

European Organization for Nuclear Research
- *Summer Student Programme* -

Efficiency Improvements in the ROOT-based Detector Geometry Modeler

Work supervised by
Andrei GHEATA
andrei.gheata@cern.ch

Prepared by
Jean-Marie GUYADER
jeanmarie.guyader@gmail.com

September 2011

Table of contents

1. Introduction	3
2. A brief outlook of USolids	3
3. UMultiUnion class - Voxelization	4
3. 1. Synoptic diagram of the class.....	4
3. 2. Voxelization of an instance of UMultiUnion	4
4. Operations on the solids.....	5
4. 1. Description of the methods	5
4. 2. Results for Inside - Interest of voxelization - Scalability	5
4. 3. Other methods	7
5. Conclusion.....	7

1. Introduction

ROOT and GEANT4 are two specific softwares that have been under constant improvement for many years at CERN. While ROOT was initially designed for particle physics data analysis, GEANT4's original aim was to simulate the passage of particles through matter. Among the large range of tools offered by these two programmes, some analogous features can be found. In particular, GEANT4 and ROOT both include geometry packages allowing computations on 3D volumes. It was therefore decided to gather these similar functionalities together in a unique new package called USolids (standing for Universal Solids).

2. A brief outlook of USolids

One of the first targets of USolids lays in the creation of 3D volumes. Each particular sort of volume (e.g. box, sphere, polyhedron, etc.) derives from a mother class called *VUSolid*. USolid contributors had already created *VUSolid*, as well as the class *UBox*, which permits the creation of solids in the shape of a parallelepiped. In addition, several utility classes were implemented before the beginning of the project. For instance, they define translations, rotations, mathematical applications, points or vectors in 3D.

This summer student project first focused on creating a new class for geometries based on the Boolean union of several sub solids. This new class is called *UMultiUnion*. So far, only *UBoxes* can make up a *UMultiUnion*.

For reminder, let us consider the example of the Boolean union of a sphere with a cube. The resulting geometry is the following:

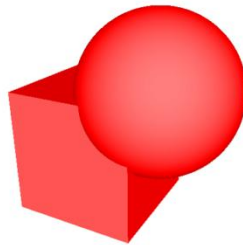


Figure 1 **Boolean union of a sphere and a cube**

NB: The Boolean union of more than two solids (disjoint or not), is possible.

Below is a schematic of the information given so far:

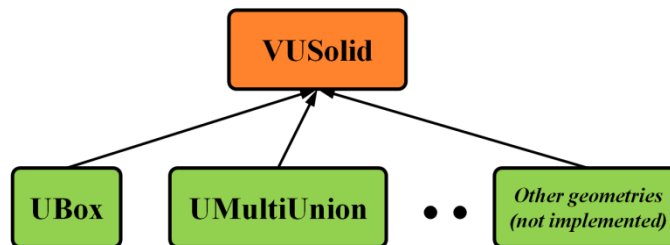


Figure 2 **Heritage from VUSolid mother class**

3. UMultiUnion class - Voxelization

Building up a union of several nodes makes it possible to realize subsequent treatments on the resulting geometry. This is the main usage of *UMultiUnion*, the first class that was defined during this summer student project.

3. 1. Synoptic diagram of the class

An instance of *UMultiUnion* is characterized by an array of several nodes. Each node has to be instantiated using the internal class *UNode*. The creation of a node requires two elements: an intrinsic solid (instance of one of *VUSolid*'s daughter classes) and a 3D transformation (translation and rotation).

