New software library of geometrical primitives for modeling of solids used in Monte Carlo detector simulations

2012 J. Phys.: Conf. Ser. 396 052035

(http://iopscience.iop.org/1742-6596/396/5/052035)

Journal of Physics: Conference Series **396** (2012) 052035 doi:10.1088/1742-6596/396/5/052035

New software library of geometrical primitives for modeling of solids used in Monte Carlo detector simulations

Marek Gayer, John Apostolakis, Gabriele Cosmo, Andrei Gheata, Jean-Marie Guyader and Tatiana Nikitina

European Organization for Nuclear Research, CERN CH-1211, Genève 23, Switzerland

E-mail: marek.gayer@cern.ch john.apostolakis@cern.ch gabriele.cosmo@cern.ch andrei.gheata@cern.ch jeanmarie.guyader@gmail.com tatiana.nikitina@cern.ch

Abstract. We present our effort for the creation of a new software library of geometrical primitives, which are used for solid modeling in Monte Carlo detector simulations. We plan to replace and unify the current implementations for geometrical primitive classes in the software projects Geant4 and ROOT with this library. Each solid is implemented as a C++ class providing methods to compute distances of rays to the surface of a solid or to find whether a position is located inside, outside or on the surface of the solid. A numerical tolerance is used for determining whether a position is on the surface. The class methods also contain basic support for visualization. We use dedicated test suites for the validation of the code; these also include performance and consistency tests used for the analysis of candidate implementations of class methods for the new library. We have implemented simple adapter classes to allow the use of the new optimized solids with Geant4 and ROOT geometries.

1. Introduction

1.1. Geant4

Geant4 is a software toolkit for the simulation of the passage of particles through matter. It is used by a large number of experiments and projects in a variety of application domains, including high energy physics, astrophysics and space science, medical physics and radiation protection. It comprises of a complete range of functionality for particle and radiation transport simulation, with tracking, geometry, physics modeling and the capabilities to do biasing, scoring, manage and create output as hits and visualization. The physics processes offered cover a comprehensive range, including electromagnetic, hadronic and optical processes, a large set of long-lived particles, materials and elements, over a wide energy range. It has been designed and constructed to expose the physics models utilized, to handle complex geometries, and to enable its easy adaptation for optimal use in different sets of applications. The toolkit is the result of a worldwide collaboration of physicists and software engineers [1, 2].

The geometry modeler is a key component of the Geant4 software. It offers the ability to describe the geometrical structure of a detector in a natural way. It has been designed for allowing efficient propagation of particles in the geometrical detector model to exploit at the best the features provided by the Geant4 simulation toolkit. Advanced techniques for optimizing tracking in the geometrical model have been seriously taken into consideration in the design, both in order to optimize the run-

time performance and reduce the physical memory consumption when dealing with complex geometrical setups. A great variety of geometrical shapes are defined, including the possibility of combining them with Boolean operations to achieve nearly any possible detector geometry configuration [3].

1.2. ROOT

The ROOT system in an object-oriented framework for large-scale data analysis. ROOT is written in C++ and contains, among others, an efficient hierarchical OO database, a C++ interpreter, advanced statistical analysis (multi-dimensional histograms, fitting, minimization, cluster finding algorithms) and visualization tools. The user interacts with ROOT via a graphical user interface (GUI), the command line or batch scripts. The commands and scripting language is C++ (using the interpreter) and large scripts can be compiled and dynamically linked in. The OO database design has been optimized for parallel access (reading as well as writing) by multiple processes [4]. ROOT contains optimized libraries for geometry modeling that provide similar features and functionality as the Geant4 geometry package. These libraries are designed to supply geometry services to HEP applications for particle transport, tracking, event display or geometry databases.

1.3. Unified Solids

The geometrical models describing the experimental setups in either simulation or reconstruction programs are using solid primitives as building blocks to represent more complex detectors. There is a large collection of supported primitives, ranging from simple 3D shapes like boxes, tubes or cones to more complex ones, allowing for Boolean combinations among these. Geant4 provides a well-tested implementation of such primitives; a similar implementation is also provided within the ROOT framework.

We estimate that in both Geant4 and ROOT, a fraction not less than 70-80% of the effort spent over the past decade on code maintenance in the geometry modeler has been devoted to improving the implementation of the geometrical primitives. This code, written in C++, includes all algorithms required for tracking efficiently particles with high level of precision in an infinite variety of geometrical setups.

In order to reduce the effort required for support and maintenance, and converge on a unique solution based on high quality code, we aim to build a new software library of geometrical primitives. The new library can then be used for solid modeling in Monte Carlo detector simulations [5, 6], and will unify the existing implementations in Geant4 and ROOT, improving wherever possible both reliability and CPU performance of the implemented algorithms. We will adopt a single type (class) for each shape. The library will include about 25 primitives, corresponding to the complete set of solids that are part of the GDML (Geometry Description Mark-up Language) schema [7]. In future, we plan to replace the implementations existing in Geant4 and ROOT with this new library.

The new solids library will contain the common features of Geant4 and ROOT modelers, but also some features available only in one of the two. For example, Geant4 uses a thickness or tolerance, which is a small numerical value (e.g. 10^{-9} mm), so as to determine whether a point is located on surface, inside or outside a solid [3]. The ROOT modeler considers points either inside or outside a solid. In Unified Solids, the concept of points on surfaces is included. The reasons for this choice are to make navigation algorithms more robust, and to provide additional information for use in debugging overlaps in a geometry description.

