



Tales from the Trenches: Developing sciview, a new 3D viewer for the ImageJ community

Ulrik Günther^{1,2,3}  and Kyle I.S. Harrington⁴ 

¹CASUS – Center for Advanced Systems Understanding, Görlitz, Germany

²Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany

³Center for Systems Biology, Dresden, Germany

⁴Virtual Technology & Design, University of Idaho, Moscow, Idaho, USA

Abstract

ImageJ/Fiji is a widely-used tool in the biomedical community for performing everyday image analysis tasks. However, its 3D viewer component (aptly named 3D Viewer) has become dated and is no longer actively maintained. We set out to create an alternative tool that not only brings modern concepts and APIs from computer graphics to ImageJ, but is designed to be robust to long-term, open-source development. To achieve this we divided the visualization logic into two parts: the rendering framework, scenery, and the user-facing application, sciview. In this paper we describe the development process and design decisions made, putting an emphasis on sustainable development, community building, and software engineering best practises. We highlight the motivation for the Java Virtual Machine (JVM) as a target platform for visualisation applications. We conclude by discussing the remaining milestones and strategy for long-term sustainability.

CCS Concepts

• **Software and its engineering** → Software development techniques; • **Computing methodologies** → Rendering; Graphics systems and interfaces;

1. Introduction

Scientific image processing and analysis is common throughout the scientific and engineering disciplines. While there are numerous software tools that support scientific image processing, one of leading open-source tools is ImageJ [SRE12]. ImageJ is a Java-based tool that dates back to 1997[†], and continues to be developed to this date. ImageJ was developed using Java to facilitate portability between systems. Additional features, such as a plugin system, have contributed to ImageJ's longevity. However, a number of limitations inherent to ImageJ's design were revealed in the 2010s, such as a fragile plugin ecosystem, and limited support for large images.

To alleviate these limitations of ImageJ, the Fiji (Fiji Is Just ImageJ) distribution of ImageJ [SAF*12] was developed. Fiji introduced an update site mechanism for managing and distributing ImageJ plugins. Fiji also introduced support for a new backend library for image support, ImgLib2 [PPTS12]. The Fiji distribution revolutionized the ImageJ community by making numerous ImageJ plugins readily accessible within the tool itself. This is accomplished via *update sites*, where the user can easily select plugin sources and manage plugin updates using a GUI, as opposed to the original

approach which involved browsing to websites, downloading a pre-compiled JAR or compiled Java class, and installing it into ImageJ. Additionally, the introduction of the ImgLib2 library enabled support for the large scale image data that has now become popular, where original ImageJ data structures were limited to 2^{31} pixels, corresponding to the maximum integer value that can be used to index into an array of pixel data. Fiji also introduced the original 3D viewer of ImageJ [SSC*10].

Since the introduction of Fiji, we have developed a new visualization framework for the JVM to prototype VR interactions with scientific instruments, such as high-speed volumetric microscopes. This framework, *scenery*, has been designed to support visualisation of large amounts of image data [GPG*19b], generated by microscopes that can easily output images at a rate of 1GB/s or higher [RPHT14]. The increasing abundance of imaging data with total sizes reaching into the terabytes requires new techniques, such as out-of-core volume rendering, to be broadly available to researchers that face these challenges. ImageJ and the Fiji distribution of ImageJ continue to be popular tools for scientific image processing and analysis. This includes the use of ImageJ for 3D image analysis and visualization. However, Fiji currently does not support such advances from the visualization field. To this end we designed and developed *sciview*, a scenery- and ImageJ- based tool for supporting visualisation of N-dimensional data.

[†] ImageJ is the Java-based successor to the NIH Image software package.

2. Motivations and Technical Aspects

As a bridge between ImageJ and scenery, sciview provides visualisation support for commonly used data structures and algorithms for scientific image processing. For example, a user may apply a smoothing operation to an input image volume, and inspect the results via volumetric rendering. The integration of data and visualisation is not limited to voxel data. For example, ImageJ includes an implementation of the marching cubes algorithm for generating triangulated meshes from volumetric data, and sciview enables visualisation of these mesh outputs overlaid with the input volume data. Additional algorithms from ImageJ, such as voxelization, convex hull calculation, and 3D cropping, are also made available in sciview. In general, functionality that requires both the visualization and/or 3D interaction features of scenery, and image processing features of ImageJ, is developed within sciview.

