General Trapezoid is formed using 11 dimensions:

<i>z</i>	length along z axis
<i>theta</i>	polar angle to faces joining at -/+z
<i>phi</i>	azimuthal angle of line joining centre of -z face to centre of +z face
<i>y1</i>	length along y at the face -z
<i>x1</i>	length along x at side y = -y1 of the face at -z
<i>x2</i>	length along x at side y = +y1 of the face at -z
<i>alpha1</i>	angle with respect to the y axis from the centre of side at y = -y1 to centre of y = +y1 of the face at -z
<i>y2</i>	length along y at the face +z
<i>x3</i>	length along x at side y = -y1 of the face at +z
<i>x4</i>	length along x at side y = +y1 of the face at +z
<i>alpha2</i>	angle with respect to the y axis from the centre of side at y = -y2 to centre of y = +y2 of the face at +z

The General Trapezoid therefore has a minimum of 12 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a General Trapezoid made from the above dimensions:

```
<trap name = "thetrap" z="10" theta="1" phi="2" y1="15" x1="10" x2="10" alpha1="1" y2="15" x3="10" x4="10" alpha2="1" aunit="rad" lunit= "mm"/>
```

3.4.14 Tube with Hyperbolic Profile

The GDML Tube with Hyperbolic Profile is formed using 5 dimensions:

<i>rmin</i>	inside radius of tube
<i>rmax</i>	outside radius of tube
<i>inst</i>	inner stereo
<i>outst</i>	outer stereo
<i>z</i>	z length

The Hyperbolic Tube therefore has a minimum of 6 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a hyperbolic tube made from the above dimensions:

```
<hype name = "thehype" rmin="1" rmax="2" z="20" inst="3" outst="4" lunit= "mm"/>
```

3.4.15 Cut Tube

NB: Currently not supported by Geant4

The GDML Cut Tube is formed using 11 dimensions:

<i>z</i>	length along z axis
<i>rmin</i>	inner radius – if not given 0.0 is defaulted
<i>rmax</i>	outer radius
<i>startphi</i>	starting phi angle of segment – if not given 0.0 is defaulted
<i>deltaphi</i>	delta phi of angle
<i>lowX</i>	normal at lower z plane
<i>lowY</i>	normal at lower z plane
<i>lowZ</i>	normal at lower z plane
<i>highX</i>	normal at upper z plane
<i>highY</i>	normal at upper z plane
<i>highZ</i>	normal at upper z plane

The cut Tube therefore has a minimum of 12 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a cut Tube made from the above dimensions:

```
<cutTube name = "thecuttube" z="20" rmin="1" rmax="5" startphi="1" deltaphi="4" lowX="15" lowY="15" lowZ="15" highX="10" highY="10" highZ="10" aunit="rad" lunit= "mm"/>
```

3.4.16 Tube Segment

The GDML Tube Segment is formed using 5 dimensions:

<i>rmin</i>	inside radius of segment - if not given 0.0 is defaulted
<i>rmax</i>	outside radius of segment
<i>z</i>	z length of tube segment
<i>startphi</i>	starting phi position angle of segment - if not given 0.0 is defaulted
<i>deltaphi</i>	delta angle of segment

The Tube segment therefore has a minimum of 6 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a tube segment made from the above dimensions:

```
<tube name = "thetube" rmin="1" rmax="4" z="20" deltaphi="3" startphi="1"
aunit="rad" lunit="mm"/>
```

3.4.17 Twisted Box

The GDML Twisted box is formed using the three standard dimensions: *x*, *y* and *z* and one added dimension: *PhiTwist*. The box therefore has 5 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a box made from the above dimensions:

```
<twistedbox name = "twistbox" PhiTwist= "1" x= "30" y= "30" z= "30" aunit="rad"
lunit="mm"/>
```

3.4.18 Twisted Trapezoid

The GDML Twisted Trapezoid is formed using 6 dimensions:

<i>PhiTwist</i>	Twist Angle
<i>x1</i>	x length at -z
<i>x2</i>	x length at +z
<i>y1</i>	y length at -z
<i>y2</i>	y length at +z
<i>z</i>	z length

The Twisted Trapezoid therefore has a minimum of 7 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a Twisted Trapezoid made from the above dimensions:

```
<twistedtrd name = "twisttrd" PhiTwist="1" x1="9" x2="8" y1="6" y2="5" z="10" aunit="rad"
lunit= "mm"/>
```

3.4.19 Twisted General Trapezoid

The GDML Twisted General Trapezoid is formed using 11 dimensions:

<i>PhiTwist</i>	twist angle
<i>z</i>	length along z axis
<i>Theta</i>	polar angle to faces joining at -/+z
<i>Phi</i>	azimuthal angle of line joining centre of -z face to centre of +z face
<i>y1</i>	length along y at the face -z
<i>x1</i>	length along x at side y = -y1 of the face at -z
<i>x2</i>	length along x at side y = +y1 of the face at -z
<i>y2</i>	length along y at the face +z
<i>x3</i>	length along x at side y = -y1 of the face at +z
<i>x4</i>	length along x at side y = +y1 of the face at +z
<i>Alph</i>	angle with respect to the y axis from the centre of the side

The Twisted General Trapezoid therefore has a minimum of 12 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a General Twisted Trapezoid made from the above dimensions:

```
<twistedtrap name = "twisttrap" PhiTwist="1" z="10" Theta="1" Phi="2" y1="15"
x1="10" x2="10" y2="15" x3="10" x4="10" Alph="1" aunit="rad" lunit= "mm"/>
```

3.4.20 Twisted Tube Segment

The GDML Twisted Tube Segment is formed using 5 dimensions:

