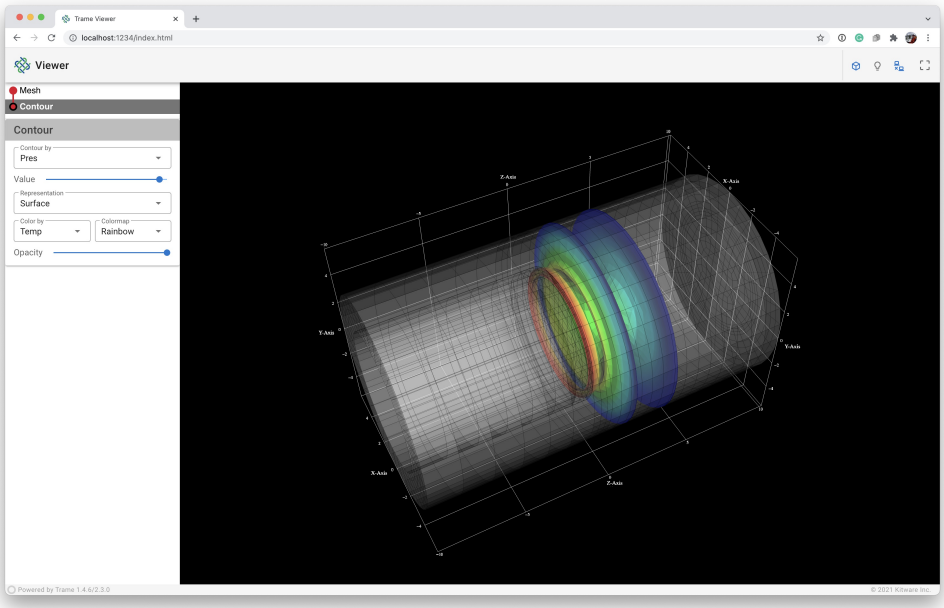
Tutorial of *trame*

We walk you through the basics of ***trame*** and end up creating a simple but full-featured VTK example application.



It is easiest to work on the tutorial from within the ***trame-tutorial*** repository. The tutorial are broken down into the following ordered sections:

1. Download (./tutorial-download.html)
2. Setup (./tutorial-setup-vtk.html)
3. VTK (./tutorial-vtk.html)
4. Layouts (./tutorial-layouts.html)
5. HTML (./tutorial-html.html)
6. Application (./tutorial-application.html)
7. ParaView (./tutorial-paraview.html)

Download *trame*

```
git clone https://github.com/Kitware/trame-tutorial.git
```

There is now a directory called ***trame-tutorial*** with the following file structure:

```
$ tree .
.
└── trame-tutorial
    ├── README.md
    ├── 00_setup
    |   └── app.py
    ├── 01_vtk
    |   ├── app_cone.py
    |   ├── app_flow.py
    |   ├── solution_cone.py
    |   ├── solution_flow.py
    |   └── solution_ray_cast.py
    ├── 02_layouts
    |   ├── app_cone.py
    |   ├── solution_FullScreenPage.py
    |   ├── solution_SinglePage.py
    |   └── solution_SinglePageWithDrawer.py
    ├── 03_html
    |   ├── app_cone.py
    |   ├── solution_buttons_a.py
    |   ├── solution_buttons_b.py
    |   └── solution_final.py
    ├── 04_application
    |   ├── app.py
    |   └── solution.py
    ├── 05_paraview
    |   ├── SimpleCone.py
    |   └── StateLoader.py
    └── data
        ├── carotid.vtk
        ├── disk_out_ref.vtu
        ├── ironProt.vtk
        ├── pv-state-diskout.pvsm
        └── pv-state.pvsm

8 directories, 24 files
```

In the following steps of the tutorial we will assume to be inside the `trame-tutorial` root directory.

Setup environment for VTK

trame requires Python 3.6+ but since ParaView 5.10 is bundling Python 3.9 we should use Python 3.9 for our environment.

```
python3.9 -m venv .venv
source ./.venv/bin/activate
python -m pip install --upgrade pip
pip install "trame"
pip install "vtk>=9.1.0"
```

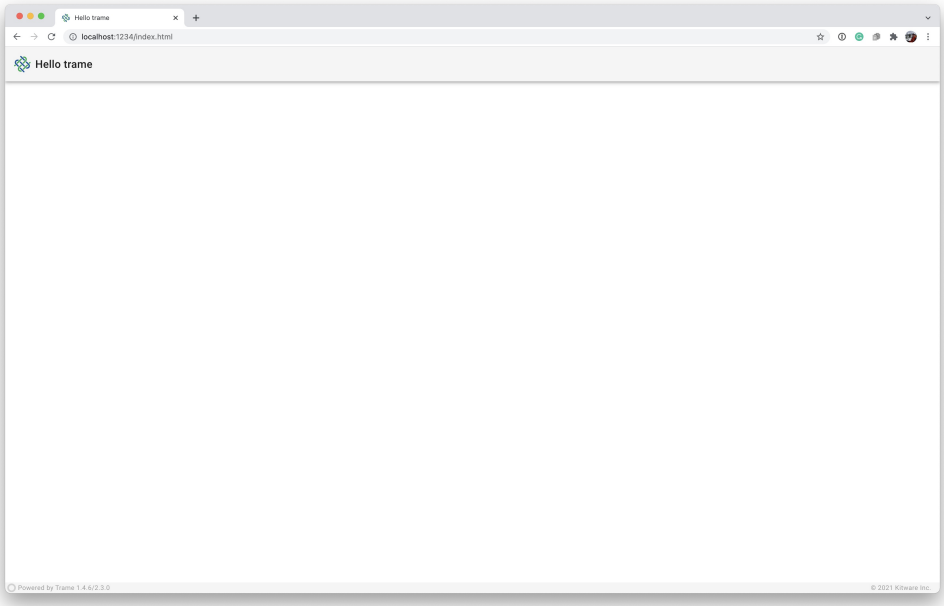
Notes:

- `venv` was added in Python 3.3.
- On mac with Arm architecture, VTK is only available on Python 3.9

Running the application

```
python ./00_setup/app.py --port 1234
```

Your browser should open to `http://localhost:1234/`



Notes:

- The default port is 8080, but since this is very common we will use 1234 for our Tutorial.
- If you are running this on a remote machine, you may have to set the host to 0.0.0.0 to allow any external connection. (`python ./app.py --port 1234 --host 0.0.0.0`)

Annotation of Hello *trame* Application

We start by importing the basic building blocks for our client-server application.

```
from trame.layouts import SinglePage
```

from ***trame***'s `layouts` , we import a skeleton for a single page client application.

Next, we define the graphical user interface (GUI) using a bare minimum of options. We instantiate a `SinglePage` GUI setting the browser tab title as "Hello trame", and then set the GUI `title` text to hold `"Hello trame"` .

```
layout = SinglePage("Hello trame")
layout.title.set_text("Hello trame")
```

Finally, we start the Web server using

```
if __name__ == "__main__":
    layout.start()
```

`start` can take an optional argument of a *port* number. However, this can be set with command-line arguments (`--port 1234`).

Running the Application

```
$ python ./00_setup/app.py --port 1234

 App running at:
 - Local:   http://localhost:1234/
 - Network: http://192.168.1.34:1234/

Note that for multi-users you need to use and configure a launcher.
And to prevent your browser from opening, add '--server' to your command line.
```

Your browser should open automatically to `http://localhost:1234/`

VTK

So we want to start adding VTK visualizations to our application.

VTK Imports

Start editing the file in `01_vtk/app_cone.py` which has the same content as `00_setup/app.py`.

First, what we need to add is an import for `vtk` and `vuetify` from `trame.html`.

```
from trame.html import vtk, vuetify
```

This provides us access to ***trame***'s helper functions for vtk and vuetify.

Next, we will need to import the required objects from vtk. We will visualize a simple cone in this example so we will need `vtkConeSource`.

```
from vtkmodules.vtkFiltersSources import vtkConeSource
```

Other VTK objects will need to be imported based on the desired visualization pipelines.

Next, we need to import the VTK rendering core

```
from vtkmodules.vtkRenderingCore import (
    vtkActor,
    vtkPolyDataMapper,
    vtkRenderer,
    vtkRenderWindow,
    vtkRenderWindowInteractor,
)
```

Finally, we need to import the required modules for the interactor and rendering.

`vtkInteractorStyleSwitch` is required for the interactor initialization.

```
from vtkmodules.vtkInteractionStyle import vtkInteractorStyleSwitch #noqa
```

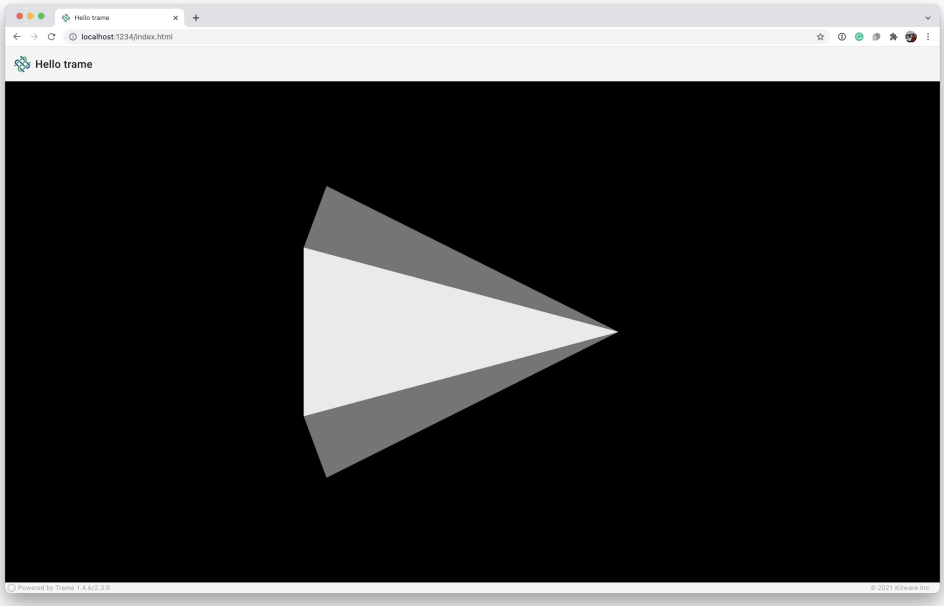
The `vtkRenderingOpenGL2` module is required for the rendering initialization. It is not necessary for local rendering, but you'll want to include it so that you can seamlessly switch between *local* and *remote* rendering.

```
import vtkmodules.vtkRenderingOpenGL2 #noqa
```

Note: `#noqa` tells the linter to ignore checking this problematic line.

VTK Pipeline

As promised, to add visualization to your ***trame*** application you simply write VTK pipelines in Python.



There are a number of ways to learn VTK:

- VTK User Guide (https://www.kitware.com/products/books/VTKUsersGuide.pdf)
- VTK Textbook (https://gitlab.kitware.com/vtk/textbook/raw/master/VTKBook/VTKTextBook.pdf)
- VTK Examples (https://kitware.github.io/vtk-examples/site/Python)
- VTK Documentation (https://www.vtk.org/doc/nightly/html/)

This tutorial will not provide an adequate background for VTK, but we will explain the pieces and parts of the provided exmples at a high level.

First, we create a `vtkRenderer` and `vtkRenderWindow`. The we tie them together by adding the `renderer` to the `renderWindow`.

```
renderer = vtkRenderer()
renderWindow = vtkRenderWindow()
renderWindow.AddRenderer(renderer)
```

Second, we define a `vtkRenderWindowInteractor` which provides a platform-independent interaction mechanism for mouse/key/time events.

```
renderWindowInteractor = vtkRenderWindowInteractor()
renderWindowInteractor.SetRenderWindow(renderWindow)
renderWindowInteractor.GetInteractorStyle().SetCurrentStyleToTrackballCamera()
```

We create the interactor, connect it to the `renderWindow` and set the interaction style.

Third, we create the desired visualization. This process requires the creation of an object, a mapper, and an actor.

```
cone_source = vtkConeSource()
mapper = vtkPolyDataMapper()
mapper.SetInputConnection(cone_source.GetOutputPort())
actor = vtkActor()
actor.SetMapper(mapper)
```

The instantiated `vtkConeSource` is our mesh source that will produce a polydata. Then we want to map it to a graphical representation by creating a `vtkPolyDataMapper`. We must digitally create a connection from the cone to the mapper by setting the input connection. We then create an actor and connect the mapper to the actor.

