nature portfolio

https://doi.org/10.1038/s41592-022-01482-7

Supplementary information

Viv: multiscale visualization of high-resolution multiplexed bioimaging data on the web

Supplementary Note 1

Motivation

Visualization of highly multiplexed, high-resolution tissue datasets on the web relies on client software that displays server-side rendered images[1,2] rather than rendering data in the client. The JavaScript clients for many of these applications, such as Minerva[2], Facetto[3], Human Protein Atlas Microscopy Viewer[4], OMERO iViewer[5], Digital Slide Archive[6], and Cytomine[7] are based on tools like OpenSeadragon[8] or OpenLayers[9], which are popular web-based viewers for zoomable images. The viewers load image tiles encoded in RGB/A data formats that are natively supported by web browsers (e.g., PNG, JPEG). Therefore primary high-resolution datasets, including those stored in open standard formats, must first be rendered to image tiles in order to use these tools. Image tiles can be rendered dynamically, as with client-server models like OMERO[5], or statically, where each level of the multi-resolution representation (pyramidal data) is pre-rendered as a nested directory of PNG or JPEG files (see *Figure 1*).

Whole-slide datasets derived from bright-field assays such as Hematoxylin and Eosin (H&E) stains or Periodic acid-Schiff (PAS) stains only contain red, green, and blue (RGB) channels and do not require additional data transformations to render. This means current web technologies are sufficient for viewing these datasets, despite the overhead of conversion to browser-friendly image tiles, since only zooming and panning interactions are necessary for exploration.

Datasets produced by multiplexed imaging methods[10–13], however, are instead most often stored as multidimensional stacks of 2D planes where each plane—or channel—typically represents a chemical signal, such as a fluorophore bound to an antibody in immunofluorescence. Images derived from non-targeted methods like imaging mass spectrometry[14] (IMS) are stored similarly with a deep m/z dimension. Multidimensional stacks (including T, C, Z dimensions) may contain over a hundred 16- or 32-bit grayscale planes that must be pseudo-colorized for visualization, and it is desirable to view each channel separately since the specificity of the acquisition methods is high. Therefore, effective interactive analysis of these datasets require both zoom and pan interactions as well as rapid switching between groups of channels via sophisticated rendering.

Multi-Channel Rendering

Multi-channel rendering involves the blending of data from separate channels into a single image, allowing users to view isolated signals or blended composites during analysis. Individual channel data transformations are applied per pixel to generate a final image. Efficient multi-channel rendering is especially important to oncologists and pathologists who rely on seamless visual experience when making a diagnosis[3]. Multi-channel color mapping is increasingly important in modern cell-based image analysis, where many channel combinations are possible with limited overlap, and channels may be grouped according to some biologically relevant criteria[3].

Desktop applications are the current gold standard for interactive analysis and support many types of rendering. In contrast to existing web-based viewers, desktop-oriented programs make use of low-latency connections to primary data and leverage graphics cards for rapid rendering. Specific software and hardware requirements, however, restrict the availability of these tools to a wider scientific audience, and the coupling of data storage and computational environment limit the user experience when viewing remote data, if supported, over low-latency connections.

Most existing JavaScript tools for viewing multiscale images do not implement multi-channel rendering, and instead tightly coupled client-server architectures are utilized to enable interactive analysis of highly multiplexed datasets on the web[3,15]. It is important to note that this approach strictly avoids sending raw data to the client, which has the benefit of smaller data transfer per request, but ultimately the client is totally dependent on the server for information. Sampling many rendering settings within a server-rendered viewer incurs a server-round-trip and additional data transfer per configuration, whereas loading raw data on the client is expensive upfront but subsequent re-renders are instantaneous and do not require additional data transfer.

Offline server-side rendering is accomplished by applying a set of data transformations to primary imaging data *prior* to configuring a web server, yielding a unique rendered copy of the original dataset (directory hierarchy of JPEG or PNG image tiles) for each combination of channel groupings and data transformations. Although this approach is simple to deploy, it prevents end users from on-demand visual exploration since data transformations and channel groupings are fixed and only determined offline. Consequently, web viewers of this type are tailored towards explanatory visualization where users are guided through a narrative that accompanies the pre-rendered image pyramids[2].

