Viv: Multiscale Visualization of High-Resolution Multiplexed Tissue Data on the Web

Trevor Manz¹, Ilan Gold¹, Nathan Heath Patterson², Chuck McCallum¹, Mark S Keller¹, Bruce W Herr Il³, Katy Börner³, Jeffrey M Spraggins², Nils Gehlenborg¹,*

¹ Harvard Medical School, Boston, MA, USA
² Vanderbilt University, Nashville, TN, USA
³ Indiana University, Bloomington, IN, USA

* Corresponding author: nils@hms.harvard.edu

Abstract

Recent advances in highly multiplexed imaging have enabled the comprehensive profiling of complex tissues in healthy and diseased states, facilitating the study of fundamental biology and human disease in spatially-resolved contexts at subcellular resolution. However, current computational infrastructure to distribute and visualize these data on the web remains complex to set up and maintain. To address these limitations, we have developed Viv—an open-source image visualization library for high-resolution multiplexed image data that is implemented in JavaScript and builds on modern web technologies. Viv directly renders Bio-Formats-compatible Zarr and OME-TIFF data formats. Three use cases, including integration into Jupyter Notebooks (https://github.com/hms-dbmi/vizarr) and a data portal, as well as an image viewer (http://avivator.gehlenborglab.org) demonstrate the capabilities of our proposed approach.
Manuscript

Recent advances in highly multiplexed imaging\textsuperscript{1-4} have enabled the comprehensive profiling of complex tissues in healthy and diseased states, facilitating the study of fundamental biology and human disease in spatially-resolved contexts at subcellular resolution\textsuperscript{5,6}. However, current computational infrastructure to distribute and visualize these data on the web remains complex to set up and maintain. Most primary, high-resolution images continue to be shared only as static figures in publications\textsuperscript{7}. To address these limitations, we have developed Viv—an open-source image visualization library for high-resolution multiplexed image data that is implemented in JavaScript and builds on modern web technologies. Viv directly renders Bio-Formats-compatible\textsuperscript{8} Zarr\textsuperscript{9} and OME-TIFF\textsuperscript{10} data formats and is therefore compatible with over 150 image formats.

Existing image visualization tools support the interactive viewing of high-resolution, multiplexed images across scales. Desktop applications can analyze and visualize primary image data stored on local or network drives\textsuperscript{7}, but require installing software and often downloading the image data, as well as hardware with sufficient memory and disk space. Web-based applications, on the other hand, are designed from the ground up for viewing remote data without the need to download gigabyte-sized datasets or to install any software. There are two popular approaches to create web-based visualization tools for high-resolution multiplexed image data (see Figure 1). Both methods rely on server-side image rendering that converts primary data—usually in pyramidal form—into PNG or JPEG image tiles that are then displayed by the browser-based client. The offline option relies on image tiles that are rendered based on a predefined data transformation, stored, and served via a standard HTTP web server. The online option relies on a specialized server that renders primary data into image tiles on-demand based on a user-defined data transformation.

The offline option requires the preselection of channels and color mappings, preventing effective exploration of highly multiplexed datasets. This option also introduces latency when exploring data transformations and requires users to set up, maintain, and potentially scale server software, which is complex and resource-intensive.

Viv implements client-side image rendering to create a solution that does not rely on a specialized server but offers the flexibility of on-demand multi-channel rendering. The library consists of two components: (1) modules for accessing and decoding raw intensities from standard data formats generated by the Bio-Formats library and (2) WebGL shaders that render primary data on the GPU as Deck.gl\textsuperscript{12} layers.
**Figure 1.** Overview of data flow and image generation approaches for web-based image data visualization. DT indicates the location where data is transformed into an image. HTTP represents a web server.

The data loading component of Viv is designed to support dynamic access to data chunks via HTTP. Data loader modules are responsible for retrieving and decoding compressed data tiles. Currently, Viv supports OME-TIFF and multiscale, OME metadata-annotated Zarr formats stored on file servers or cloud object storage, but additional modules can be added to extend Viv with support for other data sources. Zarr provides a specification for implementing chunked, compressed, N-dimensional arrays (see Supplementary Note 1), and OME-Zarr is an emerging standard supported by the Open Microscopy Environment (OME) community.

