bioRxiv preprint doi: https://doi.org/10.1101/217349; this version posted November 10, 2017. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY 4.0 International license.

A *substrate* for modular, extensible data-visualization

Jordan Matelsky, Joseph Downs, Brock Wester, Will Gray Roncal

Johns Hopkins University Applied Physics Laboratory; Laurel, Maryland
Correspondence: jordan.matelsky@jhuapl.edu, william.gray.roncal@jhuapl.edu

November 9, 2017

Abstract

As scientific questions grow in scope and datasets grow larger, collaborative research teams and data dissemination have emerged as core research enablers. However, simply visualizing datasets is challenging, especially when sharing information across research groups or to the broader scientific community. We present `substrate`, a data-visualization platform designed to enable communication and code reuse across diverse research teams. Written in `three.js`, our platform provides a rigid and simple, yet powerful interface for scientists to rapidly build tools and effective visualizations.

1 Introduction

With modern web-frameworks like `three.js` [1] and `React` [2], it is increasingly easy to generate beautiful, interactive, and informative visualizations of scientific data. These visualizations simplify the process of exploring and sharing data with the community. In many domains (e.g. neuroinformatics, healthcare), this has become a key step of the research pipeline [3].

One challenge with these technologies is the difficulty of adapting other researchers' prior work in visualization: these tools are often built as single-purpose, not interoperable tools, and it can be difficult or even impossible to combine aspects of disparate visualization platforms, even when the platforms use the same technologies or frameworks. This challenge leads to software duplication instead of reuse and makes it difficult to share ideas across research efforts. In general, modern visualization solutions also often fail to address required capabilities, such as co-located visualization and analysis, extensibility, and data fusion [4].

Several frameworks have been designed to remedy these challenges [3, 5, 6] and we leverage some of these ideas in our solution, called `substrate`. We follow a compositional model similar in spirit to others [2, 6], but with additional functionality —including integrated Jupyter notebook capabilities found in systems such as Mayavi [5]. Specifically, we have developed `pytri`, a Python module that enables Python developers to access and interact with WebGL-based `substrate` from inside a Jupyter environment. Unlike Jupyter visualization packages such as plotly [7], `substrate` visualizatons are unopinionated and fully customizable by an end user. Users are not constrained by the limits of prepackaged visualization data structures or plot types.

`substrate` is modular — components may be added or removed without affecting the rest of the visualization; extensible — data scientists can easily use existing visualization components, and developers can easily extend or implement their own; and accessible — using `pytri`, data scientists can leverage common Python libraries such as `numpy`, `pandas`, or `networkx`.

2 Software Architecture

We present `substrate`, a JavaScript library that exposes a simple but powerful developer-facing API to help ameliorate the challenges facing modern scientific visualization. This abstraction enables visualization projects to easily share resources and logic. We first describe the architecture and design of `substrate`, and we then demonstrate use-cases in which this interoperability can reduce the engineering overhead of a new visualization project. We refer to demos and tutorials which are available at https://iscoe.github.io/substrate/.

bioRxiv preprint doi: https://doi.org/10.1101/217349; this version posted November 10, 2017. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY 4.0 International license.

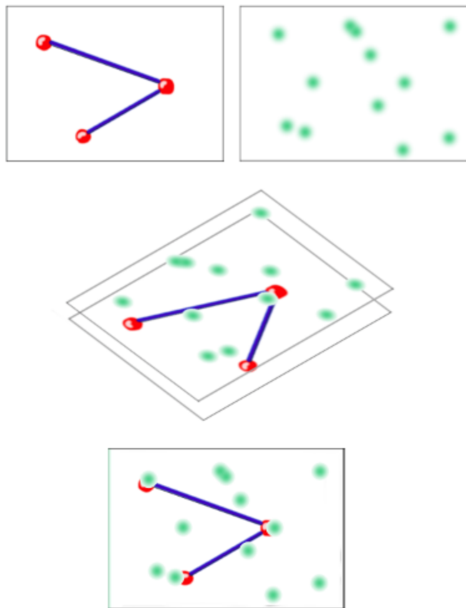

Figure 1: A GraphLayer and a ScatterLayer, representative of the same 3D space. substrate overlays the two and controls the two independently.

2.1 Design

substrate is designed to enable reusability and composability of data-visualization structures. This is implemented through an abstract Layer class. A Layer can accept arbitrary inputs and must expose render functionality to a parent Visualizer object. For example, a ScatterLayer implements Layer by accepting an array of $[x, y, z]$-tuples, and will render these data when its requestRender function is called. In the same scene, a GraphLayer might render a 3D graph embedding as shown in Figure 1.

