



© 2026 ANSYS, Inc. or affiliated companies  
Unauthorized use, distribution, or duplication prohibited.

## Visualization Interface Tool

---



ANSYS, Inc.  
Southpointe  
2600 Ansys Drive  
Canonsburg, PA 15317  
[ansysinfo@ansys.com](mailto:ansysinfo@ansys.com)  
<http://www.ansys.com>  
(T) 724-746-3304  
(F) 724-514-9494

Feb 27, 2026

ANSYS, Inc. and  
ANSYS Europe,  
Ltd. are UL  
registered ISO  
9001:2015  
companies.

# CONTENTS

<b>1 Getting started</b>	<b>3</b>
1.1 Installation . . . . .	3
1.1.1 Quick start . . . . .	3
<b>2 User guide</b>	<b>5</b>
2.1 Default plotter usage . . . . .	5
2.1.1 Use with PyVista meshes . . . . .	5
2.1.2 Use with PyAnsys custom objects . . . . .	5
2.2 Customize your own plotter . . . . .	6
2.3 Customizing the picker and hover callbacks . . . . .	7
<b>3 Migration</b>	<b>9</b>
3.1 Code migration . . . . .	9
3.1.1 Replace PyVista plotter code with Ansys Tools Visualization Interface plotter code . . . . .	9
3.1.2 Convert your custom meshes to objects usable by the Ansys Tools Visualization Interface plotter	10
3.1.3 Customize the PyVista backend . . . . .	10
3.1.4 Customize the picker or hover behavior . . . . .	12
3.1.5 Use the PyVista Qt backend . . . . .	12
3.2 Documentation configuration migration . . . . .	12
<b>4 API reference</b>	<b>15</b>
4.1 The <code>ansys.tools.visualization_interface</code> library . . . . .	15
4.1.1 Summary . . . . .	15
4.1.2 Description . . . . .	105
4.1.3 Module detail . . . . .	105
<b>5 Examples</b>	<b>107</b>
<b>6 Basic usage examples</b>	<b>109</b>
<b>7 Basic Plotly usage examples</b>	<b>111</b>
<b>8 Advanced usage examples</b>	<b>113</b>
8.1 Basic usage examples . . . . .	113
8.1.1 Use trame as a remote service . . . . .	113
8.1.2 Use a PyVista Qt backend . . . . .	114
8.1.3 Use a clipping plane . . . . .	115
8.1.4 Use the <code>MeshObjectPlot</code> class . . . . .	117
8.1.5 Use the plotter . . . . .	118
8.1.6 Animation Example . . . . .	121
8.1.7 MeshObjectPlot tree structure . . . . .	123

8.1.8	Activate the picker . . . . .	125
8.1.9	Customization API example . . . . .	129
8.1.10	Tree view menu . . . . .	132
8.1.11	Create custom picker . . . . .	137
8.2	Basic Plotly usage examples . . . . .	141
8.2.1	Plain usage of the plotly dash backend . . . . .	141
8.2.2	Plain usage of the plotly backend . . . . .	142
8.2.3	Backend-Agnostic Customization APIs (Plotly) . . . . .	144
8.3	Advanced usage examples . . . . .	146
8.3.1	Postprocessing simulation results using the <code>MeshObjectPlot</code> class . . . . .	146
<b>9</b>	<b>Contribute</b>	<b>151</b>
9.1	Install in developer mode . . . . .	151
9.2	Run tests . . . . .	151
9.3	Adhere to code style . . . . .	151
9.4	Build the documentation . . . . .	152
9.5	Post issues . . . . .	152
<b>Python Module Index</b>		<b>153</b>
<b>Index</b>		<b>155</b>

The Visualization Interface Tool is a Python API that provides an interface between PyAnsys libraries and different plotting backends.

The Visualization Interface Tool offers these main features:

- Serves as an interface between PyAnsys and other plotting libraries (although only [PyVista](#) is supported currently).
- Provides out-of-the box picking, viewing, and measuring functionalities.
- Supplies an extensible class for adding custom functionalities.

Getting started Learn how to install the Visualization Interface Tool in user mode and quickly begin using it.

[\*Getting started\*](#) User guide Understand key concepts for implementing the Visualization Interface Tool in your workflow.

[\*User guide\*](#) API reference Understand how to use Python to interact programmatically with the Visualization Interface Tool.

[\*API reference\*](#) Examples Explore examples that show how to use the Visualization Interface Tool to perform many different types of operations.

[\*Examples\*](#) Contribute Learn how to contribute to the Visualization Interface Tool codebase or documentation.

[\*Contribute\*](#)



## GETTING STARTED

This section describes how to install the Visualization Interface Tool in user mode and quickly begin using it. If you are interested in contributing to the Visualization Interface Tool, see [Contribute](#) for information on installing in developer mode.

### 1.1 Installation

To use `pip` to install the Visualization Interface Tool, run this command:

```
pip install ansys-tools-visualization-interface
```

Alternatively, to install the latest version from this library's [GitHub](#) repository, run these commands:

```
git clone https://github.com/ansys/ansys-tools-visualization-interface
cd ansys-tools-visualization-interface
pip install .
```

#### 1.1.1 Quick start

The following examples show how to use the Visualization Interface Tool to visualize a mesh file.

This code uses only a PyVista mesh:

```
from ansys.tools.visualization_interface import Plotter

my_mesh = my_custom_object.get_mesh()

# Create a Visualization Interface Tool object
pl = Plotter()
pl.plot(my_mesh)

# Plot the result
pl.show()
```

This code uses objects from a PyAnsys library:

```
from ansys.tools.visualization_interface import Plotter, MeshObjectPlot

my_custom_object = MyObject()
my_mesh = my_custom_object.get_mesh()

mesh_object = MeshObjectPlot(my_custom_object, my_mesh)
```

(continues on next page)

(continued from previous page)

```
# Create a Visualization Interface Tool object
pl = Plotter()
pl.plot(mesh_object)

# Plot the result
pl.show()
```

## USER GUIDE

This section explains key concepts for implementing the Visualization Interface Tool in your workflow. You can use the Visualization Interface Tool in your examples as well as integrate this library into your own code. For information on how to migrate from PyVista to the Ansys Visualization Interface Tool, see [Migration](#).

### 2.1 Default plotter usage

The Visualization Interface Tool provides a default plotter that can be used out of the box, using the PyVista backend. This default plotter provides common functionalities so that you do not need to create a custom plotter.

#### 2.1.1 Use with PyVista meshes

You can use the default plotter to plot simple PyVista meshes. This code shows how to use it to visualize a simple PyVista mesh:

```
## Usage example with pyvista meshes ##

import pyvista as pv
from ansys.tools.visualization_interface import Plotter

# Create a pyvista mesh
mesh = pv.Cube()

# Create a plotter
pl = Plotter()

# Add the mesh to the plotter
pl.plot(mesh)

# Show the plotter
pl.show()
```

#### 2.1.2 Use with PyAnsys custom objects

You can also use the default plotter to visualize PyAnsys custom objects. The only requirement is that the custom object must have a method that returns a PyVista mesh a method that exposes a name or id attribute of your object. To expose a custom object, you use a `MeshObjectPlot` instance. This class relates PyVista meshes with any object.

The following code shows how to use the default plotter to visualize a PyAnsys custom object:

```
## Usage example with PyAnsys custom objects ##

from ansys.tools.visualization_interface import Plotter
from ansys.tools.visualization_interface import MeshObjectPlot


# Create a custom object for this example
class CustomObject:
    def __init__(self):
        self.name = "CustomObject"
        self.mesh = pv.Cube()

    def get_mesh(self):
        return self.mesh

    def name(self):
        return self.name


custom_object = CustomObject()

# Create a MeshObjectPlot instance
mesh_object = MeshObjectPlot(custom_object, custom_object.get_mesh())

# Create a plotter
pl = Plotter()

# Add the MeshObjectPlot instance to the plotter
pl.plot(mesh_object)

# Show the plotter
pl.show()
```

## 2.2 Customize your own plotter

The Visualization Interface Tool provides a base class, `PlotterInterface`, for customizing certain functions of the plotter. This class provides a set of methods that can be overridden so that you can adapt the plotter to the specific need of your PyAnsys library.

The first thing you must do is to create a class that inherits from the `PlotterInterface` class. After that, see these main use cases for customizing the plotter:

- The most common use case is to customize the way that the objects you represent are shown in the plotter. To this end, you can override the `plot` and `plot_iter` methods. These methods are called every time a new object is added to the plotter. The default implementation of this method is to add a PyVista mesh or a `MeshObjectPlot` instance to the plotter. You can override this method to add your own meshes or objects to the plotter in a manner that fits the way that you want to represent the meshes.
- Another use case is the need to have custom button functionalities for your library. For example, you may want buttons for hiding or showing certain objects. To add custom buttons to the plotter, you use the implementable interface provided by the `PlotterWidget` class.

Some practical examples of how to use the `PlotterInterface` class are included in some PyAnsys libraries, such as `PyAnsys Geometry`.

For comprehensive migration information with code examples, see [Migration](#).

## 2.3 Customizing the picker and hover callbacks

The Visualization Interface Tool provides a base class, `AbstractPicker`, for customizing the picker and hover callbacks of the plotter. This class provides a set of methods that can be overridden so that you can adapt the picker and hover functionalities to the specific need of your PyAnsys library.

The first thing you must do is to create a class that inherits from the `AbstractPicker` class. After that, see these main use cases for customizing the picker and hover callbacks:

- You may want to change the way that objects are picked in the plotter. To do this, you can override the `pick_select_object` and `pick_unselect_object` methods. These methods are called when an object is selected or unselected, respectively.
- Similarly, you may want to change the way that objects are hovered over in the plotter. To do this, you can override the `hover_select_object` and `hover_unselect_object` methods. These methods are called when an object is hovered over or unhovered, respectively.

A practical example of how to use the `AbstractPicker` class is included in [Create custom picker](#).



## MIGRATION

This section helps you migrate from PyVista plotters to the Ansys Tools Visualization Interface plotters. It consists of two major topics:

- *Code migration*
- *Documentation configuration migration*

### 3.1 Code migration

This topic explains how to migrate from PyVista plotters to the new Ansys Tools Visualization Interface plotters. Because cases vary greatly, it provides a few examples that cover the most common scenarios.

#### 3.1.1 Replace PyVista plotter code with Ansys Tools Visualization Interface plotter code

If you only need to plot simple PyVista meshes, you can directly replace your PyVista plotter code with the Ansys Tools Visualization Interface plotter code. On top of common PyVista functionalities, the Ansys Tools Visualization Interface plotter provides additional interactivity such as view buttons and mesh slicing.

The following code shows how to do the plotter code replacement:

- PyVista code:

```
import pyvista as pv

# Create a PyVista mesh
mesh = pv.Cube()

# Create a plotter
pl = pv.Plotter()

# Add the mesh to the plotter
pl.add_mesh(mesh)

# Show the plotter
pl.show()
```

- Ansys Tools Visualization Interface code:

```
import pyvista as pv
from ansys.tools.visualization_interface import Plotter
```

(continues on next page)

(continued from previous page)

```
# Create a PyVista mesh
mesh = pv.Cube()

# Create a plotter
pl = Plotter()

# Add the mesh to the plotter
pl.plot(mesh)

# Show the plotter
pl.show()
```

### 3.1.2 Convert your custom meshes to objects usable by the Ansys Tools Visualization Interface plotter

Your custom object must have a method that returns a PyVista mesh and a method that exposes a `name` or `id` attribute of your object:

```
class CustomObject:
    def __init__(self):
        self.name = "CustomObject"
        self.mesh = pv.Cube(center=(1, 1, 0))

    def get_mesh(self):
        return self.mesh

    def name(self):
        return self.name
```

You then need to create a `MeshObjectPlot` instance that relates the PyVista mesh with your custom object:

```
from ansys.tools.visualization_interface import MeshObjectPlot

custom_object = CustomObject()
mesh_object_plot = MeshObjectPlot(
    custom_object=custom_object,
    mesh=custom_object.get_mesh(),
)
```

With this, you can use the Ansys Tools Visualization Interface plotter to visualize your custom object. It enables interactivity such as picking and hovering.

### 3.1.3 Customize the PyVista backend

You can customize the backend of the Ansys Tools Visualization Interface plotter to enable or turn off certain functionalities. The following code shows how to enable picking:

```
from ansys.tools.visualization_interface import Plotter
from ansys.tools.visualization_interface.backends import PyVistaBackend

backend = PyVistaBackend(allow_picking=True)
```

(continues on next page)

(continued from previous page)

```
# Create a plotter
pl = Plotter(backend=backend)

# Add the MeshObjectPlot instance to the plotter
pl.plot(mesh_object_plot)

# Show the plotter
pl.show()
```

If you want to customize the backend even more, you can create your own backend by inheriting from the PyVistaBackendInterface class and implementing the required methods:

```
@abstractmethod
def plot_iter(self, plottable_object: Any, name_filter: str = None, **plotting_options):
    """Plot one or more compatible objects to the plotter.

    Parameters
    -----
    plottable_object : Any
        One or more objects plot.
    name_filter : str, default: None.
        Regular expression with the desired name or names to include in the plotter.
    **plotting_options : dict, default: None
        Keyword arguments. For allowable keyword arguments, see the
        :meth:`Plotter.add_mesh <pyvista.Plotter.add_mesh>` method.

    """
    pass
```

```
@abstractmethod
def plot(self, plottable_object: Any, name_filter: str = None, **plotting_options):
    """Plot a single object to the plotter.

    Parameters
    -----
    plottable_object : Any
        Object to plot.
    name_filter : str
        Regular expression with the desired name or names to include in the plotter.
    **plotting_options : dict, default: None
        Keyword arguments. For allowable keyword arguments, see the
        :meth:`Plotter.add_mesh <pyvista.Plotter.add_mesh>` method.

    """
    pass
```

The rest of the methods are implemented for you. This ensures that while you can customize what you need for plotting, the rest of the functionalities still work as expected. For more information, see the backend documentation. If you need to even go further, you can create your own plotter by inheriting from the BaseBackend class and implementing the required methods, although this may break existing features.

### 3.1.4 Customize the picker or hover behavior

You can customize the picker of the Ansys Tools Visualization Interface plotter to decide what happens when you pick or hover over an object. For example, if you want to print the name of the picked object, you can do it as described in the [Create custom picker](#) example.

### 3.1.5 Use the PyVista Qt backend

You can use the PyVista Qt backend with the Ansys Tools Visualization Interface plotter. To do this, you must set the PyVista backend to Qt before creating the plotter:

```
cube = pv.Cube()
pv_backend = PyVistaBackend(use_qt=True, show_qt=True)
pl = Plotter(backend=pv_backend)
pl.plot(cube)
pl.backend.enable_widgets()
pv_backend.scene.show()
```

You can then integrate the plotter into a PyQt or PySide app by disabling the `show_qt` parameter. For more information about this, see the [PyVista documentation](#).

## 3.2 Documentation configuration migration

This topic explains how to migrate from the PyVista documentation configuration to the new Ansys Tools Visualization Interface documentation configuration.

1. Add environment variables for documentation:

```
os.environ["PYANSYS_VISUALIZER_DOC_MODE"] = "true"
os.environ["PYANSYS_VISUALIZER_HTML_BACKEND"] = "true"
```

2. Use PyVista DynamicScraper:

```
from pyvista.plotting.utilities.sphinx_gallery import DynamicScraper

sphinx_gallery_conf = {
    "image_scrapers": (DynamicScraper(),),
}
```

3. Add PyVista viewer directive to extensions:

```
extensions = ["pyvista.ext.viewer_directive"]
```

4. Make sure you are executing the notebook cells:

```
nbsphinx_execute = "always"
```

For Plotly, in `conf.py`, do the following:

1. Add environment variables for documentation:

```
os.environ["PYANSYS_VISUALIZER_DOC_MODE"] = "true"
```

2. Add plotly configuration

```
import plotly.io as pio
pio.renderers.default = "sphinx_gallery"
```

3. Import and add scraper

```
from plotly.io._sg_scraper import plotly_sg_scraper
sphinx_gallery_conf = {
    "image_scrapers": (DynamicScraper(), "matplotlib", plotly_sg_scraper),
}
```

4. [IMPORTANT] The `pl.show()` must be the last line of code in the cell, or else it won't show.



## API REFERENCE

This section describes ansys-tools-visualization-interface endpoints, their capabilities, and how to interact with them programmatically.

### 4.1 The `ansys.tools.visualization_interface` library

#### 4.1.1 Summary

##### Subpackages

<code>backends</code>	Provides interfaces.
<code>types</code>	Provides custom types.
<code>utils</code>	Provides the Utils package.

##### Submodules

<code>plotter</code>	Module for the Plotter class.
----------------------	-------------------------------

##### Attributes

<code>__version__</code>
--------------------------

##### Constants

<code>USE_TKFRAME</code>	
<code>DOCUMENTATION_BUILD</code>	Whether the documentation is being built or not.
<code>TESTING_MODE</code>	Whether the library is being built or not, used to avoid showing plots while testing.
<code>USE_HTML_BACKEND</code>	Whether the library is being built or not, used to avoid showing plots while testing.

#### The `backends` package

##### Summary

##### Subpackages

<code>plotly</code>	Plotly initialization.
<code>pyvista</code>	Provides interfaces.

## The `plotly` package

### Summary

### Subpackages

<code>widgets</code>	Widgets module init.
----------------------	----------------------

### Submodules

<code>plotly_dash</code>	Module for dash plotly.
<code>plotly_interface</code>	Plotly backend interface for visualization.

## The `widgets` package

### Summary

### Submodules

<code>button_manager</code>	Module for button management.
<code>dropdown_manager</code>	Module for dropdown management in Plotly figures.

## The `button_manager.py` module

### Summary

### Classes

<code>ButtonManager</code>	Class to manage buttons in a Plotly figure.
----------------------------	---

#### `ButtonManager`

```
class ansys.tools.visualization_interface.backends.plotly.widgets.button_manager.ButtonManager(fig:  
plotly.grap
```

Class to manage buttons in a Plotly figure.

This class allows adding buttons to a Plotly figure for various functionalities such as toggling visibility of traces, resetting the view, and custom actions.

#### Parameters

`fig`

[`go.Figure`] The Plotly figure to which buttons will be added.

### Overview

## Methods

<code>add_button</code>	Add a button to the Plotly figure.
<code>show_hide_bbox_dict</code>	Generate dictionary for showing/hiding coordinate system elements.
<code>update_layout</code>	Update the figure layout with all controls as buttons in a single row.
<code>args_xy_view_button</code>	Get camera configuration for XY plane view (top-down view).
<code>args_xz_view_button</code>	Get camera configuration for XZ plane view (front view).
<code>args_yz_view_button</code>	Get camera configuration for YZ plane view (side view).
<code>args_iso_view_button</code>	Get camera configuration for isometric view (3D perspective).
<code>add_measurement_toggle_button</code>	Get configuration for measurement toggle button.
<code>args_projection_toggle_button</code>	Get configuration for projection toggle button.
<code>args_theme_toggle_button</code>	Get configuration for theme toggle button.

## Import detail

```
from ansys.tools.visualization_interface.backends.plotly.widgets.button_manager import_
    ButtonManager
```

## Method detail

`ButtonManager.add_button(label: str, x: float, y: float, xanchor: str = 'left', yanchor: str = 'bottom', method: str = 'restyle', args: List[Any] = None, args2: List[Any] = None) → None`

Add a button to the Plotly figure.

### Parameters

#### `label`

[`str`] The text to display on the button.

#### `x`

[`float`] X position of the button (0-1).

#### `y`

[`float`] Y position of the button (0-1).

#### `xanchor`

[`str`, optional] X anchor point for the button, by default “left”.

#### `yanchor`

[`str`, optional] Y anchor point for the button, by default “bottom”.

#### `method`

[`str`, optional] The method to call when the button is clicked. Options include: ‘restyle’, ‘relayout’, ‘update’, ‘animate’, by default ‘restyle’.

#### `args`

[`List[Any]`, optional] Arguments to pass to the method when the button is clicked, by default None.

#### `args2`

[`List[Any]`, optional] Secondary arguments for toggle functionality, by default None.

`ButtonManager.show_hide_bbox_dict(toggle: bool = True)`

Generate dictionary for showing/hiding coordinate system elements.

### Parameters

**toggle**

[bool, optional] Whether to show (True) or hide (False) the coordinate system, by default True.

**Returns****dict**

Dictionary with coordinate system visibility settings.

`ButtonManager.update_layout()` → `None`

Update the figure layout with all controls as buttons in a single row.

This method builds buttons using the configuration methods and any additional buttons that were added via `add_button()`.

`ButtonManager.args_xy_view_button(label: str = 'XY View', x: float = 0.02, y: float = 1.02)` → `dict`

Get camera configuration for XY plane view (top-down view).

**Parameters****label**

[str, optional] The text to display on the button, by default “XY View”.

**x**

[float, optional] X position of the button (0-1), by default 0.02.

**y**

[float, optional] Y position of the button (0-1), by default 1.02.

**Returns****dict**

Camera configuration for XY plane view.

`ButtonManager.args_xz_view_button(label: str = 'XZ View', x: float = 0.02, y: float = 1.02)` → `dict`

Get camera configuration for XZ plane view (front view).

**Parameters****label**

[str, optional] The text to display on the button, by default “XZ View”.

**x**

[float, optional] X position of the button (0-1), by default 0.02.

**y**

[float, optional] Y position of the button (0-1), by default 1.02.

**Returns****dict**

Camera configuration for XZ plane view.

`ButtonManager.args_yz_view_button(label: str = 'YZ View', x: float = 0.02, y: float = 1.02)` → `dict`

Get camera configuration for YZ plane view (side view).

**Parameters****label**

[str, optional] The text to display on the button, by default “YZ View”.

**x**

[float, optional] X position of the button (0-1), by default 0.02.

**y**  
[`float`, optional] Y position of the button (0-1), by default 1.02.

#### Returns

`dict`

Camera configuration for YZ plane view.

`ButtonManager.args_iso_view_button(label: str = 'ISO View', x: float = 0.02, y: float = 1.02) → dict`

Get camera configuration for isometric view (3D perspective).

#### Parameters

**label**

[`str`, optional] The text to display on the button, by default “ISO View”.

**x**

[`float`, optional] X position of the button (0-1), by default 0.02.

**y**

[`float`, optional] Y position of the button (0-1), by default 1.02.

#### Returns

`dict`

Camera configuration for isometric view.

`ButtonManager.add_measurement_toggle_button(label: str = 'Toggle Measurement', x: float = 0.02, y: float = 0.87) → Tuple[dict, dict]`

Get configuration for measurement toggle button.

#### Parameters

**label**

[`str`, optional] The text to display on the button, by default “Toggle Measurement”.

**x**

[`float`, optional] X position of the button (0-1), by default 0.02.

**y**

[`float`, optional] Y position of the button (0-1), by default 0.87.

#### Returns

`Tuple[dict, dict]`

Tuple containing (enable\_measurement\_config, disable\_measurement\_config).

`ButtonManager.args_projection_toggle_button() → Tuple[dict, dict]`

Get configuration for projection toggle button.

#### Parameters

**label**

[`str`, optional] The text to display on the button, by default “Toggle Projection”.

**x**

[`float`, optional] X position of the button (0-1), by default 0.14.

**y**

[`float`, optional] Y position of the button (0-1), by default 1.02.

#### Returns

`Tuple[dict, dict]`

Tuple containing (perspective\_projection\_config, orthographic\_projection\_config).

```
ButtonManager.args_theme_toggle_button(label: str = 'Toggle Theme', x: float = 0.32, y: float = 1.02) →  
    Tuple[dict, dict]
```

Get configuration for theme toggle button.

#### Parameters

##### label

[str, optional] The text to display on the button, by default “Toggle Theme”.

##### x

[float, optional] X position of the button (0-1), by default 0.22.

##### y

[float, optional] Y position of the button (0-1), by default 1.02.

#### Returns

##### Tuple[dict, dict]

Tuple containing (light\_theme\_config, dark\_theme\_config).

## Description

Module for button management.

## The dropdown\_manager.py module

### Summary

### Classes

<code>DashDropdownManager</code>	Class to manage dropdown menus in a Plotly figure.
----------------------------------	--

#### DashDropdownManager

```
class ansys.tools.visualization_interface.backends.plotly.widgets.dropdown_manager.DashDropdownManager(
```

Class to manage dropdown menus in a Plotly figure.

This class allows adding dropdown menus to a Plotly figure for controlling mesh visibility and other properties.

#### Parameters

##### fig

[go.Figure] The Plotly figure to which dropdowns will be added.

### Overview

### Methods

<code>add_mesh_name</code>	Add a mesh name to track for dropdown functionality.
<code>get_mesh_names</code>	Get the list of tracked mesh names.
<code>get_visibility_args_for_meshes</code>	Get visibility arguments for showing only specified meshes.
<code>clear</code>	Clear all tracked mesh names.

## Import detail

```
from ansys.tools.visualization_interface.backends.plotly.widgets.dropdown_manager import_
DashDropdownManager
```

## Method detail

`DashDropdownManager.add_mesh_name(name: str) → None`

Add a mesh name to track for dropdown functionality.

### Parameters

#### `name`

[`str`] The name of the mesh to track.

`DashDropdownManager.get_mesh_names() → List[str]`

Get the list of tracked mesh names.

### Returns

#### `List[str]`

List of mesh names.

`DashDropdownManager.get_visibility_args_for_meshes(visible_mesh_names: List[str]) → Dict[str, Any]`

Get visibility arguments for showing only specified meshes.

### Parameters

#### `visible_mesh_names`

[`List[str]`] List of mesh names that should be visible.

### Returns

#### `Dict[str, Any]`

Arguments for restyle method to set mesh visibility.

`DashDropdownManager.clear() → None`

Clear all tracked mesh names.

## Description

Module for dropdown management in Plotly figures.

## Description

Widgets module init.

## The `plotly_dash.py` module

### Summary

### Classes

<code>PlotlyDashBackend</code>	Plotly Dash interface for visualization.
--------------------------------	--

## PlotlyDashBackend

```
class ansys.tools.visualization_interface.backends.plotly.plotly_dash.PlotlyDashBackend(app:  
                                         dash.Dash  
                                         =  
                                         None)  
  
Bases: ansys.tools.visualization_interface.backends.plotly.plotly_interface.  
        PlotlyBackend  
  
Plotly Dash interface for visualization.
```

## Overview

### Methods

<code>plot</code>	Plot a single object using Plotly and track mesh names for dropdown.
<code>create_dash_layout</code>	Create the Dash layout with optional dropdown for mesh visibility.
<code>show</code>	Render the Plotly scene.

### Properties

<code>dropdown_manager</code>	Get the dropdown manager for this backend.
-------------------------------	--

### Import detail

```
from ansys.tools.visualization_interface.backends.plotly.plotly_dash import_  
    PlotlyDashBackend
```

### Property detail

`property PlotlyDashBackend.dropdown_manager: ansys.tools.visualization_interface.  
backends.plotly.widgets.dropdown_manager.DashDropdownManager`

Get the dropdown manager for this backend.

#### Returns

`DashDropdownManager`

The dropdown manager instance.

### Method detail

`PlotlyDashBackend.plot(plottable_object, name: str = None, **plotting_options) → None`

Plot a single object using Plotly and track mesh names for dropdown.

#### Parameters

`plottable_object`

[Any] The object to plot.

`name`

[`str`, optional] Name of the mesh for labeling in Plotly.

`plotting_options`

[`dict`] Additional plotting options.

`PlotlyDashBackend.create_dash_layout()` → dash.html.Div

Create the Dash layout with optional dropdown for mesh visibility.

#### Returns

`html.Div`

The Dash layout component.

`PlotlyDashBackend.show(plotable_object=None, screenshot: str = None, name_filter=None, **kwargs) → plotly.graph_objects.Figure | None`

Render the Plotly scene.

#### Parameters

`plotable_object`

[Any, optional] Object to show, by default None.

`screenshot`

[str, optional] Path to save a screenshot, by default None.

`name_filter`

[bool, optional] Flag to filter the object, by default None.

`kwargs`

[dict] Additional options the selected backend accepts.

#### Returns

`Union[go.Figure, None]`

The figure of the plot if in doc building environment. Else, None.

## Description

Module for dash plotly.

## The `plotly_interface.py` module

### Summary

### Classes

<code>PlotlyBackend</code>	Plotly interface for visualization.
----------------------------	-------------------------------------

### PlotlyBackend

```
class ansys.tools.visualization_interface.backends.plotly.plotly_interface.PlotlyBackend
```

Bases: `ansys.tools.visualization_interface.backends._base.BaseBackend`

Plotly interface for visualization.

