From Big Data to Big Displays
High-Performance Visualization at Blue Brain

Stefan Eilemann, et.al.*

Blue Brain Project, Ecole Polytechnique Federale de Lausanne

Abstract. The Blue Brain project has been investing in high-performance visualization (HPV) to complement its HPC strategy since its inception in 2007. In 2011, this strategy has been accelerated to develop innovative visualization solutions through through increased funding and strategic partnerships with other research institutions.
We present the key elements of this HPV ecosystem, which integrates C++ visualization applications with novel collaborative display systems. We motivate how our strategy of transforming high-performance visualization engines into services enables a variety of use cases, not only for the integration with high-fidelity displays, but also to build service oriented architectures, to link into web applications and to provide remote services to python applications.

1 Motivation

The Blue Brain Project (BBP) uses simulation-based research to analyze and reverse engineer cortical neuron circuits. The simulated models go beyond using detailed models of individual neurons or large-scale network models of simplified neurons, they model in the order of hundreds of thousands of detailed neurons in a fully connected circuit. The project generates a multitude of data for the model building and during the simulation of these models. This data ranges from terabyte-sized image stacks for data extraction to detailed in-silico circuit models of large geometric complexity and terabyte-size simulation reports.

Visualization supports the BBP along all parts of this project to understand and debug model data, building and simulation algorithms as well as validating and discovering new insight from the in-silico experiments. Our strategy to support this mission is based on components linked through network protocols: High-fidelity display systems to see more detail in complex data, a set of standard rendering engines (rasterization, out-of-core volume rendering, interactive raytracing), and decoupled, light-weight applications using remote these components. In the following we will present these components along with a few use cases.

2 High-Fidelity Displays

High-fidelity display systems are the integration point of the Blue Brain visualization capabilities. They are the evolution of existing visualization systems, enabling

* *firstname.lastname*@epfl.ch

higher resolution, immersion and team collaboration. Compared to current single-user or single-presenter systems, collaborative display systems allow real team work through a combination of size, resolution and user friendly implementation. Compared to immersive visualization systems like the CAVE, they provide a more approachable environment for high-fidelity visualization. For all use cases, the increased display size and resolution allows better data exploration for 2D and 3D content, facilitating large data exploration.

2.1 Tiled Multitouch Display Walls

The core of the Blue Brain visualization infrastructure are multiple high-resolution tiled display walls driven by our Tide software [1]. All walls are equipped with a multitouch user interface and can be remote controlled from any web browser. The walls are build using thin-bezel, 55 inch, Full HD LCD panels with a hardened glass sheet. We use 4×3 and 3×3 configurations for a total of 24 and 18 Megapixel resolution, respectively. The display size of over five meter diagonally (four meter for 3×3) allows team-size collaboration (up to ten people) or project-wide presentations (up to a hundred people). Figure 1 shows one wall during a project-wide presentation with multiple interactive applications running remotely on the wall.

