Royal Holloway University of London

PH4100: Major Project

Meshing of Primitive Solids in pyg4ometry & BDSIM

Ben Shellswell

Abstract

When testing new concepts and devices within particle physics it is often a very expensive and time consuming process. The software packages pyg4ometry & BDSIM are designed to enable scientists and people within in the industry to virtually simulate these tests, with accurate physics concepts. This project looks at improving the 3D simulation of the events and devices, by remeshing the basic primitive solids.

Supervised by
Prof. S Boogert
February 4, 2020

Contents

1 Introduction

1.1 Project Aims

The aims of this project are to contribute towards the optimization of the pyg4ometry package 2.2 (and subsequently BDSIM 2.1), by improving parts of the code and conducting performance test to produce results that can be analysed. The main areas for improvement and where most of the computational energy in wasted, is in the meshing of the primitive Geant4 2.3 compatible solids, due to the unessecary use of boolean operations 3.3.2.

1.2 Report Structure

The subsequent sections are constructed in the following way, the software packages that are used and referenced through out this report (Section 2), the concept and details of the primitive meshing used in pyg4ometry (Section 3), then a conclusion and summary of the results of the report (Section 4).

2 Software Packages

This section goes through each package of software related to and used throughout the duration of the project. It outlines the key details of each package, describing its function and link to the project.

2.1 BDSIM

BDSIM (or Beam Delivery SIMulation) is a open source software package written by the John Adams Institute (JAI) [**?**], for the use of modelling particle beam interactions. BDSIM has many applications, such as modelling complex particle accelerators for example the Large Hadron Collider (LHC) or concepts magnets for medical scanners used to treat tumors. The package allows a user to specify the physics being used for a particular particle of a set energy colliding with a provided object. The scattering of the particle trajectories and decays are computed using monte carlo simulations, to make the results as consistent with experimental results as possible. The software outputs a full analysis of each run, and can even allow multiple runs to run at once (batch mode).

2.2 pyg4ometry

pyg4ometry is an open source python package also generated by JAI, its purpose is to convert 3D CAD (Computer Aided Design) models between different representations to allow compatibility with BDSIM for the testing of new concepts. The '4' in 'pyg4ometry' comes from the consistencey the package has with Geant4 2.3. The package is a key tool for allowing multiple file formats to become compatible with BDSIM, which increases the number of people who can utilise the package.

2.3 Geant4

Geant4 (or GEometry ANd Tracking) is a software developed for the simulation and tracking of particles traveling through matter. The package is used by many particle physicits and is one of the more popular options for handiling the geometry within interactions. Geant4 has its own preset solids used for simulating with particle interactions. For ease of conversion between file formats pyg4ometry uses the same conventions when meshing its primitive solids.

2.4 Root

Root files are the default output for analysis by BDSIM, as it has been heavily used by particle physicsts since its release. Root has its own file browsing system due to the nature of its formatting. Root is adopted by many physics communities such as CERN [**?**], where it was first written.

3 Primitive Meshing

This section will describe the work done to optimize the python scripts that generate the three dimensional meshing for the primitive solids within the pyg4ometry package 2.2. All the primitive solids used are constructed such that they are compatible with Geant4's solids. It was orginally thought that it would be best to use trianglar based meshes in combination with boolean operations 3.3.2 to construct the 3D solids. However it has been realised that the computation of triangles and boolean operations compared with polygons and adapted trigonometry is much more intensive and ineffecient, in most cases. In particular with the curved solids, i.e circular and elliptical based solids.

One of the major improvements to the pyg4ometry 2.2 code is the computation of cut up primitive solids. The meshing of hollow or sliced solids were previously computed by Boolean subtractions and unions, which involved creating two separate solids and acting upon both of them. Discussed more in Section 3.3.2. Which resulted in a very computationally heavy and less aestetic outcome, where the mesh lines ('slice and stack'), were not meshed in radial directions.

3.1 Co-ordinate Systems

The various primitive solids are all constructed by using the predefined parameters used by Geant4 2.3, to be consistent with Geant4's own solids. The parameters of a 3D solid are properties relating to the coordinate system it is constructed in, such as height or radius. The parameters are then used to define the points of the object via basic trigonometry.