<i>endinnerrad</i>	inside radius of segment
<i>endouterrad</i>	outside radius of segment
<i>zlen</i>	z length of tube segment
<i>twistedangle</i>	twist angle
<i>phi</i>	phi angle of segment

The Twisted Tube segment therefore has a minimum of 6 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of a twisted tube segment made from the above dimensions:

```
<twistedtubs name = "twisttube" endinnerrad="1" endouterrad="4" zlen="20" phi="3"
twistedangle="1" aunit="rad" lunit="mm"/>
```

3.4.21 Xtru

The GDML Xtru is formed using 2 attributes:

<i>twoDimVertex</i>	vertices of an unbound blueprint polygon
<i>section</i>	z sections

The *twoDimVertex* element is required to be used at least 3 times and the *section* element is required to be used at least twice. You can use both as many times as you like. The *twoDimVertex* is formed using the 2 attributes shown below:

<i>x</i>	x coordinate of the vertex
<i>y</i>	y coordinate of the vertex

The *section* is formed using the 5 attributes shown below:

<i>zOrder</i>	index of the section, must be between 0 and n-1 where n is the number of sections
<i>zPosition</i>	distance from the plane z=0
<i>xOffset</i>	x offset from centre point of original plane
<i>yOffset</i>	y offset from centre point of original plane
<i>scalingFactor</i>	proportion to original blueprint

The Xtru therefore has a minimum of 17 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of an Xtru made from the above dimensions:

```
<xtru name = "theXtru" lunit= "mm" >
  <twoDimVertex x="3" y="9" />
  <twoDimVertex x="1" y="5" />
  <twoDimVertex x="2" y="4" />
  <section zOrder="1" zPosition="2" xOffset="5" yOffset="3" scalingFactor="3" />
  <section zOrder="2" zPosition="5" xOffset="3" yOffset="5" scalingFactor="1" />
</xtru>
```

3.4.22 Arb8

NB: Currently not supported by Geant4

The GDML Arb8 (arbitrary trapezoid with less than 8 vertices) is formed using 17 dimensions. It's made up of two quadrilaterals sitting on parallel planes. V1 to V4 define the vertices for the base quadrilateral situated at -dz, V5 to V8 define the vertices for the other quadrilateral situated at +dz.

v1x	vertex 1 x position
v1y	vertex 1 y position
v2x	vertex 2 x position
v2y	vertex 2 y position
v3x	vertex 3 x position
v3y	vertex 3 y position
v4x	vertex 4 x position
v4y	vertex 4 y position

v5x	vertex 5 x position
v5y	vertex 5 y position
v6x	vertex 6 x position
v6y	vertex 6 y position
v7x	vertex 7 x position
v7y	vertex 7 y position
v8x	vertex 8 x position
v8y	vertex 8 y position
dz	half z length

The Arb8 therefore has a minimum of 18 attributes, the other being name which is standard for any SolidType element and corresponds to the name the solid can be referenced by.

The following example illustrates the construction of an Arb8 made from the above dimensions:

```
<arb8 name = "thearb8" v1x="1" v1y="1" v2x="2" v2y="2" v3x="3" v3y="3" v4x="4"
v4y="4" v5x="10" v5y="10" v6x="11" v6y="11" v7x="12" v7y="12" v8x="13"
v8y="13" dz="20" lunit= "mm"/>
```

3.4.23 Tessellated solid

NB: Currently not supported by ROOT

The GDML tessellated solid can be made of any number of triangular and/or quadrangular facets forming a closed space. Each facet is defined by specifying references to (names of) three (for the triangular one) or four (for the quadrangular one) vertices. The vertices are standard `position` elements and should be declared beforehand within the `define` section. If a referenced vertex is not defined, an error message is displayed. By looking from the outside of the tessellated solid, vertices are listed in the anti-clockwise order for each of the facets. There are two ways of defining vertices: `ABSOLUTE` (default), and `RELATIVE`. In the first case each vertex coordinates are considered to be absolute coordinates in the reference system of the solid. In the second case, only the first listed vertex has absolute coordinates, while the remaining ones are considered to be translations from the first one.

The tessellated solid has only one attribute `name` which is standard for any `SolidType` element and corresponds to the name the solid can be referenced by. The subelements of tessellated solid element are all of `FacetType`, in other words they are the triangular and/or quadrangular facets the solid is made of.

The triangular (quadrangular) facet element contains three (four) attributes `vertex1`, `vertex2`, `vertex3`, (`vertex4`), being references to the vertices (defined beforehand) of the facet and an optional attribute `type` indicated whether the coordinates of the vertices are `ABSOLUTE` (default) or `RELATIVE`.

The following example illustrates the construction of a simple tessellated solid made from a square base and six triangular walls. We first define the vertices

```
<define>
  <position name="v1" x="10" y="10" z="0" unit="m"/>
  <position name="v2" x="-10" y="10" z="0" unit="m"/>
  <position name="v3" x="-10" y="-10" z="0" unit="m"/>
  <position name="v4" x="10" y="-10" z="0" unit="m"/>
  <position name="v5" x="7" y="3" z="20" unit="m"/>
  <position name="v6" x="-3" y="7" z="5" unit="m"/>
</define>
```

and then we build the tessellated solid out of them

```
<tessellated name="pyramid">
  <triangular vertex1="v1" vertex2="v2" vertex3="v6" type="ABSOLUTE"/>
  <triangular vertex1="v2" vertex2="v3" vertex3="v6" type="ABSOLUTE"/>
  <triangular vertex1="v3" vertex2="v4" vertex3="v5" type="ABSOLUTE"/>
  <triangular vertex1="v4" vertex2="v1" vertex3="v5" type="ABSOLUTE"/>
  <triangular vertex1="v1" vertex2="v6" vertex3="v5" type="ABSOLUTE"/>
  <triangular vertex1="v6" vertex2="v3" vertex3="v5" type="ABSOLUTE"/>
  <quadrangular vertex1="v4" vertex2="v3" vertex3="v2" vertex4="v1" type="ABSOLUTE"/>
</tessellated>
```