We design a universal Layer interface that should accommodate all visualization tasks: Layers must include:

1. requestInit function, which is called before the visualization starts: This function generally includes instructions to provision objects in a 3D scene.
2. requestRender, which runs on every frame. In static, non-animated Layers, this function may be empty.
3. children is an array attribute of all objects in a scene associated with a Layer. When a Layer is removed from the visualization, all objects in this list are cleaned up by substrate internally.

This simple interface is flexible, so as to apply to any visualization object or group of objects, but still fully adequate to enable modularity and interoperability without namespacing conflicts, which are common in many three.js scenes.

```
let V = new Visualizer({
    renderLayers: {
        scatter: new ScatterLayer({ ... }),
        graph: new GraphLayer({ ... })
    }
});

V.triggerRender();
```

Figure 2: A sample Visualizer, with two Layers. One renders a 3D scatter-plot, and the other renders nodes and edges of an undirected graph.

In order to maximize accessibility, we use three.js as a convenience to wrap WebGL: Despite the prevalence of three.js in our codebase, substrate aims to be framework-agnostic. Authors of new Layers may choose to write WebGL directly, or use another wrapper or framework. substrate will support these Layers provided they subscribe to the Layer interface.

This can be expressed in code using the syntax shown in Figure 2. Here we show a simple Visualizer containing two Layers; this short snippet can run a complete visualization without any extra configuration. A data scientist need only bring her own data.

2.2 Capabilities

2.2.1 Modular Design

One common use of separate Layers is to place objects — such as a mesh — in one Layer, and place lighting or other environmental factors in another. This enables a developer to share their data visualization, such as a 3D mesh, with others, without extraneous features such as light sources. In Figure 3, we illustrate a sample implementation that can be ported to any substrate visualization. Our add-and-remove-layer demo provides an example in which a MeshLayer is added or removed, without affecting other objects in the scene.

2.2.2 Focus on Extensibility

Layers written for one visualization or application are repurposable with no additional developer-effort; this means that for most visualization use-cases, such as graph displays or scatter plots, no substrate knowledge is required at all; instead, prebuilt Layers are available for public use, including a ScatterLayer, GraphLayer, MeshLayer, and many others.

If a developer instead decides to implement her own Layer, it can be trivially integrated into new visualizations, as all substrate Layers subscribe to the same simple interface.

bioRxiv preprint doi: https://doi.org/10.1101/217349; this version posted November 10, 2017. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY 4.0 International license.

```
class ScatterLayer extends Layer {
    constructor(opts) {
        super(opts);
        // Default to empty array of points
            :
        this.points = opts.points || [];
    }

    requestInit(scene) {
        // For each point, create a sphere
            at that [x, y, z] location:
        for (let i = 0; i < this.points.
            length; i++) {
            let sphere = new window.THREE.
                Mesh(
                // A small sphere
                new window.THREE.
                    SphereGeometry(1, 16,
                    16),
                // A new color
                new window.THREE.
                    MeshLambertMaterial({
                    color: 0xc0fefe
                })
            );
            // Set the position of the mesh
                :
            sphere.position.set(...this.
                points[i]);
            // Add it to this.children so
                that it is automatically
            // marked for deletion when the
                 layer is deconstructed:
            this.children.push(sphere);
            // Add it to the scene:
            scene.add(sphere);
        }
    }
}
```

Figure 3: A sample implementation of a Layer that generates a point-cloud from the data provided in the constructor. This exact implementation can be dropped into any substrate visualization without modification. This code, and other Layer examples, are available online.

Our brownian-particle-motion example (available online) demonstrates how a developer can easily implement a new Layer, while still taking advantage of prebuilt code. We envision that users will merge these new Layers into the codebase to extend functionality and cover a diverse set of use cases. As groups work together to achieve research goals, these researchers may separately develop Layers (e.g. a raw experimental Layer and an annotation Layer for the analysis) which can be combined when needed. An example of a Layer definition is shown in Figure 3.

