High-performance interactive scientific visualization with DatoViz via the Vulkan low-level GPU API

Cyrille Rossant[1], International Brain Laboratory[2], and Nicolas P. Rougier[3]

[1]International Brain Laboratory
[2]Affiliation not available
[3]INRIA Bordeaux Sud-Ouest

May 17, 2021

Introduction

Most scientific disciplines are facing exponentially increasing amounts of data, which require scalable interactive scientific visualization technology. The development of massively parallel graphics processing units (GPU), fostered by the video game industry and artificial intelligence, represents a remarkable opportunity in this respect.

Real-time computer graphics technology has been used in scientific visualization for decades, mostly via OpenGL, a popular open-source graphics library created in 1992. It has been used by many video games, graphics applications, and scientific visualization software. During its first decade, OpenGL provided a fixed function pipeline that was simple to use, but not particularly powerful because of the lack of control of the rendering pipeline. In 2004, OpenGL 2.X introduced a programmable pipeline that gave the user a way to customize the various stages of rendering. Since then, there have been various open-source libraries providing OpenGL-based scientific visualization, mostly focused on 3D rendering. Mayavi is a popular example in Python (Ramachandran & Varoquaux, 2011).

In 2013, Luke Campagnola, Almar Klein, Cyrille Rossant, and Nicolas P. Rougier wrote a new visualization library: VisPy (Campagnola et al., 2015). This library took full advantage of the GL ES 2.x API to achieve both fast and scalable rendering of the most common plotting objects, in both 2D and 3D (lines, scatters, images, colormaps, volumes, meshes, etc.). Within a few years, VisPy reached a large scientific audience and became the main real-time 2D/3D scientific rendering library in Python.

Although usage of OpenGL is still widespread in the graphics and scientific communities for legacy reasons, the industry is steadily moving to newer low-level graphics APIs such as Vulkan (Khronos), WebGPU (W3C), Metal (Apple), and DirectX 12 (Microsoft). In this context, VisPy is now facing the same problem as Mayavi faced a few years back: it must decide on its future.

The Khronos group introduced the Vulkan API in 2016 (`https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification`). This has been a complete redesign from the ground up to give much more control (compared to OpenGL) and to support all features of the latest GPU hardware. However, Vulkan has a huge barrier to entry: drawing a simple triangle using the Vulkan API directly involves about a thousand lines of code. In particular, all the logic related to the presentation of images to the screen, using a swapchain for double- or triple-buffering, all the synchronization of the different GPU tasks and CPU-GPU interactions in the main rendering loop must be done manually. To leverage the power of Vulkan for applications such as scientific visualization, there is thus a crucial need for intermediate-level libraries that drastically simplify the access to Vulkan.

One potential solution isbe to use existing rendering engines such as Ogre (https://www.ogre3d.org/), Unity (https://unity.com/), or Unreal (https://www.unrealengine.com/). However, although scientific visualization and video games do share many similarities, they are quite different in their ends. Games are generally highly dynamic and interactive, whereas scientific visualization is much more static and less interactive (to some extent) and can be indifferently 1D, 2D, or 3D. Furthermore, scientific visualization involves a number of concepts that are generally not present in game engines, such as colormaps, labeled axes, non-cartesian projections, image interpolations, etc. Scientific visualization must also be faithful to the data and this requires a highly precise rendering to achieve high representational accuracy. Besides, it is not unusual to render millions or even billions of points in a scientific visualization — while a modern GPU has no problem rendering such a large collection, the corresponding API must be aware of such extreme cases to ensure proper rendering. All these limitations disqualify typical video game rendering engines as a general purpose API for scientific visualization.

This exposes a crucial need for a rendering engine that is to scientific visualization what game engines are to video games: an intermediate-level library that allows developers of custom scientific visualizations, or developers of high-level plotting libraries, to leverage Vulkan without delving into an incredibly complicated low-level API. We report here progress that we have made in the past couple of years towards a cross-platform, cross-language scientific visualization engine that leverages Vulkan for scalable, low-overhead, high-performance scientific visualization.

