# Python Libraries for Mesh, Point Cloud, and Data Visualization (Part 1)

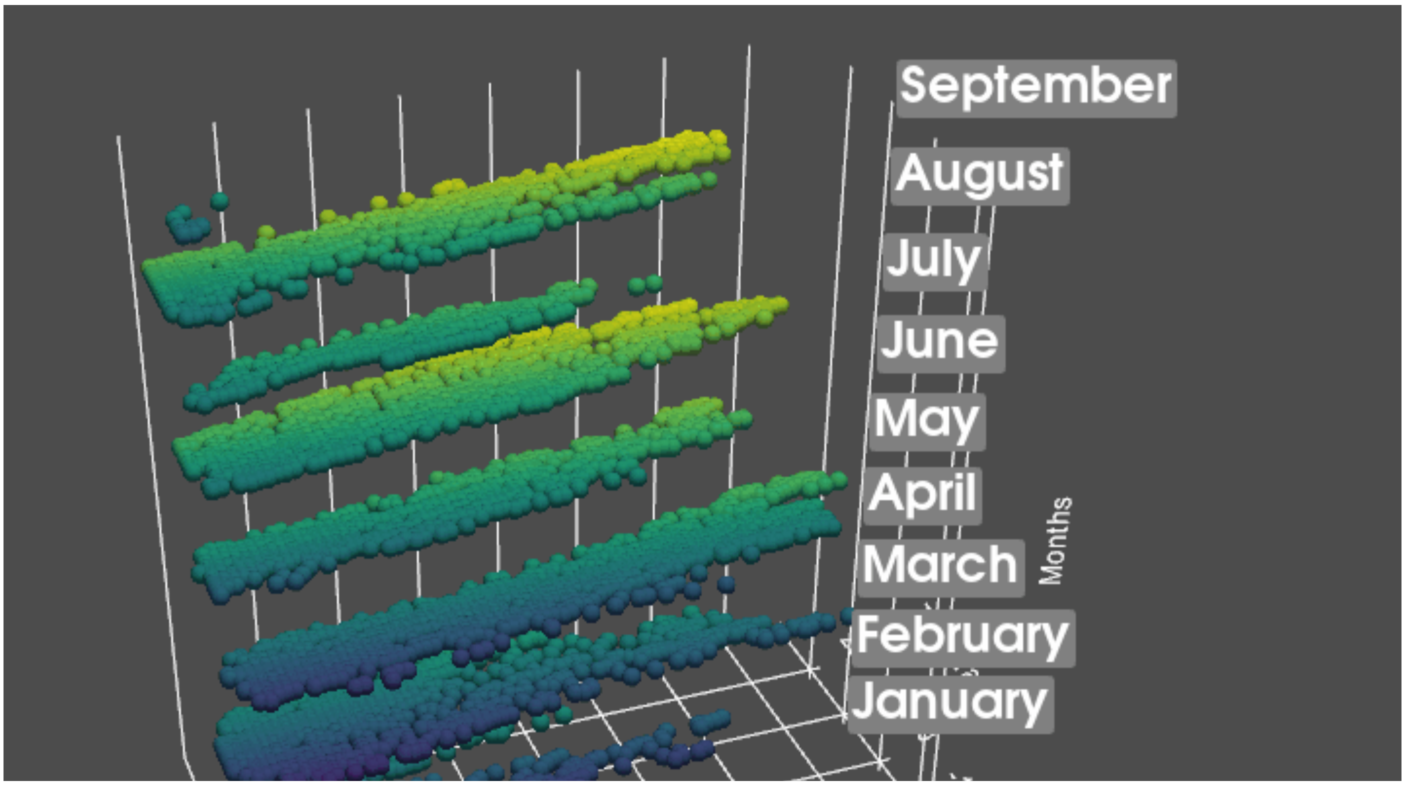
*Ivan Nikolov*

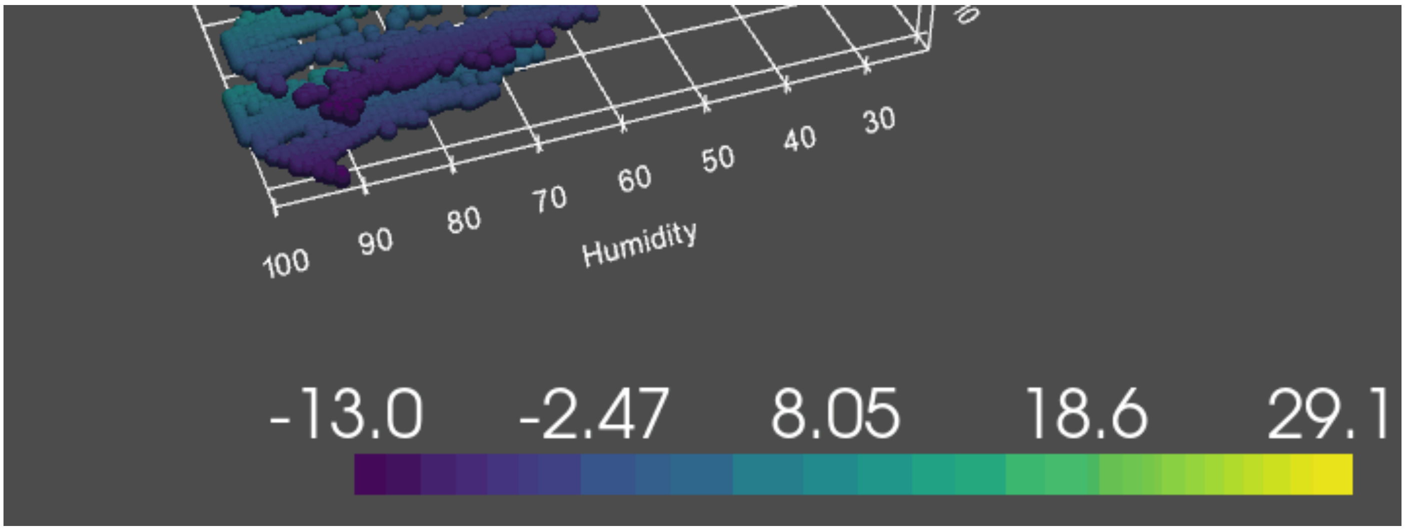
## Eight of the best Python libraries for stunning 3D visualizations, plots, and animations

tds

A tutorial on **8** of the best libraries for creating stunning 3D visualizations, plots and animations in Python. Who said that you need C++ knowledge to create fast, responsive point cloud, mesh or dataset visualizations? This hands-on tutorial will give you a rundown and code snippets to get you up and running these 8 libraries — Open3D, Trimesh, Vedo(V3do), Pyrender, PlotOptiX, Polyscope, PyVista and Simple-3dviz. This is Part 1, which takes a closer look at Open3D, Trimesh, PyVista and Vedo(V3do).