The python meshing scripts for all coordinate systems follow a similar structure, of first defining an empty list of faces (polygons). Then running the associated trigonometric equations through a number of loops to generate and append polygons to that list. The number of loops is associated with the number of sections a surface of a solid is being split up into in a given coordinate system. The density of the meshing is defined by a user inputted number of slices and stacks, demonstrated in Figure 1.

3.1.1 Cylindrical Co-ordinate System

The meshing for the primitive solids in cylindrical polar coordinate systems are constructed by looping though the user defined number of slices and stacks (as shown in Figure 1) which the cylinder is being cut into (Listing 1). The Loop then creates the coordinates for 3 or 4 points at a time using an adapation to the trigonometry in Equations 1, which can then be defined as a triangular or polygonal face. The only cases where the mesh produces triangles is at the top and bottom faces of the cylinder, provided it does has a minimum radius equal to zero (creating a tube or cone). The same logic for the polygons also applied to triangles, just using 3 vertex points to make a face.

The trigonometry that converts the points from cylindrical polar coordinates to cartesian, are:

$$\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \\ z &= z \end{aligned} \tag{1}$$

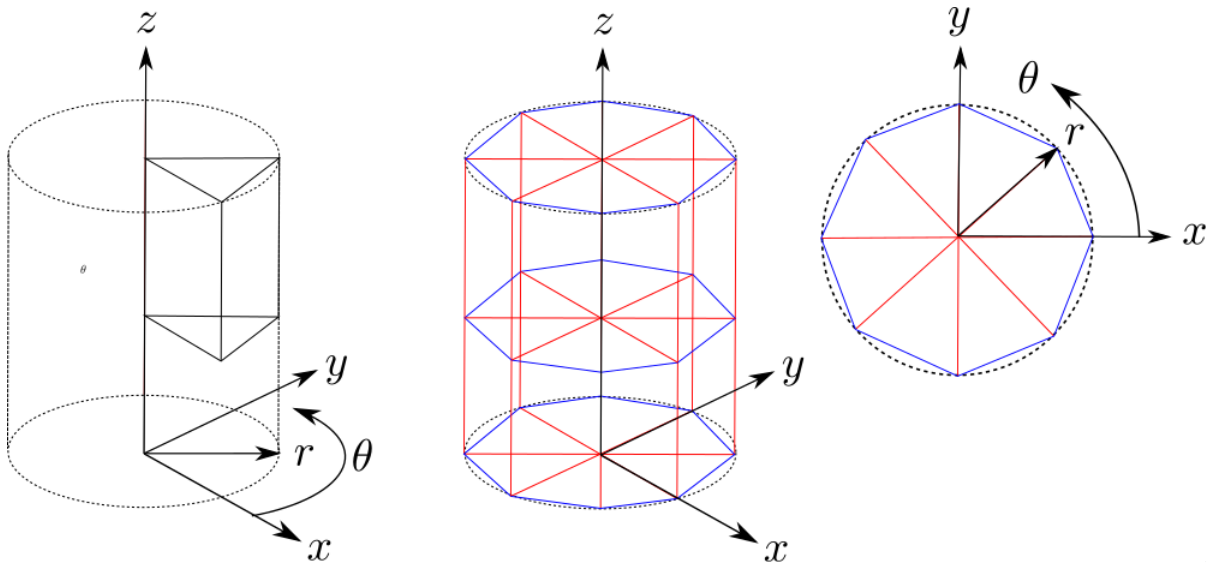

Figure 1: Diagram showing the meshing method for a cylindrical coordinate system
Red = Slices (8)
Blue = Stacks (2)

```
polygons = []

for j0 in range(nslice):
    j1 = j0
    j2 = j0 + 1

    vertices = []

    for i0 in range(nstack):
            i1 = i0
            i2 = i0 + 1
```

Listing 1: Basic method structure for pyg4ometry primitive meshing of solids

The code in Listing 1 generates counters so that you can choses from two slices and two stacks, in order to gain the four points surrounding a desired face. These points are then used to define a polygon.

The only time a stack is needed in the cylindrical coordinate system is when the solid has a non linear function in the r-z plane. For example a paraboild (Figure 24) would need a stack, but a linear cone (Figure 10) would not. This is due to the fact how that a plane can't represent a curved surface with a singe face.

3.1.2 Spherical Coordinate System

The meshing for the primitive solids in spherical coordinate systems are constructed by similar means the that of the cylindrical 3.1.1. Just with different trigonometric equations (Equations 2) as a result of two angle parameters ϕ and θ. The stack (blue) and slice (red) for solids in the spherical coordinate system works, like the longitude and latitude on a globe, as shown in Figure 2.