A key feature of scenery that is leveraged by sciview is out-of-core volume rendering, where a subset volumetric data is loaded into GPU memory as needed. This feature is critical with the previously mentioned rapid growth of biological imaging data. Apart from out-of-core volume rendering, we thought the VR support already present in scenery might be of more general use for the biomedical community. While the original 3D Viewer only supported OpenGL 1.2, we also aimed for support of more up-to-date APIs – a major reason for the abandonment of the original 3D Viewer was its strong dependency on Java3D, which has not been updated since Java 6 was released in 2006. In addition to supporting more up-to-date APIs, we observed that it would be beneficial to design a rendering framework that can utilize multiple APIs, and is not strongly tied to a single API.

The full goals for the development of scenery are [GPG*19a]:

- G_1 **Virtual/Augmented Reality support:** The framework should make the use of VR/AR in an application possible with minimal effort. Distributed systems, such as CAVEs or Powerwalls, should also be supported.
- G_2 **Out-of-core volume rendering:** The framework should be able to handle datasets that do not fit into graphics memory and/or main memory, contain multiple channels, views, and timepoints. It should be possible to visualize multiple such datasets in a single scene.
- G_3 **User/Developer-friendly API:** The framework should have a simple API that makes only limited use of advanced features, such as generics, so the user/developer can quickly comprehend and customize it.
- G_4 **Cross-platform:** The framework should run on the major operating systems: Windows, Linux, and macOS.
- G_5 **JVM-native and embeddable:** The framework should run natively on the JVM, and be embeddable, such that it can be used in popular biomedical image analysis tools like Fiji [SAF*12, RSH*17], Icy [dCDC*12], and KNIME [BCD*08].

While scenery focuses on establishing a rendering framework based upon the above goals, sciview translates these goals into interfaces that are accessible from ImageJ. Two key efforts of sciview involve making data from ImageJ available to scenery for rendering (volumes, meshes, point clouds, etc.) and introducing new al-

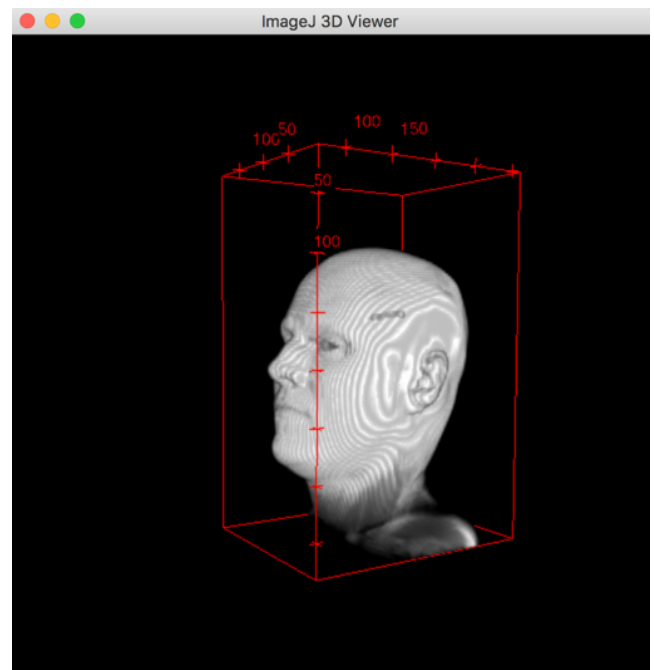


Figure 1: Example volume rendering from Fiji’s original 3D viewer.

gorithms for both image and mesh processing. To this end, sciview primarily consumes ImageJ and ImgLib2 data structures, and generates/manipulates scenery data structures for visualisation.

3. Approach

With ImageJ being Java-based, the choice of target platform was already fixed, but the actual language was not. While the new viewer would need to run on the Java Virtual Machine (JVM), a lot of new languages that improve upon the developer experience with Java came up when development started: The closest contenders were Kotlin and Clojure (which is a Lisp dialect). We ultimately decided for Kotlin, as it is commercially supported by the company JetBrains, the developers of the popular IntelliJ IDE. Kotlin later became a first-class citizen on Android, contributing much to its popularity nowadays – it is one of the fastest-growing languages currently on Github[‡]. Kotlin offers a way more concise writing style than Java does, and introduces additional functional programming constructs. When comparing Kotlin and Clojure, Kotlin is notably more similar to Java, making it an appealing choice for this case due to the large number of Java programmers involved in the ImageJ community. In the JVM world, stable build systems and good dependency management have a long and successful history. In our case, we use Maven[§] as a build system, which makes the build as easy as running the `mvn package` command, and is probably the most widespread build system used in the JVM ecosystem.