The optimization of selected complex solids and also the development of specific new ones are considered, wherever a performance deficit is identified in the existing implementations. An example of this is the realization of the multi-union solid, which we describe later.

The most important methods of a Solid are the ones that implement its navigation functionality.

These are:

1. Location of point either inside, outside or on surface (*Inside*)
2. Shortest distance to surface for outside points (*SafetyFromOutside*)
3. Distance to surface for inside points with given direction (*DistanceToIn*)
4. Distance to surface for outside points with given direction (*DistanceToOut*)
5. Shortest distance to surface for inside points (*SafetyFromInside*)
6. Normal vector for closest surface from given point (*Normal*)

The first three methods are also those most performance critical and effort should be concentrated on their optimization. Additional services are methods for obtaining bounding box, capacity, geometrical volume, generation of points on surface/edge/inside of a solid. There are also services that provide data for the visual representation of the solid, like the polyhedral or mesh data for visualization.

2. Methodology

One fundamental aspect of the procedure for implementing the new library is to put in place a comprehensive testing suite, to systematically monitor correctness and performance of the new algorithms in comparison to the corresponding implementations in Geant4 and ROOT.

With this purpose, we defined and implemented a basic infrastructure, which allows to exercise the new code in Geant4 and ROOT and to compare the responses.

A first step was to enable the use of solids of the Unified Solid library in both Geant4 and ROOT geometries. To do this simple adapter classes as native type solids in Geant4 and ROOT. These embed a solid from the new library and delegate to it each navigation function. In figure 1, the design for these classes is shown. The adapters allow for direct performance and consistency comparisons between the new solids and the native ones. Once the new library will be fully validated, the adapter classes can be removed as presented in the final class diagram from figure 2.

The key requirements for the new library are:

- The implementations of primitives should offer better or equal performance to that obtained by the existing implementations in Geant4 or ROOT.
- The code should be easy to maintain, readable, portable.
- The new library should not depend on any external package, but be realized as a standalone package.
- The implementation should take into account the potential of vectorizing the code where possible.
- It should make use, wherever possible, of tools and data structures provided by the Standard C++ library.

The approach envisioned for the realization of the new library can be summarized as follows:

1. Define the interface to be adopted, by combining the APIs provided in Geant4 and ROOT, and implement this using the adaptor pattern [8].
2. Review all algorithms of the existing solids in Geant4 and ROOT; when possible adapt them, and ensure correctness, performance and overall high quality of code.
3. For the cases where new shapes are introduced, develop the new algorithms and code according to the specifications required by the defined interfaces.
4. The implementation of each geometrical primitive should be done in parallel with comprehensive testing for the validity of results and performance.

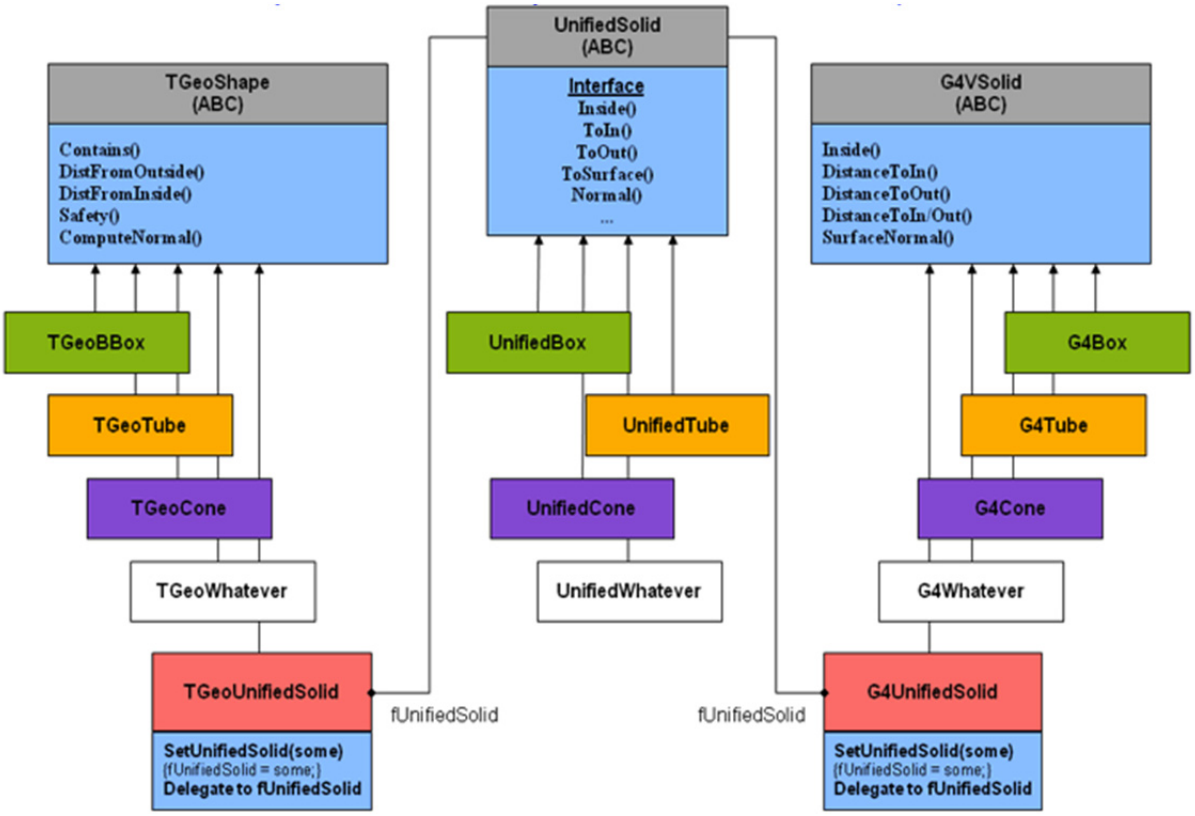

Figure 1. Adaptor pattern classes are used to facilitate transition and enable comparisons.

