A Modular and Open-Source Framework for Virtual Reality Visualisation and Interaction in Bioimaging

A thesis submitted to attain the degree of

DOCTOR RERUM NATURALIUM

(Dr. rer. nat.)

presented by

Ulrik Günther

Diplom-Physiker, Technische Universität Dresden

born on 19th May, 1987 in Dresden, Germany

First reviewer:

Prof. Dr. sc. techn. Ivo F. Sbalzarini

Technische Universität Dresden & Max Planck Institute of Molecular Cell Biology and Genetics

Dresden, Germany

Second reviewer:

James Sharpe, PhD

European Molecular Biology Laboratory

Barcelona, Spain

Fachreferent:

Prof. Dr.-Ing. Raimund Dachselt

Technische Universität Dresden

Dresden, Germany

Submitted on 11th October, 2019

Defended on 18th May, 2020

Typeset using pandoc, Markdown, and LuaLaTeX in EB Garamond, Roboto Slab,
and Source Code Pro.

Abstract

Life science today involves computational analysis of a large amount and variety of data, such as volumetric data acquired by state-of-the-art microscopes, or mesh data from analysis of such data or simulations. The advent of new imaging technologies, such as lightsheet microscopy, has resulted in the users being confronted with an ever-growing amount of data, with even terabytes of imaging data created within a day. With the possibility of gentler and more high-performance imaging, the spatiotemporal complexity of the model systems or processes of interest is increasing as well. Visualisation is often the first step in making sense of this data, and a crucial part of building and debugging analysis pipelines. It is therefore important that visualisations can be quickly prototyped, as well as developed or embedded into full applications. In order to better judge spatiotemporal relationships, immersive hardware, such as Virtual or Augmented Reality (VR/AR) headsets and associated controllers are becoming invaluable tools.

In this work we present *scenery*, a modular and extensible visualisation framework for the Java VM that can handle mesh and large volumetric data, containing multiple views, timepoints, and color channels. scenery is free and open-source software, works on all major platforms, and uses the Vulkan or OpenGL rendering APIs. We introduce scenery's main features, and discuss its use with VR/AR hardware and in distributed rendering.

In addition to the visualisation framework, we present a series of case studies, where scenery can provide tangible benefit in developmental and systems biology: With *Bionic Tracking*, we demonstrate a new technique for tracking cells in 4D volumetric datasets via tracking eye gaze in a virtual reality headset, with the potential to speed up manual tracking tasks by an order of magnitude. We further introduce ideas to move *towards virtual reality-based laser ablation* and perform a user study in order to gain insight into performance, acceptance and issues when performing ablation tasks with virtual reality hardware in fast developing specimen. To tame the amount of data originating from state-of-the-art volumetric microscopes, we present ideas how to render the highly-efficient *Adaptive Particle Representation*, and finally, we present *sciview*, an ImageJ2/Fiji plugin making the features of scenery available to a wider audience.

Foreword and Acknowledgements

The journey to this thesis began when I was looking for more applicable fields of research after doing my Diploma in particle physics phenomenology. By sheer coincidence, I wandered into a talk by a newly-appointed professor, Ivo Sbalzarini. Ivo's enthusiasm was apparent, and so I soon applied to do my PhD in his group in the IMPRS CellDevoSys at the Max Planck Institute for Molecular Cell Biology and Genetics in Dresden. Starting to work on microscope design (together with the lab of Pavel Tomancak), and control software (together with Loïc Royer and Martin Weigert of the lab of Gene Myers), it soon became apparent that the hardware required to realise the ideas we had existed, but the software did not.

This is how the development of scenery – the framework developed and used in this thesis – started, motivated by the idea to be able to do laser ablation in virtual reality. It did not take long until more ideas popped up, brought on by meetings and collaborations with Kyle Harrington, Jan Huisken, Stephan Grill, and Raimund Dachselt and their labs: Wouldn't it be awesome to make the framework available to the ImageJ community as part of a plugin? Can we use scenery to render alternative data representations, such as the APR developed by Bevan Cheeseman? Could we even use eye tracking to help users perform cell tracking tasks without going through the annoying task of selecting corresponding cells by-slice? So now, this thesis is the culmination of four years of work on these topics, accompanied by a lot of highs (that I love to remember), and probably just as many lows (most of which I think I have already successfully repressed), and it would not have been possible without the support of many people:

First and foremost, a big thanks to Ivo, for providing outstanding support and freedom, and creating a lab full of friendly, curious, inspiring, and helpful people that make up an amazing team. Thank you, MOSAIC group, for making work fun! Particularly my office mates Bevan, and later Aryaman have not only made day-to-day business fun and interesting, but continue to be awesome collaborators! Thanks to all other MOSAICans, past and present, for great parties, hikes, amazing food (looking especially at you, Yaser!), and support.