Finally, we add all the pipelines (actors) to the `renderer`, reset the camera, and render.

```
renderer.AddActor(actor)
renderer.ResetCamera()
```

The VTK specific imports and pipelines defined for a ***trame*** application are precisely the specific imports and pipelines required for a Python VTK script.

Local and Remote Rendering

Why do we care about *local* and *remote* rendering? Well each method of rendering has it's advantages and disadvantaages.

Local Rendering

Advantages

- The server doesn't need a graphics processing unit (GPU). Systems with GPUs are expensive to purchase and expensive to rent (cloud). These costs are pushed to the end-points on end-users.
- The frames per second rendering is higher. Advancements in the browser's access to local GPU resources implies that the performance is nearly as good as available to a desktop application.

Disadvantages

- The data to be rendered must move from the server to the client. This transfer may be too slow and it may be too large.
- Where will the data be processed into graphic primitives? The processing may increase load and latency on the server side or the client side.

Remote Rendering

Advantages

- The data to be rendered doesn't move, only the resulting image.
- Rendering can utilize parallel and distributed processing to handle larger and larger data.
- The server can serve a more diverse set of clients. From cell phone to workstation, their requirements are limited to recieving and rendering images.

Disadvantages

- The frames per second rendering might be capped by the speed and latency of image delivery.
- The servers have to be more capable. Remote software rendering is possible, but the framerates will be further impacted.

Implementation

Down in the GUI section of the application, we **first** need to select a rendering scheme

for *local rendering*

```
html_view = vtk.VtkLocalView(renderWindow)
```

for *remote rendering*

```
html_view = vtk.VtkRemoteView(renderWindow)
```

and define a container to hold the renderer

```
layout.content.children += [
    vuetify.VContainer(
        fluid=True,
        classes="pa-0 fill-height",
        children=[html_view],
    )
]
```

We add a **Vuetify** component to the Web application. In this case, a `VContainer`. The arguments include: fluid (to get full width container), classes (CSS stylings), and children components (the local or remote view).

More information on vuetify (https://vuetifyjs.com/en/introduction/why-vuetify/).

Update and Start

Once the client and server are ready, we need to update the view (`html_view`) by calling `html_view.update()`.

To do it, we revist the `layout` to provide the `on_ready` variable, which takes a function to call when the server and client are ready. So we modify our layout constructor call to look like

```
layout = SinglePage("Hello trame", on_ready=html_view.update)
```

Running the Application

```
python ./01_vtk/app_cone.py --port 1234
```

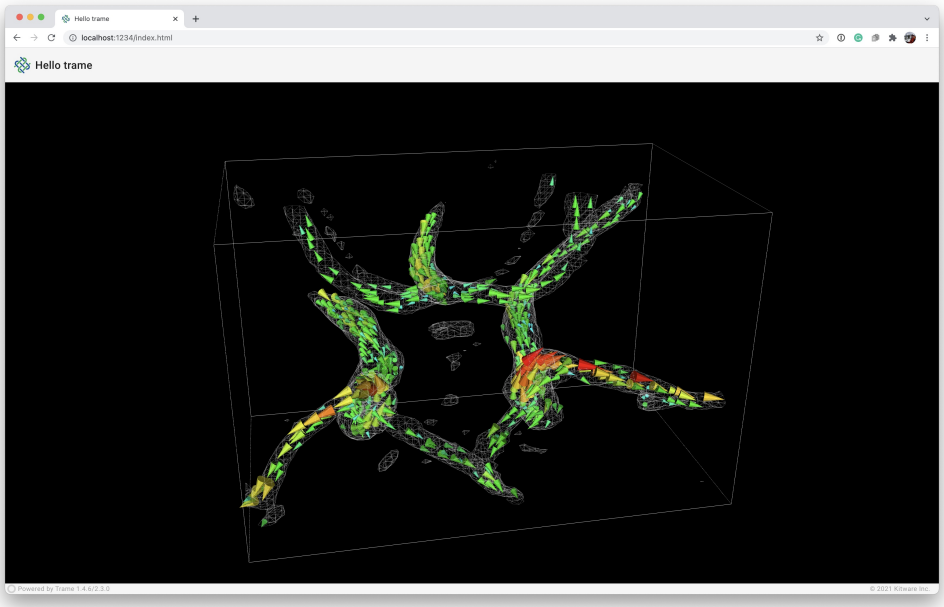
Your browser should open automatically to `http://localhost:1234/`

Interaction

- Rotate: Hold down the mouse and move
- Zoom: Hold down mouse + control key and move up (out) and down (in)
- Pan: Hold down the mouse + option key and move

Other VTK Examples

Now you can take most of the code examples at VTK Examples (https://kitware.github.io/vtk-examples/site/Python) and port them to ***trame***.



(https://kitware.github.io/vtk-examples/site/Python/VisualizationAlgorithms/CarotidFlowGlyphs/)

We are going to implement CarotidFlowGlyphs (https://kitware.github.io/vtk-examples/site/Python/VisualizationAlgorithms/CarotidFlowGlyphs/) by editing the file in `01_vtk/app_flow.py.py` which start from our cone rendering example solution.

Imports

We are going to read a file. So we will want to import the `os` module and set the current directory. Starting with our Hello ***trame*** cone application, we add

```
import os

CURRENT_DIRECTORY = os.path.abspath(os.path.dirname(__file__))
```

We replaced the imports for our cone VTK pipeline

```
from vtkmodules.vtkFiltersSources import vtkConeSource
```

with imports for multiple VTK pipelines

```
from vtkmodules.vtkCommonColor import vtkNamedColors
from vtkmodules.vtkCommonCore import vtkLookupTable
from vtkmodules.vtkFiltersCore import (
    vtkContourFilter,
    vtkGlyph3D,
    vtkMaskPoints,
    vtkThresholdPoints
)
from vtkmodules.vtkFiltersModeling import vtkOutlineFilter
from vtkmodules.vtkFiltersSources import vtkConeSource
from vtkmodules.vtkIOLegacy import vtkStructuredPointsReader
```

to create three-dimensional (3D) contours of the carotid artery, 3D Glyphs of the flow field at various points, and an outline of the computational domain.

Pipelines

First, we replaced the cone pipeline line 36-40

```
cone_source = vtkConeSource()
mapper = vtkPolyDataMapper()
mapper.SetInputConnection(cone_source.GetOutputPort())
actor = vtkActor()
actor.SetMapper(mapper)
```

With our three pipelines for the glyphs, contours, and outline.

We read the data using the `vtkStructuredPointsReader` and the full file path, provided by adding `CURRENT_DIRECTORY` to the beginning of the relative file path.

```
# Read the Data

reader = vtkStructuredPointsReader()
reader.SetFileName(os.path.join(CURRENT_DIRECTORY, "../data/carotid.vtk"))
```

```
# Glyphs
threshold = vtkThresholdPoints()
threshold.SetInputConnection(reader.GetOutputPort())
threshold.ThresholdByUpper(200)

mask = vtkMaskPoints()
mask.SetInputConnection(threshold.GetOutputPort())
mask.SetOnRatio(5)

cone = vtkConeSource()
cone.SetResolution(11)
cone.SetHeight(1)
cone.SetRadius(0.25)

cones = vtkGlyph3D()
cones.SetInputConnection(mask.GetOutputPort())
cones.SetSourceConnection(cone.GetOutputPort())
cones.SetScaleFactor(0.4)
cones.SetScaleModeToScaleByVector()

lut = vtkLookupTable()
lut.SetHueRange(.667, 0.0)
lut.Build()

scalarRange = [0] * 2
cones.Update()
scalarRange[0] = cones.GetOutput().GetPointData().GetScalars().GetRange()[0]
scalarRange[1] = cones.GetOutput().GetPointData().GetScalars().GetRange()[1]

vectorMapper = vtkPolyDataMapper()
vectorMapper.SetInputConnection(cones.GetOutputPort())
vectorMapper.SetScalarRange(scalarRange[0], scalarRange[1])
vectorMapper.SetLookupTable(lut)

vectorActor = vtkActor()
vectorActor.SetMapper(vectorMapper)

# Contours
iso = vtkContourFilter()
iso.SetInputConnection(reader.GetOutputPort())
iso.SetValue(0, 175)

isoMapper = vtkPolyDataMapper()
isoMapper.SetInputConnection(iso.GetOutputPort())
isoMapper.ScalarVisibilityOff()

isoActor = vtkActor()
isoActor.SetMapper(isoMapper)
isoActor.GetProperty().SetRepresentationToWireframe()
isoActor.GetProperty().SetOpacity(0.25)

# Outline
colors = vtkNamedColors()

outline = vtkOutlineFilter()
outline.SetInputConnection(reader.GetOutputPort())

outlineMapper = vtkPolyDataMapper()
outlineMapper.SetInputConnection(outline.GetOutputPort())

outlineActor = vtkActor()
outlineActor.SetMapper(outlineMapper)
outlineActor.GetProperty().SetColor(colors.GetColor3d("White"))
```

Note: In the Colors and Data section we instantiate a helper for accessing named colors and read the desired data file on the **server**.

Finally, we replace the single cone pipeline actor

```
renderer.AddActor(actor)
```

with the three pipeline actors to the renderer.

```
renderer.AddActor(outlineActor)
renderer.AddActor(vectorActor)
renderer.AddActor(isoActor)
```

Our pipelines are the same pipelines used in VTK Examples (https://kitware.github.io/vtk-examples/site/Python) except for some cosmetic edits (I like `renderer` and `renderWindow` instead of `ren1` and `renWin`).

Running the Application

```
python ./01_vtk/app_flow.py --port 1234
```

Your browser should open automatically to `http://localhost:1234/`

Ray Casting example

If you want on your own time you can try to implement the Volume Rendering example Simple Ray Cast (https://kitware.github.io/vtk-examples/site/Python/VolumeRendering/SimpleRayCast/) using **trame**.

The solution of that example is available under `01_vtk/solution_ray_cast.py`.

Layouts

To simplify creation of the graphical user interface (GUI) for the web application, ***trame*** defines layouts such as `FullScreenPage`, `SinglePage`, and `SinglePageWithDrawer`.

All core layouts start with a `VApp` (Vuetifies `v-app`) component. The `VApp` is **REQUIRED** for all applications. It is the mount point for other Vuetify components and functionality and ensures that it propagates the default application variant (dark/light) to children components while ensuring proper cross-browser support for certain click events in browsers like Safari. `VApp` should only be rendered within your application **ONCE**.

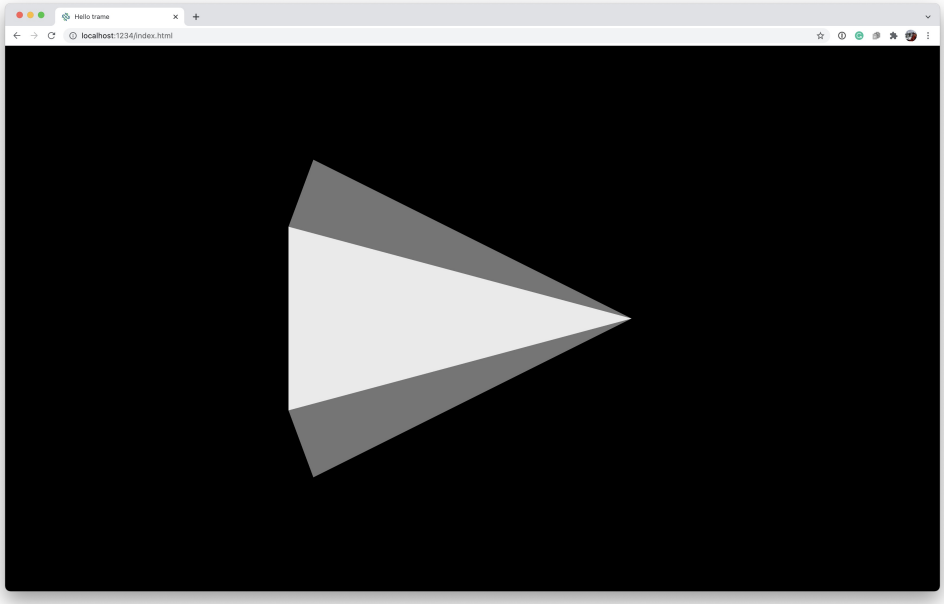
More on Vuetify in the next chapter of the tutorial.

Each of these layouts can be utilized by importing, instantiating, and serving it via its `start` function.

FullScreenPage

If you want to experiement with it you can edit `02_layouts/app_cone.py` which was the latest cone example we built using the `SinglePage` layout.

The `FullScreenPage` layout starts with VApp and exposes the children array where one could add in other desired HTML elements (Vuetify UI Components).