2.3 pytri

In order to provide a convenient visualization solution for data scientists, we have created pytri, a Python package that enables visualization of substrate Layers in a Jupyter notebook [8] or other IPython environment (as seen in Figure 4). Jupyter is increasingly a standard platform for many communities; by bringing composable, extensible functionality to this platform, data scientists can quickly visualize and explore data in a familiar paradigm without needing to understand the underlying substrate codebase.

```
In [1]: from pytri import pytri
        p = pytri()

In [2]: p.axes()
        p.mesh("./neuron.obj")
        p.show()
```

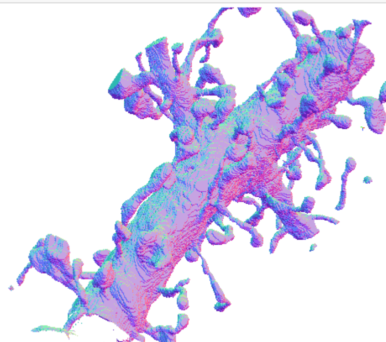

Figure 4: Here, pytri runs substrate visualizations inside a Jupyter notebook, using WebGL to enable real-time interaction with the visualization. This mesh was generated using manual annotations from a recent electron microscopy study [9].

3 Use-Cases

One of the advantages of substrate is its use as a general framework for visualization. Here we highlight two diverse applications that benefit from this package.

3.1 Neuroimagery

Neuroimaging datasets are often used as inputs to analysis pipelines to extract properties or features about the brain. For example, in diffusion weighted imaging, anatomical and connectivity information are gathered through Magnetic Resonance Imaging (MRI) sequences, which can be combined with neuroanatomical labels to produce a brain graph (i.e., a connectome) [10]. Visualizing the result is challenging with existing tools, but important for both data exploration and quality control [11]. substrate provides an elegant solution, rendering a single Layer for each of (1) raw MRI images; (2) mesh parcellations indicating brain regions; (3) fibers showing putative connections in the brain; and (4) the derived connectome graph (Figure 5).

bioRxiv preprint doi: https://doi.org/10.1101/217349; this version posted November 10, 2017. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY 4.0 International license.

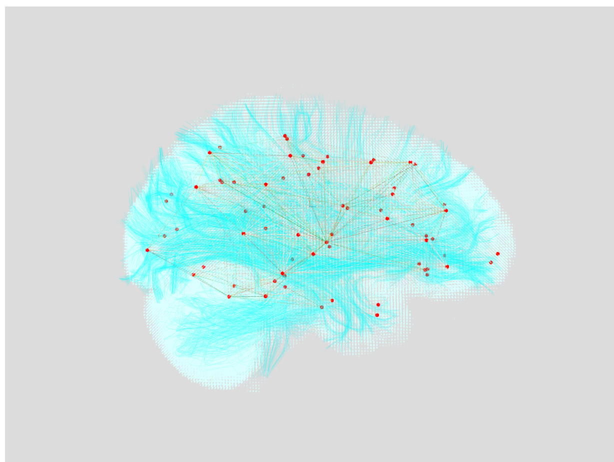

Figure 5: The ndmg pipeline [10] results for a subject are shown in a web-based substrate visualization. We show the overlay of three Layers: This visualization includes a VolumeLayer to render the diffusion-tensor imaging of the brain tissue in the form of a numpy 3D array; a FibersLayer that renders the hair-like streamlines; and a GraphLayer that shows the 3D embedding of the graph (i.e., connectome).

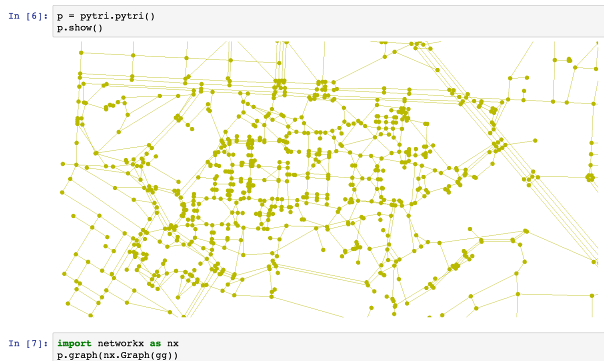

Figure 6: Using osmnx[12], pytri displays a graph representation of the roads and paths surrounding the Johns Hopkins University Homewood campus. This visualization uses a GraphLayer to represent streets, paths, and intersections as generated by the osmnx library in networkx.Graph format.

3.2 GIS Visualization in a Notebook

Geospatial information is of interest to researchers in a variety of domains; we demonstrate the ability to show a graph of street connectivity and regions of interest. This provides a flexible framework to enrich a scene as additional sensors and data fusion products become available. We use pytri to demonstrate the flexibility of the tool for data science applications substrate in a Jupyter notebook in Figure 6.

4 Discussion

substrate follows the "standalone-component" model of web development popularized by frameworks such as Angular [13] and React [2] by exposing an interface for discrete visualization entities. By compositing several of these entities, complex and deeply informative scenes can be designed with minor composition-engineering, as we demonstrate in Section 3.