The trigonometry that converts the points from spherical coordinates to cartesian, are:

$$\begin{aligned} x &= r\cos\theta\sin\phi \\ y &= r\sin\theta\sin\phi \\ z &= z \end{aligned} \tag{2}$$

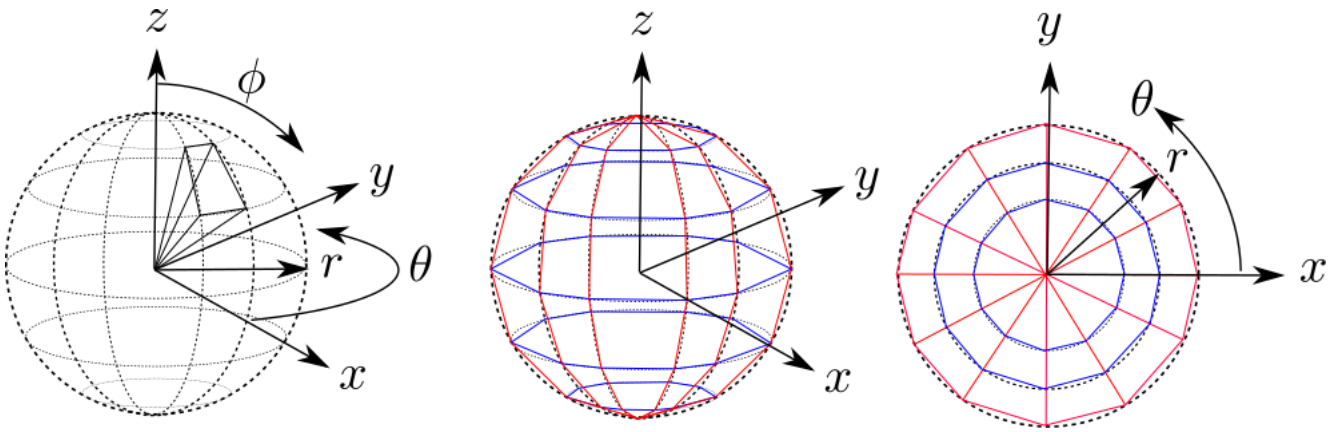

Figure 2: Diagram showing the meshing method for a spherical coordinate system
Red = Slices (12)
Blue = Stacks (6)

The structure of the code is the same as used in Listing 1.

The only time triangles are constructed in the spherical coordinate system is if the solid has a complete pole at the top or bottom of the solid. The solids constructed in the spherical always have both a stack and a slice.

3.1.3 Toroidal Coordinate System

A toridal shape is much harder to visualise a stack and slice, due to the fact it is a rotating coordinate system. A toroidal slice is an R_{Torus} radial cut taken out of the angle ϕ, as shown in Figure 3. The toroidal stack is a R radial cut out of the angle θ.

The trigonometry that converts the points from toroidal coordinates to cartesian, are:

$$\begin{aligned} x &= R_{Torus} + R\cos\theta\cos\phi \\ y &= R_{Torus} + R\cos\theta\sin\phi \\ z &= R\sin\theta \end{aligned} \tag{3}$$