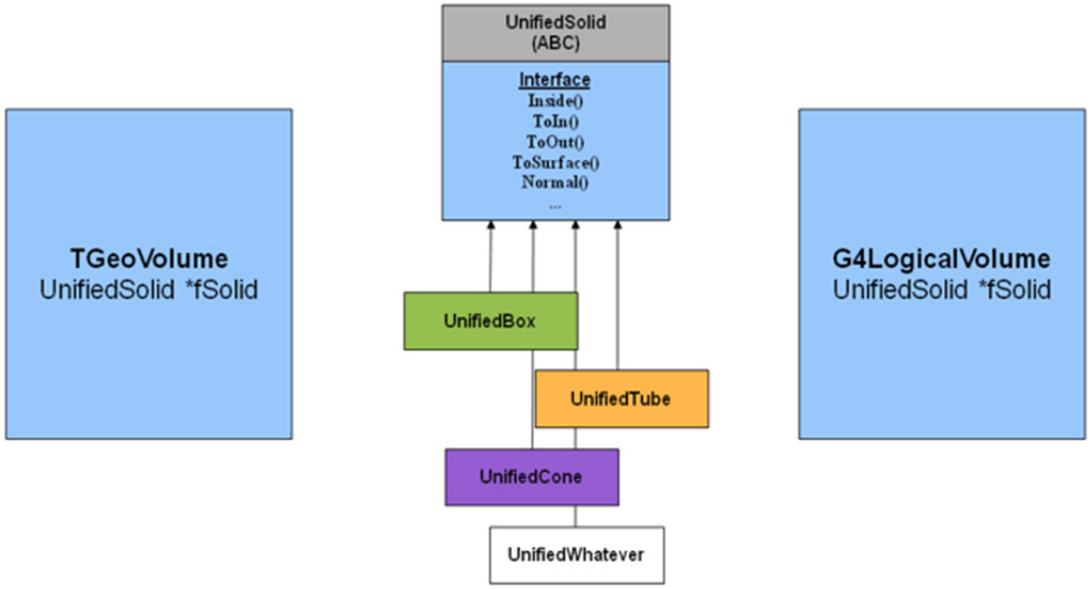

Figure 2. Final configuration in Geant4 and ROOT, for the use of the new geometrical primitives.

3. Testing

Tests and verification of the new code are performed using a comprehensive testing suite, which is created in parallel with the development of the algorithms and implementations for each shape. The testing suite is used to verify the correctness under many possible conditions and for optimization of CPU performance.

Testing include the development of a tool designed to measure computing performance and compare the responses, to identify differences between implementations and areas of improvement.

Specialized tests to verify the scalability of the algorithms as a function of the complexity of the geometrical setup are implemented for selected solids.

Some of these tests are described in the following sections.

3.1. Solid Batch Test (SBT)

The Solid Batch Test (SBT) application is an extendible framework for performing batch tests to solids in Geant4. The test is part of the existing testing suite for the Geant4 geometry modeler. It checks the consistency of all the methods of a solid, using the *Inside* method as a baseline. It contains several tests applicable to all geometrical primitives. It provides a report of each inconsistency found, with its location and type. In one type of test it generates either a set of random points. In another it starts with a grid of points. In both cases it tests the full set of combination of methods for outside, inside and surface points.

The use of the adaptor patterns allows us to make use of the SBT suite for our new solids.

3.2. Data Analysis and Performance (DAP)

We implemented extensions to the SBT suite in order to realize a comprehensive Data Analysis and Performance (DAP) platform for testing the results of each method between different implementations. The core concept of DAP is testing each element in the Unified Solids library in direct comparison with Geant4 and ROOT implementations. The final testing suite allows automatic verification of correctness and performance results. It also represents the core part of the testing suite for the Unified Solids library. The approach used for the testing is divided into two phases:

1. Simulation phase
    - Select a geometrical primitive (solid).
    - Create a set of tests points for it, spanning inside, outside and surface points, and a set of direction vectors.
    - Test all available implementations of the solid, from Geant4, ROOT and the new library.
    - Execute all methods, using the pre-calculated sets of points and, where relevant, directions.
    - Store all results data produced, organized in separate data sets.
    - Perform time measurements of each method.
2. Data analysis phase
    - Visual and numerical analysis of differences of results from Geant4, ROOT and Unified Solids.
    - Evaluation of values and visualization of stored scalar and vector data sets.
    - Creation of 3D plots allowing analysis of all or parts of the data sets produced in the simulation phase (e.g. figure 3 and 4), including the visualization of the shape under investigation (e.g. figure 5).
    - Generate graphs with comparison of performance for each relevant method for solids interface.
    - Visualization of scalability graphs for performance based on concurrent analysis of data sets.

