NanoShaper 0.7 Guide

Sergio Decherchi
Fondazione Istituto Italiano di Tecnolgia

November 5, 2013

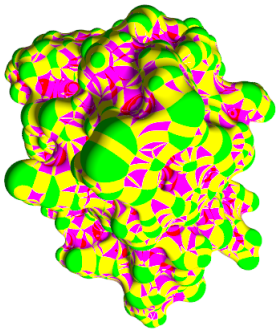
Analyzing molecular surfaces it is a key issue in biophysical modeling. The aim of NanoShaper is to measure molecular properties such as the volume, the surface area and cavities, build a Finite Difference grid for PDE solution, compute pockets and to triangulate the molecular surface; this is achieved by employing ray-casting [4, 2].
Ray-casting here is used as a tool to inspect, in a grid based world, the inner part of the surface. This allows to build an in/out map by which one can perform floodfill to identify cavities, estimate their volume, and fill them if requested. The software can be used also with non molecular surfaces provided that they are manifold. The algorithm can deal with analytical or meshes surfaces. The algorithm is robust to duplicated vertices/faces nad almost degenerate triangles. Volume is estimated by a triple ray-casting process using as starting points of the ray the x,y,z sides of the cube that contains the molecule; the three obtained values of the volume are averaged thus getting an highly accurate result.
In order to estimate the surface area an ad hoc parallel Marching Cubes algorithm has been developed in order both to triangulate the surface and to accurately perform surface area estimation [4]; moreover if voids are removed the final triangulation is consistent with cavities removal. All the framework is developed in portable, expandable C++.
One can load a .off/.ply triangulated mesh, a msms file [5] and build the Skin [3], Blobby [1] and the SES (Connolly) molecular surfaces.

1 What's new

In this version new tools and several improvements are present:

- This version of NanoShaper, using the DelPhiPatcher, can be interfaced to DelPhi Fortran 77 solver. From DelPhi all the necessary structures are imported to let solve the PB equation on th grid colouring induced by NanoShaper. PDB data is also imported such that additional atoms info is available.

- The first grid operator is introduced, namely the minus operator. By grid operator we mean a function that takes as input one or more grids



(surf.u3d)

Figure 1: Example of Skin surface ray-traced by PovRay and triangulated (interactive)

and produces an output grid. The minus operator is introduced because it is used to support pocket/cavity detection. To support cavity/pocket detection the keyword Operative_Mode is introduced. When set to pockets pockets detection is carried instead of the usual flow.

- Normals to vertices can be computed, analytically where possible. Meshes can now be saved in the, here introduced, OFF+N, OFF+N+A formats. These new formats save the nearest atom index (A) and the normals (N). Now meshes can be saved in MSMS format too when explicitly requested.
- Octree are now used when collecting the analytical intersections for the triangulation. This leads to a significant memory saving, in particular when an high resolution is used for triangulation.
- A parallel version of the Marching Cubes algorithm is present. It is a major change with respect to the previous version in that the algorithm as been significantly reformulated; often it gives a linearly scaling performance.
- Load balancing during ray-tracing is significantly improved leading to an improved performance.
- The floodfill algorithm for closed cavity detection can lead to signficant performance improvements (4x) in particular when the number of detected cavities is relatively small (< 10).
- An algorithm to convert a mesh to a set of approximating atoms (balls) is developed. This can be useful for those codes which, for instance, use as input only a set of atoms (e.g. DelPhi Fortrant 95 code) and are not able to directly manipulate an arbitrary mesh.
- The classes that derive from Surface can be very easily registered. It is sufficient to add the .cpp and .h files of the new surface in src directory

and recompile to make the surface available to the rest of the framework. To make clear how the surface derived classes should be written the ExampleSurface class is provided.

2 Compilation/Installation

NanoShaper can be automatically configured and compiled with the setup.py script in Linux/Mac or compiled and installed manually. For Windows user an executable is in bin directory. For optimal performance Linux is suggested. To run NanoShaper type the NanoShaper executable followed by the configuration file. Remember that NanoShaper needs at least 4 atoms to work: if you have less atoms put the missing atoms with 0 radi on the same position of the original atoms. Here the two procedures for compiling NanoShaper.