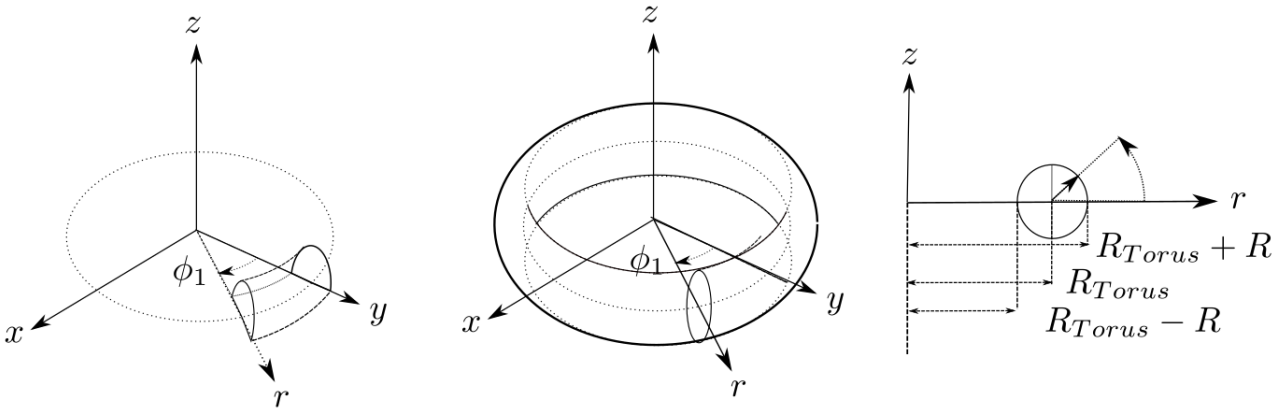

Figure 3: Diagram showing the meshing method for a toroidal coordinate system

3.2 Plane Direction

One key thing to be taken into account is the convention being used in the code for the order in which points are appended to make a plane, i.e to define a face on a solid. This is important as the direction the normal of the plane points in, dictates wether a face is considered an inside or outside face on the given solid. Getting this incorrect, will lead to missing faces, when the meshing is made. The concept is demonstrated in Figure 4.

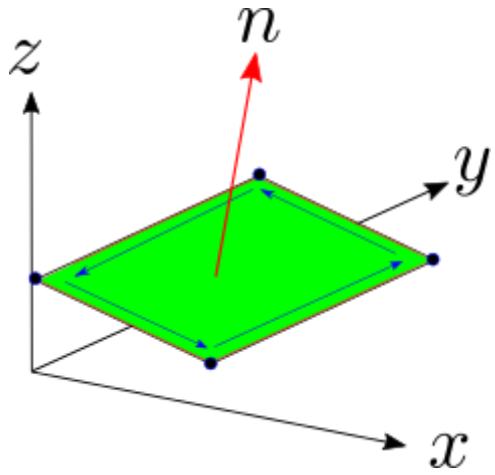

Figure 4: Diagram showing the order convention of appending points to define the normal to a plane

The simplest way to test this is by performing boolean operations with a box, as the boolean operation will only work nicely if all the planes are correct on both shapes.

3.3 New Meshing of Curved Primitve Solids

In total there are 12 curved primitive solids, of which many of the examples and concepts are very similar. Therefore only a few solids will be discussed in this section, but their development can all be viewed in Appendix B.

can give one eaxmple and reference rest in appendix, radial meshing clean up, can remove stack in some cases, boolean slow and messy

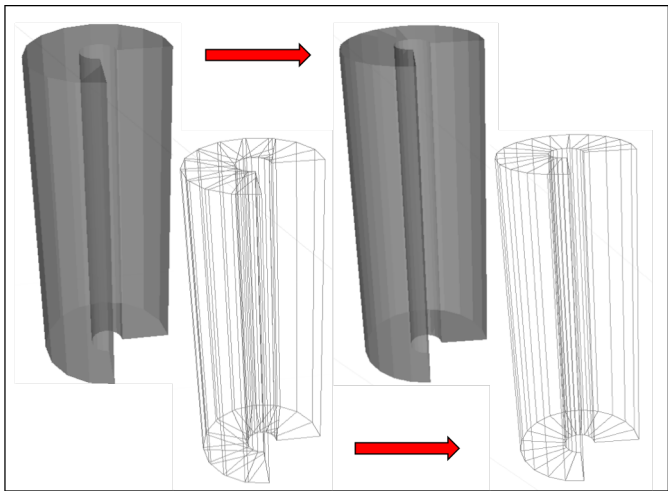

Figure 5: Meshing Development for CutTubs (Solid & Mesh View)

3.3.1 Degenerate points

Multiple meshing points occupying the same area can spring a few errors, sometimes without entirely crashing the code, making it a hard error to identify. I is typically given away when a $DivisionByZero$ error occours, within the pycsg meshing of a solid. You can identify wether this is the error by debugging each polygon and triangle in a mesh, looking out for a face that has two or more vertices with the same (x, y, z) coordinates.

3.3.2 Boolean operations

One of the largest changes to the performance of the new meshing compared with the previous method, is the disgarding of boolean operations in order to create hollow or cut-up primitive solids. The Old meshing algorithurms would make two solids one smaller that the other and subtract it, with the aim of creating a new shape that is hollow. For example Tubs is made from two cylinders being subtracted in order to create a tube as seen in Figure 6. The boolean operations worked, however are very computationally heavy compared with that of some adapated trigonometry. Another thing the boolean operations affected was the appearance of the mesh its self, the boolean operations worked by trying to identify common mesh points and then remeshing. This created a lot of non radially uniform mesh sections as seen in Tubs.