I would like to thank Kyle Harrington, for being an amazing collaborator for many years now. Without Kyle, a lot of the things presented in this thesis would not have materialised, and many bugs would not have been fixed.

Thanks to my TAC members, Pavel Tomancak and Stephan Grill, for providing

great feedback, insights and especially support through the years. Thanks to James Sharpe, for agreeing to review this thesis.

Thanks to Ulrich von Zadow and Wolfgang Büschel, for introducing me to the intricacies of Human-Computer Interaction and to user tests, and Raimund Dachselt for agreeing to be my Fachreferent, for insightful discussions and feedback, often nudging me in directions I had never thought of before. Thanks to Stephan Gumhold, for interesting computer graphics discussions, and never letting go unless everything is absolutely clear.

Big thanks to the whole ImageJ community, and especially Curtis Rueden, the hero of the community. Loïc Royer and Martin Weigert, collaborating with you on ClearVolume was an amazing experience (I am actually still amazed about how smooth that went!), and the discussions with you provided a lot of inspiration and insight. Tobias Pietzsch, I very much enjoyed our collaboration on out-of-core volume rendering in scenery, and am very thankful for your critical and helpful remarks on software design.

Thanks to Bevan Cheeseman, Wolfgang Büschel, Martin Weigert, Aryaman Gupta, Kyle Harrington, and Tobias Pietzsch for proofreading this thesis and various help and suggestions. Thanks to all the participants of the user tests, and to the whole MPI-CBG community for being an awesome place to work. And of course to Katja, Alex and Corinna, for a constant supply of high-quality, caffeinated beverages.

Thanks to Holger Steinfurth, for helping me pave the way that ultimately led to this thesis.

Most importantly, thanks to my family and friends, who provided a constant source of support through thesis time. Special thanks to my parents, for their continuous support and always being there for me, and for proofreading and checking my thesis for printing errors. Finally, thanks to this one crazy software bug in my Diploma thesis which took months to find, and seconds to fix, and led me to meet Steffi – thank you for your help, motivation, support, keen eye on design, amazing cooking, and for your love. And of course for taking my mind off work with hiking, climbing, and vertical gardening in Saxon Switzerland!

Contents

Overview and Contributions

Current Trends and Challenges in Biology

In the last three decades, biology has gone through a remarkable development from being a discipline that is mostly wetlab-based to one utilising the tools and methods of mathematics, physics, and computer science, becoming more and more reliant and intertwined with them:

To just highlight a few, developments on the experimental side like light-sheet fluorescence microscopy [Huisken, 2004a], optogenetics [Boyden et al., 2005, Li et al., 2005], or Cryo-EM [Adrian et al., 1984] have opened new venues for investigation, while theoretical contributions, e.g. to active matter theory [Mietke et al., 2018] now shed more light on these results and provide analytical guidance.

Investigating biological systems in two dimensions has already led to fascinating and important results in the past (Drosophila embryonic genes, etc.). Building on these, and utilising the mathematical, physical, and computational techniques mentioned above, we are now able to investigate biological systems in three dimensions, and over time, and at high speed, enabling in-depth observation and reasoning about spatiotemporal processes.

Even more recent developments, like CRISPR/Cas9 [Jinek et al., 2012] or gene drives enable us to manipulate specimen in ways and on fast timescales thought impossible before.

What these developments lack to a certain extent, are ways to again bring the experimenter into the loop, both during the experiment, and during analysis, to enable new and flexible ways of interacting the large and complex amounts of data state-of-the-art experiments create, and also with the scientific instruments producing this data.

Virtual and Augmented Reality

With the advent of the first small-enough computers, and small-enough cathode-ray tubes (CRTs), the development of devices that give the user the ability to inhabit a virtual environment or use more than just a keyboard for input of data started. Early examples of such systems — from the 1960s — are The Sword of Damocles[Sutherland, 1968] or the Sketchpad system[Sutherland, 1963], made famous as *The Mother*

of All Demos. In the meantime, and over the course of two Virtual Reality Revolutions — with important developments like the CAVE (CAVE Automatic Virtual Environment, [Cruz-Neira et al., 1992]) — head-mounted displays (HMDs) have now become a commodity and can be bought for about 400$, bringing a once highly specialised and expensive device into the homes or offices of potential users.

