

PyMOLViz

PyMOLViz is a library to simplify displaying arbitrary meshes in PyMOL.

Installation

In order to install the module you should navigate to the `module` folder and type:

```
pip install -e .
```

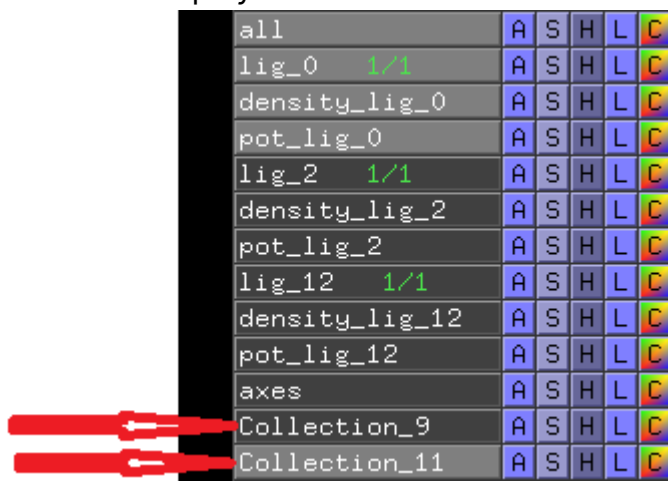
```
~/PyMolViz$ cd module/  
~/PyMolViz/module$ pip install -e .
```

This should install the module in your current environment and automatically load any updates to the module.

Introduction










At the base of the library stand *Meshes* which can be sets of Points, Lines or arbitrary meshes defined as vertices, colors, vertex normals and face indices.

Meshes are parts of *Collections*. Collections describe an arbitrary amount of meshes which are displayed under the same name in PyMOL.



all	A	S	H	L	C
lig_0 1/1	A	S	H	L	C
density_lig_0	A	S	H	L	C
pot_lig_0	A	S	H	L	C
lig_2 1/1	A	S	H	L	C
density_lig_2	A	S	H	L	C
pot_lig_2	A	S	H	L	C
lig_12 1/1	A	S	H	L	C
density_lig_12	A	S	H	L	C
pot_lig_12	A	S	H	L	C
axes	A	S	H	L	C
Collection_9	A	S	H	L	C
Collection_11	A	S	H	L	C

Collections can be added to *Scripts* which represent a single python script, loading the contained collections.

 force_partitioning		28.02.2023 09:02	Python File	416 KB
 force_partitioning		28.02.2023 09:02	SDF-Datei	2.079 KB
 Contributions		27.02.2023 09:39	Python File	123.409 KB
 test_set_error_distribution_favorable_		27.02.2023 09:37	PNG-Datei	88 KB
 force_partitioning_saliency		26.02.2023 17:36	Python File	179.223 KB
 X_Bond_Contributions		25.02.2023 15:46	Python File	39.958 KB

Creating Meshes

As a goal of the library was to accelerate visualization in PyMOL we tried to make creating meaningful meshes as fast as possible.

Creating Point Sets

Visualizing a set of Points can be done via:

```
import numpy as np
import pymolviz as pmv

vertices = np.random.random(20, 3)

points = pmv.Points(vertices)
```

To quickly generate a loadable script you can convert every mesh directly to a script and write it to disk:

```
points.to_script().write("test_points")
```

When converting to a script without giving a name for the corresponding collection in pymol, a warning will be displayed and a default name will be provided:

WARNING:root:No name provided for Collection. Using default name: Collection_10. It is highly recommended to provide meaningful names.