2.1 Automatic mode for Linux/Mac

In a Linux shell run:

```
python setup.py
```

This will try to automatically guess your distribution, download and install the necessary packages accordingly, download CGAL, compiling CGAL, patching CGAL, configuring and compiling NanoShaper as Stand Alone (.exe), as a shared object for DelPhi, or as a Python module. You will find NanoShaper in the build folder.

2.2 Manual mode for any OS

In order to build the full version of the software the system must have installed CGAL (for the Skin/SES support CGAL 4.2 was used), Boost libraries and cmake; In Linux/Mac GCC is required and in Windows NanoShaper it has been tested on Visual Studio 2008/2010 but it should work also on next versions. Building the software consists in a cmake configuration step and make; additionally patching CGAL is required (for Visual Studio 2010 also Power_test_3.h is required as patch); prior to building, overwrite the files contained in CGALpatch in the CGAL include directory where the same .h are present. For simplicity and to minimize the number of packages to be installed use the following cmake command for CGAL:

```
cmake . -DWITH_examples=false -DWITH_CGAL_Qt4=false
 -DWITH_CGAL_Qt3=false -DWITH_CGAL_ImageIO=false
```

Once CGAL is compiled to build the configuration for the software go to NanoShaper build folder and type cmake ... If you are on Windows and on a 64 bit machine perform cmake by expliciting setting the current compiler and 64 bits build to avoid a cmake bug.
On a Windows system you will get a Visual Studio project and Linux/Mac a

make file. Build either typing make or using Visual Studio.
If CGAL is not present, mesh, msms and Blobby surfaces will be still available and the code should be compilable with any C++ compiler that has boost support. If neither boost is present the same functionalities will be given but using a single execution thread; in this last case the software does not depend on any external library except from STL containers.
NanoShaper can be directly used in Python: to this aim you will need the Swig package installed (Enthough Python distro already has). Rename *CMakeLists_python.txt* into CMakeLists.txt and run cmake as in the previous case but this time inside build_python folder. This process will generate a project or a make file; by building the project/or performing make you will be end up with the following files: NanoShaper.py and _NanoShaper.pyd. You should put these files in a place where they can be reached by Python such as the same folder where you write your script: then you can use NanoShaper classes as for any other Python package by importing NanoShaper. In the folder python_example see the file example.py where the NanoShaper classes are used and the surface is visualized via MolFX.py script.

3 Usage

3.1 Triangulation

The first step for computing the molecular surface is obtaining an input file. NanoShaper as input format for atom positions/radii uses .xyzr files, where every line represents an atom and each column represents the x,y,z coordinate and the last one represents the radius expressed in Angstrom.
We developed a Python (thus portable) script, called pdb2xyzr.py, to support the conversion of a pdb file into a .xyzr file; this script (as all the scripts/programs that will be discussed) is freely downloadable from:

www.eletrostaticszone.eu

by this script the user can either give a pdb code ot the path to the pdb file; in the first case the pdb will be downloaded from the pdb repository in the second it will be accessed from the disk. This script assigns radii based on a .siz file that represents a database of radii for proteins; for those users that are familiar with DelPhi, these are the same database files used by DelPhi to assign radii. This script uses the VMD atom selection string such that the user can easily filter the interesting part of the protein; the selected part will be also saved in pdb format. To get help it is sufficient to run the script without arguments.
Suppose that we want to download the Purine Nucleoside Phosphorylase monomer (code 1RSZ) and get only the protein part. We would run:

```
python pdb2xyzr.py 1rsz protein
```

On disk we will get the file 1rsz.xyzr and the pdb selection_1rsz.pdb. Now we could protonate this last file with tleap from AmberTools. It is sufficient to run tleap executing the following commands:

- Start tleap by tleap. Assure to have AmberTools home in the path.
- Load ff99SBildn parameters by source leaprc.ff99SBildn
- Load the pdb, A = loadpdb selection_1rsz.pdb
- Now the protein has been protonated; save it by savepdb A 1rszH.pdb