Vulkan for scientific visualization via the Datoviz library

Although the idea of using Vulkan for scientific computing dates from 2015 (https://cyrille.rossant.net/compiler-data-visualization/), we started seriously investigating this technology for scientific visualization purposes in 2019. It took more than a year to understand the technology and design the low-level foundations for a general-purpose scientific visualization library. In February 2021, Cyrille Rossant and the International Brain Laboratory unveiled an early version of Datoviz (https://datoviz.org/), a C/C++ Vulkan-based scientific visualization toolkit that aims at providing a unified platform for 2D, 3D, and graphical user interfaces (see Fig. 1). Datoviz provides a C API and early Python bindings using Cython (Behnel et al., 2011). Datoviz could be wrapped by other languages as well, such as Julia, R, MATLAB, or Rust. As of today, we know of no other public initiative that has explored using Vulkan for scientific computing. Worth noting is ANARI (https://www.khronos.org/anari), a Khronos working group that will be defining a future API for GPU-based scientific visualization.

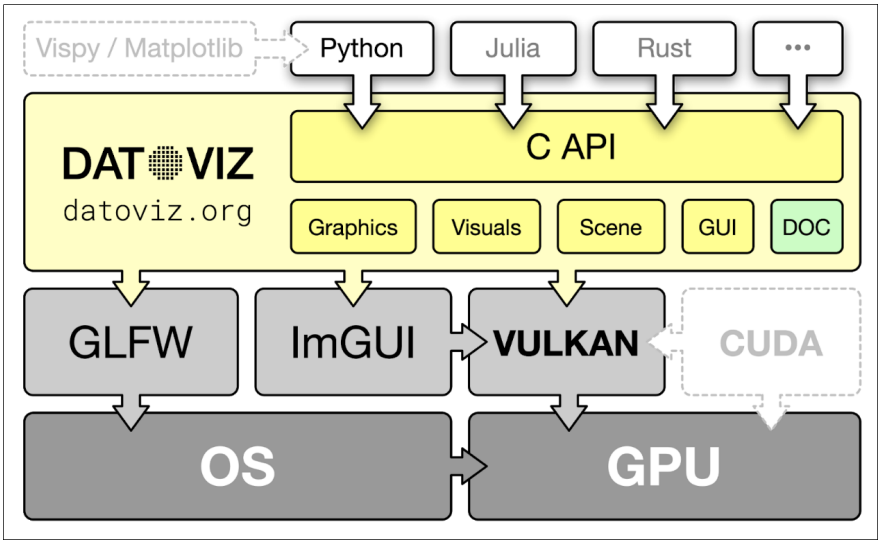

Figure 1: **Datoviz architecture.** Datoviz provides a C API that may be wrapped by Python and other high-level languages. Datoviz comprises several modular layers which rely on a few third-party libraries and APIs, the most important ones being GLFW, a cross-platform window creation library (https://www.glfw.org/), Dear ImGui, a GPU-based GUI library, and Vulkan.

The point of using technology as complex and low-level as Vulkan is *performance* , especially with huge datasets. Vulkan provides low-overhead, fine-grained control of GPU and GPU-CPU interactions. These features are essential when writing generic engines for video games or scientific visualization, although the latter has much less demanding requirements than the former. A key goal was therefore to identify the fundamental aspects of Vulkan and to understand how they could be used to achieve optimal performance with typical scientific visualization applications.

How to use the GPU for scientific visualization? Essentially, GPUs can only render points and triangles (and also thin aliased lines that have limited use when it comes to high-quality antialiased rendering). In the early 2000s, with the programmable pipeline, OpenGL offered developers a way to customize, in so-called *shaders* , the transformation of points in the 3D space (*vertex shaders*), and the color of every pixel in each primitive (point, line, or triangle, in so-called *fragment shaders*). Historically, these features were designed for advanced transformation and lighting effects, but they can also be used for very different purposes. Shaders are central to GPU-based scientific visualization, especially when it comes to 2D rendering.

To take full advantage of the Vulkan rendering pipeline, we have developed Datoviz in several layers to achieve modularity, separation of concerns, and facilitate testing and maintenance.