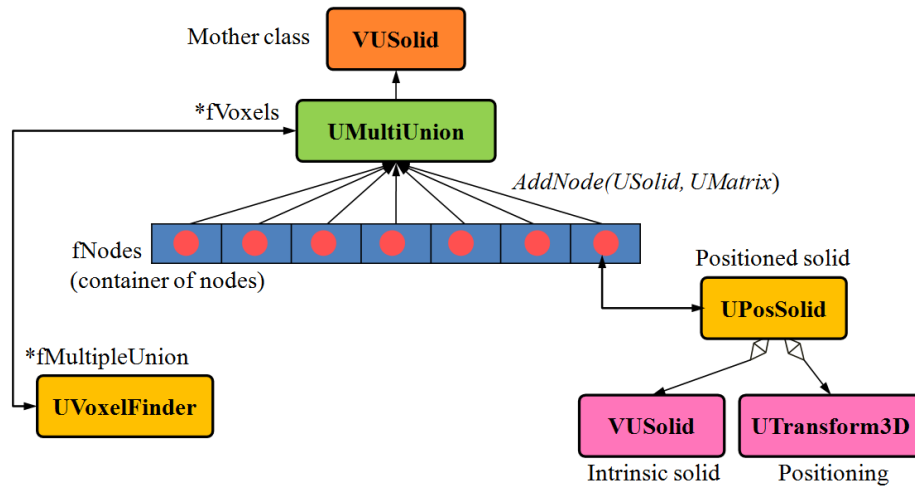


Figure 3 Synoptic diagram of *UMultiUnion*

Since *UMultiUnion* derives from *VUSolid*, it is possible that a node of an instance of *UMultiUnion* might contain itself a Boolean union of several solids.

3. 2. Voxelization of an instance of UMultiUnion

The algorithms carried out in the context of the *USolids* library have to be efficient when it comes to the speed of execution. With this in prospect, voxelization techniques were considered. Indeed, they enable one to know which sub solids are located in a delimited part of the 3D space. In a nutshell, voxelization operations create a virtual irregular grid used to spot the different nodes.

The operations of voxelization are carried out in a specific class named *UVoxelFinder*, the other important class which was developed during the summer project. Voxelization is realized following several steps. First of all, slices are determined along each axis using the bounding boxes of each node. Then, these slices are sorted in increasing order and some are deleted if they are too close to each other. To finish, the nodes contained in each slice are then stored in a memory (under the form of an array of char), for each axis.

The following example shows the voxelized structure, using a 2D example.

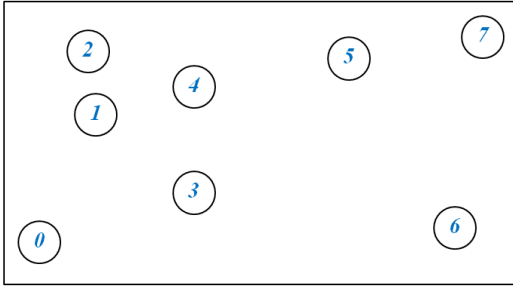


Figure 4 UMultiUnion instance to be voxelized

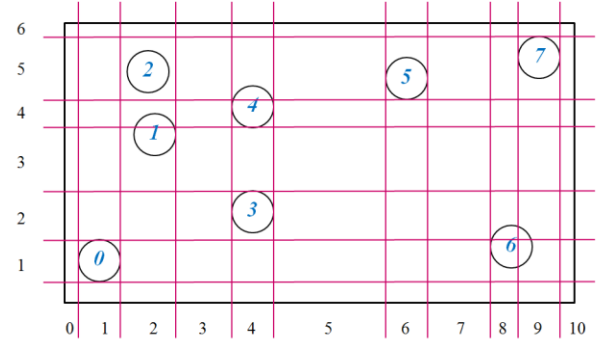


Figure 5 Voxelized structure

Applying Boolean operators (e.g. OR, NOT, AND) to the memories of each axis, it is possible to determine which solids are contained in a given voxel. Considering a point (x,y,z) , one can also compute in which voxel it is located via a binary search. *UVoxelFinder* enables the user to find the candidates corresponding to a particular point.