Viv’s rendering component uses a custom Deck.gl layer to manage retrieval of data tiles via data loaders that fetch and decompress the data, which is then uploaded directly into the GPU. WebGL shaders render the image based on user-defined properties, enabling continuous and immediate updates of properties such as color mapping, opacity, and channel visibility without additional data transfer. Channel intensities can be displayed continuously and magic lenses that locally transform the data on the fly, e.g., to rescale brightness or to filter specific channels, can be implemented efficiently.

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 (see Supplementary Note 2). For example, we integrated Viv into Jupyter Notebooks as an ImJoy plugin to demonstrate how it can be used to enable a human-in-the-loop multimodal image registration workflow. Viv was also integrated into an Angular-based (https://angular.io), ontology-driven exploration tool
for human tissue datasets, further establishing its utility and compatibility. Finally, we developed Avivator (http://avivator.gehlenborglab.org), an image viewer based on Viv that can be used to make primary image data accessible for visual exploration without the need for a specialized server.

The ability to render primary data from community-supported standard formats makes the approach implemented by Viv an efficient and effective solution for a wide range of applications due to minimal deployment requirements and rich user experience. The use of Deck.gl provides a foundation for extensions of the core Viv functionality, for example, vector-based annotation layers and 3D views. By supporting client-side image rendering in combination with Zarr, Viv introduces a new model to enable a seamless transition between computationally intensive data analysis and interactive visual exploration independently of the location of the data.

Acknowledgments

Viv was developed with funding from the National Institutes of Health (OT2OD026677, T15LM007092) and the Harvard Stem Cell Institute (CF-0014-17-03). The use cases in this manuscript were supported with additional funding from the National Institutes of Health (U54DK120058, 2P41GM103391, OT2OD026671) and National Science Foundation (CBET 1828299). We would also like to thank members of the OME community for their guidance as well as Mark deCaestecker, Elizabeth Neumann, and Maya Brewer from Vanderbilt University and Vanderbilt University Medical Center for their efforts generating the microscopy data highlighted in the Use Cases.

Code Availability


References


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


Supplementary Note 1

Motivation

Visualization of highly multiplexed, high-resolution tissue data on the web relies on client software that displays server-side rendered images\textsuperscript{1,2} rather than rendering images in the client. The JavaScript clients for many of these applications, such as Minerva\textsuperscript{2}, Facetto\textsuperscript{3}, Human Protein Atlas Microscopy Viewer\textsuperscript{4}, OMERO iViewer\textsuperscript{5}, Digital Slide Archive\textsuperscript{6}, and Cytomine\textsuperscript{7} are based on tools like OpenSeadragon\textsuperscript{8} or OpenLayers\textsuperscript{9}, which are popular web-based viewers for zoomable images. The viewers load image tiles encoded in RGB data formats that are natively supported by web browsers (e.g., PNG, JPEG). However, primary multi-scale images must first be rendered into image pyramids to be supported by these tools, resulting in hundreds or thousands of individual JPEG or PNG image tiles that are organized in a hierarchy. Image tiles can be rendered dynamically, as with client-server models like OMERO\textsuperscript{5}, or statically, where every pyramidal level is generated from the source, creating a complete rendering of the primary data which can be hosted with a simple web-server (see Figure 1).

For high-resolution, whole-slide images from bright-field assays such as Hematoxylin and Eosin (H&E) stains or Periodic acid-Schiff (PAS) stains, the resulting data only contain red, green, and blue (RGB) channels and do not require special rendering. This means current web-technologies are sufficient for viewing primary RGB images, despite requiring the conversion of the pyramidal, high-resolution data to browser-friendly images since zooming and panning are sufficient for interactive exploration.

Primary data produced by multiplexed imaging methods\textsuperscript{10-13}, however, are instead stored as multi-dimensional image stacks where each image—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\textsuperscript{14} (IMS) can be stored similarly with very deep stacks along the m/z dimension. These multi-dimensional stacks may contain over a hundred 16- or 32-bit images 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 primary data requires both zoom and pan interactions as well as sophisticated rendering that enables rapid switching between groups of channels.