### Overview

## Methods

<code>plot_iter</code>	Plot multiple objects using Plotly.
<code>plot</code>	Plot a single object using Plotly.
<code>show</code>	Render the Plotly scene.
<code>add_points</code>	Add point markers to the scene.
<code>add_lines</code>	Add line segments to the scene.
<code>add_planes</code>	Add a plane to the scene.
<code>add_text</code>	Add text to the scene.
<code>add_labels</code>	Add labels at 3D point locations.
<code>clear</code>	Clear all traces from the figure.

## Properties

<code>layout</code>	Get the current layout of the Plotly figure.
---------------------	--

## Import detail

```
from ansys.tools.visualization_interface.backends.plotly.plotly_interface import_
    PlotlyBackend
```

## Property detail

**property** `PlotlyBackend.layout: Any`

Get the current layout of the Plotly figure.

### Returns

**Any**

The current layout of the Plotly figure.

## Method detail

`PlotlyBackend.plot_iter(plotting_list: Iterable[Any]) → None`

Plot multiple objects using Plotly.

### Parameters

**plotting\_list**

[Iterable[Any]] An iterable of objects to plot.

`PlotlyBackend.plot(plottable_object: pyvista.PolyData | pyvista.MultiBlock | ansys.tools.visualization_interface.types.mesh_object_plot.MeshObjectPlot | plotly.graph_objects.Mesh3d, name: str = None, **plotting_options) → None`

Plot a single object using Plotly.

### Parameters

**plottable\_object**

[Union[PolyData, pv.MultiBlock, MeshObjectPlot, go.Mesh3d]] The object to plot.  
Can be a PyVista PolyData, MultiBlock, a MeshObjectPlot, or a Plotly Mesh3d.

**plotting\_options**

[dict] Additional plotting options.

**name**

[`str`, optional] Name of the mesh for labeling in Plotly. Overrides the name from MeshObjectPlot if provided.

`PlotlyBackend.show(plotable_object=None, screenshot: str = None, name_filter=None, **kwargs) → plotly.graph_objects.Figure | None`

Render the Plotly scene.

**Parameters****plotable\_object**

[Any, optional] Object to show, by default None.

**screenshot**

[`str`, optional] Path to save a screenshot, by default None.

**name\_filter**

[`bool`, optional] Flag to filter the object, by default None.

**kwargs**

[`dict`] Additional options the selected backend accepts.

**Returns****Union[go.Figure, None]**

The figure of the plot if in doc building environment. Else, None.

`PlotlyBackend.add_points(points: List | Any, color: str = 'red', size: float = 10.0, **kwargs) → Any`

Add point markers to the scene.

**Parameters****points**

[`Union[List, Any]`] Points to add. Expected format: [[x1, y1, z1], [x2, y2, z2], ...] or Nx3 array.

**color**

[`str`, default: “red”] Color of the points.

**size**

[`float`, default: 10.0] Size of the point markers.

**\*\*kwargs**

[`dict`] Additional keyword arguments passed to Plotly’s Scatter3d.

**Returns****go.Scatter3d**

Plotly Scatter3d trace representing the added points.

`PlotlyBackend.add_lines(points: List | Any, connections: List | Any | None = None, color: str = 'white', width: float = 1.0, **kwargs) → Any`

Add line segments to the scene.

**Parameters****points**

[`Union[List, Any]`] Points defining the lines. Expected format: [[x1, y1, z1], [x2, y2, z2], ...] or Nx3 array.

**connections**

[`Optional[Union[List, Any]]`, default: `None`] Line connectivity. If None, connects points

sequentially. Expected format: [[start\_idx1, end\_idx1], [start\_idx2, end\_idx2], ...] or Mx2 array.

**color**

[`str`, default: “white”] Color of the lines.

**width**

[`float`, default: 1.0] Width of the lines.

**\*\*kwargs**

[`dict`] Additional keyword arguments passed to Plotly’s Scatter3d.

**Returns**

`Union[go.Scatter3d, List[go.Scatter3d]]`

Plotly Scatter3d trace(s) representing the added lines.

PlotlyBackend.`add_planes`(center: `Tuple[float, float, float]` = (0.0, 0.0, 0.0), normal: `Tuple[float, float, float]` = (0.0, 0.0, 1.0), i\_size: `float` = 1.0, j\_size: `float` = 1.0, \*\*kwargs) → Any

Add a plane to the scene.

**Parameters****center**

[`Tuple[float, float, float]`, default: (0.0, 0.0, 0.0)] Center point of the plane (x, y, z).

**normal**

[`Tuple[float, float, float]`, default: (0.0, 0.0, 1.0)] Normal vector of the plane (x, y, z).

**i\_size**

[`float`, default: 1.0] Size of the plane in the i direction.

**j\_size**

[`float`, default: 1.0] Size of the plane in the j direction.

**\*\*kwargs**

[`dict`] Additional keyword arguments passed to Plotly’s Mesh3d (e.g., color, opacity).

**Returns**

`go.Mesh3d`

Plotly Mesh3d trace representing the added plane.

PlotlyBackend.`add_text`(text: `str`, position: `Tuple[float, float] | Tuple[float, float, float] | str`, font\_size: `int` = 12, color: `str` = ‘white’, \*\*kwargs) → Any

Add text to the scene.

**Parameters****text**

[`str`] Text string to display.

**position**

[`Union[Tuple[float, float], Tuple[float, float, float], str]`] Position for the text as 2D screen coordinates (x, y). Values should be between 0 and 1 for normalized coordinates, or pixel values for absolute positioning.

**font\_size**

[`int`, default: 12] Font size for the text.

**color**

[`str`, default: “white”] Color of the text.

**\*\*kwargs**

[`dict`] Additional keyword arguments passed to Plotly's annotation.

**Returns****dict**

Plotly annotation representing the added text.

`PlotlyBackend.add_labels(points: List | Any, labels: List[str], font_size: int = 12, point_size: float = 5.0, **kwargs) → Any`

Add labels at 3D point locations.

**Parameters****points**

[Union[List, Any]] Points where labels should be placed.

**labels**

[List[str]] List of label strings to display at each point.

**font\_size**

[`int`, default: 12] Font size for the labels.

**point\_size**

[`float`, default: 5.0] Size of the point markers shown with labels.

**\*\*kwargs**

[`dict`] Additional keyword arguments.

**Returns****Any**

Plotly trace representing the labels.

`PlotlyBackend.clear() → None`

Clear all traces from the figure.

## Description

Plotly backend interface for visualization.

## Description

Plotly initialization.

## The pyvista package

### Summary

### Subpackages

---

<code>widgets</code>	Provides widgets for the Visualization Interface Tool plotter.
----------------------	--

---

## Submodules

<code>animation</code>	Animation support for PyVista backend.
<code>picker</code>	Module for managing picking and hovering of objects in a PyVista plotter.
<code>pyvista</code>	Provides a wrapper to aid in plotting.
<code>pyvista_interface</code>	Provides plotting for various PyAnsys objects.
<code>trame_local</code>	Provides <code>trame</code> visualizer interface for visualization.
<code>trame_remote</code>	Module for trame websocket client functions.
<code>trame_service</code>	Trame service module.

## The widgets package

### Summary

### Submodules

<code>button</code>	Provides for implementing buttons in PyAnsys.
<code>dark_mode</code>	Provides the dark mode button widget for the PyAnsys plotter.
<code>displace_arrows</code>	Provides the displacement arrows widget for the PyVista plotter.
<code>dynamic_tree_menu</code>	Provides a dynamic tree menu widget with clickable buttons.
<code>hide_buttons</code>	Provides the hide buttons widget for the PyAnsys plotter.
<code>measure</code>	Provides the measure widget for the PyAnsys plotter.
<code>mesh_slider</code>	Provides the measure widget for the PyAnsys plotter.
<code>next_button</code>	Provides the next button widget for animation control.
<code>parallel_projection</code>	Provides the parallel projection button widget.
<code>pick_rotation_center</code>	Provides the measure widget for the PyAnsys plotter.
<code>play_pause_button</code>	Provides the play/pause button widget for animation control.
<code>previous_button</code>	Provides the previous button widget for animation control.
<code>ruler</code>	Provides the ruler widget for the Visualization Interface Tool plotter.
<code>save_gif_button</code>	Provides the save GIF button widget for animation control.
<code>screenshot</code>	Provides the screenshot widget for the Visualization Interface Tool plotter.
<code>stop_button</code>	Provides the stop button widget for animation control.
<code>tree_menu_toggle</code>	Provides a button to toggle the tree menu visibility.
<code>view_button</code>	Provides the view button widget for changing the camera view.
<code>widget</code>	Provides the abstract implementation of plotter widgets.

## The `button.py` module

### Summary

### Classes

<code>Button</code>	Provides the abstract class for implementing buttons in PyAnsys.
---------------------	--

### Button

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.button.Button(plotter:
    pyvista.Plotter,
    button_config:
    tuple,
    dark_mode:
    bool =
    False)

Bases: ansys.tools.visualization_interface.backends.pyvista.widgets.widget.
PlotterWidget
```

Provides the abstract class for implementing buttons in PyAnsys.

### Parameters

#### plotter

[Plotter] Plotter to draw the buttons on.

#### button\_config

[tuple] Tuple containing the position and the path to the icon of the button.

#### dark\_mode

[bool, optional] Whether to activate the dark mode or not.

### Notes

This class wraps the PyVista `add_checkbox_button_widget()` method.

## Overview

### Abstract methods

<code>callback</code>	Get the functionality of the button, which is implemented by subclasses.
-----------------------	--

### Methods

<code>update</code>	Assign the image that represents the button.
---------------------	--

### Attributes

<code>button_config</code>
----------------------------

### Import detail

<code>from ansys.tools.visualization_interface.backends.pyvista.widgets.button import Button</code>
---

### Attribute detail

`Button.button_config`

## Method detail

**abstractmethod** `Button.callback(state: bool) → None`

Get the functionality of the button, which is implemented by subclasses.

### Parameters

`state`

[bool] Whether the button is active.

`Button.update() → None`

Assign the image that represents the button.

## Description

Provides for implementing buttons in PyAnsys.

## The dark\_mode.py module

### Summary

### Classes

---

`DarkModeButton` Provides the dark mode widget for the Visualization Interface Tool Plotter class.

---

#### DarkModeButton

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.dark_mode.DarkModeButton(plotter:  
    an-  
    sys.tools.visual-  
    dark_mode:  
    bool  
    =  
    False)  
Bases: ansys.tools.visualization_interface.backends.pyvista.widgets.widget.  
PlotterWidget
```

Provides the dark mode widget for the Visualization Interface Tool Plotter class.

### Parameters

`plotter_helper`

[PlotterHelper] Plotter to add the dark mode widget to.

`dark_mode`

[bool, optional] Whether to activate the dark mode or not.

## Overview

## Methods

---

`callback` Remove or add the dark mode widget actor upon click.  
`update` Define the dark mode widget button parameters.

---

## Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.dark_mode import  
    DarkModeButton
```

## Method detail

**DarkModeButton.callback(state: *bool*) → None**

Remove or add the dark mode widget actor upon click.

## Parameters

state

[bool] Whether the state of the button, which is inherited from PyVista, is active.

`DarkModeButton.update() → None`

Define the dark mode widget button parameters.

## Description

Provides the dark mode button widget for the PyAnsys plotter.

## The `displace_arrows.py` module

## Summary

## Classes

*DisplacementArrow* Defines the arrow to draw and what it is to do.

## Enums

**CameraPanDirection** Provides an enum with the available movement directions of the camera.

## DisplacementArrow

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.displace_arrows.DisplacementArrow(plt  
    pyv  
    di-  
    rec  
    tion  
    Ca  
    er-  
    a-  
    Pa  
    rec  
    tion  
    dan  
    boo  
    =  
    Ea
```

Bases: `ansys.tools.visualization_interface.backends.pyvista.widgets.button.Button`

Defines the arrow to draw and what it is to do.

### Parameters

#### plotter

[Plotter] Plotter to draw the buttons on.

#### direction

[CameraPanDirection] Direction that the camera is to move.

#### dark\_mode

[bool, optional] Whether to activate the dark mode or not.

## Overview

### Methods

<code>callback</code>	Move the camera in the direction defined by the button.
-----------------------	---

### Attributes

<code>direction</code>
------------------------

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.displace_arrows import_
DisplacementArrow
```

### Attribute detail

`DisplacementArrow.direction`

### Method detail

`DisplacementArrow.callback(state: bool) → None`

Move the camera in the direction defined by the button.

### Parameters

#### state

[bool] Whether the state of the button, which is inherited from PyVista, is active. However, this parameter is unused by this `callback` method.

### CameraPanDirection

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.displace_arrows.
CameraPanDirection
```

Bases: `enum.Enum`

Provides an enum with the available movement directions of the camera.

## Overview

### Attributes

XUP
XDOWN
YUP
YDOWN
ZUP
ZDOWN

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.displace_arrows import_
    CameraPanDirection
```

### Attribute detail

```
CameraPanDirection.XUP = (0, 'upxarrow', (5, 230))
CameraPanDirection.XDOWN = (1, 'downarrow', (5, 190))
CameraPanDirection.YUP = (2, 'upyarrow', (35, 230))
CameraPanDirection.YDOWN = (3, 'downarrow', (35, 190))
CameraPanDirection.ZUP = (4, 'upzarrow', (65, 230))
CameraPanDirection.ZDOWN = (5, 'downarrow', (65, 190))
```

### Description

Provides the displacement arrows widget for the PyVista plotter.

### The dynamic\_tree\_menu.py module

#### Summary

#### Classes

<a href="#">DynamicTreeMenuWidget</a>	Provides a dynamic tree menu with clickable buttons for each object.
---------------------------------------	--

### DynamicTreeMenuWidget

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.dynamic_tree_menu.DynamicTreeMenuWidget
```

Bases: [ansys.tools.visualization\\_interface.backends.pyvista.widgets.widget.PlotterWidget](#)

Provides a dynamic tree menu with clickable buttons for each object.

This widget displays a hierarchical tree view of all plotted objects on the right side of the screen. Each object has a clickable checkbox button to toggle visibility.

#### Parameters

##### **plotter**

[[Plotter](#)] The plotter to add the tree menu widget to.

##### **position**

[[tuple](#), optional] The (x, y) position of the top-left corner of the menu. Default is (0.65, 0.95).

##### **button\_size**

[[int](#), optional] Size of the checkbox buttons in pixels. Default is 20.

##### **spacing**

[[int](#), optional] Vertical spacing between items in pixels. Default is 30.

##### **font\_size**

[[int](#), optional] Font size for the object labels. Default is 12.

##### **dark\_mode**

[[bool](#), optional] Whether to use dark mode colors. Default is False. In light mode (False): text is black In dark mode (True): text is white

## Notes

Each object gets a checkbox button that toggles visibility of the object and its subtree. The menu updates dynamically as objects are added or removed.

## Overview

### Methods

<code>refresh</code>	Refresh the menu display.
<code>update</code>	Update the widget's appearance for current dark mode setting.
<code>show_menu</code>	Show the menu by enabling all text actors and buttons.
<code>hide_menu</code>	Hide the menu by disabling all text actors and buttons.
<code>callback</code>	Placeholder callback (not used for this widget).

## Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.dynamic_tree_menu_
import DynamicTreeMenuWidget
```

### Method detail

`DynamicTreeMenuWidget.refresh()`

Refresh the menu display.

`DynamicTreeMenuWidget.update()`

Update the widget's appearance for current dark mode setting.

`DynamicTreeMenuWidget.show_menu()`

Show the menu by enabling all text actors and buttons.

`DynamicTreeMenuWidget.hide_menu()`

Hide the menu by disabling all text actors and buttons.

`DynamicTreeMenuWidget.callback(state: bool) → None`

Placeholder callback (not used for this widget).

## Description

Provides a dynamic tree menu widget with clickable buttons.

### The `hide_buttons.py` module

#### Summary

#### Classes

<code>HideButton</code>	Provides the hide widget for the Visualization Interface Tool Plotter class.
-------------------------	--

## HideButton

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.hide_buttons.HideButton(plotter:  
    an-  
    sys.tools.visualization  
    dark_mode:  
    bool  
    =  
    False)
```

Bases: *ansys.tools.visualization\_interface.backends.pyvista.widgets.widget.PlotterWidget*

Provides the hide widget for the Visualization Interface Tool Plotter class.

### Parameters

#### plotter\_helper

[PlotterHelper] Plotter to add the hide widget to.

#### dark\_mode

[bool, optional] Whether to activate the dark mode or not.

## Overview

## Methods

<code>callback</code>	Remove or add the hide widget actor upon click.
<code>update</code>	Define the hide widget button parameters.

## Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.hide_buttons import  
    HideButton
```

## Method detail

`HideButton.callback(state: bool) → None`

Remove or add the hide widget actor upon click.

### Parameters

#### state

[bool] Whether the state of the button, which is inherited from PyVista, is active.

`HideButton.update() → None`

Define the hide widget button parameters.

## Description

Provides the hide buttons widget for the PyAnsys plotter.

## The `measure.py` module

### Summary

### Classes

<code>MeasureWidget</code>	Provides the measure widget for the Visualization Interface Tool Plotter class.
----------------------------	---

#### MeasureWidget

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.measure.MeasureWidget(plotter_helper:  
    an-  
    sys.tools.visualization  
    dark_mode:  
    bool  
    =  
    False)
```

Bases: `ansys.tools.visualization_interface.backends.pyvista.widgets.widget.PlotterWidget`

Provides the measure widget for the Visualization Interface Tool Plotter class.

#### Parameters

**plotter\_helper**  
`[PlotterHelper]` Plotter to add the measure widget to.

**dark\_mode**  
`[bool, optional]` Whether to activate the dark mode or not.

### Overview

### Methods

<code>callback</code>	Remove or add the measurement widget actor upon click.
<code>update</code>	Define the measurement widget button parameters.

### Attributes

<code>plotter_helper</code>
-----------------------------

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.measure import  
    MeasureWidget
```

### Attribute detail

`MeasureWidget.plotter_helper`

## Method detail

`MeasureWidget.callback(state: bool) → None`

Remove or add the measurement widget actor upon click.

### Parameters

`state`

[bool] Whether the state of the button, which is inherited from PyVista, is active.

`MeasureWidget.update() → None`

Define the measurement widget button parameters.

## Description

Provides the measure widget for the PyAnsys plotter.

## The `mesh_slider.py` module

### Summary

### Classes

---

`MeshSliderWidget` Provides the mesh slider widget for the Visualization Interface Tool Plotter class.

---

#### MeshSliderWidget

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.mesh_slider.MeshSliderWidget(plotter_helper: ansys.tools.visualization_interface.backends.pyvista.plotter_helper, dark_mode: bool = False)
```

Bases: `ansys.tools.visualization_interface.backends.pyvista.widgets.widget.Widget`

Provides the mesh slider widget for the Visualization Interface Tool Plotter class.

### Parameters

`plotter_helper`

[PlotterHelper] Plotter to add the mesh slider widget to.

`dark_mode`

[bool, optional] Whether to activate the dark mode or not.

## Overview

## Methods

---

`callback` Remove or add the mesh slider widget actor upon click.

`update` Define the mesh slider widget button parameters.

---

## Attributes

*plotter\_helper*

## Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.mesh_slider import..  
..MeshSliderWidget
```

## Attribute detail

`MeshSliderWidget.plotter_helper`

## Method detail

`MeshSliderWidget.callback(state: bool) → None`

Remove or add the mesh slider widget actor upon click.

### Parameters

`state`

[bool] Whether the state of the button, which is inherited from PyVista, is active.

`MeshSliderWidget.update() → None`

Define the mesh slider widget button parameters.

## Description

Provides the measure widget for the PyAnsys plotter.

## The next\_button.py module

### Summary

### Classes

*NextButton* Provides next frame control for animations.

### Enums

*NextButtonConfig* Provides configuration for the next button.

### NextButton

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.next_button.NextButton(plotter:  
    pyvista.Plotter,  
    an-  
    i-  
    ma-  
    tion,  
    dark_mode:  
    bool  
    =  
    False)
```

Bases: `ansys.tools.visualization_interface.backends.pyvista.widgets.button.Button`

Provides next frame control for animations.

This button steps forward one frame.

#### Parameters

**plotter**  
[Plotter] Plotter to draw the button on.

**animation**  
[Animation] Animation instance to control.

**dark\_mode**  
[bool, optional] Whether to activate the dark mode or not.

### Overview

#### Methods

```
callback Step to next frame.
```

#### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.next_button import  
    NextButton
```

#### Method detail

`NextButton.callback(state: bool) → None`

Step to next frame.

#### Parameters

**state**  
[bool] Whether the button is active.

#### NextButtonConfig

```
class  
ansys.tools.visualization_interface.backends.pyvista.widgets.next_button.NextButtonConfig
```

Bases: `enum.Enum`

Provides configuration for the next button.

## Overview

### Attributes

[NEXT](#)

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.next_button import_
    NextButtonConfig
```

### Attribute detail

NextButtonConfig.NEXT = (0, 'next', (160, 10))

### Description

Provides the next button widget for animation control.

## The parallel\_projection.py module

### Summary

### Classes

<i>ParallelProjectionButton</i>	Toggle parallel projection for the camera.
---------------------------------	--

#### ParallelProjectionButton

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.parallel_projection.ParallelProjec
```

Bases: [ansys.tools.visualization\\_interface.backends.pyvista.widgets.widget.PlotterWidget](#)

Toggle parallel projection for the camera.

#### Parameters

**plotter**  
[Plotter] Plotter to add the widget to.

**dark\_mode**  
[bool, optional] Whether dark mode is active.

## Overview

### Methods

<code>callback</code>	Toggle parallel projection.
<code>update</code>	Update the button appearance.

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.parallel_projection_
    ↪ import ParallelProjectionButton
```

### Method detail

`ParallelProjectionButton.callback(state: bool) → None`

Toggle parallel projection.

#### Parameters

##### `state`

[bool] Button state from PyVista.

`ParallelProjectionButton.update() → None`

Update the button appearance.

### Description

Provides the parallel projection button widget.

## The `pick_rotation_center.py` module

### Summary

### Classes

`PickRotCenterButton` Provides the pick rotation center widget for the Visualization Interface Tool Plotter class.

#### `PickRotCenterButton`

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.pick_rotation_center.PickRotCenterBu
```

Bases: `ansys.tools.visualization_interface.backends.pyvista.widgets.widget.  
PlotterWidget`

Provides the pick rotation center widget for the Visualization Interface Tool Plotter class.

#### Parameters

**plotter\_helper**

[PlotterHelper] Plotter to add the pick rotation center widget to.

**dark\_mode**

[bool, optional] Whether to activate the dark mode or not.

**Overview****Methods**

<code>callback</code>	Remove or add the pick rotation center widget actor upon click.
<code>update</code>	Define the measurement widget button parameters.

**Attributes****`plotter_helper`****Import detail**

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.pick_rotation_center_
import PickRotCenterButton
```

**Attribute detail**

PickRotCenterButton.**plotter\_helper**

**Method detail**

PickRotCenterButton.**callback**(state: `bool`) → `None`

Remove or add the pick rotation center widget actor upon click.

**Parameters****state**

[bool] Whether the state of the button, which is inherited from PyVista, is active.

PickRotCenterButton.**update**() → `None`

Define the measurement widget button parameters.

**Description**

Provides the measure widget for the PyAnsys plotter.

**The play\_pause\_button.py module****Summary****Classes**

<code>PlayPauseButton</code>	Provides play/pause toggle control for animations.
------------------------------	--

## Enums

<code>PlayPauseButtonConfig</code>	Provides configuration for the play/pause button.
------------------------------------	---

## PlayPauseButton

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.play_pause_button.PlayPauseButton(plo
pyv
an-
i-
ma-
tion-
dan-
bo-
=
Fa-
```

Bases: `ansys.tools.visualization_interface.backends.pyvista.widgets.button.Button`

Provides play/pause toggle control for animations.

This button switches between play and pause icons dynamically based on the animation state.

### Parameters

**plotter**  
[Plotter] Plotter to draw the button on.

**animation**  
[Animation] Animation instance to control.

**dark\_mode**  
[bool, optional] Whether to activate the dark mode or not.

## Overview

### Methods

<code>callback</code>	Toggle between play and pause.
-----------------------	--------------------------------

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.play_pause_button_
↪import PlayPauseButton
```

### Method detail

`PlayPauseButton.callback(state: bool) → None`

Toggle between play and pause.

#### Parameters

**state**  
[bool] Whether the button is active (playing).

## PlayPauseButtonConfig

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.play_pause_button.
PlayPauseButtonConfig
```

Bases: `enum.Enum`

Provides configuration for the play/pause button.

### Overview

#### Attributes

<code>PLAY_PAUSE</code>
-------------------------

#### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.play_pause_button_
import PlayPauseButtonConfig
```

#### Attribute detail

`PlayPauseButtonConfig.PLAY_PAUSE = (0, 'play', (10, 10))`

#### Description

Provides the play/pause button widget for animation control.

#### The previous\_button.py module

#### Summary

#### Classes

<code>PreviousButton</code> Provides previous frame control for animations.
---

#### Enums

<code>PreviousButtonConfig</code> Provides configuration for the previous button.
---

#### PreviousButton

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.previous_button.PreviousButton(plotter,
pyvista,
an-
i-
ma-
tion,
dark_n
bool
=
False)
```

Bases: `ansys.tools.visualization_interface.backends.pyvista.widgets.button.Button`

Provides previous frame control for animations.

This button steps backward one frame.

#### Parameters

##### `plotter`

[`Plotter`] Plotter to draw the button on.

##### `animation`

[`Animation`] Animation instance to control.

##### `dark_mode`

[`bool`, `optional`] Whether to activate the dark mode or not.

## Overview

### Methods

<code>callback</code>	Step to previous frame.
-----------------------	-------------------------

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.previous_button import_
    PreviousButton
```

### Method detail

`PreviousButton.callback(state: bool) → None`

Step to previous frame.

#### Parameters

##### `state`

[`bool`] Whether the button is active.

### PreviousButtonConfig

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.previous_button.
    PreviousButtonConfig
```

Bases: `enum.Enum`

Provides configuration for the previous button.

## Overview

### Attributes

<code>PREVIOUS</code>
-----------------------

## Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.previous_button import_
    PreviousButtonConfig
```

## Attribute detail

PreviousButtonConfig.PREVIOUS = (0, 'previous', (110, 10))

## Description

Provides the previous button widget for animation control.

## The ruler.py module

### Summary

### Classes

<i>Ruler</i>	Provides the ruler widget for the Visualization Interface Tool Plotter class.
--------------	---

#### Ruler

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.ruler.Ruler(plotter:
    pyvista.Plotter,
    dark_mode:
    bool =
    False)
```

Bases: *ansys.tools.visualization\_interface.backends.pyvista.widgets.widget.Widget*

Provides the ruler widget for the Visualization Interface Tool Plotter class.

#### Parameters

##### plotter

[*Plotter*] Provides the plotter to add the ruler widget to.

##### dark\_mode

[*bool, optional*] Whether to activate the dark mode or not.

## Overview

## Methods

<i>callback</i>	Remove or add the ruler widget actor upon click.
-----------------	--

<i>update</i>	Define the configuration and representation of the ruler widget button.
---------------	---

## Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.ruler import Ruler
```

## Method detail

`Ruler.callback(state: bool) → None`

Remove or add the ruler widget actor upon click.

### Parameters

#### state

[bool] Whether the state of the button, which is inherited from PyVista, is True.

### Notes

This method provides a callback function for the ruler widet. It is called every time the ruler widget is clicked.

`Ruler.update() → None`

Define the configuration and representation of the ruler widget button.

## Description

Provides the ruler widget for the Visualization Interface Tool plotter.

## The `save_gif_button.py` module

### Summary

### Classes

<code>SaveGifButton</code>	Provides GIF export control for animations.
----------------------------	---

### Enums

<code>SaveGifButtonConfig</code>	Provides configuration for the save GIF button.
----------------------------------	---

### `SaveGifButton`

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.save_gif_button.SaveGifButton(plotter:  
    pyvista.  
    an-  
    i-  
    ma-  
    tion,  
    dark_ma-  
    bool  
    =  
    False)
```

Bases: `ansys.tools.visualization_interface.backends.pyvista.widgets.button.Button`

Provides GIF export control for animations.

This button exports the animation as a timestamped GIF file.

### Parameters

#### plotter

[Plotter] Plotter to draw the button on.

**animation**

[Animation] Animation instance to control.

**dark\_mode**

[bool, optional] Whether to activate the dark mode or not.

**Overview****Methods**

<code>callback</code>	Export animation as GIF file.
-----------------------	-------------------------------

**Import detail**

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.save_gif_button import_
SaveGifButton
```

**Method detail**

`SaveGifButton.callback(state: bool) → None`

Export animation as GIF file.