First, add the import to `FullScreenPage` .

```
from trame.layouts import FullScreenPage
```

Second, we instantiate the `layout` object, create the `vtk.VtkLocalView` component, and add it directly to the `VApp` 's `children` using a Vuetify `VContainer` with arguments descripted in the VTK (tutorial-vtk.html) chapter.

```
html_view = vtk.VtkLocalView(renderWindow)

layout = FullScreenPage("Hello trame", on_ready=html_view.update))

layout.children += [
    vuetify.VContainer(
        fluid=True,
        classes="pa-0 fill-height",
        children=[html_view],
    )
]
```

Finally, we call start on the `layout` directly

```
if __name__ == "__main__":
    layout.start()
```

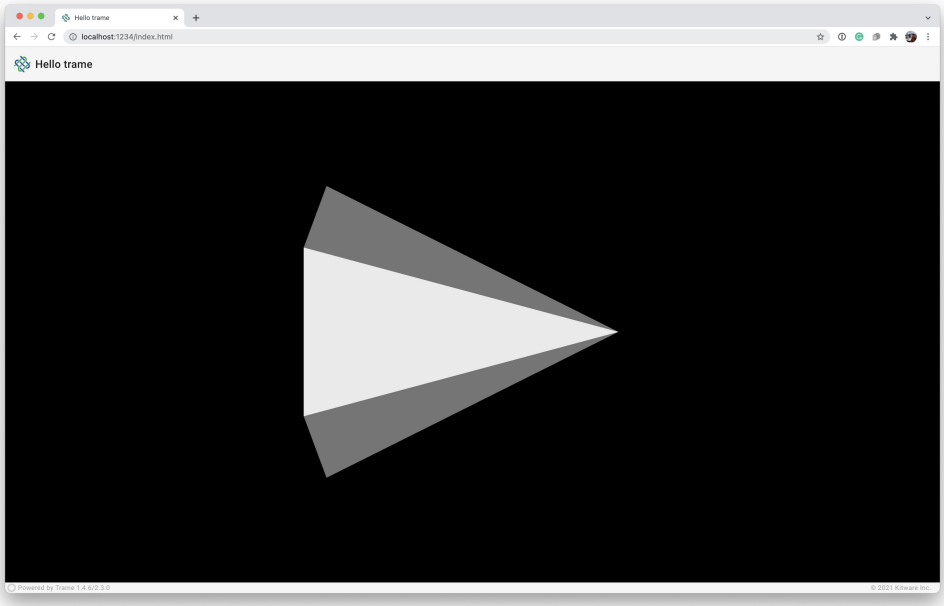
The `FullScreenPage` is really a blank canvas to add your desired Vuetify components.

Running the Application

```
python 02_layouts/app_cone.py --port 1234
# or
python 02_layouts/solution_FullScreenPage.py --port 1234
```

SinglePage

The `SinglePage` extends the `FullScreenPage` with a few predifined components such as *logo*, *title*, *toolbar*, *content*, and *footer*.



The *logo* and *title* sit on the left-hand side of the *toolbar* and customized as necessary. The logo accepts an 32x32 image or an VIcon such as those found at Material Design Icons (https://materialdesignicons.com/). The *toolbar* itself exposes its *children* array where one can add components as needed. The *footer* can be hidden, but currently has ***trame*** branding and the progress bar. The *content* has a *children* array to which you may add your desired Vuetify components.

First, we import the `SinglePage` class.

```
from trame.layouts import SinglePage
```

Second, we instantiate the `layout` object, maybe change the title, create the `vtk.VtkLocalView` component, and add it to the `content` component's `children` using a Vuetify `VContainer` with arguments descripted in the VTK () chapter.

```
html_view = vtk.VtkLocalView(renderWindow)

layout = SinglePage("Hello trame", on_ready=html_view.update)
layout.title.set_text("Hello trame")

layout.content.children += [
    vuetify.VContainer(
        fluid=True,
        classes="pa-0 fill-height",
        children=[html_view],
    )
]
```

Finally, we serve the `layout` via the `start` function.

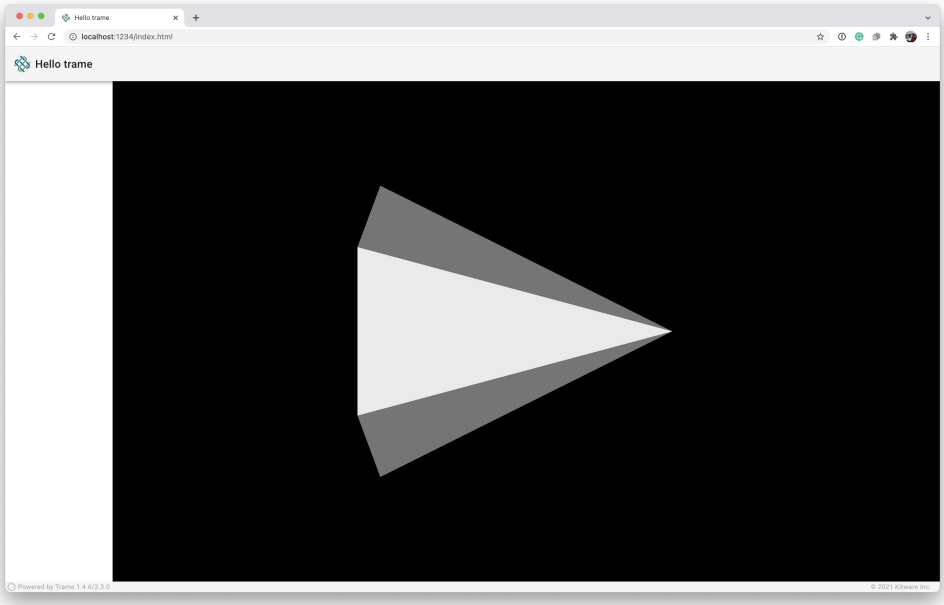
```
if __name__ == "__main__":
    layout.start()
```

Running the Application

```
python 02_layouts/app_cone.py --port 1234
# or
python 02_layouts/solution_SinglePage.py --port 1234
```

SinglePageWithDrawer

The `SinglePageWithDrawer` extends the `SinglePage` with a *drawer*. You can show and hide the *drawer* by clicking on the application *logo* on the *toolbar*. The *drawer* has a *children* array to which you may add necessary Vuetify components.



First, we import the `SinglePageWithDrawer` class.

```
from trame.layouts import SinglePageWithDrawer
```

Second, we instantiate the `layout` object with access to everything within the `SinglePage` layout.

```
html_view = vtk.VtkLocalView(renderWindow)


layout = SinglePageWithDrawer("Hello trame", on_ready=html_view.update)
layout.title.set_text("Hello trame")

layout.content.children += [
    vuetify.VContainer(
        fluid=True,
        classes="pa-0 fill-height",
        children=[html_view],
    )
]
```

Finally, we serve the `layout` via the `start` function.

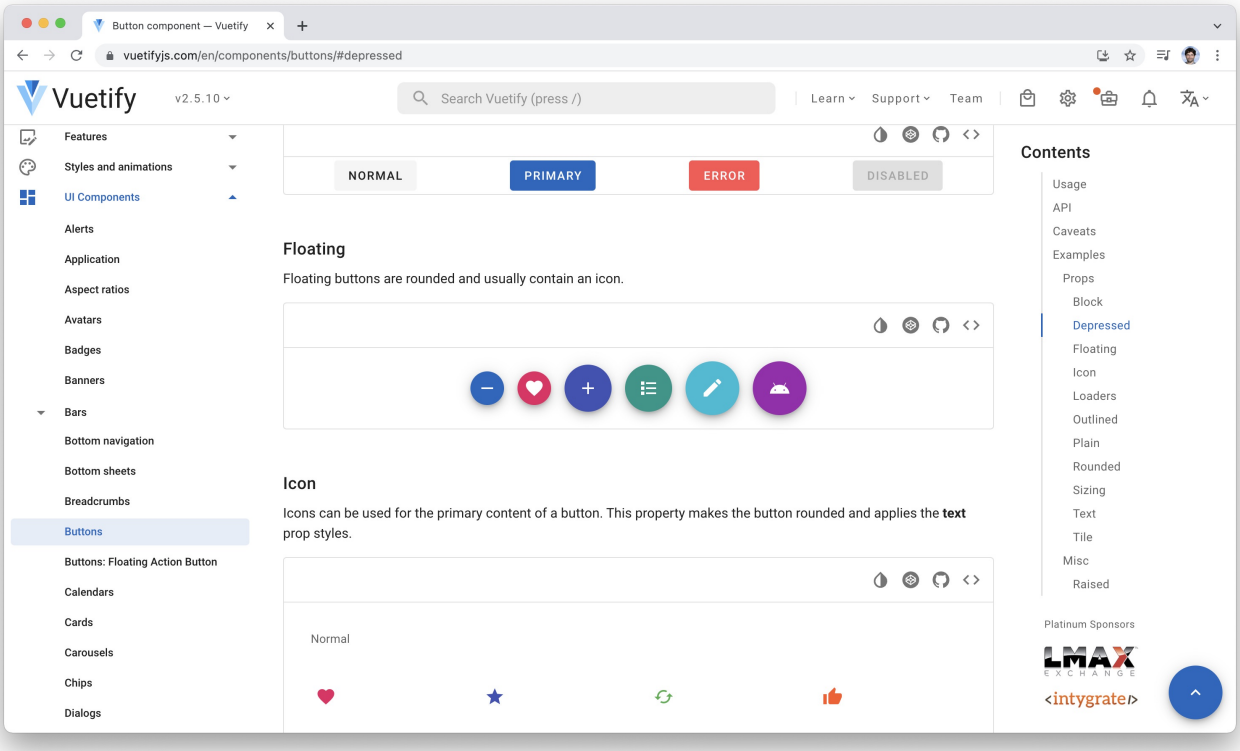
```
if __name__ == "__main__":
    layout.start()
```

Running the Application

```
python 02_layouts/app_cone.py --port 1234
# or
python 02_layouts/solution_SinglePageWithDrawer.py --port 1234
```

HTML

trame leverages Vuetify as its primary UI Component Library for defining HTML graphics user interfaces (GUI). Vuetify (https://vuetifyjs.com/en/introduction/why-vuetify/#what-is-vuetify3f) is a mature, efficient, and expansive framework for good-looking web applications with the same simple state management system as ***trame***. ***trame*** makes Vuetify available in your Python with minimal overhead.



(https://vuetifyjs.com/en/)

Using Vuetify

We expose all Vuetify components. As an example, let's look at how we would make a simple text box. This is taken from Vuetify's excellent examples and documentation (https://vuetifyjs.com/en/components/text-fields/), which we recommend you consult while writing frontends with ***trame***.

```
// Somewhere in javascript
const currentSuffix = "lbs";
const myWeight = 28.0;


<!-- Somewhere in html -->
<v-text-field label="Weight" v-model="myWeight" :suffix="currentSuffix"></v-text-field>
```

Here we have a vuetify text field (`v-text-field`). In Vue, the `v-model` is a directive that provides two-way data binding between an input and form data or between two components. The variable `myWeight` is bound by the `v-model` attribute, so the shared state can read from it (shown in the GUI form) and write to it (input to the form stored as the variable contents).

We've included, optionally, a `label` and a `suffix` for the text box. The `label` is a static string or title, and the `suffix` could be a static string, but the " `:` " in `:suffix` means we will look up and use the contents of a variable `currentSuffix` . This variable could change to 'kg' if our user prefers the metric system.

Looking through the Vuetify documentation, we see a large number of wonderful user interface (UI) components. ***trame*** exposes Vuetify from within Python. Access to Vuetify is provided through ***trame*** using the following import.

```
from trame.html import vuetify
```

Python Vuetify Rules

Exposing Vuetify in Python was accomplished by making a few syntax changes.

1. We use CamelCase in our Python component's name, while attribute hyphens become underscores. For example, the `v-text-field` component becomes VTextField, and the `v-model` attribute becomes `v_model` .
2. Strings, ints, floats, and booleans used to set attributes are assigned as normal like `vuetify.VTextField(label="myLabel")` for the `"myLabel"` String.
3. Variables used to set attributes are surrounded by parenthesis like `vuetify.VTextField(label=("myLabel",))` . The comma is used to provide an initial value like `vuetify.VTextField(label=("myLabel", "Initial Label"))` .
4. Vuetify implicitly sets boolean properties. For example, if something is to be `disabled` , then one simply writes disabled. In our Python implementation, this is done explicitly like `vuetify.VTextField(disabled=True)` .
5. For events, HTML uses the `@` like `@click="runMethod"` to set the function to call upon a click event and double quotes on the String name of the function to run. In our Python version of Vuetify, we ignore the `@` and use the reference to the function instead of a the String name of the function call like `vuetify.VBtn(click=runMethod)` .