The Figures 7 & 8 are of the meshed boolean union and subtraction of a box with a hollow sphere (in solid view). The coloured lines are representing the perpendicular planes in which the final object is place in.

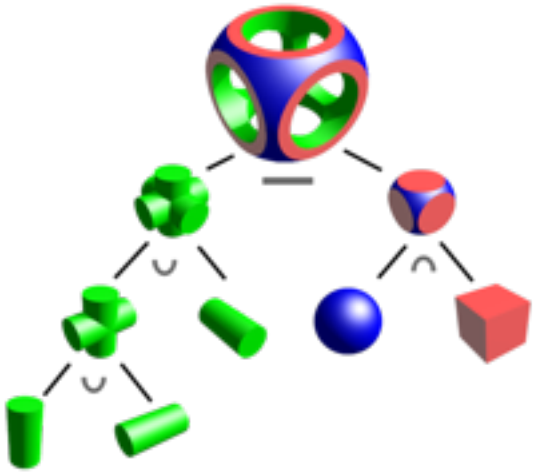

Figure 6: A diagram showing the basic method of constructing a more complicated 3D solid out of boolean operations with simpler primitive solids.

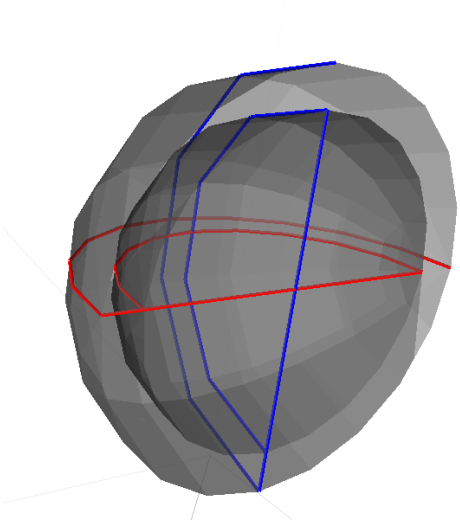

Figure 7: Example screenshoot of a Boolean Union produced in pyg4ometry.

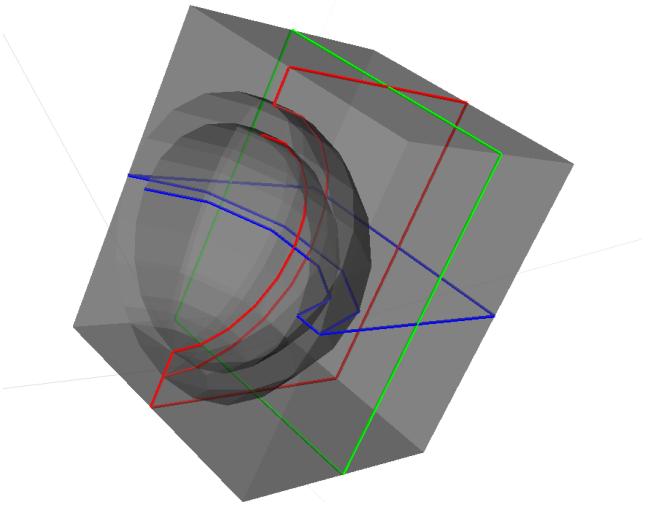

Figure 8: Example screenshot of a Boolean Subtraction produced in pyg4ometry.

3.4 Meshing performance testing

3.4.1 Polygon Count

One way in which the meshing performance of the pyg4ometry primitive solids is conducted, is by counting the number of polygons produced by both the old and new meshings, in order to make a comparison. A plot for each primitive solid counting its number of generated polygons was produced. They were generated by varying the user inputted number of slices across a range. Most were put through the range of (10-100) slice whilst keeping the number of stacks at a constant 10. However a few old solid meshings took so long to produce at higher mesh densities, they were only measured across shorter ranges e.g the old Hyperboloid meshing was left to run for over an hour.