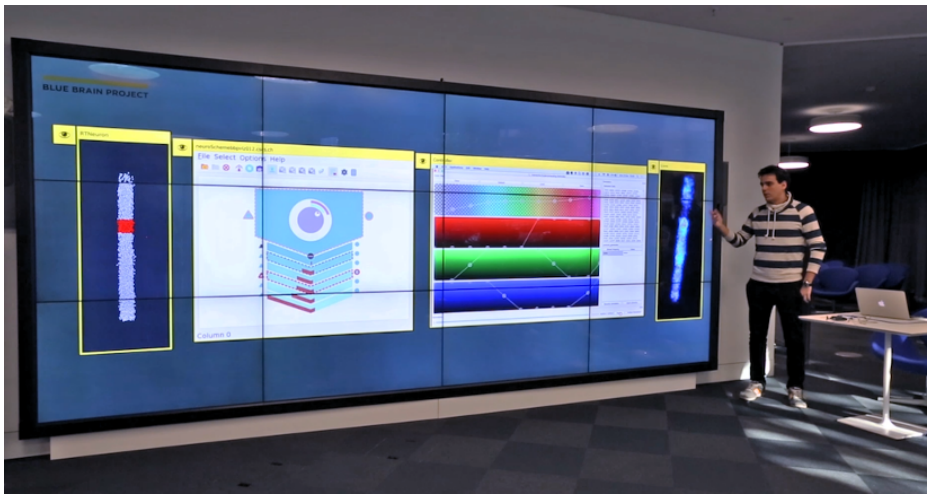


Fig. 1: Blue Brain 4×3 tiled display wall

While there is substantial research on tiled display wall software [3, 4, 6, 10, 11, 13], we found that most solutions where not ready production use in a 24×7 unattained environment. For this reason, we started with the TACC DisplayCluster open source software [11], which we incrementally refactored and improved to the current TIDE implementation. On the other hand, the hardware has been commoditized to make these type of installations affordable to medium-sized institutions which allowed us to build

the software integration for a reasonable startup cost. We have focused on the multitouch user interface, which implements a low entry barrier for new users, a unique capability of our solution.

2.2 OpenDeck

OpenDeck is our next-generation visualization system, aiming to integrate the success of tiled display walls with a seamless transition to fully immersive environments. We are currently in the process of installing a system which consists of a semi-cylindrical back-projection screen with 41 Megapixel usable resolution on a 36 m^2 surface (Figure 2). Like the display walls, it is equipped with multitouch capabilities which makes it usable as a monoscopic collaboration system from the first day of installation. Unlike tiled display walls, it is active stereo capable and equipped with a 3D tracking system for immersive rendering. For increased immersion, a lower resolution front projection system fills in the floor area. OpenDeck will run TIDA and our immersive applications based on Equalizer [5] once the system is installed.

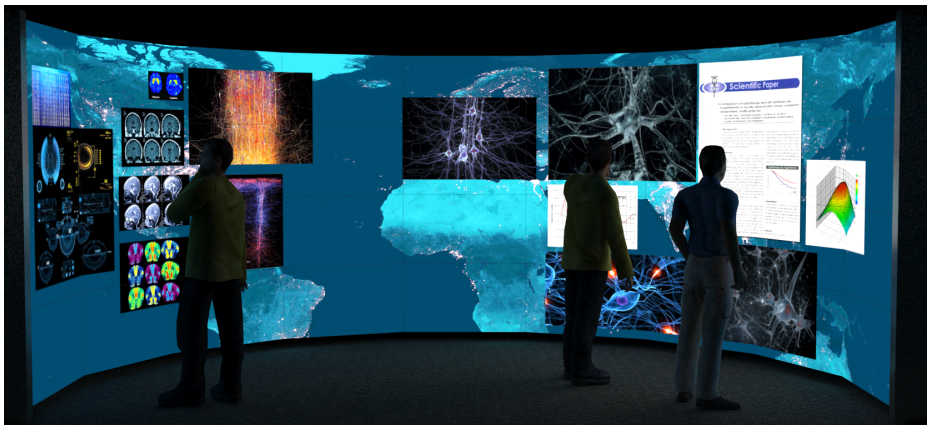


Fig. 2: OpenDeck concept rendering

OpenDeck will provide a unique environment for the research and development of new visualization techniques. It will open a set of questions along immersive touch user interfaces, transitions and mixing of monoscopic to immersive usage, the combination of tracked and touch devices, multi-user immersion, latency for remote immersive rendering as well as multi-site collaboration.

2.3 Tide

Tide (Tiled Interactive Display Environment) is the software driving the Blue Brain tiled display walls and OpenDeck. It provides multi-window, multi-user touch interaction on large surfaces — think of a giant collaborative wall-mounted tablet. Tide is a distributed

application that can run on multiple machines to power display walls or projection systems of any size. Its user interface is designed to offer an intuitive experience on touch walls. It works just as well on non touch-capable installations by using its web interface from any web browser. Figure 1 shows Tide on a 4×3 display wall and Figure 3 shows the Tide web interface in a browser.