Given these rules, we can recreate the JavaScript/HTML text field example in ***trame*** as follows.

```
field = VTextField(
    label="Weight",
    v_model=("myWeight",28),
    suffix=("currentSuffix","lbs"),
)
```

State

In both the previous statements `v_model` and `suffix`, we defined and initialized state variables. These variables are available from both the client and server side.

First, we need to import two more functions from ***trame***, `get_state` and `update_state`.

```
from trame import get_state, update_state
```

From here, we have two capabilities

- `get_state` - returns the value of a given state variable.
- `update_state` - intializes, if not previously defined, or updates a state variable.

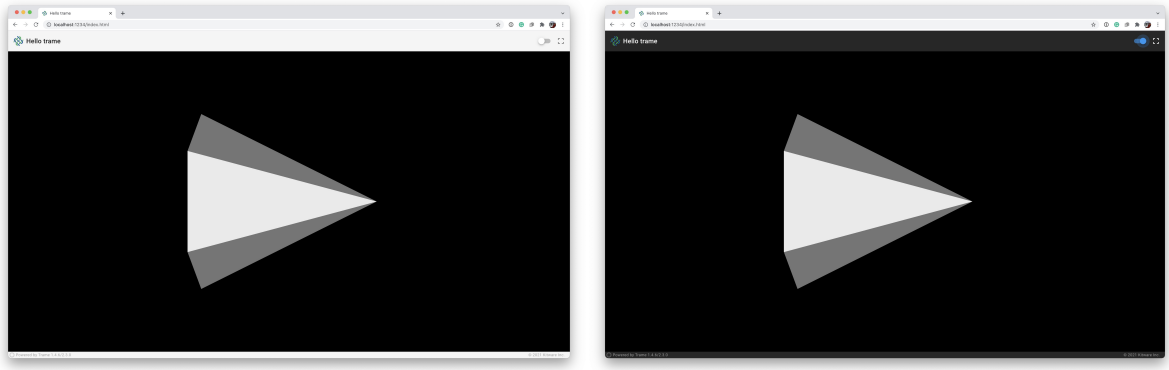
Let's look at an example leveraging the previously defined text field.

```
def increment_weight():
    w, = get_state("myWeight")
    w += 1
    update_state("myWeight", w)

def set_metric():
    w,s = get_state("myWeight","currentSuffix")
    w = 0.453592 * w
    s = "kg"
    update_state("myWeight", w)
    update_state("currentSuffix", s)

def set_imperial():
    w,s = get_state("myWeight","currentSuffix")
    w = 2.20462 * w
    s = "lb"
    update_state("myWeight", w)
    update_state("currentSuffix", s)
```

In the `increment_weight` function, we use the `get_state` function to access the `"myWeight"` value. Notice that get_state returns a list, so the comma in `w,` is necessary. We then increment the weight. Finally, we update the state variable with the `update_state` function.

GUI

Let's modify the Hello ***trame*** application to add some GUI elements by starting editing the file `03_html/app_cone.py`.



So with the `SinglePage` layout, we could add UI elements to either the `toolbar`, `content`, or the `footer`. We'll limit ourself to the `toolbar`, but the proceedure is still the same. Let us add to the right side of the `toolbar` a switch to toggle between light and dark mode of the application and a button to reset the view after panning and/or zooming.

- The VSpacer Vuetify component pushes the extra space on the left side of the component.
- The VSwitch component toggles between two different states. In this case, we will update a Vuetify variable `$vuetify.theme.dark`. The hide_details and dense attribute creates a smaller, tighter switch.
- The VBtn component is a button. We decorate the button with a VIcon component where the argument is a String identifying the Material Design Icons (https://materialdesignicons.com/) instead of text in this case. The VBtn icon attribute provides proper sizing and padding for the icon. Finally, the click attribute tells the application what method to call when the button is pressed. In this case, we use an internal ***trame*** function, `$refs.view.resetCamera()`.

Note:

- A ref (reference) is made by `vtk.VtkLocalView(renderWindow)` or `vtk.VtkRemoteView(renderWindow)`. By default, `ref="view"`. If you would like to change its name or add additional views, one can use `vtk.VtkLocalView(renderWindow, ref="newViewName")`.

We add all the Vuetify components in a *flow* from left to right, top to bottom to the `layout.toolbar.children` array.

```
layout.toolbar.children += [
    vuetify.VSpacer(),
    vuetify.VSwitch(
        v_model="$vuetify.theme.dark",
        hide_details=True,
        dense=True,
    ),
    vuetify.VBtn(
        vuetify.VIcon("mdi-crop-free"),
        icon=True,
        click="$refs.view.resetCamera()",
    ),
]
```

Running the Application

```
python 03_html/app_cone.py --port 1234
# or
python 03_html/solution_buttons_a.py --port 1234
```

Your browser should open automatically to `http://localhost:1234/`

`with` Construct

The Python `with` construct can be used in our Vuetify GUI creation to make the code cleaner and much more readable. You use the `with` construct to add components to a component. The `toolbar` is a component, so we can add the `VSpacer`, `VSwitch`, and `VBtn`. The `VBtn` is a component that we want to decorate with a `VIcon` component, so we use the with construct with the `VBtn` to accomplish this effect.

```
with layout.toolbar:
    vuetify.VSpacer()
    vuetify.VSwitch(
        v_model="$vuetify.theme.dark",
        hide_details=True,
        dense=True,
    )
    with vuetify.VBtn(icon=True, click="$refs.view.resetCamera()"):
        vuetify.VIcon("mdi-crop-free")
```

In addition, the `content` can be modified to add a `VContainer` component as follows.

```
with layout.content:
    vuetify.VContainer(
        fluid=True,
        classes="pa-0 fill-height",
        children=[html_view],
    )
```

We think it's easy to see that utilizing the `with` construct is much more Pythonic and creates clean readable code, but use either coding style according to your preferences.

Note:

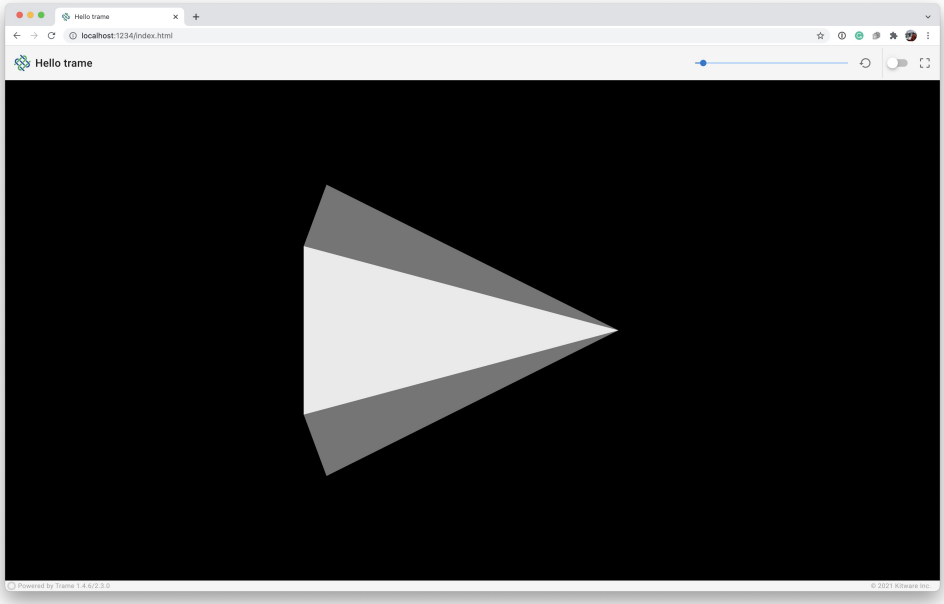
- When using `with` instantiating any `trame.html.AbstractElement` will add it to the children of the element of the with.

Running the Application

```
python 03_html/app_cone.py --port 1234
# or
python 03_html/solution_buttons_b.py --port 1234
```

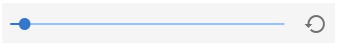
Callbacks

We really want to enable our GUI to interact with our visualization (or application, in general). For example, we want to adjust the `resolution` (number of line segments) that approximates circle used in defining the cone.



By default, the `resolution` is 6, defined in the Globals section.

```
DEFAULT_RESOLUTION = 6
```



Let's add a `VSlider` for adjusting the resolution, a `VBtn` with `VIcon` to reset the resolution to the default value, and a vertical `VDivider` to separate our visualization GUI from the application GUI. The following is added after the `VSpacer` component at the beginning of the `with toolbar` *flow*.

```
with layout.toolbar:
    vuetify.VSpacer()
    vuetify.VSlider(
        v_model=("resolution", DEFAULT_RESOLUTION),
        min=3,
        max=60,
        step=1,
        hide_details=True,
        dense=True,
        style="max-width: 300px",
    )
    with vuetify.VBtn(icon=True, click=reset_resolution):
        vuetify.VIcon("mdi-restore")
    vuetify.VDivider(vertical=True, classes="mx-2")

    vuetify.VSwitch(
        v_model="$vuetify.theme.dark",
        hide_details=True,
        dense=True,
    )
    with vuetify.VBtn(icon=True, click="$refs.view.resetCamera()"):
        vuetify.VIcon("mdi-crop-free")
```

The `VSlider` creates `resolution` as a state variable and is initialized to the default resolution. When interacting with the slider, the code will call a function decorated with `@change("resolution")` .

```
@change("resolution")
def update_resolution(resolution, **kwargs):
    cone_source.SetResolution(resolution)
    html_view.update()
```

There is no need to get or update the `resolution` state variable. This update is carried out on the client-side by the v_model. We simply update the `cone_source` appropriately and update the view.

The `VBtn` resets the the resolution when pressed by calling the `reset_resolution` function. This is a `trigger` event, where `v_models` are `change` events. Since, we use the function reference here, there is no need to use a `@trigger("...")` decorator here. It is created by default behind the scene.

```
def reset_resolution():
    update_state("resolution", DEFAULT_RESOLUTION)
```

Note:

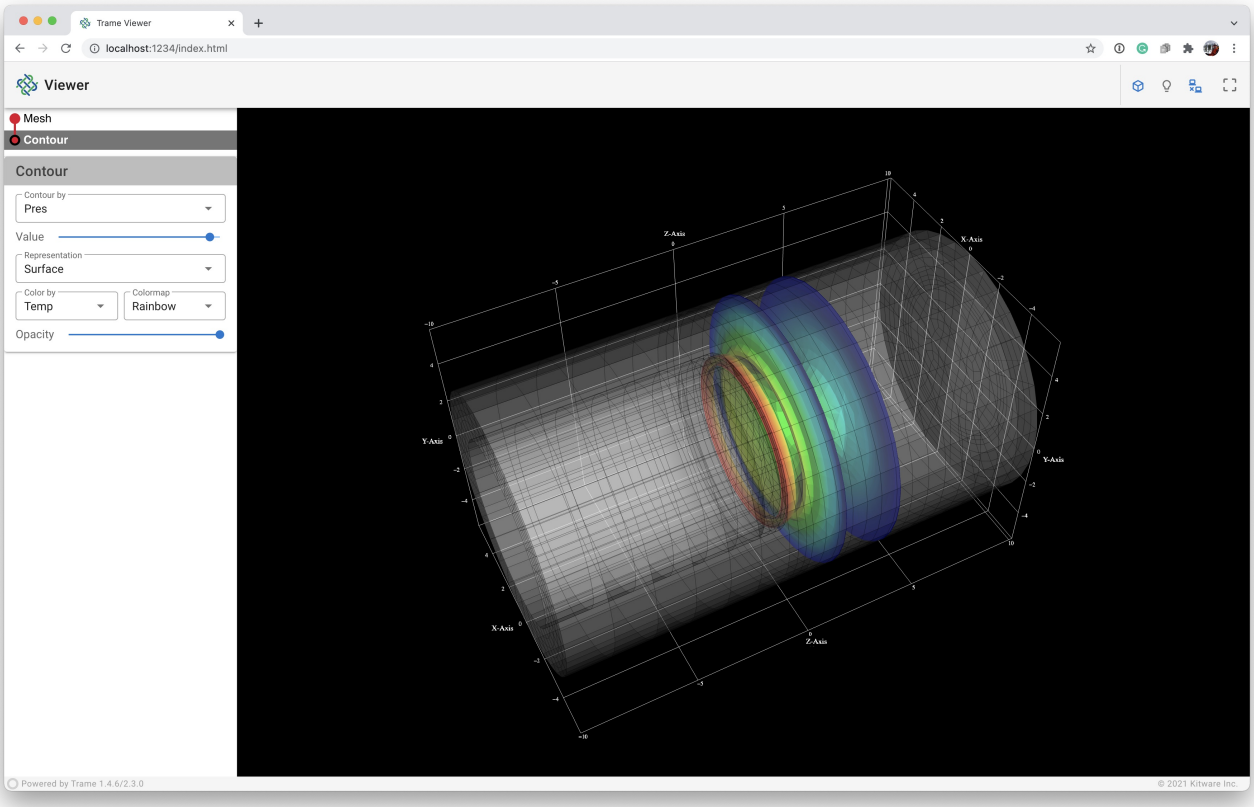
- If you plan to pass arguments to the `trigger` function, then you would use the decorator.
- In this case because we listen to `resolution` change, the call to `update_state("resolution", ...)` will also trigger the change callback. That is the reason why we do not need to update the view or the cone source resolution in `reset_resolution()`.