Support is provided for batch scripting in the simulation phase to allow for detailed configuration of conditions in the test.

The analysis enables the exploration of regions of data, to focus on sub-sets of data. This enables us to visualize problematic points and vectors, and identify problematic cases.

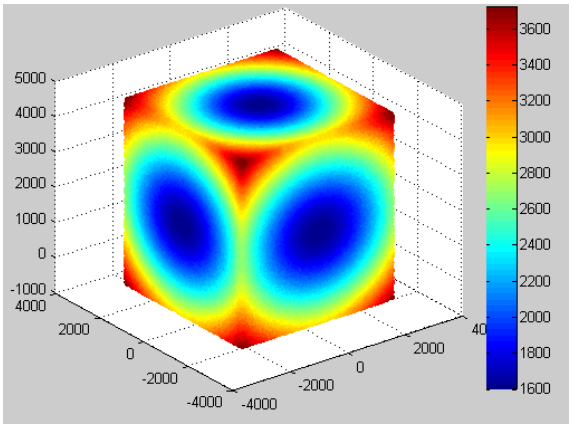

Figure 3. Visualization of *SafetyFromOutside,* the closest distance from outside of a polycone solid

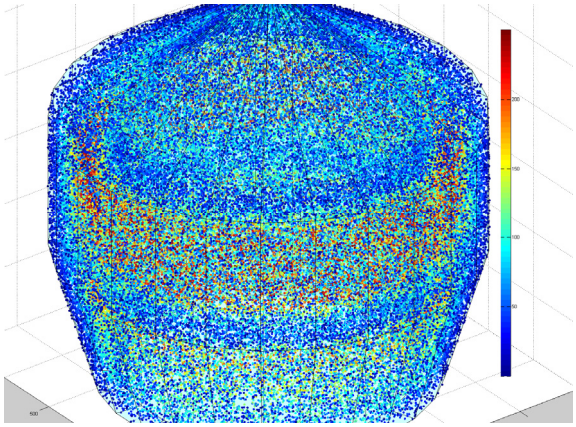

Figure 4. Visualization of closest distances from points located inside of a polycone solid

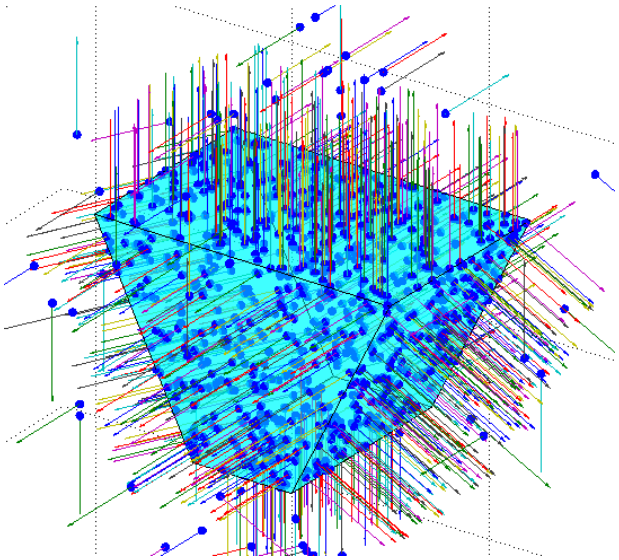

Figure 5. Visualization of a trapezoid solid with normal vectors using SBT DAP.

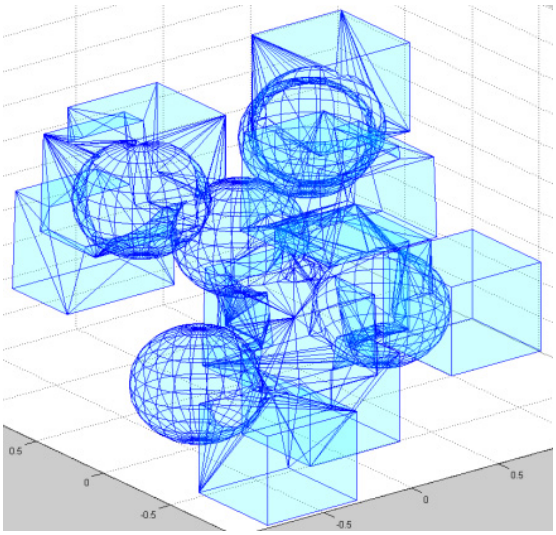

Figure 6. Visualization of the new multi-union construct in a prototype of DAP post-processing.

3.3. Optical Escape

An additional test, derived from the Geant4 geometry modeler testing-suite is the Optical Escape test. The test simulates the propagation of optical photons inside a volume of a given shape. All the internal surfaces are reflective, so that photons cannot escape (see figure 7). Any exception indicates an implementation error.

Using the Geant4 adapter classes makes it possible to cross-compare the implementations. Optical photons are generated and reflected on the inner surface of the solid, with the aim to easily identify error conditions of tracks exiting the solid. We use Optical Escape (and additional tests employed in Geant4 and ROOT) as a supplemental test to complement the DAP suite.

