STAG: Smart Tools and Applications in Graphics (2019)
M. Agus, M. Corsini and R. Pintus (Editors)

The Py3DViewer project:
a Python library for fast prototyping in geometry processing

G. Cherchi[1], L. Pitzalis[1,2], G. L. Frongia[1] and R. Scateni[1]

[1]University of Cagliari, Department of Mathematics and Computer Science, Italy
[2]Visual Computing, CRS4, Italy

Abstract

Fast research and prototyping, nowadays, is shifting towards languages that allow interactive execution and quick changes. Python is very widely used for rapid prototyping. We introduce Py3DViewer*, a new Python library that allows researchers to quickly prototype geometry processing algorithms by interactively editing and viewing meshes. Polygonal and polyhedral meshes are both supported. The library is designed to be used in conjunction with Jupyter environments, which allow interactive Python code execution and data visualization in a browser, thus opening up the possibility of viewing a mesh while editing the underlying geometry and topology.*

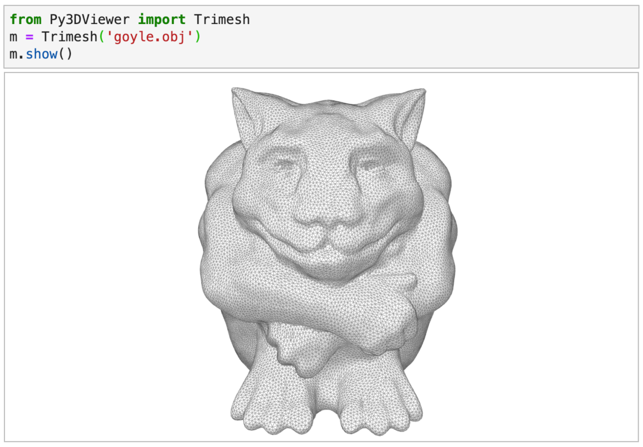
CCS Concepts
• ***Computing methodologies*** → *Mesh models; Volumetric models; Rendering;*

1. Introduction

Computer Graphics has been traditionally dominated by code written in the C++ language, because of its high speed and widespread diffusion of libraries that allow programmers to access the graphical capabilities of the underlying machine directly. Even though interpreted languages, such as Python, are much slower than C++, they allow to quickly prototype ideas by avoiding the compilation step. Moreover, in their interactive variants, they allow inherent debugging of programs by being able to execute blocks of code in arbitrary order. The prototyping speed, coupled with the high number of performant C-based libraries that the Python ecosystem possesses, made it the de facto standard for research in Deep Learning. Libraries such as Numpy [Oli], Tensorflow [AAB*15] and PyTorch [PGC*17] allow researchers to develop algorithms by using tensors, n-dimensional arrays, whose operations can run either on the CPU or directly on the GPU. This abstraction can greatly improve development speed, since there is no need to write C++ or CUDA directly.

Prototypes in the above-mentioned fields are often developed through the use of Jupyter Notebooks [Jupa], which are browser-based environments that allow to run code in blocks, called cells. Cells can also include Rich Text and arbitrary Javascript-based interactive views.

We introduce **Py3DViewer**, a library designed to provide fast prototyping and visualization capabilities with a modern interactive approach. Our tool is based on the data structures provided by the aforementioned Numpy library, since it is the most commonly used library in the scientific computing context (for more details see section 4). This design choice allows any user to quickly extend the functionalities of our library whenever they need so, and to easily integrate existing Numpy-based algorithms.



```
from Py3DViewer import Trimesh
m = Trimesh('goyle.obj')
m.show()
```

Figure 1: *A screenshot of Py3DViewer running inside a JupyterLab cell. The Goyle tri-mesh model is courtesy of [ZJ16].*

The main driving idea behind our project is to provide researchers in the Computer Graphics field with a tool to read, edit and view meshes directly in a Jupyter Notebook without needing to

DOI: 10.2312/stag.20191374

compile any code and being able to run their prototypes remotely on any Jupyter instance. Not only does this speed up the prototyping phase, but it seamlessly allow to run any code on the GPU by utilizing any Numpy-compatible tensor library such as PyTorch, a task which would be more verbose if done in the typically used C++ and CUDA languages directly. Moreover, to provide value that, to the best of our knowledge, no other 3D library in Python does, our project supports visualization and editing of both polygonal and polyhedral meshes without depending on any other library other than Numpy for computing and PyThreeJS [The17] for visualization.

Py3DViewer is available on GitHub, licensed with MIT: (`github.com/cg3hci/py3DViewer`). The required dependencies are Numpy, PyThreeJS and IPyWidgets [Jupb].

2. State of the art

In the context of Computer Graphics, the visualization of data is tightly coupled with the research field itself. For this reason, there exist a multitude of viewers, with a different focus based on what they were originally needed for. For quick visualization and basic processing that is decoupled from the development phase, many standalone tools exist. Meshlab [CCC*08] is a software that bundles mesh visualization and common mesh processing algorithms with a comprehensive GUI, and is widely used in the community as a secondary tool during development to quickly check mesh properties and to remesh models. While a very feature-rich tool, it lacks ways to deal with polyhedral meshes. In this context, there also exist a number of tools to visualize polyhedral meshes, such as Graphite [ALI] and ParaView [Aya15]. While these tools allow volumetric representations to be visualized, they can not be easily coupled with the development itself, since they are standalone programs.