**Parameters****state**

[bool] Whether the button is active.

**SaveGifButtonConfig**

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.save_gif_button.
SaveGifButtonConfig
```

Bases: `enum.Enum`

Provides configuration for the save GIF button.

**Overview****Attributes**

<code>SAVE_GIF</code>
-----------------------

**Import detail**

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.save_gif_button import_
SaveGifButtonConfig
```

**Attribute detail**

`SaveGifButtonConfig.SAVE_GIF = (0, 'save', (210, 10))`

## Description

Provides the save GIF button widget for animation control.

## The screenshot.py module

### Summary

### Classes

<code>ScreenshotButton</code>	Provides the screenshot widget for the Visualization Interface Tool Plotter class.
-------------------------------	--

#### ScreenshotButton

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.screenshot.ScreenshotButton(plotter:  
    pyvista.Plotter  
    dark_mode:  
    bool  
    =  
    False)
```

Bases: `ansys.tools.visualization_interface.backends.pyvista.widgets.widget.  
PlotterWidget`

Provides the screenshot widget for the Visualization Interface Tool Plotter class.

#### Parameters

##### plotter

[`Plotter`] Provides the plotter to add the screenshot widget to.

##### dark\_mode

[`bool`, optional] Whether to activate the dark mode or not.

## Overview

### Methods

<code>callback</code>	Remove or add the screenshot widget actor upon click.
<code>update</code>	Define the configuration and representation of the screenshot widget button.

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.screenshot import_  
↳ ScreenshotButton
```

### Method detail

`ScreenshotButton.callback(state: bool) → None`

Remove or add the screenshot widget actor upon click.

#### Parameters

##### state

[`bool`] Whether the state of the button, which is inherited from PyVista, is True.

## Notes

This method provides a callback function for the screenshot widget. It is called every time the screenshot widget is clicked.

`ScreenshotButton.update() → None`

Define the configuration and representation of the screenshot widget button.

## Description

Provides the screenshot widget for the Visualization Interface Tool plotter.

### The `stop_button.py` module

#### Summary

#### Classes

<code>StopButton</code>	Provides stop control for animations.
-------------------------	---------------------------------------

#### Enums

<code>StopButtonConfig</code>	Provides configuration for the stop button.
-------------------------------	---

#### `StopButton`

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.stop_button.StopButton(plotter:  
    pyvista.Plotter,  
    an-  
    i-  
    ma-  
    tion,  
    dark_mode:  
    bool  
    =  
    False)
```

Bases: `ansys.tools.visualization_interface.backends.pyvista.widgets.button.Button`

Provides stop control for animations.

This button stops the animation and resets to the first frame.

#### Parameters

**plotter**  
[Plotter] Plotter to draw the button on.

**animation**  
[Animation] Animation instance to control.

**dark\_mode**  
[bool, optional] Whether to activate the dark mode or not.

## Overview

### Methods

```
callback Stop animation and reset to first frame.
```

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.stop_button import_
StopButton
```

### Method detail

`StopButton.callback(state: bool) → None`

Stop animation and reset to first frame.

#### Parameters

##### state

[bool] Whether the button is active.

### StopButtonConfig

#### class

```
ansys.tools.visualization_interface.backends.pyvista.widgets.stop_button.StopButtonConfig
```

Bases: `enum.Enum`

Provides configuration for the stop button.

## Overview

### Attributes

```
STOP
```

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.stop_button import_
StopButtonConfig
```

### Attribute detail

`StopButtonConfig.STOP = (0, 'stop', (60, 10))`

### Description

Provides the stop button widget for animation control.

## The tree\_menu\_toggle.py module

### Summary

### Classes

<code>TreeMenuToggleButton</code>	Provides a button to show/hide the tree menu widget.
-----------------------------------	--

#### `TreeMenuToggleButton`

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.tree_menu_toggle.TreeMenuToggleButton
```

Bases: `ansys.tools.visualization_interface.backends.pyvista.widgets.widget.  
PlotterWidget`

Provides a button to show/hide the tree menu widget.

This button toggles the visibility of the dynamic tree menu panel.

#### Parameters

##### `plotter`

[`Plotter`] Plotter to draw the button on.

##### `dark_mode`

[`bool`, optional] Whether to activate dark mode. Default is False.

##### `tree_menu`

[`DynamicTreeMenuWidget`, optional] The tree menu widget to toggle. If None, will search for it in plotter widgets.

### Overview

### Methods

<code>update</code>	Assign the image that represents the button.
<code>callback</code>	Toggle the tree menu visibility.

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.tree_menu_toggle.  
import TreeMenuToggleButton
```

### Method detail

`TreeMenuToggleButton.update()` → `None`

Assign the image that represents the button.

`TreeMenuToggleButton.callback(state: bool)` → `None`

Toggle the tree menu visibility.

## Parameters

### state

[bool] The state of the button (True = show menu, False = hide menu).

## Description

Provides a button to toggle the tree menu visibility.

## The view\_button.py module

### Summary

### Classes

<code>ViewButton</code>	Provides for changing the view.
-------------------------	---------------------------------

### Enums

<code>ViewDirection</code>	Provides an enum with the available views.
----------------------------	--

## ViewButton

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.view_button.ViewButton(plotter:  
    pyvista.Plotter,  
    di-  
    rec-  
    tion:  
    tu-  
    ple,  
    dark_mode:  
    bool  
    =  
    False)
```

Bases: `ansys.tools.visualization_interface.backends.pyvista.widgets.button.Button`

Provides for changing the view.

## Parameters

### plotter

[Plotter] Plotter to draw the buttons on.

### direction

[ViewDirection] Direction of the view.

### dark\_mode

[bool, optional] Whether to activate the dark mode or not.

## Overview

## Methods

<code>callback</code>	Change the view depending on button interaction.
-----------------------	--

## Attributes

*direction*

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.view_button import...
↪ViewButton
```

### Attribute detail

`ViewButton.direction`

### Method detail

`ViewButton.callback(state: bool) → None`

Change the view depending on button interaction.

#### Parameters

`state`

[bool] Whether the state of the button, which is inherited from PyVista, is True.

#### Raises

`NotImplementedError`

Raised if the specified direction is not implemented.

## ViewDirection

### class

`ansys.tools.visualization_interface.backends.pyvista.widgets.view_button.ViewDirection`

Bases: `enum.Enum`

Provides an enum with the available views.

### Overview

## Attributes

<i>XYPLUS</i>
<i>XYMINUS</i>
<i>XZPLUS</i>
<i>XZMINUS</i>
<i>YZPLUS</i>
<i>YZMINUS</i>
<i>ISOMETRIC</i>

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.view_button import...
↪ViewDirection
```

## Attribute detail

```
ViewDirection.XYPLUS = (0, '+xy', (5, 280))
ViewDirection.XYMINUS = (1, '-xy', (5, 311))
ViewDirection.XZPLUS = (2, '+xz', (5, 342))
ViewDirection.XZMINUS = (3, '-xz', (5, 373))
ViewDirection.YZPLUS = (4, '+yz', (5, 404))
ViewDirection.YZMINUS = (5, '-yz', (5, 435))
ViewDirection.ISOMETRIC = (6, 'isometric', (5, 466))
```

## Description

Provides the view button widget for changing the camera view.

## The `widget.py` module

### Summary

### Classes

<code>PlotterWidget</code>	Provides an abstract class for plotter widgets.
----------------------------	---

#### `PlotterWidget`

```
class ansys.tools.visualization_interface.backends.pyvista.widgets.widget.PlotterWidget(plotter:
    pyvista.Plotter)
```

Bases: `abc.ABC`

Provides an abstract class for plotter widgets.

##### Parameters

`plotter`  
[`Plotter`] Plotter instance to add the widget to.

##### Notes

These widgets are intended to be used with PyVista plotter objects. More specifically, the way in which this abstraction has been built ensures that these widgets can be easily integrated with the Visualization Interface Tool's widgets.

## Overview

### Abstract methods

<code>callback</code>	General callback function for <code>PlotterWidget</code> objects.
<code>update</code>	General update function for <code>PlotterWidget</code> objects.

## Properties

<i>plotter</i>	Plotter object that the widget is assigned to.
----------------	--

## Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.widgets.widget import...
    PlotterWidget
```

## Property detail

**property** PlotterWidget.**plotter**: `pyvista.Plotter`

Plotter object that the widget is assigned to.

## Method detail

**abstractmethod** PlotterWidget.**callback**(*state*) → None

General callback function for PlotterWidget objects.

**abstractmethod** PlotterWidget.**update**() → None

General update function for PlotterWidget objects.

## Description

Provides the abstract implementation of plotter widgets.

## Description

Provides widgets for the Visualization Interface Tool plotter.

## The animation.py module

### Summary

### Classes

<i>FrameSequence</i>	Abstract interface for frame data sources.
<i>InMemoryFrameSequence</i>	Frame sequence with all frames pre-loaded in memory.
<i>Animation</i>	Animation controller for PyVista visualizations.

### Enums

<i>AnimationState</i>	Animation playback states.
-----------------------	----------------------------

### Constants

<i>DARK_MODE_THRESHOLD</i>
----------------------------

## FrameSequence

```
class ansys.tools.visualization_interface.backends.pyvista.animation.FrameSequence
```

Bases: abc.ABC

Abstract interface for frame data sources.

This class provides an abstraction for different frame storage strategies, allowing for in-memory, lazy-loaded, or computed frame sequences.

## Overview

### Abstract methods

<code>get_frame</code>	Retrieve frame at given index.
<code>__len__</code>	Return total number of frames.

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.animation import FrameSequence
```

### Method detail

**abstractmethod** `FrameSequence.get_frame(index: int) → Any`

Retrieve frame at given index.

#### Parameters

`index`

[int] Frame index to retrieve.

#### Returns

`Any`

Frame data (typically a PyVista mesh or MeshObjectPlot).

**abstractmethod** `FrameSequence.__len__() → int`

Return total number of frames.

#### Returns

`int`

Total frame count.

## InMemoryFrameSequence

```
class ansys.tools.visualization_interface.backends.pyvista.animation.InMemoryFrameSequence(frames:  
List[Any])
```

Bases: FrameSequence

Frame sequence with all frames pre-loaded in memory.

This is the simplest strategy, suitable for small to medium datasets where all frames can fit in memory.

#### Parameters

`frames`

[List[Any]] List of frame objects (meshes or mesh objects).

## Overview

### Methods

<code>get_frame</code>	Retrieve frame at given index.
------------------------	--------------------------------

### Special methods

<code>__len__</code>	Return total number of frames.
----------------------	--------------------------------

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.animation import_
    InMemoryFrameSequence
```

### Method detail

`InMemoryFrameSequence.get_frame(index: int) → Any`

Retrieve frame at given index.

`InMemoryFrameSequence.__len__() → int`

Return total number of frames.

### Animation

```
class ansys.tools.visualization_interface.backends.pyvista.animation.Animation(plotter:
    pyvista.Plotter,
    frames:
        FrameSe-
        quence | |
        List[Any],
    fps: int = 30,
    loop: bool =
        False,
    scalar_bar_args:
        dict | |
        None = None,
    **plot_kwargs)
```

Animation controller for PyVista visualizations.

This class manages animation playback, providing play/pause/stop controls, frame stepping, timeline scrubbing, and export capabilities.

#### Parameters

##### `plotter`

[`pv.Plotter`] PyVista plotter instance to animate.

##### `frames`

[`FrameSequence` or `List[Any]`] Frame sequence or list of frame objects to animate.

##### `fps`

[`int`, optional] Frames per second for playback (default: 30).

**loop**

[bool, optional] Whether to loop animation continuously (default: False).

**scalar\_bar\_args**

[dict, optional] Scalar bar and rendering arguments to apply. Supports:  
- clim : tuple  
- Fixed color scale (min, max) for all frames  
- title : str - Scalar bar title  
- color : str - Scalar bar text color  
- Other parameters accepted by PyVista's add\_mesh method

## Examples

Create and play a simple animation:

```
>>> from ansys.tools.visualization_interface import Plotter
>>> plotter = Plotter(backend='pyvista')
>>> frames = [mesh1, mesh2, mesh3]
>>> animation = plotter.animate(frames, fps=30, loop=True)
>>> animation.play()
```

## Overview

### Methods

<i>play</i>	Start or resume animation playback.
<i>pause</i>	Pause animation playback.
<i>stop</i>	Stop animation and reset to first frame.
<i>step_forward</i>	Advance one frame forward.
<i>step_backward</i>	Rewind one frame backward.
<i>seek</i>	Jump to specific frame.
<i>save</i>	Export animation to video file.
<i>show</i>	Display animation with the plotter.

### Properties

<i>state</i>	Current animation state.
<i>current_frame</i>	Current frame index.
<i>total_frames</i>	Total number of frames.
<i>fps</i>	Frames per second.

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.animation import Animation
```

### Property detail

**property** Animation.state: *AnimationState*

Current animation state.

**property** Animation.current\_frame: *int*

Current frame index.

**property** Animation.total\_frames: *int*

Total number of frames.

```
property Animation.fps: int
```

Frames per second.

## Method detail

```
Animation.play()
```

Start or resume animation playback.

```
Animation.pause()
```

Pause animation playback.

```
Animation.stop()
```

Stop animation and reset to first frame.

```
Animation.step_forward()
```

Advance one frame forward.

```
Animation.step_backward()
```

Rewind one frame backward.

```
Animation.seek(frame_index: int)
```

Jump to specific frame.

### Parameters

**frame\_index**

[int] Target frame index.

```
Animation.save(filename: str | pathlib.Path, quality: int = 5, framerate: int | None = None, close_plotter: bool = False)
```

Export animation to video file.

### Parameters

**filename**

[str or Path] Output filename (.mp4, .gif, .avi).

**quality**

[int, optional] Video quality (1-10, higher is better). Default is 5.

**framerate**

[int, optional] Output framerate. If None, uses animation fps.

**close\_plotter**

[bool, optional] If True, closes plotter after saving. Default is False.

```
Animation.show(show_controls: bool = True, **kwargs)
```

Display animation with the plotter.

### Parameters

**show\_controls**

[bool, optional] If True, shows interactive controls. Default is True.

**\*\*kwargs**

Additional arguments passed to plotter.show().

## AnimationState

```
class ansys.tools.visualization_interface.backends.pyvista.animation.AnimationState
Bases: enum.Enum
Animation playback states.
```

### Overview

#### Attributes

STOPPED
PLAYING
PAUSED

#### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.animation import AnimationState
```

#### Attribute detail

```
AnimationState.STOPPED = 'stopped'
AnimationState.PLAYING = 'playing'
AnimationState.PAUSED = 'paused'
```

#### Description

Animation support for PyVista backend.

#### Module detail

```
animation.DARK_MODE_THRESHOLD = 120
```

## The picker.py module

### Summary

### Classes

<i>AbstractPicker</i>	Abstract base class for pickers.
<i>Picker</i>	Class to manage picking and hovering of objects in the plotter.

#### AbstractPicker

```
class ansys.tools.visualization_interface.backends.pyvista.picker.AbstractPicker(plotter_backend:
an-
sys.tools.visualization_interfa
**kwargs)
```

Bases: abc.ABC

Abstract base class for pickers.

## Overview

### Abstract methods

<code>pick_select_object</code>	Determine actions to take when an object is selected.
<code>pick_unselect_object</code>	Determine actions to take when an object is unselected.
<code>hover_select_object</code>	Determine actions to take when an object is hovered over.
<code>hover_unselect_object</code>	Determine actions to take when an object is unhovered.

### Properties

<code>picked_dict</code>	Return the dictionary of picked objects.
--------------------------	--

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.picker import AbstractPicker
```

### Property detail

`property AbstractPicker.picked_dict: dict`

#### Abstractmethod

Return the dictionary of picked objects.

### Method detail

`abstractmethod AbstractPicker.pick_select_object(custom_object: an-`  
`sys.tools.visualization_interface.types.mesh_object_plot.MeshObjectPl-`  
`| an-`  
`sys.tools.visualization_interface.types.edge_plot.EdgePlot,`  
`pt: numpy.ndarray) → None`

Determine actions to take when an object is selected.

`abstractmethod AbstractPicker.pick_unselect_object(custom_object: an-`  
`sys.tools.visualization_interface.types.mesh_object_plot.MeshObjec-`  
`| an-`  
`sys.tools.visualization_interface.types.edge_plot.EdgePlot)`  
`→ None`

Determine actions to take when an object is unselected.

`abstractmethod AbstractPicker.hover_select_object(custom_object: an-`  
`sys.tools.visualization_interface.types.mesh_object_plot.MeshObjectPl-`  
`| an-`  
`sys.tools.visualization_interface.types.edge_plot.EdgePlot,`  
`pt: numpy.ndarray) → None`

Determine actions to take when an object is hovered over.

`abstractmethod AbstractPicker.hover_unselect_object(custom_object: an-`  
`sys.tools.visualization_interface.types.mesh_object_plot.MeshObjec-`  
`| an-`  
`sys.tools.visualization_interface.types.edge_plot.EdgePlot)`  
`→ None`

Determine actions to take when an object is unhovered.

## Picker

```
class ansys.tools.visualization_interface.backends.pyvista.picker.Picker(plotter_backend: an-
    sys.tools.visualization_interface.backends.pyvista.plotter.Plotter,
    plot_picked_names: bool = True)
```

Bases: `AbstractPicker`

Class to manage picking and hovering of objects in the plotter.

This class is responsible for managing the selection and deselection of objects in the plotter, both through direct picking and hovering. It keeps track of the currently selected and hovered objects, and provides methods to select and unselect them.

### Parameters

#### `plotter_backend`

[Plotter] The plotter instance to which this picker is attached.

#### `plot_picked_names`

[bool, optional] Whether to display the names of picked objects in the plotter. Defaults to True.

## Overview

## Methods

<code>pick_select_object</code>	Add actor to picked list and add label if required.
<code>pick_unselect_object</code>	Remove actor from picked list and remove label if required.
<code>hover_select_object</code>	Add label to hovered object if required.
<code>hover_unselect_object</code>	Remove all hover labels from the scene.

## Properties

<code>picked_dict</code>	Return the dictionary of picked objects.
--------------------------	--

## Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.picker import Picker
```

## Property detail

### `property Picker.picked_dict: dict`

Return the dictionary of picked objects.

## Method detail

### `Picker.pick_select_object(custom_object:`

```
        ansys.tools.visualization_interface.types.mesh_object_plot.MeshObjectPlot |
        ansys.tools.visualization_interface.types.edge_plot.EdgePlot, pt: numpy.ndarray) |
        → None
```

Add actor to picked list and add label if required.

#### Parameters

##### `custom_object`

[Union[MeshObjectPlot, EdgePlot]] The object to be selected.

##### `pt`

[np.ndarray] The point where the object was picked.

`Picker.pick_unselect_object(custom_object:`

```
ansys.tools.visualization_interface.types.mesh_object_plot.MeshObjectPlot |
ansys.tools.visualization_interface.types.edge_plot.EdgePlot) → None
```

Remove actor from picked list and remove label if required.

#### Parameters

##### `custom_object`

[Union[MeshObjectPlot, EdgePlot]] The object to be unselected.

`Picker.hover_select_object(custom_object:`

```
ansys.tools.visualization_interface.types.mesh_object_plot.MeshObjectPlot |
ansys.tools.visualization_interface.types.edge_plot.EdgePlot, actor:
pyvista.Actor) → None
```

Add label to hovered object if required.

#### Parameters

##### `custom_object`

[Union[MeshObjectPlot, EdgePlot]] The object to be hovered over.

##### `actor`

[vtkActor] The actor corresponding to the hovered object.

`Picker.hover_unselect_object()`

Remove all hover labels from the scene.

## Description

Module for managing picking and hovering of objects in a PyVista plotter.

### The `pyvista.py` module

#### Summary

#### Classes

<code>PyVistaBackendInterface</code>	Provides the interface for the Visualization Interface Tool plotter.
<code>PyVistaBackend</code>	Provides the generic plotter implementation for PyAnsys libraries.

#### Constants

<code>DARK_MODE_THRESHOLD</code>
----------------------------------

**PyVistaBackendInterface**

```
class ansys.tools.visualization_interface.backends.pyvista.pyvista.PyVistaBackendInterface(use_trame:  
    bool  
    |  
    None  
    =  
    None,  
    al-  
    low_picking:  
    bool  
    |  
    None  
    =  
    False,  
    al-  
    low_hovering:  
    bool  
    |  
    None  
    =  
    False,  
    plot_picked_name:  
    bool  
    |  
    None  
    =  
    False,  
    show_plane:  
    bool  
    |  
    None  
    =  
    False,  
    use_qt:  
    bool  
    |  
    None  
    =  
    False,  
    show_qt:  
    bool  
    |  
    None  
    =  
    True,  
    cus-  
    tom_picker:  
    an-  
    sys.tools.visualization  
    =  
    None,  
    cus-  
    tom_picker_kwarg:  
    Dict[str,  
    Any]  
    |  
    None  
    =  
    None,  
    **plot-  
    ter_kwarg)
```

Bases: `ansys.tools.visualization_interface.backends._base.BaseBackend`

Provides the interface for the Visualization Interface Tool plotter.

This class is intended to be used as a base class for the custom plotters in the different PyAnsys libraries. It provides the basic plotter functionalities, such as adding objects and enabling widgets and picking capabilities. It also provides the ability to show the plotter using the `trame` service.

You can override the `plot_iter()`, `plot()`, and `picked_operation()` methods. The `plot_iter()` method is intended to plot a list of objects to the plotter, while the `plot()` method is intended to plot a single object to the plotter. The `show()` method is intended to show the plotter. The `picked_operation()` method is intended to perform an operation on the picked objects.

### Parameters

<b>use_trame</b>
[Optional[bool], default: <code>None</code> ] Whether to activate the usage of the trame UI instead of the Python window.
<b>allow_picking</b>
[Optional[bool], default: <code>False</code> ] Whether to allow picking capabilities in the window. Incompatible with hovering. Picking will take precedence over hovering.
<b>allow_hovering</b>
[Optional[bool], default: <code>False</code> ] Whether to allow hovering capabilities in the window. Incompatible with picking. Picking will take precedence over hovering.
<b>plot_picked_names</b>
[Optional[bool], default: <code>False</code> ] Whether to plot the names of the picked objects.
<b>show_plane</b>
[Optional[bool], default: <code>False</code> ] Whether to show the plane in the plotter.
<b>use_qt</b>
[Optional[bool], default: <code>False</code> ] Whether to use the Qt backend for the plotter.
<b>show_qt</b>
[Optional[bool], default: <code>True</code> ] Whether to show the Qt window.
<b>custom_picker</b>
[AbstractPicker, default: <code>None</code> ] Custom picker class that extends the <code>AbstractPicker</code> class.
<b>custom_picker_kwargs</b>
[Optional[Dict[str, Any]], default: <code>None</code> ] Keyword arguments to pass to the custom picker class.

## Overview

### Abstract methods

<code>plot_iter</code>	Plot one or more compatible objects to the plotter.
<code>plot</code>	Plot a single object to the plotter.

## Methods

<code>enable_widgets</code>	Enable the widgets for the plotter.
<code>add_widget</code>	Add one or more custom widgets to the plotter.
<code>picker_callback</code>	Define the callback for the element picker.
<code>hover_callback</code>	Define the callback for the element hover.
<code>focus_point_selection</code>	Focus the camera on a selected actor.
<code>compute_edge_object_map</code>	Compute the mapping between plotter actors and EdgePlot objects.
<code>enable_picking</code>	Enable picking capabilities in the plotter.
<code>enable_set_focus_center</code>	Enable setting the focus of the camera to the picked point.
<code>enable_hover</code>	Enable hover capabilities in the plotter.
<code>disable_picking</code>	Disable picking capabilities in the plotter.
<code>disable_hover</code>	Disable hover capabilities in the plotter.
<code>disable_center_focus</code>	Disable setting the focus of the camera to the picked point.
<code>show</code>	Plot and show any PyAnsys object.
<code>show_plotter</code>	Show the plotter or start the <code>trame</code> service.
<code>picked_operation</code>	Perform an operation on the picked objects.

## Properties

<code>pv_interface</code>	PyVista interface.
<code>scene</code>	PyVista scene.

## Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.pyvista import_
    ↪PyVistaBackendInterface
```

## Property detail

`property PyVistaBackendInterface.pv_interface:`  
`ansys.tools.visualization_interface.backends.pyvista.pyvista_interface.PyVistaInterface`

PyVista interface.

`property PyVistaBackendInterface.scene: pyvista.Plotter`

PyVista scene.

## Method detail

`PyVistaBackendInterface.enable_widgets(dark_mode: bool = False) → None`

Enable the widgets for the plotter.

### Parameters

#### `dark_mode`

[bool, default: `False`] Whether to use dark mode for the widgets.

`PyVistaBackendInterface.add_widget(widget: an-
 sys.tools.visualization_interface.backends.pyvista.widgets.widget.PlotterWidget
 |
 List[ansys.tools.visualization_interface.backends.pyvista.widgets.widget.PlotterWidget])`

Add one or more custom widgets to the plotter.

**Parameters****widget**

[Union[PlotterWidget, List[PlotterWidget]]] One or more custom widgets.

PyVistaBackendInterface.**picker\_callback**(*actor*: *pyvista.Actor*) → None

Define the callback for the element picker.

**Parameters****actor**

[Actor] Actor to select for the picker.

PyVistaBackendInterface.**hover\_callback**(*\_widget*, *event\_name*) → None

Define the callback for the element hover.

**Parameters****actor**

[Actor] Actor to hover for the picker.

PyVistaBackendInterface.**focus\_point\_selection**(*actor*: *pyvista.Actor*) → None

Focus the camera on a selected actor.

**Parameters****actor**

[Actor] Actor to focus the camera on.

PyVistaBackendInterface.**compute\_edge\_object\_map**() → Dict[*pyvista.Actor*, *an-*  
*sys.tools.visualization\_interface.types.edge\_plot.EdgePlot*]

Compute the mapping between plotter actors and EdgePlot objects.

**Returns****Dict[Actor, EdgePlot]**

Dictionary defining the mapping between plotter actors and EdgePlot objects.

PyVistaBackendInterface.**enable\_picking**()

Enable picking capabilities in the plotter.

PyVistaBackendInterface.**enable\_set\_focus\_center**()

Enable setting the focus of the camera to the picked point.

PyVistaBackendInterface.**enable\_hover**()

Enable hover capabilities in the plotter.

PyVistaBackendInterface.**disable\_picking**()

Disable picking capabilities in the plotter.

PyVistaBackendInterface.**disable\_hover**()

Disable hover capabilities in the plotter.

PyVistaBackendInterface.**disable\_center\_focus**()

Disable setting the focus of the camera to the picked point.

PyVistaBackendInterface.**show**(*plottable\_object*: Any = None, *screenshot*: str | None = None, *view\_2d*: Dict = None, *name\_filter*: str = None, *dark\_mode*: bool = False, \*\*kwargs: Dict[str, Any]) → List[Any]

Plot and show any PyAnsys object.

The types of objects supported are MeshObjectPlot, pv.MultiBlock, and pv.PolyData.

## Parameters

### plottable\_object

[Any, default: `None`] Object or list of objects to plot.

### screenshot

[`str`, default: `None`] Path for saving a screenshot of the image that is being represented.

### view\_2d

[`Dict`, default: `None`] Dictionary with the plane and the viewup vectors of the 2D plane.

### name\_filter

[`str`, default: `None`] Regular expression with the desired name or names to include in the plotter.

### dark\_mode

[`bool`, default: `False`] Whether to use dark mode for the widgets.

### \*\*kwargs

[Any] Additional keyword arguments for the show or plot method.

## Returns

### List[Any]

List with the picked bodies in the picked order.

`PyVistaBackendInterface.show_plotter(screenshot: str | None = None, **kwargs) → None`

Show the plotter or start the `trame` service.

## Parameters

### plotter

[`Plotter`] Visualization Interface Tool plotter with the meshes added.

### screenshot

[`str`, default: `None`] Path for saving a screenshot of the image that is being represented.

**abstractmethod** `PyVistaBackendInterface.plot_iter(plottable_object: Any, name_filter: str = None, **plotting_options)`

Plot one or more compatible objects to the plotter.

## Parameters

### plottable\_object

[Any] One or more objects to add.

### name\_filter

[`str`, default: `None`.] Regular expression with the desired name or names to include in the plotter.

### \*\*plotting\_options

[`dict`, default: `None`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

**abstractmethod** `PyVistaBackendInterface.plot(plottable_object: Any, name_filter: str = None, **plotting_options)`

Plot a single object to the plotter.

## Parameters

### plottable\_object

[Any] Object to plot.

**name\_filter**

[`str`] Regular expression with the desired name or names to include in the plotter.