[‡] See octoverse.github.com/#top-languages.

[§] See maven.apache.org



Figure 2: A volume rendering of a developing *Drosophila* embryo, rendered on a CAVE system. Distributed rendering support is handled by scenery, but not yet exposed in sciview. Image from [GPG*19b].

As mentioned before, our approach to developing the 3D Viewer replacement consists of two parts, *scenery*, the rendering backend and framework, and *sciview*, the user-facing frontend, which is the actual ImageJ/Fiji plugin. This split enables us to integrate new features into scenery, test them, and expose them later in sciview – for example, support for distributed rendering (see Figure 2) is implemented in scenery and usable, but not yet exposed in sciview. It also enables more stability for user-created scripts, where large backend changes in scenery are “buffered” by sciview, minimizing the number of changes that must be propagated to user scripts. While the user can choose to use pure scenery objects in her code, it is not necessary. We will continue with a description of the two parts, starting with scenery:

In terms of rendering, scenery originally featured only an OpenGL 4.1-based renderer, with version 4.1 being chosen in order to support macOS. The Vulkan renderer was already being planned, and was added about a year after development started. The development of the Vulkan renderer led to a re-thinking of the renderer architecture, which led to improvements on both the Vulkan and the OpenGL side. In scenery, the renderers are completely decoupled from the rest of the library, such that the rendering-independent components can be tested on their own, even on systems without GPUs. The renderers are also interchangeable at runtime and feature a render graph architecture for the actual rendering pipeline, with the rendering pipelines also being exchangeable at runtime. To interface with native APIs, we use the LWJGL (Vulkan) and JOGL (OpenGL) libraries. While JOGL tries to wrap the OpenGL state machine into an object-oriented architecture, LWJGL provides access to the raw APIs. This means that in contrast to regular JVM applications, memory management has to be handled by the developer and not by the garbage collector as usual. However, LWJGL provides excellent and fast APIs for memory management[¶].

Scene contents are handled in a mostly traditional scene graph architecture, which we restricted to a scene tree [BK15], for more efficient parallel discovery of scenes with a large number of ele-

ments, as we require, e.g. for rendering large number of cells or neurons.

Originally, scenery featured in-core volume rendering, being limited by the available memory of the GPU. In 2019, we have switched to an out-of-core volume rendering architecture based on technology developed from and in conjunction with the BigDataViewer [PSPT15] project. The technique implemented there is a combination of the previously-published approach hierarchical blocking [LHJ99, BHMF08] and the missing data scheme introduced in the BigDataViewer paper [PSPT15]. This enables us to not only render very large datasets, but also run filters on volumetric data on-the-fly.

4. Sustainability

In this section, we want to detail decisions we have made in order to make the project sustainable in the longer run, split into technical aspects, and community/social aspects:

4.1. The Technical Side: Kotlin, Polyglot Scripting, and Continuous Integration/Delivery

With the choice of Kotlin as a language, we believe to have made a sustainable choice, for multiple reasons: First, should the JVM at some point not be sufficient anymore for our endeavours, Kotlin offers experimental support for compilation to native machine code^{||} via LLVM [LA04]. Second, Kotlin can also be compiled to JavaScript, enabling the possibility of a web-based viewer with the same basis – although this would require additional WebGL/WebGPU rendering code. Third, all software written in Kotlin is completely interoperable with existing Java/JVM software, and to the developer it is mostly transparent whether Java or Kotlin was used. Kotlin libraries could even be decompiled to Java code.

In terms of extensibility, we offer a plugin-based system building on top of SciJava (www.scijava.org), and polyglot scripting of both scenery and sciview, with JavaScript, Jython, (J)Ruby, Matlab, BeanShell, Java, Scala, and Clojure being the currently available options. We hope this lowers the entry barrier to interacting with our software on a programmatic level (it will be interesting to see which language becomes the most-often used). The plugins required for interacting with all of these languages are developed in the SciJava community, are easily integrated, and do not need to be maintained by our team, which is a large plus. Furthermore, with the rising popularity of Python, a Kotlin-Numpy bridge has also been developed by the community (github.com/kotlin/kotlin-numpy/), which we hope to be able to utilise in the future.