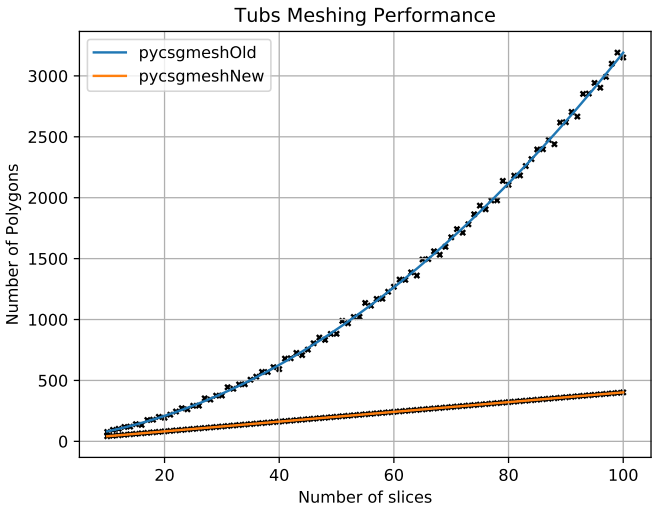

Figure 9: A plot showing the comparison of the number of polygons (and triangles) generated by the new and old meshing methods, across a range of slice 10-100.

3.4.2 BDSIM interactions

Within BDSIM the charge of a particle is represented by its colour, green = nuetral, blue = positive and red = negative. The probability of certain events and the scattering of a particles trajectories is generated by a monte carlo simulation. Each seed that produces a different outcome in the monte carlo simulations has a unique seed number. This allows a particular event to be repeated with the same physics, which is key to this project when it comes to comparing the interactions of two different objects under the same conditions. Other properties of the particle and physics processes that it may undergo can also be user defined to tweak the experiment. Properties such as the particles initial energy, wether secondaries get produced and much more.

Each BSIM event outputs a *.root* file which contains a detailed analysis of the interaction. One of the elements analysed within this project is the energy loss across the Z-axis.

BDSIM works by running a $.GMAD$ file and then outputs *.root* file with the option of a 3D visualisation of the interactions. The $.GMAD$ contains the basic information needed to produce an output, i.e a particle and a target fie (Typically of the format $.GDML$).

4 Conclusion & Summary

4.1 Improvements

meshing is quicker
improved coverage unit test speeds
meshing is neater and more uniform in structure
higher meshing density = closer to tru solid as expected with bdsim interactions

4.2 Applications

BDSIM and pyg4ometry are both very powerful software packages that can be used to to aid not only the scientific research community of particle physicist, but also help everyday people treat patients. Thanks to the software being open source and its wide range of file compatibilities it can be used to simulate a growing number of projects.

A Appendix (Python scripts)

A.1 Sphere BDSIM Vary Mesh Test

```
from string import Template
import Target_Sphere as t
import numpy as np
import pybdsim

################################################################################

def run_gdml_spheres_same_slice_and_stack(min,max):

    """
        generate a .txt wth four lists of data, number of slices & runtimes,
        from a minimum and maxiumum number of slices and stacks
        """

    Meshing_ver = "New"

    for val in range(min,max+1):

        # create gdml
        t.Test(False,False,n_slice = val,n_stack = val)

        #make gmad
        f = open('Template.gmad','r')
        contents = f.read()
        f.close()
        template = Template(contents)
        d = {'value': str("gdml:../../GDMLs/"+Meshing_ver+"/Target_Sphere_"+str(val)+"_"+str(val)+".gdml")}
        rendered = template.substitute(d)
        gmadfilename = "GMADs/"+Meshing_ver+"/slice_"+str(val)+"_stack_"+str(val)+".gmad"
        f = open(gmadfilename, 'w')
        f.write(rendered)
        f.close()

        # use gmad and gdml to get root output
        pybdsim.Run.RunBdsim(gmadfilename,"root_outputs/"+Meshing_ver+"/"+str(val)+"_"+str(val))

        #load in root file to do analysis ...

        print "{}%".format(val-min+1//(max-min))

################################################################################

run_gdml_spheres_same_slice_and_stack(10,100)
```

Scripts//Run_New_Meshes.py

B All Meshed Solids and Polygon Count Plots

The following Figures are the meshing and polygon data for each primitive solid constructed in pyg4ometry 2.2. The first Figure for each solid is screenshots of the old meshing and new meshing visualized in VTK. They show the before and after of each primitive solid in both "solid view" and "mesh view". The second Figures Show the number of polygons and triangles produced by the solid as you increase the slice across a range of 10-100 (if there is a stack it is kept at a constant 10). The polygon data plots are generate using python 2.7. The naming convention is the one used by Geant4 2.3.