If you want to provide a name for the mesh (actually the corresponding collection) you can pass it as an argument to the to_script call:

```
points.to_script(name="Random Points").write("test_points")
```


PyMOL also offers a fairly efficient way to render points as spheres instead of single pixels. If you want to render points as spheres you can pass *render_as_spheres = True* to the

Points constructor. The default radius is 0.1. You then can further specify the spheres' radii as r via: `sphere_radius = r`.

```
points = pmv.Points(vertices, render_as_spheres=True, sphere_radius = 0.3)
```

Colors

Every Mesh allows also to pass a color value. PyMOLViz tries to interpret different color types gracefully. The following types are supported:

- Strings:
 - Names of matplotlib colors
 - Element Names or Symbol, giving colors from the PyMOL element color codes
 - e.g. Br, mn, Zinc, hydrogen
 - Arrays:
 - Either 3 or 4 element containing arrays describing the color as rgb contributions from 0 to 1. While alpha values are accepted, they are currently not applied.
 - Lists/Arrays of colors (strings or arrays):
 - Instead of providing a single color for all points, you can also pass lists (or arrays) of strings or 3/4 element arrays. These should then have a sensible format, usually the same length as the number of vertices passed.
 - Lists/Arrays of scalar values:
 - When given a list of scalar values, the values will be mapped to a colormap and colors determined accordingly. By default the *coolwarm* matplotlib colormap will be applied, however any colormap can be passed using the keyword *colormap*.
 - **coolwarm** 
 - See <https://matplotlib.org/stable/tutorials/colors/colormaps.html> for other existing color maps. You can also pass a custom colormap.
 - Additionally you can pass the upper and lower limit of the colormap as a tuple to the *clims* keyword.
 - You can extract the actually used colormap from the mesh via `get_color_map()`.
- When you save it to an image, make sure to use the `bbox_inches='tight'` option to also properly show the tick values.**

```
# reading coordinates and electrostatic surface potential from a pandas dataframe
coords = np.array(df_pot[['x', 'y', 'z']].values)
potential = df_pot['potential'].values

# plotting points at coords with potential determining their color
# the colormap is set to rainbow instead of the default coolwarm
```

```

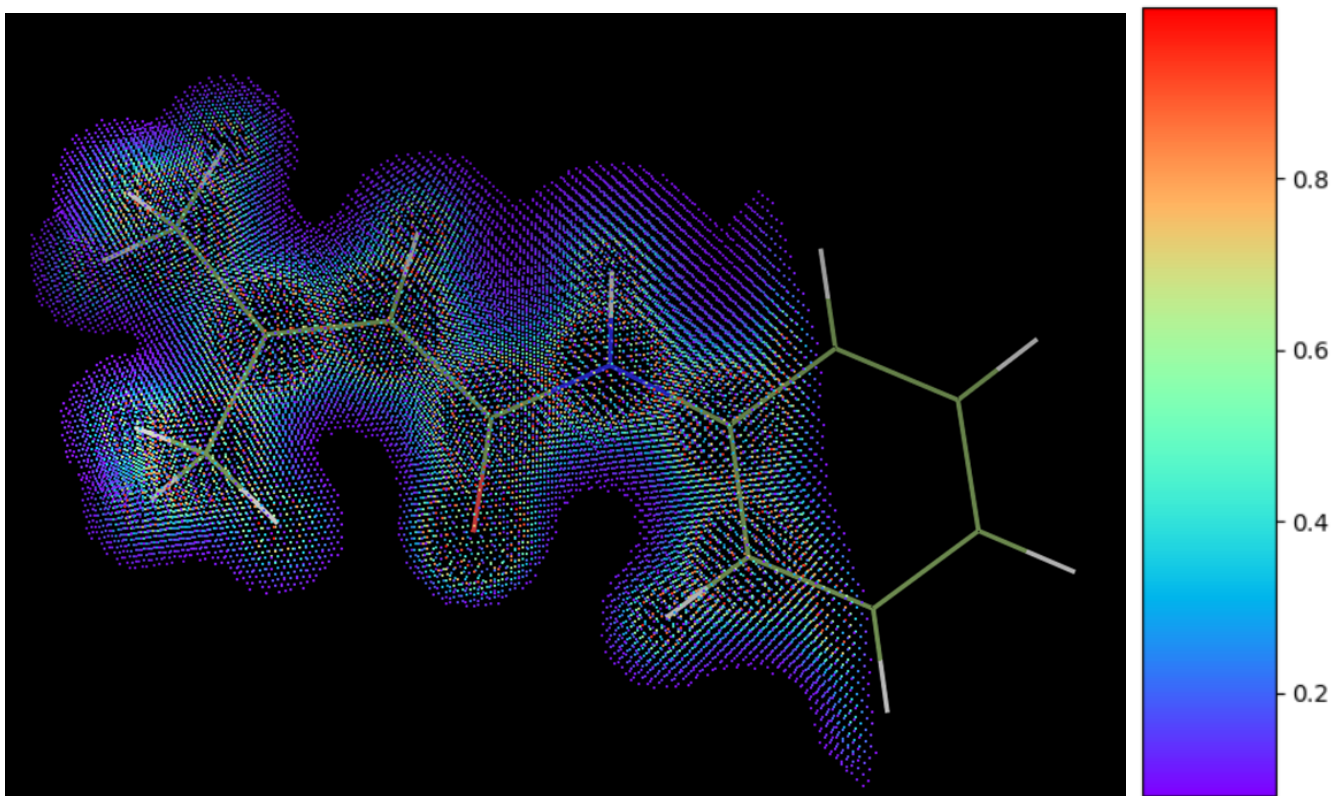
points_pot = pmv.Points(coords, potential, colormap = 'rainbow')

# direct conversion of the generated points to a script, that is then also
# immediately
# written to a file
points_pot.to_script(f'pot_test').write(f'pot_test.py')

# extracting the colormap used
fig = points_pot.get_color_map()

# storing the colormap image to a file, make sure to always pass
bbox_inches='tight'
fig.savefig(f'colorbar/pot_test.png', bbox_inches='tight')