In online server-side rendering, users define channel groupings and data transformations via a web user interface, and then a server applies the desired transformations on-the-fly and renders image tiles to send back to the client. While this approach has been implemented successfully, it requires substantial maintenance and the tight coupling of client and server inhibits the use of either in isolation. User experience also degrades when the server is at capacity or over a slow network connection since adjusting individual channel transformations, sampling channel groupings, or toggling channel visibility requires the client to await a new server-rendering. In contrast, identical user interactions in analogous desktop software yield low-latency, continuous updates since graphics cards are exploited to quickly apply new data transformations to the same in-memory data.

WebGL (Web Graphics Library) allows similar access to the GPU as desktop software but has yet to be applied to enable multi-channel rendering of high-resolution, multiplexed image data on the web. Existing WebGL-based viewers showcase the potential for complex browser-based rendering but are primarily designed for single-channel volumetric datasets (typically lossy-compressed, 8-bit dense arrays), rely on data to fit in-memory for visualization, or still use custom server implementations[16–19]. Additionally, these viewers are fully-integrated applications and as such do not provide reusable components for building or extending existing interactive bioimaging visualizations.

Name	Description	URL
WebGL	Low-level JavaScript API for rendering high-performance interactive graphics.	https://www.khronos.org/webgl/
WebAssembly	Portable binary-code format that is executable in modern web browsers.	https://www.w3.org/TR/wasm-core-1/
WebWorkers	Web API to run scripts in a separate thread from the main execution thread.	https://www.w3.org/TR/workers/
geotiff.js	A JavaScript library to parse TIFF files for web-based visualization.	https://geotiffjs.github.io/
Zarr.js	A TypeScript implementation of Zarr[20].	https://github.com/gzuidhof/zarr.js/
Deck.gl[21]	WebGL-powered framework for high-performance visualization.	https://deck.gl/
Bio-Formats[15]	Software tool suite for reading and writing image data using standardized, open formats.	https://www.openmicroscopy.org/bio-formats/
bioformats2raw	Java application to convert proprietary image file formats to intermediate N5/Zarr structure.	https://github.com/glencoesoftware/bioformats2raw
raw2ometiff	Java application to convert outputs of bioformats2raw to an OME-TIFF pyramid.	https://github.com/glencoesoftware/raw2ometiff
Blosc[22]	Lossless compression library used by Zarr[20].	https://github.com/Blosc/c-blosc