This is Part 1 of the planned tutorials on Python libraries for 3D work and visualization. In this part, we present the first 4 libraries for visualization — Open3D, Trimesh, PyVista, and Vedo, while Part 2 will focus on Pyrender, PlotOptiX, Polyscope, and Simple-3dviz. If you are interested in libraries for analysis and manipulation of meshes, point clouds, or datasets I will also cover libraries for widely used tasks like voxelization, feature extraction, distance calculation, surface generation, and meshing, among others in later articles. In these first two articles, we will mainly focus on visualization and animation.

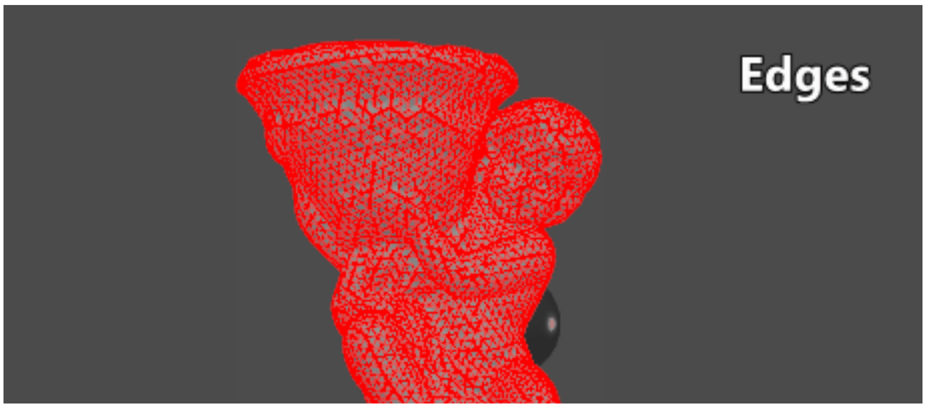


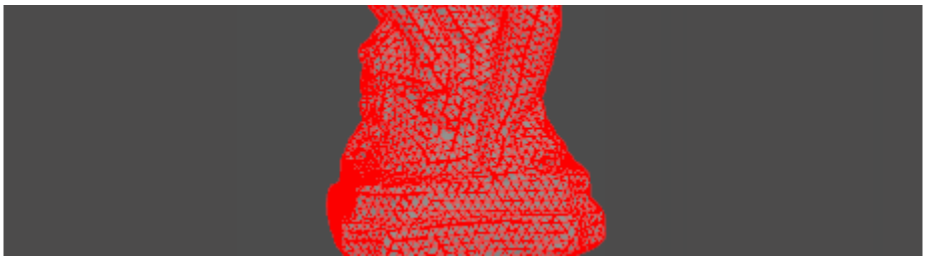


Visualizing weather (Temperature/Humidity) data for different months in 3D using PyVista | Image by the author

For a long time, people needed to use Matplotlib to visualize 3D content in Python. The good thing about using it is that if you have a Python installation, you most probably have everything that you need to visualize data in 3D using Matplotlib. The bad thing is that it does not support GPU hardware acceleration, meaning that once you pass a certain number of points or faces in 3D, things tend to become slow and unresponsive. This led to many people using the Point Cloud Library (**PCL**) as a catch-all solution for the visualization and analysis of meshes and point clouds. PCL is still one of the best libraries out there for 3D analysis and the fact that it is built in C++ guarantees that is versatile and responsive. Two main problems stop it from being perfect — the Python wrapper contains only a small subset of the functionality of the main PCL library and running it on Windows is an exercise in frustration, with multitudes of compilation errors, missing or broken functionality, and less than optimal tutorials. In short — if you work in Linux and you are not afraid of C++, PCL is the way to go. But what about everybody else?

In recent years more and more Python libraries have started to pop up to try to fill this void. Some of these libraries like Open3D, Trimesh, and Vedo are extremely robust and contain many different functionalities for the analysis, generation, and manipulation of meshes and point clouds. Others like Simple-3dviz, Polyscope, and Pyrender are more directed towards creating fast and beautiful visualizations and animations. For this article, I have chosen 8 libraries that provide different levels of visualizations and a number of additional options — from built-in manual controls to a large number of lighting and animation options and raytracing visuals. All the code is available at the GitHub repository HERE. The figure below gives a taste of some of the visualization possibilities, as well as extracted features.




Examples of some of the visualization options and extracted features that are possible. These are made with PyVista | Image by the author

This article is far from an exhaustive list of all available libraries and notably, is missing ones like pyntcloud, vpython, Mayavi, and the base VTK that other libraries are based on. I have chosen to skip these libraries, because of their relative installation complexity or because their visualization capabilities for large point clouds and meshes are less than ideal. But I plan to include them in the next articles exploring analysis and manipulation in 3D.

In addition, there are also Python wrapper libraries for using widely known visualization and manipulation applications like CloudCompare and Meshlab, directly through Python. CloudComPy and PyMeshLab create interfaces that directly connect to Python and bypass the necessity to use GUIs or even open the applications. We will discuss these wrapper libraries in more detail in later articles.

Let's jump right into each of these libraries — looking into how to install them, do the initial setup, and finally write some simple visualization code. For the purpose of showing visualization possibilities for meshes and point clouds, as well as datasets, I have provided both. First, an angel statue mesh in **.obj** format **HERE** and point cloud in **.txt** format **HERE**. The object has been featured in a number of articles [1], [2], [3] and can be also downloaded as part of larger photogrammetry datasets [4], [5]. Second, a time-series dataset containing weather data in **.csv** format **HERE**. The weather metadata was used as part of research for detecting long-term anomalies in surveillance footage [6]. The data has been extracted using the open-source API provided by the Danish Meteorological Institute (DMI) and is free to use in commercial and non-commercial, public, and private projects. For the purpose of working with the dataset, Pandas is required. It comes by default in Anaconda installations and it can be easily installed by calling `conda install pandas`.

## Visualization using Open3D

Open3D result | Image by the author

One of the fastest-growing 3D processing libraries in Python. It is built on C++, but exposes all functionality in both C++ and Python, making it fast, versatile, and robust. The library is aiming to be to 3D processing, what OpenCV is to computer vision — a one-stop solution containing everything that you would need and being easily connected to other libraries. For example PyTorch and Tensorflow for building deep learning networks for processing point clouds, voxels, and depth maps, through the extension called Open3D-ML.

Installation is extremely straightforward with a provided downloads picker for customizing your installation preferences. The library also works out of the box on Linux, Mac, and Windows and supports Python versions from 3.6 onward.

If you are using Anaconda, we can start by first creating a new environment `open3d_env`and then installing Open3D. In my case, I directly specify that I would like to use Python 3.8 in the new environment, but you can choose any of the supported versions.

```
conda create -n open3d_env python=3.8
conda activate open3d_env
pip install open3d
```

Once it has been installed, we can check if everything works by importing it`import open3d as o3d` and calling `print(o3d.__version__)` . If no errors are present we are good to go!