With the choice of Vulkan as an additional rendering API, we have stepped a bit into unknown territory, as it is a new API that has not been used extensively in scientific visualisation. Reasons for this choice were that Vulkan maps better to the current architecture of GPUs than OpenGL does, and provides a better developer experience, with excellent, and fine-grained debugging tools, like the Vulkan Validation Layers. These can be activated during

[¶] See blog.lwjgl.org/memory-management-in-lwjgl-3/.

^{||} See kotlinlang.org/docs/reference/native-overview.html.

development and perform runtime checks, which are mostly absent from the driver – a stark contrast to OpenGL. Performance is enhanced in multiple ways: Vulkan offers much better multithreading support, and rendering/compute calls are sent to the GPU by submitting work in batches via so-called command buffers, which can (and should!) be reused as much as possible. Finally, Vulkan offers more fine-grained synchronisation primitives than OpenGL. As with OpenGL, Nvidia offers interoperability with CUDA, e.g. for sharing buffers and images. The downside to using Vulkan is the up-front investment necessary: Vulkan differs significantly from OpenGL, is more verbose, and has a very steep learning curve – especially synchronisation is something that bites rather often, and not all issues on that side can be caught reliably by the validation layers. All in all however, these downsides are more than balanced out by the improved development experience and performance improvements. Programmers familiar with similarly explicit APIs like DirectX 12 or Metal will probably not find it difficult to learn. In the medium run, we plan on retiring the OpenGL renderer, as we see Vulkan as the more future-proof API. In the other direction, considering that Vulkan is already supported by currently 8 year-old Kepler-generation GPUs and later ones, we feel we do not need to worry about backwards compatibility too much. Unfortunately, one problem remains: Apple decided to take its own path, dropping out of the Khronos Group’s Vulkan Working Group, and develop Metal – so Vulkan is not natively available on macOS. However, a compatibility layer, *MoltenVK*, was developed that maps between Vulkan and Metal, which has now been handed over to the Khronos Group for further development (see github.com/KhronosGroup/MoltenVK). We would like to use MoltenVK in the future to also support rendering via Vulkan on macOS. Further, we plan on adding a software renderer, probably using OSPRay [WJA*17] in the future. Finally, Vulkan does not feature the strong coupling to the windowing system that OpenGL had anymore, so running on headless systems is much easier than with OpenGL.

For automated testing, we have created a Continuous Integration (CI) pipeline based on Travis CI (travis-ci.org) that automatically runs unit tests and checks code coverage (via jacoco and codecov.io) on each commit to our git repositories. In addition, scenery includes a range of examples, which are also used as integration tests. At the moment, we are changing our CI pipeline to use Gitlab CI and automatically run these integration tests headlessly on different GPUs, and compare the resulting renderings with known-good images. Taking unit tests and integration tests together, we can reach a code coverage of about 70%**. Testing code paths that require external hardware, such as HMDs, remain a problem, though.

sciview (and with it, scenery) is delivered to the user through an ImageJ update site: Release versions are created manually, and in addition, we have a separate nightly update site, which is updated with each commit to the git repositories, facilitating Continuous Delivery (CD). The latter enables a fast response cycle for user-

reported issues and for testing new developments with a limited set of users.

In the previous section, we have already mentioned the Maven-based build system – in addition to providing an easy way to build our software, Maven also sets rather strict requirements for how the software is distributed: Maven-built software can be deployed to central repositories, such as Maven Central (sonatype.org), with each released version of a software being immutable and digitally signed, contribution to both security and reproducibility of builds. For versioning scenery and sciview, we use semantic versioning (semver.org), where major version changes indicate incompatible API changes, and minor version changes indicate backwards-compatible API changes. Now, we cannot (and probably should not) create a new release for each commit to the repository, so in order to keep sciview up-to-date with respect to current developments in the scenery repository, we utilise JitPack (jitpack.io), where dependencies can be declared not only on release versions, but directly on Git commit hashes. The downside of this is that we cannot use semantic versioning for non-release builds, but we can guarantee that any build done in the past is going to work in the future, boosting reproducibility, even with versions still in development. The versions used, may it be the actual release version, or the Git commit used, are also shown to the user and in the log file, making error tracing easier for the developers.