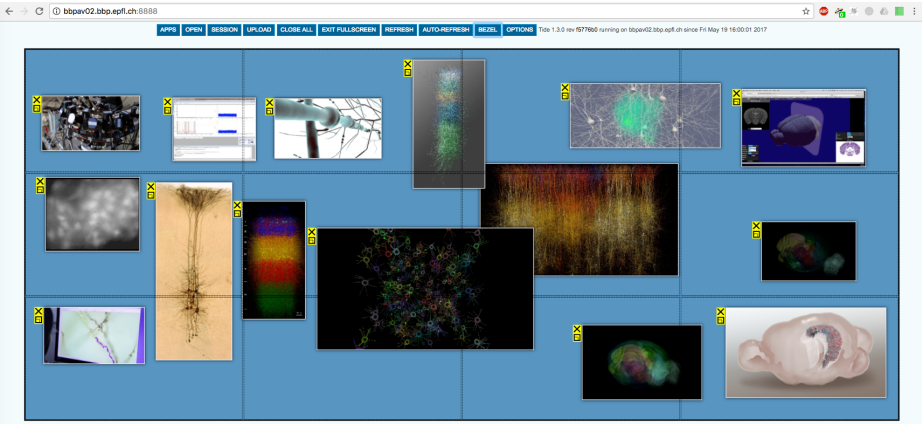


Fig. 3: Tide web interface

Compared to other solutions [3, 11, 13], the development of Tide was driven by the need of a stable and easy to use implementation. Tide supports three types of content: files (high-resolution images, movies, pdfs), built-in applications (web browser, whiteboard) and remote applications using the Deflect library (DesktopStreamer, Equalizer-based applications, Brayns). The multitouch user interface can handle multiple users manipulating different windows and their content simultaneously.

2.4 Deflect

Deflect is the client library for Tide. It provides an API for pixel streaming to Tide and for receiving events from Tide. The pixel streaming allows synchronized parallel streaming from parallel rendering application as well as monoscopic and stereoscopic streams. Various events allow the application to react to multi-touch input from the wall.

Deflect is integrated into the Equalizer parallel rendering framework [5], enabling transparent usage of Equalizer applications on Tide walls. Furthermore, the DesktopStreamer application mirrors the desktop of other machines onto a wall window and allows interaction with the remote desktop. Other rendering applications, such as our interactive raytracing engine Brayns [2] are easily integrated with Deflect and Tide.

3 Rendering Applications

The rendering applications form the backbone of our ecosystem. They cover a wide range of established rendering algorithms to serve a broad set of use cases for visual debugging, scientific illustrations and communication.

3.1 Brayns: Interactive Raytracing

Advances in computer hardware and software have brought raytracing to the point where it replaces classical rasterization for virtually all use cases in scientific visualization. On one hand, OpenGL-like local illumination for typical data sets used with rasterization (up to hundred million triangles) can be done at similar framerates to OpenGL [15]. For small datasets, OpenGL performs better, but for larger data sets raytracing outperforms OpenGL. This is due to a better scalability with respect to the data set size ($O(\log N)$ vs $O(n)$). Furthermore, CPU-based raytracing allows the rendering of larger data sets without any level-of-detail algorithms due to the larger memory size. Last, but not least, advanced rendering algorithms such as shadowing, reflections and global illuminations are significantly easier to implement in a raytracer. The only area where rasterization provides a benefit is for rendering at very high frame rates, needed for example for immersive visualization.

It is for these reasons that interactive raytracing is the future technology for interactive and offline rendering at Blue Brain. We have developed a first open source implementation of a visualization engine with different backends: OSPRay [15] for CPU-based raytracing and OptiX [14] for GPU-based raytracing. This application called Brayns can load and visualize our data sets in a variety of modes and integrates with our messaging solution to be accessible from Python. We are currently integrating the key algorithms from Livre for out-of-core volume rendering in Brayns, which will also facilitate mixing polygonal with volume data in a single scene.