The io module of Open3d contains convenient functions for loading both meshes `o3d.io.read_triangle_mesh` , as well as point clouds `o3d.io.read_point_cloud` . In case the mesh has a texture and material, then the easier way to load them is to enable mesh post-processing by setting the flag `enable_post_processing = True` when calling the read mesh. In our case, the code for reading the angel statue mesh is:

In our case, I also call numpy as by default the Open3D structures cannot be viewed and printed, but they can be easily transformed into numpy arrays, as shown in the code above. All the information on the imported mesh can be easily accessed.

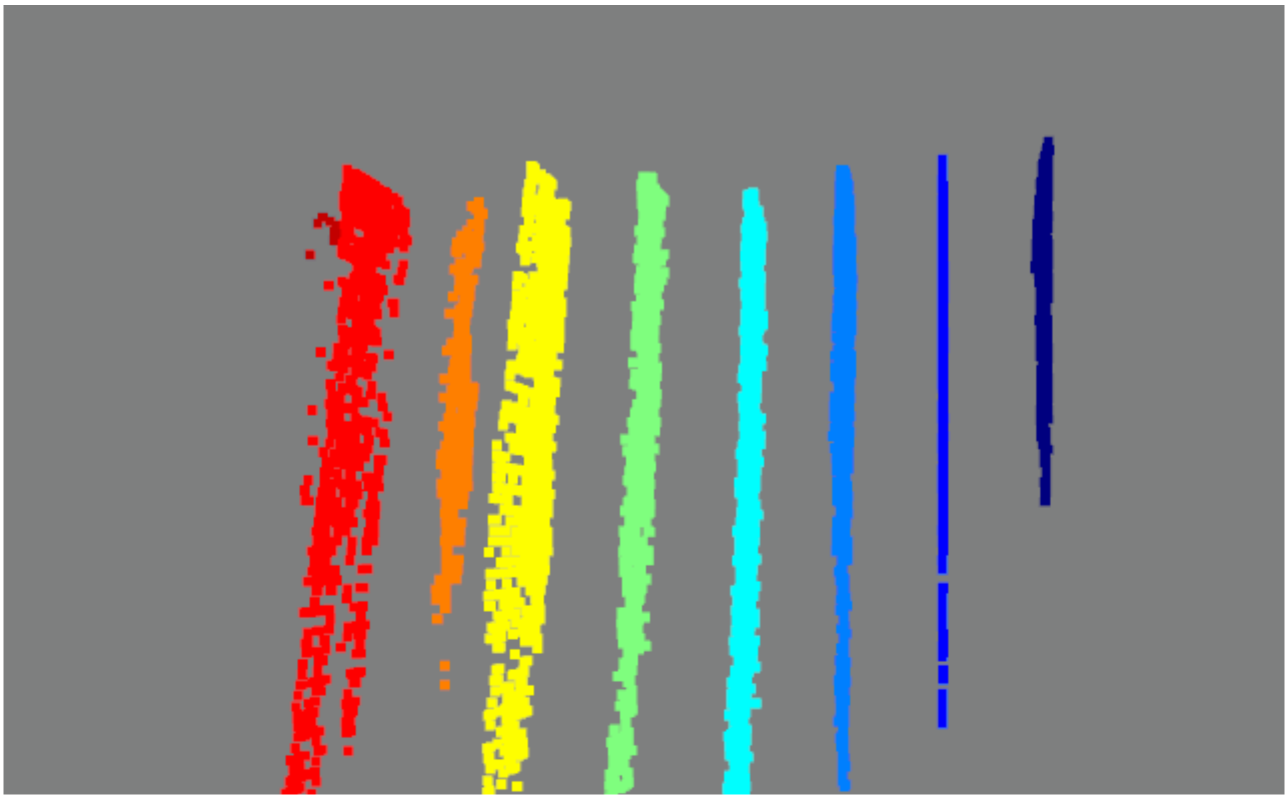
To visualize the imported mesh we need to first create a `Visualizer()` object that will contain all the 3D objects, lights, and cameras and which will listen for keyboard, mouse, and animation callbacks. The code below sets up the scene, creates a window, and imports all the 3D objects.

We can create a new window in the visualizer using the `create_window` function, which takes inputs for name, width, height, as well as position on the screen. We can then add any 3D geometry to the visualizer by calling the `add_geometry` method. In our case, we also create a primitive object — the sphere that circles the angel statue, by calling `create_sphere` . We can also create other primitives like cubes, cylinders, torus, etc. We can compute the normals of a specific 3D object by calling either `compute_vertex_normals()` or `compute_triangle_normals()` . Finally, we can translate an object by calling the `translate()` method directly on it.

Next, we set up any callback functions that we want to be run in the update cycle. In our case, we want to create an animation, so we call `register_animation_callback(rotate_around)` , where rotate_around is the name of the function. The same thing can be done with keyboard and mouse callbacks. The function is shown below.

In the function, we first create the rotation matrices for the angel mesh and the sphere by calling `get_rotation_matrix_from_xyz` and giving it the required rotation values in each direction, specified in radians. We then invoke the `rotate` method on each mesh with the created 3D rotation matrix and a center, which is the pivot around which they will be rotated. Finally, to be able to visualize these changes we invoke the `update_geometry()` method for each mesh and call the `update_renderer()` to update the whole visualizer. The full code is shown below.

Open3D is mostly directed toward the analysis and manipulation of meshes and 3D point clouds, but we can also visualize multidimensional data in 3D. We can demonstrate that by first extracting interesting columns from the weather dataset. For our purposes, we will look at the 'temperature' and 'humidity' columns. As the data contains timestamps, we use those to extract the month from which each of the weather readings has been taken. We finally transform these three columns into a numpy array, as Open3D can transform arrays to point cloud objects. The code is given below.



Temperature and Humidity by month | Image by the author

Once we have the data, we can visualize it by initializing a point cloud in Open3D. To get a better view of the points, we can also rotate the camera, by calling `get_view_control()`. The code for the visualization is given below.

## Visualization using Trimesh

Trimesh result | Image by the author

Trimesh is a purely Python-based library for loading, analysis, and visualization of meshes and point clouds. It is widely used for pre-processing of 3D assets before statistical analysis and machine learning. As well as part of applications for 3D printing like Cura. It is also a basis for other libraries that we will be discussing like Pyrender and Vedo.

Trimesh does not contain a visualization part in the base library but uses the optional pyglet game and interactive application library as a basis. It implements the visualization, callback, and manipulation features on top of it and contains a wide range of built-in visualization interactions out of the box. It also works on Linux, Mac, and Windows without any problems or workarounds. It also can be used with Python 2 and Python 3.