On the other hand, a number of libraries also exist that can be used as tools to visualize, store and perform common processing algorithms on meshes all in a single tool. For example, a recent C++ header only library, called Cinolib [Liv19], aims to simplify the tasks most commonly performed in the field by providing a visualization interface and many algorithms and data structures all without needing to depend on other libraries. This has the advantage of speeding up the prototyping and development phases, but on the other hand still suffers from the larger complexity of C++ based solutions. Based on this language, which is the one most commonly used to implement Computer Graphics software, there also exist CGAL [The19], VCG [CG13], libigl [JP17], OpenMesh [BSBK02] and OpenVolumeMesh [KBK13] etc. These libraries also aim to provide an all-in-one solution to visualization and computation, with a deeper focus on efficiency of storage and speed compared to ease of use. It is noteworthy that many other libraries that try to solve this same problem exist in the field. While many C++ based solutions eventually started supporting Python bindings to simplify the prototyping process, such as the aforementioned CGAL, libigl, OpenMesh library, they are still often harder to use compared to native Python solutions that implement many features of the language that C++ based tools do not have access to.

A widespread way to prototype geometry processing algorithms is to write them in a Matlab [Mat] environment. The issue with this approach is that Matlab is a commercial software that requires to be installed, and as such might have compatibility issues and requires local storage in a machine. Compared to Python and Jupyter enviroments, Matlab is also harder to use on a remote machine.

When it comes to Python, a few tools already exist that try to solve this same problem. For example, PyMesh [The] is a library that is widely used for Geometry Processing tasks in Python. The main limitations that come from PyMesh is that it is not built in Python from the ground up, but to allow users to use all its features rather relies on a number of C++ dependencies such as CGAL and libigl. CGAl is one of the most widely used libraries in the field, and contains a very extensive collection of algorithms and tools. The high complexity of some of its dependencies makes it harder for the library to use many features of the language, sometimes requires a basic knowledge of the underlying libraries (for example for using CGAL based algorithms) and can sometimes prove more difficult to install. Moreover, PyMesh is designed as a purely processing library, and does not try to tackle the visualization problem. A library that does try to both visualize and process meshes in a single tool is Trimesh [Daw], which is a Numpy-based solution that leverages the full capabilities of the Python language and also supports Jupyter environments natively. The main limitations of Trimesh are that it can only process triangular meshes, and that it only provides very basic visualization features.

In the context of browser based mesh visualization, a very recent notable work is Hexalab [BTP*19], a Javascript based online tool that allows users to visualize their hexahedral mesh with a multitude of viewing options. The aforementioned Meshlab software also has an online javascript implementation [Vis17] as a standalone project.

While many other smaller projects exist, to the best of our knowledge the community still needed a comprehensive all-in-one tool that allows users to store, visualize and process both polygonal and polyhedral meshes, with a modern Jupyter native support inspired by the recent diffusion of this tool in the Deep Learning field, which is the problem this project aims to solve.

3. Architecture

The library is divided into modules. We decided to split the visualization, data structures and algorithms in three separate blocks to improve readability and extensibility. So far, the focus of the project has been providing data structures and visualization tools, therefore the algorithms module is still a work in progress compared to the rest of the library. We aim to add algorithms as the necessity arises and based on the community feedback. The visualization and data structures modules, instead, are ready to be used.

3.1. Data structures

To maximize efficiency and to properly utilize the underlying Numpy capabilities, we decided to design data structures that are index-based. Compared to a more classical rigid approach, where the internal attributes of the data structure can not be accessed and modified directly, we decided to make our data structures reactive,

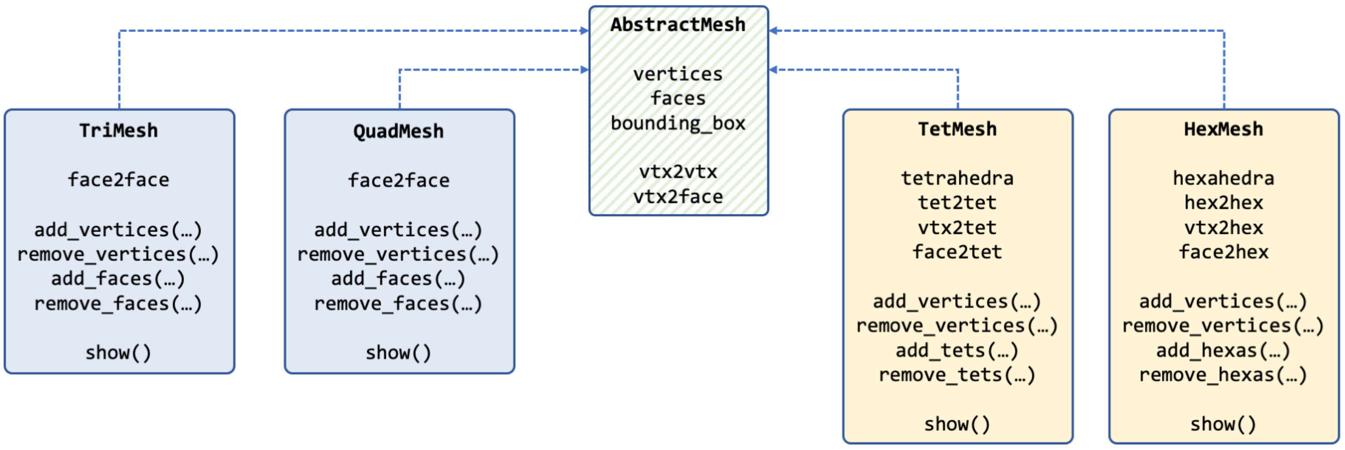


Figure 2: *The structure of the classes for the mesh representation, as explained in section 3.1*

meaning that a direct modifications to a property (e.g. removing a face) automatically refreshes the other properties (e.g. refreshing the vertex list) to maintain a consistent mesh structure. On top of this reactive approach, we also expose methods to do the same operations in a more human-readable way (e.g. `add_faces()` method).