FIXME: teaser gallery

3.2 RTNeuron: OpenGL Parallel Rendering

3.3 Livre: Out-of-Core Volume Rendering

Livre is an interactive volume rendering engine available under a permissive open source license. Our main contributions are: a state-of-the-art implementation of an octree-based level-of-detail (LOD) selection, a task parallel rendering pipeline, a multi-GPU parallel rendering engine, and an easily extensible renderer through the use of plugin data sources. Our system brings together state-of-the art algorithms to create a volume rendering engine capable of handling extremely high-resolution volumes using a high degree of parallelism, both on a single system and in a distributed cluster. We employ a GPU-based ray casting algorithm to compute the radiance absorption of the given volumetric data. The computation is executed per pixel on the pixel shader hardware of the GPU.

In our out-of-core data access layer, multi-resolution data is represented as an octree data structure. This representation accelerates the selection of the proper level-of-detail

and to track the status of the LODs (in CPU memory, in GPU memory, not loaded). While rendering, view-based LOD selection is performed using the screen-space-error (SSE) [9] technique. Figure 4 shows the rendering of 300 GB volume with an illustration of the selected LOD levels.

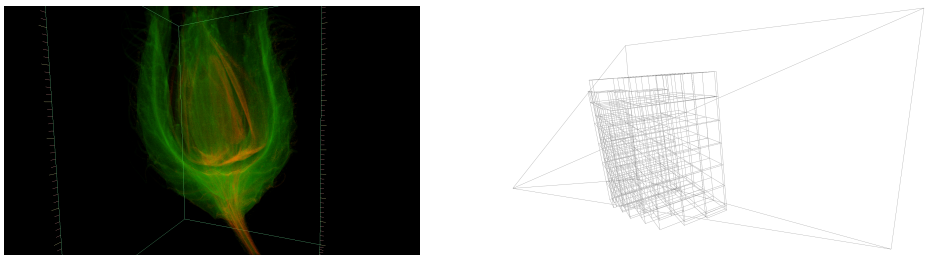


Fig. 4: Livre rendering a MicroCT dataset (left) and the selected LODs (right)

The creation of volume bricks, their upload to the GPU and the rendering are executed in separate tasks. These tasks run asynchronously, that is, rendering is decoupled from data loading. Livre uses a plugin mechanism to access the volume data, where data sources are implemented as shared libraries and are loaded on application startup based on the URI of the input data. Data sources only have to provide the requested volume bricks, that is, there is no defined file format or even requirement to read the input data from a file system. This flexibility of the plugin approach lead to novel volume rendering use cases, where volume representations are created on the fly from different input data sets, for example from simulation data.

4 Messaging and Service Architecture

All Blue Brain applications integrate messaging libraries which allow them to be used as services in a variety of use cases. For example, the Tide web server providing the user interface shown in Figure 3, is based on this messaging solution. Other use cases are remote python APIs, JavaScript user interfaces and service architectures combining multiple visualization applications with data providers such as HPC simulations (Figure 5).

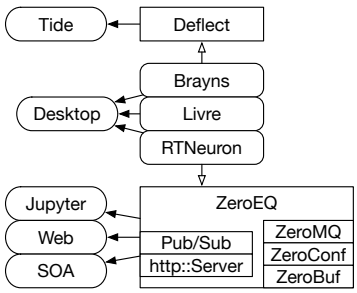


Fig. 5: Messaging enables new use cases

The base communication layer ZeroEQ utilizes ZeroMQ as the transport layer, the ZeroConf protocol for discovery, and our novel ZeroBuf serialization library for high-performance messaging. A fully integrated HTTP server provides a bridge to JavaScript, Python

and similar environments by implementing REST APIs with JSON payload. Figure 6 shows a class diagram of our messaging solution.