Vulkan thin wrapper. If Vulkan provides a rich, powerful, and complex C API, Datoviz aims at providing a thin wrapper on top of it for simplification purposes, leading to C code that is typically about four times shorter. Only the essential features used when doing scientific visualization purposes are covered in this wrapper. More precisely, the wrapper provides functions to store data on the GPU (in either raw binary buffers or 1D, 2D, or 3D textures); to create graphics and compute pipelines with custom shaders; to record commands on the GPU; to perform synchronization between the CPU and the GPU; and to implement a swapchain-aware rendering loop.

Graphics primitives. Providing the foundations for writing graphics pipelines with custom shaders and managing data on the GPU memory is only a first step. A second step is to provide ready-to-use *graphics primitives* that are frequently used in scientific computing. It turns out that 3D graphics are much simpler to implement than high-quality antialiased 2D graphics. Meshes and volumes are the most typical 3D graphical

primitives and they are natively supported in Datoviz.

Two-dimensional graphics such as line segments, continuous paths, markers, arrows, and text are much harder to implement on the GPU, as one is limited to points, single pixel width lines and triangles. These visual elements must be implemented manually in custom shaders. Several years ago, Nicolas P. Rougier designed GLSL algorithms for high-quality antialiased 2D graphics (Rougier, 2013)(Rougier, 2014). Although his code used OpenGL, it was straightforward to port it to Vulkan, as Khronos provides a compiler toolchain that natively supports GLSL (Vulkan uses a low-level shader intermediate language called SPIR-V). For example, curved paths are rendered by triangulating them as a function of the path thickness. Antialiasing is implemented in the fragment shader: for each pixel, the distance of the pixel to the path median along the normal axis is computed. The transparency value of that pixel is a function of that distance. Signed distance fields are also used for text (Chlumský, 2015), arrows, markers, and other 2D elements.

An advantage of this approach is that it unifies 2D and 3D graphics within the same underlying graphics stack, which is not typical in existing scientific visualization solutions.

Visuals. A *visual* encompasses one or several graphics pipelines and provides a way to map user-facing data (marker positions, path colors, text contents, and so on) to GPU vertex data and parameters. The visual's developer is free to implement arbitrary data transformations, although typically, user-level data is just copied into the vertex buffer. The distinction between graphics primitives and visuals is that the latter provide a way to hide GPU implementation details of the visual elements from the end-user.

Datoviz provides a library of common ready-to-use visuals: line segments, paths, markers, arrows, images, text, meshes. As an example, the polygon visual relies on a standard triangle graphics pipeline, but it also implements a custom transformation function that performs an automatic triangulation of a polygon into a set of triangles, for example using the earcut algorithm (Meisters, 1975).

Datoviz also allows the user to write entirely custom visuals, although this is expected to be an advanced use-case.

The scene. The scene puts everything together: visuals are organized into subplots (or panels) arranged in a two-dimensional grid layout. Each panel may implement various types of interactivity: pan and zoom, 3D arcball, and first-person cameras, depending on the type of scientific visualization. Interactivity is implemented by mapping mouse and keyboard events to real-time updates of GPU parameter values (typically, model, view, and projection matrices). Datoviz also natively provides a library of 150 common colormaps.

Graphical user interfaces (GUIs). Datoviz natively supports the Dear ImGui C++ library (`https://github.com/ocornut/imgui`), which implements rich, complex GUI controls directly on the GPU using Vulkan. This allows for fast, low-footprint rendering of GUIs and leaves most of the computing power to the scientific data and visual elements. This library uses immediate rendering: the controls are recreated at every frame on the GPU, which is, perhaps surprisingly, quite fast given the relative low complexity of typical GUI elements.

General-purpose GPU computing. Another selling point of Vulkan and Datoviz is native support for GPGPU computing for visualization purposes. Compute shader support is mandatory in Vulkan implementations, and Datoviz fully supports it. Compute shaders are similar to CUDA or OpenCL kernels. These GPU programs can access GPU buffers and textures arbitrarily, in a massively parallel fashion. When the same GPU buffers or textures are used for both compute and rendering, highly complex and dynamic visualizations may be implemented. For example, one could simulate systems of partial differential equations in real-time using a numerical method on the GPU, and display the evolution of the system without requiring repeated transfers between the host and the GPU. Interoperability with CUDA kernels is also a possibility that needs to be investigated more thoroughly in the future. It is worth noting that proper synchronization of GPU objects needs to be carefully implemented using functionality provided by Vulkan and Datoviz.