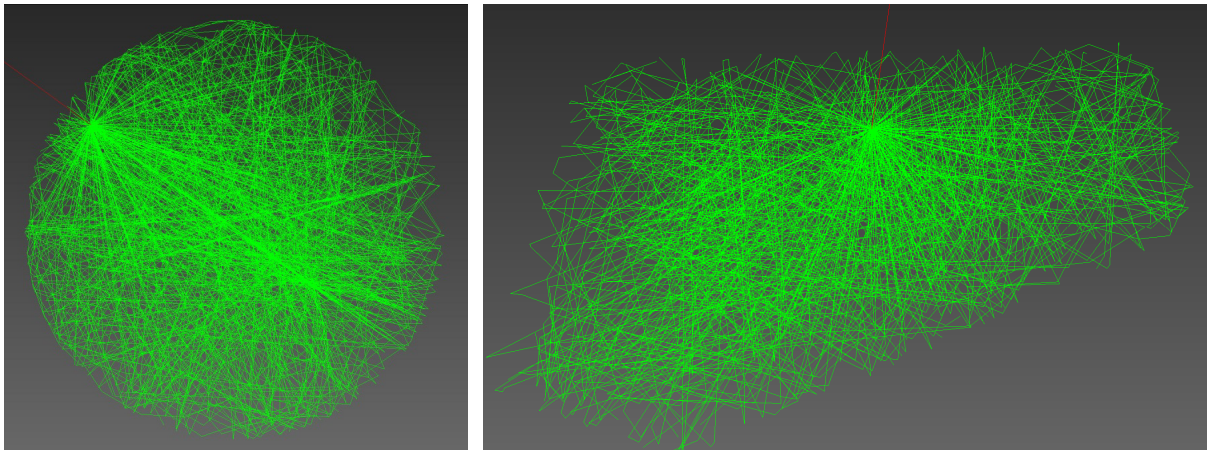

Figure 7. Successful Optical Escape test will show no particles escaping the solid, such as in cases of tests of an orb (left) and a trapezoid (right).

4. Multi-Union solid

A new solid has been implemented with the purpose of providing significant performance improvement in comparison to the Geant4 and ROOT implementations. This is the Multi-Union solid, which represents the union of a set of solids.

ROOT and Geant4 have the capability of composing a solid from several others using Boolean operations. In both cases, the three basic Boolean operations are supported (union, intersection and subtraction), but only binary operations are used in the implementation. As a result, a binary tree of Boolean operations is required in order to represent a complicated composition.

Boolean solids expand the shapes available for modeling volumes and geometry scenes, beyond what could be done using only CSG solids [9, 10]. A complex solid can be created using several Boolean operations, which can be represented as a binary tree. In such an implementation, answering most navigation queries requires that all (or most of) the constituent solids be queried. The cost scales linearly with the number of constituents – which provides poor performance for solids composed of many parts.

The multi-union solid addresses the frequent case of multiple constituent solids built up using the union operation. Figure 8 shows two simple examples of the usage of a Boolean Union Solid. We implemented the multi-union solid using voxelization techniques [11] to optimize the speed. The voxelization uses a 3D space partition for fast localization of components. It provides for the ability to determine which sub-solids are localized in a delimited portion of the partitioned space. This restrains the search to a subset of solids, those that are candidates to be considered in particle tracking algorithms and exclude the rest, which do not need to be considered because of their position.

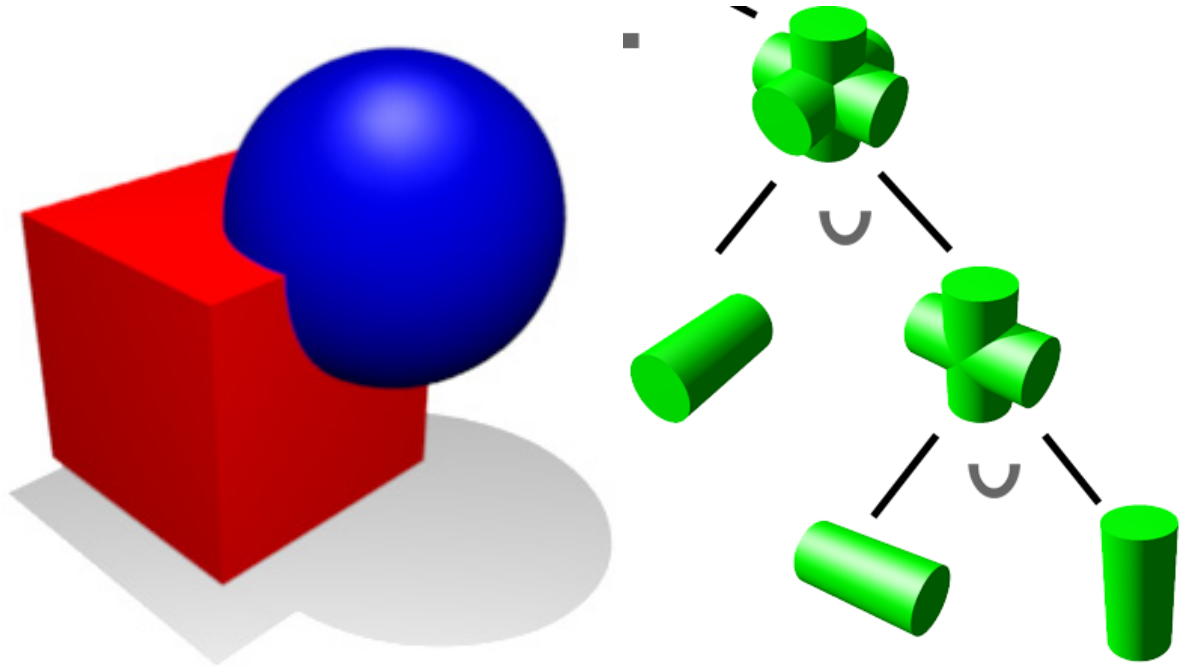

Figure 8. Boolean Union solid: is composite of two solids, either primitive or Boolean[1].