4.2. The Social Side

On the more soft side of things, we have found it slightly difficult to have people move away from 3D Viewer, even though both outdated and unmaintained, simply because they are used to it, and of course habits are difficult to change. A recent innovation in the ImageJ community, the image.sc forum for image analysis – instead of the mailing list used over decades – has alleviated that a bit, and led to more widespread knowledge about our project. We have also realised that the tool most often required is a simple piece of software users can just drop a single image or a time series onto. One of the authors was involved with the development of ClearVolume [RWG*15], a visualisation tool originally intended to be used for live visualisation for microscopes, directly on the machine where the images are acquired. From the citations of the paper, we have noticed that ClearVolume is substantially more often used as Fiji plugin for visualisation of already acquired datasets than as software for actual live microscopy visualisation, and that is probably because it makes it very simple to just view a single image or a time series.

In the meantime, according to GitHub, there are 18 projects using either scenery or sciview, with a six of them being our own. Exemplary usage of our tools include:

- *Multi-sample imaging and visualisation* [DGM*19], a paper using sciview to investigate, visualise, and analyse the development of vasculature in *Danio rerio* (zebra fish) (see Figure 3A),
- *EmbryoSim*, a toolkit for the generation of plausible-looking *Drosophila* (fruitfly) microscopy images for the training of machine learning algorithms for cell tracking and segmentation (see Figure 3B),
- *SNT*, a tool for neuron tracing, and successor to Simple Neurite

** On Github, around 25% coverage is shown at the moment, because the Travis pipeline used at the moment does not yet run the tests requiring a GPU. Pending further testing, we will switch to the new CI pipeline in the near future.

Tracer [LBA11], which uses sciview as a viewer (see Figure 3C), and

- *Visualising regeneration in the mouse incisor*, a writeup in the popular biology blog, the Node, featured visualisations from sciview ††

In the latter two, the authors have been helping out with the development, an example of the tag team approach described in [RCM*20].

For coordinated development, we hosted a hackathon dedicated to sciview development at the University of Idaho in 2018, and will again organise one in 2020, this time either at MPI-CBG in Dresden, or at CASUS in Görlitz. Hackathons have proven to be invaluable tools in the ImageJ community to drive development forward, and define new development goals. While we also heavily utilise online communication, e.g. via Gitter (gitter.im/scenerygraphics/SciView/), in-person meetings, with the associated socialising seem irreplaceable.

In order to keep development organised online, we utilise Github's Pull Request (PR) features heavily: All new features and bug fixes are submitted as a pull request before being merged into the master branch. This enables both code review and testing of code before something broken or inadequate reaches the main line code. For each PR, we run our CI tests on Windows, macOS, and Linux, and in addition run the code coverage and code quality tools. Issues in any PR are then resolved by the team members interactively, before the PR is finally merged, or rejected (with most PRs being actually accepted in the end).

5. Outlook

While we believe our approach on the technical side is sound, there are still issues that we need to address:

- *Documentation* – The JVM ecosystem forces the developer to document their code via Javadoc (Java) or Kdoc (Kotlin) by requiring dedicated documentation packages to be deployed to the central Maven package repository (see e.g. javadoc.scijava.org/SciView/). We also require that new features in the form of Pull Requests on Github have adequate documentation. But as API-level documentation is not enough, we have started to write better, more explanatory documentation using Gitbook at docs.scenery.graphics. The documentation there is unfortunately far from complete. Already mentioned by [RCM*20], software documentation often is the last part in the scientific process, and very unrewarding in the short-term, especially when the metric is "publications produced". However, there cannot be sustainable software development without adequate documentation.
- *Funding* – while the authors both develop and use scenery and sciview, and try to publish technology and papers based on them, it is sometimes difficult to acquire funding "just" for software, an issue already mentioned in [RCM*20]. Slowly, the large funding bodies, such as DFG in Germany, or the NIH in the US, are

recognising the need for longer-term, stable software development, that sometimes also has to be done by the scientists requiring the software. We hope that this trend continues. Commercialisation might be an alternative option, but as we would ensure that our software stay open-source, it is not an easy task to derive a viable business model.