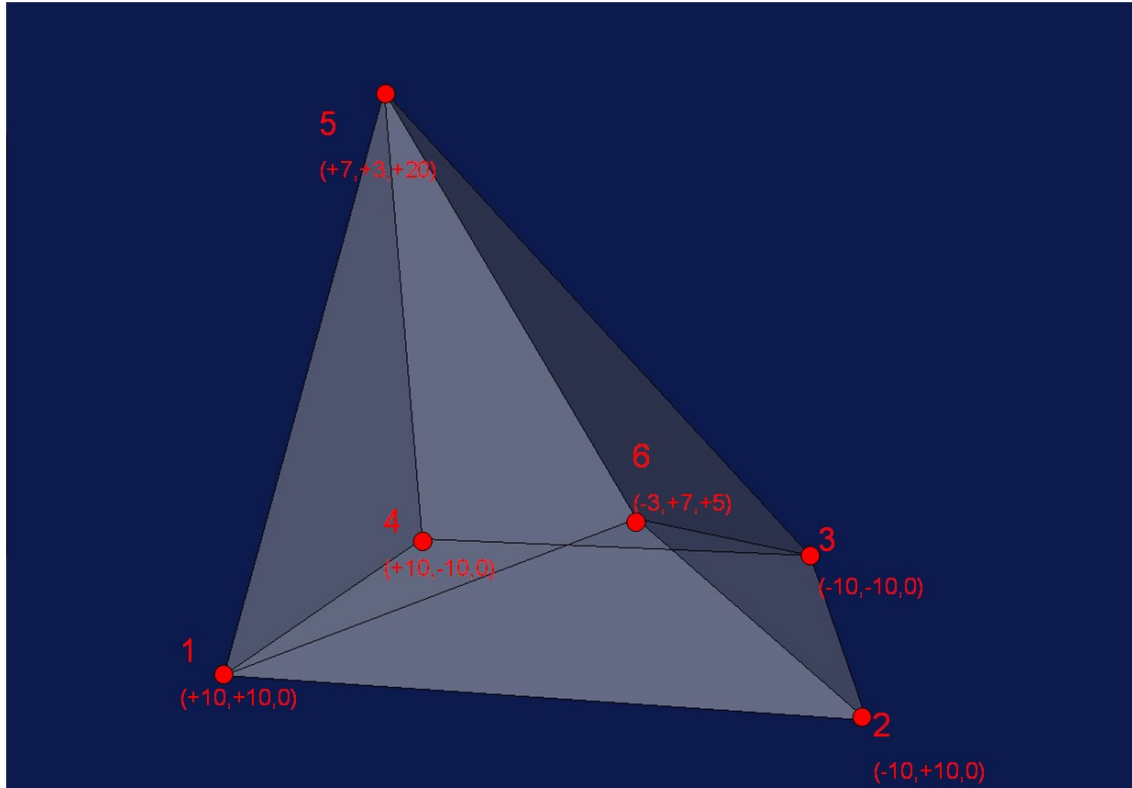



Figure: graphical representation of the tessellated solid as described in the example.

3.4.24 Tetrahedra

The tetrahedron solid in GDML is defined by specifying the coordinates of the four vertices. In order for the solid not to be degenerated, the four points must not lie on the same surface. The `tet` solid element contains the standard name attribute and four attributes being references to the vertices namely `vertex1`, `vertex2`, `vertex3` and `vertex4`. These vertices are standard position elements which must be defined beforehand within the `define` part.

The following example illustrates the construction of the tetrahedron.

```
<define>
  <position name="v1" x="10" y="10" z="0"/>
  <position name="v2" x="-10" y="10" z="0"/>
  <position name="v3" x="-10" y="-10" z="0"/>
  <position name="v4" x="0" y="0" z="10"/>
</define>
```

```
<tet name="halfpyramid" vertex1="v1" vertex2="v2" vertex3="v3" vertex4="v4"/>
```

3.4.25 Using loops for solids

Suppose you want to define a set of solids with the same shape but with dimensions varying according to a particular function. In this case loops are very useful, because they avoid repetition of solids definitions. For a more detailed discussion on loops please refer to Section 3.5.1. The following example shows you how to use a loop to define a set (10) of boxes with different dimensions. Without a loop you would write the following code:

```
<box name="box0" x="1" y="5" z="20" />
<box name="box1" x="4" y="4.5" z="18" />
<box name="box2" x="7" y="4" z="16" />
<box name="box3" x="10" y="3.5" z="14" />
<box name="box4" x="13" y="3" z="12" />
<box name="box5" x="16" y="2.5" z="10" />
<box name="box6" x="19" y="2" z="8" />
<box name="box7" x="22" y="1.5" z="6" />
<box name="box8" x="25" y="1" z="4" />
<box name="box9" x="28" y="0.5" z="2" />
```

Using a loop you would only need four lines

```
<variable name="x" value="0" />
```

declared in the "define" section and

```
<loop for="x" to="9" step="1">
  <box name="box" x="1+(x*3)" y="5-(x/2)" z="20-(2*x)" />
</loop>
```