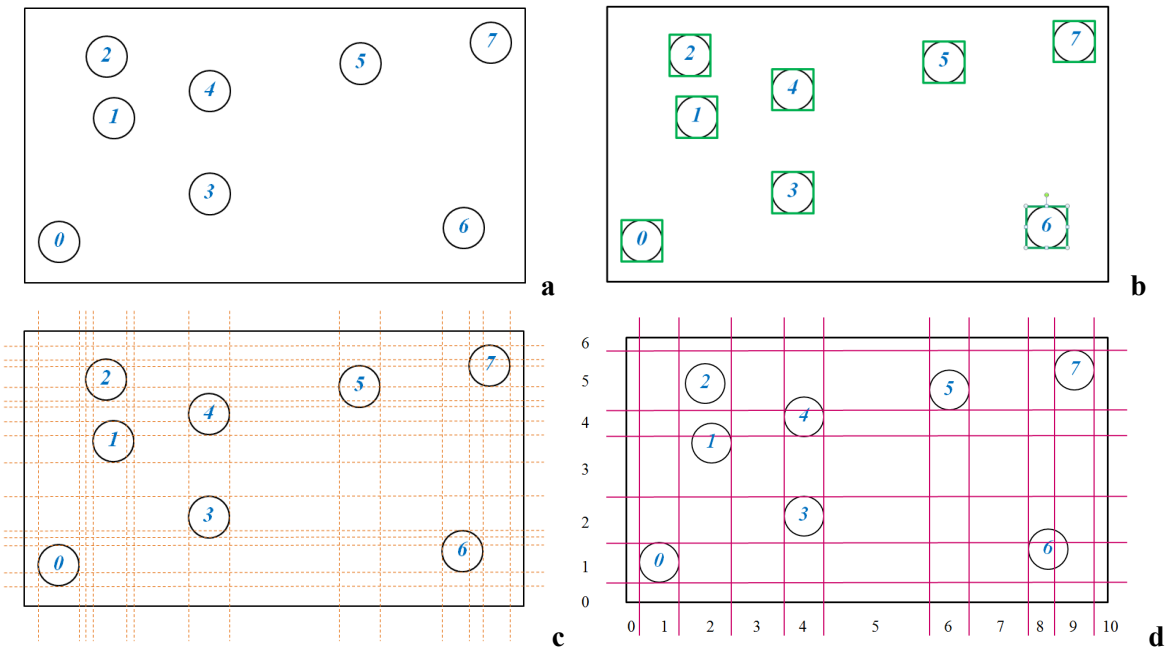

Figure 9. Creation of voxelized space (2D simplification).

[1] The pictures were produced by users of Wikipedia "Captain Sprite" and "Zottie" and are available under Creative Commons Attribution-Share Alike 3.0 Un-ported license

4.1. Voxelization

The multi-union implemented using voxelization techniques aims for logarithmic scalability. The voxelization is realized through several sequential steps (see below). We give an illustration of these steps on an 2D simplified case consisting of eight circular nodes, as on figure 9a.

1) The coordinates of the bounding boxes are calculated for each node as in figure 9b. Note that we increase this extent by the tolerance used for surface thickness.

2) Based on bounding boxes, the limits are computed on each axis, as depicted on figure 9c.

3) These limits are ordered for each axis, and those found to lie within a numerical tolerance of a previous value are removed. This is done to reduce memory consumption and improve performance. Figure 9d depicts the result of such removal.

4) Slices are defined as the space between two successive boundaries along one axis. Any solid at least partly located within a given slice is then recorded. We store this information using binary masks. The number of bits set corresponds to the number of solids present in that slice of the axis, see figure 10.

To determine which solids are contained in given voxel we use the bitmasks corresponding to this voxel for each axis. By making a bitwise *AND* operation on each masks of slices for the three axes we will obtain the final mask filled with bits corresponding to sub-voxel. This identifies the candidate solids to be used for location or navigation.

