REPORT ON OpenDreamKit DELIVERABLE D4.12

JUPYTER extension for 3D visualisation, demonstrated with computational fluid dynamics

BENJAMIN RAGAN-KELLEY, VIDAR TONAAS FAUSKE, MARCIN KOSTUR

Due on	31/08/2018 (M36)
Delivered on	31/08/2018
Lead	Simula Research Laboratory (Simula)
Progress on and finalization of this deliverable has been tracked publicly at: `https://github.com/OpenDreamKit/OpenDreamKit/issues/86`	

CONTENTS

1. INTRODUCTION

The Jupyter Notebook is a web application that enables the creation and sharing of executable documents containing live code, equations, visualisations and explanatory text. In particular, Jupyter is used actively for interactive and exploratory computations, often involving visualisation of data.

Two-dimensional visualisation in Jupyter is an area with popular and well-established tools such as matplotlib, bokeh, and altair, but there have been fewer mature solutions for interactive three-dimensional visualisation. It is an area of active exploration, so we first set out to identify the current state of the art, and determine where our efforts could best be placed. We did this in the form of a report on the landscape of 3D visualisation in Jupyter in 2017, see Appendix B.

Because Jupyter is a web-based application, it can benefit from the very active development, in the wider community, of browser-based interactive visualisation tools. Therefore, the primary task when developing visualisation tools for Jupyter is to connect 'kernel' code, where the user's code runs, to visualisation code running in the browser. This is a standard need and the Jupyter ecosystem provides a powerful and extensible system for bidirectional communication between

the kernel and the browser, which serves as foundation for a wide collection of interactive tools and projects for Jupyter called 'Jupyter Widgets'.

We therefore naturally chose this foundation and focused our efforts on improving compatibility between existing browser-based tools and the Jupyter Widgets system.

This deliverable had three objectives: to better understand the existing landscape of community efforts for three-dimensional visualisation in notebooks; to contribute where we may have the best impact, whether it be in contributing to existing projects or creating new tools; and finally to demonstrate the technology and tools in the demanding and high impact area of Computational Fluid Dynamics visualisation (tasks **T4.8**: "Visualisation system for 3D data in web-notebook" and **T4.9**: "Visualisation of 3D fluid dynamics data in web-notebook").

We have accomplished all three objectives. In particular, we have helped improve the core Jupyter Widgets framework and the JupyterLab application, and we have developed new software in K3D-jupyter, SciviJS, ipyscales, ipydatawidgets, unray, and others projects. Overall, OpenDreamKit contributed approximately 18 person-months.

2. Outcomes

We have developed several new software packages and contributed to the core Jupyter widget frameworks used by hundreds of thousands of people.

2.1. **Report on the landscape of 3D visualisation**

To identify where to most effectively put our efforts, we examined the landscape of existing 3D visualisation projects, included in Appendix B. We observed several existing projects with various levels of maturity and activity, and different strengths and weaknesses. The primary conclusions of the report were that 1. three.js is a good candidate as a common ground on which 3D visualisation tools could be built because it is an active project and already in use by several tools, and 2. the Jupyter Widgets provide a good system for implementing communication between the Jupyter kernel and the browser. This report helped us spend our efforts most effectively, but is a useful guide in its own right for observing the state of 3D visualisation in Jupyter.

2.2. **Improving core Jupyter tools and infrastructure for visualisation**

There are a number of common aspects of 3D visualisation, which can be shared across 3D visualisation and other tools in the notebook. In order to maximize the impact of our work, we aimed to improve this infrastructure, both by contributing to existing packages and creating some new ones that should be broadly useful.

Given the identification of three.js as a common denominator among visualisation tools we decided to pick up and finalize an existing effort to rewrite the package *pythreejs*. The package allows for full 3D scene creation and management from the kernel. By rewriting the package to use auto-generation of code, a much larger part of the three.js API could be exposed to the kernel side, greatly extending its capabilities, and reducing the future maintenance burden. We also incorporated functionality into pythreejs to allow it serve as an extension point for other extensions, thereby being capable of serving as an interoperability platform between different libraries as well. OpenDreamKit members completed this effort and led the process of making a 1.0 release of the pythreejs package, and are now serving as a maintainer of the pythreejs package.

One of the challenges with 3D visualisation in the browser of data originating from a possibly remote kernel is the efficient transfer of data between the two. To help solve this problem, the library *ipydatawidgets* was created by OpenDreamKit. It incorporates best practices for the binary transfer of array data between the kernel and the browser, and for avoiding unnecessary retransmission of data. It also includes features to facilitate extensibility and interoperability

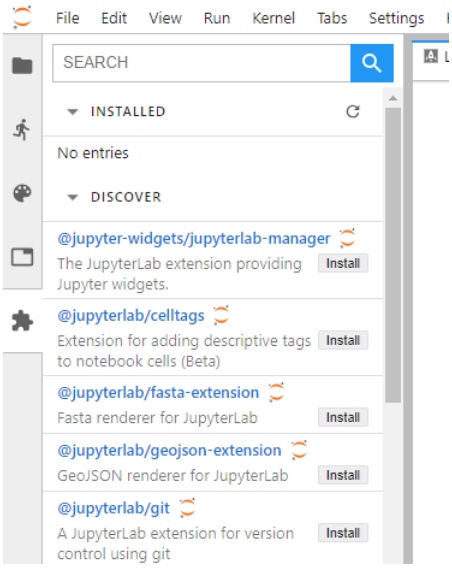

FIGURE 1. Extension manager for JupyterLab. Contributed by OpenDreamKit to ease distribution of widget libraries.

between different consumers. In this manner, a dataset created or used by one package can easily be reused by another package, without extra resource usage.