```



Creating Lines

Another type of "mesh" are lines. These are created by passing pairs of points the Lines constructor. A line will be drawn between each pair of points. So for [a,b,c,d] we would get a-b c-d.

The shape of the input array can be otherwise arbitrary as it will be reshaped (flattened) in the constructor.

While it is possible to pass colors as previously described, the Lines constructor also allows to pass a color for each line (so the number of colors should be half of the number of passed vertices).

```

import pymolviz as pmv
import numpy as np

# generating dummy data
start_points = np.random.random((5, 3)) * 10
end_points = np.random.random((5, 3)) * 10 + 5

# setting colors via strings. Note that for 10 points, we pass only 5 colors, one
for each line
color = ["red", "blue", "green", "violet", "yellow"]

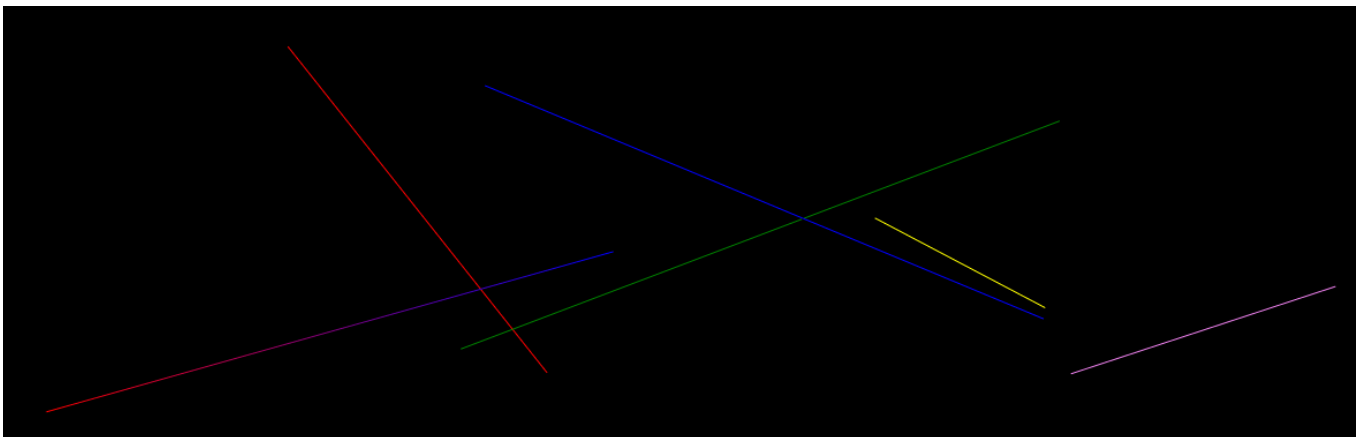
# the way that points are passed here to the Lines constructor is fairly elegant,
as we often have start and end points separately. Here they are joined in such a
way that each start point is followed by its corresponding end point. While the
shape of the array is (2, 5, 3) it will automatically be reshaped to (-1, 3).
sc_lines = pmv.Lines(np.array([start_points, end_points]), color = color)

# alternatively we can pass a color for each point, creating a gradient colored
line from the origin [0,0,0] to [5,5,5]. (bottom left line in the example image)
gradient_line = pmv.Lines(np.array([0,0,0, 5,5,5]), color = ["red", "blue"])

# as we like to show both line meshes "gradient_line" and "sc_lines", we add them
to one script and then write that script to a file.
s = pmv.Script([sc_lines, gradient_line], name = ["single_color_lines",
"grad_line"])

s.write("test_lines.py")