Complimentary to these display devices, new input methods are also on the rise, such as free-air gesture input (e.g. the widely available Microsoft Kinect or the Leap Motion), free-air controller input (e.g., HTC Vive controllers), touch input (multitouch screens in nearly every contemporary mobile phone) or even devices controlled by the gaze of the user (such as eye trackers from Tobii or Pupil Labs). Such devices and their interaction modalities are commonly called Natural User Interfaces.

Despite the common availability of VR display devices, or NUI input device, many of the analysis and visualisation tasks in bioimaging are still done on a 2D screen, using a keyboard and a mouse, while VR might actually provide tangible benefits, beyond just a quick "wow" effect.

Scope of this thesis and contributions

In this thesis, we aim to demonstrate that the inclusion or Virtual Reality and associated input devices, and advanced realtime rendering techniques can enhance the biologist's workflow, and enable new kinds of experiments, *in vivo* and *in silico*.

To achieve this, we develop an open-source realtime rendering and interaction framework named *scenery* that enables rapid prototyping of visualisations of geometric and volumetric biological data, and interaction with such on the basis of Natural User Interfaces. The framework supports rendering on regular desktop screens, virtual reality headsets (like the Oculus Rift or HTC Vive), and augmented reality headsets (like the Microsoft HoloLens).

We will detail the architecture of the framework and demonstrate its necessity, utility and comprehensiveness on a set of case studies, and show further contributions made possible by the use of the framework.

Specifically, we will detail the following contributions:

- *scenery*, a framework for creating visualisation and interaction interfaces with both volumetric and geometric data, supporting virtual and augmented reality, and clustered rendering.
- *Bionic Tracking*, an algorithm for utilising the user's gaze to solve tracking problems involving moving particles and objects or tracing of neurons, implemented on top of *scenery*.
- *Towards interactive laser ablation*, where laser-based complex microsurgical procedures on microscopic specimens are enhanced and simplified by the use of virtual reality and natural user interfaces. A simulation of this workflow is also implemented on top of *scenery* and a user study performed to show benefits and challenges, as well as identify issues.

- *Rendering the Adaptive Particle Representation*, where we introduce ideas how to render the highly-efficient, particle-based Adaptive Particle Representation (APR) [Cheeseman et al., 2018] of volumetric data. The APR can be displayed as point-based graphics, as maximum intensity projection (MIP), or full volume rendering. All rendering algorithms are implemented on top of *scenery*.
- *sciview*, a plugin for the ImageJ2/Fiji ecosystem, make scenery's flexible visualisation solutions available to the end user.

Publications

Some of the results presented in this thesis have already been published :

Peer-reviewed Papers

- **Günther, U.**, Harrington, K.I.S.: Tales from the Trenches – Developing sciview, a new 3D viewer for the ImageJ community. *VisGap workshop at Eurovis 2020*, Norrköping, Sweden. arXiv preprint 2004.11897, DOI 10.2312/visgap20201112.
- **Günther, U.**, Pietzsch, T., Gupta, A., Harrington, K.I.S., Tomancak, P., Gumhold, S., and Sbalzarini, I.F.: scenery: Flexible Virtual Reality Visualization on the Java VM. *IEEE VIS* 2019, Vancouver, Canada. arXiv preprint 1906.06726, DOI 10.1109/VISUAL.2019.8933605.
- Daetwyler S., **Günther, U.** , Modes, Carl D., Harrington, K.I.S., and Huisken, J.: Multi-sample SPIM image acquisition, processing and analysis of vascular growth in zebrafish. *Development*, 2019. bioRxiv preprint 478149, DOI 10.1242/dev.173757.
- Cheeseman, B.L., **Günther, U.**, Susik, M., Gonciarz, K., and Sbalzarini, I.F.: Adaptive Particle Representation of Fluorescence Microscopy Images. *Nature Communications*, 9(5160), 2019. bioRxiv preprint 263061, DOI 10.1038/s41467-018-07390-9
- Royer, L.A., Weigert, M., **Günther, U.**, Maghelli, N., Jug, F., Sbalzarini, I.F. and Myers, E.W.: ClearVolume: open-source live 3D visualization for light-sheet microscopy. *Nature Methods*, 2015. DOI 10.1038/nmeth.3372.