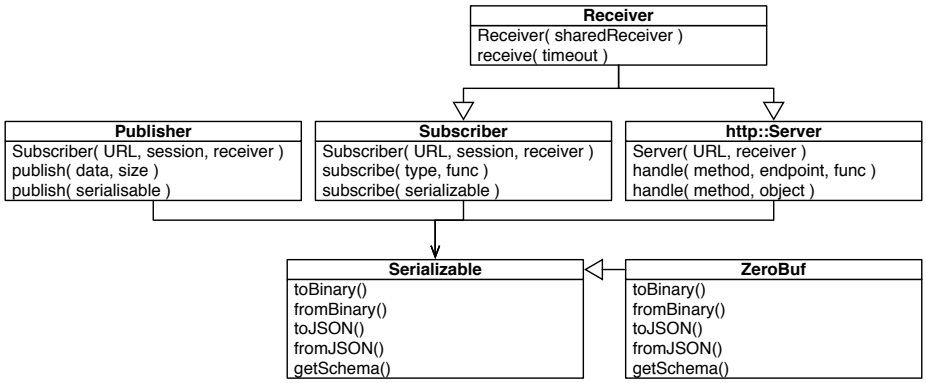


Fig. 6: UML Diagram of the main messaging classes

4.1 ZeroEQ

ZeroEQ is our C++ messaging library, wrapping up existing technologies into an API which is convenient to use and easy to integrate into C++ code. It provides two messaging services: publish-subscribe and HTTP. For binary and JSON encodings it relies on a simple Serializable interface, for which ZeroBuf provides a sample implementation. To facilitate the simple use case of linking a few applications, ZeroEQ uses the zeroconf protocol to discover and connect to related applications. For more complex scenarios, explicit connections are supported.

Publish-Subscribe The publish-subscribe service is implemented in a Publisher and Subscriber class. It provides event-based messaging, based on a 128-bit message type with arbitrary payload. The message type is used for message subscription, filtering and routing. The payload is expected to be uniquely identified by the message type, that is, all applications agree for the decoding and semantics of any given message type. ZeroBuf provides a sample implementation for this. The underlying transport uses ZeroMQ pub-sub sockets.

The pub-sub service is stateless, that is, applications have no expectation of when messages are received or who receives published messages. This communication pattern naturally leads to robust services, since there is no possiblity for deadlocks or undefined behaviour. The pub-sub API is provided in two flavors: a simple *pointer* & *size* memory buffer, and a higher level object-based abstraction. The object-based API is syntactic sugar for the low-level API, and allows automatic publish and update of objects with a few lines of code. It uses the toBinary() and fromBinary() methods of the Serializable interface to call the low-level API.

```
zerobuf::render::Camera camera;
zeroeq::Publisher publisher;
zeroeq::Subscriber subscriber;

subscriber.subscribe( camera );

while( rendering )
{
    subscriber.receive( 0 /*ms, poll*/ );
    updateCamera( camera );
    publisher.publish( camera );
    renderFrame( camera );
}
```

Listing 1.1: Publish-Subscribe Example

```
zeroeq::URI uri( "tcp://localhost" );
zeroeq::Publisher publisher( uri, zeroeq::NULL_SESSION ); // deactivate zeroconf
zeroeq::Subscriber subscriber( publisher.getURI( )); // use concrete port
```

Listing 1.2: Explicit Addressing

The example in Listing 1.1 shows the integration of camera synchronization in a visualization application. This example relies on the builtin zeroconf protocol to connect application instances. Subscribers only subscribe to events from publishers within the same session. The default session name is the user name, and can be customized using an environment variable or non-default constructor. Similarly, the subscriber can subscribe by session or address. Listing 1.2 illustrates an explicitly addressed subscription. Notice that the subscriber uses the publisher URI, which will contain the concrete port chosen for the publisher.

Subscribers are derived from a (Receiver) base class, which is shared with the http server. All receivers can share their receive() operation at construction time, that is, the blocking receive operation applies to all receivers in the shared group. Listing 1.3 shows an example of selectively receiving different updates on different input sockets.

