IPython Notebook & VTK

Summary

This post will show the only, currently available, way to integrate those beautiful renderings you might create with VTK into an IPython Notebook by means of off-screen-rendering.

The Visualization Toolkit (VTK)

What & Why

If you haven't heard of the Visualization Toolkit (VTK) before, then chances are that either you're not into image processing and visualization, or that you've been on Mars for the last decade, in a cave, with yours eyes shut (yes, its a Simpsons quote, don't you judge me :)).

For you see, while I'm as strong an advocate of Python as can be (if it was possible to marry a programming language I would have), there are certain tasks that just can't be done effectively or efficiently in Python. Sure, you can do your simpler plots in Matplotlib, Seaborn, or even Bokeh and Plotly (on which I just wrote two posts here and here respectively). Sure, you can try and extract a mesh-surface from a dataset with the marching-cubes implementation in scikit-image or interpolate gargantuan 3D arrays with the routines in SciPy (good luck by the way). At the end of the day, however, the routines in VTK are way faster, more flexible, and have been tested with enormous datasets.

It's not just happenstance that VTK is the core-component in invaluable, albeit as clunky as VTK itself, applications such as ParaView, VolView, Osirix, and 3D Slicer. The thing just works!

Now don't get me wrong, I hate VTK! I find it overly convoluted, extremely poorly documented (talk about the steepest of learning curves), and the simplest of tasks requires 10 times the code and effort you would have in Python (as is the case with anything in C++). However, the functionality it provides can't be found elsewhere and at least the Kitware folk were kind enough to provide a very extensive set of Python bindings.

> If you've followed my advice in the first post about Anaconda, then you should already have VTK in your Python distro or you can simply install it through `conda install vtk`. If not, then at least I hope you're using EPD , Canopy, or some distro where VTK has been pre-compiled. Otherwise, good luck with that :)

Lack of Integration with IPython Notebook

To make things worse, if you're a IPython Notebook fanatic such as myself, you're in for a let-down. VTK doesn't come with an HTML output option that would allow you to directly embed the rendering results into a notebook, or any other web-application for that matter. Obviously, being able to interactively pan/rotate/zoom the 3D scene from within the notebook isn't possible either.

While, some notable effort is being invested in making the above possible, e.g., there's a lot of buzz around ParaViewWeb and the X Toolkit (XTK) but more on this in a later post, I haven't managed to find a simple out-of-the-box solution thus far.

Till something like that becomes available, and I strongly believe it will be sometime soon(ish), forcing an off-screen rendering and pushing an image of that render into the notebook is the only viable pain-free approach I've come across. This is what I'll be showing here.

Embedding VTK Renders in IPython Notebook

Disclaimer: The code I'll be giving in this section is **not mine**. I've come across this solution in the form of a 'gist' here , while a snippet of this code can also be found here. However, as these might be pulled down one day, I wanted to outline that code, explain how it works, and be able to refer to it in subsequent posts that will be dealing with VTK.

The function we'll be using is the following:

```
import vtk
from IPython.display import Image
def vtk_show(renderer, width=400, height=300):
    """
    Takes vtkRenderer instance and returns an IPython Image with the rendering.
    """
    renderWindow = vtk.vtkRenderWindow()
    renderWindow.SetOffScreenRendering(1)
    renderWindow.AddRenderer(renderer)
    renderWindow.SetSize(width, height)
    renderWindow.Render()

    windowToImageFilter = vtk.vtkWindowToImageFilter()
    windowToImageFilter.SetInput(renderWindow)
    windowToImageFilter.Update()

    writer = vtk.vtkPNGWriter()
    writer.SetWriteToMemory(1)
    writer.SetInputConnection(windowToImageFilter.GetOutputPort())
    writer.Write()
    data = str(buffer(writer.GetResult()))

    return Image(data)
```

As you can see the `vtk_show` function takes three parameters: a `vtkRenderer` instance which contains what you want to render, and the `width` and `height` of the resulting rendering.