Supplementary Table 1. *Software libraries and web technologies used by Viv.*

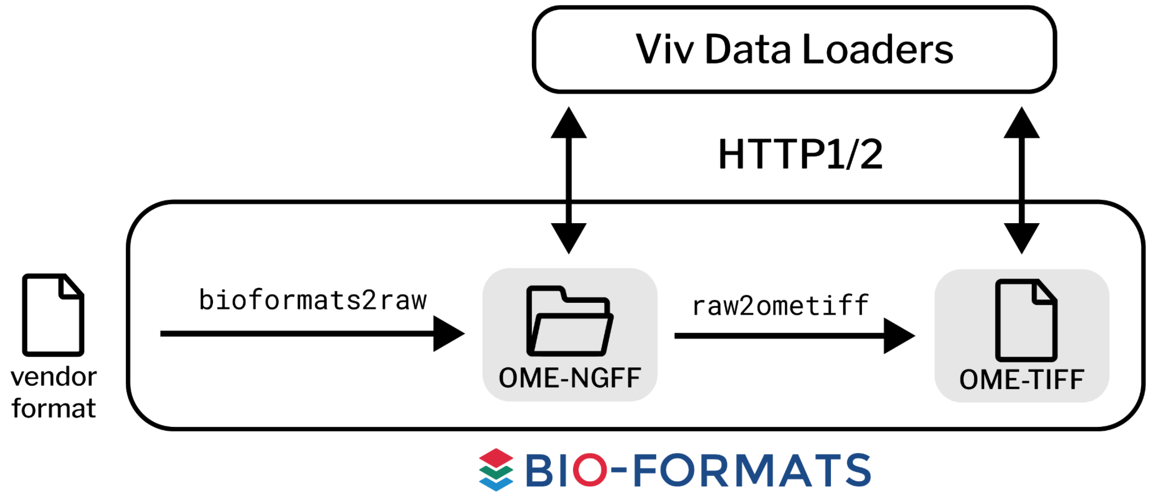
Software Architecture

Viv is a JavaScript library that provides GPU-accelerated multi-channel rendering of high-resolution multiplexed images directly in the web browser. It is implemented in the TypeScript programming language (https://www.typescriptlang.org), allowing ease of use in other JavaScript and TypeScript projects. As a design philosophy, we built Viv on top of open standard image formats created by the Open Microscopy Environment (OME), OME-TIFF[23] and OME-NGFF[24]. Therefore primary data which can be viewed by popular desktop software[25–27] may also be viewed directly by Viv without converting to a transient browser-friendly format.

Due to its modular architecture, Viv is embeddable and can be deployed to support visualization for exploration and explanation in a wide range of different settings (doi:10.6084/m9.figshare.19416401) Since Viv operates on multi-resolution data formats, the rendering performance is not affected by the size of the images. We designed Viv as a modular JavaScript library using modern web technologies (see *Supplementary Table 1*) to support dynamic fetching, decoding, and rendering of pyramidal multiplex datasets. The data loading component of Viv is built on the geotiff.js and zarr.js libraries, to which we contributed additional features to support efficient data chunk retrieval. Viv exposes its rendering functionality as reusable Deck.gl[21] layers, allowing the composition of multiple image sources in a single interactive view. The Deck.gl library has no dependencies on web user interface frameworks and can be used in any JavaScript application, enabling Viv to be incorporated into existing client software with little overhead. Finally, Viv offers several custom high-level React components that handle complex rendering and interactivity, such as overview and detail and multiple linked views.

Data Preparation

Bio-Formats[15] is a software tool for reading proprietary microscopy image data and metadata using standardized, open formats. It provides the ability to translate over 150 file formats and associated metadata to the OME data model. OME-TIFF has been available for over a decade and is a common format for sharing imaging data. It is recommended and used by the Image Data Resource[28] and various consortia such as Human BioMolecular Atlas Program[29] and the 4D Nucleome Consortium[30]. OME-NGFF is a complementary open format that is designed to address fundamental limitations of OME-TIFF at scale. The format is under active collaborative development by the OME community as a cloud-friendly solution to provide flexible storage of multidimensional datasets from established and emerging imaging assays.



Supplementary Figure 1. *Viv Bio-Formats compatibility. The Bio-Formats command-line tools are used sequentially to generate OME-NGFF and OME-TIFF images. The* `bioformats2raw` *utility creates OME-NGFF which can be converted to an OME-TIFF via* `raw2ometiff`*. Viv's data loader utilities are compatible with both OME-NGFF and OME-TIFF formats.*

Viv supports viewing both OME-NGFF and OME-TIFF directly via HTTP (Supplementary Figure 1). Writing OME-NGFF and OME-TIFF can be accomplished using the Bio-Formats command-line suite. The `bioformats2raw` utility (v0.4.0, https://github.com/glencoesoftware/bioformats2raw) is responsible for converting proprietary image formats to Zarr, generating pyramidal levels from large resolution planes if not available. The Bio-Formats pipeline subsequently converts this representation to OME-TIFF via `raw2ometiff` (v0.3.0, https://github.com/glencoesoftware/raw2ometiff).

Once created, Viv can access these data on-demand via HTTP. A local web-server is sufficient for viewing datasets locally, and full web applications may be deployed by uploading the same datasets to a commodity web-server or commercial cloud object storage (e.g., Amazon S3, Google Cloud Storage, Microsoft Azure Blob).