B.0.1 Cons

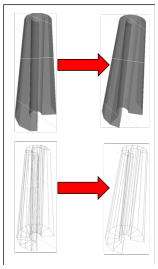

Figure 10

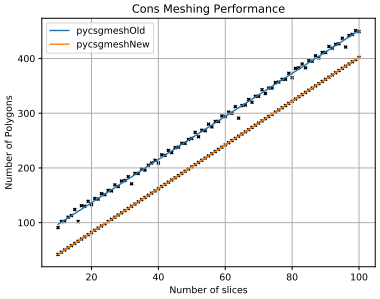

Figure 11

B.0.2 CutTubs

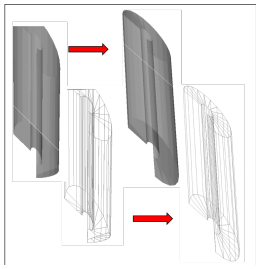

Figure 12

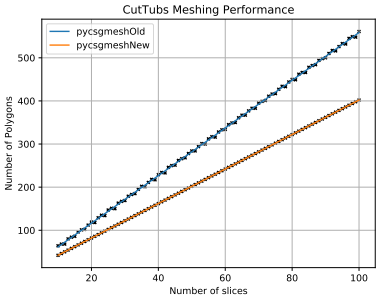

Figure 13

B.0.3 Ellipsoid

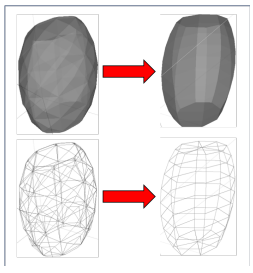

Figure 14

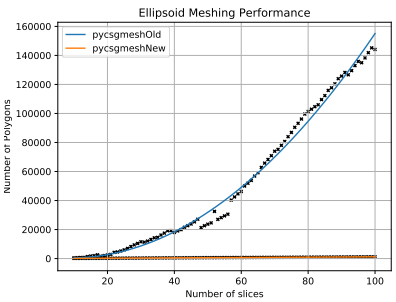

Figure 15

B.0.4 EllipticalCone

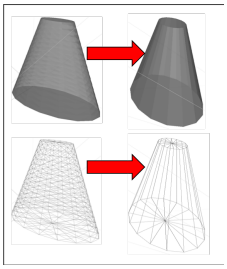

Figure 16

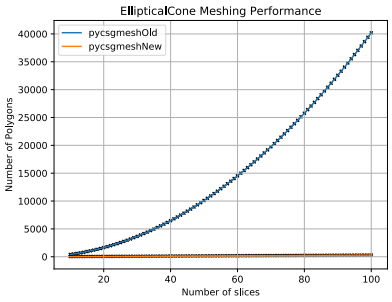

Figure 17

B.0.5 EllipticalTube

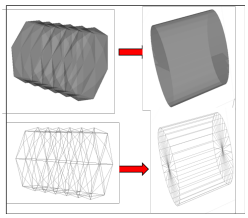

Figure 18

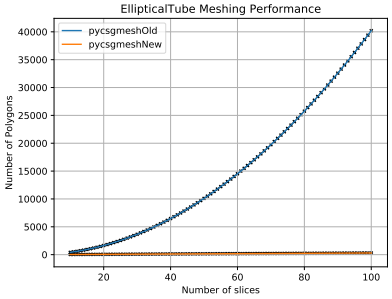

Figure 19

B.0.6 Hyperboloid

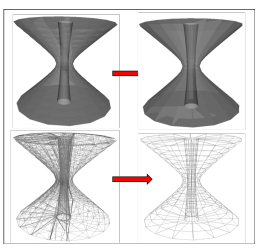

Figure 20

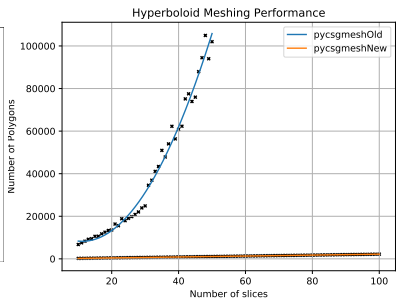

Figure 21

B.0.7 Orb

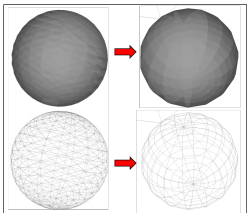

Figure 22

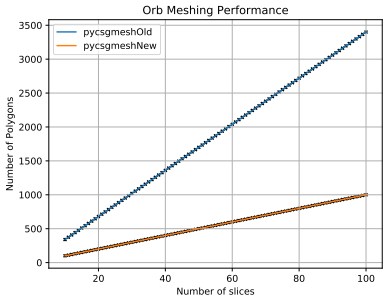

Figure 23

B.0.8 Paraboloid