In order to reduce double-work across the fragmented landscape of visualisation libraries, an effort has also been made into creating high-quality, reusable components. This allows visualisation library authors to focus more on their unique features, while sharing the load of common tasks across multiple projects. This includes work on:

- a set of common 3D plotting components, e.g. a set of 3D grids/axes.
- a component for rending a color bar.
- utilities for creating and using colormaps.
- other scales for transforming data from one domain to another.

We contributed to the existing jupyter-widgets (ipywidgets) and the widget cookie cutter project to make it easier to embed and share widgets with others. OpenDreamKit's contributions are namely:

- sharing an HTML conversion of the notebook including widgets.
- exporting certain widgets from a notebook to standalone HTML pages.
- including widgets in package/library documentation.

An effort has also been made by OpenDreamKit to help ease the creation and distribution of Jupyter widget libraries. This is critical in order to ease adaption of the visualisation libraries based on Jupyter widgets. Notable contributions include:

- work done on the widget cookie cutter projects, in order to simplify and standardize best practices for widget packaging. OpenDreamKit contributed to improving this existing project.
- the creation of an extension manager for JupyterLab (Figure 1), that also knows how to handle companion packages for kernels. OpenDreamKit contributed this functionality to the existing JupyterLab project.

As a result, all Jupyter widget authors can benefit from the work of OpenDreamKit, making it more efficient and accessible to create, distribute, and install custom Jupyter widget-based projects.

2.3. **Working with the community**

During the extent of the work task, we organized two workshops focused on 3D visualisation. One was at XFEL (22 June 2018) in relation to the project meeting and was meant to bring together the relevant stakeholders (developers and consumers) of the work. The other was held at the University of Silesia (16-19 July 2018), and focused on integration and interoperability between K3D-jupyter and other visualisation libraries.

Additionally, we participated in a community organized workshop on Jupyter widgets (23-26 January 2018, Ecole Polytechnique, Paris, France). The stated goal of this workshop was to "[g]et Jupyter Widgets developers to meet for a week of coding sessions with a few presentations" and to "[...] foster synergy, get everybody to meet, and resolve common issues".

2.4. **New 3D visualisation tools**

2.4.1. *K3D-jupyter.* K3D-jupyter is a package created by OpenDreamKit which provides a fast and simple 3d plotting tool in the Jupyter notebook (Figures 4 and **??**). The primary aim of K3D is to be easy for use as stand alone package like matplotlib, but also to allow interoperation with existing libraries as VTK. The power of ipywidgets makes it also a fast and performant visualisation tool for HPC computing e.g. fluid dynamics.

2.4.2. *unray.* unray is a scientific visualisation package created by OpenDreamKit for Jupyter notebooks for displaying scalar data on unstructured tetrahedral meshes (Figure 2). It allows for various volumetric rendering modes, isosurface rendering, and segment rendering. It relies on pythreejs, and the reusable components outline above to handle user control and interaction beyond the specific needs of the volumetric rendering.

2.4.3. *SciviJS.* SciviJS is a new general javascript library developed by OpenDreamKit for visualising volumetric data (Figure 3). It is built on threejs, and enables GUI-based interactive exploration of mesh data on a regular webpage, without any special software. Additionally, a *jupyter-scivijs* package enables using SciviJS from within Jupyter notebooks.

2.5. **Computational Fluid Dynamics visualisation**

Computational Fluid Dynamics (CFD) was one of OpenDreamKit's major use cases to steer, evaluate, and demonstrate the work on 3D visualisation (**T4.9**: "Visualisation of 3D fluid dynamics data in web-notebook"). In this area, the Lattice Boltzmann Method (LBM) has recently became a very popular method. The most popular variant is based on regular grids which makes it very efficient on GPU architectures. This makes it in turn very important to have web based visualisation tool, as it is very common to run the simulation on a remote GPU server through a Jupyter notebook interface.

We have equipped the K3D-jupyter package with a few features which make the visualisation of boundary conditions and fields in computational fluid dynamics especially easy:

- a native method for displaying voxel geometry, which performance and quality is appropriate for typical geometries in LBM.
- standard functions for plotting vectors and lines;
- specialized method for displaying cross section of a computational domain, optimized for real time update during the simulation (K3D.texture);
- volume rendering, a very powerful tool in visualisation of scalar fields;
- dynamic update of K3D objects during the simulation.

Additionally K3D-jupyter can be easily used in a VTK pipeline and therefore can display various sophisticated visualisation methods. It also extends most of its CFD capabilities beyond regular grid based LBM simulations, which makes it viable tool for any CFD package.

An important aspect of K3D design was to provide a tool for displaying live LBM simulation without any performance penalty. The classical approach was usually to store all fields to files

for postprocessing done usually in software like paraview. Live inspection of data during a simulation was very limited. If one takes a typical simulation taking place on GPU, then it is not uncommon to have it running with 1 GLUPS (giga lattice updates per second) on modern hardware. It means that the full velocity field in each time iteration would need bandwidth c.a. 12 GB/s. Those numbers are many orders of magnitude higher than capabilities of web connections. The reasonable solution is to perform some of the visualisation work on GPU and send a reduced dataset to the web-browser. We have experimented with two such methods: a slice of the data and tracer particles. In sailfish-cfd we have implemented a method which can on request do a 2d slice through the computational domain. The slice is done in GPU memory and at no point the whole dataset is transferred to the host. Then we fetch the slice and display the data using k3d.texture function. In this way it is possible to monitor a running simulation with very high refresh rate inside the Jupyter notebook.