3.5 Structure

The structure section is where volumes in the geometry are defined. Volumes are referenced to solids and materials defined earlier to form the shape and composition of that volume. This volume is then a logical volume. Volumes can be placed within this logical volume, these volumes are called physical volumes. These 'physvols' can be positioned and rotated how you wish within the logical volume. An example of the GDML for this is below:

```
<volume name="World">
  <materialref ref="Air"/>
  <solidref ref="WorldBox"/>

  <physvol>
    <volumeref ref="vol0"/>
    <positionref ref="center"/>
    <rotationref ref="identity"/>
  </physvol>
</volume>
```

3.5.1 Loops

Loops are a means of avoiding tedious repetitions of solids and volumes definitions. To use a loop you need to define a variable first (section 3.2.1). Suppose you defined a variable x with an initial value of 0, like this:

```
<variable name="x" value="0"/>
```

then you can use it inside a loop statement as in the following example:

```
<loop for="x" to="8" step="2">  
...  
</loop>
```

The code within the loop tags will be executed 5 times in this case, because the variable x starting from the initial value of 0 (as in its definition) will be increased by 2 (according to the step value) for each loop until it reaches the value of 8 (as stated in the to attribute).

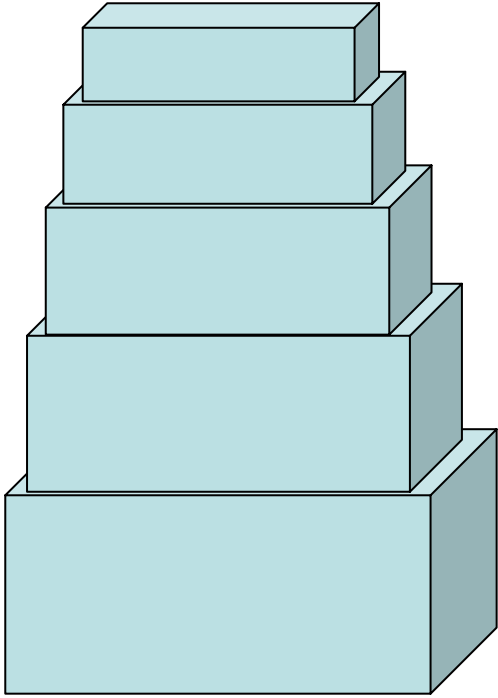
Loops are very useful to access elements of matrices (see section 3.2.2). Consider the following definitions:

```
<variable name="i" value="1"/>  
<matrix name="m" coldim="3" values=" 0.4  9  126  
                                8.5  7   21  
                                34.6 7    9" />
```

Variable i can be used as an index inside a loop to access a certain subset of the elements of the matrix. For example we can write the following:

```
<loop for="i" to="3" step="1">  
... m[1,i] ...  
</loop>
```

With that code we will be able to access the three elements of the first row of the matrix, i.e. 0.4, 9 and 126. Loops are used when defining solids, logical volumes and physical volumes, let's see how this is done in practice starting from an example. Suppose we want to create the following structure:



We need to write the code for 5 boxes, 5 logical volumes that refer to the boxes and 5 physical volumes that refer to the 5 logical volumes! The code would be the following:

```
<solids>
<box name="box1" x="10" y="5" z="5" />
<box name="box2" x="9" y="4.5" z="4" />
<box name="box3" x="8" y="4" z="3.2" />
<box name="box4" x="7" y="3.5" z="2.56" />
<box name="box5" x="6" y="3" z="2.048" />
</solids>
<structure>
<volume name="volbox1">
<solidref ref="box1" />
<materialref ref="iron" />
</volume>
<volume name="volbox2">
<solidref ref="box2" />
<materialref ref="iron" />
</volume>
<volume name="volbox3">
<solidref ref="box3" />
<materialref ref="iron" />
</volume>
<volume name="volbox4">
<solidref ref="box4" />
<materialref ref="iron" />
</volume>
<volume name="volbox5">
<solidref ref="box5" />
<materialref ref="iron" />
</volume>
<volume name="world">
...
<physvol>
<volumeref ref="volbox1" />
<position x="5" y="0" z="40" />
</physvol>
<physvol>
<volumeref ref="volbox2" />
<position x="5" y="2" z="38" />
</physvol>
<physvol>
<volumeref ref="volbox3" />
<position x="5" y="3.5" z="36" />
</physvol>
<physvol>
<volumeref ref="volbox4" />
<position x="5" y="3.9" z="34" />
</physvol>
<physvol>
<volumeref ref="volbox5" />
<position x="5" y="4.15" z="32" />
</physvol>
</volume>
</structure>
```

Of course this is always possible; but using the loop tags (and combining them properly with matrices), you would have the same result with fewer code lines and less complexity. Here's the equivalent code using the `loop` and `matrix` tags:

```

<define>
  <matrix name="m" coldim="5" values="0 2 3.5 3.9 4.15
                                     5 4 3.2 2.56 2.048" />
  <variable name="x" value="1" />
  <variable name="x1" value="0" />
  <variable name="x2" value="0" />
</define>
<solids>
  <loop for="x" to="5" step="1">
    <box name="box" x="10-(x-1)" y="5-(x-1)/2" z="m[2,x]" />
  </loop>
</solids>
<structure>
  <loop for="x1" to="4" step="1">
    <volume name="volbox">
      <solidref ref="box" />
      <materialref ref="iron" />
    </volume>
  </loop>
  <volume name="world">
    ...
    <loop for="x2" to="8" step="2">
      <physvol>
        <volumeref ref="volbox" />
        <position x="5" y="m[1,x2/2+1]" z="40-x2"/>
      </physvol>
    </loop>
  </volume>
</structure>

```

In this example we only "looped" 5 times, but just think if we needed to loop 100 times or more, the difference (in terms of code lines) between the two approaches would be enormous.