```



Creating Arrows

While it is nice to show relations between points via lines, one is often also interested in the direction of that relation. For this one can use the Arrow mesh. This will generate 4 additional lines indicating the head of the arrow. The width and length of the additional 4 lines can be controlled by the `head_length` and `head_width` parameters.

```

import pymolviz as pmv
import numpy as np

# generating dummy data
start_points = np.random.random((5, 3)) * 10
end_points = np.random.random((5, 3)) * 10 + 5

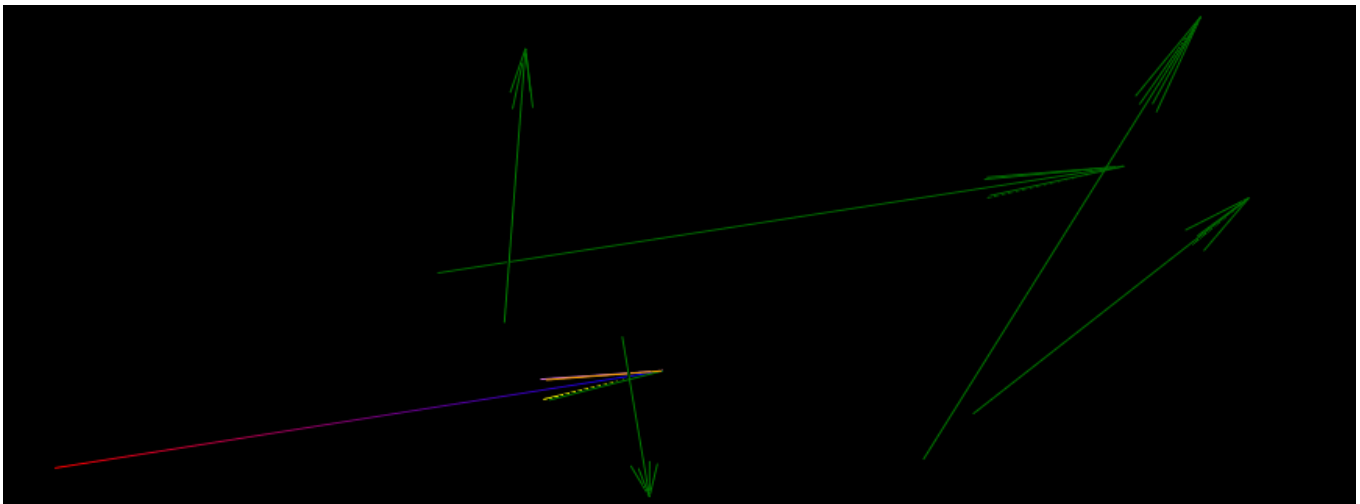
# setting colors is similar to that of lines. Here we pass a single color for all
arrows
# start and end points can be passed as for lines
sc_arrows = pmv.Arrows(np.array([start_points, end_points]), color = "green")

# alternatively we can pass a color for each point, creating a gradient colored
arrow from the origin [0,0,0] to [5,5,5] with each head line having a different
color. (bottom left line in the example image)
# colors are set as main line colors then head line colors.
all_arrow_colors = ["red", "blue", "green", "green", "violet", "violet", "yellow",
"yellow", "orange", "orange"]
gradient_arrow = pmv.Arrows(np.array([0,0,0, 5,5,5]), color = all_arrow_colors)

# as we like to show both line meshes "gradient_line" and "sc_lines", we add them
to one script and then write that script to a file.
s = pmv.Script([sc_arrows, gradient_arrow], name = ["sc_arrows", "grad_arrow"])

s.write("test_arrows.py")