APPENDIX A. SCREENSHOTS

In [10]:
```
# Specify color lookup table as an array of rgb triplets
lut = ur.ArrayColorMap(values=[
    [224/255, 243/255, 219/255],
    [67/255, 162/255, 202/255],
])
color = ur.ColorField(field=field, lut=lut)
wp = ur.WireframeParams(enable=True, color="#00aaaa", opacity=0.25)
plot = ur.SurfacePlot(mesh=mesh, color=color, wireframe=wp)
display_plots(plot, scale=scale, background="white")
```

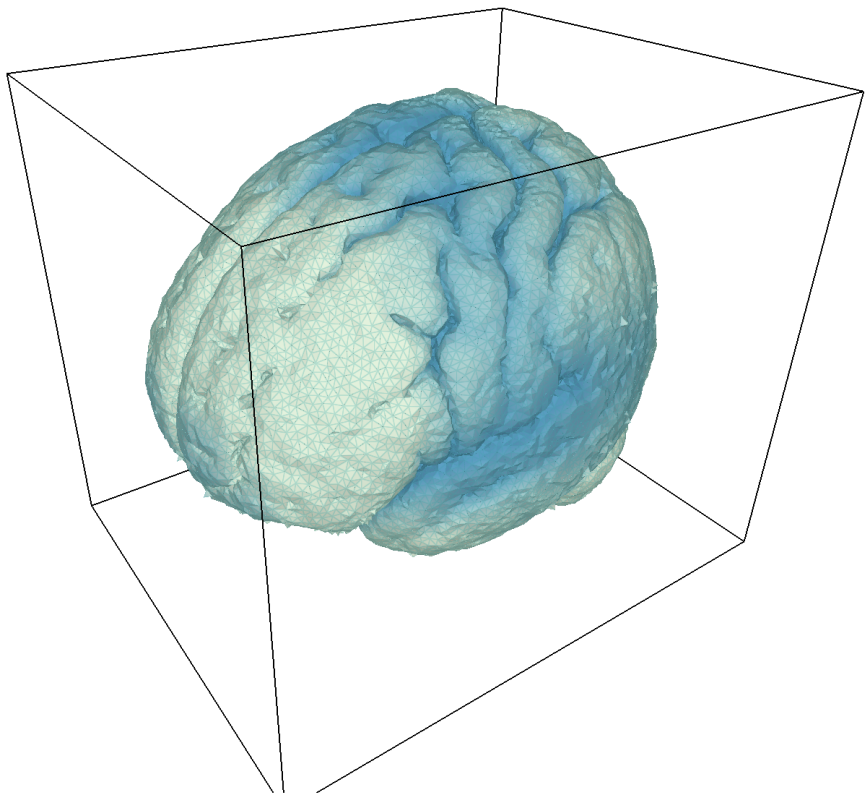

FIGURE 2. Unray visualisation of data on a human brain mesh.