HTTP Server The http server is built using cppnetlib [8] for the transport and http protocol handling. It supports all standard http verbs (GET, POST, PUT, PATCH, DELETE). It is a zeroeq::Receiver, that is, it can share its receive() update operation with other

```
zeroeq::Subscriber local( zeroeq::URI( "localhost:29387" ));
zeroeq::Subscriber global( local );

local.subscribe( colorMap );
global.subscribe( camera );

while( true )
    local.receive(); // updates colorMap and camera
```

Listing 1.3: Subscriber Sharing

subscribers and http servers. Unlike a subscriber, the http server follows the HTTP request-reply semantics, that is, a request received by a server has to be followed directly by its reply. To allow asynchronous request processing, the return value from the request handler is a std::future which is retrieved from an internal thread, thus allowing the application to continue operations.

The data served by the http server is introspectable, it allows querying the available endpoints (objects) and the JSON schema [12] for each endpoint. Listing 1.4 shows an excerpt of the Tide registry, and Listing 1.5 an excerpt of the schema for one of the exposed objects. This REST API is used by the Tide web interface from Javascript and to generate remote python APIs.

```
> GET /registry HTTP/1.0
{
[...]
   "tide/open" : [ "PUT" ],
   "tide/options" : [ "GET", "PUT" ],
   "tide/resize-window" : [ "PUT" ],
[...]
}
```

Listing 1.4: HTTP Server Registry

```
> GET /tide/options/schema HTTP/1.0
{
[...]
    "properties": {
        "alphaBlending": {
            "type": "boolean"
        },
[...]
}
```

Listing 1.5: Object JSON Schema

4.2 ZeroBuf

ZeroBuf is a sample implementation of serialization for ZeroEQ. Based on a grammar closely related to Flatbuffers schemas [7], it generates C++ classes with random set/get access. All data is stored internally in one continuous memory buffer, which can be used for the ZeroEQ binary serialization without any copy. The conversion to and from a JSON representation involves a copy using a std::string. ZeroBuf can store:

- (u)int[8,16,32,64,128]_t, float, double and string members
- fixed size arrays and dynamic vectors of static-sized elements (intrinsic members or composite types)
- static and dynamic sub-classes (composite types of the above)

Figure 7 shows two simple ZeroBuf schemas together with example usage of the generated code in C++ and their memory layout. The static example shows nested ZeroBuf classes for the camera used in Listing 1.1, and the dynamic example shows how raw data access is used to prepare a JPEG image for publishing.

4.3 Remote Python API

The remote python API provides easy to use access to remote applications using the http server. It integrates two features: generic code generation for the REST API exposed by the application, and automatic resource allocation and application launch.

The generic code generation is implemented in a pure python module, which has no dependency to the interfaced C++ application. It queries the http server and generates a python API for all exposed objects. This API can then be conveniently used in python to remote control the application.

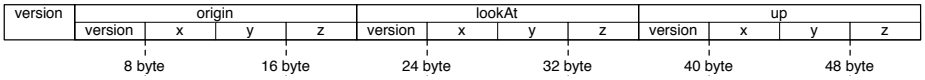
FBS Schema

```
namespace zerobuf.render;

table Vector3f
{
  x: float;
  y: float;
  z: float;
}

table Camera
{
  origin: Vector3f = 0, 0, 1;
  lookAt: Vector3f;
  up: Vector3f = 0, 1, 0;
}
```

C++ Code

```
namespace render = zerobuf::render;

render::Camera camera;
camera.getOrigin() = render::Vector3f(0, 0, 1);
```

Memory Layout



FBS Schema

```
namespace zerobuf.render;

table ImageJPEG
{
  data:[ubyte];
}
```

C++ Code

```
namespace render = zerobuf::render;

render::ImageJPEG imageJPEG;
glReadPixels( … );
tjCompress2( …, ptr, size );
imageJPEG.setData( ptr, size );
```