Scientific applications

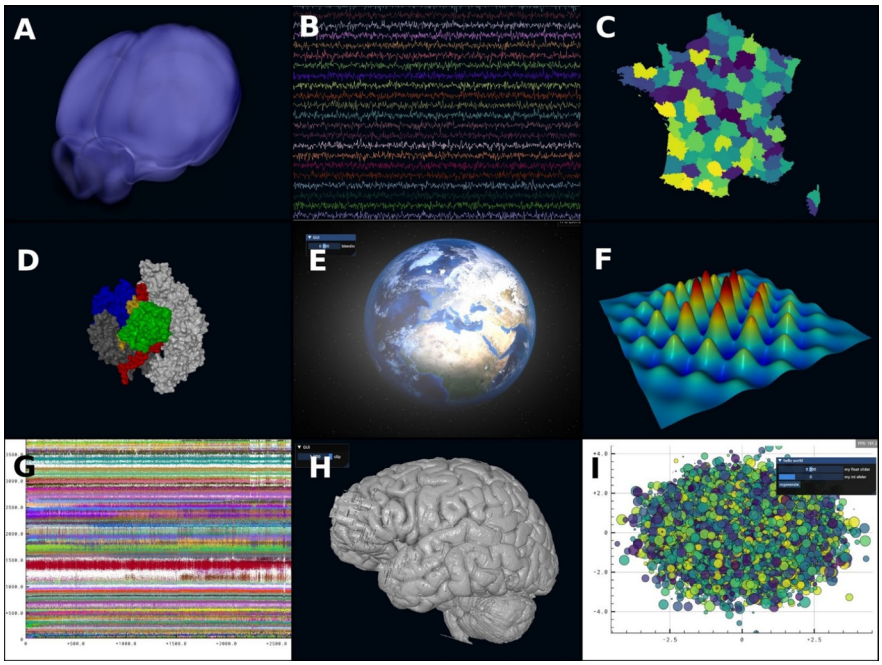

Figure 2: **Datoviz rendering example.** A. The Allen Mouse Brain volume (Wang et al., 2020) is rendered with a standard raymarching algorithm written in the fragment shader. B. Several multichannel time-dependent signals, rendered with fast, but low-quality, GPU lines. C. A choropleth map implemented as a set of polygons triangulated with earcut, a C++ implementation of the ear-clipping algorithm (Mapbox earcut library, `https://github.com/mapbox/earcut`). D. A 3D molecule rendered as a 3D mesh (RCSB PDB, 4UN3 Crystal structure of Cas9 bound to PAM-containing DNA target, `https://www.rcsb.org/3d-view/4UN3/1`). E. An example of two images that are blended on the GPU (Earth image: Pixabay PIRO4D). F. A 3D lighted and colored surface plot. G. A raster plot, a particular type of scatter plot where each dot represents an action potential of a neuron, the x-axis is the action potential time, and the y-axis is the neuron depth within the cortex. H. An ultra-high-resolution of a 3D brain mesh (Alkemade et al., 2020) (courtesy of Pierre-Louis Bazin). I. A standard scatter plot, with a small graphical user interface controlling the marker size and colors.

Figure 2 shows a few visualization examples implemented with Datoviz.

The level of performance powered by Vulkan is unprecedented in scientific visualization. For example, on a 2019 high-end NVIDIA RTX 2070 SUPER GPU, a scatter plot with ten million points renders at 250 FPS; a mesh with ten million triangles and standard lighting renders at 400 FPS; and a set of one thousand signals containing 30,000 points each (total of 30 million points) renders at 200 FPS.

Future perspectives

We are still improving the robustness of the code and coding practices: testing, continuous integration, and documentation. While the current version of the code is open to all currently (*`https://github.com/datoviz/datoviz`*), a first packaged release will be done once the project reaches sufficient maturity in these respects.