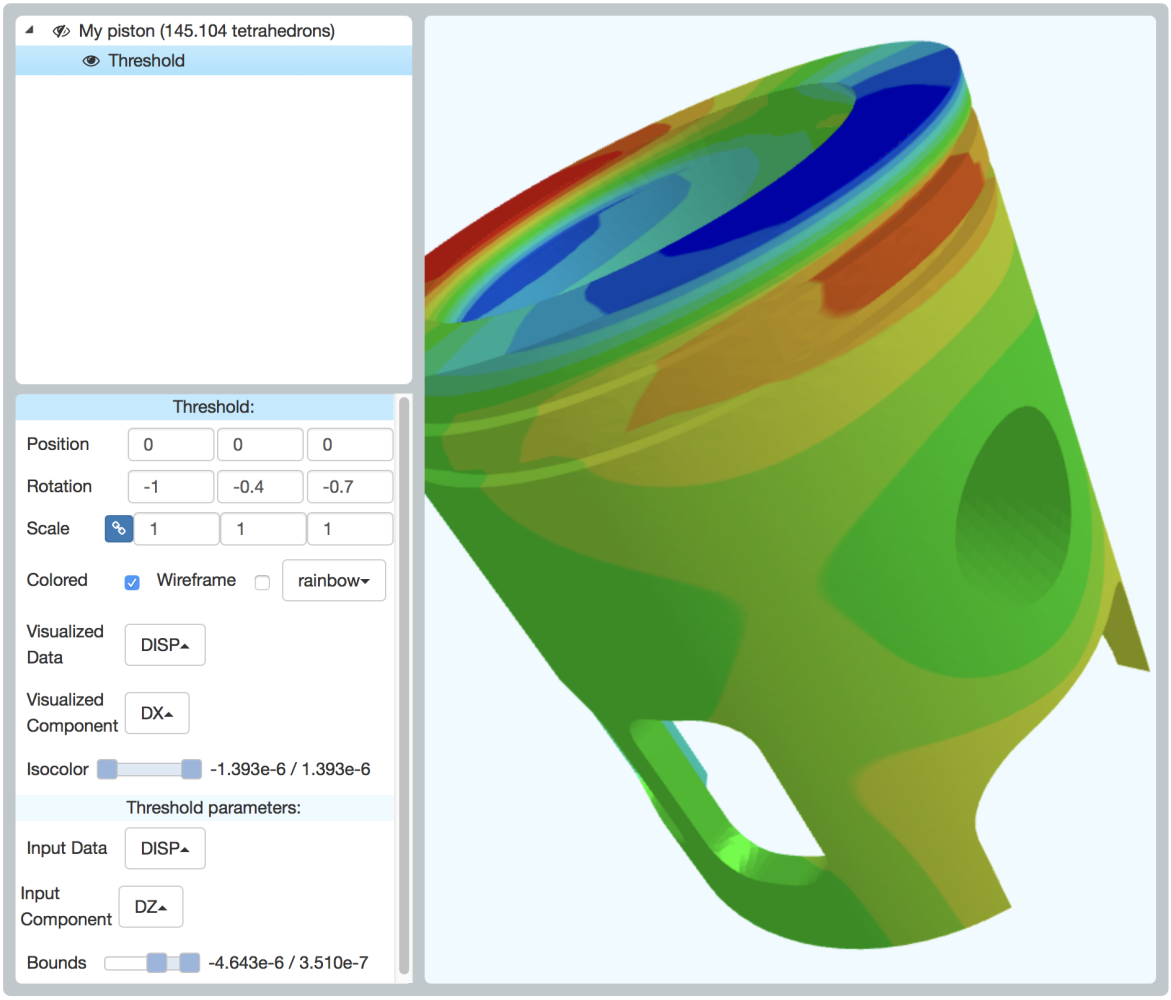

FIGURE 3. SciviJS visualisation of a piston.

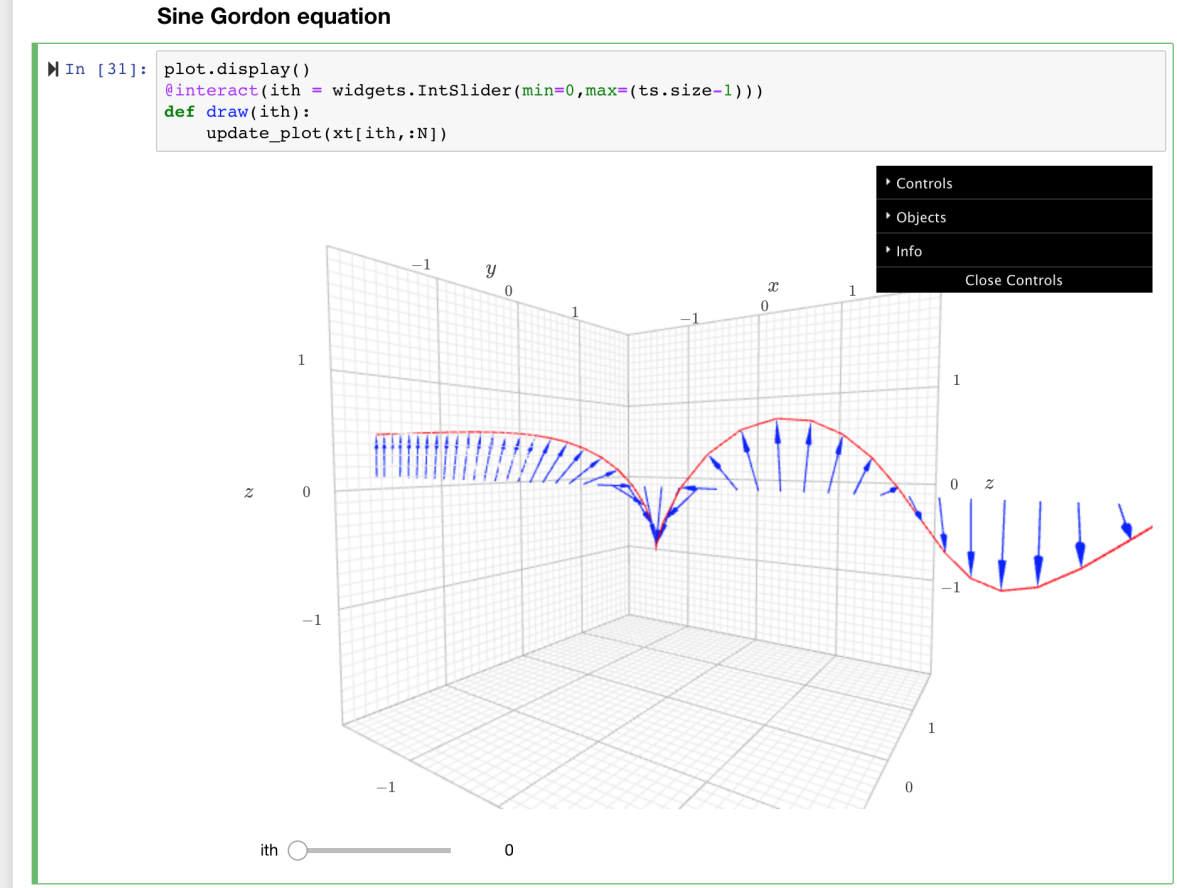

FIGURE 4. K3D visualisation in Jupyter. The K3D widget is mixed with a slider to make interactive animation of Sine-Gordon equation in 3d space.