Submitted Papers

- **Günther, U.**, Harrington, K.I.S., Dachselt, Raimund, Sbalzarini, I.F.: Bionic Tracking: Using Eye Tracking to Track Biological Cells in Virtual Reality. *Submitted to BioImageComputing at ECCV 2020*. arXiv preprint 2005.00387.
- Arshadi, C., Eddison, M., **Günther, U.**, Harrington, K.I.S., Ferreira, T.A.: SNT: A Unifying Toolbox for Quantification of Neuronal Anatomy. *Submitted to Nature Methods*. bioRxiv preprint 2020.07.13.179325.

Conference Abstracts

- Gupta, A., **Günther, U.**, Incardona, P., Aydin, A.D., Dachselt, R., Gumhold, S., Sbalzarini, I.F.: A Framework for Interactive Virtual Reality *In Situ* Visualisation of Parallel Numerical Simulations. *The 9th IEEE Symposium on Large Data*

Analysis and Visualization at IEEE VIS, 2019. arXiv preprint 1909.02986, DOI 10.1109/LDAV48142.2019.8944368.

- **Günther, U.**, Pietzsch, T., Rueden, C., Daetwyler, S., Huisken, J., Elicieri, K., Tomancak, P., Sbalzarini, I.F., Harrington, K.I.S.: sciview - Next-generation 3D visualisation for ImageJ & Fiji, *From Images to Knowledge with ImageJ and Friends*, EMBL Heidelberg, 2018
- **Günther, U.**, Harrington, K.I.S., Sbalzarini, I.F.: Exploring the scenery of light-sheet microscopy with virtual reality, *LSFM2018*, Dresden, 2018.
- Royer, L.A., Weigert, M., **Günther, U.**, Maghelli, N., Jug, F., Sbalzarini, I.F. and Myers, E.W.: ClearVolume - from microscope to visualisation in seconds, *VizBi*, EMBL Heidelberg, 2016.
- Royer, L.A., Weigert, M., **Günther, U.**, Maghelli, N., Jug, F., Sbalzarini, I.F. and Myers, E.W.: ClearVolume - open-source 4D live visualisation for light-sheet microscopy. *Focus on Microscopy*, Göttingen, 2015.
- **Günther, U.**, Cheeseman, B.L., Tomancak, P., Sbalzarini, I.F.: dive into data — immersive 3D particle visualisation, *BioImageInformatics*, Leuven, 2014.

Papers in Preparation

The following papers containing material from this work are currently under preparation:

- **Günther, U.**, Pietzsch, T., Rueden, C., Daetwyler, S., Huisken, J., Elicieri, K., Tomancak, P., Sbalzarini, I.F., Harrington, K.I.S.: sciview - Next-generation 3D visualisation for ImageJ & Fiji.
- **Günther, U.**, Simson, J., Sbalzarini, I.F., Harrington, K.I.S.: Linear Genetic Programming for robust pupil detection in interactive, virtual reality eye tracking applications.

Supervision

The following students have been supervised by the author in the duration of the thesis:

- Sahil Loomba, intern, May - August 2014.
- Aryaman Gupta, intern and master student, June - December 2017.
- Luke J. Hyman, intern, August - September 2018.

What follows

In the following, first part of the thesis we are going to introduce the basic concepts and technologies as the *Preliminaries* that have ultimately led to the challenges addressed in this thesis. We start with *fluorescence microscopy*, describing the developments from first light microscopes to modern lightsheet volumetric microscopes, followed by a chapter about *visual processing*, detailing the detection and processing of visual information in the human nervous system. The visual processing chapter is followed by the *XR* chapter, describing the historic and current developments in virtual and

augmented reality, together termed as *cross reality*, or *XR*. The final chapter of the preliminaries part introduces *Eye Tracking and Gaze-based interaction*, going into detail how a mode of perception can also be used for control purposes. At the end of each of *preliminaries* chapters, we state specific challenges that will be addressed in the *Case Studies* in Part Three of this thesis. In Part Two, we describe our visualisation framework *scenery* in detail, which is the enabling technology for the case studies described in Part Three. Finally, in Part Four, we are going to conclude our findings and provide an outlook to future work.

Part I:

Introduction

> In theory, there is no difference between
> theory and practice
>
> —Yogi Berra

Chapter 1:

Fluorescence Microscopy

Fluorescence microscopy is one of the major techniques used in cell, developmental and systems biology. In the most basic version, a fluorescent molecule is introduced into the biological specimen, staining it. The early fluorescent markers used (e.g. FITC, Hoechst), are however highly cytotoxic and incompatible with life, and can therefore only be used for fixed specimen. Newer developments led the the introduction of fluorescent proteins into to organism of interest via genetic engineering. These fluorescent proteins are biocompatible and can therefore be used to study processes in a living organism.