Now two important rules you need to follow when using loops:

- 1) Always use a different (and properly initialized) variable for each loop tag (in our example `x`, `x1`, `x2` for the three loops).
- 2) When you use linked loop triples (*loop triple* = 3 loop tags, as we did in our example; *linked* in the sense that we looped over a solid, then we looped over a logical volume that referred to that solid, and finally we looped over a physical volume that referred to that logical volume) you always have to be sure that the number of steps is the same for every loop tag of that particular triple. In our example that number was 5: from 1 to 5 in the first loop, from 0 to 4 in the second loop, and from 0 to 8 (but with a step of 2) in the third loop.

3.5.2 GDML Modules

GDML modules are GDML files used in the definition of (complex) geometries. They allow an easier to understand view of big geometries as they split it into smaller (and therefore more readable) pieces.

A GDML module is a normal GDML file and it is completely independent from all the other modules (also from its mother module): it contains all the information (definitions, materials, solids and volumes) it needs, to be fully defined. It defines a logical volume (which

corresponds to its world volume) which can be referenced by physical volumes of other modules (which are called mother modules).

To include a GDML module (say *child.gdml*) in a mother module (say *mother.gdml*), in the structure section of the mother module you should add a physical volume which points to the child module, as in the following example:

-Instead of having one file:

```
// mother.gdml

<volume name="child">
  <materialref ref="Alluminium"/>
  <solidref ref="ChildBox"/>
</volume>
<volume name="mother">
  <materialref ref="Iron"/>
  <solidref ref="MotherBox"/>
  <physvol>
    <volumeref ref="child"/>
    <positionref ref="center"/>
    <rotationref ref="identity"/>
  </physvol>
</volume>
```

-You can define `child` logical volume in a different GDML file, so you will have two files:

```
// mother.gdml

<volume name="mother">
  <materialref ref="Iron"/>
  <solidref ref="MotherBox"/>
  <physvol>
    <file name="child.gdml"/>
    <positionref ref="center"/>
    <rotationref ref="identity"/>
  </physvol>
</volume>
```

```
// child.gdml

<volume name="child">
  <materialref ref="Alluminium"/>
  <solidref ref="ChildBox"/>
</volume>
```

So in the general case instead of writing:

```
<volumeref ref="_logical_volume_"/>
```

You will need to write:

```
<file name="_logical_volume_.gdml"/>
```

And then define `_logical_volume_` inside `_logical_volume_.gdml`.

In the example given above, the mother module will only contain the material `Iron` and the solid `MotherBox`, while the child module will contain only the material `Aluminium` and the solid `ChildBox`, thus enhancing readability and modularity.

Some advice:

-While it is not useful to have a module for every single volume of the geometry, it is good to have one for each group of volumes that you would consider as a module in the real world (see par. "Multiple GDML files").

-Modules should have low coupling, i.e. they should be as independent as possible from other modules (this improves reusability dramatically).

-Modules should have high cohesion, i.e. they should contain all (and only) the volumes that contribute to the geometry of the corresponding real world modules (this improves readability).

3.5.3 Volume Auxiliary Information

Volumes can have auxiliary information attached to them through the GDML. The auxiliary tags look like this:

```
<auxiliary auxtype="SensDet" auxvalue="veloSD1"/>
```

They have two attributes, 'auxtype' and 'auxvalue' which are meant to specify what kind of auxiliary information it is and what is its value. It is important to stress here that the GDML parser does not interpret in any way the auxiliary attributes. The auxiliary attributes are stored in memory and available to the user once the parsing is done. The auxiliary tags are placed within a volume like this:

```
<volume name="main">
  <materialref ref="Iron"/>
  <solidref ref="Box"/>
  <physvol>
    <volumeref ref="world"/>
    <positionref ref="center"/>
    <rotationref ref="identity"/>
  </physvol>
  <auxiliary auxtype="SensDet" auxvalue="veloSD1"/>
  <auxiliary auxtype="Color" auxvalue="blue"/>
</volume>
```

In order to access the auxiliary information one has to use the `GetAuxiliaryMap` method of the `GDMLProcessor`. This method returns a map with the keys being the pointers to the volumes containing auxiliary information and the values being vectors of pairs (`auxtype`, `auxvalue`).

A full example of how to retrieve this information once imported is contained within:

`$GDMLTOP/CPPGDML/Examples/g4gogdml/g4godml.cpp`

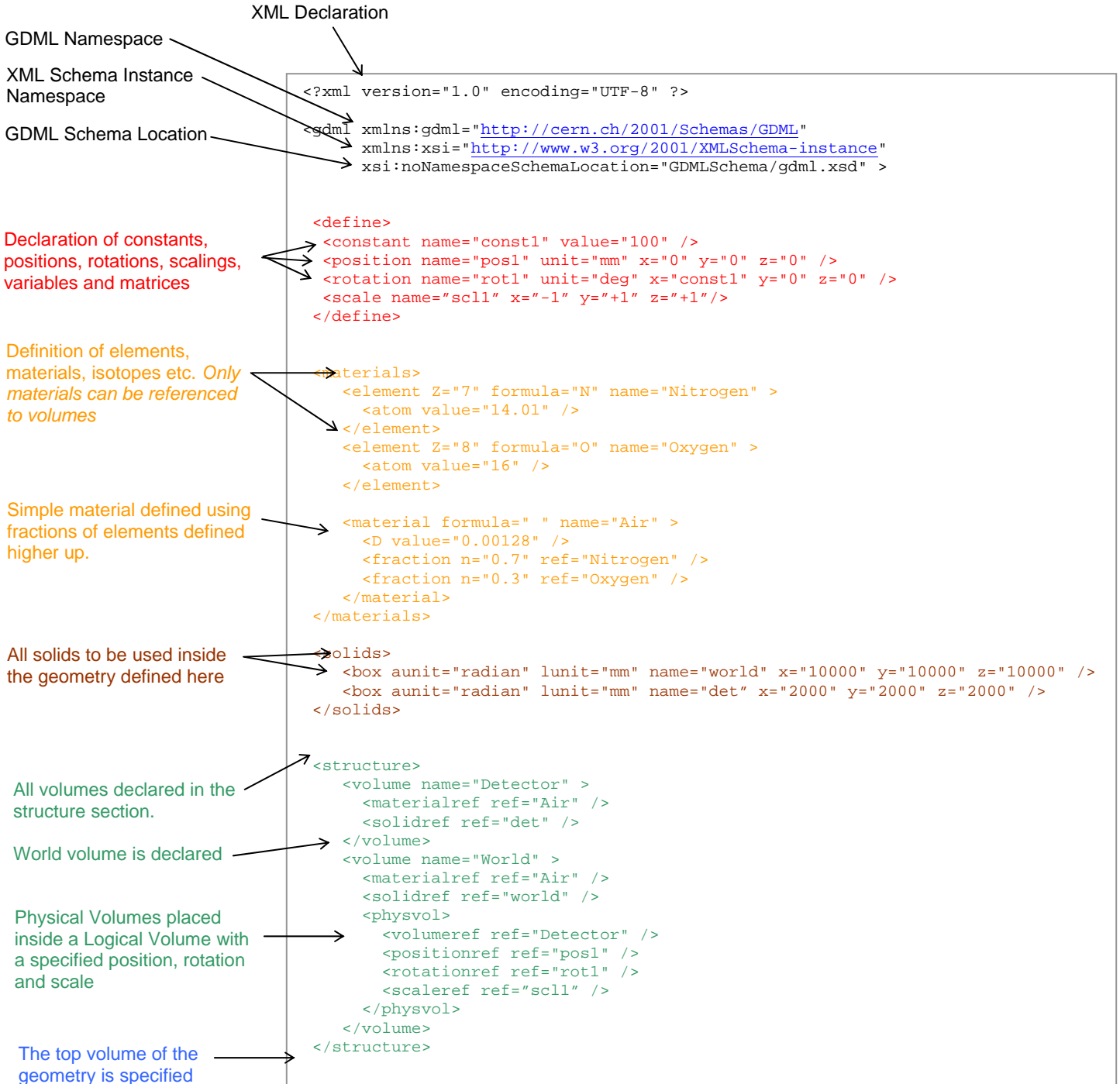
3.6 Setup

The top volume of the geometry needs to be specified. This is done using the setup section. The following GDML is the generic and most common for the setup section.

```
<setup name="Test1" version="1.0">  
  <world ref="World"/>  
</setup>
```

The only part you should need to change is the `world ref` name to the volume name of the top volume of your geometry. Only one setup is defined here, however it is possible to define multiple geometry setups choosing different volumes as world volumes from all the already defined volumes.

3.7 GDML file example



4. CPPGDML

4.1 GDML Reading

4.1.1 Importing GDML files into GEANT4 Geometry

GDML files can be directly imported into GEANT4 geometry, using the CPPGDML libraries in your GEANT4 application. Generally you will want to put the following lines into your `DetectorConstruction` class:

- In the class constructor (as far as the initializations are concerned – step 2)
- In the `Construct` method (steps 3 and 4)

This can be done in four steps:

1) Include the necessary header files

```
#include "saxana/SAXProcessor.h"
#include "saxana/ProcessingConfigurator.h"
#include "G4Processor/GDMLProcessor.h"
```

2) Declare and initialize properly the objects needed by the parser

```
SAXProcessor saxp;
ProcessingConfigurator config;
GDMLProcessor* proc = GDMLProcessor::GetInstance();

saxp.Initialize();
config.SetURI("source_file.gdml");
config.SetSetupName("Default");
saxp.Configure(&config);
```

`source_file.gdml` will be your GDML source file, in case of multiple GDML files (modules) you should put the main GDML file there (the main mother module).

3) Run the parser

```
proc->SetDuplicationWarning(config.URI()); //optional
saxp.Run();
```

The first line is optional and will turn the duplication warning flag on. So that when solids with the same name are found in different modules, a warning message is generated. Usually this happens when the same logical volume is defined by different GDML modules. This is not a problem, but it may suggest that the input GDML modules have a low cohesion, therefore it maybe useful to review the way you have organized your geometry.

4) Get the World volume

```
G4VphysicalVolume* W; //world volume
W = (G4VphysicalVolume*)proc->GetWorldVolume();
W->GetLogicalVolume()->SetVisAttributes(G4VisAttributes::Invisible);
```

The third line is optional, but commonly used, because you generally don't want to be able to see the world volume (usually the world volume is nothing but a box made of air or vacuum).