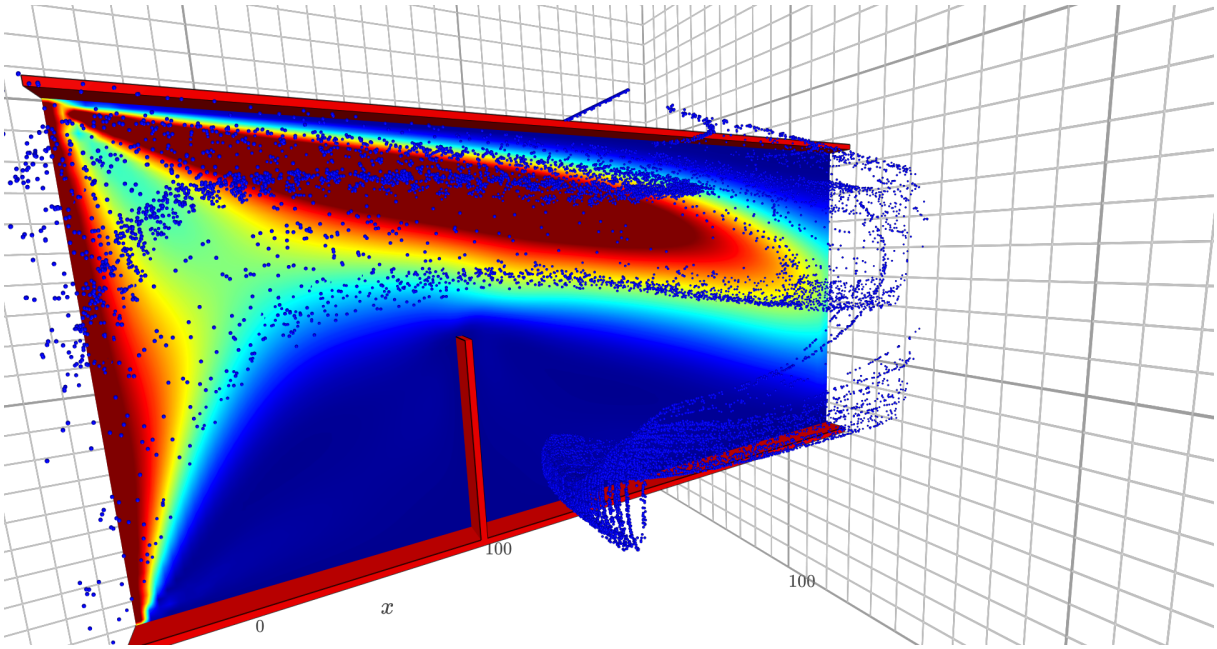

FIGURE 5. K3D visualisation of fluid dynamics. Boundaries are shown using a `k3dvoxels` object and tracers are dynamically updated during simulation the notebook (`k3dpoints`). The slice through the velocity field is shown using `k3dtexture` object. The update makes real time observation of even large lattice Bolzmann dynamics simulation possible. The refresh rate of 10-20fps does not impact the performance of the simualtion which is run on GPU.

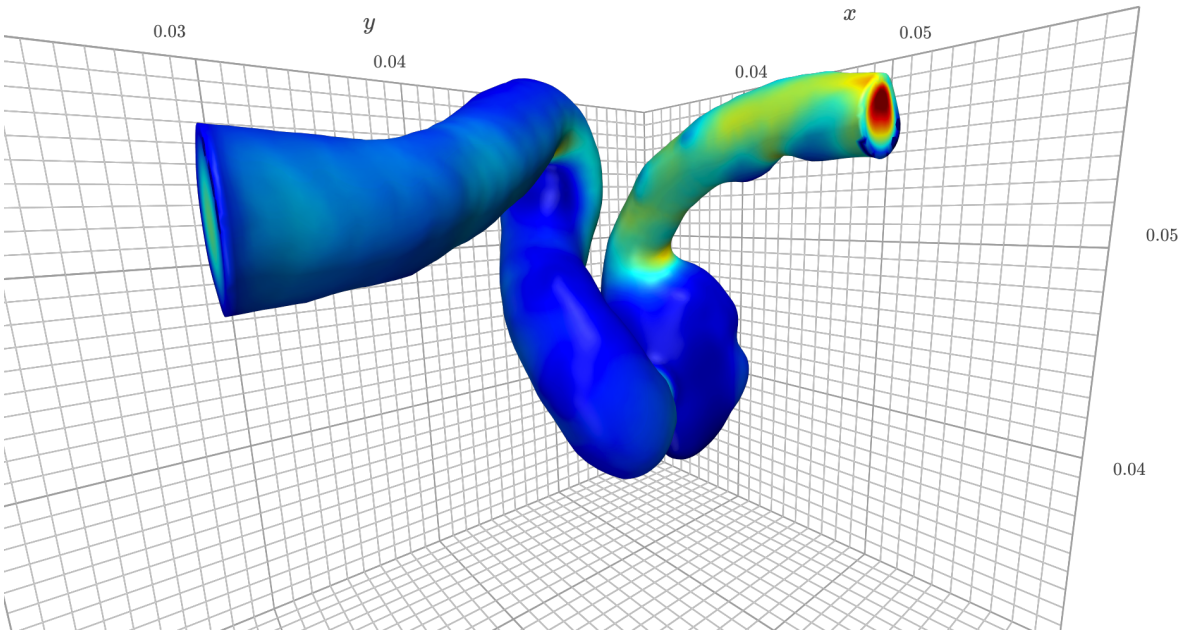

FIGURE 6. K3D visualisation of wall shear stress on the boundaries in the fluid dynamics simulation. This picture uses VTK on the server-side which extracts surface data as VTK-PolyData which are shown using `k3d.vtk_poly_data` function.