A fluorescent protein emits photons of wavelength $\lambda_{\text{illumination}}$ after being illuminated with a shorter excitation wavelength $\lambda_{\text{detection}}$. One of the most popular fluorescent proteins is GFP, or green fluorescent protein, was originally isolated from the jellyfish *Aequorea victoria* [Heim et al., 1995]. GFP has a *quantum yield* of 0.79 photons per excitation photon. The emission of photons in the fluorescent protein itself originates from an active *chromophore*, usually located in the center of the protein (see Figure 1.1 for a cartoon of the structure). The chromophore can emit a certain number of photons after excitation before stopping (photobleaching). This leads to the problem that in each microscopy application, one has to take into account the available photon budget, resulting from the interplay of excitation intensity and quantum yield.

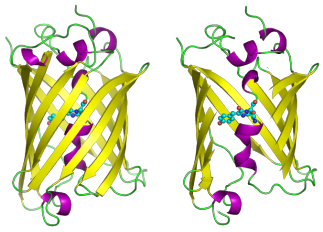

Figure 1.1: The green fluorescent protein GFP, with the beta barrel cut away on the right sight, revealing the chromophore. Image courtesy of Raymond Keller, Public Domain.

In *widefield fluorescence microscopy*, the full specimen is illuminated at once with the excitation wavelength, leading to a single 2D image of the specimen. The resolution of a microscope is defined via its *point spread function* (PSF), which is not directly observable, but measurable as its square $|h(x,y,z)|^2$. The intensity $I(x,y,z)$ of an object $O(x,y,z)$ in the image plane is then its convolution with the PSF,

$$I(x,y,z) = O(x,y,z) \otimes |h(x,y,z)|^2. \tag{1.1}$$

In the widefield microscope, illumination and detection are done through the same optical path. Were this not the case, the total PSF would be a product of both the illumination and detection PSFs,

$$|h_{\text{full}}|^2 = |h_{\text{i}}|^2 \cdot |h_{\text{d}}|^2. \tag{1.2}$$

For more complex specimen, and to study three-dimensional structures, more advanced techniques are used, which we will introduce in the following.

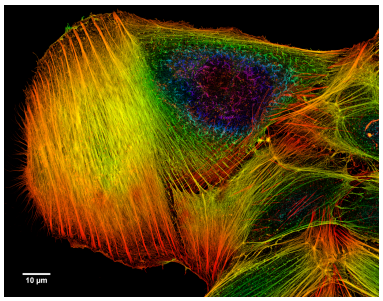

Figure 1.2: Z projection of a phalloidin-labeled osteosarcoma cancer cell, making actin filaments visible. Image taken on a Zeiss LSM780 confocal microscope. Image (cc) by Howard Vidin, Wikimedia Commons

1.1 Confocal Microscopy

The confocal microscope was developed in the 1960s by Marvin Minsky, one of the pioneers of Artificial Intelligence (AI), in order to investigate neural connections in the central nervous system of mammals, and draw conclusions for AI from that. The confocal microscope enables optical sectioning by illuminating the sample sample with a coherent light source, nowadays lasers, and rejecting the emitted fluorescence with a pinhole. With this principle, it is possible to scan the specimen point-wise in X and Y directions, and provide optical sectioning by also moving it in the Z direction. Occurring fluorescence that was not rejected as background by the pinhole gets collected point-wise by a photomultiplier, and an image or volume reconstructed from that. The principle of operation is also pictured in 1.3. An example of an image taken with a confocal microscope is shown in Figure 1.1

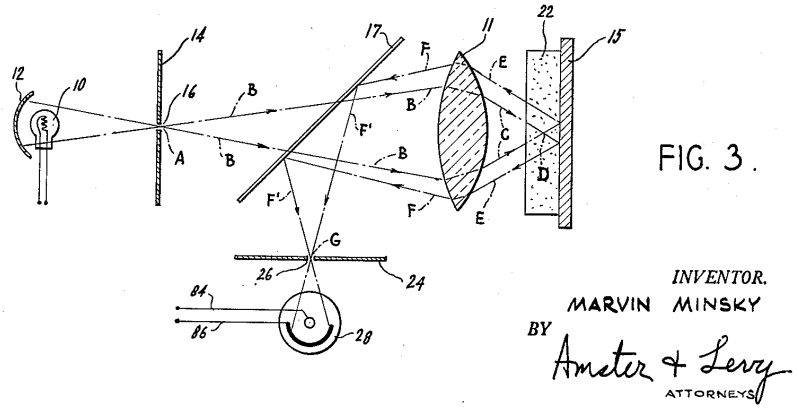

Figure 1.3: Confocal microscope operating principle, 10: Arc lamp (laser, nowadays), 12: Illumination pinhole, 16: Dichroic mirror, 22: specimen, 26: Pinhole, 28: Photomultiplier diode (Public Domain, from Marvin Minksky's original patent application).