Data Loading

Viv's data loaders provide a consistent interface for retrieving data tiles from arbitrary sources. Data loaders fetch individual compressed chunks from OME-TIFF or OME-NGFF via HTTP, and subsequently decompress these data for rendering. Custom loaders may be implemented to support additional data sources.

An important component of Viv's data loaders is the support for Zarr[20], the binary format underlying OME-NGFF. Zarr is an open-source format for the storage of chunked, compressed, multidimensional arrays. The original implementation of Zarr is written in Python, but its popularity has led to implementations in several languages (C++, Java, Julia, JavaScript). The underlying compressed chunk data and array metadata can be saved in *any* key-value store, most commonly a local file directory or cloud object storage. This flexibility allows the configuration of custom storage for application-specific needs, meaning various backends can be utilized to support visualizations with Viv. For example, our Viv-based Vizarr viewer implements a custom Zarr interface to securely transfer data from a Python backend via the ImJoy[31] Remote Procedure Call (https://github.com/imjoy-team/imjoy-rpc).

Data loading from OME-TIFF files is handled through HTTP byte-range requests. OME-TIFF defines its multi-resolution representation via TIFF Sub-Image File Directories (SubIFDs) (https://www.adobe.io/content/dam/udp/en/open/standards/tiff/TIFFPM6.pdf) which are accessed on the client to load specific sub-resolutions. Simply reading OME-TIFF via HTTP is limited, however, due to TIFF's linear binary layout which was designed for local filesystems.

We propose the use of a complementary pre-computed index to avoid seeking on the client and improve the efficiency of random chunk access. We developed a python command line utility (and website) to generate an OME-TIFF index (JSON) containing the corresponding byte-offsets for each TIFF Image File Directory (IFD) (https://github.com/hms-dbmi/generate-tiff-offsets). Viv utilizes the byte-offsets to skip the otherwise required step of linearly seeking the series of IFDs, providing more direct access to individual chunks. We found that OME-TIFF with an IFD index (Indexed OME-TIFF) substantially reduces OME-TIFF chunk access latencies (doi:10.6084/m9.figshare.19416344). Critically, our method is scalable since the total number of byte-offsets in an IFD index is independent of the number of pyramidal sub-resolutions for a dataset.

Data retrieval for all loaders is done in an asynchronous event loop so that multiple compressed chunks can be fetched concurrently. Web Workers are used to perform chunk decoding on separate threads, providing parallelism and freeing the user interface to remain responsive to user interactions. Popular image compression methods have been ported to JavaScript previously, but desktop software typically relies on libraries written in low-level languages like C or C++ for performing binary decompression. WebAssembly enables the reuse of these same libraries in a web browser with similar performance. We compiled the modern high-performance compressor Blosc[22] (Zarr default), Lempel-Ziv-Welch (LZW)[32] (OME-TIFF default), and the popular Zstd and LZ4 algorithms to WebAssembly to support rapid and flexible decoding.

Rendering Component

The multi-channel rendering component of Viv is implemented as custom Deck.gl layers. Deck.gl is a WebGL-powered framework for exploratory visualization of large, spatial datasets. A layer is a core concept of Deck.gl. It describes a packaged visualization type that combines a collection of data and methods to render these data in a shared coordinate system. Interactive Deck.gl visualizations can be constructed by composing layers with others (points, polygons, text annotations, etc.), enabling highly customizable and efficient rendering of complex scenes.

Viv layers control what is rendered when a user interacts with the WebGL canvas. The declarative layer API provides the ability to define specific channel selections from a multidimensional source as well as desired data transformations per channel. Additionally, affine transformations are supported per layer via a 4x4 transformation matrix. These parameters can be updated within a reactive paradigm, enabling GPU-accelerated rendering across modern web frameworks and user interfaces. Each Viv layer uses a data loader to retrieve individual chunks for the corresponding channel selections as a user zooms and pans in the coordinate space. Chunks are fetched and decoded by the loaders and then loaded on the GPU, where shaders apply the user-defined data transformations. Once the data have been retrieved for a particular region, changes to desired transformations (contrast limits, opacity, color mapping, visibility) simply re-render using the previously loaded data. This creates a low-latency user experience when exploring channel combinations and data transformations.