**\*\*plotting\_options**

[`dict`, default: `None`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

**PyVistaBackendInterface.picked\_operation()** → `None`

Perform an operation on the picked objects.

**PyVistaBackend**

```
class ansys.tools.visualization_interface.backends.pyvista.pyvista.PyVistaBackend(use_trame:  
    bool |  
    None =  
    None, al-  
    low_picking:  
    bool |  
    None =  
    False, al-  
    low_hovering:  
    bool |  
    None =  
    False,  
    plot_picked_names:  
    bool |  
    None =  
    True,  
    use_qt:  
    bool |  
    None =  
    False,  
    show_qt:  
    bool |  
    None =  
    False,  
    cus-  
    tom_picker:  
    an-  
    sys.tools.visualization_inter-  
    =None,  
    **plot-  
    ter_kwargs)
```

Bases: `PyVistaBackendInterface`

Provides the generic plotter implementation for PyAnsys libraries.

This class accepts `MeshObjectPlot`, `pv.MultiBlock` and `pv.PolyData` objects.

**Parameters****use\_trame**

[`bool`, default: `None`] Whether to enable the use of `trame`. The default is `None`, in which case the `USE_TRAME` global setting is used.

**allow\_picking**

[`Optional[bool]`, default: `False`] Whether to allow picking capabilities in the window.

Incompatible with hovering. Picking will take precedence over hovering.

#### allow\_hovering

[Optional[bool], default: False] Whether to allow hovering capabilities in the window.  
Incompatible with picking. Picking will take precedence over hovering.

#### plot\_picked\_names

[bool, default: True] Whether to plot the names of the picked objects.

## Overview

### Methods

<code>plot_iter</code>	Plot the elements of an iterable of any type of object to the scene.
<code>plot</code>	Plot a pyansys or PyVista object to the plotter.
<code>close</code>	Close the plotter for PyVistaQT.
<code>create_animation</code>	Create an animation from a sequence of frames.
<code>add_points</code>	Add point markers to the scene.
<code>add_lines</code>	Add line segments to the scene.
<code>add_planes</code>	Add a plane to the scene.
<code>add_text</code>	Add text to the scene.
<code>add_labels</code>	Add labels at 3D point locations.
<code>clear</code>	Clear all actors from the scene and reset the plotter.

### Properties

<code>base_plotter</code>	Return the base plotter object.
---------------------------	---------------------------------

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.pyvista import PyVistaBackend
```

### Property detail

#### property PyVistaBackend.base\_plotter

Return the base plotter object.

### Method detail

#### PyVistaBackend.plot\_iter(plotting\_list: List[Any], name\_filter: str = None, \*\*plotting\_options) → None

Plot the elements of an iterable of any type of object to the scene.

The types of objects supported are Body, Component, List[pv.PolyData], pv.MultiBlock, and Sketch.

#### Parameters

##### plotting\_list

[List[Any]] List of objects to plot.

##### name\_filter

[str, default: None] Regular expression with the desired name or names to include in the plotter.

**\*\*plotting\_options** [dict, default: `None`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

```
PyVistaBackend.plot(plottable_object: Any, name_filter: str = None, **plotting_options)
```

Plot a pyansys or PyVista object to the plotter.

## Parameters

## plottable\_object

[Any] Object to plot.

**name\_filter**

[str] Regular expression with the desired name or names to include in the plotter.

## **\*\*plotting\_options**

[dict, default: None] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

## PyVistaBackend.close()

Close the plotter for PyVistaQT.

```
PyVistaBackend.create_animation(frames: List[Any] | an-  
    sys.tools.visualization_interface.backends.pyvista.animation.FrameSequence,  
    fps: int = 30, loop: bool = False, scalar_bar_args: dict | None = None,  
    **plot_kwargs) →  
    ansys.tools.visualization_interface.backends.pyvista.animation.Animation
```

## Create an animation from a sequence of frames

This method creates an **Animation** object that can be used to visualize time-series simulation results, transient analyses, and dynamic phenomena.

## Parameters

## frames

[List[Any] or FrameSequence] Sequence of frame objects to animate. Can be a list of PyVista meshes, MeshObjectPlot objects, or a custom FrameSequence implementation for lazy loading.

fps

[int, optional] Frames per second for playback. Default is 30.

loop

[bool, optional] Whether to loop animation continuously. Default is False.

## scalar\_bar\_args

[`dict`, optional] Scalar bar arguments to apply to all frames (e.g., `clim` for fixed color scale). If not provided, a global color scale is calculated automatically.

**\*\*plot\_kwargs**

Additional keyword arguments passed to `add_mesh` for all frames (e.g., `cmap='viridis'`, `opacity=0.8`).

## Returns

## Animation

Animation controller object with playback controls.

**See also****Animation**

Animation controller class

**Examples**

Create and play an animation from transient simulation results:

```
>>> from ansys.tools.visualization_interface import Plotter
>>> plotter = Plotter(backend='pyvista')
>>> frames = [mesh1, mesh2, mesh3, mesh4] # Time series data
>>> animation = plotter.backend.create_animation(frames, fps=30, loop=True)
>>> animation.play()
>>> animation.show()
```

Export animation to video:

```
>>> animation = plotter.backend.create_animation(frames)
>>> animation.save("output.mp4", quality=8)
```

Use fixed color scale for accurate comparison:

```
>>> animation = plotter.backend.create_animation(
...     frames,
...     scalar_bar_args={"clim": (0.0, 1.0), "title": "Displacement [m]"}
... )
```

`PyVistaBackend.add_points(points: List | Any, color: str = 'red', size: float = 10.0, **kwargs) → pv.Actor`  
Add point markers to the scene.

**Parameters****points**

[Union[List, Any]] Points to add. Can be a list of coordinates or array-like object. Expected format: [[x1, y1, z1], [x2, y2, z2], ...] or Nx3 array.

**color**

[str, default: “red”] Color of the points.

**size**

[float, default: 10.0] Size of the point markers.

**\*\*kwargs**

[dict] Additional keyword arguments passed to PyVista’s add\_mesh method.

**Returns****pv.Actor**

PyVista actor representing the added points.

`PyVistaBackend.add_lines(points: List | Any, connections: List | Any | None = None, color: str = 'white', width: float = 1.0, **kwargs) → pv.Actor`

Add line segments to the scene.

**Parameters**

**points**

[Union[List, Any]] Points defining the lines. Can be a list of coordinates or array-like object. Expected format: [[x1, y1, z1], [x2, y2, z2], ...] or Nx3 array.

**connections**

[Optional[Union[List, Any]]], default: `None`] Line connectivity. If None, connects points sequentially. Expected format: [[start\_idx1, end\_idx1], [start\_idx2, end\_idx2], ...] or Mx2 array where M is the number of lines.

**color**

[str, default: “white”] Color of the lines.

**width**

[float, default: 1.0] Width of the lines.

**\*\*kwargs**

[dict] Additional keyword arguments passed to PyVista’s add\_mesh method.

**Returns****pv.Actor**

PyVista actor representing the added lines.

PyVistaBackend.add\_planes(*center*: tuple[float, float, float] = (0.0, 0.0, 0.0), *normal*: tuple[float, float, float] = (0.0, 0.0, 1.0), *i\_size*: float = 1.0, *j\_size*: float = 1.0, \*\*kwargs) → pv.Actor

Add a plane to the scene.

**Parameters****center**

[Tuple[float, float, float], default: (0.0, 0.0, 0.0)] Center point of the plane (x, y, z).

**normal**

[Tuple[float, float, float], default: (0.0, 0.0, 1.0)] Normal vector of the plane (x, y, z).

**i\_size**

[float, default: 1.0] Size of the plane in the i direction.

**j\_size**

[float, default: 1.0] Size of the plane in the j direction.

**\*\*kwargs**

[dict] Additional keyword arguments passed to PyVista’s add\_mesh method (e.g., color, opacity).

**Returns****pv.Actor**

PyVista actor representing the added plane.

PyVistaBackend.add\_text(*text*: str, *position*: tuple[float, float] | str, *font\_size*: int = 12, *color*: str = ‘white’, \*\*kwargs) → pv.Actor

Add text to the scene.

**Parameters****text**

[str] Text string to display.

**position**

[Union[Tuple[float, float], str]] Position for the text. Can be:

- 2D tuple (x, y) for screen coordinates (pixels from bottom-left)

- String position like ‘upper\_left’, ‘upper\_right’, ‘lower\_left’, ‘lower\_right’, ‘upper\_edge’, ‘lower\_edge’ (PyVista-specific)

**font\_size**

[`int`, default: 12] Font size for the text.

**color**

[`str`, default: “white”] Color of the text.

**\*\*kwargs**

[`dict`] Additional keyword arguments passed to PyVista’s add\_text method.

**Returns****`pv.Actor`**

PyVista actor representing the added text.

`PyVistaBackend.add_labels(points: List[Any], labels: List[str], font_size: int = 12, point_size: float = 5.0, **kwargs) → pv.Actor`

Add labels at 3D point locations.

**Parameters****points**

[Union[List, Any]] Points where labels should be placed. Can be a list of coordinates or array-like object. Expected format: [[x1, y1, z1], …] or Nx3 array.

**labels**

[List[str]] List of label strings to display at each point.

**font\_size**

[`int`, default: 12] Font size for the labels.

**point\_size**

[`float`, default: 5.0] Size of the point markers shown with labels.

**\*\*kwargs**

[`dict`] Additional keyword arguments passed to PyVista’s add\_point\_labels method.

**Returns****`pv.Actor`**

PyVista actor representing the added labels.

`PyVistaBackend.clear() → None`

Clear all actors from the scene and reset the plotter.

This method removes all previously added objects (meshes, points, lines, text, etc.) from the visualization scene by fully reinitializing the plotter.

## Description

Provides a wrapper to aid in plotting.

## Module detail

`pyvista.DARK_MODE_THRESHOLD = 120`

## The `pyvista_interface.py` module

### Summary

### Classes

<code>PyVistaInterface</code>	Provides the middle class between PyVista plotting operations and PyAnsys objects.
-------------------------------	--

#### `PyVistaInterface`

```
class ansys.tools.visualization_interface.backends.pyvista.pyvista_interface.PyVistaInterface(scene:  
    pyvista.Plotter  
    |  
    None  
    =  
    None,  
    color_opts:  
    Dict  
    |  
    None  
    =  
    None,  
    num_points:  
    int  
    =  
    100,  
    en-  
    able_widget:  
    bool  
    =  
    True,  
    show_plane:  
    bool  
    =  
    False,  
    use_qt:  
    bool  
    =  
    False,  
    show_qt:  
    bool  
    =  
    True,  
    **plot-  
    ter_kwargs)
```

Provides the middle class between PyVista plotting operations and PyAnsys objects.

The main purpose of this class is to simplify interaction between PyVista and the PyVista backend provided. This class is responsible for creating the PyVista scene and adding the PyAnsys objects to it.

#### Parameters

##### `scene`

[`Plotter`, default: `None`] Scene for rendering the objects. If passed, `off_screen` needs to

be set manually beforehand for documentation and testing.

**color\_opts**

[`dict`, default: `None`] Dictionary containing the background and top colors.

**num\_points**

[`int`, default: 100] Number of points to use to render the shapes.

**enable\_widgets**

[`bool`, default: `True`] Whether to enable widget buttons in the plotter window. Widget buttons must be disabled when using `trame` for visualization.

**show\_plane**

[`bool`, default: `False`] Whether to show the XY plane in the plotter window.

**use\_qt**

[`bool`, default: `False`] Whether to use the Qt backend for the plotter window.

**show\_qt**

[`bool`, default: `True`] Whether to show the Qt plotter window.

## Overview

### Methods

<code>view_xy</code>	View the scene from the XY plane.
<code>view_xz</code>	View the scene from the XZ plane.
<code>view_yx</code>	View the scene from the YX plane.
<code>view_yz</code>	View the scene from the YZ plane.
<code>view_zx</code>	View the scene from the ZX plane.
<code>view_zy</code>	View the scene from the ZY plane.
<code>enable_parallel_projection</code>	Enable parallel projection for the camera.
<code>disable_parallel_projection</code>	Disable parallel projection for the camera.
<code>clip</code>	Clip a given mesh with a plane.
<code>plot_meshobject</code>	Plot a generic <code>MeshObjectPlot</code> object to the scene.
<code>plot_edges</code>	Plot the outer edges of an object to the plot.
<code>hide_children</code>	Hide all the children of a given <code>MeshObjectPlot</code> object.
<code>show_children</code>	Show all the children of a given <code>MeshObjectPlot</code> object.
<code>toggle_subtree_visibility</code>	Toggle visibility of an object and its entire subtree.
<code>plot</code>	Plot any type of object to the scene.
<code>plot_iter</code>	Plot elements of an iterable of any type of objects to the scene.
<code>show</code>	Show the rendered scene on the screen.
<code>set_add_mesh_defaults</code>	Set the default values for the plotting options.

### Properties

<code>scene</code>	Rendered scene object.
<code>object_to_actors_map</code>	Mapping between the PyVista actor and the PyAnsys objects.

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.pyvista_interface import_
PyVistaInterface
```

## Property detail

**property** `PyVistaInterface.scene: pyvista.plotting.plotter.Plotter`

Rendered scene object.

### Returns

`Plotter`

Rendered scene object.

**property** `PyVistaInterface.object_to_actors_map: Dict[pyvista.Actor, ansys.tools.visualization_interface.types.mesh_object_plot.MeshObjectPlot]`

Mapping between the PyVista actor and the PyAnsys objects.

## Method detail

`PyVistaInterface.view_xy() → None`

View the scene from the XY plane.

`PyVistaInterface.view_xz() → None`

View the scene from the XZ plane.

`PyVistaInterface.view_yx() → None`

View the scene from the YX plane.

`PyVistaInterface.view_yz() → None`

View the scene from the YZ plane.

`PyVistaInterface.view_zx() → None`

View the scene from the ZX plane.

`PyVistaInterface.view_zy() → None`

View the scene from the ZY plane.

`PyVistaInterface.enable_parallel_projection() → None`

Enable parallel projection for the camera.

`PyVistaInterface.disable_parallel_projection() → None`

Disable parallel projection for the camera.

`PyVistaInterface.clip(mesh: pyvista.PolyData | pyvista.MultiBlock | pyvista.UnstructuredGrid, plane: ansys.tools.visualization_interface.utils.clip_plane.ClipPlane) → pyvista.PolyData | pyvista.MultiBlock`

Clip a given mesh with a plane.

### Parameters

**mesh**

`[Union[pv.PolyData, pv.MultiBlock]]` Mesh.

**normal**

`[str, default: "x"]` Plane to use for clipping. Options are "x", "-x", "y", "-y", "z", and "-z".

**origin**

`[tuple, default: None]` Origin point of the plane.

**plane**

`[ClipPlane, default: None]` Clipping plane to cut the mesh with.

**Returns**

**Union[pv.PolyData,pv.MultiBlock]**  
Clipped mesh.

**PyVistaInterface.plot\_meshobject**(*custom\_object*: an-  
sys.tools.visualization\_interface.types.mesh\_object\_plot.MeshObjectPlot,  
*plot\_children*: *bool* = *True*, *\*\*plotting\_options*)

Plot a generic MeshObjectPlot object to the scene.

**Parameters**

**plottable\_object**  
[MeshObjectPlot] Object to add to the scene.

**plot\_children**  
[*bool*, default: *True*] Whether to plot the children of the object.

**\*\*plotting\_options**  
[*dict*, default: *None*] Keyword arguments. For allowable keyword arguments, see the  
`Plotter.add_mesh` method.

**PyVistaInterface.plot\_edges**(*custom\_object*:  
anys.tools.visualization\_interface.types.mesh\_object\_plot.MeshObjectPlot,  
*\*\*plotting\_options*) → *None*

Plot the outer edges of an object to the plot.

This method has the side effect of adding the edges to the MeshObjectPlot object that you pass through the parameters.

**Parameters**

**custom\_object**  
[MeshObjectPlot] Custom object with the edges to add.

**\*\*plotting\_options**  
[*dict*, default: *None*] Keyword arguments. For allowable keyword arguments, see the  
`Plotter.add_mesh` method.

**PyVistaInterface.hide\_children**(*custom\_object*: an-  
sys.tools.visualization\_interface.types.mesh\_object\_plot.MeshObjectPlot)  
→ *None*

Hide all the children of a given MeshObjectPlot object.

**Parameters**

**custom\_object**  
[MeshObjectPlot] Custom object whose children will be hidden.

**PyVistaInterface.show\_children**(*custom\_object*: an-  
sys.tools.visualization\_interface.types.mesh\_object\_plot.MeshObjectPlot)  
→ *None*

Show all the children of a given MeshObjectPlot object.

**Parameters**

**custom\_object**  
[MeshObjectPlot] Custom object whose children will be shown.

**PyVistaInterface.toggle\_subtree\_visibility**(*custom\_object*: an-  
sys.tools.visualization\_interface.types.mesh\_object\_plot.MeshObjectPlot,  
*include\_root*: *bool* = *True*) → *None*

Toggle visibility of an object and its entire subtree.

This method toggles the visibility state of the given object and all its descendants. If the object is currently visible, it and all children will be hidden. If hidden, they will all be shown.

#### Parameters

**custom\_object**

[MeshObjectPlot] Root object of the subtree to toggle.

**include\_root**

[bool, default: `True`] Whether to toggle the root object's visibility. If `False`, only children are toggled.

#### Examples

Toggle visibility of picked object and its children:

```
>>> picked = plotter.pick()
>>> if picked:
...     backend.toggle_subtree_visibility(picked[0])
```

Toggle only children, keeping parent visible:

```
>>> backend.toggle_subtree_visibility(obj, include_root=False)
```

`PyVistaInterface.plot(plottable_object: pv.PolyData | pv.UnstructuredGrid |  
ansys.tools.visualization_interface.types.mesh_object_plot.MeshObjectPlot |  
pv.UnstructuredGrid, name_filter: str = None, plot_children: bool = False,  
**plotting_options) → None`

Plot any type of object to the scene.

Supported object types are `List[pv.PolyData]`, `MeshObjectPlot`, and `pv.MultiBlock`.

#### Parameters

**plottable\_object**

[Union[pv.PolyData, pv.MultiBlock, MeshObjectPlot, pv.UnstructuredGrid, pv.StructuredGrid]] Object to plot.

**name\_filter**

[str, default: `None`] Regular expression with the desired name or names to include in the plotter.

**\*\*plotting\_options**

[dict, default: `None`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

`PyVistaInterface.plot_iter(plotting_list: List[Any], name_filter: str = None, **plotting_options) → None`

Plot elements of an iterable of any type of objects to the scene.

#### Parameters

**plotting\_list**

[List[Any]] List of objects to plot.

**name\_filter**

[str, default: `None`] Regular expression with the desired name or names to include in the plotter.

**\*\*plotting\_options**

[`dict`, default: `None`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

`PyVistaInterface.show(show_plane: bool = False, jupyter_backend: str | None = None, **kwargs: Dict | None) → None`

Show the rendered scene on the screen.

**Parameters****show\_plane**

[`bool`, default: `True`] Whether to show the XY plane.

**jupyter\_backend**

[`str`, default: `None`] PyVista Jupyter backend.

**\*\*kwargs**

[`dict`, default: `None`] Plotting and show keyword arguments. For allowable keyword arguments, see the `Plotter.show` and `Plotter.show` methods.

**Notes**

For more information on supported Jupyter backends, see [Jupyter Notebook Plotting](#) in the PyVista documentation.

`PyVistaInterface.set_add_mesh_defaults(plotting_options: Dict | None) → None`

Set the default values for the plotting options.

**Parameters****plotting\_options**

[`Optional[Dict]`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

**Description**

Provides plotting for various PyAnsys objects.

**The `trame_local.py` module****Summary****Classes**

<code>TrameVisualizer</code>	Defines the trame layout view.
------------------------------	--------------------------------

**Constants**

<code>CLIENT_TYPE</code>
--------------------------

**TrameVisualizer**

`class ansys.tools.visualization_interface.backends.pyvista.trame_local.TrameVisualizer`

Defines the trame layout view.

## Overview

## Methods

<code>set_scene</code>	Set the trame layout view and the mesh to show through the PyVista plotter.
<code>show</code>	Start the trame server and show the mesh.

## Attributes

<code>server</code>
<code>plotter</code>

## Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.trame_local import_
    TrameVisualizer
```

## Attribute detail

`TrameVisualizer.server`

`TrameVisualizer.plotter = None`

## Method detail

`TrameVisualizer.set_scene(plotter)`

Set the trame layout view and the mesh to show through the PyVista plotter.

### Parameters

`plotter`  
[Plotter] PyVista plotter with the rendered mesh.

`TrameVisualizer.show()`

Start the trame server and show the mesh.

## Description

Provides `trame` visualizer interface for visualization.

## Module detail

```
trame_local.CLIENT_TYPE = 'vue2'
```

## The `trame_remote.py` module

### Summary

### Functions

<code>send_pl</code>	Send the plotter meshes to a remote trame service.
<code>send_mesh</code>	Send a mesh to a remote trame service.

## Description

Module for trame websocket client functions.

## Module detail

`trame_remote.send_pl(plotter: pyvista.Plotter, host: str = 'localhost', port: int = 8765)`

Send the plotter meshes to a remote trame service.

Since plotter can't be pickled, we send the meshes list instead.

### Parameters

#### plotter

[pv.Plotter] Plotter to send.

#### host

[str, optional] Websocket host to connect to, by default “localhost”.

#### port

[int, optional] Websocket port to connect to, by default 8765.

`trame_remote.send_mesh(mesh: pyvista.PolyData | pyvista.MultiBlock, host: str = 'localhost', port: int = 8765)`

Send a mesh to a remote trame service.

### Parameters

#### mesh

[Union[pv.PolyData, pv.MultiBlock]] Mesh to send.

#### host

[str, optional] Websocket host to connect to, by default “localhost”.

#### port

[int, optional] Websocket port to connect to, by default 8765.

## The `trame_service.py` module

### Summary

### Classes

<code>TrameService</code>	Trame service class.
---------------------------	----------------------

### TrameService

```
class ansys.tools.visualization_interface.backends.pyvista.trame_service.TrameService(websocket_host:
    str
    =
    'lo-
    cal-
    host',
    web-
    socket_port:
    int
    =
    8765)
```

Trame service class.

Initializes a trame service where you can send meshes to plot in a trame webview plotter.

#### Parameters

##### websocket\_host

[`str`, optional] Host where the webserver will listen for new plotters and meshes, by default “localhost”.

##### websocket\_port

[`int`, optional] Port where the webserver will listen for new plotters and meshes, by default 8765.

## Overview

### Methods

<code>clear_plotter</code>	Clears the web view in the service.
<code>set_scene</code>	Sets the web view scene for the trame service.
<code>run</code>	Start the trame web view and the websocket services.

### Import detail

```
from ansys.tools.visualization_interface.backends.pyvista.trame_service import_
    TrameService
```

### Method detail

#### TrameService.`clear_plotter()`

Clears the web view in the service.

#### TrameService.`set_scene()`

Sets the web view scene for the trame service.

#### TrameService.`run()`

Start the trame web view and the websocket services.

### Description

Trame service module.

### Description

Provides interfaces.

### Description

Provides interfaces.

## The types package

### Summary

### Submodules

<code>edge_plot</code>	Provides the edge type for plotting.
<code>mesh_object_plot</code>	Provides the MeshObjectPlot class.

## The edge\_plot.py module

### Summary

### Classes

<code>EdgePlot</code>	Provides the mapper class for relating PyAnsys object edges with its PyVista actor.
-----------------------	---

#### EdgePlot

```
class ansys.tools.visualization_interface.types.edge_plot.EdgePlot(actor: pyvista.Actor | plotly.graph_objects.Mesh3d, edge_object: Any, parent: Any = None)
```

Provides the mapper class for relating PyAnsys object edges with its PyVista actor.

#### Parameters

**actor**  
[Union[Actor, Mesh3d]] PyVista actor that represents the edge.

**edge\_object**  
[Edge] PyAnsys object edge that is represented by the PyVista actor.

**parent**  
[MeshObjectPlot, default: `None`] Parent PyAnsys object of the edge.

### Overview

### Properties

<code>actor</code>	PyVista actor of the object.
<code>edge_object</code>	PyAnsys edge.
<code>parent</code>	Parent PyAnsys object of the edge.
<code>name</code>	Name of the edge.

### Import detail

```
from ansys.tools.visualization_interface.types.edge_plot import EdgePlot
```

## Property detail

**property EdgePlot.actor: `pyvista.Actor`**

PyVista actor of the object.

### Returns

**Actor**

PyVista actor.

**property EdgePlot.edge\_object: `Any`**

PyAnsys edge.

### Returns

**Any**

PyAnsys edge.

**property EdgePlot.parent: `Any`**

Parent PyAnsys object of the edge.

### Returns

**Any**

Parent PyAnsys object.

**property EdgePlot.name: `str`**

Name of the edge.

### Returns

**str**

Name of the edge.

## Description

Provides the edge type for plotting.

## The `mesh_object_plot.py` module

### Summary

### Classes

---

**`MeshObjectPlot`** Relates a custom object with a mesh, provided by the consumer library.

---

### MeshObjectPlot

```
class ansys.tools.visualization_interface.types.mesh_object_plot.MeshObjectPlot(custom_object:
    Any, mesh:
    pyvista.PolyData
    |
    pyvista.MultiBlock
    |
    plotly.graph_objects.Mesh3d,
    actor:
    pyvista.Actor
    = None,
    edges:
    List[ansys.tools.visualization_...
    = None,
    children:
    List[MeshObjectPlot]
    = None,
    parent:
    MeshOb-
    jectPlot =
    None)
```

Relates a custom object with a mesh, provided by the consumer library.

## Overview

### Methods

<code>add_child</code>	Set a child MeshObjectPlot to the current object.
<code>is_visible_in_tree</code>	Check if this object is visible considering parent visibility.

### Properties

<code>parent</code>	Get the parent MeshObjectPlot of the current object.
<code>mesh</code>	Mesh of the object in PyVista format.
<code>custom_object</code>	Custom object.
<code>actor</code>	PyVista actor of the object in the plotter.
<code>edges</code>	Edges of the object.
<code>name</code>	Name of the object.
<code>mesh_type</code>	Type of the mesh.
<code>visible</code>	Whether this object is currently visible.

### Import detail

```
from ansys.tools.visualization_interface.types.mesh_object_plot import MeshObjectPlot
```

### Property detail

`property MeshObjectPlot.parent: MeshObjectPlot`

Get the parent MeshObjectPlot of the current object.

This method is used to set a parent MeshObjectPlot to the current object. It is useful when the custom object has a hierarchical structure, and the consumer library wants to relate the parent objects with their meshes.

**Parameters****parent**

[MeshObjectPlot] Parent MeshObjectPlot to be set.

**property** MeshObjectPlot.mesh: `pyvista.PolyData` | `pyvista.MultiBlock` | `plotly.graph_objects.Mesh3d`

Mesh of the object in PyVista format.

**Returns****Union[pv.PolyData, pv.MultiBlock]**

Mesh of the object.

**property** MeshObjectPlot.custom\_object: Any

Custom object.

**Returns****Any**

Custom object.

**property** MeshObjectPlot.actor: `pyvista.Actor`

PyVista actor of the object in the plotter.

**Returns****pv.Actor**

PyVista actor of the object.

**property** MeshObjectPlot.edges:

`List[ansys.tools.visualization_interface.types.edge_plot.EdgePlot]`

Edges of the object.

**Returns****List[EdgePlot]**

Edges of the object.

**property** MeshObjectPlot.name: str

Name of the object.

**Returns****str**

Name of the object.

**property** MeshObjectPlot.mesh\_type: Type

Type of the mesh.

**Returns****type**

Type of the mesh.

**property** MeshObjectPlot.visible: bool

Whether this object is currently visible.

This property reflects the visibility state of the object. If an actor is assigned, it reads the visibility from the actor to stay synchronized.

**Returns**

**bool**

True if the object is visible, False otherwise.

**Method detail****MeshObjectPlot.add\_child(*child*: MeshObjectPlot)**

Set a child MeshObjectPlot to the current object.

This method is used to set a child MeshObjectPlot to the current object. It is useful when the custom object has a hierarchical structure, and the consumer library wants to relate the child objects with their meshes.

**Parameters****child**

[MeshObjectPlot] Child MeshObjectPlot to be set.

**MeshObjectPlot.is\_visible\_in\_tree() → bool**

Check if this object is visible considering parent visibility.

An object is only truly visible if both itself and all its ancestors in the tree are visible.

**Returns****bool**

True if object and all ancestors are visible, False otherwise.

**Description**

Provides the MeshObjectPlot class.

**Description**

Provides custom types.

**The utils package****Summary****Submodules**

<i>clip_plane</i>	Provides the ClipPlane class.
<i>color</i>	Provides an enum with the color to use for the plotter actors.
<i>logger</i>	Provides the singleton helper class for the logger.
<i>vtkhdf_converter</i>	Utils module for VTKHDF management.

**The clip\_plane.py module****Summary****Classes**

<i>ClipPlane</i>	Provides the clipping plane for clipping meshes in the plotter.
------------------	---

## ClipPlane