4.2 GDML Writing

4.2.1 Exporting GEANT4 Geometry

GEANT4 geometry can be exported into GDML files. To do so you should do the following:

1) Include the G4 writer header file

```
#include "G4Writer/G4GDMLWriter.h"
```

2) Get the world volume from the geometry

```
G4VphysicalVolume* g4wv = G4TransportationManager::  
GetTransportationManager()->GetNavigatorForTracking()  
->GetWorldVolume();
```

3) Instantiate the G4GDMLWriter

```
G4GDMLWriter g4writer("/path/to/gdml.xsd", "/path/to/output.gdml");
```

The first parameter is the GDML Schema path, it is very important to set this right, otherwise you will get error messages while reading the file back to GEANT4. The second parameter is the path to the file you want to export the geometry into.

Then there is an optional third parameter which is an integer and its default value is 0. This parameter tells how you would like the names of volumes (as well as of solids and materials) to be formatted in your output GDML file. Here an explanation of the three possible values:

0 (default) – The names will be concatenated with their logical address in hexadecimal format. This is to avoid name duplication in the GDML output, in fact GEANT4 allows different volumes (as well as materials and solids) with the same name. When you will read the GDML output file back into GEANT4 the address part will be stripped off and you will have the original names. This is almost always safe.

1 – The names will be concatenated with their logical address in long format. When you will read the GDML output file back into GEANT4 the address part will NOT be stripped off and you will have the original names plus their old logical address in long format. This is always safe but it is a bit ugly to have names with long numbers inside GEANT4.

2 – The names will NOT be concatenated with anything. So the names in the output file will correspond exactly to the ones you have in GEANT4. This is the prettiest format although you have to be REALLY sure that you don't have name duplication inside GEANT4, otherwise this will generate a GDML output with duplicated names, which will be unreadable by any parser of course.

4) Call the dump method

There are three ways of doing this:

a) `g4writer.DumpGeometryInfo(g4wv);`

Dumps all the geometry in a single GDML output file.

b) `g4writer.DumpGeometryInfo(g4wv, levels);`

Dumps the geometry modularizing the output. The second parameter is a `vector<int>` containing the numbers of the geometry levels you want to modularize. So if you want to create modules for all the volumes of the first and the third level you should set the levels variable like this:

```
vector<int> levels;  
levels.push_back(1);  
levels.push_back(3);
```

and then call the dump method:

```
g4writer.DumpGeometryInfo(g4wv, levels);
```

c) `g4writer.DumpGeometryInfo(g4wv, names);`

Dumps the geometry modularizing the output. The second parameter is a `vector<string>` containing the names of the volumes you want to modularize. So if you want to create modules for a volume called “door” and a volume called “window” you should set the names variable like this:

```
vector<string> names;  
names.push_back("door");  
names.push_back("window");
```

and then call the dump method:

```
g4writer.DumpGeometryInfo(g4wv, names);
```

4.2.2 Exporting ST-Viewer CAD Geometry

ST-Viewer is an interactive tool for viewing 3D CAD information. ST-Viewer includes functions to display and relate product identification information, assembly structure information and 3D geometry information. ST-Viewer implements the ISO STEP standard for product model data. ST-Viewer works with a very large range of CAD, CAM, CAE and PDM systems. A list of systems that have been tested with the ST-Viewer can be found at the www.steptools.com.

ST-Viewer uses intermediate files to store CAD geometry and structure information. These files (with `.geom` and `.tree` extensions respectively) use an internal non-standard format that can be parsed and translated into GDML format using the libraries provided by the GDML processor (CPPGDML).

The geometry of the solids is defined by means of triangular facets, i.e. every solid in the `.geom` files is a tessellated solid made of *only* triangular facets. Therefore the GDML output will be a collection of tessellated solids where each solid has its own set of facets which are in turn defined by a set of three vertices. The vertices are named vectors of three values (x,y and z coordinates) and are defined in the beginning of the GDML file. The facets within a tessellated solid contain references to the names (*not the values*) of the three vertices they are made of (for more information on how a tessellated solid is defined in GDML please refer to the GDML Solids Schema : `solids.xsd`).

Of course multiple facets may share a common vertex, therefore in their definition there will be a reference to the same vertex name. Again, the actual values of the vertices' vectors are stored in the first part (the one enclosed by the `<define>` and `</define>` tags) of the GDML file.

In order to use the parser in your own application you need to

1) Include the *STEPWriter* header:

```
#include "STEPWriter/STEPWriter.h"
```

2) Instantiate a *STEPWriter* object (or a pointer to it):

```
STEPWriter* sw = new STEPWriter();
```

3) Call the *parse* method:

```
sw->parse("/path/to/file.geom", "/path/to/file.tree", false);
```

This method takes three parameters: the first one is a string containing the path to the `.geom` file, the second one is a string containing the path to the `.tree` file and the third one is a boolean flag which should be set properly:

- a) Set it to `false` [recommended] if you don't want common vertices to be repeated in the define section, i.e. facets sharing the same vertex will refer to the same vertex name defined at the beginning of the GDML file. In this case there will not be any two vertices with the same coordinates.
- b) Set it to `true` if you don't care about repetition of vertices in the "define" section of the GDML file. Facets sharing the same vertex will refer to different vertices names. Therefore there will be vertices with the same coordinates and different names.

Setting the flag to `false` will make the parsing process generally slower but the size of the output GDML file will be a lot smaller. Setting the flag to `true` will allow the parser to go faster (because it doesn't need to look for existing vertices) but this will generally increase the size of the GDML output file (because of the repetitions of vertices with same coordinates). During a test, a input `.geom` file of 9.5 MB took 50 seconds to be parsed with the flag set to `false` and the result was a 10.5 MB GDML output file; the same input file took 10 seconds to be parsed with the flag set to `true` and the result was a 33 MB GDML output file.