Multi-Channel Rendering

Multi-channel rendering involves the blending of data from separate channels into a single image, allowing users to choose a single channel or blended composites of multiple channels 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\textsuperscript{3}. Although it is difficult to interpret resulting pixel colors when mixing different channels, multi-channel color mapping is increasingly important in state-of-the-art cell-based image analysis where many channel combinations are possible with limited spatial overlap\textsuperscript{3}.

Desktop applications are particularly good for interactive analysis and can support many types of rendering. In contrast to existing web-based viewers, desktop software makes use of high-bandwidth connections to primary data and utilizes graphics cards to apply data transformations for image rendering. Specific software and hardware requirements, however, limit the availability of these tools to a wider scientific audience, and support for viewing remote data over HTTP is often a secondary concern.

Since existing JavaScript tools for viewing multiscale images do not support multi-channel rendering, client-server architectures must be implemented to enable interactive analysis of primary highly multiplexed images on the web\textsuperscript{5,15}. Offline server-side rendering in combination with basic web servers are not a practical option for exploratory viewing because pre-rendering even a fraction of potential channel groupings for images with more than a few channels is computationally intractable. Consequently, previously described web-based viewers\textsuperscript{2} that utilize web servers with access to file storage or commercial cloud storage to share multiplexed imaging data are used primarily for explanatory rather than exploratory visualization since all channel groupings and rendering must be pre-determined.

Under the client-server model, a user defines color and opacity transformations to apply to individual channels from a web UI, and then a server is responsible for rendering browser-compatible RGB image tiles on-demand as the user interacts with a zoomable view. Therefore, a user must wait for new renderings of the primary data when adjusting individual channel transformations, sampling channel groupings, or toggling channel visibility to investigate spatial overlap. These same interactions, in contrast, yield low-latency, continuous updates in desktop software where graphics cards are exploited to quickly re-render the primary image 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\textsuperscript{16,17} showcase the potential for complex browser-based rendering, but these tools are primarily designed for single-channel volumetric data or rely on data to fit in-memory for visualization. Additionally, current viewers lack a declarative API for composing images from different sources within the same spatial context and for embedding viewers within existing UI frameworks.

Software Architecture

To address current limitations in web-based image viewers, we created Viv to provide performant multi-channel rendering of high-resolution multiplexed images directly in the browser. As a design philosophy, we built Viv on top of open standard formats for primary data
created by the OME community, OME-TIFF\cite{Bach2010}, and Bio-Formats “raw” Zarr, relying on dynamic access to these sources over HTTP. Compatibility with Bio-Formats\cite{Bach2010} means that primary data which can be viewed by popular desktop software\cite{Bach2010-1,Bach2010-2} can also be viewed directly by Viv, without the need to convert to some intermediate browser-friendly format.

By moving all rendering to the browser, Viv doesn’t depend on server-rendering like previous web-viewers for multi-scale images, making it flexible and embeddable within a variety of applications. Since Viv operates on pyramidal image formats, the rendering of the primary data in the clients is not affected by the size of the images. We designed Viv as a modular JavaScript library and built it with modern web technologies (WebGL: https://www.khronos.org/webgl/, WebAssembly: https://www.w3.org/TR/wasm-core-1/, Web Workers: https://www.w3.org/TR/workers/) to support dynamic fetching, decoding, and rendering of primary multi-channel images. The data loader modules in Viv are built on the geotiff.js\cite{Bach2010-3} and zarr.js\cite{Bach2010-4} libraries, to which we contributed additional features to support efficient data chunk retrieval. The rendering components of Viv are packaged as Deck.gl\cite{Bach2010-5} layers, which provides a declarative API for users to compose separate image sources together in a single interactive view. The Deck.gl core library has no dependencies on web UI 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\cite{Bach2010} 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 their metadata to the open OME data model. OME-TIFF has been available for over a decade and is a common format for sharing imaging data used by the Image Data Resource\cite{Bach2010-6} and various consortia such as Human BioMolecular Atlas Program\cite{Bach2010-7} and the 4D Nucleome Consortium\cite{Bach2010-8}. An extension to the OME-TIFF format provides support for pyramidal resolutions, which can be created using the Bio-Formats command-line utilities. OME-Zarr is a developing standard from the OME community. It is meant to move beyond a single binary format like OME-TIFF and extend the OME Model to describe both multi-scale and multi-dimensional data. It is under active collaborative development by the OME community as a cloud-friendly format that can meet the data volumes of established and emerging imaging assays.
Supplementary Figure 1. Viv Bio-Formats compatibility. The Bio-Formats command-line tools are used sequentially to generate Zarr and OME-TIFF pyramidal and non-pyramidal images. The `bioformats2raw` creates a “raw” intermediate Zarr or N5 which is subsequently converted to an OME-TIFF via `raw2ometiff`. Viv’s data loader utilities are compatible with both the “raw” Zarr and OME-TIFF formats.