```



Creating Actual Meshes

In addition to the basic data representations presented thus far, PyMOLViz also provides an Interface to the Mesh representation in PyMOL. Such a mesh has, in addition to points and colors also faces and vertex normals.

Faces should be a list or array of 3 dimensional indices into vertices. Faces indicate which 3 vertices should be connected to generate an opaque triangle.

As a mesh will display an actual surface the additional information of the direction of the surface is used to compute lighting on the surface. For this, **vertex normals** can be provided which should for every provided vertex indicate the surface normal at that vertex. If you have no information of vertex normals you can either compute them from surrounding vertices or just pass zero vectors.

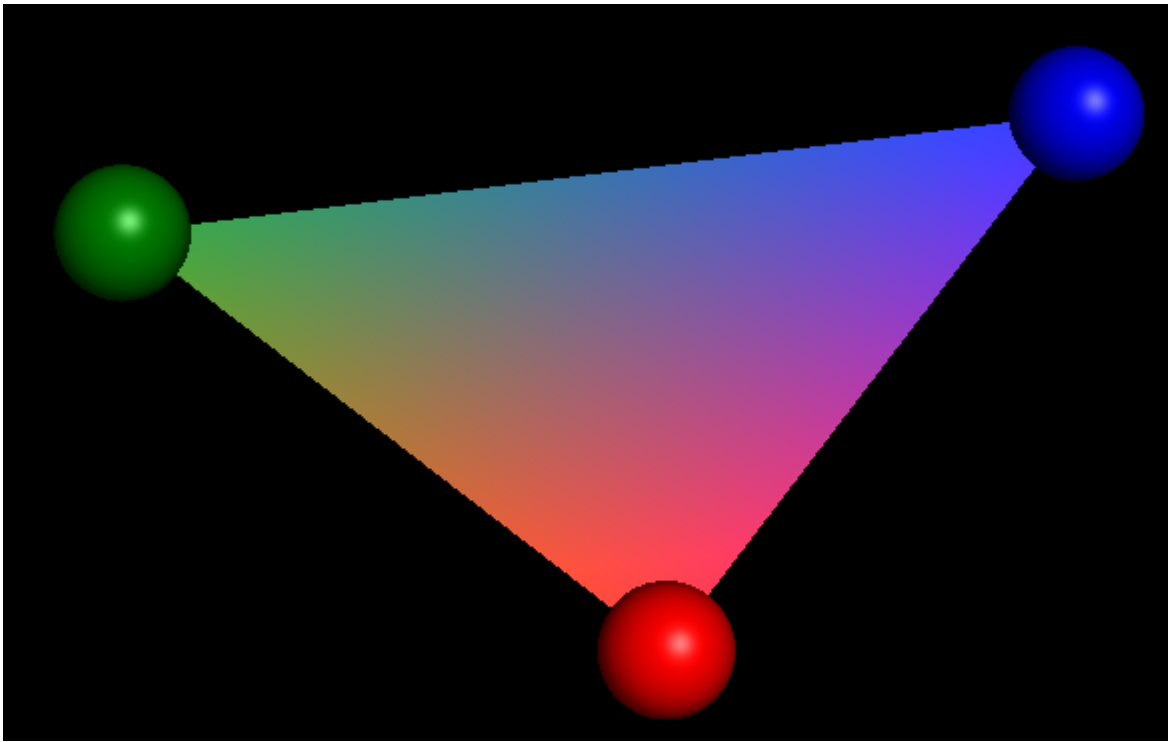
```
import pymolviz as pmv
import numpy as np

# creating dummy points
points = np.array([[0, 0, 0], [0, 0, 1], [0, 1, 0]])
a, b, c = points
colors = np.array(["red", "green", "blue"])

# creating a point mesh of the dummy points, rendering the points as spheres
point_mesh = pmv.Points(points, color = colors, render_as_spheres=True)

# creating the actual mesh. As we dont know the vertex normals, we set them to 0
vectors. The faces input means, the points at index 0, 1 and 2 should be connected
to create a triangle. Any color between vertices is interpolated (by PyMOLs
shaders).
mesh = pmv.Mesh(points, faces = [[0, 1, 2]], normals = np.zeros_like(points), color
= colors)

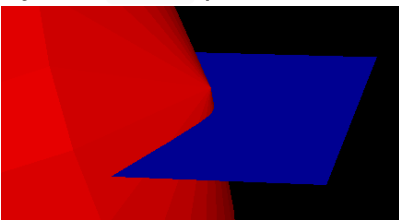
# Once again we combine the meshes into a script.
s = pmv.Script([point_mesh, mesh], name = ["mesh_points", "mesh"])
s.write("test_mesh.py")
```