There are domains that will require Layers that have not yet been developed. We intend to support some level of integration with other languages besides Python (e.g. R or Julia), based on ongoing community feedback. While we have devoted effort to maximize performance, we have optimized for mesoscale data, and have not yet optimized for very large dataset representations. Users with different tooling requirements, who require custom import formats, or very large scale visualizations (e.g. billions of vertices and edges) may need to add functionality to fully meet their requirements. We hope in the future to accommodate the layer-based or component-based visualization layers of other related projects such as Uber's deck.gl.

pytri can easily invoke substrate Layers for portable, on-the-fly visualization in a Jupyter notebook. This reduces the barrier for data scientists, who can create publication quality figures and communicate breaking analyses by sharing (or rendering) their notebooks, without having to learn JavaScript or be concerned with the implementation details of substrate. Furthermore, these analyses can be done inline with existing data science pipelines, without needing to import or export data.

This enables an individual developer to reuse their own Layers from previous projects, or to transplant and integrate Layers from their colleagues. We anticipate that this open-source contribution will enable collaboration between research teams, and reduce the overhead to produce new visualizations by localizing component-specific logic inside Layers.

We provide the codebase for substrate, documented and open-source at https://iscoe.github.io/substrate/, and welcome community feedback in the form of a pull request or bug report. We also provide demonstrations of common uses and tutorials for users to extend the current functionality. Finally, we provide a Dockerfile to enable anyone to trivially launch a pytri-enabled Jupyter notebook in their browser. pytri can be downloaded either via pypi (pip install pytri) or from our open-source repository at https://iscoe.github.io/pytri/.

bioRxiv preprint doi: https://doi.org/10.1101/217349; this version posted November 10, 2017. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY 4.0 International license.

5 Acknowledgements

We would like to thank Hannah Cowley for her work developing `substrate` `Layers` and demos, as well as for visualization prototypes informing development.

This material is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via IARPA Contract No. 2017-17032700004-005 under the MICrONS program. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation therein.

References

[1] three.js. [Online]. Available: https://threejs.org/

[2] react.js. [Online]. Available: https://reactjs.org/

[3] D. Hähn, N. Rannou, B. Ahtam, P. Grant, and R. Pienaar, "Neuroimaging in the browser using the x toolkit," in Frontiers in Neuroinformatics, 2014. [Online]. Available: https://f1000research.com/posters/1092491

[4] P. C. Wong, H. W. Shen, C. R. Johnson, C. Chen, and R. B. Ross, "The top 10 challenges in extreme-scale visual analytics," IEEE Computer Graphics and Applications, vol. 32, no. 4, pp. 63–67, July 2012.

[5] G. Varoquaux and P. Ramachandran, "Mayavi: Making 3D Data Visualization Reusable," in SciPy 2008: 7th Python in Science Conference, Pasadena, United States, Aug. 2008. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00502548

[6] deck.gl. [Online]. Available: https://uber.github.io/deck.gl/#/

[7] P. T. Inc. (2015) Collaborative data science. Montréal, QC. [Online]. Available: https://plot.ly

[8] F. Pérez and B. E. Granger, "IPython: a system for interactive scientific computing," Computing in Science and Engineering, vol. 9, no. 3, pp. 21–29, May 2007. [Online]. Available: http://ipython.org

[9] N. Kasthuri, K. Hayworth, D. Berger, R. Schalek, J. Conchello, S. Knowles-Barley, D. Lee, A. Vázquez-Reina, V. Kaynig, T. Jones, and et al., "Saturated reconstruction of a volume of neocortex," Cell, vol. 162, no. 3, p. 648–661, Jul 2015. [Online]. Available: http://dx.doi.org/10.1016/j.cell.2015.06.054

[10] G. Kiar, W. Gray Roncal, D. Mhembere, E. Bridgeford, R. Burns, and J. T. Vogelstein, "ndmg: Neurodata's mri graphs pipeline," Aug. 2016. [Online]. Available: https://doi.org/10.5281/zenodo.60206

[11] P. Rudolph, "Realtime visualization of the connectome in the browser using webgl," Frontiers in Neuroinformatics, vol. 5, 2011. [Online]. Available: http://dx.doi.org/10.3389/conf.fninf.2011.08.00095

[12] G. Boeing, "Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks," CoRR, vol. abs/1611.01890, 2016. [Online]. Available: http://arxiv.org/abs/1611.01890

[13] M. Ramos, M. T. Valente, R. Terra, and G. Santos, "Angularjs in the wild: A survey with 460 developers," CoRR, vol. abs/1608.02012, 2016. [Online]. Available: http://arxiv.org/abs/1608.02012