4. Operations on the solids

4. 1. Description of the methods

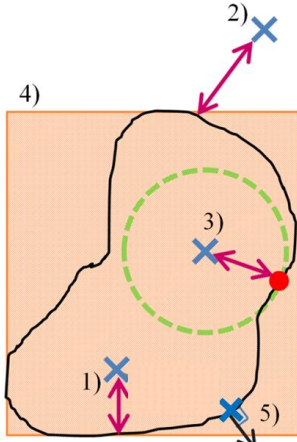


Figure 6 Description of the methods related to 3D shapes

For each class of solids, several similar methods have to be defined. Let us take the following example:

1) *DistanceToIn*: from a point and a given direction, this method shall return the distance between the point and the solid, along the set direction;

2) *DistanceToOut*: idem, with a point located outside the solid;

3) *SafetyFromInside* / *SafetyFromOutside*: these methods are identical to the two previous ones, except that in this case, no particular direction is set;

4) *Extent*: computes the extension of the structure along one axis, or along the three axes;

5) *Normal*: this method returns the normal vector corresponding to a given point.

In addition, a method named *Inside* shall return the state of the point: inside, outside or on a surface of the *UMultiUnion* structure.

These methods had already been written before the beginning of this project. The work was therefore to implement them in the case of a *UMultiUnion*, using voxelization and optimizing the methods as much as possible.

4. 2. Results for Inside - Interest of voxelization - Scalability

In order to assess the efficiency of voxelization, two methods aiming at determining the state of a point have been implemented: *Inside* and *InsideDummy*. The first one uses the voxelized structure, while the second does not.

The scalabilities of the methods were tested as a function of the number of nodes. Thus, a regular arrangement of a variable number of nodes was created and a large number of random points were generated in the 3D space. For each of these points, the inside methods were called and the corresponding times summed.