Datoviz's long-term plan is to offer a simple, relatively low-level but highly performant graphical API that people can use to build higher-level packages such as Matplotlib, VisPy, or Napari. Datoviz is not meant to be used directly by scientist end-users who expect ready-to-use plotting functions. A parallel could be made with NumPy that is the foundation of many libraries and software (Harris et al., 2020), whereas SciPy provides more advanced functionality. We plan to investigate the integration of Datoviz as a low-level backend for a future version of VisPy, and have a clear separation between the low-level graphics backend, and the high-level user plotting interface.

In that context and based on our experience with VisPy, we want to emphasize the critical role of documentation in open source software. If VisPy received a warm welcome from the community, lack of documentation represented a high entry barrier to many end-users. By contrast, Datoviz is already heavily documented and we intend to continue this effort. In particular, we plan to write detailed tutorials for beginners and fully document the API.

Datoviz is a desktop-only library at the moment. Web integration (e.g., Jupyter) would be an interesting avenue of research and development in the future. Two main approaches are possible: real-time video streaming with a GPU-powered visualization server, or WebGPU / WebAssembly, which is still at the specification and development stage by the major browser vendors.

Conclusion

We reported initial work towards a new fast and scalable scientific visualization technology that leverages the Vulkan API to achieve unprecedented performance through GPUs. This technology is implemented in a C/C++ library called Datoviz that offers an intermediate-level API for scientific visualization libraries and software. Datoviz provides a unified graphics stack for 2D, 3D, graphical user interfaces, and natively supports efficient interactions between rendering and general-purpose GPU computing. A major direction of development is to investigate the integration of Datoviz as a low-level backend of a future version of VisPy, a popular Python scientific plotting library.

Author bios

Cyrille Rossant is a neuroscience researcher and software engineer at the International Brain Laboratory and University College London. He has been investigating GPU for scientific visualization for a decade, and he is one of the original VisPy developers. Contact him at cyrille.rossant@gmail.com.

The **International Brain Laboratory** (IBL) is a virtual laboratory, unifying a group of 22 highly experienced neuroscience groups distributed across the world. Its mission is to develop a standardized mouse decision-making behavior, to make coordinated measurements of neural activity across the mouse brain, and to use theory and analyses to uncover the neural computations that support decision-making. The IBL is committed to the use and development of open source tools and open access to data.

Nicolas Rougier is a researcher at Inria Bordeaux Sud-Ouest and team leader at the Institute of Neurodegenerative Diseases (Bordeaux, France). He has written several Python libraries, articles and books about scientific visualization and is one the original VisPy developers. Contact him at Nicolas.Rougier@inria.fr.

References

Mayavi: 3D Visualization of Scientific Data. (2011). *Computing in Science & Engineering*, *13*(2), 40–51. https://doi.org/10.1109/mcse.2011.35

VisPy: harnessing the GPU for fast, high-level visualization. (2015). *Proceedings of the 14th Python in Science Conference*. https://hal.inria.fr/hal-01208191

Cython: The Best of Both Worlds. (2011). *Computing in Science & Engineering*, *13*(2), 31–39. https://doi.org/10.1109/mcse.2010.118

Shader-based antialiased dashed stroked polylines. (2013). *Journal of Computer Graphics Techniques*, *2*(2), 91–10. https://hal.inria.fr/hal-00907326

Antialiased 2d grid, marker, and arrow shaders. (2014). *Journal of Computer Graphics Techniques*, *3*(4), 52. https://hal.archives-ouvertes.fr/hal-01081592

Shape Decomposition for Multi-channel Distance Fields. (2015). [Master's thesis]. Czech Technical University.

Polygons Have Ears. (1975). *The American Mathematical Monthly*, *82*(6), 648–651. https://doi.org/10.1080/00029890.1975.11993898

The Allen Mouse Brain Common Coordinate Framework: A 3D Reference Atlas. (2020). *Cell*, *181*(4), 936–953.e20. https://doi.org/10.1016/j.cell.2020.04.007

7 Tesla MRI Followed by Histological 3D Reconstructions in Whole-Brain Specimens. (2020). *Frontiers in Neuroanatomy*, *14*. https://doi.org/10.3389/fnana.2020.536838

Array programming with NumPy. (2020). *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2