Viv is designed to access both pyramidal and nonpyramidal Bio-Formats “raw” Zarr (a precursor to OME-Zarr) and OME-TIFF formats over HTTP (Supplementary Figure 1). Writing OME compliant Zarr and OME-TIFF can be accomplished using the Bio-Formats command-line suite. The `bioformats2raw` utility ([https://github.com/glencoesoftware/bioformats2raw](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 then supports the conversion of this “raw” format to (pyramidal) OME-TIFF via `raw2ometiff` ([https://github.com/glencoesoftware/raw2ometiff](https://github.com/glencoesoftware/raw2ometiff)). Viv can support viewing both the multiscale Bio-Formats “raw” Zarr as well as derived OME-Zarr and OME-TIFF.

Once created, Viv can access these data over HTTP. A local web-server is sufficient for viewing images locally and full web applications can be deployed by uploading the same data to a commodity web server or commercial cloud object storage (e.g., Amazon S3, Google Cloud Storage, Microsoft Azure Blob) where it can be accessed via Viv’s data loader modules.

Data Loader Component

Data loaders provide an interface for retrieving data tiles from arbitrary sources. Viv provides two built-in data loaders which fetch compressed byte-chunks from OME-TIFF and Zarr sources over HTTP, and subsequently decode these compressed byte arrays into uncompressed data tiles for rendering. Because OME-TIFF and Zarr provide a specification for representing both multidimensional and multi-scale dense arrays, the data loaders provide a shared abstraction
for serializing chunks of raw data from these sources remotely. Custom loaders can be implemented to support other data sources.

An important component of Viv’s data loaders is the support for Zarr\textsuperscript{28}. Zarr is an open-source format for the storage of chunked, compressed, N-dimensional 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 OME community has taken an interest in Zarr as a next-generation, chunked (“cloud”) file format, and the emerging OME-Zarr specification is under active community-driven development to provide a long-term format that can support future imaging data that might have higher dimensionality than current data.

Zarr implementations provide classes and functions for working with N-dimensional arrays where the underlying data are divided into chunks and each chunk is compressed. The underlying chunks and array metadata can be stored in any key-value store, most commonly a local file directory or commercial cloud object storage. This flexibility, however, affords the ability to configure custom storage options for application-specific needs. The incorporation of Zarr in the data loaders means that various backends can be utilized to support visualizations with Viv. For example, the Viv-based Vizarr ImJoy\textsuperscript{29} plugin uses a custom client store to securely transfer data from a Python Zarr in a Jupyter kernel via the ImJoy Remote Procedure Call (https://github.com/imjoy-team/imjoy-rpc).

Data loading from OME-TIFF files is handled through the geotiff.js\textsuperscript{32} library, which can retrieve tiles from both pyramidal and nonpyramidal TIFF files through HTTP byte-range requests. For images with a large number of channels, a large number of z-slices, or a large number of timepoints, Viv additionally supports an optional external offsets file that provides direct access to the position of specific images within the TIFF file to support faster loading. Sub-Image File Directories (SubIFDs) as specified by Bio-Formats pyramidal OME-TIFF are used for direct access to specific sub-resolutions.