Both of these functions should be included in the Functions or Callbacks section of the code.

Running the Application

```
python 03_html/app_cone.py --port 1234
# or
python 03_html/solution_final.py --port 1234
```

Application



We will create a more complete example application that will show how to use several parts the ***trame*** library. Developing a trame application requires the following coding steps:

1. Imports (#imports-id) for appropriate ***trame*** and vtk modules
2. Create the necessary VTK pipelines (#vtk_pipeline-id)
3. Build ***trame*** Views (#trame_view-id)
4. Define the the GUI (#gui-id) components required for interaction
5. Develop the callbacks (#callbacks-id) for the GUI components
6. Start (#start-id) the application

We will start by editing `04_application/app.py` which contain the basic structure of a **trame** app with the VTK rendering code base.

Imports

First, our ***trame*** imports have also changed. Thus, we will replace

```
from trame.layouts import SinglePage
from trame.html import vtk, vuetify
```

with

```
from trame.layouts import SinglePageWithDrawer
from trame.html import vtk, vuetify, widgets
```

We are creating a single page application with a drawer (`trame.layouts`), and we want to use one of ***trame***'s predefined widgets (`trame.html`) for displaying and interacting with visualization pipelines.

Finally, our VTK pipelines are fairly straight forward, but not available as one of the vtk examples. We will add the import for our VTK objects.

```
from vtkmodules.vtkCommonDataModel import vtkDataObject
from vtkmodules.vtkFiltersCore import vtkContourFilter
from vtkmodules.vtkIOXML import vtkXMLUnstructuredGridReader
from vtkmodules.vtkRenderingAnnotation import vtkCubeAxesActor
```

to create a three-dimensional (3D) mesh, a contour, and a cube axes legend to outline the computational domain.

VTK Pipelines

First, we read the data using the `vtkXMLUnstructuredGridReader` and the full file path, provided by adding `CURRENT_DIRECTORY` to the beginning of the relative file path.

```
# Read Data
reader = vtkXMLUnstructuredGridReader()
reader.SetFileName(os.path.join(CURRENT_DIRECTORY, "../data/disk_out_ref.vtu"))
reader.Update()
```

Second, we determine the available data arrays and build an array of dictionaries, `dataset_arrays` that contains the array name, value or id, range, and type (point or cell data). We also define the default array and the default minimum and maximum of the default array.

```
# Extract Array/Field information
dataset_arrays = []
fields = [
    (reader.GetOutput().GetPointData(), vtkDataObject.FIELD_ASSOCIATION_POINTS),
    (reader.GetOutput().GetCellData(), vtkDataObject.FIELD_ASSOCIATION_CELLS),
]
for field in fields:
    field_arrays, association = field
    for i in range(field_arrays.GetNumberOfArrays()):
        array = field_arrays.GetArray(i)
        array_range = array.GetRange()
        dataset_arrays.append(
            {
                "text": array.GetName(),
                "value": i,
                "range": list(array_range),
                "type": association,
            }
        )
default_array = dataset_arrays[0]
default_min, default_max = default_array.get("range")
```

The `dataset_arrays` and defaults are used in several places of the initial pipelines, but we plan to enable switching between the available arrays for coloring and contouring.

Third, we create the *mesh* pipeline, which is simply a `vtkDataSetMapper` and a `vtkActor` that we add to the `renderer`.

```
# Mesh
mesh_mapper = vtkDataSetMapper()
mesh_mapper.SetInputConnection(reader.GetOutputPort())
mesh_actor = vtkActor()
mesh_actor.SetMapper(mesh_mapper)
renderer.AddActor(mesh_actor)
```

We want to be able to change the mesh representation, so we get a handle on the `mesh_actor` property and set the representation to surface, set the default point size, and turn off edge visibility.

```
# Mesh: Setup default representation to surface
mesh_actor.GetProperty().SetRepresentationToSurface()
mesh_actor.GetProperty().SetPointSize(1)
mesh_actor.GetProperty().EdgeVisibilityOff()
```

We also want to be able to change the mesh color map, so we get a handle on the mesh's color lookup table and set the hue, saturation, and value ranges to create a rainbow color map, and build it.

```
# Mesh: Apply rainbow color map
mesh_lut = mesh_mapper.GetLookupTable()
mesh_lut.SetHueRange(0.666, 0.0)
mesh_lut.SetSaturationRange(1.0, 1.0)
mesh_lut.SetValueRange(1.0, 1.0)
mesh_lut.Build()
```

We finally want to be able to change the array or field to color by, so we get a handle on the mesh's mapper and set the array to color by and the range of the array to the defaults, tell the mapper to use the appropriate scalar mode for the array, turn on the scalar visibility, and tell the mapper to use the scalar range assigned to the lookup table.

```
# Mesh: Color by default array
mesh_mapper.SelectColorArray(default_array.get("text"))
mesh_mapper.GetLookupTable().SetRange(default_min, default_max)
if default_array.get("type") == vtkDataObject.FIELD_ASSOCIATION_POINTS:
    mesh_mapper.SetScalarModeToUsePointFieldData()
else:
    mesh_mapper.SetScalarModeToUseCellFieldData()
mesh_mapper.SetScalarVisibility(True)
mesh_mapper.SetUseLookupTableScalarRange(True)
```

Fourth, we create the *contour* pipeline, which uses a `vtkContourFilter`, a `vtkDataSetMapper`, and a `vtkActor` that we add to the `renderer`.

```
# Contour
contour = vtkContourFilter()
contour.SetInputConnection(reader.GetOutputPort())
contour_mapper = vtkDataSetMapper()
contour_mapper.SetInputConnection(contour.GetOutputPort())
contour_actor = vtkActor()
contour_actor.SetMapper(contour_mapper)
renderer.AddActor(contour_actor)
```

We want to be able to change the array or field to contour by, so we define the initial value as the midpoint between the default minimum and maximum of the default array, set the default array as the input array, and set the computed contour value.

```
# Contour: ContourBy default array
contour_value = 0.5 * (default_max + default_min)
contour.SetInputArrayToProcess(
    0, 0, 0, default_array.get("type"), default_array.get("text")
)
contour.SetValue(0, contour_value)
```

We want to be able to change the contour representation, the contour color map, and the array to color by, so we initialize these defaults as we did for the *mesh* pipeline.

```
# Contour: Setup default representation to surface
contour_actor.GetProperty().SetRepresentationToSurface()
contour_actor.GetProperty().SetPointSize(1)
contour_actor.GetProperty().EdgeVisibilityOff()

# Contour: Apply rainbow color map
contour_lut = contour_mapper.GetLookupTable()
contour_lut.SetHueRange(0.666, 0.0)
contour_lut.SetSaturationRange(1.0, 1.0)
contour_lut.SetValueRange(1.0, 1.0)
contour_lut.Build()

# Contour: Color by default array
contour_mapper.SelectColorArray(default_array.get("text"))
contour_mapper.GetLookupTable().SetRange(default_min, default_max)
if default_array.get("type") == vtkDataObject.FIELD_ASSOCIATION_POINTS:
    contour_mapper.SetScalarModeToUsePointFieldData()
else:
    contour_mapper.SetScalarModeToUseCellFieldData()
contour_mapper.SetScalarVisibility(True)
contour_mapper.SetUseLookupTableScalarRange(True)
```

Fifth, we create the cube axes, which is a `vtkCubeAxesActor` that we add to the `renderer`.

```
# Cube Axes
cube_axes = vtkCubeAxesActor()
renderer.AddActor(cube_axes)
```

The `vtkCubeAxesActor` requires initialization by the bounds of the domain and the `renderer` camera.

```
# Cube Axes: Boundaries, camera, and styling
cube_axes.SetBounds(mesh_actor.GetBounds())
cube_axes.SetCamera(renderer.GetActiveCamera())
cube_axes.SetXLabelFormat("%6.1f")
cube_axes.SetYLabelFormat("%6.1f")
cube_axes.SetZLabelFormat("%6.1f")
cube_axes.SetFlyModeToOuterEdges()
```

Finally, we reset the `renderer` camera to initialize the view.

```
renderer.ResetCamera()
```

trame Views

For this application, we want to enable dynamic switching between *local* and *remote* rendering. Thus, we need to set up both a *local* and *remote* ***trame*** views, and set the `html_view` to initially be the `local_view`.

```
local_view = vtk.VtkLocalView(renderWindow)
remote_view = vtk.VtkRemoteView(renderWindow, interactive_ratio=(1,))
html_view = local_view
```

Note: The `interactive_ratio` implies the desired image reduction fraction to enhance the frames per second during interaction. Here we set it to 1, which means the image will not be reduced.

GUI

We are creating a single page application with a drawer using `SinglePageWithDrawer`. By defaukIt we get a title, toolbar, drawer, and a content section. So we instantiate a `SinglePageWithDrawer` with the `title` of "Viewer" and `on_ready` argument equal to `html_view.update`, which updates the three-dimensional visualization.

```
layout = SinglePageWithDrawer("Viewer", on_ready=html_view.update)
layout.title.set_text("Viewer")

with layout.toolbar:
    # toolbar components
    pass

with layout.drawer as drawer:
    # drawer components
    pass

with layout.content:
    # content components
    pass
```

Note: The `layout.drawer as drawer` syntax is used to get a reference to the drawer to set some of the drawer's properties

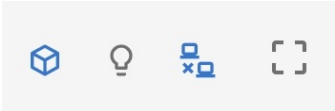
We can fill in the content with a `VContainer` and attaching the `html_view` as the zeroth child in the content's `children` array. This code creates the visualization space as we've done in all our previous examples.

```
with layout.content:
    # content components
    vuetify.VContainer(
        fluid=True,
        classes="pa-0 fill-height",
        children=[html_view],
    )
```

Toolbar GUI

We want to create a toolbar with the application logo and title on one end and some standard buttons separated by a vertical divider on the other end. The `VSpacer` is used to push the content the right-side of the application toolbar, and the `VDivider` is used to create the vertical divder, and method `standard_buttons` encapsulates the gui code to produce these buttons.

```
with layout.toolbar:
    # toolbar components
    vuetify.VSpacer()
    vuetify.VDivider(vertical=True, classes="mx-2")
    standard_buttons()
```



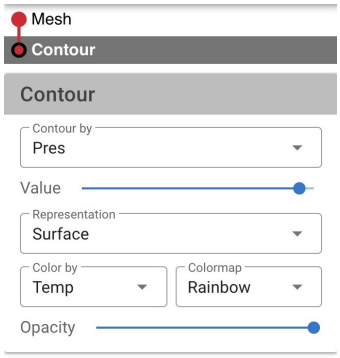
Three of the buttons have on/off states, so we will use a `VCheckbox` with the `on_icon` and `off_icon` properties and the `v_model` callback to switch between these states. The `resetCamera` button is a special case, so we use a `VButton` with the `on_click` callback to reset the camera.

```
def standard_buttons():
    vuetify.VCheckbox(
        v_model=("cube_axes_visibility", True),
        on_icon="mdi-cube-outline",
        off_icon="mdi-cube-off-outline",
        classes="mx-1",
        hide_details=True,
        dense=True,
    )
    vuetify.VCheckbox(
        v_model="$vuetify.theme.dark",
        on_icon="mdi-lightbulb-off-outline",
        off_icon="mdi-lightbulb-outline",
        classes="mx-1",
        hide_details=True,
        dense=True,
    )
    vuetify.VCheckbox(
        v_model=("local_vs_remote", True),
        on_icon="mdi-lan-disconnect",
        off_icon="mdi-lan-connect",
        classes="mx-1",
        hide_details=True,
        dense=True,
    )
    with vuetify.VBtn(icon=True, click="$refs.view.resetCamera()"):
        vuetify.VIcon("mdi-crop-free")
```

The `resetCamera` button and the `dark` checkbox are as they were in previous examples using built-in ***trame*** (`$refs.view.resetCamera()`) and *Vuetify* (`$vuetify.theme.dark`) callbacks.