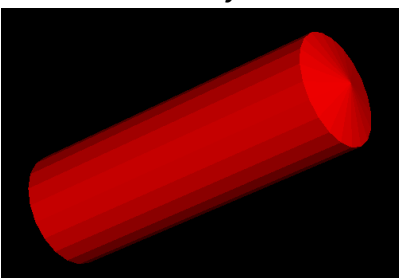
Mesh Primitives

PyMOLViz currently also provides the *Plane*, *Cylinder* and *Sphere* classes, which allow to instantiate cylinder and sphere meshes more easily.

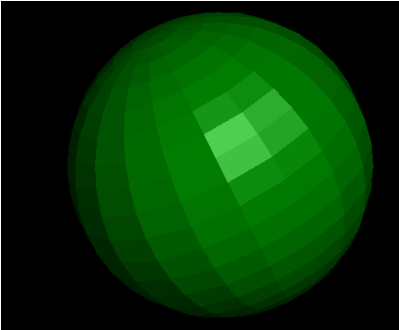
- The *Plane* class represents a mesh consisting of two triangles which are placed at a given `position` and with a given plane `normal`. The size of the triangles is controlled by the `scale` parameter.



- The *Cylinder* class is effectively a 3d line. Similarly to the *Lines* class, a cylinder is defined by a `start` and `end` point. In addition to this a `width` parameter determines the cylinders width. The `curvature` controls how rounded the ends of the cylinder will be (for $\neq 0$ the resulting mesh will not be a *true* cylinder). The `resolution` parameter determines how many points on a circle will be used to approximate the circles at the cylinders ends.