We organized our library in two main levels (see figure 2 for the complete hierarchy). The main class is **AbstractMesh** which represents a generic mesh containing all the shared elements among the specific mesh classes. In particular, it contains an array of vertices (expressed as triples of float coordinates), an array of faces (composed of generic arrays of indices referring to the vertex array) and a bounding box of the mesh. Furthermore, it contains the adjacencies among vertices and between vertices and faces.

To implement surface meshes we extended the AbstractMesh with two classes: **Trimesh** for triangle-meshes and **Quadmesh** for quadrangular-meshes. Both these classes contain the information about the surface topology. In particular they contain the adjacencies between faces and all the methods to add and remove simplices. In the same way we extended the AbstractMesh with the **Tetmesh** and **Hexmesh** classes to represent volumetric meshes, in particular tetrahedral and hexahedral meshes. Both these classes contain the information about the surface and volume topology. In particular they contain the adjacencies between polyhedra and all the methods to add and remove elements.

All the aforementioned classes (except for AbstractMesh) implement the `show()` method that draws the mesh geometry in a canvas by calling functions of the Visualization module (see section 3.2). Moreover, the standard representation of each mesh is the `show()` method itself, so that if a mesh is evaluated in a block cell, it automatically shows the viewer.

3.2. Visualization

In the prototyping phase it is very useful to have a way to visualize the mesh during the developing process. To provide this functionality we implemented a viewer that can show any mesh and that can give the user an intuitive UI to interact with. We implemented a class **Visualizer** to manage the code related to the interface, in order to keep it separated from the core of the library. The Visualization module has two different modalities to launch the canvas:

- **without UI:** this visualization mode includes only the canvas and it is meant to be used for a quick view of the mesh.
- **with UI:** this visualization mode include a simple command tool interface to interact with the shown mesh (colors, wireframe, cuts, etc.) by using graphics widgets. It is meant to be used to deeply analyze the mesh.

We implement the classical visualization options regarding meshes. We include different options for the wireframe visualization, with customizable color and thickness. The color of the surface and volume elements is also customizable. Elements can be hidden or shown in different colors depending on the chosen quality metric or label 5. In the Utilities module we included different metrics to compute the per-simplex quality. Moreover, any custom metric is also supported on a per-mesh basis (see 3.1 for details). We support different color maps to visualize the chosen metric, as shown in figure 3. The mapping between a given metric and a color map differs based on the metric upper and lower bounds. This range is defined in the metric dictionary, where each value mapped to a key is defined as a tuple of this form: $<< min, max >, metric >$. If no upper and lower bound is provided (by setting these values to the null object *None*), the metric is normalized between zero and one using the existing values in the array to calculate the bounds, then mapped to the color map. This is useful to highlight differences in a metric that has only an absolute value (such as volume). For metrics that have known bounds (such as a quality metric on a simplex), the whole range is mapped on the color map without stretching the values between the min max, to highlight absolute values compared to the possible range. This is useful to quickly visualize overall qualities of a mesh.

To allow an in-depth analysis of the mesh, we implemented a cut system to explore the interior of both surface and volumetric meshes. Two color pickers allow both the internal and external colors to be chosen independently. The UI supports a slider for each

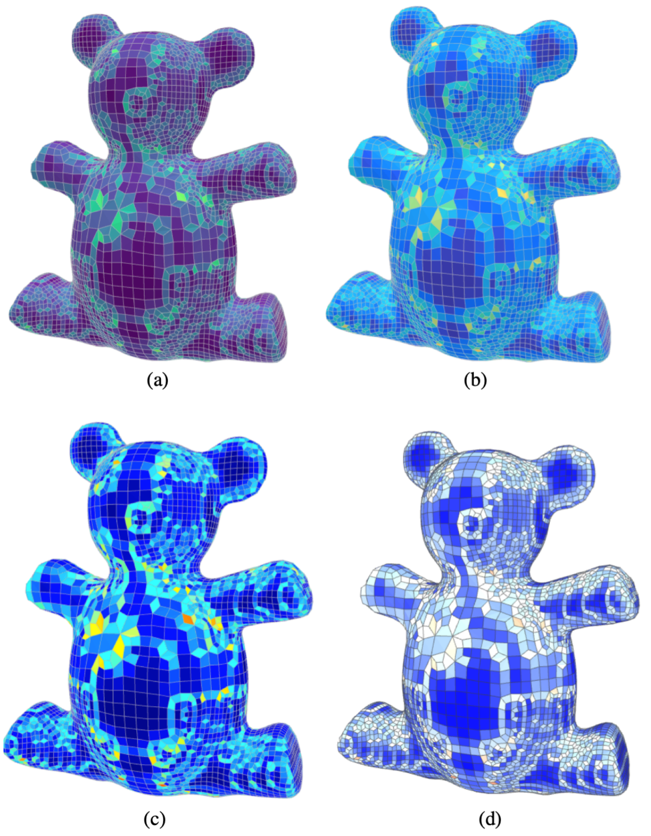


Figure 3: *Different color schemes for the quality visualization. We use the Scaled Jacobian as a metric to express the quality of the Toy tet-mesh. In (a) the "Virdis" scale, in (b) "Parula", in (c) "Jet" and in (d) the classical "Red-Blue". Model courtesy of [XLZ* 19].*