The `local_vs_remote` checkbox is used to switch between the *local* and *remote* rendering, and leverages the `local_vs_remote` callback (#toolbar_callbacks_local_vs_remote-id). The `cube_axes_visibility` checkbox is used to turn on and off the cube axes, and leverages the `cube_axes_visibility` callback (#toolbar_callbacks_cube_axes_visibility-id).

Drawer GUI



We want to create a drawer with the ***trame*** pipeline widget, a horizontal divider, and pipeline cards. The pipeline cards are shown corresponding to the sected pipeline in the ***trame*** pipeline widget. We need create state for the active pipeline card, so we add this to the layouts state.

```
# State use to track active ui card
layout.state = {
    "active_ui": None,
}
```

We want a little wider `drawer`, so we set the width to 325 pixels. Next, we add the ***trame*** pipeline widget using the `pipeline_widget` function. Then, we use a `VDivider` to separate the ***trame*** widget from the pipeline cards. Finally, we add the pipeline cards using the `mesh_card` and `contour_card` functions.

```
with layout.drawer as drawer:
    # drawer components
    drawer.width = 325
    pipeline_widget()
    vuetify.VDivider(classes="mb-2")
    mesh_card()
    contour_card()
```

Pipeline Widget

The ***trame*** widgets are easy to use. Here, we simply create the widget, define the pipelines and their relationships, and define references to the pipeline widgets two callback functions.

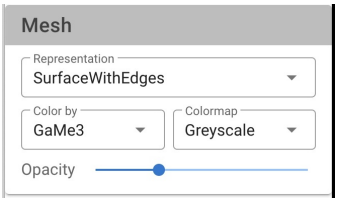
```
def pipeline_widget():
    widgets.GitTree(
        sources=(
            "pipeline",
            [
                {"id": "1", "parent": "0", "visible": 1, "name": "Mesh"},
                {"id": "2", "parent": "1", "visible": 1, "name": "Contour"},
            ],
        ),
        actives_change=(actives_change, "[$event]"),
        visibility_change=(visibility_change, "[$event]"),
    )
```

Default `ui_card`

The default GUI card is a simple card with a title, and a body. The card itself is shown (`v_show`) if the `ui_name` is the same as `active_ui` . We use the `VCard` to create the card, and the `VCardTitle` to create the card title, and the `VCardText` to create the space to add individual pipeline GUI components. The stylings for the card colors can be found at Material Design (https://materializecss.com/color.html).

```
def ui_card(title, ui_name):
    with vuetify.VCard(v_show=f"active_ui == '{ui_name}'"):
        vuetify.VCardTitle(
            title,
            classes="grey lighten-1 py-1 grey--text text--darken-3",
            style="user-select: none; cursor: pointer",
            hide_details=True,
            dense=True,
        )
        content = vuetify.VCardText(classes="py-2")
    return content
```

`mesh_card`

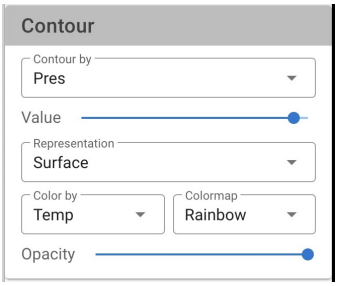


The `mesh_card` containes strictly default GUI components of a pipeline. First, a dropdown menu for the visual representation (#gui_representation-id) type. Next, a row with two columns. One contains a dropdown menu for the array/field to color by (#gui_color_by-id), and the other a dropdown menu for which color map (#gui_color_map-id) to use. Finally, a slider to control the opacity (#gui_opacity-id).

```
def mesh_card():
    with ui_card(title="Mesh", ui_name="mesh"):
        vuetify.VSelect(
            # Representation
        )
        with vuetify.VRow(classes="pt-2", dense=True):
            with vuetify.VCol(cols="6"):
                vuetify.VSelect(
                    # Color By
                )
            with vuetify.VCol(cols="6"):
                vuetify.VSelect(
                    # Color Map
                )
        vuetify.VSlider(
            # Opacity
        )
```

Since these are defaul pipeline elements, we will cover these items together with the contour_card individual components below (#default_components-id).

`contour_card`



The `contour_card` contains a dropdown menu to select the array/field to contour by (#contour-by-gui), a slider to control the contour value (#contour-value-gui), and the default GUI components of a pipeline.

```
def contour_card():
    with ui_card(title="Contour", ui_name="contour"):
        vuetify.VSelect(
            # Contour By
        )
        vuetify.VSlider(
            # Contour Value
        )
    *   vuetify.VSelect(
    *       # Representation
    *   )
    *   with vuetify.VRow(classes="pt-2", dense=True):
    *       with vuetify.VCol(cols="6"):
    *           vuetify.VSelect(
    *               # Color By
    *           )
    *       with vuetify.VCol(cols="6"):
    *           vuetify.VSelect(
    *               # Color Map
    *           )
    *   vuetify.VSlider(
    *       # Opacity
    *   )
```

For simplicity, we will cover these items the contour individual components below (#contour_components-id).

Note: * denotes essentially duplicate code\components from the `mesh_card` .

Default Components

The default components of a pipeline include the representation (#gui_representation-id) selection, the color by (#gui_color_by-id) selection, the color map (#gui_color_map-id) selection, and the opacity (#gui_opacity-id) slider.

Representation GUI

First, we created a constant class `Representation` to enumerate the different representations.

```
class Representation:
    Points = 0
    Wireframe = 1
    Surface = 2
    SurfaceWithEdges = 3
```

Second, we created a dropdown menu for the representation type. The `VSelect` component is used to create a dropdown menu. The `v_model` uses the state variable `mesh_representation` initialized to be a surface. The `items` is a list of tuples, where the first element is the display string (`text`) of the representation, and the second element is the value of the representation used for selection.

```
vuetify.VSelect(
    # Representation
    v_model=("mesh_representation", Representation.Surface),
    items=(
        "representations",
        [
            {"text": "Points", "value": 0},
            {"text": "Wireframe", "value": 1},
            {"text": "Surface", "value": 2},
            {"text": "SurfaceWithEdges", "value": 3},
        ],
    ),
    label="Representation",
    hide_details=True,
    dense=True,
    outlined=True,
    classes="pt-1",
)
```

The `update_mesh_representation` (#drawer_callbacks_update_mesh_representation-id) callback is used to update the representation of the mesh.

The dropdown menu for the representation type for the contour pipeline is similar to the mesh pipeline, but replace the `v_model` line with the `v_model` line for the contour pipeline.

```
vuetify.VSelect(
    # Representation
    v_model=("contour_representation", Representation.Surface),
    # ... same as mesh ...
```

The `update_contour_representation` (#drawer_callbacks_update_contour_representation-id) callback is used to update the representation of the contour.

Color By GUI

We created a dropdown menu for the color by. The `VSelect` component is used to create a dropdown menu. The `v_model` uses the state variable `mesh_color_array_idx` initialized to be the default_array, 0. The `items` is a list of tuples, where the first element is the display string (`text`) of the array name, and the second element is the value of the array used for selection. For `items` , we use the state variable `array_list` to create the list of arrays initialized by our `dataset_arrays` array of dictionaries we created at read time.

```
vuetify.VSelect(
    # Color By
    label="Color by",
    v_model=("mesh_color_array_idx", 0),
    items=("array_list", dataset_arrays),
    hide_details=True,
    dense=True,
    outlined=True,
    classes="pt-1",
)
```

The `update_mesh_color_by_name` (#drawer_callbacks_update_mesh_color_by_name-id) callback is used to update the color by array of the mesh.

The dropdown menu for the color by of the contour pipeline is similar to the mesh pipeline, but replace the `v_model` line with the `v_model` line for the contour pipeline.

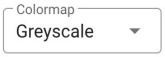
```
vuetify.VSelect(
    # Color By
    v_model=("contour_color_array_idx", 0),
    # ... same as mesh ...
```

The `update_contour_representation` (#drawer_callbacks_update_contour_color_by_name-id) callback is used to update the color by array of the contour.

Color Map GUI

First, we created a constant class `LookupTable` to enumerate the different color maps.

```
class LookupTable:
    Rainbow = 0
    Inverted_Rainbow = 1
    Greyscale = 2
    Inverted_Greyscale = 3
```



Second, we created a dropdown menu for the color map. The `VSelect` component is used to create a dropdown menu. The `v_model` uses the state variable `mesh_color_preset` initialized to be the rainbow color map. The `items` is a list of tuples, where the first element is the display string (`text`) of the color map, and the second element is the value of the color map used for selection.

```
vuetify.VSelect(
    # Color Map
    label="Colormap",
    v_model=("mesh_color_preset", LookupTable.Rainbow),
    items=(
        "colormaps",
        [
            {"text": "Rainbow", "value": 0},
            {"text": "Inv Rainbow", "value": 1},
            {"text": "Greyscale", "value": 2},
            {"text": "Inv Greyscale", "value": 3},
        ],
    ),
    hide_details=True,
    dense=True,
    outlined=True,
    classes="pt-1",
)
```

The `update_mesh_color_preset` (#drawer_callbacks_update_mesh_color_preset-id) callback is used to update the color map of the mesh.

The dropdown menu for the color map of the contour pipeline is similar to the mesh pipeline, but replace the `v_model` line with the `v_model` line for the contour pipeline.

```
vuetify.VSelect(
    # Color Map
    v_model=("contour_color_preset", LookupTable.Rainbow),
    # ... same as mesh ...
```

The `update_contour_color_preset` (#drawer_callbacks_update_contour_color_preset-id) callback is used to update the color map of the contour.

Opacity GUI

We created a slider for the opacity. The `VSlider` component is used to create a slider. The `v_model` uses the state variable `mesh_opacity` initialized to 1.0. The `min` is set to 0 and the `max` is set to 1. The `step` is set to 0.1.

```
vuetify.VSlider(
    # Opacity
    v_model=("mesh_opacity", 1.0),
    min=0,
    max=1,
    step=0.1,
    label="Opacity",
    classes="mt-1",
    hide_details=True,
    dense=True,
)
```

The `update_mesh_opacity` (#drawer_callbacks_update_mesh_opacity-id) callback is used to update the color map of the mesh.

The slider for the opacity of the contour pipeline is similar to the mesh pipeline, but replace the `v_model` line with the `v_model` line for the contour pipeline.

```
vuetify.VSlider(
    # Opacity
    v_model=("contour_opacity", 1.0),
    # ... same as mesh ...
```

The `update_contour_opacity` (#drawer_callbacks_update_contour_opacity-id) callback is used to update the opacity of the contour.

Contour Components

Contour By GUI

We created a dropdown menu for the contour by. The `VSelect` component is used to create a dropdown menu. The `v_model` uses the state variable `contour_by_array_idx` initialized to be the default_array, 0. The `items` is a list of tuples, where the first element is the display string (`text`) of the array name, and the second element is the value of the array used for selection. For `items` , we use the state variable `array_list` to create the list of arrays initialized by our `dataset_arrays` array of dictionaries we created at read time.

```
vuetify.VSelect(
    # Contour By
    label="Contour by",
    v_model=("contour_by_array_idx", 0),
    items=("array_list", dataset_arrays),
    hide_details=True,
    dense=True,
    outlined=True,
    classes="pt-1",
)
```

The `update_contour_by` (#drawer_callbacks_update_contour_by-id) callback is used to update the contour by array of the contour.

Contour Value GUI

We created a slider for the contour value. The `VSlider` component is used to create a slider. The `v_model` uses the state variable `contour_value` initialized to mid-point of the selected contour by array. The `min` is set to minimum of the contour by array and the `max` is set to maximum of the contour by array. The `step` is set to 0.01 times the range of the contour by array.


```
vuetify.VSlider(
    # Contour Value
    v_model=("contour_value", contour_value),
    min=("contour_min", default_min),
    max=("contour_max", default_max),
    step=("contour_step", 0.01 * (default_max - default_min)),
    label="Value",
    classes="my-1",
    hide_details=True,
    dense=True,
)
```

The `update_contour_value` (#drawer_callbacks_update_contour_value-id) callback is used to update the value of the contour.