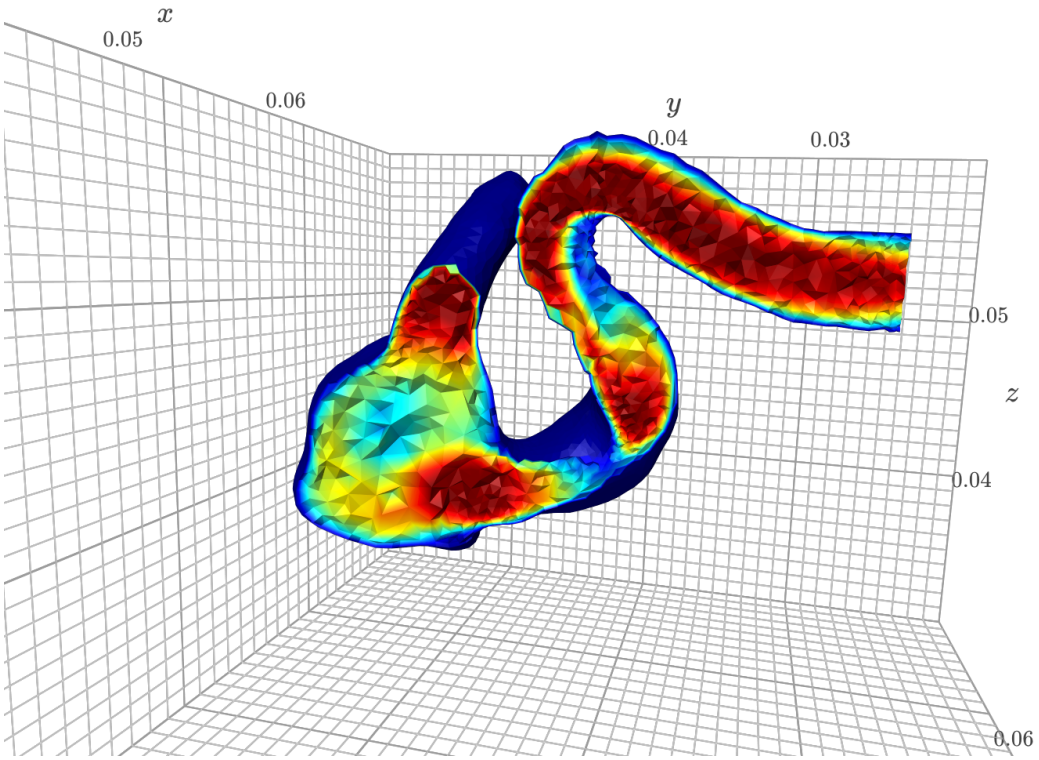

FIGURE 7. K3D visualisation of the velocity in the fluid domain using server-side VTK cutting. The cut-plane can be choosen using client side tool in the widget, which generates callback to the vtk pipeline.

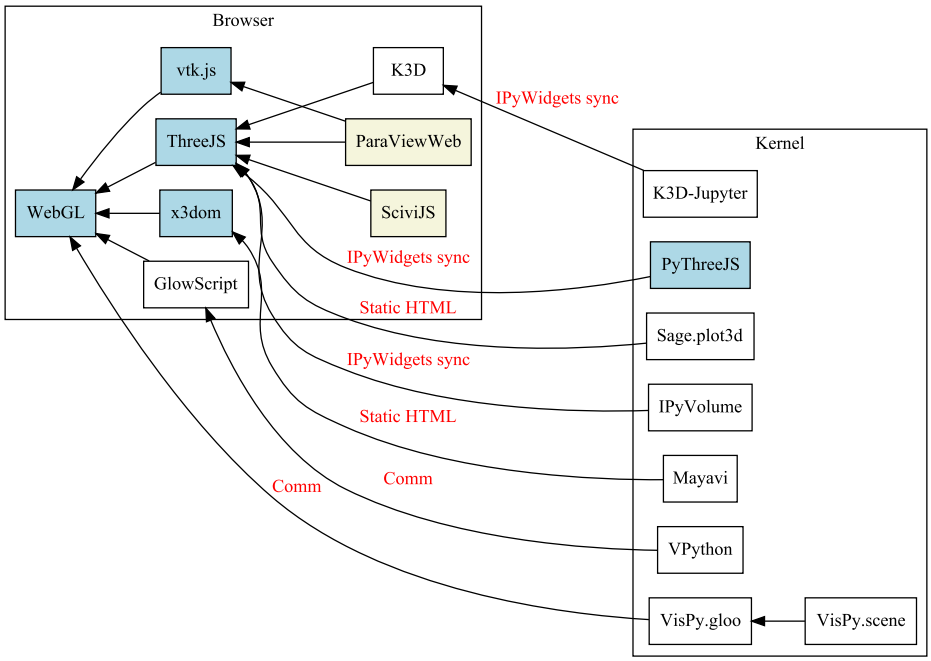

FIGURE 8. Package dependency graph

APPENDIX B. LANDSCAPE REPORT

This is a survey of the landscape of 3D visualization in Jupyter Notebooks performed in March 2017 by Vidar Tonaas Fauske, to guide OpenDreamKit contributions to 3D visualization tools.

B.1. **Summary**

Several packages currently exist that can generate 3D visualizations in Jupyter Notebooks. This review is intended to give a quick overview of the current landscape (March 2017), and consider where the contributions of OpenDreamKit (ODK) can best be directed in order to help improve the user experience in terms of both features and ease-of-use. The packages and technologies that will be considered are:

- mayavi
- ipyvolume
- vispy
- scivijs
- K3D-jupyter
- vpython
- Sage plot3d

Below you can see a (limited) dependency graph between the different project and technologies. The red text signifies how a kernel package sends/communicates 3D data to the browser side.