For installation, the only prerequisite library needed is numpy. Again, we use Anaconda environments to create an environment and install everything in it. Again, we install pyglet to be able to visualize the meshes. If you only need to analyze and manipulate meshes, you can skip the last line.

```
conda create -n trimesh_env python=3.8
conda activate trimesh_env
conda install numpy
pip install trimeshpip install pyglet
```

Once everything is installed, we can check if everything works by calling `import trimesh` . If we want to get all the messages from trimesh to the console we then need to call `trimesh.util.attach_to_log()` and a verbose output will be printed after invoking trimesh functions. This can be quite useful in debugging stages.

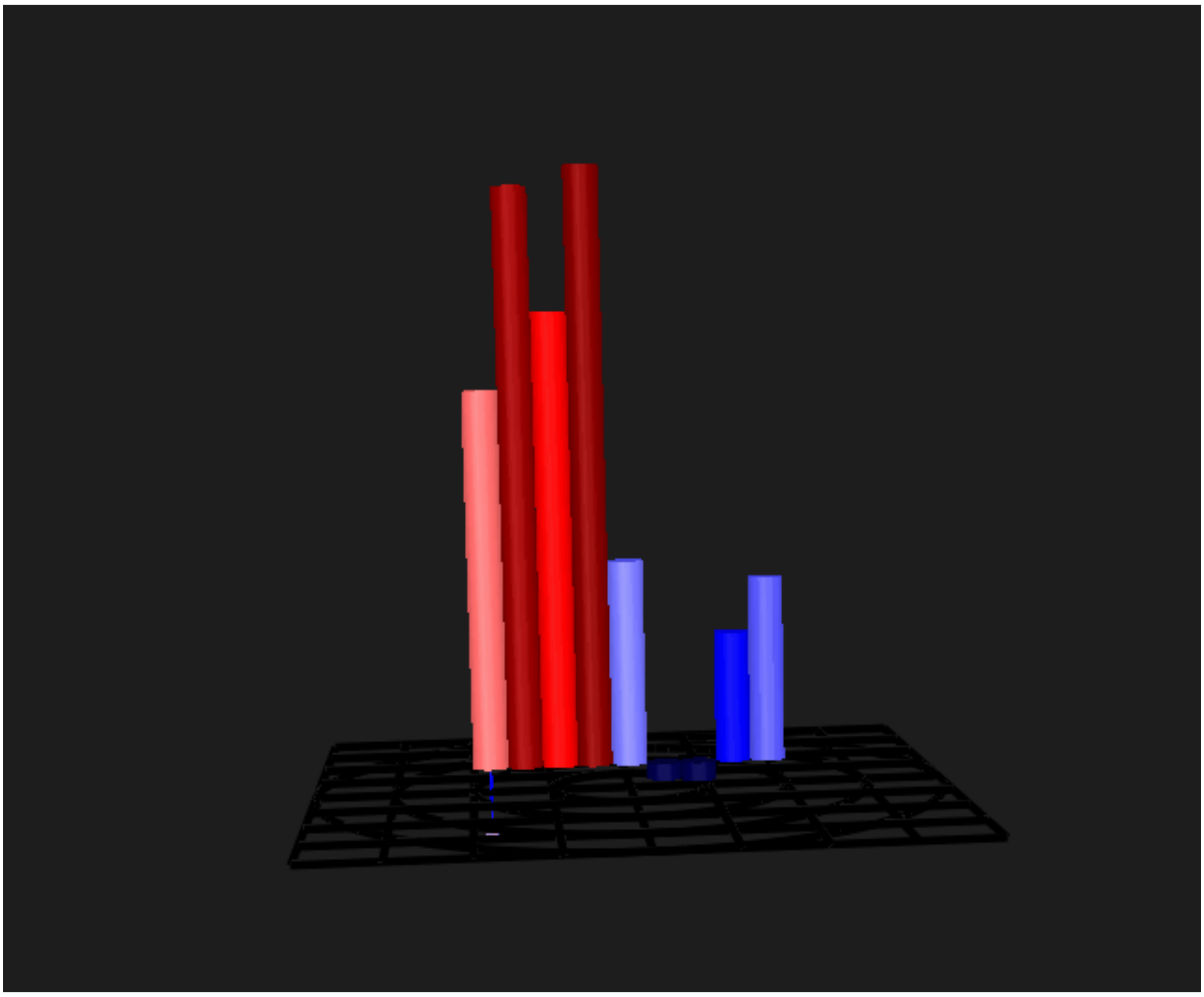
Loading meshes is done by invoking the `trimesh.load()` method, with optional inputs for file_type, where the specific mesh type can be explicitly stated. Once the mesh has been loaded, a lot of information can be directly extracted — for example, if it is watertight `mesh.is_watertight` , its convex hull `mesh.convex_hull` , volume `mesh.volume` , etc. In case, the mesh is comprised of multiple objects they can be split into multiple meshes by invoking `mesh.split()` . The code for loading the 3D mesh, creating a sphere primitive, and setting up a scene is given below.

By calling `trimesh.primitives` we can get access to different primitives like boxes, capsules, and extrusions, as well as building meshes from arrays of points and faces. In our case, we create a sphere and give it a radius. We set up a Scene that will contain the angel mesh and the sphere. As part of the scene, each of these is a separate node. Finally, we show the scene and specify a callback function called rotate_objects. The function will be invoked every time the scene is updated. It is given in the code below.

As Trimesh does not contain functions for transformations that auto-update, we need to `import time` and use time.time() to get a delta time change that we can use to update the rotations. We first create an empty homogenous transformation matrix using numpy for changing the position of the sphere. We only need to change the translation part of the matrix and we use the idea that if we change the X and Z position using a sin and cos functions we will achieve a circular motion.

We use the built-in function for generating a rotation matrix from Trimesh `trimesh.tranformations.rotation_matrix()` , which uses angle and axis as input and returns a rotation matrix. We select the nodes representing the mesh and the sphere, by selecting them based on when they were added to the scene. If we had multiple objects, we could get them based on their names from the scene or keep a reference to which object contains which node in a dictionary. We finally call the update function from the scene and give it the node that we want to update, together with the transformation matrix. The full code can be seen below.

Trimesh, the same as Open3D, is directed toward the analysis and manipulation of meshes and point clouds, but its rendering potential can be harnessed for statistical visualizations. We will demonstrate this by creating a 3D column chart for visualizing the average ‘temperature’ data for each month in the weather dataset. For this, we first calculate the mean temperature per month using Pandas’ `groupby` function and group the data based on the monthly frequency of the date-time column.


Average Temperature by month | Image by the author

Once we have extracted the average temperature readings per month, we use trimesh cylinder objects to represent the readings. The representation is given both by the height of the cylinders, but also from a color gradient going from red for highest temperatures, to blue for coldest. For generating the colormap we use the `trimesh.visual.interpolate()` function for generating normalized color values. As cylinders in trimesh are scaled from their center, we also move them down based on the average temperature, so we get a column chart that is bottom aligned. The code for the visualization is given below.