Callbacks

Toolbar Callbacks

The first toolbar callback is the `cube_axes_visibility` callback, which is used to turn on and off the cube axes. The `update_cube_axes_visibility` function is found by the @change decorator for `cube_axes_visibility` . Then we simply set the `cube_axes` actor's `Visibility` property to the value of `cube_axes_visibility` , and update the view.

```
@change("cube_axes_visibility")
def update_cube_axes_visibility(cube_axes_visibility, **kwargs):
    cube_axes.SetVisibility(cube_axes_visibility)
    html_view.update()
```

The second toolbar callback is the `local_vs_remote` callback, which is used to switch between the local and remote visualization. The `update_local_vs_remote` function is found by the @change decorator for `local_vs_remote` . Then we simply switch between the `local_view` and `remote_view` to the appropriate view. The complicated part is updating the layout by changing out the view in the `layout.content` . The child at `children[0]` is the `VContainer` and the second `children[0]` is the location of the `html_view` in the containers children. Finally, we update the view either shipping geometry (*local* rendering) or an image (*remote* rendering) to the client.

```
@change("local_vs_remote")
def update_local_vs_remote(local_vs_remote, **kwargs):
    # Switch html_view
    global html_view
    if local_vs_remote:
        html_view = local_view
    else:
        html_view = remote_view

    # Update layout
    layout.content.children[0].children[0] = html_view
    layout.flush_content()

    # Update View
    html_view.update()
```

Drawer Callbacks

Pipeline Widget Callbacks

When a pipeline is selected in the pipeline widget, we want to update the pipeline card to show in the drawer. Using the default `actives_change` function of ***trame*** pipeline widget, we simply update `layout.state` element `active_ui` to display the pipeline card of the selected pipeline.

```
# Selection Change
def actives_change(ids):
    _id = ids[0]
    if _id == "1":  # Mesh
        update_state("active_ui", "mesh")
    elif _id == "2":  # Contour
        update_state("active_ui", "contour")
    else:
        update_state("active_ui", "nothing")
```

When a pipeline visibility is toggled in the pipeline widget, we want to update the view to add or remove the toggled pipeline. Using the default `visibility_change` function of ***trame*** pipeline widget, we update the visibility of the appropriate pipeline `actor` using `actor.SetVisibility` function with the visibility accessed from the `event["visible"]` dictionary element.

```
# Visibility Change
def visibility_change(event):
    _id = event["id"]
    _visibility = event["visible"]

    if _id == "1":  # Mesh
        mesh_actor.SetVisibility(_visibility)
    elif _id == "2":  # Contour
        contour_actor.SetVisibility(_visibility)
    html_view.update()
```

Representation Callbacks

The `update_representation` function updates the `representation` property of an `actor` to the value of `mode` .

```
# Representation Callbacks
def update_representation(actor, mode):
    property = actor.GetProperty()
    if mode == Representation.Points:
        property.SetRepresentationToPoints()
        property.SetPointSize(5)
        property.EdgeVisibilityOff()
    elif mode == Representation.Wireframe:
        property.SetRepresentationToWireframe()
        property.SetPointSize(1)
        property.EdgeVisibilityOff()
    elif mode == Representation.Surface:
        property.SetRepresentationToSurface()
        property.SetPointSize(1)
        property.EdgeVisibilityOff()
    elif mode == Representation.SurfaceWithEdges:
        property.SetRepresentationToSurface()
        property.SetPointSize(1)
        property.EdgeVisibilityOn()
```

The `update_mesh_representation` function is found by the @change decorator for `mesh_representation`. We simply call the `update_representation` function with the `mesh_actor` and the `mesh_representation` state, and then update the view.

```
@change("mesh_representation")
def update_mesh_representation(mesh_representation, **kwargs):
    update_representation(mesh_actor, mesh_representation)
    html_view.update()
```

Likewise, the `update_contour_representation` function is found by the @change decorator for `contour_representation`. We simply call the `update_representation` function with the `mesh_actor` and the `contour_representation` state, and then update the view.

```
@change("contour_representation")
def update_contour_representation(contour_representation, **kwargs):
    update_representation(contour_actor, contour_representation)
    html_view.update()
```

Color By Callbacks

The `color_by_array` function updates the `SelectColorArray` of the `mapper` of an `actor` to the value of `array`. These operations are the same as the ones we used setting up the initial pipeline, so we will not go through the individual commands again here.

```
# Color By Callbacks
def color_by_array(actor, array):
    _min, _max = array.get("range")
    mapper = actor.GetMapper()
    mapper.SelectColorArray(array.get("text"))
    mapper.GetLookupTable().SetRange(_min, _max)
    if array.get("type") == vtkDataObject.FIELD_ASSOCIATION_POINTS:
        mesh_mapper.SetScalarModeToUsePointFieldData()
    else:
        mesh_mapper.SetScalarModeToUseCellFieldData()
    mapper.SetScalarVisibility(True)
    mapper.SetUseLookupTableScalarRange(True)
```

The `update_mesh_color_by_name` function is found by the @change decorator for `mesh_color_array_idx`. We simply call the `color_by_array` function with the `mesh_actor` and the `array`, and then update the view. The `array` is set using the `mesh_color_array_idx` state on the `dataset_arrays` array of dictionaries.

```
@change("mesh_color_array_idx")
def update_mesh_color_by_name(mesh_color_array_idx, **kwargs):
    array = dataset_arrays[mesh_color_array_idx]
    color_by_array(mesh_actor, array)
    html_view.update()
```

Likewise, the `update_contour_color_by_name` function is found by the @change decorator for `contour_color_array_idx`. We simply call the `color_by_array` function with the `contour_actor` and the `array`, and then update the view. The `array` is set using the `contour_color_array_idx` state on the `dataset_arrays` array of dictionaries.

```
@change("contour_color_array_idx")
def update_contour_color_by_name(contour_color_array_idx, **kwargs):
    array = dataset_arrays[contour_color_array_idx]
    color_by_array(contour_actor, array)
    html_view.update()
```

Color Map Callbacks

The `use_preset` function updates the `lut`, lookup table, hue, saturation, and value range for the `preset`. We need the `actor` to get the `lut`, and the `preset` to determine the `hue`, `saturation`, and `value_range`.

```
# Color Map Callbacks
def use_preset(actor, preset):
    lut = actor.GetMapper().GetLookupTable()
    if preset == LookupTable.Rainbow:
        lut.SetHueRange(0.666, 0.0)
        lut.SetSaturationRange(1.0, 1.0)
        lut.SetValueRange(1.0, 1.0)
    elif preset == LookupTable.Inverted_Rainbow:
        lut.SetHueRange(0.0, 0.666)
        lut.SetSaturationRange(1.0, 1.0)
```

```
        lut.SetValueRange(1.0, 1.0)
    elif preset == LookupTable.Greyscale:
        lut.SetHueRange(0.0, 0.0)
        lut.SetSaturationRange(0.0, 0.0)
        lut.SetValueRange(0.0, 1.0)
    elif preset == LookupTable.Inverted_Greyscale:
        lut.SetHueRange(0.0, 0.666)
        lut.SetSaturationRange(0.0, 0.0)
        lut.SetValueRange(1.0, 0.0)
    lut.Build()
```

The `update_mesh_color_preset` function is found by the @change decorator for `mesh_color_preset`. We simply call the `use_preset` function with the `mesh_actor` and the `mesh_color_preset` state, and then update the view.

```
@change("mesh_color_preset")
def update_mesh_color_preset(mesh_color_preset, **kwargs):
    use_preset(mesh_actor, mesh_color_preset)
    html_view.update()
```

The `update_contour_color_preset` function is found by the @change decorator for `contour_color_preset`. We simply call the `use_preset` function with the `contour_actor` and the `contour_color_preset` state, and then update the view.

```
@change("contour_color_preset")
def update_contour_color_preset(contour_color_preset, **kwargs):
    use_preset(contour_actor, contour_color_preset)
    html_view.update()
```

Opacity Callbacks

The `update_mesh_opacity` function is found by the @change decorator for `mesh_opacity`. We simply use a `mesh_actor` property and the `mesh_opacity` state to `SetOpacity`, and then update the view.

```
# Opacity Callbacks
@change("mesh_opacity")
def update_mesh_opacity(mesh_opacity, **kwargs):
    mesh_actor.GetProperty().SetOpacity(mesh_opacity)
    html_view.update()
```

The `update_contour_opacity` function is found by the @change decorator for `contour_opacity`. We simply use a `contour_actor` property and the `contour_opacity` state to `SetOpacity`, and then update the view.

```
@change("contour_opacity")
def update_contour_opacity(contour_opacity, **kwargs):
    contour_actor.GetProperty().SetOpacity(contour_opacity)
    html_view.update()
```

Contour Callbacks

The `update_contour_by` function updates the `SetInputArrayToProcess` of the `contour` filter to the value of `array`. These operations are the same as the ones we used setting up the initial pipeline, so we will not go through the individual commands again here.

```
# Contour Callbacks
@change("contour_by_array_idx")
def update_contour_by(contour_by_array_idx, **kwargs):
    array = dataset_arrays[contour_by_array_idx]
    contour_min, contour_max = array.get("range")
    contour_step = 0.01 * (contour_max - contour_min)
    contour_value = 0.5 * (contour_max + contour_min)
    contour.SetInputArrayToProcess(0, 0, 0, array.get("type"), array.get("text"))
    contour.SetValue(0, contour_value)

    # Update UI
    update_state("contour_min", contour_min)
    update_state("contour_max", contour_max)
    update_state("contour_value", contour_value)
    update_state("contour_step", contour_step)

    # Update View
    html_view.update()
```

the `update_contour_by` function is found by the @change decorator for `contour_by_array_idx`. We simply use the `contour` filter and the `array`, and then update the view. The `array` is set using the `contour_by_array_idx` state on the `dataset_arrays` array of dictionaries.

The `update_contour_value` function is found by the @change decorator for `contour_value`. We simply use a `contour` filter property and the `contour_value` state to `SetValue`, and then update the view.

```
@change("contour_value")
def update_contour_value(contour_value, **kwargs):
    contour.SetValue(0, float(contour_value))
    html_view.update()
```

Start

There is no change to the `start` function.

```
if __name__ == "__main__":
    layout.start()
```

Running the Application

```
python ./04_application/app.py --port 1234
# or
python ./04_application/solution.py --port 1234
```

Your browser should open automatically to `http://localhost:1234/`

ParaView

Download ParaView

ParaView 5.10+ can be downloaded from here (https://www.paraview.org/download).

Virtual Environment

ParaView comes with its own Python, which may be missing some dependencies for the desired usage.
We can add more Python packages into ParaView by create a virtual environment and activate it inside your application by importing our helper module venv.py (https://github.com/Kitware/trame/blob/master/examples/ParaView/venv.py).

First, we need to setup the ParaView add-on python environment, which we will only install ***trame***, but we could add any other Python libraries that are not included in the ParaView bundle.

```
python3.9 -m venv .pvenv
source ./.pvenv/bin/activate
python -m pip install --upgrade pip
pip install "trame"
deactivate
```

Note:

- We can not use our virtual environment with a `vtk` as our `vtk` library will conflict with the one inside Paraview.
- Since ParaView includes `vtk` , any VTK example can be run with ParaView assuming the proper code is used to handle the virtual-env loading to get ***trame*** inside our Python script.

Making *trame* available in ParaView

At the very top of our scripts, we need to import our helper script so the `--venv path/to/venv` can be processed.
The file `[venv.py](https://github.com/Kitware/trame/blob/master/examples/ParaView/venv.py)` needs to be next to your application so it can be found when you import it.