```
class ansys.tools.visualization_interface.utils.clip_plane.ClipPlane(normal: Tuple[float, float, float] = (1, 0, 0), origin: Tuple[float, float, float] = (0, 0, 0))
```

Provides the clipping plane for clipping meshes in the plotter.

The clipping plane is defined by both normal and origin vectors.

### Parameters

#### normal

[Tuple[float, float, float], default: (1, 0, 0)] Normal of the plane.

#### origin

[Tuple[float, float, float], default: (0, 0, 0)] Origin point of the plane.

## Overview

## Properties

<code>normal</code>	Normal of the plane.
<code>origin</code>	Origin of the plane.

## Import detail

```
from ansys.tools.visualization_interface.utils.clip_plane import ClipPlane
```

## Property detail

`property ClipPlane.normal: Tuple[float, float, float]`

Normal of the plane.

### Returns

`Tuple[float, float, float]`

Normal of the plane.

`property ClipPlane.origin: Tuple[float, float, float]`

Origin of the plane.

### Returns

`Tuple[float, float, float]`

Origin of the plane.

## Description

Provides the ClipPlane class.

## The color.py module

## Summary

## Enums

<i>Color</i>	Provides an enum with the color to use for the plotter actors.
--------------	--

### Color

`class ansys.tools.visualization_interface.utils.color.Color`

Bases: `enum.Enum`

Provides an enum with the color to use for the plotter actors.

### Overview

### Attributes

<code>DEFAULT</code>	Default color for the plotter actors.
<code>PICKED</code>	Color for the actors that are currently picked.
<code>EDGE</code>	Default color for the edges.
<code>PICKED_EDGE</code>	Color for the edges that are currently picked.

### Import detail

<code>from ansys.tools.visualization_interface.utils.color import Color</code>
--

### Attribute detail

`Color.DEFAULT = '#D6F7D1'`

Default color for the plotter actors.

`Color.PICKED = '#BB6EEE'`

Color for the actors that are currently picked.

`Color.EDGE = '#000000'`

Default color for the edges.

`Color.PICKED_EDGE = '#9C9C9C'`

Color for the edges that are currently picked.

### Description

Provides an enum with the color to use for the plotter actors.

### The logger.py module

#### Summary

#### Classes

<i>SingletonType</i>	Provides the singleton helper class for the logger.
<i>VizLogger</i>	Provides the singleton logger for the visualizer.

## Attributes

`logger`

### SingletonType

`class ansys.tools.visualization_interface.utils.logger.SingletonType`

Bases: `type`

Provides the singleton helper class for the logger.

## Overview

### Special methods

`__call__` Call to redirect new instances to the singleton instance.

### Import detail

```
from ansys.tools.visualization_interface.utils.logger import SingletonType
```

### Method detail

`SingletonType.__call__(*args, **kwargs)`

Call to redirect new instances to the singleton instance.

### VizLogger

`class ansys.tools.visualization_interface.utils.logger.VizLogger(level: int = logging.ERROR, logger_name: str = 'VizLogger')`

Bases: `object`

Provides the singleton logger for the visualizer.

#### Parameters

`to_file`

[`bool`, default: `False`] Whether to include the logs in a file.

## Overview

### Methods

<code>get_logger</code>	Get the logger.
<code>set_level</code>	Set the logger output level.
<code>enable_output</code>	Enable logger output to a given stream.
<code>add_file_handler</code>	Save logs to a file in addition to printing them to the standard output.

## Import detail

```
from ansys.tools.visualization_interface.utils.logger import VizLogger
```

## Method detail

`VizLogger.get_logger()`

Get the logger.

### Returns

`Logger`

Logger.

`VizLogger.set_level(level: int)`

Set the logger output level.

### Parameters

`level`

[`int`] Output Level of the logger.

`VizLogger.enable_output(stream=None)`

Enable logger output to a given stream.

If a stream is not specified, `sys.stderr` is used.

### Parameters

`stream: TextIO, default: ``sys.stderr```

Stream to output the log output to.

`VizLogger.add_file_handler(logs_dir: str = './log')`

Save logs to a file in addition to printing them to the standard output.

### Parameters

`logs_dir`

[`str`, default: "./log"] Directory of the logs.

## Description

Provides the singleton helper class for the logger.

## Module detail

`logger.logger`

### The `vtkhdf_converter.py` module

#### Summary

#### Functions

<code>pd_to_vtkhdf</code>	Write the PyVista PolyData directly to a VTKHDF file.
<code>vtkhdf_to_pd</code>	Read a VTKHDF file and convert it to PyVista PolyData.
<code>ug_to_vtkhdf</code>	Write the PyVista UnstructuredGrid directly to a VTKHDF file.
<code>vtkhdf_to_ug</code>	Read a VTKHDF file and convert it to PyVista UnstructuredGrid.

## Description

Utils module for VTKHDF management.

## Module detail

`vtkhdf_converter.pd_to_vtkhdf(pd: pyvista.PolyData, output_vtkhdf_file: str | pathlib.Path) → pathlib.Path`

Write the PyVista PolyData directly to a VTKHDF file.

### Parameters

#### `pd`

[`pv.PolyData`] The PyVista PolyData to be written.

#### `output_vtkhdf_file`

[Union[`str`, `Path`]] The output VTKHDF file path.

### Returns

#### `Path`

The path to the saved VTKHDF file.

`vtkhdf_converter.vtkhdf_to_pd(input_vtkhdf_file: str | pathlib.Path) → pyvista.PolyData`

Read a VTKHDF file and convert it to PyVista PolyData.

### Parameters

#### `input_vtkhdf_file`

[Union[`str`, `Path`]] The input VTKHDF file path.

### Returns

#### `pv.PolyData`

The converted PyVista PolyData.

`vtkhdf_converter.ug_to_vtkhdf(ug: pyvista.UnstructuredGrid, output_vtkhdf_file: str | pathlib.Path) → pathlib.Path`

Write the PyVista UnstructuredGrid directly to a VTKHDF file.

### Parameters

#### `ug`

[`pv.UnstructuredGrid`] The PyVista UnstructuredGrid to be written.

#### `output_vtkhdf_file`

[Union[`str`, `Path`]] The output VTKHDF file path.

### Returns

#### `Path`

The path to the saved VTKHDF file.

`vtkhdf_converter.vtkhdf_to_ug(input_vtkhdf_file: str | pathlib.Path) → pyvista.UnstructuredGrid`

Read a VTKHDF file and convert it to PyVista UnstructuredGrid.

### Parameters

#### `input_vtkhdf_file`

[Union[`str`, `Path`]] The input VTKHDF file path.

### Returns

#### `pv.UnstructuredGrid`

The converted PyVista UnstructuredGrid.

## Description

Provides the Utils package.

## The `plotter.py` module

### Summary

### Classes

<code>Plotter</code>	Base plotting class containing common methods and attributes.
----------------------	---

#### Plotter

```
class ansys.tools.visualization_interface.plotter.Plotter(backend: an-
    sys.tools.visualization_interface.backends._base.BaseBackend
    =None)
```

Base plotting class containing common methods and attributes.

This class is responsible for plotting objects using the specified backend.

#### Parameters

##### `backend`

[`BaseBackend`, optional] Plotting backend to use, by default `PyVistaBackend`.

### Overview

### Methods

<code>plot_iter</code>	Plots multiple objects using the specified backend.
<code>plot</code>	Plots an object using the specified backend.
<code>show</code>	Show the plotted objects.
<code>animate</code>	Create an animation from a sequence of frames.
<code>add_points</code>	Add point markers to the scene.
<code>add_lines</code>	Add line segments to the scene.
<code>add_planes</code>	Add a plane to the scene.
<code>add_text</code>	Add text to the scene.
<code>add_labels</code>	Add labels at 3D point locations.
<code>clear</code>	Clear all actors from the scene.

### Properties

<code>backend</code>	Return the base plotter object.
----------------------	---------------------------------

### Import detail

```
from ansys.tools.visualization_interface.plotter import Plotter
```

## Property detail

### `property Plotter.backend`

Return the base plotter object.

## Method detail

### `Plotter.plot_iter(plotting_list: List, **plotting_options)`

Plots multiple objects using the specified backend.

#### Parameters

##### `plotting_list`

[List] List of objects to plot.

##### `plotting_options`

[dict] Additional plotting options.

### `Plotter.plot(plottable_object: Any, **plotting_options)`

Plots an object using the specified backend.

#### Parameters

##### `plottable_object`

[Any] Object to plot.

##### `plotting_options`

[dict] Additional plotting options.

### `Plotter.show(plottable_object: Any = None, screenshot: str = None, name_filter: bool = None, **kwargs) → List`

Show the plotted objects.

#### Parameters

##### `plottable_object`

[Any, optional] Object to show, by default None.

##### `screenshot`

[str, optional] Path to save a screenshot, by default None.

##### `name_filter`

[bool, optional] Flag to filter the object, by default None.

##### `kwargs`

[dict] Additional options the selected backend accepts.

#### Returns

##### `List`

List of picked objects.

### `Plotter.animate(frames: List[Any], fps: int = 30, loop: bool = False, scalar_bar_args: dict | None = None, **plot_kwargs)`

Create an animation from a sequence of frames.

This method provides a convenient way to create animations from time-series simulation results, transient analyses, and dynamic phenomena. It wraps the backend's animation functionality in a simple, consistent API.

#### Parameters

##### `frames`

[List[Any]] Sequence of frame objects to animate. Can be PyVista meshes, MeshObjectPlot objects, or any plottable objects.

**fps**

[int, optional] Frames per second for playback. Default is 30.

**loop**

[bool, optional] Whether to loop animation continuously. Default is False.

**scalar\_bar\_args**

[dict, optional] Scalar bar arguments to apply to all frames (e.g., `clim` for fixed color scale). If not provided, a global color scale is calculated automatically to ensure visual integrity across frames.

**\*\*plot\_kwargs**

Additional keyword arguments passed to `add_mesh` for all frames (e.g., `cmap='viridis'`, `opacity=0.8`).

**Returns****Animation**

Animation controller object with playback controls:  
- `play()`: Start animation  
- `pause()`: Pause animation  
- `stop()`: Stop and reset to first frame  
- `step_forward()`: Advance one frame  
- `step_backward()`: Rewind one frame  
- `seek(frame_index)`: Jump to specific frame  
- `save(filename)`: Export to video (MP4, GIF, AVI)  
- `show()`: Display with plotter

**Raises****ValueError**

If frames list is empty or fps is not positive.

**NotImplementedError**

If the backend does not support animations.

**See also****Animation**

Animation controller class with detailed playback controls

**Notes**

- Fixed color scales are recommended (and calculated by default) to ensure visual integrity and prevent misleading animations where color meanings change between frames.
- For large datasets (1000+ frames or >5M cells), consider implementing a custom `FrameSequence` with lazy loading capabilities.
- The animation uses the backend's native capabilities. Currently, only PyVista backend supports animations.

**Examples**

Create and play a simple animation from transient simulation results:

```
>>> from ansys.tools.visualization_interface import Plotter
>>> import pyvista as pv
>>> # Create example meshes representing time steps
>>> sphere = pv.Sphere()
>>> frames = []
>>> for i in range(20):
...     mesh = sphere.copy()
```

(continues on next page)

(continued from previous page)

```
...     mesh["displacement"] = np.random.rand(mesh.n_points) * i * 0.1
...
frames.append(mesh)
>>> plotter = Plotter()
>>> animation = plotter.animate(frames, fps=10, loop=True)
>>> animation.show()
```

Export animation to video:

```
>>> animation = plotter.animate(frames, fps=30)
>>> animation.save("simulation.mp4", quality=8)
```

Use fixed color scale for accurate comparison:

```
>>> animation = plotter.animate(
...     frames,
...     fps=30,
...     scalar_bar_args={"clim": (0.0, 1.0), "title": "Displacement [m]"}
... )
>>> animation.play()
>>> animation.show()
```

Control playback programmatically:

```
>>> animation = plotter.animate(frames)
>>> animation.play() # Start animation
>>> # ... after some time ...
>>> animation.pause() # Pause
>>> animation.step_forward() # Advance one frame
>>> animation.seek(10) # Jump to frame 10
>>> animation.stop() # Reset to beginning
```

`Plotter.add_points(points: List[Any], color: str = 'red', size: float = 10.0, **kwargs) → Any`

Add point markers to the scene.

This method provides a backend-agnostic way to add point markers to the visualization scene. The points will be rendered using the active backend's native point rendering capabilities.

### Parameters

#### points

[Union[List, Any]] Points to add. Can be a list of coordinates or array-like object. Expected format: [[x1, y1, z1], [x2, y2, z2], ...] or Nx3 array.

#### color

[str, default: “red”] Color of the points. Can be a color name (e.g., ‘red’, ‘blue’) or hex color code (e.g., ‘#FF0000’).

#### size

[float, default: 10.0] Size of the point markers in pixels or display units (interpretation depends on backend).

#### \*\*kwargs

[dict] Additional backend-specific keyword arguments for advanced customization.

### Returns

**Any**

Backend-specific actor or object representing the added points. Can be used for further manipulation or removal.

**Examples**

Add simple point markers at three locations:

```
>>> from ansys.tools.visualization_interface import Plotter
>>> plotter = Plotter()
>>> points = [[0, 0, 0], [1, 0, 0], [0, 1, 0]]
>>> plotter.add_points(points, color='blue', size=15)
>>> plotter.show()
```

Add points with custom styling:

```
>>> import numpy as np
>>> points = np.random.rand(100, 3)
>>> plotter.add_points(points, color='yellow', size=8)
>>> plotter.show()
```

`Plotter.add_lines(points: List | Any, connections: List | Any | None = None, color: str = 'white', width: float = 1.0, **kwargs) → Any`

Add line segments to the scene.

This method provides a backend-agnostic way to add lines to the visualization scene. Lines can connect points sequentially or based on explicit connectivity information.

**Parameters****points**

[Union[List, Any]] Points defining the lines. Can be a list of coordinates or array-like object. Expected format: [[x1, y1, z1], [x2, y2, z2], ...] or Nx3 array.

**connections**

[Optional[Union[List, Any]], default: `None`] Line connectivity. If `None`, connects points sequentially (0->1, 1->2, 2->3, ...). If provided, should define line segments as pairs of point indices: [[start\_idx1, end\_idx1], [start\_idx2, end\_idx2], ...] or Mx2 array where M is the number of line segments.

**color**

[str, default: “white”] Color of the lines. Can be a color name or hex color code.

**width**

[float, default: 1.0] Width of the lines in pixels or display units (interpretation depends on backend).

**\*\*kwargs**

[dict] Additional backend-specific keyword arguments for advanced customization.

**Returns****Any**

Backend-specific actor or object representing the added lines. Can be used for further manipulation or removal.

## Examples

Add a line connecting points sequentially:

```
>>> from ansys.tools.visualization_interface import Plotter
>>> plotter = Plotter()
>>> points = [[0, 0, 0], [1, 0, 0], [1, 1, 0], [0, 1, 0]]
>>> plotter.add_lines(points, color='green', width=2.0)
>>> plotter.show()
```

Add specific line segments with explicit connectivity:

```
>>> points = [[0, 0, 0], [1, 0, 0], [0, 1, 0], [1, 1, 0]]
>>> connections = [[0, 1], [2, 3], [0, 2]] # Connect specific pairs
>>> plotter.add_lines(points, connections=connections, color='red', width=3.0)
>>> plotter.show()
```

`Plotter.add_planes(center: Tuple[float, float, float] = (0.0, 0.0, 0.0), normal: Tuple[float, float, float] = (0.0, 0.0, 1.0), i_size: float = 1.0, j_size: float = 1.0, **kwargs) → Any`

Add a plane to the scene.

This method provides a backend-agnostic way to add plane objects to the visualization scene. Planes are useful for showing reference planes, symmetry planes, or cutting planes.

### Parameters

#### `center`

[`Tuple[float, float, float]`, default: (0.0, 0.0, 0.0)] Center point of the plane in 3D space (x, y, z).

#### `normal`

[`Tuple[float, float, float]`, default: (0.0, 0.0, 1.0)] Normal vector of the plane (x, y, z). The vector will be normalized by the backend if needed.

#### `i_size`

[`float`, default: 1.0] Size of the plane in the i direction (local coordinate system).

#### `j_size`

[`float`, default: 1.0] Size of the plane in the j direction (local coordinate system).

#### `**kwargs`

[`dict`] Additional backend-specific keyword arguments for advanced customization (e.g., color, opacity, resolution).

### Returns

#### `Any`

Backend-specific actor or object representing the added plane. Can be used for further manipulation or removal.

## Examples

Add a horizontal plane at z=0:

```
>>> from ansys.tools.visualization_interface import Plotter
>>> plotter = Plotter()
>>> plotter.add_planes(center=(0, 0, 0), normal=(0, 0, 1), i_size=2.0, j_size=2.0)
>>> plotter.show()
```

Add a vertical plane with custom styling:

```
>>> plotter.add_planes(
...     center=(1, 0, 0),
...     normal=(1, 0, 0),
...     i_size=3.0,
...     j_size=3.0,
...     color='lightblue',
...     opacity=0.5
... )
>>> plotter.show()
```

`Plotter.add_text(text: str, position: Tuple[float, float] | str, font_size: int = 12, color: str = 'white', **kwargs)`  
→ Any

Add text to the scene.

This method provides a backend-agnostic way to add text labels to the visualization scene. Text is positioned using 2D screen coordinates.

#### Parameters

##### `text`

[`str`] Text string to display.

##### `position`

[`Union[Tuple[float, float], str]`] Position for the text. Can be:

- 2D tuple (x, y) for screen/viewport coordinates (pixels from bottom-left)
- String position like ‘upper\_left’, ‘upper\_right’, ‘lower\_left’, ‘lower\_right’, ‘upper\_edge’, ‘lower\_edge’ (backend-dependent support)

##### `font_size`

[`int`, default: 12] Font size for the text in points.

##### `color`

[`str`, default: “white”] Color of the text. Can be a color name or hex color code.

##### `**kwargs`

[`dict`] Additional backend-specific keyword arguments for advanced customization (e.g., font\_family, bold, italic, shadow).

#### Returns

##### `Any`

Backend-specific actor or object representing the added text. Can be used for further manipulation or removal.

## Examples

Add text at a screen position:

```
>>> from ansys.tools.visualization_interface import Plotter
>>> plotter = Plotter()
>>> plotter.add_text("Title", position=(10, 10), font_size=18, color='yellow')
>>> plotter.show()
```

Add text using a named position:

```
>>> plotter.add_text(  
...     "Corner Label",  
...     position='upper_right',  
...     font_size=14,  
...     color='red'  
... )  
>>> plotter.show()
```

`Plotter.add_labels(points: List[Any], labels: List[str], font_size: int = 12, point_size: float = 5.0, **kwargs)`  
→ Any

Add labels at 3D point locations.

This method provides a backend-agnostic way to add text labels at specific 3D coordinates in the visualization scene. Labels are displayed next to marker points.

#### Parameters

##### points

[Union[List, Any]] Points where labels should be placed. Can be a list of coordinates or array-like object. Expected format: [[x1, y1, z1], ...] or Nx3 array.

##### labels

[List[str]] List of label strings to display at each point. Must have the same length as points.

##### font\_size

[int, default: 12] Font size for the labels.

##### point\_size

[float, default: 5.0] Size of the point markers shown with labels.

##### \*\*kwargs

[dict] Additional backend-specific keyword arguments for advanced customization (e.g., text\_color, shape, fill\_shape).

#### Returns

##### Any

Backend-specific actor or object representing the added labels. Can be used for further manipulation or removal.

## Examples

Add labels at specific locations:

```
>>> from ansys.tools.visualization_interface import Plotter  
>>> plotter = Plotter()  
>>> points = [[0, 0, 0], [1, 0, 0], [0, 1, 0]]  
>>> labels = ['Origin', 'X-axis', 'Y-axis']  
>>> plotter.add_labels(points, labels, font_size=14)  
>>> plotter.show()
```

Add labels with custom styling:

```
>>> plotter.add_labels(  
...     points,  
...     labels,
```

(continues on next page)

(continued from previous page)

```

...     font_size=16,
...     point_size=10,
...     text_color='yellow',
...     shape='rounded_rect'
...
>>> plotter.show()

```

**Plotter.clear() → None**

Clear all actors from the scene.

This method removes all previously added objects (meshes, points, lines, text, etc.) from the visualization scene, therefore allowing the plotter to be reused after show() has been called.

**Examples**

Clear before showing:

```

>>> from ansys.tools.visualization_interface import Plotter
>>> import pyvista as pv
>>> plotter = Plotter()
>>> plotter.plot(pv.Sphere())
>>> plotter.clear() # Changed mind, remove sphere
>>> plotter.plot(pv.Cube())
>>> plotter.show()

```

Clear after showing

```

>>> plotter = Plotter()
>>> plotter.plot(pv.Sphere())
>>> plotter.show()
>>> plotter.clear() # Reset the plotter to reuse it
>>> plotter.plot(pv.Cube())
>>> plotter.show()

```

**Description**

Module for the Plotter class.

**4.1.2 Description**

Visualization Interface Tool is a Python client library for visualizing the results of Ansys simulations.

**4.1.3 Module detail**

`visualization_interface.USE_FRAME: bool = False`

`visualization_interface.DOCUMENTATION_BUILD: bool`

Whether the documentation is being built or not.

`visualization_interface.TESTING_MODE: bool`

Whether the library is being built or not, used to avoid showing plots while testing.

`visualization_interface.USE_HTML_BACKEND: bool`

Whether the library is being built or not, used to avoid showing plots while testing.

`visualization_interface.__version__`

---

**CHAPTER****FIVE**

---

**EXAMPLES**

This section shows how to use the Visualization Interface Tool to perform many different types of operations.



---

**CHAPTER  
SIX**

---

## **BASIC USAGE EXAMPLES**

These examples show how to use the general plotter included in the Visualization Interface Tool.



---

**CHAPTER  
SEVEN**

---

## **BASIC PLOTLY USAGE EXAMPLES**

These examples show how to use the general plotter with Plotly backend included in the Visualization Interface Tool.



## ADVANCED USAGE EXAMPLES

These examples show how to use the Visualization Interface Tool to postprocess simulation data.

### 8.1 Basic usage examples

These examples show how to use the general plotter included in the Visualization Interface Tool.

#### 8.1.1 Use trame as a remote service

This example shows how to launch a trame service and use it as a remote service.

First, we need to launch the trame service. We can do this by running the following code:

```
# import required libraries
from ansys.tools.visualization_interface.backends.pyvista.trame_service import (
    TrameService,
)

# create a trame service, in whatever port is available in your system
ts = TrameService(websocket_port=8765)

# run the service
ts.run()
```

Now, we can send meshes and plotter to the trame service. We can do this by running the following code in a separate terminal:

```
# import required libraries
import time

import pyvista as pv

from ansys.tools.visualization_interface.backends.pyvista.trame_remote import (
    send_mesh,
    send_pl,
)

# create an example plotter
plotter = pv.Plotter()
plotter.add_mesh(pv.Cube())

# send some example meshes
```

(continues on next page)

(continued from previous page)

```
send_mesh(pv.Sphere())
send_mesh(pv.Sphere(center=(3, 0, 0)))
time.sleep(4)

# if we send a plotter, the previous meshes will be deleted.
send_pl(plotter)
```

**Total running time of the script:** (0 minutes 0.000 seconds)

### 8.1.2 Use a PyVista Qt backend

PyVista Qt is a package that extends the PyVista functionality through the usage of Qt. Qt applications operate in a separate thread than VTK, you can simultaneously have an active VTK plot and a non-blocking Python session.

This example shows how to use the PyVista Qt backend to create a plotter

```
import pyvista as pv

from ansys.tools.visualization_interface import Plotter
from ansys.tools.visualization.backends.pyvista import PyVistaBackend
```

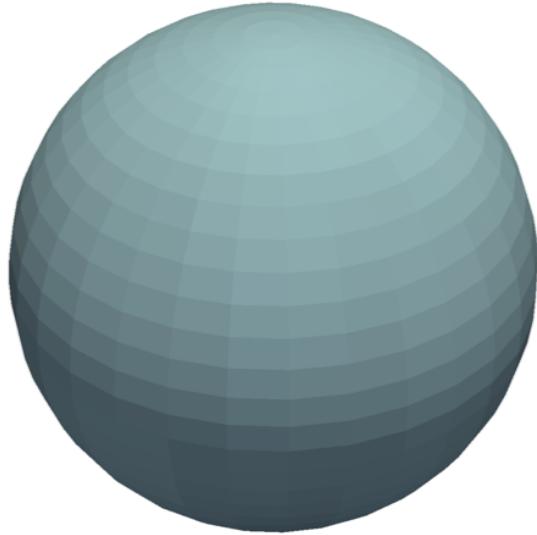
#### Open a pyvistaqt window

```
cube = pv.Cube()
pv_backend = PyVistaBackend(use_qt=True, show_qt=True)
pl = Plotter(backend=pv_backend)
pl.plot(cube)
pl.backend.enable_widgets()
pv_backend.scene.show()
```

#### Parallel VTK window

```
sphere = pv.Sphere()

pl_parallel = Plotter()
pl_parallel.plot(sphere)
pl_parallel.show()
```



[]

### Close the `pyvistaqt` window

```
pv_backend.close()
```

### Integrate the plotter in a Qt application

```
pv_backend = PyVistaBackend(use_qt=True, show_qt=False)
pv_backend.enable_widgets()

# You can use this plotter in a Qt application
pl = pv_backend.scene
```

Total running time of the script: (0 minutes 2.468 seconds)

#### 8.1.3 Use a clipping plane

This example shows how to use a clipping plane in the Visualization Interface Tool to cut a mesh.