- *Future prospects* – The integration of sciview into the ImageJ ecosystem enables new technologies within the bioimage analysis community. By making the scenery framework accessible within ImageJ, the large number of existing ImageJ-based tools can now utilize virtual reality technologies. scenery's support for multiple rendering APIs introduces additional long-term stability as the landscape of computer graphics continues to evolve. New features introduced through sciview and scenery span from new image and mesh processing algorithms to out-of-core volume rendering, serving to extend beyond current ImageJ-based visualisation tools.
- *Less Vis Gap* – While we are mostly at home in the biomedical imaging community, we would like to interface better with the computer graphics community, to ensure that new algorithms and developments are incorporated into scenery and sciview. We have started this process already by establishing collaborations in the graphics field.

6. Conclusions

In this paper, we have discussed the motivation and development process behind sciview and scenery, where sciview is intended to be the replacement for the current 3D Viewer in the ImageJ ecosystem. We have outlined how we believe language and ecosystem choice has an impact on sustainable development, and what tools we found valuable in supporting the developer. We have also described some major issues we are facing, that have already been discussed by [RCM*20], and are common to a lot of visualisation software packages.

Acknowledgements

We would like to thank all the contributors to the scenery and sciview projects, especially Tobias Pietzsch and Curtis Rueden. We would also like thank the the whole Fiji/ImageJ community for their ongoing support of our efforts. We further thank Simeon Ehrig and Tobias Huste from HZDR for their support in getting our headless CI tests going with different GPUs. We thank Aryaman Gupta, Ivo F. Sbalzarini, and Justin Bürger both for providing valuable input, and for proofreading this paper.

This work was partially funded by the Center of Advanced Systems Understanding (CASUS) which is financed by Germany's Federal Ministry of Education and Research (BMBF) and by the Saxon Ministry for Science, Culture and Tourism (SMWK) with tax funds on the basis of the budget approved by the Saxon State Parliament.

References

- [BCD*08] BERTHOLD M. R., CEBRON N., DILL F., GABRIEL T. R., KÖTTER T., MEINL T., OHL P., SIEB C., THIEL K., WISWEDEL B.: KNIME: The Konstanz Information Miner. In *Data Analysis, Machine*

†† See thenode.biologists.com/a-gnawing-question-which-cells-are-responsible-for-tooth-renewal-and-regeneration/research/, and for the article [SMZ*19].