- The *Sphere* class allows to create a sphere directly as a mesh (allowing to e.g. extract its wireframe representation). If you want to represent many points as spheres, you should use the *Points* class with `render_as_spheres = True` instead. The *sphere* is defined by its `position`, `radius` and its `resolution`, determining how many points are approximating a circle.



```
import pymolviz as pmv
import numpy as np

start_point = np.array([1,1,1])
end_point = np.array([2,2,2])

# creating cylinder
cylinder = pmv.Cylinder(start_point, end_point, 0.3, curvature=0.05, color = "red")

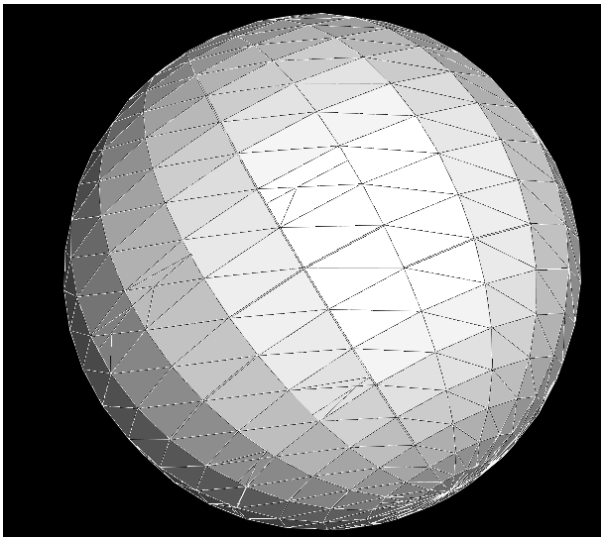
# creating plane
normal = np.array([1,1,-1])
plane = pmv.Plane(start_point, normal, scale = 0.5, color = "blue")

# creating sphere
sphere = pmv.Sphere(start_point, 0.3, color = "green")

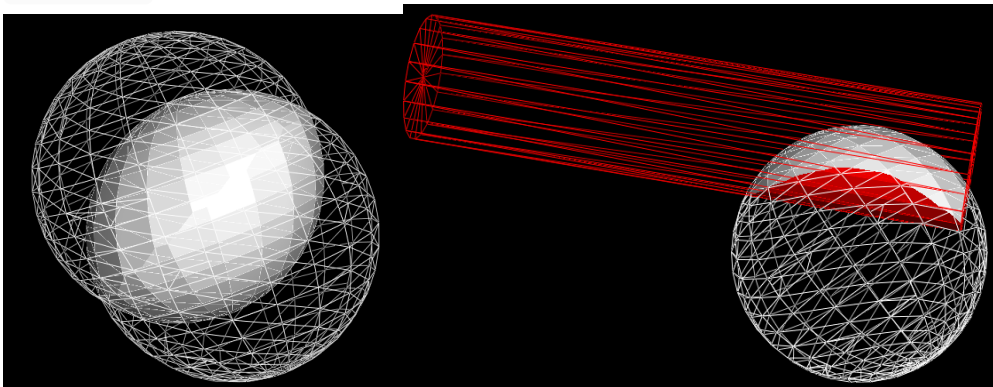
# joining meshes into a script
s = pmv.Script([cylinder, plane, sphere], name = ["cylinder", "plane", "sphere"])
s.write("test_primitives.py")
```

Mesh Functions

You can get the wireframe of a given mesh via the `to_wireframe()` method, which will return an instance of the `Lines` class.



Rudimentary *constructive solid geometry* is implemented via the `union`, `difference` and `intersect` methods, but is for most cases unfeasibly slow in python.



Meshes can be combined with other meshes to form new meshes via `Mesh.combine(meshes)`. However, Point meshes, Line meshes and regular meshes cannot be sensibly combined.

Using the `to_script()` method a mesh can be directly turned into a Script instance, which can then be written to a file using its `write` method (see the example before). To be more precise, the mesh is added to a collection which is then added to a newly created script, which is then returned. To name the newly created collection you can pass an argument to `to_script(name=name)`.