Now we can re-run pdb2xyzr.py to get the final input file:

python pdb2xyzr.py 1rszH.pdb all

The final input file is named 1rszH.pdb.xyzr.
The only additional file necessary for NanoShaper to run is the configuration file. NanoShaper, if run without arguments, will search the default name surfaceConfiguration.prm; alternatively the user can run NanoShaper giving as argument the configuration file name. NanoShaper configuration file uses the # character at the beginning of the line to define a comment; the case sensitive keywords are all of the type:

key = value

where key is a word without blank spaces and value can be either a string (a file name, true/false) or real/integer value.
Supposing that the input file is in the current directory in the configuration file we will write:

XYZR_FileName = 1rszH.pdb.xyzr

By default NanoShaper will save the mesh in Geomview .off format in the file triangulatedSurf.off. If the user has an Ubuntu distribution he can download Geomview by apt-get and easily visualize it. The position of the vertices is very accurate because they are analitycally sampled by the ray casting routine but the quality of the surface triangles is modest because NanoShaper uses the Marching Cubes triangulation rules; to improve the quality of the mesh the user can instruct NanoShaper to smooth the output mesh; this can be done by writing the following line in the configuration file:

Smooth_Mesh = true

The triangles quality signficantly improves. NanoShaper allows to save in the MSMS format, this can be done by adding the following two lines:

Compute_Vertex_Normals = true
Save_Mesh_MSMS_Format = true

Normals will be computed analytically where possible; if some ray fails the missing normals will be approximated by averaging the normals of the triangles surrounding the vertex. Thus the mesh will always have all the normals defined. The saved files are named triangulatedSurf.vert and triangulatedSurf.face. This format can be read by VMD.
To have a practical idea of the NanoShaper hardware requirements we triangulated at a scale of 2.0 $Å^{-1}$ the 1sva entry, namely the Simian Virus 40, for a total of about 1 million atoms. The test succeded with a peak memory consumption of 10 GB of RAM. Globally, excluding the time needed to save on disk the mesh, NanoShaper needs about 2 minutes to triangulate the Simian Virus 40 (normals included) on a two 8-cores SandyBridge workstation. NanoShaper exploits well parallelism so multi-cores CPUs should be used when possible.

3.2 Cavity Detection

NanoShaper can detect closed cavities inside the surface under analysis and return an estimation of their volume. NanoShaper uses iteratively a floodfill algorithm to detect cavities and to this aim builds a map to mark in/out and the cavity id. To enable and run cavity detection the user has to turn on the following keywords:

Cavity_Detection_Filling = true
Build_status_map = true

The cavity will be filled if the volume of the cavity is under a given volume threshold expressed in $Å^3$. The keyword to select the volume threshold is the following:

Conditional_Volume_Filling_Value = 11.4

where the value represents approximately the volume of a water molecule. On the console the user can find the detected cavities, the associated volume and if the cavity was filled or not.
NanoShaper is able to save the serials (1 based indices) of the atoms that surround the cavity; to save the cavities info (atoms indices) on file the following key word must be turned on:

Save_Cavities = true

If the user plans to the save the full internal map (by setting Save_Status_map = true), than irrespectively of the previous keyword, cavities info will be saved. A file named cavAtomsSerials.txt will be produced in which for each line the set of the atom serials producing the cavity will be given. The user can feed these indices to VMD by using the serial keyword as atom selection string and visually see which are the atoms that contribute to the cavity. Note that NanoShaper assumes the serials are consecutives and starts from 0; if this is not the case for the current pdb, than NanoShaper serials can be converted to VMD indices by just subtracting 1 to all the serials.

3.3 Pocket detection

The idea is that pockets could be defined as a volumetric difference map: in particular as the volumetric difference map between two volumetric surfaces. The pockets could be tought as what it can be accessed by a water molecule (the usual SES with 1.4 Å of radius) and what cannot be accessed by a virtual probe of bigger size (let us say 3.0 Å).
Operatively to detect pockets the user has to set the following keyword:

$$\mathsf{Operative\_Mode} = \mathsf{pockets}$$

whereas the default value normal means the usual surface building operative mode. The user can choose to detect only pockets or both cavities and pockets; the associated keyword is:

$$\mathsf{Pockets\_And\_Cavities} = \text{true}$$

Moreover the user can even control the size of the big and the small radii surfaces whose difference rules the pocket detection. In particular to detect very flat pockets an higher value of the big probe radius will be needed, whereas if the pocket is *well formed* (i.e. a tunnel with a relatively small mouth) the default value of 3.0 Å will be sufficient. At the limit, if the big radius goes to infinity the difference map comes from the difference of the convex hull and the SES. The two associated keywords are:

$$\mathsf{Pocket\_Radius\_Big} = 3.0$$
$$\mathsf{Pocket\_Radius\_Small} = 1.4$$

where the default values where shown. For the pockets/cavities NanoShaper can estimate the volume and the surface area; to enable surface area estimation the keyword Triangulation must be to set to true. In this case the triangulations of the pockets will be saved such that the user can visualize them. These will be named as per cav_triX.Y where X is a number starting from 0 that index the pockets/cavities and Y is the format that by default is Geomview .off, or it can be MSMS .face, .vert (such that the triangulations can be visualized in VMD). The pockets can be filtered based on the volume; to filter pockets the user can specify an equivalent number of water molecules that can stay on the pocket; the associated key word is :

$$\mathsf{Num\_Wat\_Pocket} = 2$$

The value of 2 is the default one.

3.4 DelPhi interfacing

NanoShaper can be interfaced to DelPhi when compiled as a dynamic library (.so on Linux, .dynlib on Mac, .dll on Windows). To compile NanoShaper as

a shared object the user has to choose the lib option when requested by the setup.py script. Upon compilation the user must assure that the library is visible to the system. In the case of Windows it is sufficient to put the library in the directory where DelPhi will be run or put it in the System path. On Linux the library must be put on a reachable folder; the easiest way to do that is to let point the LD_LIBRARY_PATH environment variable to the directory that contains the library. On Mac the equivalent is DYLD_LIBRARY_PATH.
In order to let DelPhi load the shared library, it must be patched and recompiled. DelPhi in Fortran 77 and double precision is the host code that is targeted by NanoShaper: this can be downloaded from

http://compbio.clemson.edu/delphi.php.

To patch DelPhi the user has to download the DelPhi patcher program present at www.electrostaticszone.eu.
It is sufficient to place the patcher directory content inside the DelPhi folder and run the script delphiPatcher.py. If additionally the NanoShaper folder is under the DelPhi folder, then the NanoShaper installer will be run; however the user can separately compile and install the NanoShaper library. Being dynamical objects when the user update NanoShaper there is no need to recompile DelPhi, unless the way it interfaces to NanoShaper changes.
To properly run DelPhi with NanoShaper the user must provide in the running directory the default NanoShaper configuration file, namely surfaceConfiguration.prm. Moreover to instruct DelPhi to use NanoShaper instead of its internal surfacing routine the new keyword surface must be used. For instance for using the Connolly-Richards surface provided by NanoShaper it is sufficient to add the following line to the DelPhi configuration file:

surface(connolly)

In an analogous ways the skin and the blobby surfaces can be selected. The user can save the potential in cube file format and visualize it with VMD.
If in the surfaceConfiguration.prm file is requested to compute the pockets, DelPhi will be used only as a pdb pre-processing tool; the electrostatic potential will not be computed but now the output will also have the residues information. The output is in a ProShaper like format; the pair mol.info and mol.pocket files contain the pockets/cavities information.

4 Python interfacing

NanoShaper can be also compiled as a Python library by choosing the corrisponding option of the installer. For Windows precompiled Python modules are provided. From Python the full functionalities of NanoShaper are provided. Triangulating the surface means using very few lines of Python code:

Listing 1: NanoShaper in Python

```
import NanoShaper

# conf file name
conf = "conf32.prm"
# consistency check of conf file and conf loading
cf = NanoShaper.init(conf)
# read from configuration file scale
scale = cf.readFloat("Grid_scale")
# read from configuration file perfil
perfil = cf.readFloat("Grid_perfil")
# read from configuration file the mol file name
molFile = cf.readString("XYZR_FileName")
# build the grid
grid = NanoShaper.DelPhiShared(scale,perfil,molFile,False,False,False);
# build the surface
surf = NanoShaper.surfaceFactory().create(cf,grid);
# run the requested operations in the configuration file
NanoShaper.normalMode(surf,grid)
# clean up
NanoShaper.dispose(cf)
```

---

5 Configuration file keywords

The configuration file has the following flags and options:

5.1 Global parameters

Operative_Mode: String value. Determines the operative mode of NanoShaper. When set to normal the default triangulation flow is executed. If the pockets value is used pocket detection will be executed.