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 UI thread 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. WebAssembly is an open specification for a portable binary-format which runs natively across all major web-browsers. We compiled the modern high-performance compressor Blosc\textsuperscript{30}, the default compressor in Zarr, to WebAssembly to support rapid decoding of Bio-Formats Zarr data tiles. Popular image compression methods have been ported to JavaScript previously, but WebAssembly extensions provide the ability to migrate modern implementations from compiled languages which do not exist in the web ecosystem.
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 renders them on a shared coordinate system. Interactive Deck.gl visualizations can be constructed by composing layers with others (points, polygons, text annotations, etc.), enabling efficient WebGL rendering of complex scenes.

The Viv layers control what is rendered when a user interacts with the WebGL canvas. Viv’s declarative layer API provides the ability to define specific channel selections from a multi-dimensional source as well as desired data transformations per channel. These parameters can be updated within a reactive paradigm, enabling efficient WebGL rendering across modern web frameworks and UIs. Each Viv layer uses a data loader to retrieve data tiles for the corresponding channel selections as a user zooms and pans in the coordinate space. Data tiles are fetched and decoded by the loaders and then efficiently loaded on the GPU, where shaders apply the user-defined data transformations.

In client-server web-viewers, since all rendering occurs on the server, nearly all user interactions require the user to wait for the server to re-render and re-send the data in order to view the desired changes. Since Viv handles all data fetching, decoding, and rendering, event handling is done in a more granular manner, which creates a more performant and responsive user experience. Once the data tiles have been retrieved for a particular region, changes to desired data transformations (contrast limits, opacity, colormap, visibility) simply re-render using the previously loaded data. This creates a very 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. The data tiles that Viv renders contain the data for each separate channel, and once these data has been bound to the GPU, data transfer functions are applied one of two ways to generate a rendering:

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 pure color to the maximum value per channel to be evaluated on the GPU by the fragment shader after min-max normalization. The contributions of individual channels are then additively blended into a single red/green/blue/alpha (RGBA) image. The respective min-max bounds, as well as the colors for each transfer function, are exposed via the Viv Layer API, enabling efficient and continuous rendering in response to user interactions in a reactive application. 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 can be effective 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. In this case, the channel intensities are all min-max normalized to 0-1 and then summed up. This final total value is then used as the input for a transfer function, as implemented in the Viv shaders. This is similar to the lookup tables supported by OMERO iViewer\textsuperscript{6}. We use glslify\textsuperscript{7} to inject transfer functions into the Viv shader code that maps the scaled intensity values to RGB colors. Alternatively, this can be implemented using a texture of the colormap that can then be sampled on the fragment shader using the normalized pixel intensity.

When multiple images are loaded into Viv, alpha compositing between layers is supported.

**Viv API**

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

1. **Viewers.** These are “drop-in” components that can be used within web development frameworks and provide interfaces for developers to supply controllers for the various rendering settings.

2. **Views.** Analogous to Deck.gl views, these provide different ways of looking at data, and form the foundation for supporting viewers with multiple views, like side-by-side or picture-in-picture. A view is associated with a particular view state defined by a zoom level and bounding box.

3. **Layers.** These 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.

4. **Loaders.** These classes implement a shared interface for accessing the metadata and channel data for primary images over HTTP.

**Supplementary Note 2**

**Use Cases**

**Image Registration Workflow**

Image registration is a critical task in biomedical image analysis. Accurate alignment of datasets captured from different modalities is imperative for integrating histological information with
molecular information to produce high-quality, comprehensive tissue profiles. The ability to quickly configure, test, and compare different registration methods is desirable during the alignment process but often requires large amounts of memory and/or CPUs to be efficient. Therefore institutional or cloud computational resources are attractive compared to personal computers when working with multiple high-resolution images, especially when the data are already located on the same high-performance servers. However, working with data remotely currently requires leaving the computational environment used for creating the alignment (e.g., Jupyter Notebook) to assess the quality of registration, slowing the ability to iterate different methods while optimizing the registration.

We created an ImJoy plugin, Vizarr (https://github.com/hms-dbmi/vizarr), using Viv to enhance an image registration workflow, enabling interactive viewing of multiple multiscale images from within a Jupyter Notebook. The registration example diagrammed in Supplementary Figure 2 details this workflow, displaying an overview of the registration of multiple image modalities using the wsireg library (https://github.com/NHPatterson/wsireg) and remote viewing of subsequent alignments all within the same Jupyter Notebook. The registration approach uses an autofluorescence (AF) microscopy intermediary captured on two serial sections prior to other imaging experiments (e.g., staining (Periodic acid-Schiff, PAS), cyclic multiplex immunofluorescence (MxF)) to enable mono-modality registration for the complex non-linear transformations desirable for serial section image registration. After registration, the resulting images are pyramidalized into local directory Zarr stores viewable through the Vizarr ImJoy plugin.