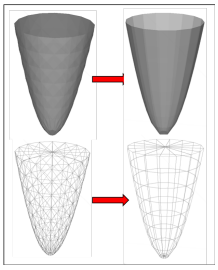

Figure 24

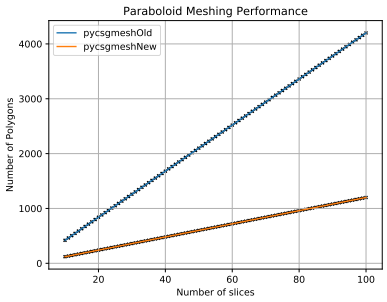

Figure 25

B.0.9 Polycone

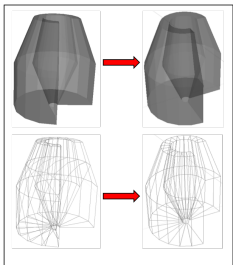

Figure 26

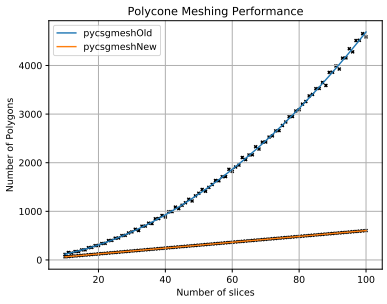

Figure 27

B.0.10 Sphere

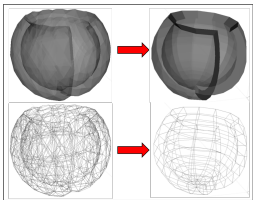

Figure 28

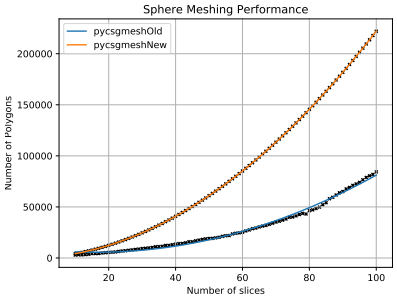

Figure 29

B.0.11 Torus

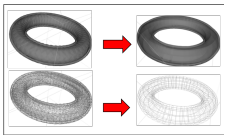

Figure 30

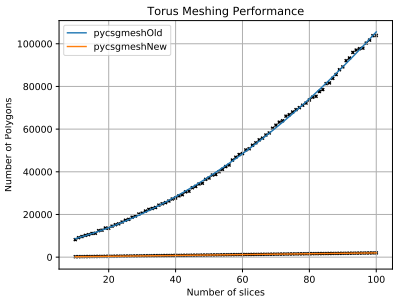

Figure 31

B.0.12 Tubs

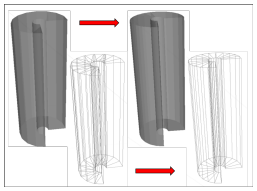

Figure 32

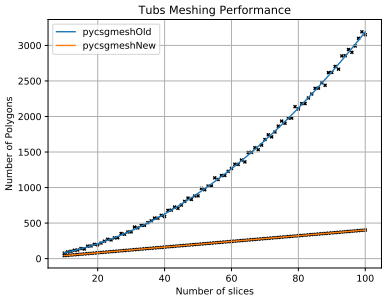

Figure 33

References

[1] BDSIM Manual
http://www.pp.rhul.ac.uk/bdsim/manual/

[2] Pg4ometry BitBucket
https://bitbucket.org/jairhul/pyg4ometry/src/

[3] Geant4 Solids
http://geant4-userdoc.web.cern.ch/geant4-userdoc/UsersGuides/ForApplicationDeveloper/html/Detector/Geometry/geomSolids.html

[4] JAI
https://www.adams-institute.ac.uk/