Grid_scale: Real value. Specify in Angstrom the inverse of the side of the grid cubes. E.g. 2.0

Grid_perfil: Percentage that the surface maximum dimension occupies with respect to the total grid size. E.g. 80.

5.2 Maps settings

Build_epsilon_maps: Bool value. Build the epsilon and salt maps needed for an FD solver (e.g. Delphi) solver of the PB equation. By default these are disabled.

Build_status_map: Bool value. Build the status map needed for cavity detection. Enable this if you need to triangulate without Accurate_Triangulation enabled.

5.3 Surface parameters

XYZR_FileName: String value. This is the file name of the molecule to be loaded in the format xyzr. The format xyzr simply has one atom per line and each line is respectively the x,y,z coordinate and the radius.

Surface_File_Name: String value. This is the name of the surface to be loaded. For .off and .ply write the full file name; for msms file write either the full name of one of the .face or .vert file.

Surface: This string specifies the possible surfaces that can be loaded/built. mesh indicates a triangulated mesh is in .ply, .off or msms format; skin will build the Skin surface, blobby will build the Blobby surface and ses means the Connolly-Richards surface. When an msms file is loaded the surface is autmatically converted in a .off file named msms.off.

Skin_Surface_Parameter: Real value. It is the $s \in (0, 1)$ parameter of the Skin Surface. Its default value is 0.45 that leads to a surface very similar to the SES surface. For $s = 1.0$ the surface is the convex hull of the atoms, if $s = 0.0$ the van der Waals surface is obtained.

Skin_Fast_Projection: Bool value. This flag says if enabling or disabling a fast point to skin surface projection routine; the fast projection routine is slightly less accurate than the slower one but it can be several times faster. In most of the cases the difference is minimal. By default, to be very conservative, it is set

to false.

Blobbyness: This is the blobbyness value of the Blobby surface. Its default value is -2.5 that makes it not too far from the SES.

Cavity_Detection_Filling: Bool value. If enabled cavity detection/filling is run after surface ray-casting.

Keep_Water_Shaped_Cavities: Bool value. This is an experimental feature. It will try to check the shape of cavity. If spherical the cavity will be maintained, if highly non spherical it will be removed. This flag is usefull when one wants to filter out cavities with volume higher than a water molecule but which shape is non spherical (e.g. tight long tube); this can happen in the skin surface.

Conditional_Volume_Filling_Value: Real value in cubic Angstrom. It indicates the threshold volume for which a void/cavity has to be filled or not. Cavities/voids whose volume is less than this threshold are filled. The triangulation is corrected accordingly. This cavity detection mode can be used in normal mode.

Num_Wat_Pocket: Integer value. This indicates the minimal number of water molecules that a pocket can accomodate the get not filtered out. This threshold refers to the pocket detection operative mode.

Pockets_And_Cavities: Bool value. If true both pockets and cavities will be computed if in pockets operative mode. If false only pockets will be computed.

Accurate_Triangulation: Bool value. If enabled the surface is triangulated and it is granted that every vertex is analytically sampled from the surface. If disabled vertices are not analytically sampled but approximated. Disabling accurate triangulation will signficantly decrease both the quality of the surface and the memory requirements. If you set false consider enabling surface smoothing to increase visual quality.

Triangulation: Bool value. If enabled the surface is triangulated and saved in triangulatedSurf.off. For the Blobby surface, triangulation is always performed and the file is saved in blobby.off; if triangulation is enabled triangulatedSurf.off represents the Blobby surface after cavity detection/removal.

Max_Probes_Self_Intersections: Max number of probes collected for self intersection grid cell during Connolly surface computation. Default value is 100. Very rarely this parameter needs changing. In case, NanoShaper will complain about that.