As can be seen, there is a variation in how the different projects couples to WebGL.

B.1.1. *Observations.* Among the packages certain patterns are repeating: - Mayavi and Sage's plot3d both generate a blob of HTML with x3dom/ThreeJS definitions of the model + interaction code. This is added as an output of the cell, and it makes the visualization persistable (i.e. the view will work the same with or without a kernel), but it also limits interactivity. - VPython and VisPy both use temporary displays that are controlled over Comm. They currently require a roundtrip to the kernel for every frame draw and/or user interaction. IPyVolume and K3D also uses a temporary display, but avoids kernel roundtrips by using pre-defined ThreeJS view controllers. All four packages should be able to persist the views, e.g. by recording the executed

DOM + Javascript to the output. - Many of the packages only supply basic inspection tools (a simple camera to rotate/move). More advanced interactive inspection tools (varying opacity of different parts, adding and adjusting clipping planes, tresholding, etc.) are more rare, but could add many benefits to the user. A possibility here would for ODK to help produce reusable inspection tools for other visualization tools to use.

In terms of persistence, there are a few things to take into account: - If the notebook is not trusted, JavaScript is not available, so a static image should preferrably be saved with the notebook. - Persisting as HTML with JavaScript is maybe the easiest as it will work out of the box without any extensions. - Persisting e.g. x3dom as a separate MIME-type could enable 3D to work while untrusted (any avenues for code injection would of course need sanitizing). Would require adding a rendrer through an nbextension or on the Jupyter level. Adding both this MIME type and the HTML one on a single output would be nice, but it would nearly double the output size, which can be significant for larger geometries/textures.

B.2. **Enabling technology**

B.2.1. *Browser side.*

WebGL. The base layer in terms of enabling technology. All other packages discussed in this report rely on this.

ThreeJS. A Javascript framework built on top of WebGL. ThreeJS is the most prominent WebGL framework today.

Combining custom WebGL code with ThreeJS is not straight forward, but adding custom shaders is reasonably easy, and ThreeJS should be a sufficient base for most scientific visualization needs.

x3dom. A Javascript/XML framework built on top of WebGL. It takes XML as source, and translates this into a 3D scene. While it sports a reasonably full feature set, it has not received the same amount of developer hours as ThreeJS, which can sometimes show. An example of a simple feature that is missing is an orbit view controller (a "camera" orbiting a point, where the up axis is fixed).

vtk.js. Aims to be (a subset of) VTK built on WebGL. Official project by Kitware, which plan to transition ParaViewWeb from ThreeJS to it over time.

B.2.2. *Kernel side (Python).*

PyThreeJS. A Python bridge to ThreeJS by syncing IPyWidgets.

> *"This is meant to be a low-level wrapper around three.js. We hope that others will use this foundation to build higher-level interfaces to build 3d plots."*

B.3. **Higher level interfaces**

B.3.1. *Mayavi.* Mayavi is a high-level interface on top of VTK. It uses x3dom to handle 3D in browser, but also supports a `png` backend.

Mayavi produces a static HTML output with an x3dom scene generated by VTK's X3DExporter. VTK also has a WebGLExporter, which is currently not used by Mayavi.

Installation. Depends on VTK (>= 5.0) with Python wrappers installed. Not straight forward to get on all Windows distrubutions, but works nicely with conda.

The docs says to enable x3dom by installing Mayavi as a notebook extension, but this has errors. It also bundles an outdated version of x3dom (bugs out on high-DPI monitor). Simply not installing the Mayavi nbextension will load the x3dom javascript dynamically from x3dom.org, which works fine.

Strengths and weaknesses. Strengths: - Mature package for generating 3D visualization. - Relies on exporting capabilities of underlying VTK, meaning it should be reasonably robust. - Output persisted naturally.

Weaknesses: - Requires full re-plot for any changes to the VTK scene; limited chances for interactivity via e.g. IPyWidgets. - Requires VTK installation.

License. BSD 3-clause

Possible contributions.

- Make a separate notebook/jupyterlab renderer for x3dom MIME type?
- Also add a static PNG snapshot to output MIME bundle.

B.3.2. *IPyVolume.* Higher-level package for quickly creating interactive 3D plots of volume and scatter data. Started December 2016, so still very new.

Installation. Simple pip install. Package is pure Python + npm package for notebook extension. Dependencies like Pillow are not pure Python.

Also available on conda.

Strengths and weaknesses. Strengths: - Integrates well into existing Jupyter environment by building on top of IPyWidgets. - Good interactivity. - Support for animations (although limited) with interpolation between frames.

Weaknesses: - Limited capabilities (no lines / surface plotting, no exploration with e.g. clip planes, etc.). - Features still need ironing out / polishing to mature.

License. MIT

Possible contributions.

- Help mature existing plotting tools.
- Add other high-level functionality like 3D lines / surfaces? That might be outside the scope of the package, but a package that does something like this might benefit from the communication framework.
- Add interactive exploration controls (clip planes etc.) based on IPyWidgets.