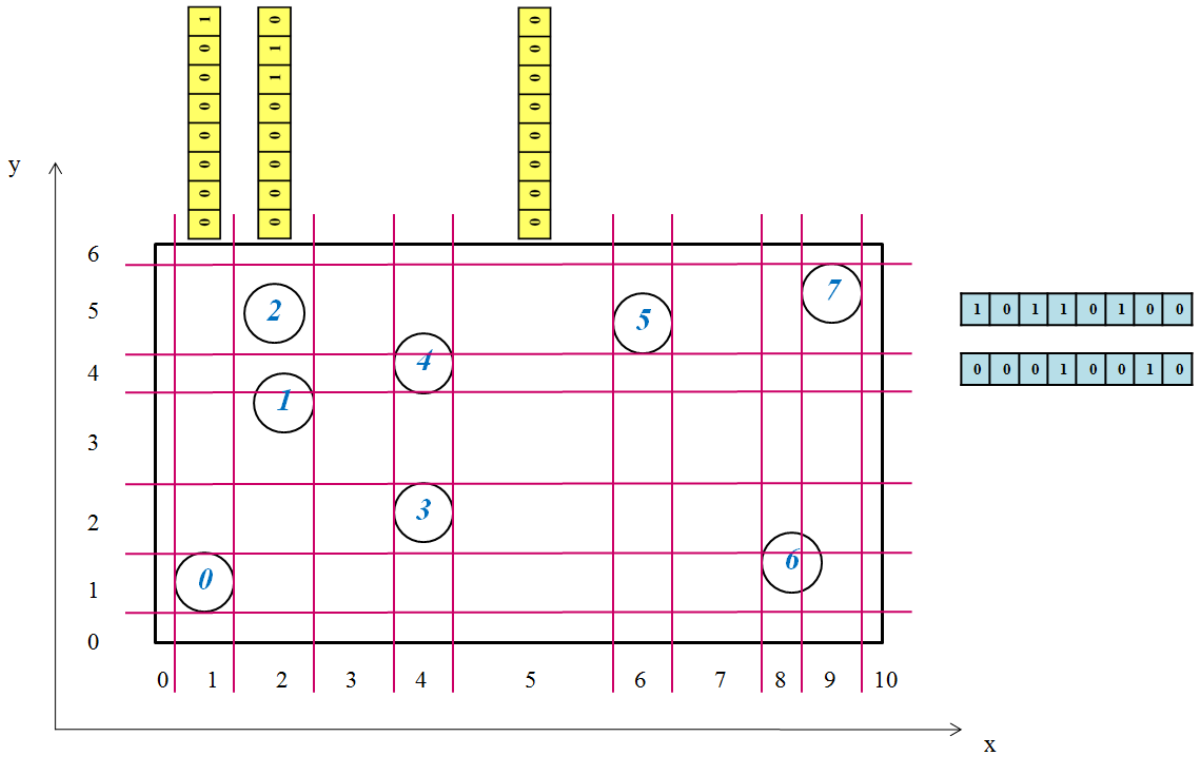

Figure 10. Bitmasks of the axes marking contained solids.

4.2. Performance and scalability tests of multi-union

The first test was developed to check the performance scalability of the method *Inside*. This test was performed on multi-unions with a total number of random boxes ranging from 2 to 10.000, see figure 11.

The volume of the boxes being joined was reduced as the number of boxes was increased. We attempt to fill a fixed ratio (around 0.5) of the filled and empty volume parts using the union of the boxes, ratio of inside vs. outside points is a bit lower than ½, as boxes can collide each other.

The test indicated that optimized version of multi-union offers same or better performance as ROOT Boolean union already from 3 boxes. While the traditional Boolean union offers only linear scalability, the performance of multi-union in these tests is logarithmic, see figure 12.

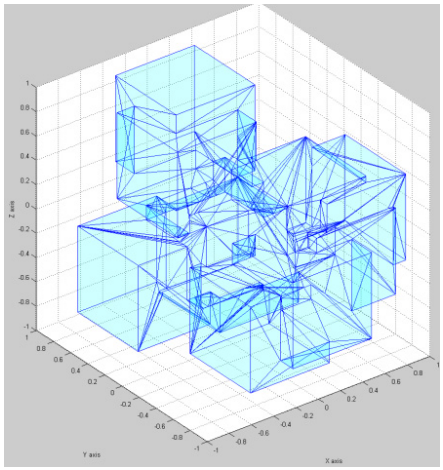

Figure 11. Example of a test solid, composed of 10 random boxes.

Figure 12. Performance scalability of Multi-Union inside method in comparison with Boolean union implemented via a binary tree in ROOT.

A second test of performance has been made on the most critical methods (*Inside*, *DistanceToIn* and *DistanceToOut*), comparing with the Geant4 and ROOT implementations. The tests run on many multi-union configurations, composed with randomly generated and positioned orbs, boxes and trapezoids, having total number of solids ranging from 2 to 100, see figure 6. Performance of multi-union is better starting from 5, 10 and 4 composing solids for the three corresponding methods, as indicated in figure 13.