Self_Intersections_Grid_Coefficient: Float value. In very rare cases you may have to increase this value to correctly compute the Connolly surface. If needed, NanoShaper will complain about that.

5.4 Mesh settings

Check_duplicated_vertices: Bool value. If enabled both during triangulation writing and reading duplicated vertices are checked.

Smooth_Mesh: Bool value. If enabled the surface is triangulated and then smoothed by Laplacian smoothing. Smoothing should be used to increase visual quality of the mesh if Accurate_Triangulation is false or in general when triangles quality matters such as in Boundary Element Solvers.

Vertex_Atom_Info: Bool value. If enabled for each vertex of the triangulation the nearest atom index is computed and saved either in OFF+N or msms format.

Compute_Vertex_Normals: Bool value. If enabled triangulation vertices normals will be computed, analytically where possible. Turn on this option if you save in msms format.

Save_Mesh_MSMS_Format: Bool value. If enabled the triangulated mesh is saved in MSMS format. To enable this format vertex normals must be computed by enabling the Compute_Vertex_Normals keyword.

5.5 Acceleration Grid Settings

Max_mesh_auxiliary_grid_size: Integer value, default 100. Maximum grid size of the acceleration grid for meshes for boundary grid projections.

Max_mesh_patches_per_auxiliary_grid_cell: Integer value, default 250. Maximum number of triangles in a grid cube of the acceleration grid for boundary grid projections.

Max_mesh_auxiliary_grid_2d_size: Integer value, default 100. Maximum grid size of the acceleration grid for meshes for ray casting.

Max_mesh_patches_per_auxiliary_grid_2d_cell: Integer value, default 250. Maximum number of triangles in a grid cube of the acceleration grid for ray casting.

Max_ses_patches_auxiliary_grid_2d_size: Integer value, default 50. Maximum grid size of the acceleration grid for ses ray casting.

Max_ses_patches_per_auxiliary_grid_2d_cell: Integer value, default 400. Maximum number of patches in a grid cube of the acceleration grid for ray casting.

Max_ses_patches_auxiliary_grid_size: Integer value, default 100. Maximum grid size of the acceleration grid for boundary grid projections.

Max_ses_patches_per_auxiliary_grid_cell: Integer value, default 400. Maximum number of patches in a grid cube of the acceleration grid for boundary grid projections.

Max_skin_patches_auxiliary_grid_size: Integer value, default 100. Maximum grid size of the acceleration grid for skin boundary grid projections.

Max_skin_patches_per_auxiliary_grid_cell: Integer value, default 400. Maximum number of patches in a grid cube of the acceleration grid for boundary grid projections.

Max_skin_patches_auxiliary_grid_2d_size: Integer value, default 50. Maximum grid size of the acceleration grid for skin ray casting.

Max_skin_patches_per_auxiliary_grid_2d_cell: Integer value, default 400. Maximum number of patches in a grid cube of the acceleration grid for ray casting.

5.6 File storage and others

Save_Status_map: Bool value. If enabled grid info is saved together with cavities info. It must be true in order to visually inspect grid points in cavities.

Save_eps_maps: Bool value. Save the epsmap needed by a Finite Difference PDE solver.

Save_PovRay: Bool value. If enabled the Skin surface is converted into a file named skin.pov ready to be ray-traced by PovRay program. The analytical Skin surface is saved in the PovRay file and not its triangulation; this is done to get the best visual effect. Conservatively the camera position is quite far from the molecule, so it may need manual adjustments.

Save_Cavities: Bool value. If enabled a file named cavAtomsSerials.txt will be produced in which for each line the set of the atom serials producing the cavity/pocket will be given. This option is available in both normal and pockets operative modes.

Number_Thread: Integer value. Here you explicitly set the number of execution threads to optimize the execution time or to use only a subset of your cores. As a rule thumb optimal performances can be obtained by setting as number of

threads the number of cores multiplied by 4; this should grant full CPU usage. If this keyword is not present NanoShaper will use a number of threads equal to the number of physical cores, remember that this choice is not optimal.

Tri2Balls: Bool value. When enabled either the input mesh or the triangulated mesh will be converted to a set of balls whose envelope well approximate the original surface.