axis of the Cartesian space and we give the user the possibility to cut the mesh along one (or more) of the axes. To guarantee this operation in real-time we implement the cut operations by exploiting the tensorial form of the elements stored in our structure and the Numpy speed for matrix computations. In figure 4 we show how a cut mesh is drawn in the visualizer canvas.

All the aforementioned rendering functionalities (color changes, cuts, etc.) are also usable directly from the source code, without using the UI.

3.3. User Interface

If the user decides to view its mesh in the augmented canvas, which has a GUI that allows him to interact with the visualized mesh, the library provides a number of widgets. The UI is shown in figure 6.

- **Cuts**: to provide the user with the possibility of not showing part of the geometry, for example to analyze the interior of a volumetric mesh, the GUI contains three range sliders, one for each axis of the Cartesian space. Moreover, each slider is paired with a check button to invert the chosen range.
- **Wireframe**: the GUI also contains widgets to modify the wireframe appearance. In particular, we provide a slider to change the line opacity, paired with a color picker that allows the user to change the wireframe color.
- **Default mesh colors**: through the use of a small menu, the user can decide to color the mesh uniformly. In the case of volumetric meshes, the user can independently choose internal or external color. We provide this functionality through one color picker for the former, and two for the latter.
- **Metric based mesh colors**: through the same menu, the user can instead choose to color the mesh based on a given metric. To choose the metric, we provide another menu that is dynamically populated based on the available metrics. The user can then choose a color map out of another dynamically populated menu of color maps. The mesh is then colored based on the absolute metric value if the metric provides a range, or on the metric normalized along the color map if it does not.
- **Label based mesh colors**: the last menu option allows the mesh to be colored based on its labels. Each label can be independently colored by using a color picker. The library dynamically adds a color picker for each label contained in the mesh data structure. An exhaustive example of this visualization mode is shown in figure 5

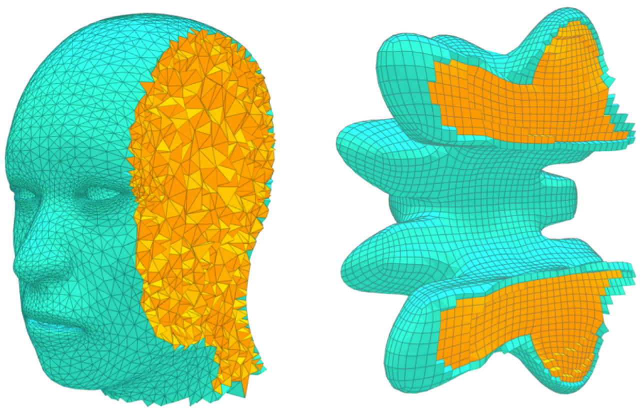


Figure 4: *Two examples of cuts in meshes. The tet-mesh is created with the MakeHuman tool and turned into a tet-mesh by using TetGen [Si15]. The hex-mesh model is courtesy of [GSZ11].*

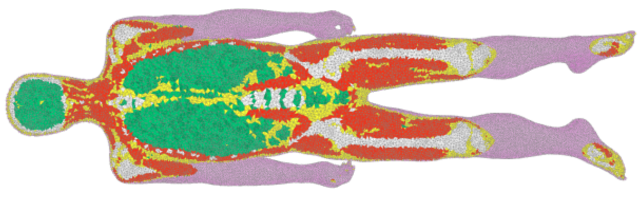
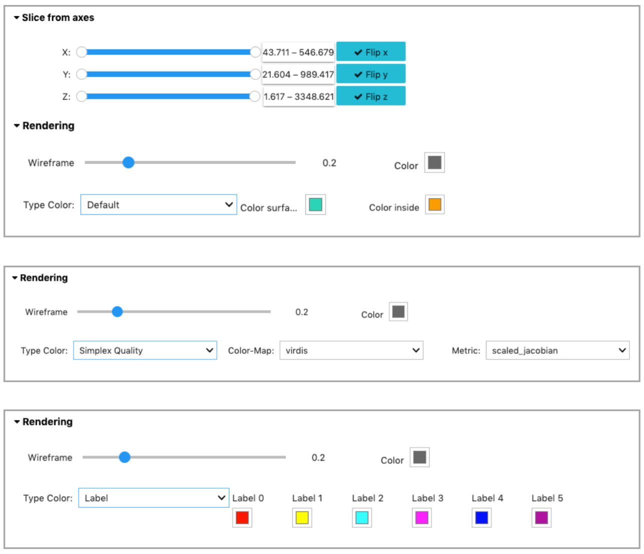