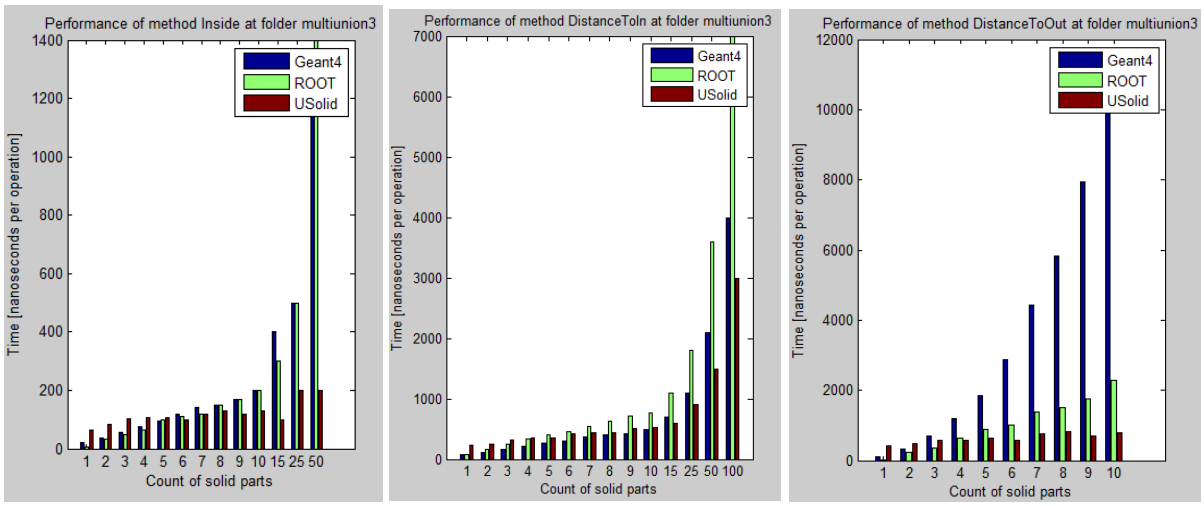

Figure 13. The performance of most critical methods on the Multi-Union solid composed of random orbs, boxes and trapezoids.

These results mean that a large number of use cases can benefit from the improvements. Today solids have been created in HEP experiments, by subtracting several small volumes from a larger one, to create a large complex shape. The factory classes, which generate this description from an experiment specific source, can be revised to create one multi-union solid, and subtract it from the original large volume. In addition tools are used in different application domains, which create Boolean solids from a large number of underlying volumes. The availability of multi-union will enable these advanced users to benefit from a significant performance increase, once they adapt their tools.

The existing implementations of Union remain the best performing for a small number of solids. We foresee that the better of these be retained and adapted, while it can provide such a benefit.

5. Status

We have defined the types and interfaces for the Unified Solids library. Adapter classes have been implemented for Geant4 and ROOT. The geometrical shapes implemented at the time of writing are Orb, Box, Trapezoid and Multi-Union. An implementation is in progress of an optimized version of tessellated solid (solid of arbitrary shape, enclosed by a set of triangular and quadrangular facets). A comprehensive testing suite has been defined, deployed and has been extended with the Data Analysis and Performance module which undertakes direct comparisons of functionality and performance against the Geant4 and ROOT implementations.

6. Conclusion

The Unified Solids library is a new software library of geometrical primitives for solid modeling in Monte Carlo detector simulations and is under preparation. Its aim is to replace and unify current geometrical primitive classes in the Geant4 and ROOT software packages. We have designed interface and adapter classes and dedicated test suites for validating the adapted and new implementations. These include performance and numerical consistency tests to help in choosing the best existing algorithms and to verify the correctness of the code. We have implemented a limited set of existing solids in the Unified Solids library. A new multi-union solid implementation offers dramatically better performance compared with the binary tree-based implementations in Geant4 or ROOT. These serve to demonstrate the potential benefits of a unification of the geometry modeling of use by complex geometrical setups, such as those used in HEP detectors.

Acknowledgments

We would like to thank our colleagues Pere Mato Vila and Astrid Münnich for useful comments and discussions.

References

[1] Agostinelli S *et al.* 2003 *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **506** 250 – 303 ISSN 0168-9002 http://www.sciencedirect.com/science/article/pii/S0168900203013688

[2] Allison J *et al.* 2006 *IEEE Transactions on Nuclear Science* **53** 270 –278 ISSN 0018-9499

[3] Cosmo G 2004 *Nuclear Science Symposium Conference Record, 2004 IEEE* vol 4 pp 2196 – 2198 Vol. 4 ISSN 1082-3654

[4] Brun R and Rademakers F 1997 *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **389** 81 – 86 ISSN 0168-9002 new Computing Techniques in Physics Research V http://www.sciencedirect.com/science/-article/pii/S016890029700048X

[5] Bielajew A F 2000 Fundamentals of the Monte Carlo method for neutral and charged particle transport http://www-personal.engin.umich.edu/bielajew/MCBook/book.pdf

[6] Apostolakis J 2011 *Detectors for Particles and Radiation. Part 1: Principles and Methods, Landolt-Börnstein - Group I Elementary Particles, Nuclei and Atoms, Volume 21B1. ISBN 978-3-642-03605-7. Springer-Verlag Berlin Heidelberg, 2011, p. 320* (*Landolt-Börnstein - Group I Elementary Particles, Nuclei and Atoms* vol 21B1) ed Fabjan C W and Schopper H (Berlin, Heidelberg: Springer Berlin Heidelberg) p 320 ISBN 978-3-642-03605-7 http://adsabs.harvard.edu/abs/-2011LanB.21B1..320A http://www.springermaterials.com/docs/info/978-3-642-03606-4_11.html
[7] Chytracek R, McCormick J, Pokorski W and Santin G 2006 *IEEE Transactions on Nuclear Science* **53** 2892 –2896 ISSN 0018-9499
[8] Gamma E, Helm R, Johnson R and Vlissides J 1995 *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading, MA: Addison-Wesley)
[9] Foley J, van Dam A, Feiner S and Hughes J 1995 *Computer Graphics: Principles and Practice, second edition in C* (Addison-Wesley Professional) ISBN 978-0201848403
[10] Shapiro V 2001 Solid modeling Tech. rep. Spatial Automation Laboratory, University of Wisconsin - Madison 1513 University Avenue Madison, WI 53706-1572 http://sal-cnc.me.wisc.edu/-index.php?option=com_remository&Itemid=143&func=fileinfo&id=53
[11] Kaufman A, Cohen D and Yagel R 1993 *Computer* **26** 51–64 ISSN 0018-9162