B.3.3. *K3D.* Described as a *"3D visualization library"*. The specifics of K3D was not readily apparent, as its documentation is rather thin. However, the following can be gleamed from the code/examples:

- K3D is intended as a Javascript library for scientific plotting (high-level API).
- It uses ThreeJS as a backend (currently the only backend).
- K3D-Jupyter is meant to shim the K3D interface to Jupyter using an nbextension (and implements an IPython client using IPyWidgets).

K3D might be overlapping in purpose with vtk.js.

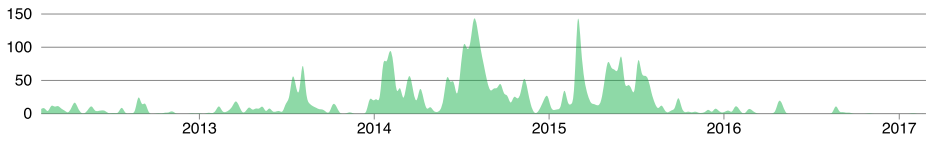

FIGURE 9. VisPy activity graph

Installation. No release on PyPI/conda yet. Was able to install it from repository after mucking around with it manually.

Strengths and weaknesses. Strengths: - Integrates well into existing Jupyter environment by building on top of IPyWidgets. - Repository control is within control of ODK participant(s), potentially avoiding conflicts of interest with other package maintainers.

Weaknesses: - Still immature project. - Installation and documentation needs work.

License. MIT

Possible contributions.

- Bring installation/distribution in line with "Jupyter standard".
- Ensure optimal use of Jupyter protocol for data transfer across kernel/browser interface.
- Documentation.
- Help mature package.

B.3.4. *VisPy.* VisPy.app / VisPy.gloo: gloo is a python -> GL abstraction layer, that supports talking to a WebGL context.

VisPy.scene: High-level visualization interface built on top of gloo. Still in development, possibly abandoned.

Installation.

Strengths and weaknesses. Strengths: - Somewhat low-level GL/WebGL access from Python via GLIR (low-level abstraction on top of various versions of GL/shader language). - Same interface whether you want to use local GL (in e.g. a Qt window) or WebGL in Notebook.

Weaknesses: - Yet another interface wrapper for GL. - User interaction and timer loops always has to round-trip to kernel side from browser. - Development seems to has stagnated:

License. BSD 3-Clause

Possible contributions.

- Take over development of high-level part.
- Add view controllers and inspectors to avoid round-trips to kernel when in browser.
- Make persisting outputs.

B.3.5. *SciviJS.* A node based framework for inspecting/deforming/shading ThreeJS based geometry. Might overlap in purpose with ParaViewWeb.

Installation. Installation via `npm`: unproblematic.

Strengths and weaknesses. Strengths: - Inspection/exploration tools. - Based on ThreeJS.
Weaknesses: - Still undocumented.

License. ISC

Possible contributions.

- Enable SciViJS as an inspector for other packages using (Py)ThreeJS
- Help mature the code and add features as needed.

B.3.6. *VPython.* Uses GlowScript (sister project) on the browser side to render GL, and VPython on the kernel side to generate scenes. Implements most things itself, which means everything is tailored for its purpose, but it loses out on the benefits of larger libraries. Code can be a bit hard to follow.

Installation. Easy install (pip install vpython worked well). A little hard to figure out which repository/package is the latest. I think this repository is the main one: https://github.com/BruceSherwood/vpytho jupyter, and `vpython` the PyPI package to install. Glowscript is bundled in a minified version in VPython repository.

Strengths and weaknesses. Strengths: - Few dependencies. - Active development.
Weaknesses: - Uses quite a lot of browser CPU when idle. - Difficult to understand code base for someone new.

License. MIT

Possible contributions.

- Clean up documentation / repositories as someone coming from outside with fresh eyes.
- Refactor codebase to ease outside contributions?
- Make persisting outputs

B.3.7. *Sage.plot3d.* Uses its own code to convert 3D plots to ThreeJS code, which it embeds into a static HTML output. As such, it is similar to Mayavi, but is based on ThreeJS instead of x3dom.

B.3.8. *VTK.js - Jupyter?* An option would be to implement Jupyter/IPython bindings for vtk.js.

Strengths and weaknesses. Strengths: - vtk.js has backing of big organization and has a mature API. - Interface should be familiar to those experienced with VTK. - Should make it easy to put ParaViewWeb on top for exploration/inspection. - Possibly good integration with Mayavi?
Weaknesses: - Mostly at mercy of Kitware and their attitude towards vtk.js (e.g. if they decide to drop it or it is poorly supported). - Unkown time-horizon.

B.4. **Concluding remarks**

The following features would all be wanted from a 3D visualization toolkit for use in Jupyter:

- It should offer a complete API for plotting 3D visualizations.
- More specialized visualizations should hopefully be able to build on top of the API for their own purposes.
- It should be open to interactive behavior, while avoiding sending redundant information between the kernel and browser.

- It should have tools for inspection/exploration that does not require a round-trip to the kernel (syncing inspection parameters is ok, but this should not block).
- It would be beneficial if the API could also be used to generate plots outside of the Notebook enviornment as well, or at least be able to switch to such a tool with little effort.
- The API can benefit greatly by being similar to existing patterns (e.g. Matplotlib or Mayavi/VTK), to ease the learning-curve for new users.

It is not necessarily aparent which package or combination of packages are the best to achieve all of these, but the following are examples of possible solutions: 0. IPyVolume + SciviJS. Uncertain difficulty of SciviJS integration. 0. K3D + SciviJS 0. "Jupyter-VTK": Jupyter bindings to VTK.js + ParaWebView. Might include Mayavi.