```
import pyvista as pv
```

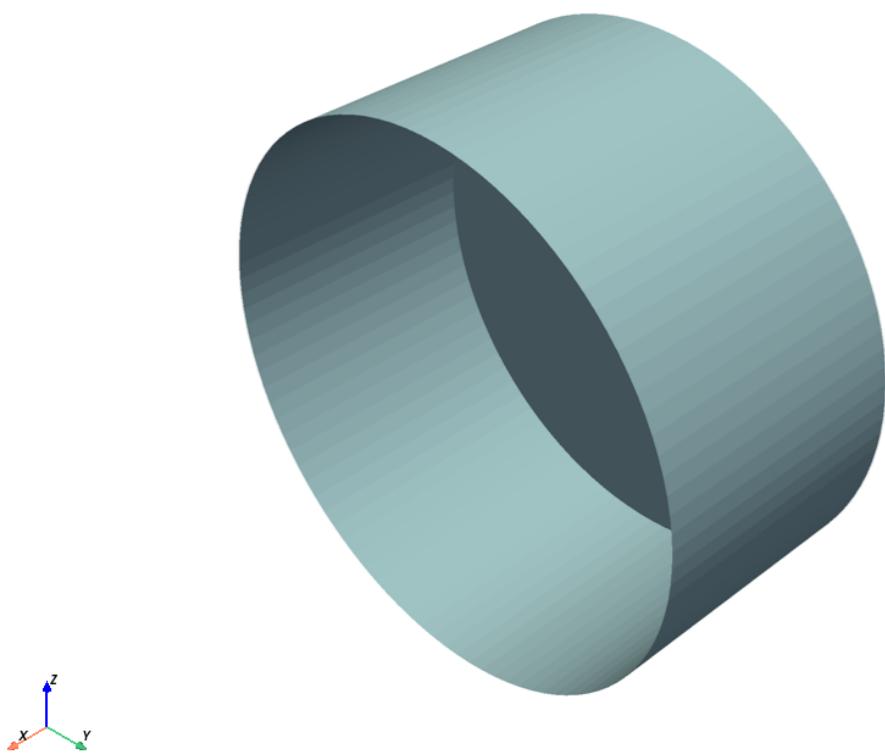
(continues on next page)

(continued from previous page)

```
from ansys.tools.visualization_interface import ClipPlane, Plotter  
mesh = pv.Cylinder()
```

### Create a plotter and clip the mesh

```
pl = Plotter()  
  
# Create a clipping plane  
clipping_plane = ClipPlane(normal=(1, 0, 0), origin=(0, 0, 0))  
  
# Add the mesh to the plotter with the clipping plane  
pl.plot(mesh, clipping_plane=clipping_plane)  
pl.show()
```



[]

Total running time of the script: (0 minutes 0.191 seconds)

### 8.1.4 Use the MeshObjectPlot class

The Visualization Interface Tool provides the `MeshObject` helper class to relate a custom object with its mesh. With a custom object, you can take advantage of the full potential of the Visualization Interface Tool.

This example shows how to use the `MeshObjectPlot` class to plot your custom objects.

#### Relate CustomObject class with a PyVista mesh

```
import pyvista as pv

# Note that the ``CustomObject`` class must have a way to get the mesh
# and a name or ID.

class CustomObject:
    def __init__(self):
        self.name = "CustomObject"
        self.mesh = pv.Cube()

    def get_mesh(self):
        return self.mesh

    def name(self):
        return self.name

# Create a custom object
custom_object = CustomObject()
```

#### Create a MeshObjectPlot instance

```
from ansys.tools.visualization_interface import MeshObjectPlot

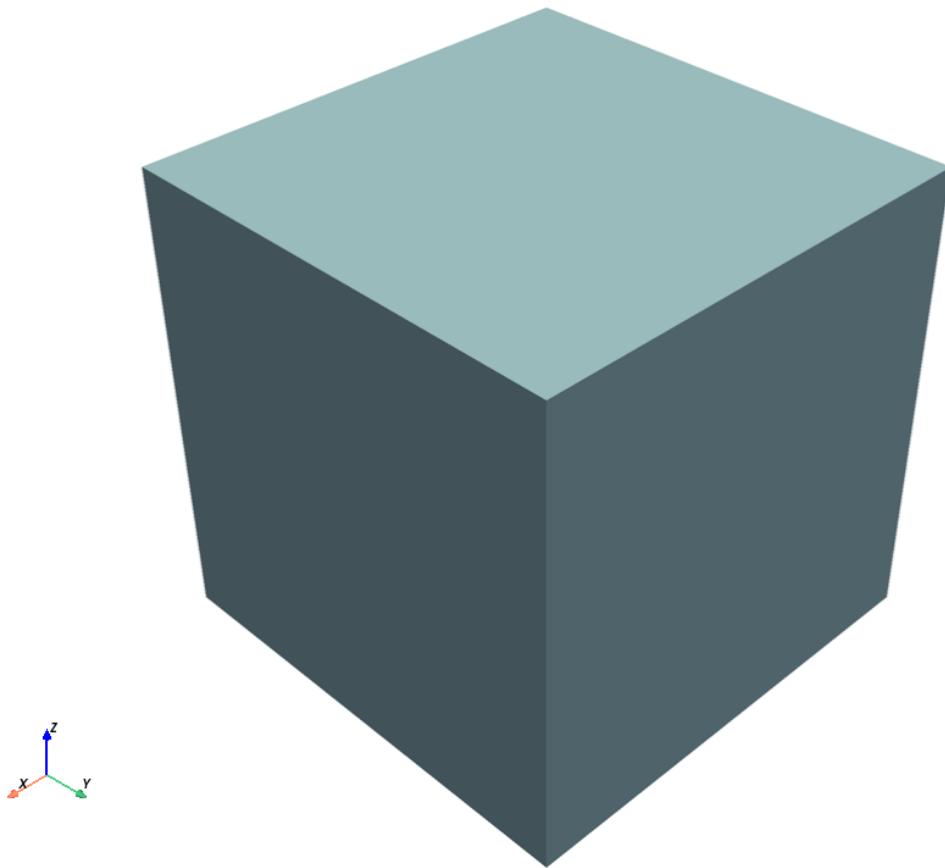
# Create an instance

mesh_object = MeshObjectPlot(custom_object, custom_object.get_mesh())
```

#### Plot the MeshObjectPlot instance

```
from ansys.tools.visualization_interface import Plotter

pl = Plotter()
pl.plot(mesh_object)
pl.show()
```



Total running time of the script: (0 minutes 0.188 seconds)

### 8.1.5 Use the plotter

This example shows how to add one or more meshes to the plotter.

#### Add a mesh to the plotter

This code shows how to add a single mesh to the plotter.

```
import pyvista as pv

from ansys.tools.visualization_interface import Plotter

mesh = pv.Cube()

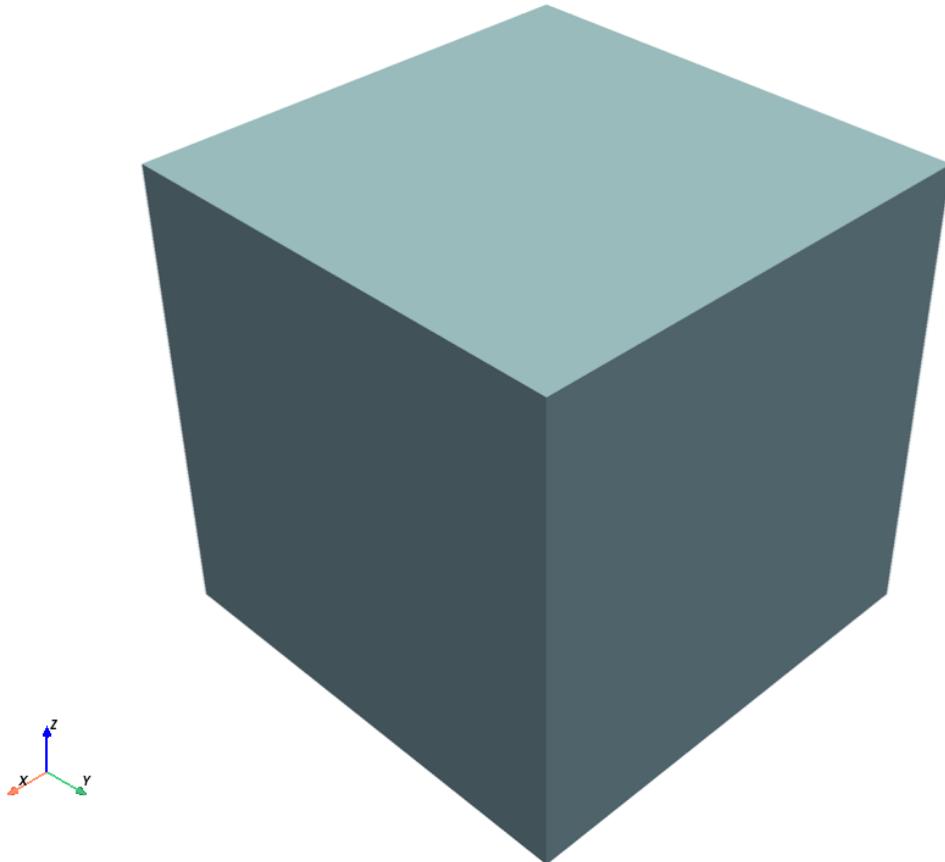
# Create a plotter
pl = Plotter()

# Add the mesh to the plotter
pl.plot(mesh)
```

(continues on next page)

(continued from previous page)

```
# Show the plotter  
pl.show()
```



[]

## Getting a screenshot

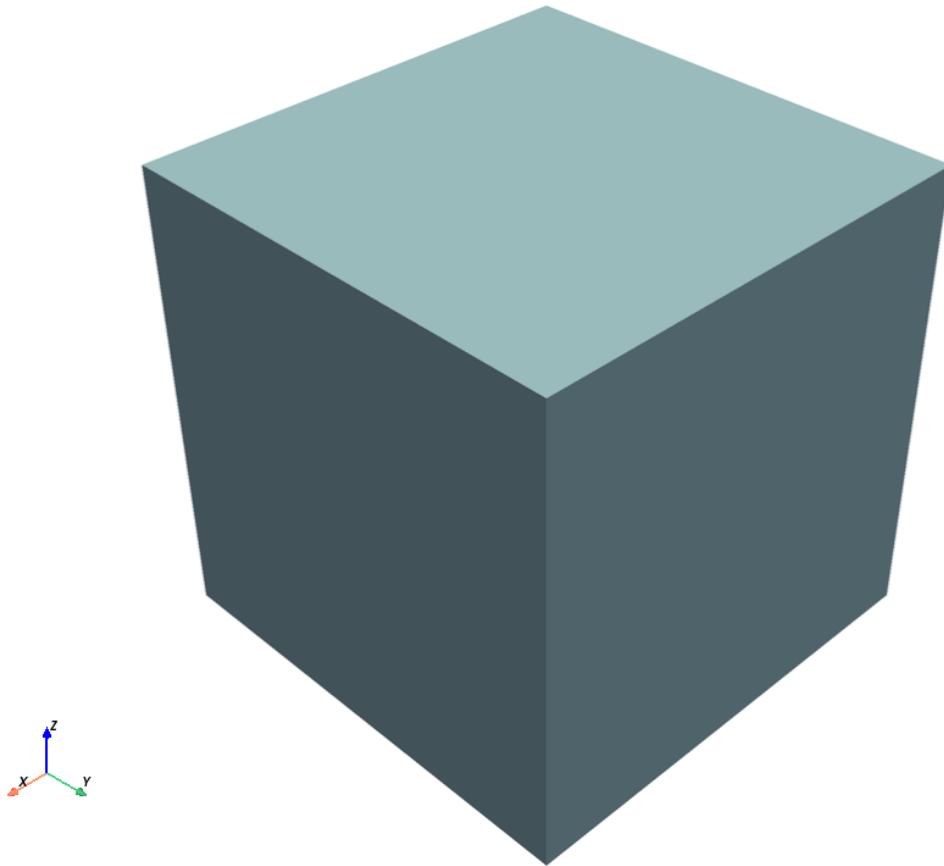
Now we will check how to get a screenshot from our plotter.

```
import pyvista as pv  
  
from ansys.tools.visualization_interface import Plotter  
  
mesh = pv.Cube()  
  
# Create a plotter  
pl = Plotter()  
  
# Add the mesh to the plotter  
pl.plot(mesh)  
  
# Show the plotter
```

(continues on next page)

(continued from previous page)

pl.show()



[]

### Add a list of meshes

This code shows how to add a list of meshes to the plotter.

```
import pyvista as pv

from ansys.tools.visualization_interface import Plotter

mesh1 = pv.Cube()
mesh2 = pv.Sphere(center=(2, 0, 0))
mesh_list = [mesh1, mesh2]
# Create a plotter
pl = Plotter()

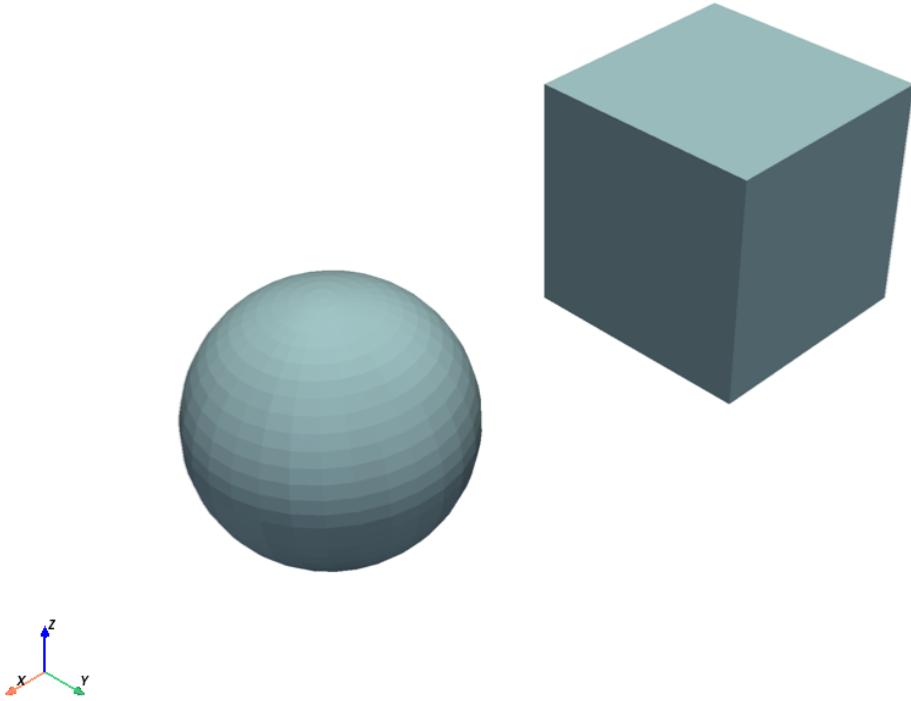
# Add a list of meshes to the plotter
pl.plot(mesh_list)

# Show the plotter
```

(continues on next page)

(continued from previous page)

pl.show()



[]

**Total running time of the script:** (0 minutes 0.538 seconds)

### 8.1.6 Animation Example

This example demonstrates how to create and display animations for time-series simulation data using the PyVista backend.

```
import numpy as np
import pyvista as pv

from ansys.tools.visualization_interface import Plotter
```

#### Create sample animation data

Generate a series of meshes representing a wave propagation over time.

```
def create_wave_mesh(time_step, n_points=50):
    """Create a mesh with a wave pattern for a given time step."""
```

(continues on next page)

(continued from previous page)

```
# Create grid
x = np.linspace(-5, 5, n_points)
y = np.linspace(-5, 5, n_points)
x, y = np.meshgrid(x, y)

# Create wave pattern that evolves over time
t = time_step * 0.2
z = np.sin(np.sqrt(x**2 + y**2) - t) * np.exp(-0.1 * np.sqrt(x**2 + y**2))

# Create structured grid
mesh = pv.StructuredGrid(x, y, z)
mesh["displacement"] = np.abs(z).ravel()

return mesh

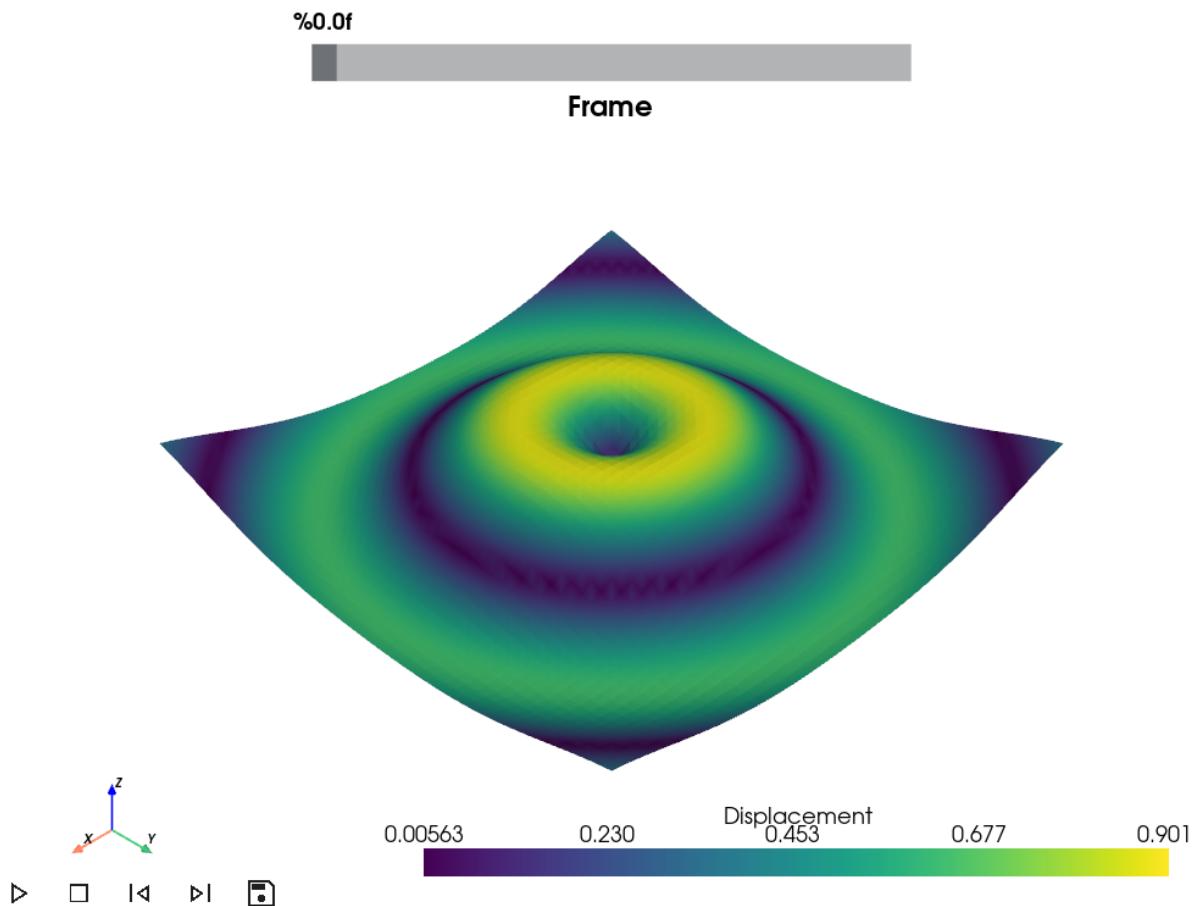
# Create 30 frames
frames = [create_wave_mesh(i) for i in range(30)]
```

### Display animation with interactive controls

Create and show an animation with play/pause, stop, and frame navigation.

```
plotter = Plotter()
animation = plotter.animate(
    frames,
    fps=20,
    loop=True,
    scalar_bar_args={"title": "Displacement"})
)

# Display with interactive controls
animation.show()
```



### Interactive Controls

The animation window includes the following controls:

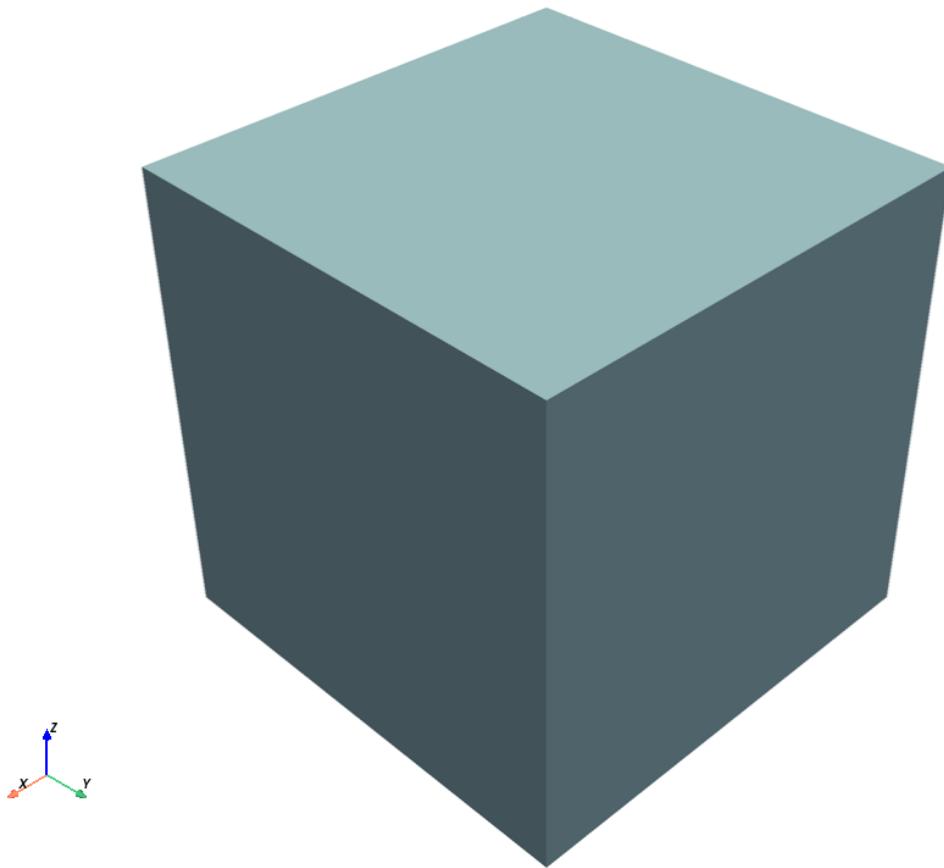
- **Play/Pause button** (green): Start/stop animation playback
- **Stop button** (red): Reset to first frame
- **Previous/Next buttons** (blue): Step through frames manually
- **Save GIF button** (orange): Export animation (requires imageio)
- **Frame slider** (top): Jump to any frame

You can rotate, zoom, and pan the view while the animation plays.

**Total running time of the script:** (0 minutes 0.820 seconds)

### 8.1.7 MeshObjectPlot tree structure

This example shows how to add a tree structure of MeshObjectPlot to the plotter.



[]

```
import pyvista as pv
from ansys.tools.visualization_interface import Plotter

class CustomObject:
    def __init__(self):
        self.name = "CustomObject"
        self.mesh = pv.Cube(center=(1, 1, 0))

    def get_mesh(self):
        return self.mesh

    def name(self):
        return self.name
```

(continues on next page)

(continued from previous page)

```
# Create a custom objects
custom_cube = CustomObject()
custom_cube.name = "CustomCube"

custom_sphere = CustomObject()
custom_sphere.mesh = pv.Sphere(center=(0, 0, 5))
custom_sphere.name = "CustomSphere"

custom_sphere1 = CustomObject()
custom_sphere1.mesh = pv.Sphere(center=(5, 0, 5))
custom_sphere1.name = "CustomSphere"

from ansys.tools.visualization_interface import MeshObjectPlot

# Create an instance
mesh_object_cube = MeshObjectPlot(custom_cube, custom_cube.get_mesh())
mesh_object_sphere = MeshObjectPlot(custom_sphere, custom_sphere.get_mesh())
mesh_object_sphere1 = MeshObjectPlot(custom_sphere1, custom_sphere1.get_mesh())

mesh_object_cube.add_child(mesh_object_sphere)
mesh_object_sphere.add_child(mesh_object_sphere1)

pl = Plotter()
pl.plot(mesh_object_cube, plot_children=True)

pl.backend._pl.hide_children(mesh_object_cube)
pl.show()
```

**Total running time of the script:** (0 minutes 0.208 seconds)

### 8.1.8 Activate the picker

This example shows how to activate the picker, which is the tool that you use to select an object in the plotter and get its name.

#### Relate CustomObject class with a PyVista mesh

```
import pyvista as pv

# Note that the ``CustomObject`` class must have a way to get the mesh
# and a name or ID.

class CustomObject:
    def __init__(self):
        self.name = "CustomObject"
        self.mesh = pv.Cube(center=(1, 1, 0))

    def get_mesh(self):
        return self.mesh

    def name(self):
```

(continues on next page)

(continued from previous page)

```
return self.name

# Create a custom object
custom_cube = CustomObject()
custom_cube.name = "CustomCube"
custom_sphere = CustomObject()
custom_sphere.mesh = pv.Sphere(center=(0, 0, 5))
custom_sphere.name = "CustomSphere"
```

### Create two MeshObjectPlot instances

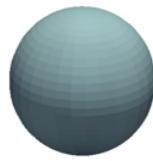
```
from ansys.tools.visualization_interface import MeshObjectPlot

# Create an instance
mesh_object_cube = MeshObjectPlot(custom_cube, custom_cube.get_mesh())
mesh_object_sphere = MeshObjectPlot(custom_sphere, custom_sphere.get_mesh())
```

### Activate the picking capabilities

```
from ansys.tools.visualization_interface import Plotter
from ansys.tools.visualization_interface.backends.pyvista import PyVistaBackend

pv_backend = PyVistaBackend(allow_picking=True, plot_picked_names=True)
pl = Plotter(backend=pv_backend)
pl.plot(mesh_object_cube)
pl.plot(mesh_object_sphere)
pl.show()
```

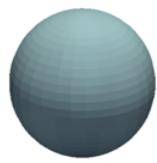


[]

### Activate the hover capabilities

```
from ansys.tools.visualization_interface import Plotter
from ansys.tools.visualization_interface.backends.pyvista import PyVistaBackend

pv_backend = PyVistaBackend(allow_hovering=True)
pl = Plotter(backend=pv_backend)
pl.plot(mesh_object_cube)
pl.plot(mesh_object_sphere)
pl.show()
```



[]

### Using StructuredGrid mesh

```
import numpy as np

class CustomStructuredObject:
    def __init__(self):
        self.name = "CustomObject"
        xrng = np.arange(-10, 10, 2, dtype=np.float32)
        yrng = np.arange(-10, 10, 5, dtype=np.float32)
        zrng = np.arange(-10, 10, 1, dtype=np.float32)
        x, y, z = np.meshgrid(xrng, yrng, zrng, indexing='ij')
        grid = pv.StructuredGrid(x, y, z)
        self.mesh = grid

    def get_mesh(self):
        return self.mesh

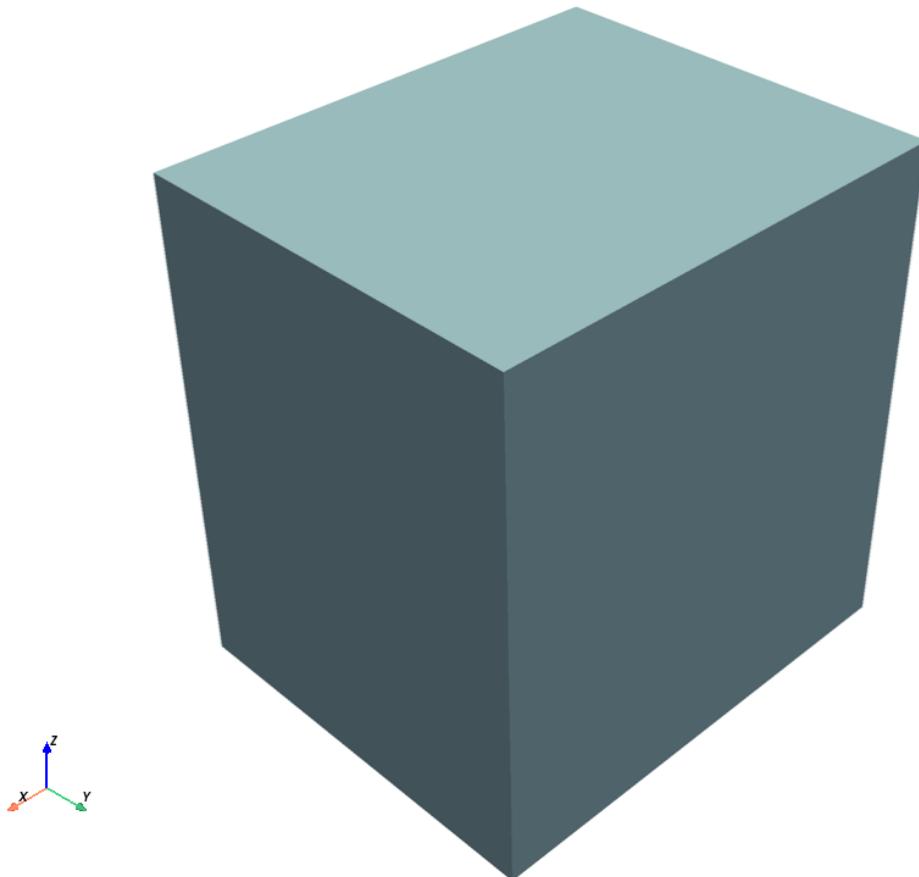
    def name(self):
        return self.name
```

(continues on next page)

(continued from previous page)

```
pv_backend = PyVistaBackend()
pl = Plotter(backend=pv_backend)

structured_object = CustomStructuredObject()
mo_plot = MeshObjectPlot(structured_object, structured_object.get_mesh())
pl.plot(mo_plot)
pl.show()
```



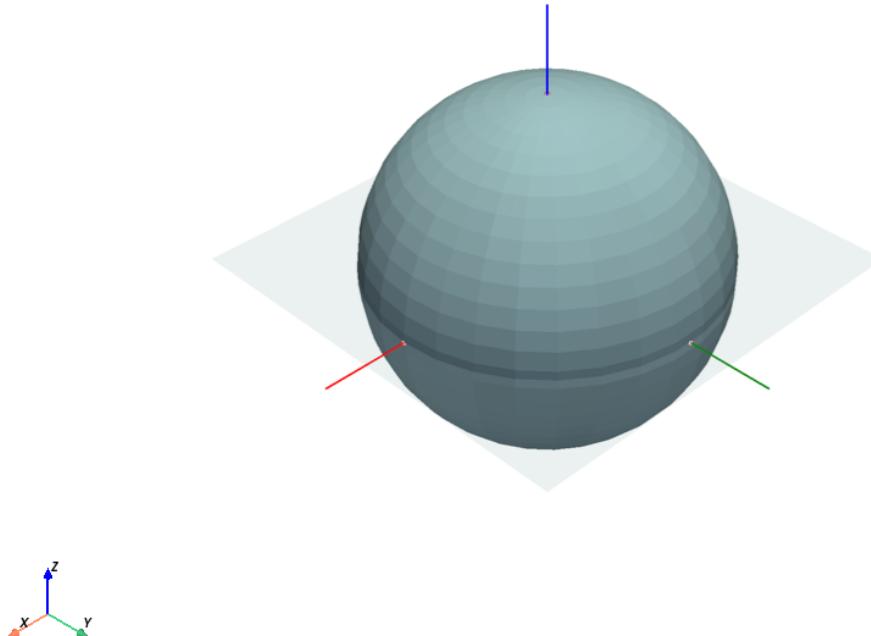
[]

**Total running time of the script:** (0 minutes 0.510 seconds)

### 8.1.9 Customization API example

This example demonstrates how to use the customization API of the visualization interface to add various elements to a PyVista scene, such as points, lines, planes, and text annotations. The example also shows how to plot a simple sphere mesh and customize its appearance.

## PyVista Backend Customization API Example



### 3D Visualization

```
/home/runner/work/ansys-tools-visualization-interface/ansys-tools-visualization-
interface/.venv/lib/python3.14/site-packages/ansys/tools/visualization_interface/
backends/pyvista/pyvista.py:807: UserWarning:
Points is not a float type. This can cause issues when transforming or applying filters. ↴
↳ Casting to ``np.float32``. Disable this by passing ``force_float=False``.