Figure 5: *The tet-mesh of the Korean Male model, obtained from the Visible Korean Human [PCH* 05] images, shown by using the label colors. In pink the label of the skin portion of the model, in red the muscles, in yellow the fat, in white the bones, and in green the internal organs.*



▾ Slice from axes

X: 43.711 – 546.679 ✓ Flip x
Y: 21.604 – 989.417 ✓ Flip y
Z: 1.617 – 3348.621 ✓ Flip z

▾ Rendering

Wireframe 0.2 Color
Type Color: Default Color surfa... Color inside

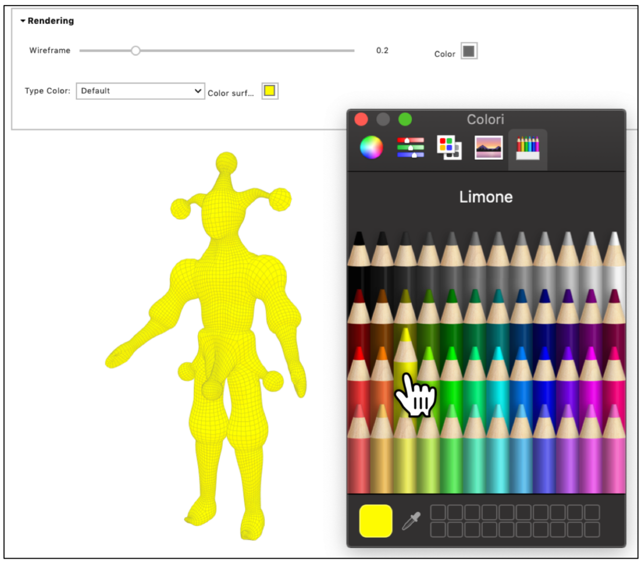
▾ Rendering

Wireframe 0.2 Color
Type Color: Simplex Quality Color-Map: virdis Metric: scaled_jacobian

▾ Rendering

Wireframe 0.2 Color
Type Color: Label Label 0 Label 1 Label 2 Label 3 Label 4 Label 5

Figure 6: *The UI interface of Py3DViewer. By changing the Type Color rendering is changed, and the related color widgets are shown.*



▾ Rendering

Wireframe 0.2 Color
Type Color: Default Color surf...

Figure 7: *Interaction with the UI interface of Py3DViewer. The Joker quad-mesh shown in the figure is courtesy of [ULP*15].*

3.4. Utilities

This module contains utilities that are not related to the storage or visualization of the data. In particular this module is meant to contain generic algorithms that can be applied to our data structures. Currently this module contains the algorithms to compute the metrics we use to determine the simplex quality. We implement different metrics for both surface and volumetric meshes:

- **Surface Meshes:** Area and Aspect Ratio.
- **Volume Meshes:** Volume and Scaled Jacobian.

Metrics can be easily added and customized to any of our data structures, and the visualizer is able to work with any custom metric. For more information on how metrics can be extended see 3.1.

The definitions of the color maps available in the viewer to represent a given metric are also contained in this module. The library contains the following commonly used color maps: Parula, Jet, Virdis and Red-Blue.

These color maps are defined as a dictionary contained in this module, of this form $'color\_map' : numpyArray(Nx1)$. The viewer is agnostic to the content of this dictionary: this can be changed at runtime by the user and any new entry will display as a color map in the interface.

This module also includes common I/O operations, such as common file extension parsers. Currently, we support only the *.OBJ* format for surface meshes and the *.MESH* one for volumetric meshes. In the future, we aim to support the most common mesh formats, prioritizing based on community feedback.

4. Design Choices

We decided to base our viewer on Numpy because of its widespread diffusion in the context of scientific computing in Python and its efficiency. Moreover, Numpy uses a syntax that is inspired by Matlab [Mat], which is another tool that is often used for quick prototyping in many scientific fields, as well as in Computer Graphics. We hope that this design choice will make it easier for our users to adopt this paradigm, making it easier for them to use Python as a language to quickly prototype ideas instead of C++. Numpy is also extremely efficient [WCV11], which makes it a valid choice when compared to more commonly used C/C++ based libraries.

Our viewer also aims to eventually expand as a more general tool for mesh processing in Python, so we designed it to be easy to extend for any need that may arise. The UI is agnostic to most of the underlying data shown: this means that a user can simply add a metric to the known metric dictionary and the viewer will automatically show it inside its interface, seamlessly allowing arbitrary metric visualization on the given mesh. Apart from custom metrics, at the moment the user can also add arbitrary color maps to visualize the metrics as the need arises. Moreover, we aim to allow the viewer to optionally be set to be truly reactive, so that any change to the underlying mesh data structure immediately causes a redrawing of the scene to mirror the changes. By making this possible, any algorithm could be visualized after each step of the computation. This could prove extremely useful to quickly visualize the output of a neural network on the mesh for any Geometric Deep Learning [BBL*17] task directly during its training. One of our priorities, apart from the complete reactive approach, is to simply allow a redraw whenever the user chooses to, so that it would become trivial to refresh the mesh at arbitrary steps of a computation.

The design of our viewer was also driven by the need for a general mesh visualization tool in Python with support for Jupyter environments. Most commonly used 3D libraries for Python, such as PyMesh or Trimesh, lack some functionalities that we deemed necessary for a prototyping pipeline. The former, while being able to work with volumetric meshes, is not able to easily show them. The latter, on the other hand, lacks support for polyhedral meshes and has limited visualization capabilities. Py3DViewer is designed as a single tool that can be easily integrated into any Python 3D prototype that deals with polygonal or polyhedral meshes.