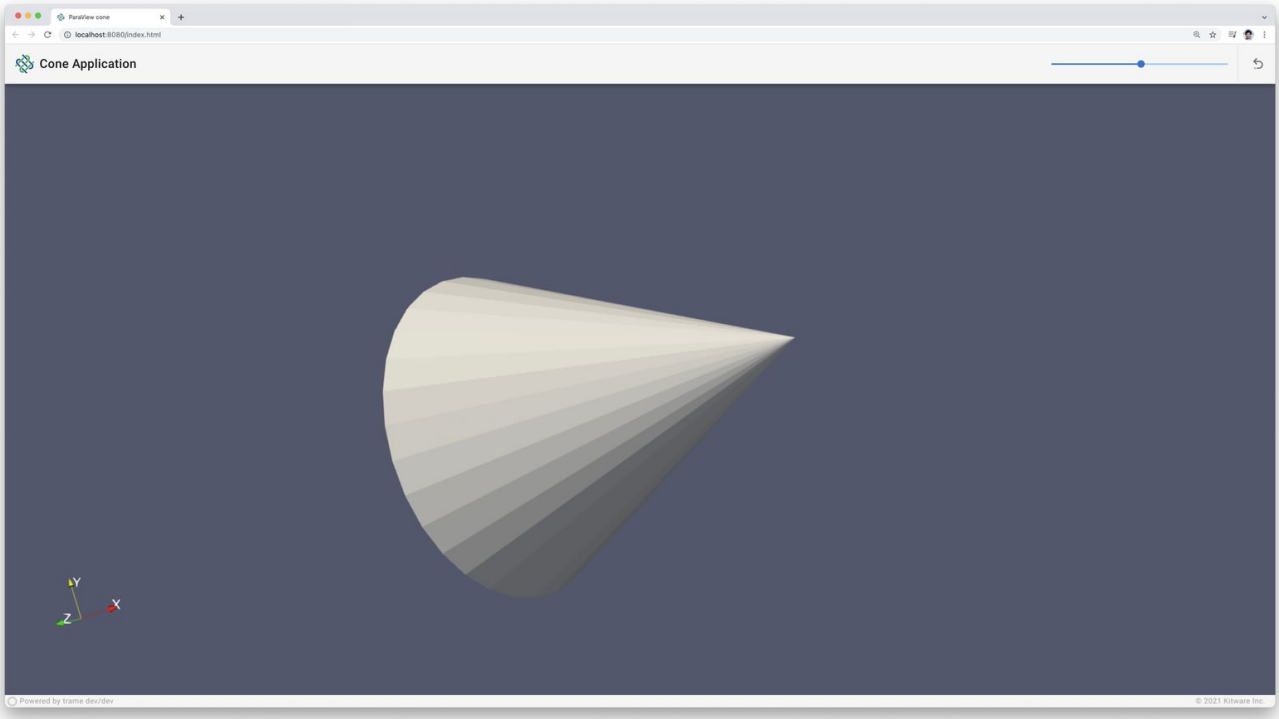
```
import venv
```

After that we can import ***trame*** and start using it (assuming we run our application with the `--venv /path/to/venv/with/trame` argument).

Running an example

The command line below illustrate how a SimpleCone example can be run on a **Mac** computer where ParaView 5.10 has been installed.

```
/Applications/ParaView-5.10.0-RC1.app/Contents/bin/pvpython \
    ./05_paraview/SimpleCone.py  \
    --venv .pvenv
```



Understanding this ParaView example

ParaView use proxies which abstracts the VTK object handling so they can be easily distributed for very large datasets.

For simplified usage, ParaView provides a `simple` package that lets us ***simply*** create and interact with these proxies. The `SimpleCone.py` example provides the core concepts needed to understand how to work with ParaView.

```
from paraview import simple


cone = simple.Cone()                # Create a source (reader, filter...)
representation = simple.Show(cone) # Create a representation in a view (if no view, one is created)
view = simple.Render()              # Ask to compute image of active view and return the corresponding view
```

With these three lines, we create a full pipeline and a view. Now, we can use ***trame*** to show that view in the client.

```
from trame.html import vuetify, paraview
from trame.layouts import SinglePage

html_view = paraview.VtkRemoteView(view)   # For remote rendering
# html_view = paraview.VtkLocalView(view)  # For local rendering

layout = SinglePage("ParaView cone", on_ready=html_view.update)

with layout.content:
    vuetify.VContainer(
        fluid=True,
        classes="pa-0 fill-height",
        children=[html_view],
    )
```

The rest of the code looks very similar to the VTK Hello ***trame*** example, but instead of importing the `vtk` module of ***trame***

```
from trame.html import vuetify, vtk
```

we import the `paraview` module

```
from trame.html import vuetify, paraview
```

GUI

Now we can start adding some UI to control some of the parameters that we want to interact with dynamically.
Let's add a slider to control the resolution of the cone. We need to create a method to react when the `resolution` is changed by the slider. In ParaView proxies, object parameters are simple properties that can be get or set in a transparent manner. At this point, we simply need to update the `cone.Resolution` and update the view to see the change.

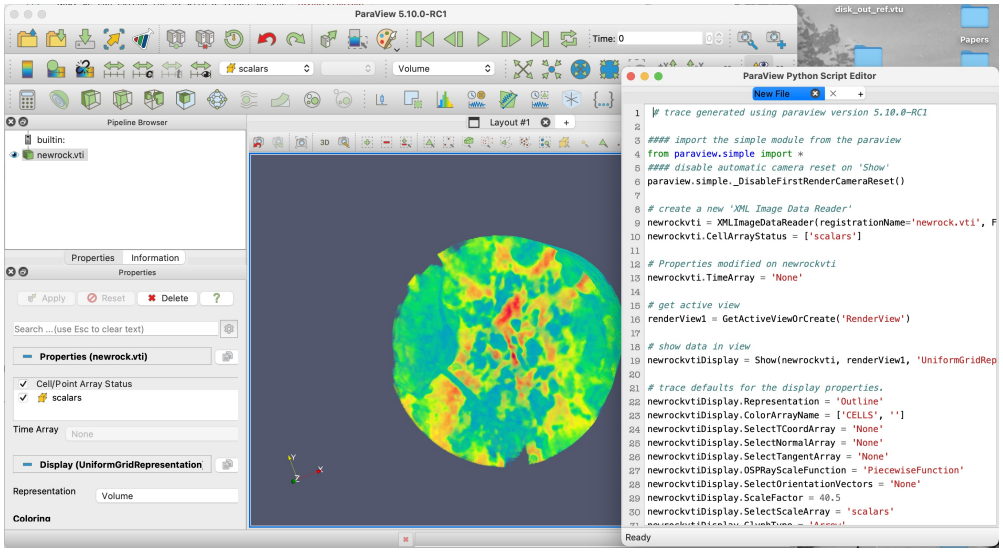
```
@change("resolution")
def update_cone(resolution, **kwargs):
    cone.Resolution = resolution
    html_view.update()
```

Now, we can extend the UI with a slider on the `layout.toolbar`

```
DEFAULT_RESOLUTION = 6

with layout.toolbar:
    vuetify.VSlider(
        v_model=("resolution", DEFAULT_RESOLUTION),
        min=3,
        max=60,
        step=1,
        hide_details=True,
        dense=True,
    )
```

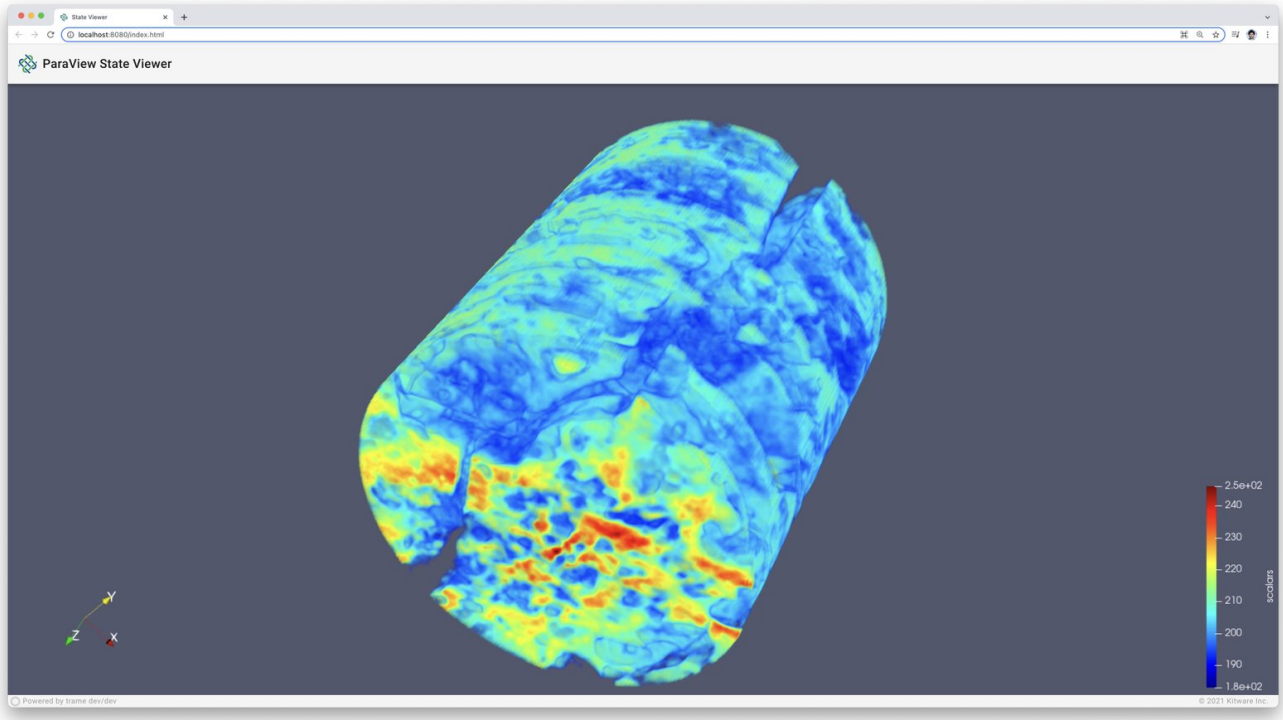
With these few lines, we have created a 3D cone, which we can adjust the resolution all leveraging ParaView.

To learn more about ParaView scripting, you should look into ParaView trace which let you convert your UI interaction into actual Python code that can then be reused in your application.



Advanced example

With the basics in place, we can now dive further in by using some built-in features of ParaView, such as saving and loading a state file. State files are a convinient way of capturing all the settings that were used to generate a visualization with Paraview.



Let's analyse the example in `./05_paraview/StateLoader.py` . The ***trame*** core of the example is as follows

Script Header

```
import venv

import os
import trame
from trame.html import vuetify, paraview
from trame.layouts import SinglePage

from paraview import simple
```

Script Core

The rest of the script we've seen before, but we are missing the details of the `load_data` function.

```
def load_data():
    pass # I'll explain later

layout = SinglePage("State Viewer", on_ready=load_data)
layout.logo.click = "$refs.view.resetCamera()"
layout.title.set_text("ParaView State Viewer")
layout.content.add_child(vuetify.VContainer(fluid=True, classes="pa-0 fill-height"))

if __name__ == "__main__":
    layout.start()
```

`load_data`

The `load_data()` function requires us to code the follow

1. Process a `--data` argument that contains the path to the file to load
2. Load the provided file path as a state file.
3. Create a view element and connect it to the view defined in the state
4. Add that view element into the content of our UI

Process CLI argument `--data`

The (1) is achieved with the following set of lines. More information on CLI are available here (https://kitware.github.io/trame/docs/howdoi-cli.html).

```
parser = trame.get_cli_parser()
parser.add_argument("--data", help="Path to state file", dest="data")
args, _ = parser.parse_known_args()

full_path = os.path.abspath(args.data)
working_directory = os.path.dirname(full_path)
```

Load the state file

To achieve (2) with ParaView the following set of lines are needed. ParaView trace should be able to explain the magic using the UI and looking at the corresponding Python code.

```
simple.LoadState(
    full_path,
    data_directory=working_directory,
    restrict_to_data_directory=True,
)
view = simple.GetActiveView()
view.MakeRenderWindowInteractor(True)
```

Create and Connect a view element

Then (3) is similarly as before for VTK.

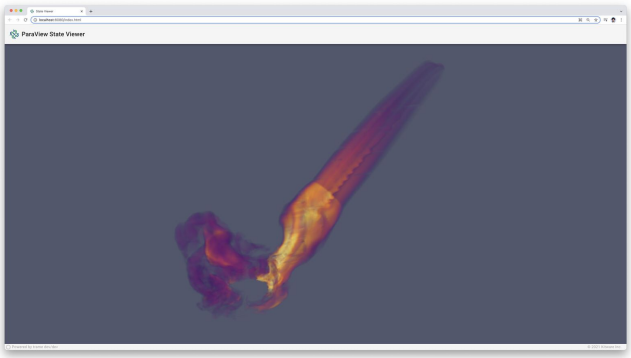
```
html_view = paraview.VtkRemoteView(view)
```

Add view element to UI

Finally (4) is achieved with the following set of lines, the same way it was achieved with VTK in ***trame*** when switching from remote to local rendering.

```
layout.content.children[0].add_child(html_view)
layout.flush_content()
```

That's it. You now have a ParaView `trame` application that let you reproduce complex visualization in a web context.



Running the StateLoader

```
/Applications/ParaView-5.10.0-RC1.app/Contents/bin/pvpython \
    ./05_paraview/StateLoader.py  \
    --venv .pvenv \
    --data ./data/pv-state-diskout.pvsm
# or
/Applications/ParaView-5.10.0-RC1.app/Contents/bin/pvpython \
    ./05_paraview/StateLoader.py  \
    --venv .pvenv \
    --data ./data/pv-state.pvsm
```

Your browser should open automatically to `http://localhost:1234/`