Now, lets examine the different parts of the above code:

```
    renderWindow = vtk.vtkRenderWindow()
    renderWindow.SetOffScreenRendering(1)
    renderWindow.AddRenderer(renderer)
    renderWindow.SetSize(width, height)
    renderWindow.Render()
```

What this part does is create a new `vtkRenderWindow` , enable off-screen rendering, and adds the

`vtkRenderer` instance to it before setting the dimensions, and performing the rendering operation.

> Note that the `Render` method above will actually create a window separate to the browser where your IPython Notebook currently is. Do **not** mess with it or close it cause you will most likely get a crash.

```
    windowToImageFilter = vtk.vtkWindowToImageFilter()
    windowToImageFilter.SetInput(renderWindow)
    windowToImageFilter.Update()
```

The above part creates a new 'vtkWindowToImageFilter' object which allows us to read the data in a `vtkWindow` and use it as input to the imaging pipeline. It then adds `renderWindow` to it and updates it.

```
    writer = vtk.vtkPNGWriter()
    writer.SetWriteToMemory(1)
    writer.SetInputConnection(windowToImageFilter.GetOutputPort())
    writer.Write()
    data = str(buffer(writer.GetResult()))

    return Image(data)
```

Lastly, a new `vtkPNGWriter` is created allowing us to create a PNG image out of the `vtkWindow` rendering. Note that we enable `SetWriteToMemory` since we don't just want a PNG file to be created bur rather to embed that image-data into the IPython Notebook. Subsequently, we establish a 'connection' between the output of the 'vtkWindowToImageFilter' and the 'vtkPNGWriter' by means of the `SetInputConnection` and `GetOutputPort` methods, a very typical mechanism in VTK. After using `Write` to create the image, we take that data, push in a Python `buffer`, and convert it to a `string`. Lastly, we use the `Image` class from the `IPython.display` module, imported outside the function, to convert that data to an actual image which can be directly embedded into the IPython Notebook.

Example: The Red Ball

Now lets give a quick example, to showcase the function we just presented. In the interest of consistency, I stole and modified this example from the original author of this function :). You can see more examples in that gist here.

The following code creates a red sphere and uses the `vtk_show` function shown in the Embedding VTK Renders in IPython Notebook section to display the output within the IPython Notebook:

```
VtkSourceSphere = vtk.vtkSphereSource()
VtkSourceSphere.SetCenter(0.0, 0.0, 0.0)
VtkSourceSphere.SetRadius(10.0)
VtkSourceSphere.SetPhiResolution(360)
VtkSourceSphere.SetThetaResolution(360)

VtkMapperSphere = vtk.vtkPolyDataMapper()
VtkMapperSphere.SetInputConnection(VtkSourceSphere.GetOutputPort())

VtkActorSphere = vtk.vtkActor()
VtkActorSphere.SetMapper(VtkMapperSphere)
VtkActorSphere.GetProperty().SetColor(1.0, 0.0, 0.0)
```

```
VtkRenderer = vtk.vtkRenderer()
VtkRenderer.SetBackground(1.0, 1.0, 1.0)
VtkRenderer.AddActor(VtkActorSphere)

vtk_show(VtkRenderer)
```

Now let us examine the code piece-by-piece:

```
VtkSourceSphere = vtk.vtkSphereSource()
VtkSourceSphere.SetCenter(0.0, 0.0, 0.0)
VtkSourceSphere.SetRadius(10.0)
VtkSourceSphere.SetPhiResolution(360)
VtkSourceSphere.SetThetaResolution(360)
```

The above code-block creates an `vtkSphereSource` object which creates a sphere represented by polygons. We use the `SetCenter` and `SetRadius` to set the coordinates of the center to the `(0, 0, 0)` origin of the cartesian system and set the radius of the sphere to `10.0`. The `SetThetaResolution` and `SetPhiResolution` methods set the number of points in the longitude and latitude directions respectively. A low 'resolution' results in the sphere looking polygonal. You can find the docs on this class here but be warned: the VTK docs are no more than an API reference.

```
VtkMapperSphere = vtk.vtkPolyDataMapper()
VtkMapperSphere.SetInputConnection(VtkSourceSphere.GetOutputPort())
```