5. Simple examples

We design our library to be easy and quick to use in the context of fast prototyping. In the example below it is shown how a mesh can be loaded from a file and drawn in a canvas with just a few lines of code. Then it is possible to interact with the mesh with the GUI widgets as shown in the figure 7.

```
from Py3Dviewer import Tetmesh

#load a Tetmesh object from file
my_mesh = Tetmesh(filename='dog.mesh')

#show the mesh into a simple canvas
my_mesh.show()

#show the mesh with the GUI
my_mesh.show(UI=True)
```

Our library also allows to create a mesh given an array of vertices and an array of faces or polyhedra. Moreover, in the code below we show some useful functionalities to change the mesh geometry and topology after its creation.

```
from Py3Dviewer import Trimesh

#create a Trimesh given the vertices and the
    faces
my_mesh = Trimesh(vertices, faces)

#remove a set of vertices from the mesh
my_mesh.remove_vertices([0, 5, 10, 25])

#remove a set of faces from the mesh
my_mesh.remove_faces([50, 51, 52, 53])

#add a set of vertices to the mesh
my_mesh.add_vertices([[10.2,25.4,3.3],
                      [7.9,8.43,5.22]])

#add a set of faces to the mesh
my_mesh.add_faces([[1,2,3],[70,72,74]])

#set the cut
my_mesh.set_clipping(min_x = 0.0, max_x = 1.0,
                     min_y = 0.5, max_y = 1.5,
                     min_z = 0.0, max_z = 1.0,
                     flip_x = True)
```

The following example shows an algorithm based on the Py3DViewer library, Numpy and Scipy [VGO*19] (a widespread scientific computing library based on Numpy). The code applies a Laplacian smoothing and shows how Python and its ecosystem of libraries allow the user to easily experiment with possible algorithm ideas.

```
from Py3DViewer import Trimesh, Viewer
import numpy as np
from scipy.sparse import lil_matrix as sp_matrix
from scipy.sparse import eye as identity
from scipy.sparse.linalg import sp_solve
from scipy.sparse.csgraph import laplacian

#loading and showing a Trimesh object from file
bunny = Trimesh('bunny.obj')
bunny.show()

#declaration and creation of the matrices for
    the Laplacian smoothing
lambda_ = 0.5
n = bunny.num_vertices
e = bunny.edges
P = bunny.vertices
A = sp_matrix((n, n))
A[e[:,0], e[:,1]] = 1
D = sp_matrix(A.shape)
D.setdiag(np.sum(A, axis=1))
L = D - A
I = identity(n)

#smoothing application and visualization of the
    new mesh
bunny.vertices = sp_solve( I + lambda_ * L, P)
bunny.show()
```

6. Comparison

We compared the loading time of our library with the C++ library "Cinolib". The comparisons are shown in table 6. We performed the tests by using meshes of different types and different numbers of simplices. We decided to compare the loading time (including the adjacencies computation) to give the reader an idea of the "price to pay" to switch from a C++ library to our Python-based library. Even though in the prototyping phase large meshes are rarely used, we still aim to improve this time by optimizing critical parts of the reading and loading process.

Model	#Vert.	#Simp.	T ([Liv19])	T (ours)
tri-mesh	14290	28576	0.25s	1.05s
quad-mesh	9650	9648	0.11s	0.49s
tet-mesh	4651	15412	0.52s	0.58s
hex-mesh	18951	15232	0.61s	1.06s

Table 1: *Differences in loading times for different kinds of meshes with different resolutions.*

All the tests have been performed by using a MacBook Pro 13" mid 2014, with an Intel core i5 CPU with integrated GPU and 8GB of ram.

We also tested the differences in visualization speed, considering both the rendering and the real-time interaction. The differences between our library and the C++ one are not appreciable.

7. Conclusion and Future Work

Py3DViewer is a library designed to facilitate the prototyping process of Computer Graphics researchers. We built a tool that allows a user to quickly load, edit and visualize a polygonal or polyhedral mesh in an interactive way. The project is also heavily inspired by the recent diffusion of prototyping phases based on Python and the Jupyter environment, especially in the blooming field of Deep Learning.

The code is available on GitHub, as a Python package (`github.com/cg3hci/py3DViewer`), and it is also available through the de facto standard Python package management system *pip*. We expect Py3DViewer to be useful to anyone that wants to quickly test an idea without the larger complexity of making a C++ prototype. We also expect our tool to facilitate sharing ideas between researchers by using online Jupyter based environments such as the popular Google Colab [Goo] system, compared to the difficulties of sharing a project that needs compilation.

Some of the features we aim to implement in the future are the following. First, we are working to extend the kinds of geometric data that the library supports, to appeal to an even broader part of the community. For the animation community, we are working to support skeletons, both for editing and visualizing (an example of a rudimentary visualization of a skeleton is shown in figure 8. Ideally, similar to what some other libraries are doing, Py3DViewer will support generic polygonal or polyhedral meshes, and point clouds. By making the system as modular as possible, we are working to make it easy to extend the supported data representations by making the viewer and the interface as data agnostic as possible. Eventually, we want the project to become the reference library for quick prototyping, and to achieve this goal we will extend the number of supported algorithms for each type of data representation that will be supported, based on what the community deems useful.