Memory Layout

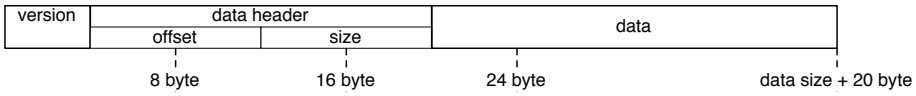


Fig. 7: ZeroBuf Examples for static (top) and dynamic (bottom) sized objects

Access to the application is established either through an explicit connection of a pre-launched application, or via a resource allocator. The allocator hides the details of allocating a resource, e.g. using a scheduling system like slurm, launching and connecting to the launched application from the python programmer.

Figure 8 shows an example session of using this Python API from a Jupyter notebook, allocating and launching a Brayns instance, setting relevant rendering parameters and retrieving an image. Note that the whole notebook runs in a light-weight VM with no GPU and interacts with a Brayns instance launched on a bare-metal visualization cluster node.

5 Discussion and Conclusion

We presented a modular visualization architecture for large data visualization over a wide range of use cases, glued together by a modern and easy to use messaging infras-

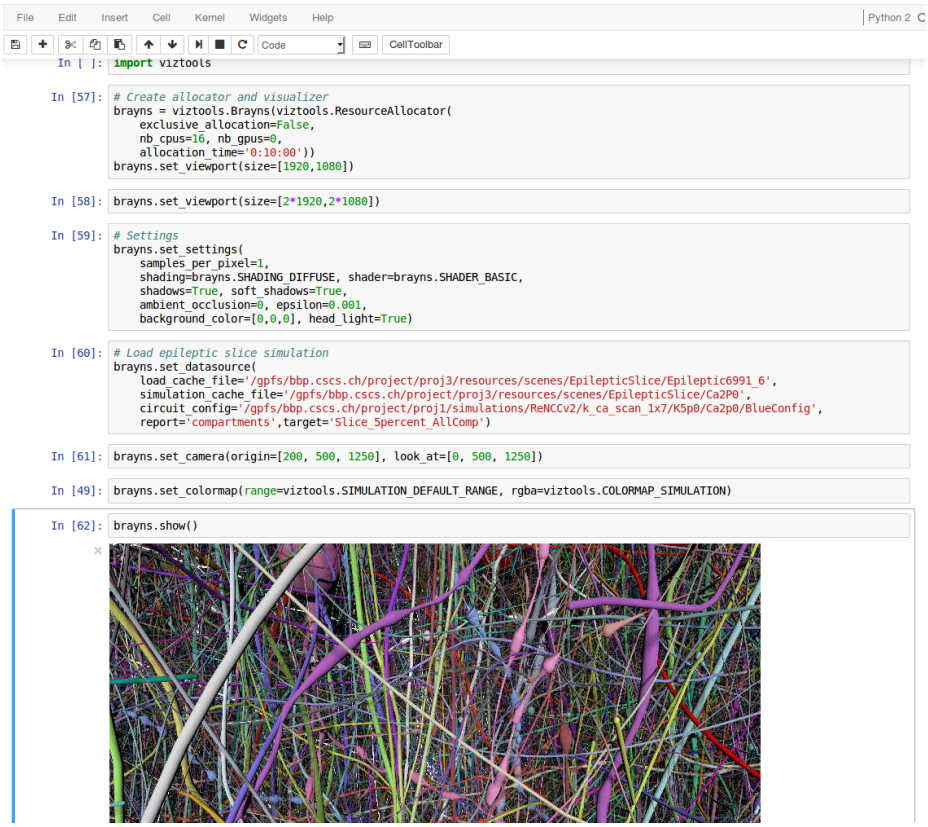


Fig. 8: Example Jupyter notebook session using Brayns

tructure. This ecosystem allows us to flexibly support novel use cases, while pushing novel visualization capabilities, providing Blue Brain with a competitive visualization infrastructure. Messaging and remote APIs not only surprised us in their versatility and ease of integration with other ecosystems, but also have a significant potential for future exploration in classical visualization software. Interactive raytracing is the future rendering algorithm for us, and Brayns is becoming the integration point for our domain-specific visual applications and algorithms. Tiled display walls are affordable for a large set of institutions, and coupled with our open source TIDE software create new ways of truely collaborative work. TIDE, together with cheaper visualization hardware, will evolve in the future towards a seamless integration of immersive visualization.