Collections

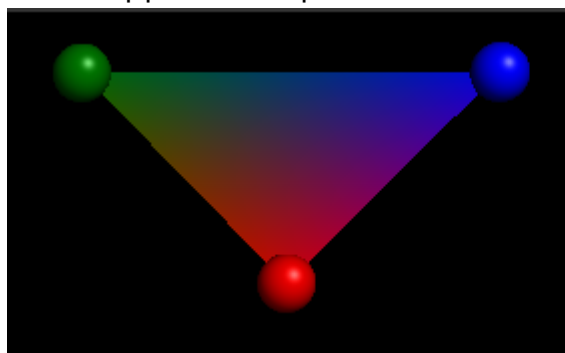
Sometimes one would not want to have each mesh as a single PyMOL object. For this case, collections can be used. Collections are named sets of meshes, which will be displayed in PyMOL as a single named object.

Meshes can be combined to a collection by either passing it to a collections constructor or adding them to a collection instance via its `add` method.

all	A	S	H	L	C
lig_0 1/1	A	S	H	L	C
density_lig_0	A	S	H	L	C
pot_lig_0	A	S	H	L	C
lig_2 1/1	A	S	H	L	C
density_lig_2	A	S	H	L	C
pot_lig_2	A	S	H	L	C
lig_12 1/1	A	S	H	L	C
density_lig_12	A	S	H	L	C
pot_lig_12	A	S	H	L	C
axes	A	S	H	L	C
Collection_9	A	S	H	L	C
Collection_11	A	S	H	L	C

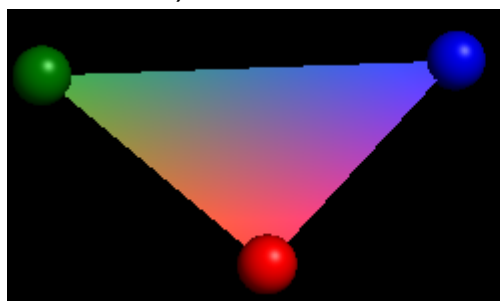
In the case of the simple mesh example above, the points and the triangle between them would appear as separate instances in PyMOL:

all	A	S	H	L	C
mesh_points	A	S	H	L	C
mesh	A	S	H	L	C



If we would want to have them under the same name, we could instead of adding the meshes separately to the script instance, add them to a collection and then add the collection to a script (or get the collection as a script using the `to_script` method of Collections).

all	A	S	H	L	C
mesh	A	S	H	L	C



```
collection = pmv.Collection([point_mesh, mesh], name = "mesh")
collection.to_script().write("test_mesh_collection.py")
```

Or:

```
collection = pmv.Collection(name = "mesh")
collection.add(point_mesh)
collection.add(mesh)
collection.to_script().write("test_mesh_collection.py")
```

Opacity

Collections are also used to provide opacity (akin to color alpha values) to the objects. Unfortunately PyMOL can only assign opacity to Collections and not single meshes. Opacity is assigned to Collections via the constructors `opacity` keyword. Or the corresponding instances `opacity` attribute. Be aware that opacity is weird in PyMOL by default and you may have to hide and show objects with opacity for it to properly apply.

States

Collections can also be used to make use of PyMOLs state system. In order to have different states of the same Collection, you can explicitly pass a Collections state to its constructor via the `state` parameter. Alternatively an instances state can be set via its `state` attribute. If multiple Collections with the same name are passed to a script, they will also be assigned to different states according to the passed order.

Scripts

In order to collect different collections which will be added to a single script, the Script class exists.

Collections can be added to script by either passing them to the scripts constructor or by adding them via a script instances' `add` method. For ease of use, meshes can be added in the same way, where they will be implicitly added to a new collection which will then be added to the script.

When a mesh or a collection is added to a script, it can be assigned a (new) name via the `name` argument of the Script constructor or the add method. Meshes or collections can be added separately or in a list. The name attribute should be a corresponding list. If the name attribute does not fit the passed meshes, PyMOLViz will try to interpret the passed names as gracefully as possible.

Alternatively a dictionary mapping names to meshes or collections can be used.

A script can be written to a file using the scripts `write` method.

Both collections and meshes provide the `to_script` functions which convert them directly to a script only containing the corresponding mesh or collection instance.