/home/runner/work/ansys-tools-visualization-interface/ansys-tools-visualization-
interface/.venv/lib/python3.14/site-packages/ansys/tools/visualization_interface/
backends/pyvista/pyvista.py:1020: UserWarning:
Points is not a float type. This can cause issues when transforming or applying filters. ↴
↳ Casting to ``np.float32``. Disable this by passing ``force_float=False``.

[]
```

```
from ansys.tools.visualization_interface import Plotter
```

(continues on next page)

(continued from previous page)

```
import pyvista as pv

# Create a plotter using the Plotly backend and add basic geometry.

plotter = Plotter()

# Add a sphere - this works fine
sphere = pv.Sphere(radius=1.0, center=(0, 0, 0))
plotter.plot(sphere)

# Add point markers to highlight specific locations.

key_points = [
    [1, 0, 0],    # Point on X axis
    [0, 1, 0],    # Point on Y axis
    [0, 0, 1],    # Point on Z axis
]
plotter.add_points(key_points, color='red', size=10)

# Add line segments to show coordinate axes.

# X axis
x_axis = [[0, 0, 0], [1.5, 0, 0]]
plotter.add_lines(x_axis, color='red', width=4.0)

# Y axis
y_axis = [[0, 0, 0], [0, 1.5, 0]]
plotter.add_lines(y_axis, color='green', width=4.0)

# Z axis
z_axis = [[0, 0, 0], [0, 0, 1.5]]
plotter.add_lines(z_axis, color='blue', width=4.0)

# Add a plane to show a reference surface.

plotter.add_planes(
    center=(0, 0, 0),
    normal=(0, 0, 1),
    i_size=2.5,
    j_size=2.5,
    color='lightblue',
    opacity=0.2
)

# Scene title at the top center
plotter.add_text("Customization API Example", position="upper_edge", font_size=18, color='black')
```

(continues on next page)

(continued from previous page)

```
# Additional labels at the top left corner using a string for the position as before
plotter.add_text("PyVista Backend", position="upper_left", font_size=12, color='lightblue'
                ↵)
# Additional labels at the bottom left corner using pixel coordinates
plotter.add_text("3D Visualization", position=(0.95, 0.95), font_size=12, color=
                ↵'lightgreen')

# Add labels at specific 3D points to annotate key locations in space.

label_points = [
    [1, 0, 0],    # X axis endpoint
    [0, 1, 0],    # Y axis endpoint
    [0, 0, 1],    # Z axis endpoint
]

labels = ['X-axis', 'Y-axis', 'Z-axis']

plotter.add_labels(label_points, labels, font_size=16, point_size=8.0)

# The clear() method resets the plotter and can be called even after show().
# This allows reusing the same plotter for multiple visualizations.

# Uncomment to clear everything added above and start fresh:
# plotter.show()
# plotter.clear()
# plotter.plot(pv.Cube()) # Would show only a cube instead

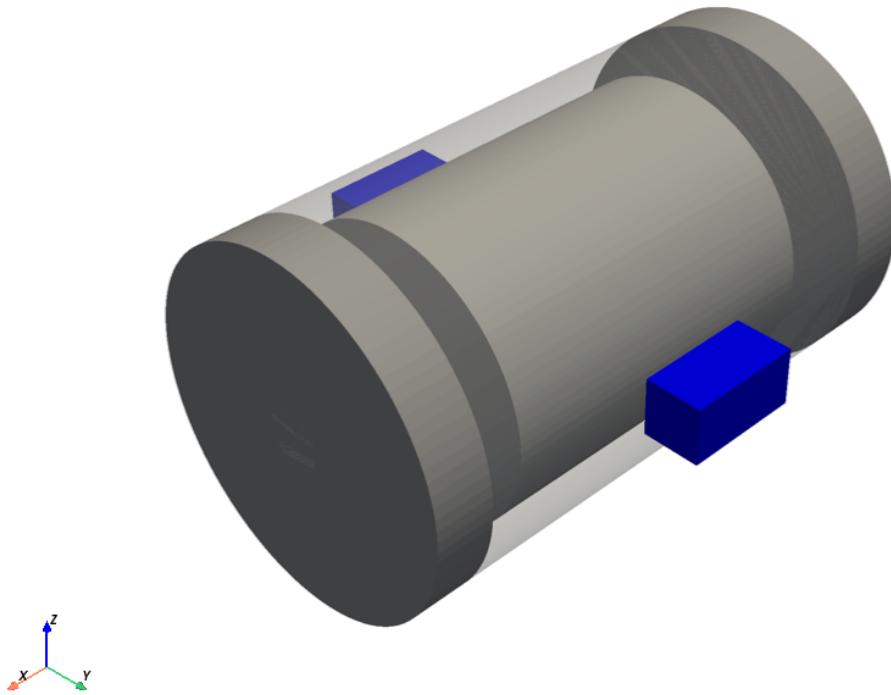
# Display the visualization with all customizations.

plotter.show()
```

**Total running time of the script:** (0 minutes 0.401 seconds)

### 8.1.10 Tree view menu

This example demonstrates the use of a tree view menu to manage the visibility of components in an encased motor block assembly. The tree menu allows users to interactively show or hide different parts of the assembly.



#### ENCASED MOTOR BLOCK ASSEMBLY

##### HIERARCHY:

```
Motor Housing (semi-transparent)
  └── Stator
    └── Rotor
      ├── Shaft
      ├── Front Bearing
      └── Rear Bearing
  └── Front End Cap
  └── Rear End Cap
  └── Top Mounting Bracket
  └── Bottom Mounting Bracket
```

##### TREE MENU (Right side):

- Click eye icons to show/hide components
- Hiding a parent hides all children
- Example: Hide 'Motor Housing' to hide entire assembly
- Example: Hide 'Rotor' to hide rotor, shaft, and bearings

##### USE CASES:

(continues on next page)

(continued from previous page)

1. Hide housing to see internal components
2. Hide stator to see only rotor assembly
3. Hide brackets to focus on motor internals
4. Hide individual bearings or shaft

**CONTROLS:**

- Toggle button (bottom left) = Show/hide tree menu
  - Eye icons in menu = Toggle component visibility
- 

[]

```

import pyvista as pv
from ansys.tools.visualization_interface import Plotter, MeshObjectPlot
from ansys.tools.visualization_interface.backends.pyvista.pyvista import PyVistaBackend

# Simple helper class to hold component names
class MotorComponent:
    """Simple class to hold motor component name."""
    def __init__(self, name: str):
        self.name = name

# Create Motor Housing (outer casing)
housing = MeshObjectPlot(
    MotorComponent("Motor Housing"),
    pv.Cylinder(center=(0, 0, 0), direction=(1, 0, 0), radius=3, height=8)
)

# Create Stator (fixed part inside housing)
stator = MeshObjectPlot(
    MotorComponent("Stator"),
    pv.Cylinder(center=(0, 0, 0), direction=(1, 0, 0), radius=2.5, height=6)
)

# Create Rotor (rotating part)
rotor = MeshObjectPlot(
    MotorComponent("Rotor"),
    pv.Cylinder(center=(0, 0, 0), direction=(1, 0, 0), radius=1.8, height=5.5)
)

# Create Shaft (central rotating shaft)
shaft = MeshObjectPlot(
    MotorComponent("Shaft"),
    pv.Cylinder(center=(0, 0, 0), direction=(1, 0, 0), radius=0.5, height=10)
)

```

(continues on next page)

(continued from previous page)

```

)
# Create Bearings
bearing_front = MeshObjectPlot(
    MotorComponent("Front Bearing"),
    pv.Cylinder(center=(3.5, 0, 0), direction=(1, 0, 0), radius=0.8, height=0.5)
)

bearing_rear = MeshObjectPlot(
    MotorComponent("Rear Bearing"),
    pv.Cylinder(center=(-3.5, 0, 0), direction=(1, 0, 0), radius=0.8, height=0.5)
)

# Create End Caps
end_cap_front = MeshObjectPlot(
    MotorComponent("Front End Cap"),
    pv.Cylinder(center=(4.5, 0, 0), direction=(1, 0, 0), radius=3, height=1)
)

end_cap_rear = MeshObjectPlot(
    MotorComponent("Rear End Cap"),
    pv.Cylinder(center=(-4.5, 0, 0), direction=(1, 0, 0), radius=3, height=1)
)

# Create Mounting Brackets
bracket_top = MeshObjectPlot(
    MotorComponent("Top Mounting Bracket"),
    pv.Box(bounds=(-1, 1, 3, 4, -0.5, 0.5))
)

bracket_bottom = MeshObjectPlot(
    MotorComponent("Bottom Mounting Bracket"),
    pv.Box(bounds=(-1, 1, -4, -3, -0.5, 0.5))
)

# Build hierarchy - Housing contains all internal components
housing.add_child(stator)
housing.add_child(end_cap_front)
housing.add_child(end_cap_rear)
housing.add_child(bracket_top)
housing.add_child(bracket_bottom)

# Stator contains rotor
stator.add_child(rotor)

# Rotor contains shaft and bearings
rotor.add_child(shaft)
rotor.add_child(bearing_front)
rotor.add_child(bearing_rear)

# Create plotter with picking enabled (automatically adds tree menu)
backend = PyVistaBackend()

```

(continues on next page)

(continued from previous page)

```

pl = Plotter(backend=backend)

# Plot all components with different colors
pl.plot(housing, color="lightgray", opacity=0.3) # Semi-transparent housing
pl.plot(stator, color="darkgray")
pl.plot(rotor, color="orange")
pl.plot(shaft, color="silver")
pl.plot(bearing_front, color="gold")
pl.plot(bearing_rear, color="gold")
pl.plot(end_cap_front, color="darkgray")
pl.plot(end_cap_rear, color="darkgray")
pl.plot(bracket_top, color="blue")
pl.plot(bracket_bottom, color="blue")

print("\n" + "="*80)
print(" " * 25 + "ENCASED MOTOR BLOCK ASSEMBLY")
print("="*80)
print("\nHIERARCHY:")
print(" Motor Housing (semi-transparent)")
print("   Stator")
print("     Rotor")
print("       Shaft")
print("       Front Bearing")
print("       Rear Bearing")
print("   Front End Cap")
print("   Rear End Cap")
print("   Top Mounting Bracket")
print("   Bottom Mounting Bracket")
print("\nTREE MENU (Right side):")
print(" - Click eye icons to show/hide components")
print(" - Hiding a parent hides all children")
print(" - Example: Hide 'Motor Housing' to hide entire assembly")
print(" - Example: Hide 'Rotor' to hide rotor, shaft, and bearings")
print("\nUSE CASES:")
print(" 1. Hide housing to see internal components")
print(" 2. Hide stator to see only rotor assembly")
print(" 3. Hide brackets to focus on motor internals")
print(" 4. Hide individual bearings or shaft")
print("\nCONTROLS:")
print(" - Toggle button (bottom left) = Show/hide tree menu")
print(" - Eye icons in menu = Toggle component visibility")
print("=*80 + "\n")

# Show the plot
pl.show()

```

**Total running time of the script:** (0 minutes 0.308 seconds)

### 8.1.11 Create custom picker

This example shows how to create a custom picker. In this case we will show how the default picker is implemented through the `AbstractPicker` class.

#### Import the `AbstractPicker` class

```
# Import the abstract picker class
from ansys.tools.visualization_interface.backends.pyvista.picker import AbstractPicker

# Import custom object meshes
from ansys.tools.visualization_interface.types.mesh_object_plot import MeshObjectPlot

# Import plotter and color enum
from ansys.tools.visualization_interface import Plotter
from ansys.tools.visualization_interface.utils.color import Color
```

#### Create a custom picker class

```
class CustomPicker(AbstractPicker):
    """Custom picker class that extends the AbstractPicker.
    This custom picker changes the color of picked objects to red and adds a label with
    the object's name.
    It also adds a label when hovering over an object.

    Parameters
    -----
    plotter_backend : Plotter
        The plotter backend to use.
    plot_picked_names : bool, optional
        Whether to plot the names of picked objects, by default True.
    label : str, optional
        Extra parameter to exemplify the usage of custom parameters.
    """

    def __init__(self, plotter_backend: "Plotter", plot_picked_names: bool = True,
                 label: str = "This label: ") -> None:
        """Initialize the ``Picker`` class."""
        # Picking variables
        self._plotter_backend = plotter_backend
        self._plot_picked_names = plot_picked_names
        self._label = label

        # Map that relates PyVista actors with the added actors by the picker
        self._picker_added_actors_map = {}

        # Dictionary of picked objects in MeshObject format.
        self._picked_dict = {}

        # Map that saves original colors of the plotted objects.
        self._origin_colors = {}

        # Hovering variables
        self._added_hover_labels = []
```

(continues on next page)

(continued from previous page)

```

def pick_select_object(self, custom_object: MeshObjectPlot, pt: "np.ndarray") ->None:
    """Add actor to picked list and add label if required.

    Parameters
    -----
    custom_object : MeshObjectPlot
        The object to be selected.
    pt : np.ndarray
        The point where the object was picked.
    """
    added_actors = []

    # Pick only custom objects
    if isinstance(custom_object, MeshObjectPlot):
        self._origin_colors[custom_object] = custom_object.actor.prop.color
        custom_object.actor.prop.color = Color.PICKED.value

    # Get the name for the text label
    text = custom_object.name

    # If picking names is enabled, add a label to the picked object
    if self._plot_picked_names:
        label_actor = self._plotter_backend.pv_interface.scene.add_point_labels(
            [pt],
            [self._label + text],
            always_visible=True,
            point_size=0,
            render_points_as_spheres=False,
            show_points=False,
        )
        # Add the label actor to the list of added actors
        added_actors.append(label_actor)

    # Add the picked object to the picked dictionary if not already present, to keep
    track of it
    if custom_object.name not in self._picked_dict:
        self._picked_dict[custom_object.name] = custom_object
    # Add the picked object to the picked dictionary if not already present, to keep
    track of it
    self._picker_added_actors_map[custom_object.actor.name] = added_actors

def pick_unselect_object(self, custom_object: MeshObjectPlot) -> None:
    """Remove actor from picked list and remove label if required.

    Parameters
    -----
    custom_object : MeshObjectPlot
        The object to be unselected.
    """
    # remove actor from picked list and from scene

```

(continues on next page)

(continued from previous page)

```

if custom_object.name in self._picked_dict:
    self._picked_dict.pop(custom_object.name)

    # Restore original color if it was changed
    if isinstance(custom_object, MeshObjectPlot) and custom_object in self._origin_
→colors:
        custom_object.actor.prop.color = self._origin_colors[custom_object]

    # Remove any added actors (like labels) associated with this picked object
    if custom_object.actor.name in self._picker_added_actors_map:
        self._plotter_backend._pl.scene.remove_actor(self._picker_added_actors_
→map[custom_object.actor.name])
        self._picker_added_actors_map.pop(custom_object.actor.name)

def hover_select_object(self, custom_object: MeshObjectPlot, actor: "Actor") -> None:
    """Add label to hovered object if required.

    Parameters
    -----
    custom_object : MeshObjectPlot
        The object to be hovered over.
    actor : vtkActor
        The actor corresponding to the hovered object.
    """
    for label in self._added_hover_labels:
        self._plotter_backend._pl.scene.remove_actor(label)
    label_actor = self._plotter_backend._pl.scene.add_point_labels(
        [actor.GetCenter()],
        [custom_object.name],
        always_visible=True,
        point_size=0,
        render_points_as_spheres=False,
        show_points=False,
    )
    self._added_hover_labels.append(label_actor)

def hover_unselect_object(self):
    """Remove all hover labels from the scene."""
    for label in self._added_hover_labels:
        self._plotter_backend._pl.scene.remove_actor(label)

@property
def picked_dict(self) -> dict:
    """Return the dictionary of picked objects.

    Returns
    -----
    dict
        Dictionary of picked objects.
    """
    return self._picked_dict

```

**Initialize the plotter backend with the custom picker**

```
from ansys.tools.visualization_interface.backends.pyvista import PyVistaBackend
pl_backend = PyVistaBackend(allow_picking=True, custom_picker=CustomPicker)
```

**Create a custom object with a name to be picked**

```
import pyvista as pv

class CustomObject:
    def __init__(self):
        self.name = "CustomObject"
        self.mesh = pv.Cube(center=(1, 1, 0))

    def get_mesh(self):
        return self.mesh

    def name(self):
        return self.name

# Create a custom object
custom_cube = CustomObject()
custom_cube.name = "CustomCube"
```

### Create a MeshObjectPlot instance

```
from ansys.tools.visualization_interface import MeshObjectPlot
# Create an instance
mesh_object_cube = MeshObjectPlot(custom_cube, custom_cube.get_mesh())
```

### Display the plotter and interact with the object

```
pl = Plotter(backend=pl_backend)
pl.plot(mesh_object_cube)
pl.show()
```

Total running time of the script: (0 minutes 0.125 seconds)

## 8.2 Basic Plotly usage examples

These examples show how to use the general plotter with Plotly backend included in the Visualization Interface Tool.

### 8.2.1 Plain usage of the plotly dash backend

This example shows the plain usage of the Plotly Dash backend in the Visualization Interface Tool to plot different objects, including PyVista meshes, custom objects, and Plotly-specific objects.

```
from ansys.tools.visualization_interface.backends.plotly.plotly_dash import_
    PlotlyDashBackend
from ansys.tools.visualization_interface.types.mesh_object_plot import MeshObjectPlot
from ansys.tools.visualization_interface import Plotter
import pyvista as pv
from plotly.graph_objects import Mesh3d

# Create a plotter with the Plotly backend
pl = Plotter(backend=PlotlyDashBackend())

# Create a PyVista mesh
mesh = pv.Sphere()
mesh2 = pv.Cube(center=(2, 0, 0))
# Plot the mesh
pl.plot(mesh, name="Sphere")
pl.plot(mesh2, name="Cube")

# -----
# Start the server and show the plot
# -----
#
# ... code-block:: python
#
#     pl.show()
```

Total running time of the script: (0 minutes 0.818 seconds)

## 8.2.2 Plain usage of the plotly backend

This example shows the plain usage of the Plotly backend in the Visualization Interface Tool to plot different objects, including PyVista meshes, custom objects, and Plotly-specific objects.

```
from ansys.tools.visualization_interface.backends.plotly.plotly_interface import_
    PlotlyBackend
from ansys.tools.visualization_interface.types.mesh_object_plot import MeshObjectPlot
from ansys.tools.visualization_interface import Plotter
import pyvista as pv
from plotly.graph_objects import Mesh3d

# Create a plotter with the Plotly backend
pl = Plotter(backend=PlotlyBackend())

# Create a PyVista mesh
mesh = pv.Sphere()

# Plot the mesh
pl.plot(mesh)

# Create a PyVista MultiBlock
multi_block = pv.MultiBlock()
multi_block.append(pv.Sphere(center=(-1, -1, 0)))
multi_block.append(pv.Cube(center=(-1, 1, 0)))

# Plot the MultiBlock
pl.plot(multi_block)

# Display the plotter
pl.show()
```

```
/home/runner/work/ansys-tools-visualization-interface/ansys-tools-visualization-
    ↵interface/.venv/lib/python3.14/site-packages/ansys/tools/visualization_interface/
    ↵backends/plotly/plotly_interface.py:68: PyVistaFutureWarning:
```

The default value of `algorithm` for the filter  
`PolyData.extract\_surface` will change in the future. It currently defaults to  
`'dataset\_surface'`, but will change to `None`. Explicitly set the `algorithm` keyword to  
silence this warning.

```
/home/runner/work/ansys-tools-visualization-interface/ansys-tools-visualization-
    ↵interface/.venv/lib/python3.14/site-packages/ansys/tools/visualization_interface/
    ↵backends/plotly/plotly_interface.py:68: PyVistaFutureWarning:
```

The default value of `algorithm` for the filter  
`PolyData.extract\_surface` will change in the future. It currently defaults to  
`'dataset\_surface'`, but will change to `None`. Explicitly set the `algorithm` keyword to  
silence this warning.

## Now create a custom object

```

class CustomObject:
    def __init__(self):
        self.name = "CustomObject"
        self.mesh = pv.Cube(center=(1, 1, 0))

    def get_mesh(self):
        return self.mesh

    def name(self):
        return self.name

# Create a custom object
custom_cube = CustomObject()
custom_cube.name = "CustomCube"

# Create a MeshObjectPlot instance
mesh_object_cube = MeshObjectPlot(custom_cube, custom_cube.get_mesh())

# Plot the custom mesh object
pl.plot(mesh_object_cube)

```

## Display the plotter again

Since Plotly is a web-based visualization, we can show the plot again to include the new object.

```
pl.show()
```

## Add a Plotly Mesh3d object directly

```

custom_mesh3d = Mesh3d(
    x=[0, 1, 2],
    y=[0, 1, 0],
    z=[0, 0, 1],
    i=[0],
    j=[1],
    k=[2],
    color='lightblue',
    opacity=0.50
)
pl.plot(custom_mesh3d, name="CustomMesh3d")

# Show other plotly objects like Scatter3d
from plotly.graph_objects import Scatter3d

scatter = Scatter3d(
    x=[0, 1, 2],
    y=[0, 1, 0],
    z=[0, 0, 1],
    mode='markers',
    marker=dict(size=5, color='red')
)

```

(continues on next page)

(continued from previous page)

```
)  
pl.plot(scatter, name="CustomScatter3d")  
  
pl.show()
```

**Total running time of the script:** (0 minutes 0.079 seconds)

### 8.2.3 Backend-Agnostic Customization APIs (Plotly)

This example demonstrates the backend-agnostic customization APIs with the Plotly backend.

The same API calls work with both PyVista and Plotly backends, demonstrating the true backend-agnostic nature of these methods.

```
import pyvista as pv  
from ansys.tools.visualization_interface import Plotter  
from ansys.tools.visualization_interface.backends.plotly.plotly_interface import_  
    PlotlyBackend  
  
# Create a plotter using the Plotly backend and add basic geometry.  
  
plotter = Plotter(backend=PlotlyBackend())  
  
# Add a sphere - this works fine  
sphere = pv.Sphere(radius=1.0, center=(0, 0, 0))  
plotter.plot(sphere)  
  
# Add point markers to highlight specific locations.  
  
key_points = [  
    [1, 0, 0],    # Point on X axis  
    [0, 1, 0],    # Point on Y axis  
    [0, 0, 1],    # Point on Z axis  
]  
  
plotter.add_points(key_points, color='red', size=10)  
  
# Add line segments to show coordinate axes.  
  
# X axis  
x_axis = [[0, 0, 0], [1.5, 0, 0]]  
plotter.add_lines(x_axis, color='red', width=4.0)  
  
# Y axis  
y_axis = [[0, 0, 0], [0, 1.5, 0]]  
plotter.add_lines(y_axis, color='green', width=4.0)  
  
# Z axis  
z_axis = [[0, 0, 0], [0, 0, 1.5]]  
plotter.add_lines(z_axis, color='blue', width=4.0)
```

(continues on next page)

(continued from previous page)

```
# Add a plane to show a reference surface.

plotter.add_planes(
    center=(0, 0, 0),
    normal=(0, 0, 1),
    i_size=2.5,
    j_size=2.5,
    color='lightblue',
    opacity=0.2
)

# Add text annotations using 2D normalized coordinates (0-1 range).

# Scene title at the top center
plotter.add_text("Customization API Example", position=(0.5, 0.95), font_size=18, color=
    'black')

# Additional labels at the top corners
plotter.add_text("Plotly Backend", position=(0.05, 0.95), font_size=12, color='lightblue'
    )
plotter.add_text("3D Visualization", position=(0.95, 0.95), font_size=12, color=
    'lightgreen')

# Add labels at specific 3D points to annotate key locations in space.

label_points = [
    [1, 0, 0],    # X axis endpoint
    [0, 1, 0],    # Y axis endpoint
    [0, 0, 1],    # Z axis endpoint
]
labels = ['X-axis', 'Y-axis', 'Z-axis']

plotter.add_labels(label_points, labels, font_size=16, point_size=8.0)

# The clear() method resets the plotter and can be called even after show().
# This allows reusing the same plotter for multiple visualizations.

# Uncomment to clear everything added above and start fresh:
# plotter.show()
# plotter.clear()
# plotter.plot(pv.Cube()) # Would show only a cube instead

# Display the visualization with all customizations.

# Uncomment to show in browser:
plotter.show()
```

Total running time of the script: (0 minutes 0.079 seconds)

## 8.3 Advanced usage examples

These examples show how to use the Visualization Interface Tool to postprocess simulation data.

### 8.3.1 Postprocessing simulation results using the MeshObjectPlot class

The Visualization Interface Tool provides the `MeshObject` helper class to relate a custom object. With a custom object, you can take advantage of the full potential of the Visualization Interface Tool.

This example shows how to use the `MeshObjectPlot` class to plot your custom objects with scalar data on mesh.

#### Necessary imports

```
from ansys.fluent.core import examples
import pyvista as pv

from ansys.tools.visualization_interface.backends.pyvista import PyVistaBackend
from ansys.tools.visualization_interface import MeshObjectPlot, Plotter
```

#### Download the VTK file

A VTK dataset can be produced utilizing `PyDPF` for Ansys Flagship products simulations results file format.

```
mixing_elbow_file_src = examples.download_file("mixing_elbow.vtk", "result_files/fluent-
˓→mixing_elbow_steady-state")
```

#### Define a custom object class

Note that the `CustomObject` class must have a way to get the mesh and a name or ID.