Acknowledgments

This publication was supported by the Blue Brain Project (BBP), the Swiss National Science Foundation under Grant 200020-129525, the King Abdullah University of Science and Technology (KAUST) through the KAUST-EPFL alliance for Neuro-Inspired

High Performance Computing, the Spanish Ministry of Science and Innovation under grant (TIN2010-21289-C02-01/02), the Cajal Blue Brain Project, Hasler Stiftung Projekt Nr. 12097, and from the European Unions Horizon 2020 research and innovation programme under grant agreement No 720270 (HBP SGA1).

References

1. Blue Brain Project. Tide: Tiled Interactive Display Environment. https://github.com/BlueBrain/Tide, 2016.
2. Blue Brain Project. Brayns: Interactive raytracing of neuroscience data. https://github.com/BlueBrain/Brayns, 2017.
3. T. A. DeFanti, J. Leigh, L. Renambot, B. Jeong, A. Verlo, L. Long, M. Brown, D. J. Sandin, V. Vishwanath, Q. Liu, M. J. Katz, P. Papadopoulos, J. P. Keefe, G. R. Hidley, G. L. Dawe, I. Kaufman, B. Glogowski, K.-U. Doerr, R. Singh, J. Girado, J. P. Schulze, F. Kuester, and L. Smarr. The optiportal, a scalable visualization, storage, and computing interface device for the optiputer. *Future Gener. Comput. Syst.*, 25(2):114–123, Feb. 2009.
4. K.-U. Doerr and F. Kuester. CGLX: A scalable, high-performance visualization framework for networked display environments. *IEEE Transactions on Visualization and Computer Graphics*, 17(2):320–332, March 2011.
5. S. Eilemann, M. Makhinya, and R. Pajarola. Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):436–452, May/June 2009.
6. A. Febretti, A. Nishimoto, V. Mateevitsi, L. Renambot, A. Johnson, and J. Leigh. Omegalib: A multi-view application framework for hybrid reality display environments. In *2014 IEEE Virtual Reality (VR)*, pages 9–14, March 2014.
7. Google, Inc. Cross Platform Serialization Library. http://google.github.io/flatbuffers/, 2017.
8. D. M. B. G. M. Google, Inc. The C++ Network Library Project. http://cpp-netlib.org/, 2017.
9. S. Guthe and W. Strasser. Advanced techniques for high-quality multi-resolution volume rendering. *Computers & Graphics*, 28(1):51–58, 2004.
10. A. Johnson, J. Leigh, P. Morin, and P. Van Keken. GeoWall: Stereoscopic visualization for geoscience research and education. *IEEE Computer Graphics and Applications*, 26(6):10–14, November-December 2006.
11. G. P. Johnson, G. D. Abram, B. Westing, P. Navr'til, and K. Gaither. DisplayCluster: An Interactive Visualization Environment for Tiled Displays. In *2012 IEEE International Conference on Cluster Computing*, pages 239–247, Sept 2012.
12. JSON Schema. JSON Schema. http://json-schema.org/, 2017.
13. T. Marrinan, J. Aurisano, A. Nishimoto, K. Bharadwaj, V. Mateevitsi, L. Renambot, L. Long, A. Johnson, and J. Leigh. SAGE2: A new approach for data intensive collaboration using Scalable Resolution Shared Displays. In *Collaborative Computing: Networking, Applications and Worksharing*, pages 177–186, 2014.
14. S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics*, August 2010.
15. I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Gnther, and P. Navratil. OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):931–940, Jan 2017.