We aim to completely integrate PyTorch tensors to represent the geometry information of a mesh, so that any tensorial operation on the underlying data structure could be seamlessly executed on the GPU by simply passing GPU-instantiated tensors to our data structures or by loading a mesh file directly on the GPU. This would make it easier to use the viewer as just a visualization layer on top of the same data that is being processed by the algorithm. The possibility of having simultaneous computation and visualization would simplify Machine Learning experiments in PyTorch. Current solutions that aim to solve this problem, such as Visdom [The], are not integrated inside the Jupyter environment.

In the near future, the interactive Jupyter interface will be extended to more widgets and will support adding custom UI elements to better suit each different user's need. Moreover, similarly to how the online viewer Hexalab approaches the mesh visualization, we will support better shading options and will allow a more extensive rendering customization. By opening up different rendering options, the user will be able to more easily display the features they need to visualize.

More generally, the library will continuously improve in its documentation and examples. To facilitate prototyping for new users, we are working to implement interactive examples in the form of tutorial notebooks, by using the aforementioned Google Colab platform as a mean to quikly try our library's features. One of the most important features we will implement in the near future, is a complete PyTorch support for the data structures and algorithms for the underlying representation, instead of Numpy, if the user so chooses. This feature will allow the library to seamlessly run its algorithms on the GPU, to speed up parallel computations and to allow researchers to easily and efficiently prototype Geometry Processing algorithms and Deep Learning networks.

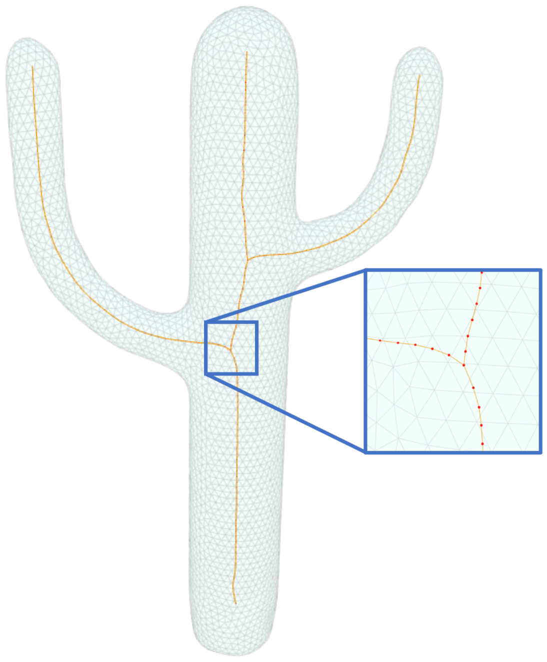


Figure 8: *An example of a rudimentary visualization of a skeleton. Model and skeleton are courtesy of [LS13] .*

Acknowledgments

The authors thank Giampaolo Perelli for his contribution to the rendering module of this library. Gianmarco Cherchi gratefully acknowledges the TDM project (POR FESR 2014-2020 Action 1.2.2.) for its financial support to his research.

References

[AAB*15] ABADI M., AGARWAL A., BARHAM P., BREVDO E., CHEN Z., CITRO C., CORRADO G. S., DAVIS A., DEAN J., DEVIN M., GHEMAWAT S., GOODFELLOW I., HARP A., IRVING G., ISARD M., JIA Y., JOZEFOWICZ R., KAISER L., KUDLUR M., LEVENBERG J., MANÉ D., MONGA R., MOORE S., MURRAY D., OLAH C., SCHUSTER M., SHLENS J., STEINER B., SUTSKEVER I., TALWAR K., TUCKER P., VANHOUCKE V., VASUDEVAN V., VIÉGAS F., VINYALS O., WARDEN P., WATTENBERG M., WICKE M., YU Y., ZHENG X.: TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: `https://www.tensorflow.org/`. 1

[ALI] ALICE: Graphite. Accessed: 2019-08-30. URL: `http://alice.loria.fr/software/graphite`. 2

[Aya15] AYACHIT U.: *The ParaView Guide: A Parallel Visualization Application*. Kitware, Inc., USA, 2015. 2

[BBL*17] Bronstein M. M., Bruna J., LeCun Y., Szlam A., Vandergheynst P.: Geometric Deep Learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine 34* (2017), 18–42. doi:10.1109/MSP.2017.2693418. 5

[BSBK02] Botsch M., Steinberg S., Bischoff S., Kobbelt L.: Openmesh-a generic and efficient polygon mesh data structure. 2

[BTP*19] Bracci M., Tarini M., Pietroni N., Livesu M., Cignoni P.: HexaLab.net: An online viewer for hexahedral meshes. *Computer-Aided Design 110* (May 2019), 24–36. URL: http://vcg.isti.cnr.it/Publications/2019/BTPLC19. 2

[CCC*08] Cignoni P., Callieri M., Corsini M., Dellepiane M., Ganovelli F., Ranzuglia G.: Meshlab: an open-source mesh processing tool. In *Eurographics Italian chapter conference* (2008), vol. 2008, pp. 129–136. 2