## Visualization using PyVista

PyVista result | Image by the author

PyVista is a robust and fully featured plotting and mesh analysis library, which is built on top of the Visualization Toolkit (VTK). It streamlines the VTK interface and makes the calls to the different functions easier and more pythonic. It can be used both with point clouds and meshes and it uses OpenGL making it easy to create smooth visualizations and animations. The library has a large number of examples and tutorials ranging from simple visualizations, to complex analysis and transformation tools like slicing, resampling, point cloud surface reconstruction, mesh smoothing, ray-tracing, voxelization, etc.

The library runs on Linux, Mac, and Windows and requires Python 3.7+. As The it is built on top of VTK it requires it to be installed, together with numpy as a minimum. The installation is straightforward and can be done through either pip or Anaconda. For our use case, we again create an anaconda environment and use Anaconda to install the library. This also installs all dependencies.

conda create -n pyvista_env python=3.8
conda activate pyvista_env
conda install numpy
conda install -c conda-forge pyvista

Once the library is installed, it can be tested by first importing it `import pyvista as pv` and then calling `pv.demos.plot_wave()` if the plotting window with a sine wave starts then the library has been successfully installed.

Meshes and point clouds can be easily imported by utilizing the connection to meshio. To import them we can just call `pyvista.read()` and to load the texture of the mesh we can just call `pyvista.read_texture()` . Once the mesh is loaded additional information can be extracted from it like edges, mesh quality, distances between vertices, etc. It can be also easily resampled.

An important point we need to mention here is that if we want a non-blocking visualization using PyVista for animations or for implementing keyboard, mouse, or UI interaction we need to install pyvistaqt, which extends PyVista with Qt functionality. The add-on library can be easily installed by calling:

```
conda install -c conda-forge pyvistaqt
```

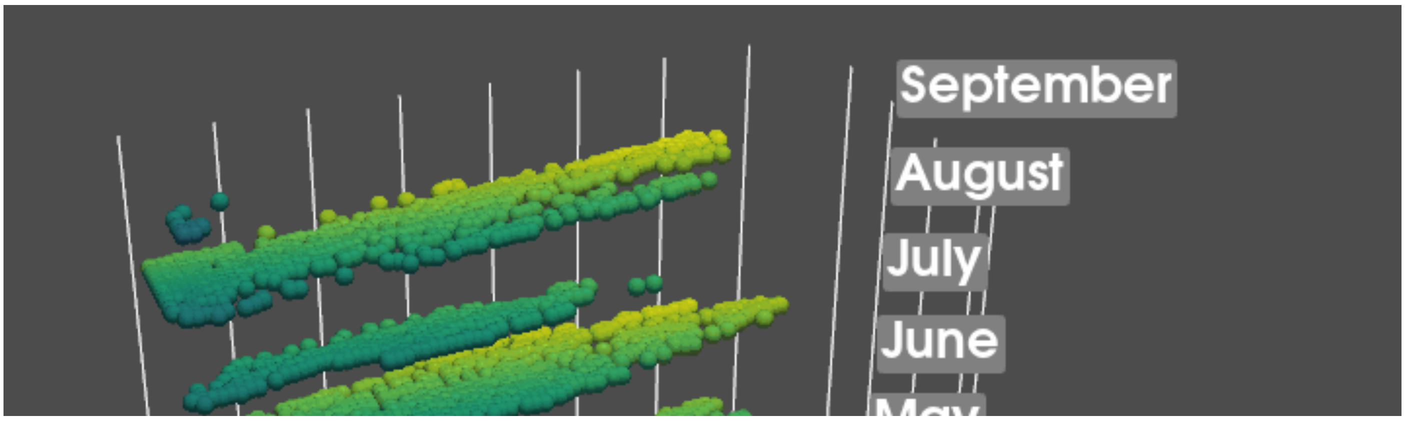
Once this is done we can create non-blocking plots using background plotter structures by importing them with `from pyvistaqt import BackgroundPlotter`. If you do not need non-blocking visualization then calling directly `pyvista.Plotter` and using it is enough. The code for loading the angel mesh and texture, creating a background plotter, orbiting sphere, and a light source is given below.

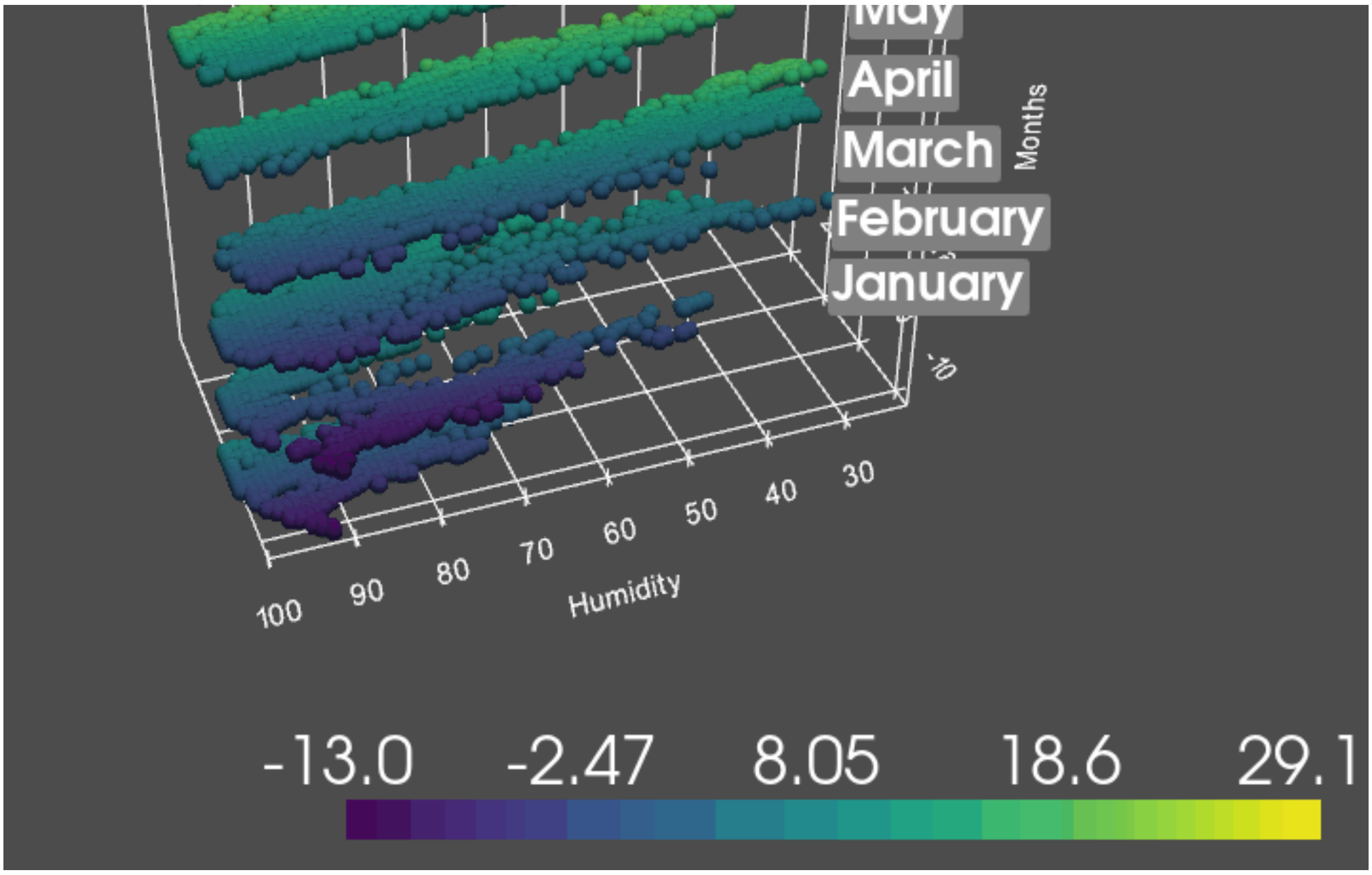
Once we create the `BackgroundPlotter` we can set up its size, background, etc. We change the camera_position so it stays at the *xy* axis. This can be done by also setting up a tuple of camera positions in X,Y,Z, as well as rotation or up direction. We also set the clipping rage of the camera, as sometimes the objects are clipped by the default values. We create a sphere primitive by giving it a radius and center position. Other primitives can be created the same way — cylinder, arrow, plane, box, cone, disc, etc. Finally, we create a point light by invoking `pyvista.Light`, the default is to create a point light, but we can implicitly create directional, spotlight, etc. We set the color of the light, the position, and the focal point. Next, we add all these objects to the plotter and create a callback listener for updating the scene and generating the simple animation.