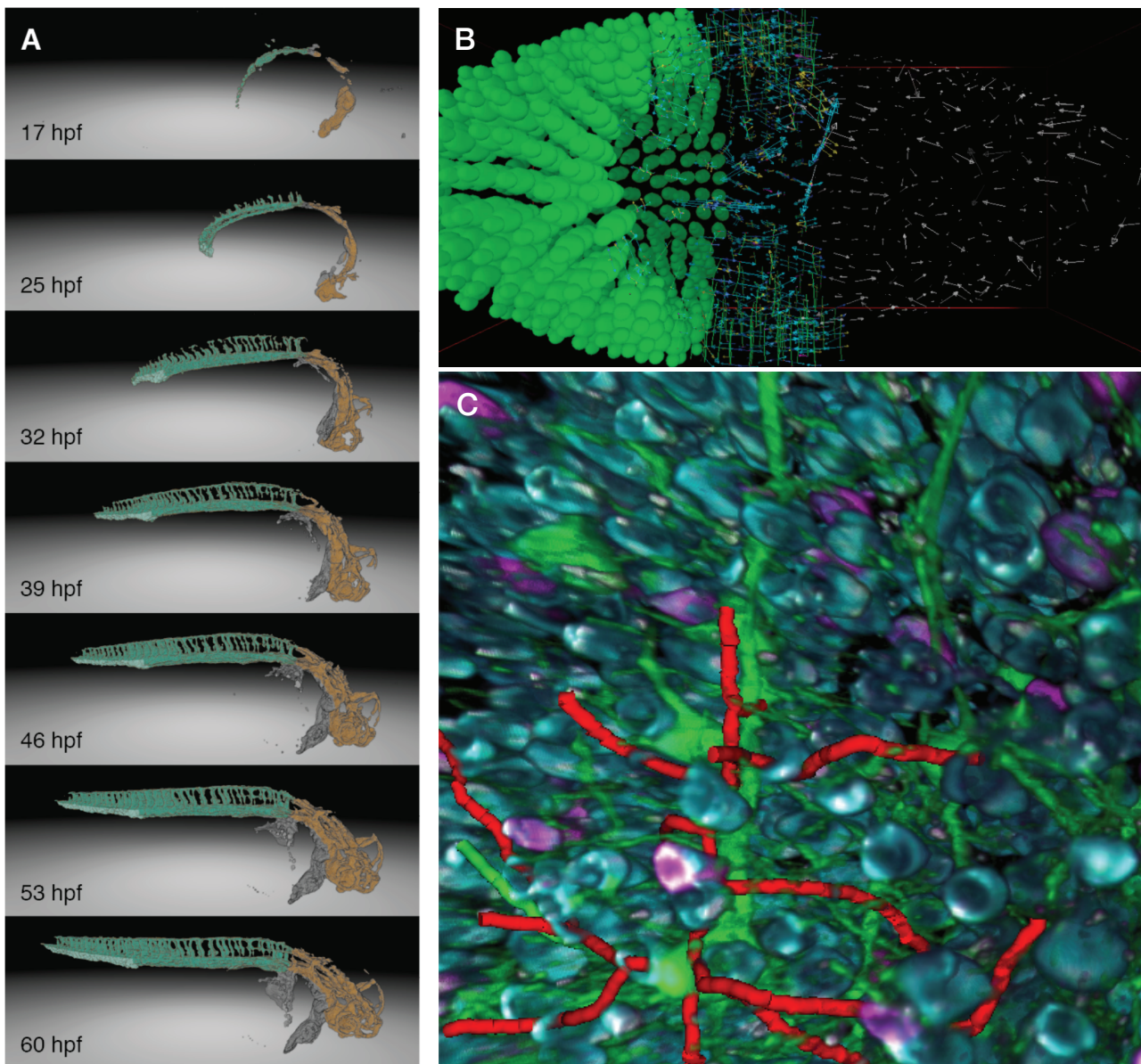


Figure 3: Example applications: A: Zebrafish vasculature timelapse, shown from 17 to 60 hours post fertilisation (hpf), using ambient occlusion to highlight smaller details; B: EmbryoSim, with forces acting on individual cells shown as arrows, and cell bodies not shown towards the right; C/D: SNT for neuron tracing in volumetric images, with neurons shown in red, embedded in a dense volumetric image.