4) Call the *dump* method:

```
sw->dump("/path/to/file.gdml", "/path/to/gdml.xsd");
```

This method is responsible for writing the GDML output file. It takes two parameters: the first one is the path to the GDML output file and the second one is the path to the GDML Schema file. There are no constraints as far as the output file location is concerned, but you should set the GDML Schema location properly because a GDML output file with a wrong GDML Schema reference will cause errors when read by Geant4 or other programs.

The result of the last statement is a GDML file corresponding to the .geom and .tree files taken as input.

5. User extensions to GDML

GDML is defined in an XML schema. Inside the schema are all the definitions of the tags and the rules associated to those tags. Seeing as all the tags are simply defined within the schema, there is the possibility to extend that schema as XML is a fully extendable markup language. This section will discuss how to add custom elements to GDML. This could be useful if certain information was required to be contained in the GDML file for an application of GDML. The extension could be made to contain a new tag and values within whichever chosen part of the GDML the user requires. It may be the case that within the binding to GDML being used, there are methods already in place for information not supported by the GDML Schema, in which case the user extension facility will help implement that.

An example of a user extension is implemented in CPPGDML/Examples/SimpleExtension. This example will be used as a point of reference throughout this section.

The Schema

The first step in extending GDML is to redefine the GDML schema. It needs to include the custom elements of the extension. The extension is done in a separate schema file that imports the original schema as opposed to adding new parts to the old one. The SimpleExtension's schema example is found in:

```
SimpleExtensionSchema/SimpleExtension.xsd
```

The original main GDML schema is imported from its local, relative location within your GDML directory. The new schema file normally will need to contain two main parts:

1. A 'redefine' element

This element uses the extension schema to modify existing GDML elements by adding additional content. This is called by this line which gives the location of the schema to redefine:

```
<xs:redefine schemaLocation="../../../GDMLSchema/gdml.xsd">
```

In most cases, the extension will be to define new objects like colour, to a volume etc. In this situation the redefine needs to add a `ReferenceType` to the base it needs to be added to:

```
<xs:extension base="VolumeType">
  <xs:sequence>
    <xs:element minOccurs="0" maxOccurs="1" name="colouref" type="ReferenceType"/>
  </xs:sequence>
</xs:extension>
```


2. New Elements

The new schema will need to define the new 'element' that the ReferenceType will be referencing to. In the example of adding a colour reference to a volume, the element 'colour' will need to be defined.

When elements are defined, attributes of that element are defined within it, but if tags within the element contain more than one attribute, a type needs to be defined, and then referenced to. This is observed in the Simple Example schema file.

All the new elements cant be contained inside the <gdml> tag as they are not part of that schema, there needs to be a new container defined that contains the new elements AND the original <gdml>. This is shown below:

```
<xs:element name="gdmlEXTENSION">
  <xs:annotation>
    <xs:documentation>
      New container for new element and gdml.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="colours" minOccurs="0" maxOccurs="1"/>
      <xs:element ref="gdml" minOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

This then means that a GDML file now takes the following structure based on the above:

```
<gdmlEXTENSION xmlns:gdml_simple_extension= "http://www.example.org"
  xmlns:xs= "http://www.w3.org/2001/XMLSchema-instance"
  xs:noNamespaceSchemaLocation="NEW_SCHEMA_LOCATION.xsd">
  <colours>
    <colour name="aColour" R="1.0" G="0.0" B="0.0"/>
  </colours>
  <gdml>
    <structure>
      <volume>
        <materialref ref="someMaterial"/>
        <solidref ref="someSolid"/>
        <colouref ref="aColour"/> <!-- new reference type -->
      </volume>
    </structure>
  </gdml>
</gdmlEXTENSION>
```

In any GDML file you wish to use your 'extended schema' in, you will need to change the Namespace GDML Schema Location to your extended schema file, not the original. The new schema location must be relative to the location of the GDML file in question. This will then import your new additions as well as the original GDML schema. This is shown above.

Processes & Subscribers

Once the GDML schema has been extended as you need, the extension application needs to have the necessary processes and subscribers that will convert the GDML into C++ objects. Each element of the new schema has a `Process` that instantiates a simple SAX object. This contains string values for all the attributes of that element. There is also a

corresponding `Subscriber` for that element that converts the generic object into objects of the binding your using.

The extension application should have its own processor for registering its custom objects; the `SimpleExtension` example has a `SimpleExtensionProcessor` where pointers to each of the new 'objects' are held.

These new objects (colour objects in this instance of the example) need to be associated to the objects formed from GDML that they referenced to. Therefore an extended subscriber for the said logical volume etc. needs to be created.

In the example, the new objects need to be associated to a volume. The file `volumeExtendedSubscriber.cpp` processes the contents of volume and finds the new `colorref` tag in the volumes child nodes. The new object is retrieved from the `SimpleExtensionProcessor` using the `ref` attribute and assigned to the Volume Object that was created by GDML's `volumeSubscriber`.

All the processes and subscribers must be listed in a Library Load `cpp` file. This then lets the new processes and subscribers be registered with the GMDL parser. The function `SimpleExtensionLibLoad` in the `SimpleExtensionLibLoad.cpp` file shows an example of this.

The `SimpleExtensionLibLoad` function needs to be called after the `SAXProcessor` has been initialized. In the instance of the Simple Example, this is done in `DetectorConstruction.cpp`. This registers the custom handlers with the GDML processing engine.

Try it..

To run the example application you need to source `setup.csh` in the `CPPGDML/Examples` directory. Then from the `CPPGDML/Examples/SimpleExtension` directory, you can run the example using this command:

```
SimpleExtension [filename.gdml]
```

The default file is `test.gdml`, which has a single colored box in the world volume.