Here we need to specify two important points. When creating a callback with `add_callback` it is highly advised to explicitly specify the interval to which the update cycle will work. The second thing is when using the pyvistaqt plotter, always call `app.exec_()` after the show function, as this ensures that the created plotter window will not close after one iteration. Finally, the function invoked in the callback `update_scene()` is given below.

Here we use the simplified rotation functions exposed by PyVista for rotation on each axis `rotate_y` for the mesh and the sphere, together with calling `implace=True` to ensure the rotations will be set directly to the meshes. As the light does not contain a method for direct rotation, we update its position based on the position of the `sphere.center`. Finally, we update the whole plotter. The full code of the PyVista visualization can be seen below.

PyVista is directed towards the analysis and visualization of 3D data, which makes it a perfect library for the generation of visuals for multidimensional and time-series datasets. We will demonstrate that by building upon the simple visualization created using Open3D in the previous chapter. For this we extract the same columns from the weather dataset — the 'temperature', and 'humidity' and we generate a 'month' column from the datetime information. Once this initial setup is done, we can directly use PyVista to create a 3D plot. PyVista has many similar features to 2D plotting libraries like matplotlib and seaborn, where values can be directly used to create pseudo-coloring and labels can be added directly to points. The initial Pandas manipulations are given in the code below.

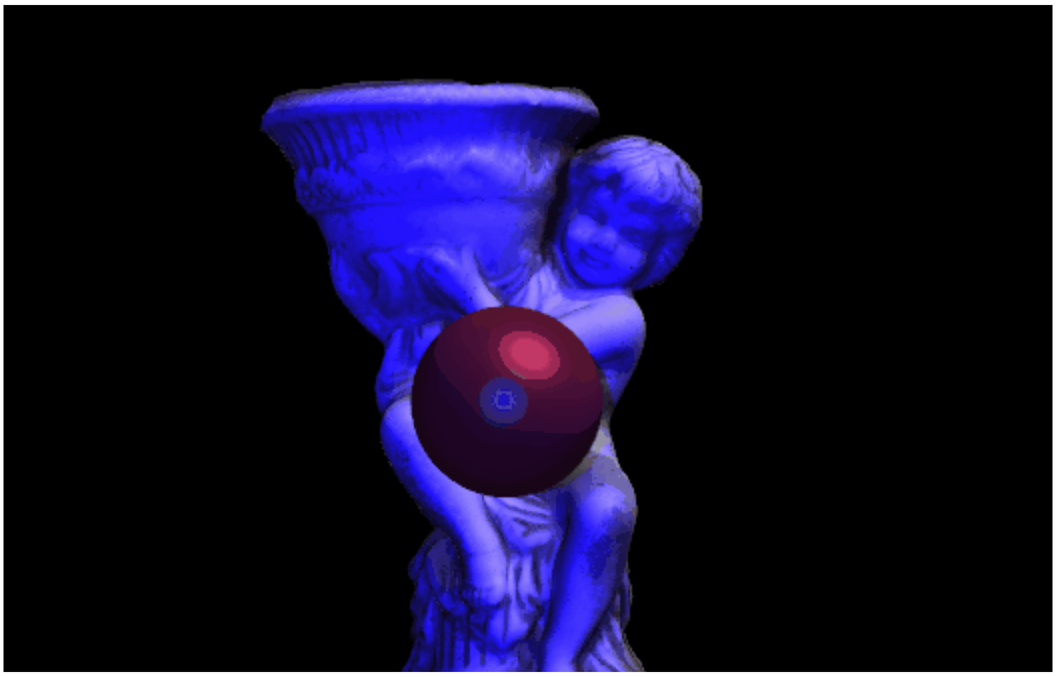


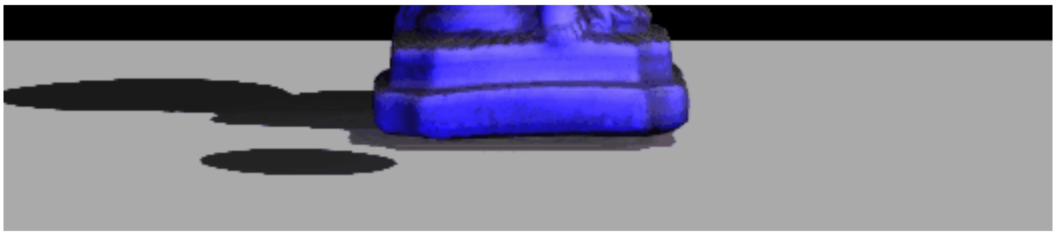


Temperature and Humidity by month with 3D grids and a legend | Image by the author

Once the pandas data has been pre-processed and required information extracted, we set up a 3D point cloud object to contain the selected dataset columns by calling `pyvista.add_points` . The function can also directly take a `scalars` field which is used to color the points and create a legend. We then set up label objects and give them a position at each month's level. We finally initialize a 3D grid, name the axis and create a simple callback function for rotating the camera. The code for the visualization and the callback function is given below.