- Learning and Applications*, Preisach C., Burkhardt H., Schmidt-Thieme L., Decker R., (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 319–326. doi:10.1007/978-3-540-78246-9_38. 2
- [BHMFO8] BEYER J., HADWIGER M., MÖLLER T., FRITZ L.: Smooth Mixed-Resolution GPU Volume Rendering. 3
- [BK15] BOUDIER P., KUBISCH C.: GPU Driven Large Scene Rendering. In *GPU Technology Conference* (San Jose, 2015). 3
- [dCDC*12] DE CHAUMONT F., DALLONGEVILLE S., CHENOUEAU N., HERVÉ N., POP S., PROVOOST T., MEAS-YEDID V., PANKAJAKSHAN P., LECOMTE T., MONTAGNER Y. L., LAGACHE T., DUFOUR A., OLIVO-MARIN J.-C.: Icy: An open bioimage informatics platform for extended reproducible research. *Nature Methods* 9, 7 (2012). doi:10.1038/nmeth.2075. 2
- [DGM*19] DAETWYLER S., GÜNTHER U., MODES C. D., HARRINGTON K. I., HUISKEN J.: Multi-sample SPIM image acquisition, processing and analysis of vascular growth in zebrafish. *Development* (2019). doi:10.1242/dev.173757. 4
- [GPG*19a] GÜNTHER U., PIETZSCH T., GUPTA A., HARRINGTON K. I., TOMANCAK P., GUMHOLD S., SBALZARINI I. F.: Scenery: Flexible Virtual Reality Visualization on the Java VM. In *2019 IEEE Visualization Conference (VIS)* (Vancouver, BC, Canada, Oct. 2019), IEEE, pp. 1–5. doi:10.1109/VISUAL.2019.8933605. 2
- [GPG*19b] GÜNTHER U., PIETZSCH T., GUPTA A., HARRINGTON K. I. S., TOMANCAK P., GUMHOLD S., SBALZARINI I. F.: Scenery: Flexible Virtual Reality Visualization on the Java VM. *arXiv:1906.06726 [cs]* (June 2019). arXiv:1906.06726. 1, 3
- [LA04] LATTNER C., ADVE V.: LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (San Jose, CA, USA, 2004), IEEE, pp. 75–86. doi:10.1109/CGO.2004.1281665. 3
- [LBA11] LONGAIR M. H., BAKER D. A., ARMSTRONG J. D.: Simple Neurite Tracer: Open source software for reconstruction, visualization and analysis of neuronal processes. *Bioinformatics* 27, 17 (2011). doi:10.1093/bioinformatics/btr390. 5
- [LHJ99] LAMAR E., HAMANN B., JOY K.: Multiresolution techniques for interactive texture-based volume visualization. *Proceedings Visualization '99 (Cat. No.99CB37067)* (1999). doi:10.1109/visual.1999.809908. 3
- [PPTS12] PIETZSCH T., PREIBISCH S., TOMANČÁK P., SAALFELD S.: ImgLib2—generic image processing in Java. *Bioinformatics* 28, 22 (2012). doi:10.1093/bioinformatics/bts543. 1
- [PSPT15] PIETZSCH T., SAALFELD S., PREIBISCH S., TOMANCAK P.: BigDataViewer: Visualization and processing for large image data sets. *Nature Methods* 12, 6 (2015). 3
- [RCM*20] REINA G., CHILDS H., MATKOVIĆ K., BÜHLER K., WALDNER M., PUGMIRE D., KOZLÍKOVÁ B., ROPINSKI T., LJUNG P., ITOH T., GRÖLLER E., KRONE M.: The moving target of visualization software for an increasingly complex world. *Computers & Graphics* 87 (Apr. 2020), 12–29. doi:10.1016/j.cag.2020.01.005. 5
- [RPHT14] REYNAUD E. G., PEYCHL J., HUISKEN J., TOMANCAK P.: Guide to light-sheet microscopy for adventurous biologists. *Nature Methods* 12, 1 (2014). doi:10.1038/nmeth.3222. 1
- [RSH*17] RUEDEN C. T., SCHINDELIN J., HINER M. C., DEZONIA B. E., WALTER A. E., ARENA E. T., ELICEIRI K. W.: ImageJ2: ImageJ for the next generation of scientific image data. *BMC Bioinformatics* 18, 1 (2017). doi:10.1186/s12859-017-1934-z. 2
- [RWG*15] ROYER L. A., WEIGERT M., GÜNTHER U., MAGHELLI N., JUG F., SBALZARINI I. F., MYERS E. W.: ClearVolume: Open-source live 3D visualization for light-sheet microscopy. *Nature Methods* 12, 6 (2015). doi:10.1038/nmeth.3372. 4
- [SAF*12] SCHINDELIN J., ARGANDA-CARRERAS I., FRISE E., KAYNIG V., LONGAIR M., PIETZSCH T., PREIBISCH S., RUEDEN C., SAALFELD S., SCHMID B., TINEVEZ J.-Y., WHITE D. J., HARTENSTEIN V., ELICEIRI K., TOMANCAK P., CARDONA A.: Fiji: An open-source platform for biological-image analysis. *Nature Methods* 9, 7 (2012). doi:10.1038/nmeth.2019. 1, 2
- [SMZ*19] SHARIR A., MARANGONI P., ZILIONIS R., WAN M., WALD T., HU J. K., KAWAGUCHI K., CASTILLO-AZOFEIFA D., EPSTEIN L., HARRINGTON K., PAGELLA P., MITSIADIS T., SIEBEL C. W., KLEIN A. M., KLEIN O. D.: A large pool of actively cycling progenitors orchestrates self-renewal and injury repair of an ectodermal appendage. *Nature Cell Biology* 21, 9 (Sept. 2019), 1102–1112. doi:10.1038/s41556-019-0378-2. 5
- [SRE12] SCHNEIDER C. A., RASBAND W. S., ELICEIRI K. W.: NIH Image to ImageJ: 25 years of image analysis. *Nature Methods* 9, 7 (2012). doi:10.1038/nmeth.2089. 1
- [SSC*10] SCHMID B., SCHINDELIN J., CARDONA A., LONGAIR M., HEISENBERG M.: A high-level 3D visualization API for Java and ImageJ. *BMC Bioinformatics* 11, 1 (2010). doi:10.1186/1471-2105-11-274. 1
- [WJA*17] WALD I., JOHNSON G., AMSTUTZ J., BROWNLEE C., KNOLL A. M., JEFFERS J., GÜNTHER J., NAVRATIL P. A.: OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 931–940. doi:10.1109/TVCG.2016.2599041. 4