The registration requires a representative 2D XY plane of the data to be loaded into memory for computational image alignment using elastix as well as requiring the plane to be cast to a 32-bit floating-point data type and generation of a multi-resolution pyramid, further increasing the memory footprint when working with whole slide images. As WSIs typically contain at least 0.1 gigapixels but routinely contain 2 to 20 gigapixels of data which consume 4 bytes of memory per pixel at 32-bit floating point precision, using remote, high-performance resources is desirable to handle the large memory footprint and perform registration quickly with elastix's multi-threaded capability. However, once registration is performed, a visual check is necessary to confirm the quality of the registration at the microscopic level and doing so remotely in the same environment as the registration allows rapid iteration if registration results are poor. Further, the examination of the whole slide image registrations is best done at the highest resolutions to see if microscopic tissue areas are aligned. Vizarr, combined with wsireg running through a remote Jupyter Notebook, allows the computation of necessary transformations and generation of pyramidal images in Zarr stores that can then be remotely visualized with all registered layers in a single view.
Supplementary Figure 2. Performing registration with wsireg and visualizing registration results with the Vizarr ImJoy plugin through a remote Jupyter Notebook. Image registration: Images are represented by rectangles. AF Image Section 1 is the global target section and all images are resampled to the coordinate space of this image. Transformation models (linear: rigid, affine, non-linear: b spline) are used for different registration paths. Linear + non-linear is used for the mono-modal serial section registration between the AF images. Remote viewing: After registration, each image is pyramidalized through downsampling and chunked into a Zarr store on the remote resources. The Vizarr ImJoy plugin can display the remote data within the Jupyter Notebook.
**Supplementary Figure 3.** Low and high-resolution views of DAPI channel (Hoechst nuclear stain) from multiplex immunofluorescence through remote Vizarr plugin in Jupyter Notebook. (a) Low resolution “thumbnail” view of the RGB merge DAPI image from all three MxIF cycles. Some incorrect color blending is apparent at low resolution but is also due to inconsistent Hoechst intensity from the three cycles. (b) Zoomed in image to the base level of the pyramid of all three DAPI channels and the RGB merge where white indicates nucleus-nucleus overlap across all cycles.

Within the embedded viewer, different layers (images) can then be toggled on and off to view select subsets of layers for quality assurance. In Supplementary Figure 3, an overlay example from cyclic multiplex immunofluorescence is demonstrated where a Hoechst nuclear stain is present in each cycle in the DAPI channel. Through Vizarr, we can visualize individual nuclei from the three registered cycle images and see their overlay, assuring that individual cells are well aligned between the cycles. Technically, this result is achieved by chunked loading of images from different stores, showing the flexibility of Viv to handle data from multiple sources merged in a single layer. Viv, therefore, enables remote viewing of subcellular features with no compromise to spatial resolution or dynamic range of the data.
Supplementary Figure 4. Vizarr ImJoy plugin within a Jupyter Notebook. The Vizarr JavaScript implementation includes the imjoy-rpc, enabling users to programatically view Python-based Zarr images within a Jupyter Notebook.

Implementation

Vizarr is a purely client-side program built on Viv’s Zarr-based data loaders and WebGL-based rendering components. It implements the imjoy-rpc, which allows users to programatically view large datasets stored locally on disk or in-memory NumPy arrays within a Jupyter Notebook (Supplementary Figure 4). The protocol enables secure bidirectional data exchange between the Python and JavaScript environments. The underlying key-value mapping for a Python-based Zarr object is wrapped within a custom imjoy-rpc class, allowing the Zarr JavaScript client to access chunks from any valid Python store on demand. This means that other Zarr storage backends (e.g., N5, Redis, HDF5) can be viewed using our plugin within the Jupyter Notebook. Since the embedded viewer is entirely web-based, our plugin provides the flexibility to remotely view large datasets stored wherever a Jupyter session is running. As computational notebook environments have changed the ways that data scientists and machine learning practitioners perform and share exploratory analyses, this integration is a critical requirement for image visualization tools. Notebook formats such as Jupyter allow authors to weave code with prose and rich web-based interactive figures, forming a computational narrative. Support for Viv in Jupyter via the ImJoy plugin allows quantitative analyses of multiplexed images to consist of code, prose, and interactive views of image channels of interest side-by-side within the same notebook. Notebooks containing code woven together with
rich text and interactive visualizations have become essential tools for teaching, explaining, reproducing, publishing, and open science.