## Visualization using Vedo




Vedo result | Image by the author

Vedo (or V3do) is a Python library for scientific analysis and visualization of 3D objects. It can be used for the plotting of 1d, 2d, and 3d data, point clouds, meshes, as well as volumetric visualization. Vedo uses Trimesh as a backend to some of the mesh and point cloud processing, import/export, and generation and builds upon it by providing a wide array of scientifically directed functionality. It can produce vector illustrations and visualization that be directly imported into Latex and is used for generating physics simulations, deep learning layer overviews, and mechanical and CAD-level slices and diagrams.

Vedo is based on VTK and requires it and numpy to be present before installation. It works on Linux, Mac, and Windows and requires Python 3.5 and above. For our use case, we again create an anaconda environment and use Anaconda to install the library.

```
conda create -n vedo_env python=3.8
conda activate vedo_env
conda install numpy
conda install -c conda-forge vedo
```

Once everything is installed, we can test the installation by calling `import vedo` and then visualizing a primitive `vedo.Sphere().show(axis=1).close()` . This is just a shorthand for generating a sphere, directly showing it with an axis, and then setting up that the program will end once we close the window. If we see a sphere with an axis next to it, everything works as intended.

Vedo can be run in a CLI interface by directly calling `vedo path_to_your_3D_object` . Once a window is opened then there are a number of keys and combinations that can be pressed to change the visualization, together with mouse rotation, translation, and zooming. Finally, Vedo can be directly called for file format transforming by calling `vedo --convert path_input_3D_object --to path_output_3d_object` or directly adding the required file format after the `--to` .

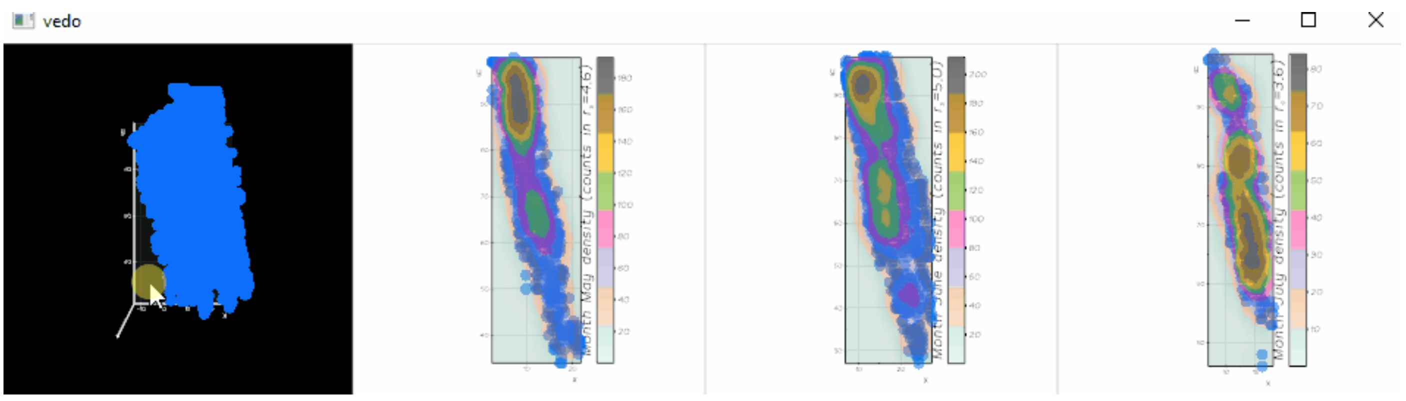
Loading meshes and point clouds is straightforward using `vedo.load(path_to_3d_object)` . If the mesh has a texture, it needs to be additionally loaded to the created 3D object using `mesh.texture(path_to_texture)`. Once the mesh is loaded then a number of fast interactions and analysis steps can be done. A cutter tool for cutting parts of the mesh using “cookie-cutter” shapes — sphere, plane, box, by calling `addCutterTool(your_mesh, mode = "sphere")` . Another one is a freehand cutter where meshes can be manually segmented using `FreeHandCutPlotter(your_mesh)` . Other manipulations on the mesh can be done as well — like filling holes by calling `your_mesh.fillHoles(size=size_to_fill)` . Primitives can be also easily generated by calling `vedo.Sphere` or `vedo.Box` for example. The code for loading the angel mesh and generating primitives is given below.

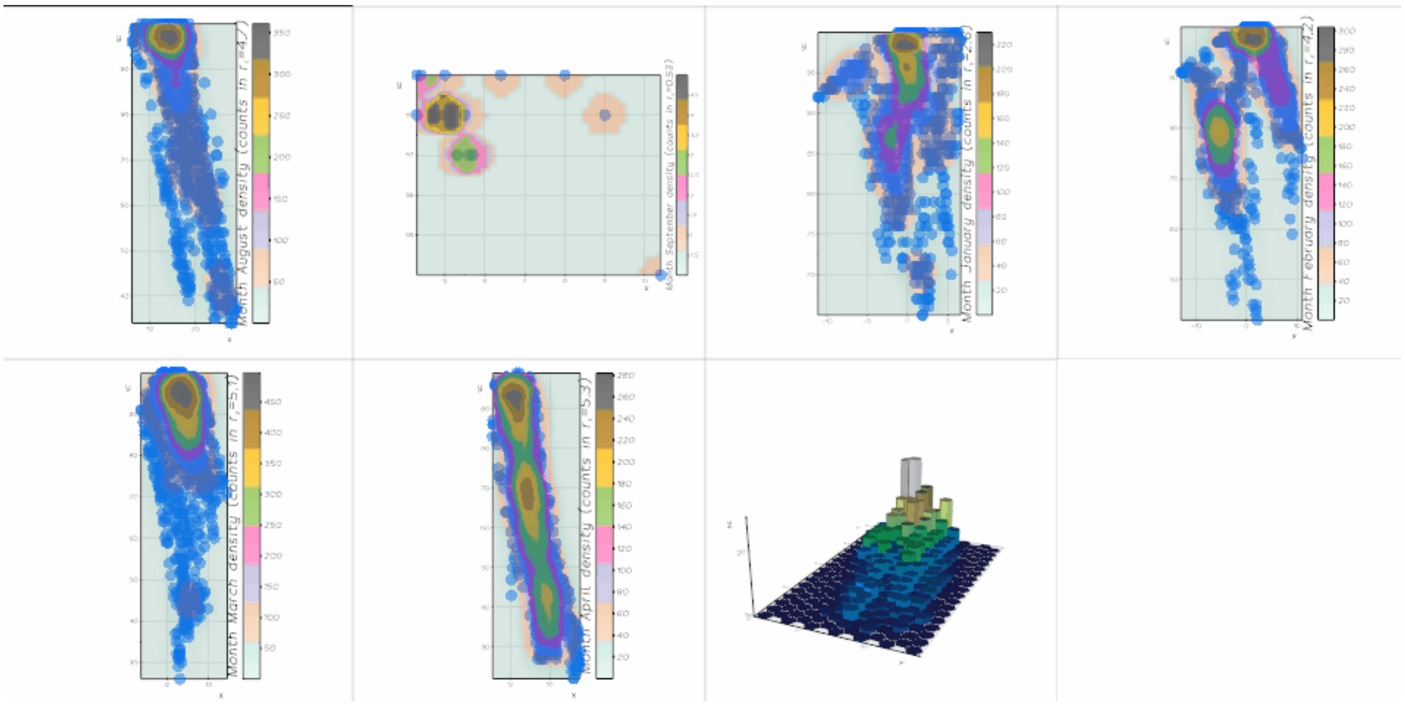
The material of the meshes, together with their properties like color, metallic and roughness can be set up by calling the `lighting()` method of the 3D object. A number of pre-made ones are available — "default", "metallic", "plastic", "shiny", "glossy", with also the possibility to create custom ones. We also create a box as a surface on which the angel mesh stays on and to cast shadows, as well as a sphere. We can move objects by either directly invoking `rotate` and `translate` or by calling `x(), y(), z()` methods. Once all the 3D objects are created and positioned, we make two lights and set up the plotter object for visualizing everything. The code for this is shown below.