[CG13] Cignoni P., Ganovelli F.: Vcg library, 2013. 2

[Daw] Dawson-Haggerty et al.: Trimesh. URL: https://trimsh.org/. 2

[Goo] Google: Google Colab. Accessed: 2019-08-30. URL: https://colab.research.google.com. 7

[GSZ11] Gregson J., Sheffer A., Zhang E.: All-Hex Mesh Generation via Volumetric PolyCube Deformation. *Computer Graphics Forum (Special Issue of Symposium on Geometry Processing 2011) 30*, 5 (2011). 4

[JP17] Jacobson A., Panozzo D.: Libigl: Prototyping Geometry Processing Research in C++. In *SIGGRAPH Asia 2017 Courses* (New York, NY, USA, 2017), SA 17, ACM, pp. 11:1–11:172. URL: http://doi.acm.org/10.1145/3134472.3134497, doi:10.1145/3134472.3134497. 2

[Jupa] Jupyter Project: Jupyter. Accessed: 2019-08-30. URL: https://jupyter.org/index.html. 1

[Jupb] Jupyter Project: Jupyter Widgets. Accessed: 2019-08-30. URL: https://ipywidgets.readthedocs.io/en/latest/. 2

[KBK13] Kremer M., Bommes D., Kobbelt L.: OpenVolumeMesh–A versatile index-based data structure for 3D polytopal complexes. In *Proceedings of the 21st International Meshing Roundtable*. Springer, 2013, pp. 531–548. 2

[Liv19] Livesu M.: cinolib: a generic programming header only C++ library for processing polygonal and polyhedral meshes. *Transactions on Computational Science XXXIV* (2019). https://github.com/mlivesu/cinolib/. doi:10.1007/978-3-662-59958-7_4. 2, 6

[LS13] Livesu M., Scateni R.: Extracting Curve-Skeletons from Digital Shapes Using Occluding Contours. *The Visual Computer 29*, 9 (2013), 907–916. doi:10.1007/s00371-013-0855-8. 7

[Mat] MathWorks: MATLAB Optimization Toolbox. Accessed: 2019-08-30. URL: https://www.mathworks.com. 2, 5

[Oli] Oliphant T.: NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–. [Online; accessed 09-19-19]. URL: http://www.numpy.org/. 1

[PCH*05] Park J. S., Chung M. S., Hwang S. B., Lee Y. S., Har D.-H., Park H. S.: Visible Korean Human. Improved Serially Sectioned Images of The Entire Body. *2005 IEEE Engineering in Medicine and Biology 27th Annual Conference* (2005), 6563–6566. 4

[PGC*17] Paszke A., Gross S., Chintala S., Chanan G., Yang E., DeVito Z., Lin Z., Desmaison A., Antiga L., Lerer A.: Automatic differentiation in PyTorch. 1

[Si15] Si H.: TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator. *ACM Trans. Math. Softw. 41*, 2 (Feb. 2015), 11:1–11:36. URL: http://doi.acm.org/10.1145/2629697, doi:10.1145/2629697. 4

[The] The Facebook Team: Visdom. Accessed: 2019-08-30. URL: https://github.com/facebookresearch/visdom/. 7

[The] The PyMesh Development Team: PyMesh, 2014–. [Online; accessed 09-19-19]. URL: https://pymesh.readthedocs.io/. 2

[The17] The PyThreeJs Development Team: PyThreeJs, 2017. [Online; accessed 09-19-19]. URL: https://pythreejs.readthedocs.io/. 2

[The19] The CGAL Project: *CGAL User and Reference Manual*, 4.14 ed. CGAL Editorial Board, 2019. URL: https://doc.cgal.org/4.14/Manual/packages.html. 2

[ULP*15] Usai F., Livesu M., Puppo E., Tarini M., Scateni R.: Extraction of the Quad Layout of a Triangle Mesh Guided by Its Curve Skeleton. *ACM Transactions on Graphics 35*, 1 (2015), 6:1–6:13. doi:10.1145/2809785. 5

[VGO*19] Virtanen P., Gommers R., Oliphant T. E., Haberland M., Reddy T., Cournapeau D., Burovski E., Peterson P., Weckesser W., Bright J., van der Walt S. J., Brett M., Wilson J., Jarrod Millman K., Mayorov N., Nelson A. R. J., Jones E., Kern R., Larson E., Carey C., Polat İ., Feng Y., Moore E. W., Vand erPlas J., Laxalde D., Perktold J., Cimrman R., Henriksen I., Quintero E. A., Harris C. R., Archibald A. M., Ribeiro A. H., Pedregosa F., van Mulbregt P., Contributors S. . .: SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python. *arXiv e-prints* (Jul 2019), arXiv:1907.10121. arXiv:1907.10121. 6

[Vis17] Visual Computing Lab of ISTI - CNR: Meshlab JS, 2017. URL: https://www.meshlabjs.net/. 2

[WCV11] Walt S. v. d., Colbert S. C., Varoquaux G.: The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science and Engg. 13*, 2 (Mar. 2011), 22–30. URL: https://doi.org/10.1109/MCSE.2011.37, doi:10.1109/MCSE.2011.37. 5

[XLZ*19] Xu G., Ling R., Zhang J., Xiao Z., Ji Z., Rabczuk T.: Singularity Structure Simplification of Hexahedral Mesh via Weighted Ranking, 2019. arXiv:1901.00238. 4

[ZJ16] Zhou Q., Jacobson A.: Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016). 1