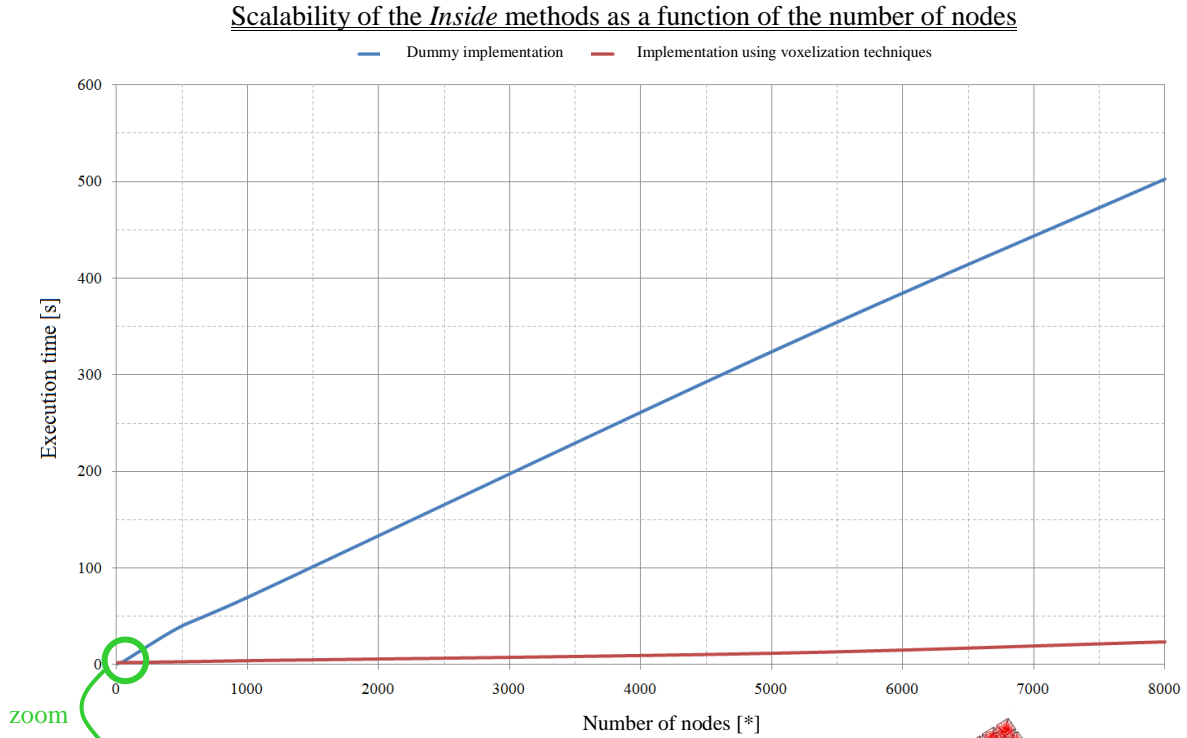


Figure 7 Scalability of the *Inside* methods

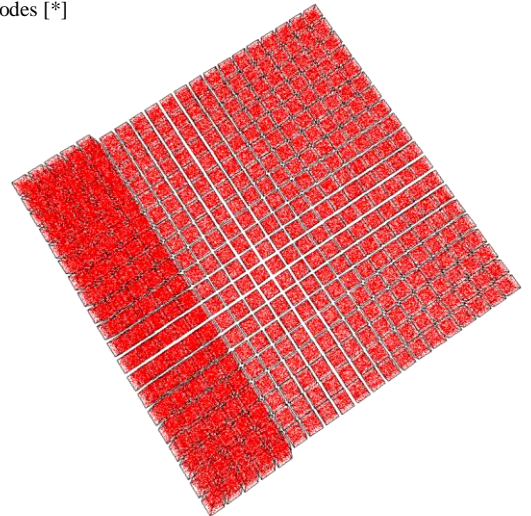


Figure 8 Red points are points found inside the structure

Provided that the number of shapes does not exceed a certain quantity (here 25), the algorithm using voxelization techniques remains more efficient than the dummy implementation. The best algorithm can thus be used in the right configuration (below and above 25, in this case).

The scalability of *Inside* was progressively improved using *Callgrind*. This software shows the repartition of the execution time between the different methods.

4. 3. Other methods

The other methods presented at the end of page 5 were implemented and most of them were optimized with regards to the dummy implementations.

Below is a report of the tests carried out in order to assess whether or not the chosen implementations were successful or not:

Nº	Tested method(s)	Nature of the test(s)	Results
1	Extent	Display of the superior and inferior limits	The limits are coherent with the placement of the nodes: OK
2	SafetyFromInside	Display of the computed safety for given points and directions	Correct safeties are found: OK
3	SafetyFromOutside	Display of the computed safety for given points and directions	Correct safeties are found: OK
4	SafetyFromInside SafetyFromOutside	An automatic ROOT bridge class is used to test the safety methods	Test returns OK
5	DistanceToIn (*)	Display of the computed distance for given points and directions	Correct distances are found: OK
6	DistanceToOut	Display of the computed distance for given points and directions	Correct distances are found: OK
7	DistanceToIn DistanceToOut	An automatic ROOT bridge class is used to test the distance methods	Test returns OK
8	Normal	Computation of the normal, even for a point not located on a surface	The method seems to return correct results
9	Normal	An automatic ROOT bridge class is used to test the normal method	(**)

(*) First drafts of *DistanceToIn* were implemented, but did not comply with the bridge tests. The dummy implementation is the one which is tested here.

(**) In the case in which *Normal* is tested with ROOT bridge tests, theses tests come to a valid end, but irrelevant error messages do appear.

On the whole, the results seem to show that the implemented methods globally give the behaviours expected in the description of the methods.

5. Conclusion

The initial objective of the project was to create a class representing the Boolean union of several solids. In this respect, the *UMultiUnion* class was implemented. In such a structure, the different solids are placed and/or rotated in the space and are called nodes.

Then, methods carrying out the voxelization of an instance of *UMultiUnion* were written in the class *UVoxelFinder*. Subsequent tests showed that algorithms using voxelization techniques were far more efficient than a simple loop, provided that the number of nodes be large enough.

To finish, other methods related to *UMultiUnion* were coded. Due to lack of time and difficulty, some remain not improved. Nevertheless, most of them show correct behaviours.