Consequently, we create a new `vtkPolyDataMapper` (docs here) which maps polygonal data, such as the data representing the sphere, to graphics primitives. We then use the same `SetInputConnection` and `GetOutputPort` pipeline mechanism we saw in the previous section.

```
VtkActorSphere = vtk.vtkActor()
VtkActorSphere.SetMapper(VtkMapperSphere)
VtkActorSphere.GetProperty().SetColor(1.0, 0.0, 0.0)
```

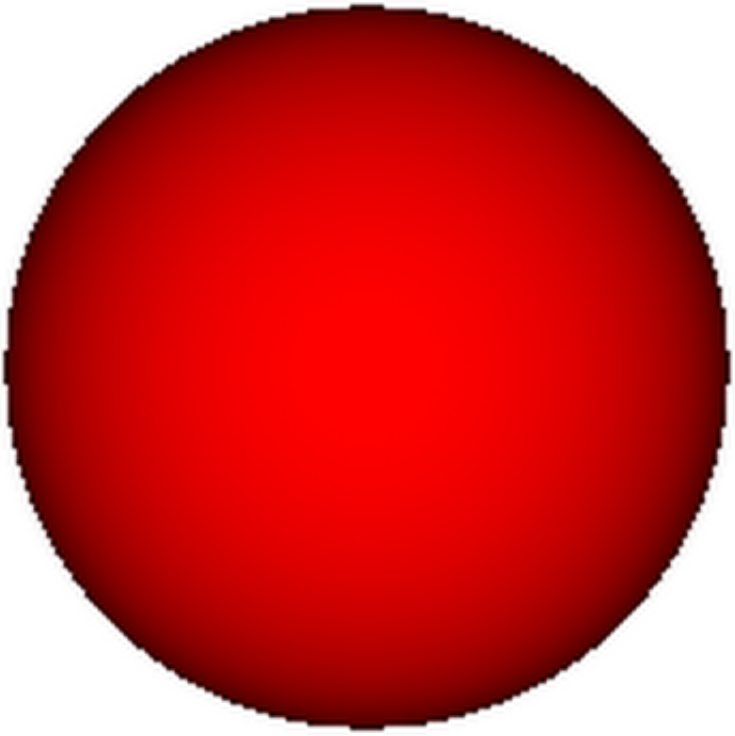
Next, we create a new `vtkActor` (docs here) which is used to represent an entity in a rendering scene. We assign the newly created `vtkMapper` using the `SetMapper` method in order to link the sphere with this new actor. Remember how I said that VTK is overly convoluted? Well while understanding and utilizing the pipeline is the bane of my VTK dealings, once you start getting into it you'll realize its well-justified. Lastly, we grab an instance of the actor's properties through `GetProperty` and set the RGB color to red (utilizing a `0.0` to `1.0` range).

```
VtkRenderer = vtk.vtkRenderer()
VtkRenderer.SetBackground(1.0, 1.0, 1.0)
VtkRenderer.AddActor(VtkActorSphere)
```

We're almost there, we merely create a new `vtkRenderer` object (docs here), set the rendering background color to white using the `SetBackground` method, and add the sphere actor we just created.

Lastly, remember how our `vtk_show` function needs a `vtkRenderer` instance to function? Well now we have

it so all we need to do is call that function as such `vtk_show(VtkRenderer)`. We could define the width and height of the rendering through a call like `vtk_show(VtkRenderer, width=100, height=200)` but we won't bother. Upon execution you should get an image akin to the one in the figure below.



Rendering of a red sphere through VTK shown directly in IPython Notebook

You can find an IPython Notebook with the code shown here under this link. Note that since we're embedding the resulting render as an `IPython.display.Image` object, the image should be viewable on the nbviewer which makes it great for portability.

Resources

If you want to learn more about VTK then you could try and procure the book or take a look at small number of external resources. Special mention should go to this free book titled 'Introduction to Programming for Image Analysis with VTK'. Keep in mind though that the majority of books and examples available out there provide code in Tcl script (who still uses that thing?) and C++ so you're in for some translating if you're using Python.

Otherwise, feel free to check back here cause I will be be providing more material in VTK which you might find useful :).

That's about it folks, thank you for reading, and I hope you gained something from this 2000-word rant.