To enable the rapid exploration of spatial distributions and correlations between channels, Viv provides a reactive API for blending the data from different channels into a single image layer. Data chunks contain the pixel intensities for individual channels, and once bound to the GPU, data transfer functions are applied one of two ways:

1. **Additive Blending.** Each channel is assigned an RGB (or equivalently, HSV) color value, defining a linear color transfer function that maps black to the minimum value and the color to the maximum value per channel. The contributions of individual channels are

then additively blended into a single RGBA image following min-max normalization. The respective contrast limits, as well as the colors for each transfer function, are exposed via the Viv Layer API. This type of additive blending ensures non-overlapping colors for up to three channels when using pure red, green, and blue. Viv currently supports up to six concurrently rendered channels per image, which is useful when viewing additional channels that have little to no spatial overlap.
2. **Additive Color Mapping.** The second option is to use a single transfer function that maps the combined channel intensities to a colormap such as Viridis or Magma. Each channel intensity is min-max normalized to 0-1, and all normalized intensities are summed. Sum is then used as the input for a transfer function. This method is similar to the lookup tables supported by OMERO iViewer[5]. We use glslify[33] to inject transfer functions into the Viv shaders which scale intensity values to RGB colors.

When multiple images are loaded into Viv, alpha compositing between layers is supported. Additionally, rendering is completely flexible in Viv, and library users are able to implement custom shaders to modify the builtin behavior described above.

Viv API

The Viv API comprises the following major elements. Details are available in the documentation (http://viv.gehlenborglab.org).

1. **Viewers.** Drop-in React components that expose interfaces for developers to supply controllers for the various rendering settings. Handle complex rendering and interactivity, such as overview and detail and multiple linked views.

2. **Views.** Building blocks for supporting viewers with multiple views, like side-by-side or picture-in-picture. A view is a stateful object defined by a particular zoom level and bounding box.

3. **Layers.** Control what spatial regions and channels to render in each view, along with what data transformations to apply per channel. Viv provides `MultiscaleImageLayer` and `ImageLayer` layers that support rendering pyramidal and nonpyramidal images, respectively. The `VolumeLayer` supports rendering 3D views via ray casting.

4. **Loaders.** Shared interface for accessing the metadata and channel data from OME-TIFF and OME-NGFF via HTTP.

Supplementary References

1. Aeffner, F. *et al.* Introduction to Digital Image Analysis in Whole-slide Imaging: A White

Paper from the Digital Pathology Association. *J. Pathol. Inform.* **10**, 9 (2019).

2. Rashid, R. *et al.* Interpretative guides for interacting with tissue atlas and digital pathology data using the Minerva browser. *bioRxiv* 2020.03.27.001834 (2020) doi:10.1101/2020.03.27.001834.
3. Krueger, R. *et al.* Facetto: Combining Unsupervised and Supervised Learning for Hierarchical Phenotype Analysis in Multi-Channel Image Data. *IEEE Trans. Vis. Comput. Graph.* **26**, 227–237 (2020).
4. Uhlen, M. *et al.* Towards a knowledge-based Human Protein Atlas. *Nat. Biotechnol.* **28**, 1248–1250 (2010).
5. Allan, C. *et al.* OMERO: flexible, model-driven data management for experimental biology. *Nat. Methods* **9**, 245–253 (2012).
6. Gutman, D. A. *et al.* The Digital Slide Archive: A Software Platform for Management, Integration, and Analysis of Histology for Cancer Research. *Cancer Res.* **77**, e75–e78 (2017).
7. Rubens, U. *et al.* Cytomine: Toward an Open and Collaborative Software Platform for Digital Pathology Bridged to Molecular Investigations. *Proteomics Clin. Appl.* **13**, e1800057 (2019).
8. OpenSeadragon. https://openseadragon.github.io/.
9. Gratier, T., Spencer, P. & Hazzard, E. *OpenLayers 3 : Beginner's Guide*. (Packt Publishing Ltd, 2015).
10. Goltsev, Y. *et al.* Deep Profiling of Mouse Splenic Architecture with CODEX Multiplexed Imaging. *Cell* **174**, 968–981.e15 (2018).
11. Rashid, R. *et al.* Highly multiplexed immunofluorescence images and single-cell data of immune markers in tonsil and lung cancer. *Sci Data* **6**, 323 (2019).