We create two point lights — one white and with less intensity to work as ambient light and one stronger, blue one that will be moving with the sphere. We set in the Vedo settings that we want to enable interactivity. We create a plotter and specify to generate the shadows by calling `plotter.addShadows()` . Finally, we call the show method, together with all the 3D objects and lights that want to display. We can also add them as a list if a large number of objects need to be generated. We set the interactivity to 0, as in the version of Vedo I use, if this is not done, the animation would sometimes not run. Finally, as Vedo is built upon VTK it uses its event system. The difference from PyVista is that Vedo does not have an implicit timer function that would be run every update loop. This is why we need to create our timer function first by calling `plt.timerCallback('create', dt=delta_time_for_updating)` . Once that's created we can add a callback function to this timer. This function will run every time the timer is invoked. Please observe that the "timer" name needs to be written as seen for the callback to work. The function `rotate_object` is given below.

In the function, we again use a shorthand method `rotateY` to rotate the angel statue and the sphere. In the method, we can also set the pivot around which the object is rotated. As the light does not contain a rotation method we take the sphere's position and directly set the light position to it. We finally invoke the `plt.render()` function so the renderer can be updated. The full code is given below.

Like PyVista, Vedo is a specialized library for visualizing data and datasets in 3D. It extends a lot of the functionality present in PyVista, by giving the possibility to create statistical analysis visualizations easily by just calling functions on the data. It also has a better implementation for adding labels and text, together with scientific notations and even LateX. Both Vedo and PyVista can also leverage VTK, to create multiple subfigures and renderings on top of renderings. We will explore this by creating multiple subfigures for the 'temperature' and 'humidity' data together with data density plots and combining them with a histogram visualization. We prepare the data the same way as with the previous examples — initial extraction of columns using pandas and creating month and month name columns. The code is shown below.


vedo
Month May density (counts in r_s=4.6)
Month June density (counts in r_s=5.0)
Month July density (counts in r_s=3.6)



Temperature and Humidity density plots, together with a 3D histogram | Image by the author

Once the data has been pre-processed, the visualizations can be created by first calling the Vedo `Plotter` object and setting the number of subfigures that will be created. Here we also specify that each subfigure would have a separate camera by setting `sharecam = 0`. This makes the visualization heaver but each of the subfigures can be manipulated separately. Then for each month, we create a plot with 3D axis and we call the `.density()` function which calculates the point density volume, which can be then overlayed, together with a scalar bar. For the final subfigure, we call the `vedo.pyplot.histogram` together with the dataset, visualization mode, and axis names. The code for the visualization is given below.

## Conclusion

When I started writing this article it was planned as just one part and then going to different libraries for analyzing and manipulating meshes and statistical data in 3D. I decided to split this article into 2 parts so I can better explain the finer details of these libraries. This is important because no matter what you want to do — pure result visualization [1, 6], generation of 3D features from meshes and point clouds [2], or creating high-definition renders [3], it is important to know the tools that are at your disposal for 3D tasks in Python. The angel statue can be freely used for visualization and experimentation, together with a large number of other 3D reconstructed objects using SfM from [4] and [5]. More information for the weather time-series dataset can be read HERE.

Next Part 2 will focus on the next 4 visualization libraries — Pyrender, PlotOptiX, Polyscope, and Simple-3dviz. From those, we will put a lot of emphasis on PlotOptiX and Pyrender, which can produce ray-tracing results (PlotOptiX) or highly-dynamic lighting scenes (Pyrender). We will also talk about Polyscope and Simple-3dviz as lightweight and simple no-hassle 3D visualizers for everyday use and scientific data presentation.

## References

**Nikolov, I.**, & Madsen, C. (2016, October). Benchmarking close-range structure from motion 3D reconstruction software under varying capturing conditions. In *Euro-Mediterranean Conference* (pp. 15–26). Springer, Cham; https://doi.org/10.1007/978-3-319-48496-9_2

**Nikolov, I.**, & Madsen, C. (2020). Rough or Noisy? Metrics for Noise Estimation in SfM Reconstructions. *Sensors, 20*(19), 5725; https://doi.org/10.3390/s20195725

**Nikolov, I. A.**, & Madsen, C. B. (2019, February). Interactive Environment for Testing SfM Image Capture Configurations. In *14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (Visigrapp 2019)* (pp. 317–322). SCITEPRESS Digital Library; https://doi.org/10.5220/0007566703170322

**Nikolov, I.**; Madsen, C. (2020), “GGG-BenchmarkSfM: Dataset for Benchmarking Close-range SfM Software Performance under Varying Capturing Conditions”, Mendeley Data, V4; https://doi.org/10.17632/bzxk2n78s9.4

**Nikolov, I.**; Madsen, C. (2020), “GGG — Rough or Noisy? Metrics for Noise Detection in SfM Reconstructions”, Mendeley Data, V2; https://doi.org/10.17632/xtv5y29xvz.2

**Nikolov, I.**, Philipsen, M. P., Liu, J., Dueholm, J. V., Johansen, A. S., Nasrollahi, K., & Moeslund, T. B. (2021). Seasons in Drift: A Long-Term Thermal Imaging Dataset for Studying Concept Drift. In *Thirty-fifth Conference on Neural Information Processing Systems;* https://openreview.net/forum?id=LjjqegBNtPi