The source code for Vizarr is available at https://github.com/hms-dbmi/vizarr. The Jupyter Notebook containing the image registration workflow and other examples are available in the same repository.

Common Coordinate Framework Exploration User Interface

The Common Coordinate Framework (CCF) consists of ontologies, reference object libraries, and user interfaces that enable biomedical experts to (1) semantically annotate tissue samples and to precisely describe their locations in the human body (“registration”), (2) align multi-modal tissue data extracted from different individuals to a reference coordinate system (“mapping”), and (3) provide tools for searching and browsing HuBMAP data at multiple levels, from the whole body down to single cells (“exploration”).

To support the “exploration” of the CCF, HuBMAP implemented the Exploration User Interface (EUI) which represents tissue data across multiple scales, supporting efficient spatial and semantic search, browsing, and filtering of CCF data. The CCF EUI was developed for biomedical experts which are interested to advance our collective understanding of anatomical structures, cell types, and biomarkers (ASCT+B) and their type and range in human tissue (e.g., for different sex, ethnicity, BMI, or age).

A first prototype of the EUI became available in June 2019 (see Supplementary Figure 5, interactive user interface at https://hubmapconsortium.github.io/ccf-ui). It comprises search and filter options, an ontology browser, a tissue viewer, and a details-on-demand panel. Using the tissue viewer, users can turn on/off anatomical structures, functional tissue units (FTUs, e.g., glomeruli using the Cy3 Synaptopodin channel as a glomerular marker), but also different cell types. The EUI also features a whole body view and an organ view (not shown here).
**Supplementary Figure 5.** First prototype of the CCF EUI user interface, supporting the exploration of major anatomical structures such as renal capsule (red), outer and inner cortex (purple and blue), outer medulla (green) and inner medulla (orange). Analogously, functional tissue units such as glomeruli as well as different cell types can be highlighted.

The tissue viewer in the first EUI prototype did not support the examination of biomarkers. In addition, there was an interest to unify tissue viewer development efforts across HuBMAP. Combining expertise from different teams, we developed the general-purpose Viv tissue viewer that supports efficient viewing of high-resolution, multiplexed tissue images.

Supplementary Figure 6 shows Viv in the CCF EUI for the CCF v1.0.0. An autofluorescence microscopy tissue data sample generated by the Vanderbilt University Medical Center team is featured. Three channels (DAPI, eGFP, and dsRed) are included in the native autofluorescence microscopy OME-TIFF file. Channels can be turned on and off on right; a legend is automatically generated below the tissue viewer window.
Supplementary Figure 6. Integration of Viv into the HuBMAP CCF EUI displaying an autofluorescence microscopy OME-TIFF. Different layers can be turned on and off on the right. A legend is automatically generated below the tissue viewer for use in full-screen viewing mode. Multichannel autofluorescence microscopy images enable visualization of tissue functional units and are used to assess gross cell morphology.

Going forward, we plan to expand the CCF EUI and Viv with functionality to support the annotation and visualization of anatomical structures (including FTUs) and cell types. This way, manually or algorithmically generated segmentation data for gross and microanatomical structures but also single cells can be made visible—so users are able to explore the nested structure of cells within functional tissue units (FTU) within gross anatomical structures. Plus, it is highly desirable that users can identify and select one cell/FTU and query for all cells/FTUs of the same type to examine spatial density and distribution patterns.