12. Ijsselsteijn, M. E., van der Breggen, R., Farina Sarasqueta, A., Koning, F. & de Miranda, N. F. C. C. A 40-Marker Panel for High Dimensional Characterization of Cancer Immune Microenvironments by Imaging Mass Cytometry. *Front. Immunol.* **10**, 2534 (2019).

13. Lin, J.-R., Fallahi-Sichani, M. & Sorger, P. K. Highly multiplexed imaging of single cells using a high-throughput cyclic immunofluorescence method. *Nat. Commun.* **6**, 8390 (2015).

14. Neumann, E. K., Djambazova, K., Caprioli, R. M. & Spraggins, J. M. Multimodal Imaging Mass Spectrometry: Next Generation Molecular Mapping in Biology and Medicine. *J. Am. Soc. Mass Spectrom.* (2020) doi:10.1021/jasms.0c00232.

15. Moore, J. *et al.* OMERO and Bio-Formats 5: flexible access to large bioimaging datasets at scale. in vol. 9413 941307 (International Society for Optics and Photonics, 2015).

16. itk-vtk-viewer. *itk-vtk-viewer* https://kitware.github.io/itk-vtk-viewer/index.html.

17. neuroglancer. https://github.com/google/neuroglancer.

18. Boergens, K. M. *et al.* webKnossos: efficient online 3D data annotation for connectomics. *Nat. Methods* **14**, 691–694 (2017).

19. Saalfeld, S., Cardona, A., Hartenstein, V. & Tomancak, P. CATMAID: collaborative annotation toolkit for massive amounts of image data. *Bioinformatics* **25**, 1984–1986 (2009).

20. Miles, A. *et al. zarr-developers/zarr-python: v2.4.0*. (2020). doi:10.5281/zenodo.3773450.

21. Wang, Y. Deck. gl: Large-scale web-based visual analytics made easy. *arXiv preprint arXiv:1910.08865* (2019).

22. Alted, F. Blosc, an extremely fast, multi-threaded, meta-compressor library. https://blosc.org/.

23. Goldberg, I. G. *et al.* The Open Microscopy Environment (OME) Data Model and XML file: open tools for informatics and quantitative analysis in biological imaging. *Genome Biol.* **6**,

R47 (2005).

24. Moore, J. *et al.* OME-NGFF: a next-generation file format for expanding bioimaging data-access strategies. *Nat. Methods* 1–3 (2021).
25. Sofroniew, N. *et al. napari/napari: 0.2.8*. (2019). doi:10.5281/zenodo.3592005.
26. Schindelin, J. *et al.* Fiji: an open-source platform for biological-image analysis. *Nat. Methods* **9**, 676–682 (2012).
27. Bankhead, P. *et al.* QuPath: Open source software for digital pathology image analysis. *Sci. Rep.* **7**, 16878 (2017).
28. Williams, E. *et al.* The Image Data Resource: A Bioimage Data Integration and Publication Platform. *Nat. Methods* **14**, 775–781 (2017).
29. Consortium, H. & HuBMAP Consortium. The human body at cellular resolution: the NIH Human Biomolecular Atlas Program. *Nature* vol. 574 187–192 (2019).
30. Dekker, J. *et al.* 4D Nucleome Network. *Corrigendum: The 4D nucleome project. Nature* **552**, 278 (2017).
31. Ouyang, W., Mueller, F., Hjelmare, M., Lundberg, E. & Zimmer, C. ImJoy: an open-source computational platform for the deep learning era. *Nat. Methods* **16**, 1199–1200 (2019).
32. Welch, T. A. A technique for high-performance data compression. *Computer* 8–19 (1984).
33. *WebGL Insights*. (A K Peters/CRC Press, 2015).