Print_Available_Surfaces: If enabled the set of available surface will be printed. This is a way to debug if a possibly newly defined surface has been recognised.

6 Extending the framework

The architecture of NanoShaper is in plain C++ and allows easy extensibility thanks to template metaprogramming for the Surface class: adding a new Surface boils down in extending the Surface class via an additional class and re-compiling without any other modification to the rest of the framework. From inside the added surface class the developer will be able to add new keywords seamlessly.
Most of the algorithms are in the Surface class in which also the parallelization is performed. In order to expand the framework one has to extend the Surface class; to this aim two strategies are possible: in the first case one re-implements the method getSurf and the Surface class facilities are not used, in a second scenario one wants to use Surface class facilities.
In this latter case the following pure virtuals must be implemented:

- bool build()
- bool save(char* fileName)
- bool load(char* fileName)
- void printSummary()
- bool getProjection(double p[3],double* proj1,double* proj2,double* proj3,double* normal1,double* normal2,double* normal3)
- void getRayIntersection(double p1[3],double p2[3],vector<pair<double,double*>>& intersections,int thdID)

The functions save, printSummary and getProjection are only formally mandatory and can be safely implemented as empty functions; in particular getProjection is used when NanoShaper is interfaced with DelPhi and it is not necessary

for triangulation. Instead the only core routine is the getRayIntersection routine. If this routine is implemented then, from Surface inheritance, one will get: volume estimation, surface area estimation, cavity detection, cavities removal, cavities-removal-consistent surface triangulation. The ray casting will be automatically parallelized by Surface class; one has to assure that getRayIntersection routine is thread safe, that is rays can be casted in parallel without compromising consistency of data; note that no concurrent writing should be requested. If you need concurrent writing (such as concurrent cout) a boost mutex in Surface class is available.
For more information you could follow the ExampleSurface class that is an example class that step by step shows the structure of the class that is needed to extend NanoShaper; also consider reading the doxygen documentation in the doc folder; for any /bug fix/help/improvement/ contact the Author at sergio.decherchi@iit.it.

7 Examples

In the examples folder some examples molecules can be run; moreover for convenience a precompiled win 32,64 bits NanoShaper is given.
The first one is given by the configuration file conf.prm; this file loads the atoms file barstar.xyzr and compute the Skin surface. Playing with this file you can change the shape of the surface by changing the Skin surface parameter or using the Blobby surface instead of the Skin.
The file bunny_conf.prm shows how a standard mesh can be processed. In order to load the surface two *virtual* atoms are defined which represents the two opposite points of the bounding cube of the mesh; this is done in order to build a grid consistent with the mesh. Other two dummy atoms are added on the same positions because NanoShaper needs at least four atoms to work.
In the folder python_example the file example.py shows how to interact with NanoShaper classes from Python.

Acknowledgments

The Author would like to thank Walter Rocchia for invaluable help, Nico Kruithof for useful discussions and code snippets about the CGAL implementation of the Skin surface, Marco Attene from IMATI for providing the .ply file reader and the IIT Computational Platform for the computing resources.
This work is supported by NIGMS, NIH, grant number 1R01GM093937-01.

References

[1] Y. Zhang G. Xu C. Bajaj. Quality meshing of implicit solvation models of biomolecular structures. *Computer Aided Geometric Design*, 23:510530, 2006.

[2] Sergio Decherchi and Walter Rocchia. A general and robust ray casting based algorithm for triangulating surfaces at the nanoscale. *PLOS ONE*, 8(4), 2013.

[3] H. Edelsbrunner. Deformable smooth surface design. *Discrete and Computational Geometry*, 21(1):87–115, 1999.

[4] M. Phillips, I. Georgiev, A.K. Dehof, S. Nickels, L. Marsalek, H.-P. Lenhof, A. Hildebrandt, and P. Slusallek. Measuring properties of molecular surfaces using ray casting. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –7, april 2010.

[5] M. F. Sanner, A. J. Olson, and J.C. Spehner. Reduced surface: An efficient way to compute molecular surfaces. *Biopolymers*, 38:305320, 1996.