```
class CustomObject:
    def __init__(self):
        self.name = "CustomObject"
        self.mesh = pv.read(mixing_elbow_file_src)

    def get_mesh(self):
        return self.mesh

    def get_field_array_info(self):
        return self.mesh.array_names

    def name(self):
        return self.name

# Create a custom object
custom_vtk = CustomObject()
```

### Create a MeshObjectPlot instance

```
mesh_object = MeshObjectPlot(custom_vtk, custom_vtk.get_mesh())

# Define the camera position
cpos = (
    (-0.3331763564757694, 0.08802797061044923, -1.055269197114142),
    (0.08813476356878325, -0.03975174212669032, -0.012819952697089087),
    (0.045604530283921085, 0.9935979348314435, 0.10336039239608838),
)
```

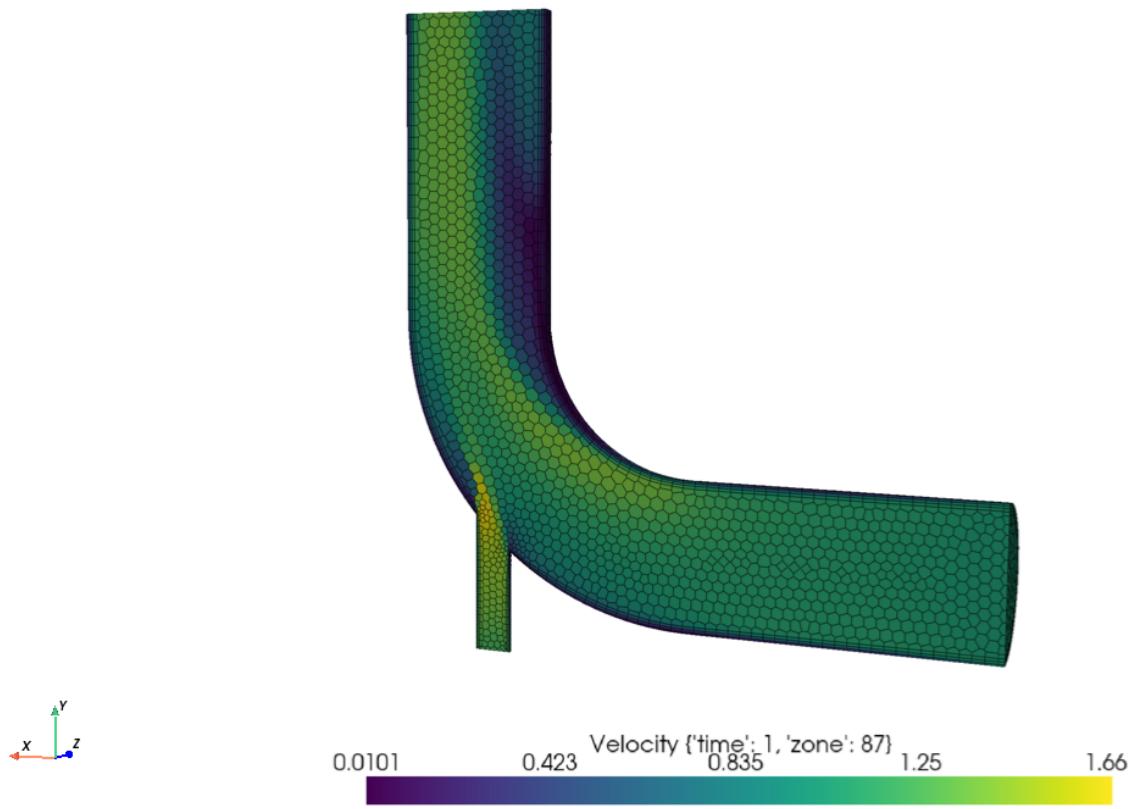
### Get the available field data arrays

```
field_data_arrays = custom_vtk.get_field_array_info()
print(f"Field data arrays: {field_data_arrays}")
```

```
Field data arrays: ["Velocity {'time': 1, 'zone': 87}", "Temperature {'time': 1, 'zone': 87}"]
```

### Plot the MeshObjectPlot instance with mesh object & field data (0)

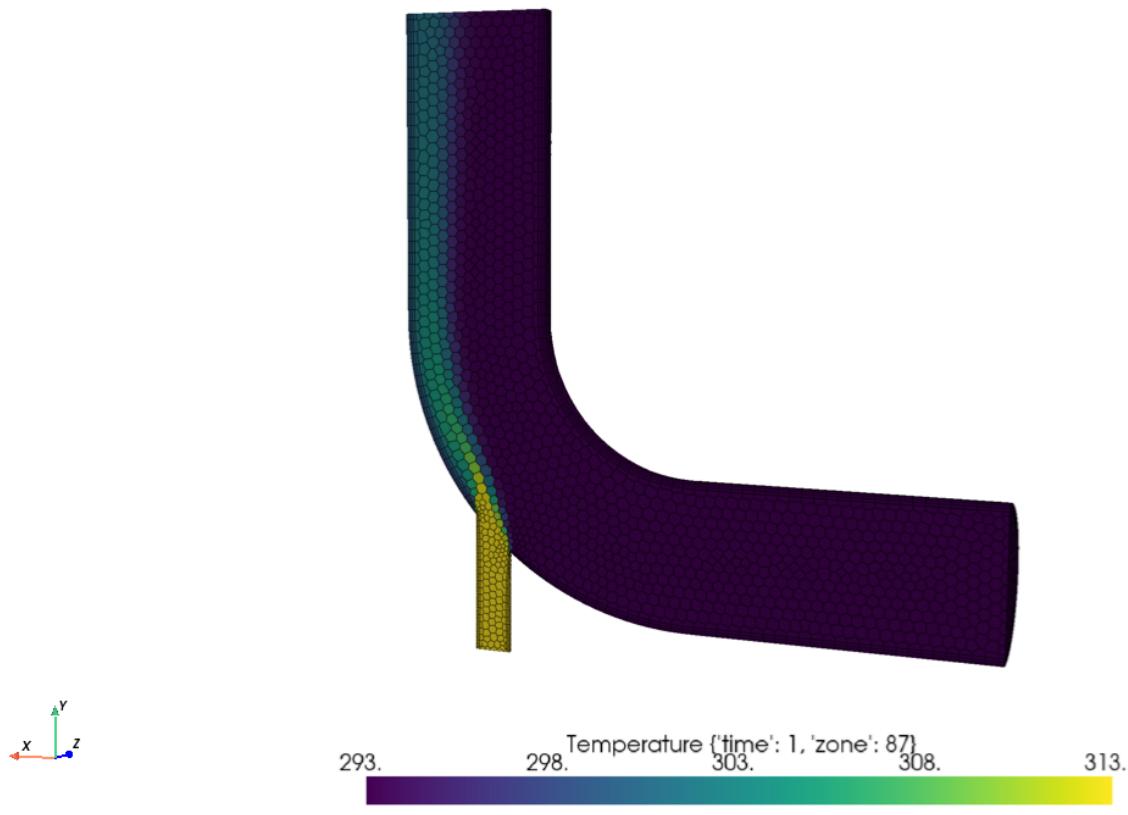
```
pv_backend = PyVistaBackend()
pl = Plotter(backend=pv_backend)
pl.plot(
    mesh_object,
    scalars=field_data_arrays[0],
    show_edges=True,
    show_scalar_bar=True,
)
pl._backend.pv_interface.scene.camera_position = cpos
pl.show()
```



[]

#### Plot the MeshObjectPlot instance with mesh object & other field data (1)

```
pv_backend = PyVistaBackend()
pl = Plotter(backend=pv_backend)
pl.plot(
    mesh_object,
    scalars=field_data_arrays[1],
    show_edges=True,
    show_scalar_bar=True,
)
pl._backend.pv_interface.scene.camera_position = cpos
pl.show()
```



[ ]

**Total running time of the script:** (0 minutes 2.149 seconds)



## CONTRIBUTE

Overall guidance on contributing to a PyAnsys library appears in the [Contributing](#) topic in the *PyAnsys developer's guide*. Ensure that you are thoroughly familiar with this guide before attempting to contribute to the Visualization Interface Tool.

The following contribution information is specific to the Visualization Interface Tool.

### 9.1 Install in developer mode

Installing the Visualization Interface Tool in developer mode allows you to modify and enhance the source.

To clone and install the latest Visualization Interface Tool release in development mode, run these commands:

```
git clone https://github.com/ansys/ansys-tools-visualization-interface
cd ansys-tools-visualization-interface
python -m pip install --upgrade pip
pip install -e .
```

### 9.2 Run tests

The Visualization Interface Tool uses [pytest](#) for testing.

- Prior to running tests, you must run this command to install test dependencies:

```
pip install -e .[tests]
```

- To then run the tests, navigate to the root directory of the repository and run this command:

```
pytest
```

### 9.3 Adhere to code style

The Visualization Interface Tool follows the PEP8 standard as outlined in [PEP 8](#) in the *PyAnsys developer's guide* and implements style checking using [pre-commit](#).

To ensure your code meets minimum code styling standards, run these commands:

```
pip install pre-commit
pre-commit run --all-files
```

You can also install this as a pre-commit hook by running this command:

```
pre-commit install
```

This way, it's not possible for you to push code that fails the style checks:

```
$ pre-commit install
$ git commit -am "added my cool feature"
black.....Passed
blacken-docs.....Passed
isort.....Passed
flake8.....Passed
docformatter.....Passed
codespell.....Passed
pydocstyle.....Passed
check for merge conflicts.....Passed
debug statements (python).....Passed
check yaml.....Passed
trim trailing whitespace.....Passed
Add License Headers.....Passed
Validate GitHub Workflows.....Passed
```

## 9.4 Build the documentation

You can build the Visualization Interface Tool documentation locally.

1. Prior to building the documentation, you must run this command to install documentation dependencies:

```
pip install -e .[doc]
```

2. To then build the documentation, navigate to the docs directory and run this command:

```
# On Linux or macOS
make html

# On Windows
./make.bat html
```

The documentation is built in the docs/\_build/html directory.

You can clean the documentation build by running this command:

```
# On Linux or macOS
make clean

# On Windows
./make.bat clean
```

## 9.5 Post issues

Use the [Visualization Interface Tool Issues](#) page to report bugs and request new features. When possible, use the issue templates provided. If your issue does not fit into one of these templates, click the link for opening a blank issue.

If you have general questions about the PyAnsys ecosystem, email [pyansys.core@ansys.com](mailto:pyansys.core@ansys.com). If your question is specific to the Visualization Interface Tool, ask your question in an issue as described in the previous paragraph.

## PYTHON MODULE INDEX

a

ansys.tools.visualization_interface, 15	35
ansys.tools.visualization_interface.backends, 15	37
ansys.tools.visualization_interface.backends.plotly, 16	38
ansys.tools.visualization_interface.backends.plotly, 21	39
ansys.tools.visualization_interface.backends.plotly_interface, 23	41
ansys.tools.visualization_interface.backends.plotly_widgets, 16	42
ansys.tools.visualization_interface.backends.plotly_widgets.dropdown_manager, 16	43
ansys.tools.visualization_interface.backends.plotly_widgets.dropdown_manager, 20	45
ansys.tools.visualization_interface.backends.pyvista, 27	47
ansys.tools.visualization_interface.backends.pyvista.animation, 57	48
ansys.tools.visualization_interface.backends.pyvista.picker, 62	50
ansys.tools.visualization_interface.backends.pyvista, 65	51
ansys.tools.visualization_interface.backends.pyvista_interface, 78	53
ansys.tools.visualization_interface.backends.pyvista.frame_local, 83	54
ansys.tools.visualization_interface.backends.pyvista.frame_remote, 84	56
ansys.tools.visualization_interface.backends.pyvista.frame_service, 85	97
ansys.tools.visualization_interface.backends.pyvista.widgets, 28	87
ansys.tools.visualization_interface.backends.pyvista.widgets.button, 28	87
ansys.tools.visualization_interface.backends.pyvista.widgets.dark_mode, 30	88
ansys.tools.visualization_interface.backends.pyvista.widgets.frame_arrows, 31	91
ansys.tools.visualization_interface.backends.pyvista.widgets.dynamic_tree_menu, 33	92
ansys.tools.visualization_interface.backends.pyvista.widgets.visualization_types, 87	ansys.tools.visualization_interface.types.edge_plot,
ansys.tools.visualization_interface.backends.pyvista.widgets.visualization_types, 28	ansys.tools.visualization_interface.types.mesh_object_plot
ansys.tools.visualization_interface.backends.pyvista.widgets.visualization_types, 87	ansys.tools.visualization_interface.utils.clip_plane,
ansys.tools.visualization_interface.backends.pyvista.widgets.visualization_types, 91	ansys.tools.visualization_interface.utils.color,
ansys.tools.visualization_interface.backends.pyvista.widgets.visualization_types, 92	ansys.tools.visualization_interface.utils.dynamic_tree_menu,

ansys.tools.visualization\_interface.utils.logger,  
93  
ansys.tools.visualization\_interface.utils.vtkhdf\_converter,  
95

# INDEX

## Symbols

`__call__()` (*in module SingletonType*), 94  
`__len__()` (*in module FrameSequence*), 58  
`__len__()` (*in module InMemoryFrameSequence*), 59  
`__version__` (*in module visualization\_interface*), 105

## A

`actor` (*in module EdgePlot*), 88  
`actor` (*in module MeshObjectPlot*), 90  
`add_button()` (*in module ButtonManager*), 17  
`add_child()` (*in module MeshObjectPlot*), 91  
`add_file_handler()` (*in module VizLogger*), 95  
`add_labels()` (*in module PlotlyBackend*), 27  
`add_labels()` (*in module Plotter*), 104  
`add_labels()` (*in module PyVistaBackend*), 77  
`add_lines()` (*in module PlotlyBackend*), 25  
`add_lines()` (*in module Plotter*), 101  
`add_lines()` (*in module PyVistaBackend*), 75  
`add_measurement_toggle_button()` (*in module ButtonManager*), 19  
`add_mesh_name()` (*in module DashDropdownManager*), 21  
`add_planes()` (*in module PlotlyBackend*), 26  
`add_planes()` (*in module Plotter*), 102  
`add_planes()` (*in module PyVistaBackend*), 76  
`add_points()` (*in module PlotlyBackend*), 25  
`add_points()` (*in module Plotter*), 100  
`add_points()` (*in module PyVistaBackend*), 75  
`add_text()` (*in module PlotlyBackend*), 26  
`add_text()` (*in module Plotter*), 103  
`add_text()` (*in module PyVistaBackend*), 76  
`add_widget()` (*in module PyVistaBackendInterface*), 69  
`animate()` (*in module Plotter*), 98  
`ansys.tools.visualization_interface`  
    `module`, 15  
`ansys.tools.visualization_interface.backends`  
    `module`, 15  
`ansys.tools.visualization_interface.backends.plotly`  
    `module`, 16  
`ansys.tools.visualization_interface.backends.plotly.plotly`  
        `module`, 21  
`ansys.tools.visualization_interface.backends.plotly.plotly`  
        `(built-in class)`, 23  
`ansys.tools.visualization_interface.backends.plotly.widget`  
        `module`, 16  
`ansys.tools.visualization_interface.backends.plotly.widget`  
        `module`, 16  
`ansys.tools.visualization_interface.backends.plotly.widget`  
        `(built-in class)`, 16  
`ansys.tools.visualization_interface.backends.plotly.widget`  
        `module`, 20  
`ansys.tools.visualization_interface.backends.plotly.widget`  
        `(built-in class)`, 20  
`ansys.tools.visualization_interface.backends.pyvista`  
    `module`, 27  
`ansys.tools.visualization_interface.backends.pyvista.animation`  
    `module`, 57  
`ansys.tools.visualization_interface.backends.pyvista.animation`  
    `(built-in class)`, 59  
`ansys.tools.visualization_interface.backends.pyvista.animation`  
    `(built-in class)`, 62  
`ansys.tools.visualization_interface.backends.pyvista.animation`  
    `(built-in class)`, 58  
`ansys.tools.visualization_interface.backends.pyvista.animation`  
    `(built-in class)`, 58  
`ansys.tools.visualization_interface.backends.pyvista.pickling`  
    `module`, 62  
`ansys.tools.visualization_interface.backends.pyvista.pickling`  
    `(built-in class)`, 62  
`ansys.tools.visualization_interface.backends.pyvista.pickling`  
    `(built-in class)`, 64  
`ansys.tools.visualization_interface.backends.pyvista.pyvis`  
    `module`, 65  
`ansys.tools.visualization_interface.backends.pyvista.pyvis`  
    `(built-in class)`, 72  
`ansys.tools.visualization_interface.backends.pyvista.pyvis`  
    `(built-in class)`, 66  
`ansys.tools.visualization_interface.backends.pyvista.pyvis`  
    `plotly`  
        `plotly.dash`  
            `module`, 78



ansys.tools.visualization\_interface.types  
 module, 87

ansys.tools.visualization\_interface.types.edgecallback()  
 module, 87

ansys.tools.visualization\_interface.types.edgecallback()  
 (built-in class), 87

ansys.tools.visualization\_interface.types.meshcallback()  
 module, 88

ansys.tools.visualization\_interface.types.meshcallback()  
 (built-in class), 88

ansys.tools.visualization\_interface.utils  
 module, 91

ansys.tools.visualization\_interface.utils.clipplane  
 module, 91

ansys.tools.visualization\_interface.utils.clipplane  
 (built-in class), 92

ansys.tools.visualization\_interface.utils.colorclear\_plotter()  
 module, 92

ansys.tools.visualization\_interface.utils.colorclipper  
 (built-in class), 93

ansys.tools.visualization\_interface.utils.loggercompute\_edge\_object\_map()  
 module, 93

ansys.tools.visualization\_interface.utils.loggercreate\_dash\_layout()  
 (built-in class), 94

ansys.tools.visualization\_interface.utils.logger.VizLogger  
 (built-in class), 94

ansys.tools.visualization\_interface.utils.vtkhdfstomeshobject  
 module, 95

args\_iso\_view\_button() (in module ButtonManager), 19

args\_projection\_toggle\_button() (in module ButtonManager), 19

args\_theme\_toggle\_button() (in module ButtonManager), 20

args\_xy\_view\_button() (in module ButtonManager), 18

args\_xz\_view\_button() (in module ButtonManager), 18

args\_yz\_view\_button() (in module ButtonManager), 18

**B**

backend (in module Plotter), 98

base\_plotter (in module PyVistaBackend), 73

button\_config (in module Button), 29

**C**

callback() (in module Button), 30

callback() (in module DarkModeButton), 31

callback() (in module DisplacementArrow), 32

callback() (in module DynamicTreeMenuWidget), 35

callback() (in module HideButton), 36

callback() (in module MeasureWidget), 38

callback() (in module MeshSliderWidget), 39

callback() (in module NextButton), 40

callback() (in module ParallelProjectionButton), 42

callback() (in module PickRotCenterButton), 43

callback() (in module PlayPauseButton), 44

callback() (in module EdgePlotter), 57

callback() (in module PreviousButton), 46

callback() (in module Ruler), 48

callback() (in module SaveGifButton), 49

callback() (in module MeshObjectPlotButton), 50

callback() (in module StopButton), 52

callback() (in module TreeMenuToggleButton), 53

callback() (in module ViewButton), 55

callback() (in module DashDropdownManager), 21

clear() (in module PlotlyBackend), 27

clear() (in module PyVistaBackend), 77

CLIENT\_TYPE (in module trame\_local), 84

close() (in module PyVistaBackend), 74

compute\_edge\_object\_map() (in module PyVistaBackendInterface), 70

create\_dash\_layout() (in module PlotlyDashBackend), 74

current\_frame (in module Animation), 60

dfs\_to\_meshobject (in module MeshObjectPlot), 90

**D**

DARK\_MODE\_THRESHOLD (in module animation), 62

DARK\_MODE\_THRESHOLD (in module pyvista), 77

DEFAULT (in module Color), 93

direction (in module DisplacementArrow), 32

direction (in module ViewButton), 55

disable\_center\_focus() (in module PyVistaBackendInterface), 70

disable\_hover() (in module PyVistaBackendInterface), 70

disable\_parallel\_projection() (in module PyVistaInterface), 80

disable\_picking() (in module PyVistaBackendInterface), 70

DOCUMENTATION\_BUILD (in module visualization\_interface), 105

dropdown\_manager (in module PlotlyDashBackend), 22

**E**

EDGE (in module Color), 93

edge\_object (in module EdgePlot), 88

edges (in module MeshObjectPlot), 90

enable\_hover() (in module PyVistaBackendInterface), 70

enable\_output() (in module VizLogger), 95

`enable_parallel_projection()` (*in module* `PyVistaInterface`), 80  
`enable_picking()` (*in module* `PyVistaBackendInterface`), 70  
`enable_set_focus_center()` (*in module* `PyVistaBackendInterface`), 70  
`enable_widgets()` (*in module* `PyVistaBackendInterface`), 69

F

`focus_point_selection()` (in module `PyVistaBack-endInterface`), 70  
`fps` (in module `Animation`), 60

G

`get_frame()` (*in module FrameSequence*), 58  
`get_frame()` (*in module InMemoryFrameSequence*), 59  
`get_logger()` (*in module VizLogger*), 95  
`get_mesh_names()` (*in module DashDropdownManager*), 21  
`get_visibility_args_for_meshes()` (*in module DashDropdownManager*), 21

H

```
    hide_children() (in module PyVistaInterface), 81
    hide_menu() (in module DynamicTreeMenuWidget), 35
    hover_callback() (in module PyVistaBackendInterface),
                      70
    hover_select_object() (in module AbstractPicker),
                          63
    hover_select_object() (in module Picker), 65
    hover_unselect_object() (in module AbstractPicker),
                           63
    hover_unselect_object() (in module Picker), 65
```

1

`is_visible_in_tree()` (*in module* `MeshObjectPlot`), 91  
`ISOMETRIC` (*in module* `ViewDirection`), 56

L

`layout` (*in module PlotlyBackend*), 24  
`logger` (*in module logger*), 95

M

```
mesh (in module MeshObjectPlot), 90
mesh_type (in module MeshObjectPlot), 90
module
    ansys.tools.visualization_inter
    ansys.tools.visualization_inter
        15
    ansys.tools.visualization_inter
```

```
ansys.tools.visualization_interface.backends.plotly.p  
    21  
ansys.tools.visualization_interface.backends.plotly.p  
    23  
ansys.tools.visualization_interface.backends.plotly.wi  
    16  
ansys.tools.visualization_interface.backends.plotly.wi  
    16  
ansys.tools.visualization_interface.backends.plotly.wi  
    20  
ansys.tools.visualization_interface.backends.pyvista,  
    27  
ansys.tools.visualization_interface.backends.pyvista.a  
    57  
ansys.tools.visualization_interface.backends.pyvista.p  
    62  
ansys.tools.visualization_interface.backends.pyvista.p  
    65  
ansys.tools.visualization_interface.backends.pyvista.p  
    78  
ansys.tools.visualization_interface.backends.pyvista.t  
    83  
ansys.tools.visualization_interface.backends.pyvista.t  
    84  
ansys.tools.visualization_interface.backends.pyvista.t  
    85  
ansys.tools.visualization_interface.backends.pyvista.w  
    28  
ansys.tools.visualization_interface.backends.pyvista.w  
    30  
ansys.tools.visualization_interface.backends.pyvista.w  
    31  
ansys.tools.visualization_interface.backends.pyvista.w  
    33  
ansys.tools.visualization_interface.backends.pyvista.w  
    35  
ansys.tools.visualization_interface.backends.pyvista.w  
    37  
ansys.tools.visualization_interface.backends.pyvista.w  
    38  
ansys.tools.visualization_interface.backends.pyvista.w  
    39  
ansys.tools.visualization_interface.backends.pyvista.w  
    41  
ansys.tools.visualization_interface.backends.pyvista.w  
    42  
ansys.tools.visualization_interface.backends.pyvista.w  
    43  
ansys.tools.visualization_interface.backends.pyvista.w  
    45  
ansys.tools.visualization_interface.backends.pyvista.w  
    47
```

```

ansys.tools.visualization_interface.backends.pv_backend.PICKED_WIDGETS_ABOUT_PLOTTER, 3
    48
    picked_dict (in module Picker), 64
ansys.tools.visualization_interface.backends.pv_backend.PICKED_WIDGETS_CROSS_HOVER,
    50
    picked_operation() (in module PyVistaBackendInterface)
ansys.tools.visualization_interface.backends.pyvista.PICKED_WIDGETS_TREE_TOGGLE,
    51
    stop_button, 52
    tree_menu_toggle, 53
    play() (in module Animation), 61
ansys.tools.visualization_interface.backends.pyvista.PAUSE(widgets.ViewPortBottomConfig), 45
    54
    PLAYING (in module AnimationState), 62
ansys.tools.visualization_interface.backends.pyvista.PLOTLY_DASH_BACKEND, 24
    56
    plot() (in module PlotlyDashBackend), 22
ansys.tools.visualization_interface.plotter.plot() (in module Plotter), 98
    97
    plot() (in module PyVistaBackend), 74
ansys.tools.visualization_interface.types.plot() (in module PyVistaBackendInterface), 71
    87
    plot() (in module PyVistaInterface), 82
ansys.tools.visualization_interface.types.plot_edges() (in module PyVistaInterface), 81
    87
    plot_iter() (in module PlotlyBackend), 24
ansys.tools.visualization_interface.types.mesh_object() (in module Plotter), 98
    88
    plot_iter() (in module PyVistaBackend), 73
ansys.tools.visualization_interface.utils.plot_iter() (in module PyVistaBackendInterface), 71
    91
    plot_iter() (in module PyVistaInterface), 82
ansys.tools.visualization_interface.utils.plot_meshobject() (in module PyVistaInterface), 81
    91
    plotter (in module PlotterWidget), 57
ansys.tools.visualization_interface.utils.plotter (in module TrameVisualizer), 84
    92
    plotter_helper (in module MeasureWidget), 37
ansys.tools.visualization_interface.utils.plotter_helper (in module MeshSliderWidget), 39
    93
    plotter_helper (in module PickRotCenterButton), 43
ansys.tools.visualization_interface.utils.PREVIOUS_CONVERTER, PreviousButtonConfig), 47
    95
    pv_interface (in module PyVistaBackendInterface), 69

```

## N

name (in module EdgePlot), 88  
 name (in module MeshObjectPlot), 90  
 NEXT (in module NextButtonConfig), 41  
 normal (in module ClipPlane), 92

## O

object\_to\_actors\_map (in module PyVistaInterface), 80  
 origin (in module ClipPlane), 92

## P

parent (in module EdgePlot), 88  
 parent (in module MeshObjectPlot), 89  
 pause() (in module Animation), 61  
 PAUSED (in module AnimationState), 62  
 pd\_to\_vtkhdf() (in module vtkhdf\_converter), 96  
 pick\_select\_object() (in module AbstractPicker), 63  
 pick\_select\_object() (in module Picker), 64  
 pick\_unselect\_object() (in module AbstractPicker), 63  
 pick\_unselect\_object() (in module Picker), 65  
 PICKED (in module Color), 93

## R

refresh() (in module DynamicTreeMenuItem), 35  
 run() (in module TrameService), 86

## S

save() (in module Animation), 61  
 SAVE\_GIF (in module SaveGifButtonConfig), 49  
 scene (in module PyVistaBackendInterface), 69  
 scene (in module PyVistaInterface), 80  
 seek() (in module Animation), 61  
 send\_mesh() (in module trame\_remote), 85  
 send\_pl() (in module trame\_remote), 85  
 server (in module TrameVisualizer), 84  
 set\_add\_mesh\_defaults() (in module PyVistaInterface), 83  
 set\_level() (in module VizLogger), 95  
 set\_scene() (in module TrameService), 86  
 set\_scene() (in module TrameVisualizer), 84  
 show() (in module Animation), 61  
 show() (in module PlotlyBackend), 25  
 show() (in module PlotlyDashBackend), 23  
 show() (in module Plotter), 98  
 show() (in module PyVistaBackendInterface), 70

show() (in module *PyVistaInterface*), 83  
show() (in module *TrameVisualizer*), 84  
show\_children() (in module *PyVistaInterface*), 81  
show\_hide\_bbox\_dict() (in module *ButtonManager*),  
    17  
show\_menu() (in module *DynamicTreeWidget*), 35  
show\_plotter() (in module *PyVistaBackendInterface*),  
    71  
state (in module *Animation*), 60  
step\_backward() (in module *Animation*), 61  
step\_forward() (in module *Animation*), 61  
STOP (in module *StopButtonConfig*), 52  
stop() (in module *Animation*), 61  
STOPPED (in module *AnimationState*), 62

## T

TESTING\_MODE (in module *visualization\_interface*), 105  
toggle\_subtree\_visibility() (in module *PyVistaInterface*), 81  
total\_frames (in module *Animation*), 60

## U

ug\_to\_vtkhdf() (in module *vtkhdf\_converter*), 96  
update() (in module *Button*), 30  
update() (in module *DarkModeButton*), 31  
update() (in module *DynamicTreeWidget*), 35  
update() (in module *HideButton*), 36  
update() (in module *MeasureWidget*), 38  
update() (in module *MeshSliderWidget*), 39  
update() (in module *ParallelProjectionButton*), 42  
update() (in module *PickRotCenterButton*), 43  
update() (in module *PlotterWidget*), 57  
update() (in module *Ruler*), 48  
update() (in module *ScreenshotButton*), 51  
update() (in module *TreeMenuToggleButton*), 53  
update\_layout() (in module *ButtonManager*), 18  
USE\_HTML\_BACKEND (in module *visualization\_interface*),  
    105  
USE\_TRAME (in module *visualization\_interface*), 105

## V

view\_xy() (in module *PyVistaInterface*), 80  
view\_xz() (in module *PyVistaInterface*), 80  
view\_yx() (in module *PyVistaInterface*), 80  
view\_yz() (in module *PyVistaInterface*), 80  
view\_zx() (in module *PyVistaInterface*), 80  
view\_zy() (in module *PyVistaInterface*), 80  
visible (in module *MeshObjectPlot*), 90  
vtkhdf\_to\_pd() (in module *vtkhdf\_converter*), 96  
vtkhdf\_to\_ug() (in module *vtkhdf\_converter*), 96

## X

XDOWN (in module *CameraPanDirection*), 33

XUP (in module *CameraPanDirection*), 33  
XYMINUS (in module *ViewDirection*), 56  
XYPLUS (in module *ViewDirection*), 56  
XZMINUS (in module *ViewDirection*), 56  
XZPLUS (in module *ViewDirection*), 56

## Y

YDOWN (in module *CameraPanDirection*), 33  
YUP (in module *CameraPanDirection*), 33  
YZMINUS (in module *ViewDirection*), 56  
YZPLUS (in module *ViewDirection*), 56

## Z

ZDOWN (in module *CameraPanDirection*), 33  
ZUP (in module *CameraPanDirection*), 33