Implementation

In the CCF EUI source code repository, TypeScript typings were added to Viv for ease of use from the TypeScript language, and a wrapper class was written in pure TypeScript (https://www.typescriptlang.org). Then from the Angular (https://angular.io) side, the wrapper class was used to create a generic Angular component and library for using Viv features from Angular. In both cases, the wrappers were designed to emulate how Viv works, but in the semantics of the target paradigm (i.e., TypeScript and Angular).

The Angular component was then integrated into the CCF EUI. It appears in the user interface in a full-screen view when a user selects a tissue sample from a set of spatially registered tissues that have associated TIFF images retrieved from the HuBMAP Search API. Images retrieved from the HuBMAP infrastructure can then be interacted with using Viv, including on-the-fly dynamic recoloring of image channels to make specific cell types, FTUs, or gross
anatomical structures easier to recognize. User interface components were implemented in Angular for listing and customizing the channel color-coding and brightness settings. Another Angular component was implemented to provide a legend of the current channel settings applied to Viv.

The MIT-licensed open source code for the CCF EUI is available at https://github.com/hubmapconsortium/ccf-ui.

Avivator Image Viewer

We have created Avivator (http://avivator.gehlenborglab.org), a JavaScript and React-based image viewer that displays well-formed, pyramidal OME-TIFF, and Bio-Formats-compatible Zarr files directly from a user-provided remote URL.

Supplementary Figure 7. Avivator showing an OME-TIFF file of a multiplexed immunofluorescence microscopy image of a human kidney using additive blending to render four image channels into a single RBG image in the client. The picture-in-picture view in the lower left corner provides context. A magic lens applying a local transformation of the image data to emphasize the “Cy3 - Synaptotodin” channel.

Avivator implements continuous zooming and panning and showcases the advantages of client-side image rendering. A slider interface allows users to modify the channel intensities and the image updates as soon as the sliders are being dragged. As the updates are instantaneous due to client-side image rendering, this approach is very efficient for finding adequate intensity
settings. Avigator implements both additive blending and additive color mapping. Furthermore, Avigator provides magic lenses\textsuperscript{37} that locally re-render the image with different transformation settings, e.g., to highlight a particular image channel (see Supplementary Figure 7). As all operations are taking place in the client, there is no noticeable lag when moving the lens over the image. Finally, Avigator also provides support for scale bars, as well as a picture-in-picture (see Supplementary Figure 7) and linked side-by-side views (see Supplementary Figure 8) that enable the overview + detail visualization paradigm. Support for image annotations are beyond the scope of this Viv use case but can be added to Avigator by integrating vector layers provided by Deck.gl.

\textbf{Supplementary Figure 8.} User interface of Avigator showing an OME-TIFF file of a multiplexed immunofluorescence microscopy image of a human kidney as two linked views with a detail view on the left and an overview on the right. Three active channels (FITC, Cy3, Cy5) are mapped to the “magma” color map using the additive color mapping approach built into Viv.

Since the images rendered by Avigator are Bio-Formats-compatible OME-TIFF files or Zarr stores, users can directly download the primary data for further analysis. This compatibility of image formats shared by Viv and the rest of the image analysis ecosystem illustrates a powerful mechanism for users to share data internally, with collaborators, or the broader scientific community. To enable this, users can upload primary data to persistent cloud object storage or local web servers and provide a URL in a client application to view these resources, eliminating the need to pre-render or configure sophisticated server-client architectures, while providing the option to download and analyze with other tools. Cloud object storage is becoming increasingly
popular for storing archival data for growing image resources, such as the European Bioinformatics Institute (EBI)-based Image Data Resource (IDR)\textsuperscript{25}, and having client software that can visualize the data in the resources directly is essential to making these resources accessible at scale.

Implementation

Avivator is an open source React application that combines multiple Viv layers and views, which are extensions of Deck.gl. Its source code is available as part of Viv at https://github.com/hms-dbmi/viv and at https://www.npmjs.com/package/@hms-dbmi/viv. A collection of sample images are provided on commercial cloud storage and accessible directly via URL in the web application.

Supplementary References


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).


35. Marstal, K., Berendsen, F., Staring, M. & Klein, S. SimpleElastix: A User-Friendly,
Multi-lingual Library for Medical Image Registration. in *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)* 574–582 (2016).