1.1.1 Image formation

The confocal microscope uses a single lens for illumination and detection, achieves focused illumination through beam scanning, and detects photons from fluorescence via its pinhole. As the illumination and detection wavelengths differ, the confocal's PSF is given as the product of the illumination and the detection PSF. However, when illumination and detection wavelengths nearly coincide, e.g. for GFP $\lambda_{\text{illumination}} =$ 488nm and $\lambda_{\text{detection}} =$ 520nm, the total PSF is approximately equal to the square of the PSF of the widefield microscope, resulting in a $\approx 1/\sqrt{2}$ better resolution.

1.1.2 Shortcomings

Though a very useful instrument with a higher resolution than a widefield microscope, the confocal microscope has a few issues, namely:

- A lot of the photon yield is discarded, due to the pinhole. For fast acquisitions, this can lead to images with a very low signal-to-noise ratio.
- Due to scanning the laser over each point to be imaged, the specimen is exposed to high levels of light. In addition, the scanning process can be quite slow, up to 1 or 2 seconds per image. Spinning Disk Confocal Microscopes alleviate this issue to some extent, leading to much higher acquisition speeds by multiplexing the illumination process.

In the next section, we will explain lightsheet microscopy, which aims to alleviate these shortcomings.

1.2 Lightsheet Microscopy

In lightsheet, or similarly, selective plane illumination microscopy [Huisken, 2004a][1], the specimen is illuminated by a coherent light source in a 90º angle to the detection plane. The illumination light is focused into a thin sheet of light by means of a cylindrical lens (selective plane illumination microscopy, SPIM), or by scanning a Gaussian laser beam (digitally scanned lightsheet microscope, DSLM). This means that a full-frame 2D image of the specimen can be acquired at once, without point scanning, lowering the required light intensity by a substantial amount. Further, many 2D acquisitions can happen sequentially, enabling the capture of fast biological processes in 3D and 4D, such as the beating of the zebrafish *Danio rerio*'s heart [Mickoleit et al., 2014].

[1] Throughout the remainder of this work, for the sake of brevity, we are going to refer to both lightsheet microscopy and SPIM microscopy simply as "lightsheet microscopy".

1.2.1 Image formation

In the lightsheet microscope, illumination and detection are again separate, leading to separate PSFs for illumination and detection. The microscope's lightsheet thickness should be optimised for the detection optics, choosing the numerical aperture of the illumination such that the lightsheet has a uniform thickness across the entire field of view [Huisken, 2004b]. However the lightsheet can usually not be made completely uniform, and will be thinnest at the focus. One possible compromise is to choose the NA so that the lightsheet is twice as thick at the edges of the field-of-view as it is in the middle. The PSF of the SPIM can finally be approximated with Gaussians, yielding [Huisken, 2004b]:

$$|h_{\mathrm{SPIM}}(x,y,z)|^2 \propto \exp\left(-\frac{2x}{w_{\mathrm{lat}}^2}-\frac{2y}{w_{\mathrm{lat}}^2}-\frac{2z}{w_{\mathrm{axial}}^2}\right) \tag{1.3}$$

, where w_{lat} is the lightsheet thickness.

The lightsheet microscope also has the benefit that samples can be mounted in a movable way, and imaged from multiple directions, finally fusing the best parts of the

image together for optimal image quality [Preibisch et al., 2014]. Furthermore, the sample can also be illuminated from multiple directions [Weber and Huisken, 2012]. Many realisations of lightsheet microscopes enable the user to move the sample in X, Y, and Z directions, and additionally rotate it at high speeds.

1.2.2 Data rates

Confocal microscopes equipped with EM-CCD cameras produce an image data rate of about 1 MB/s. Lightsheet microscopes however, play in a different league, as they are usually equipped with state-of-the-art sCMOS cameras (which offer about 60% quantum efficiency, a low readout noise, and high readout speeds of about 100fps at full frame size, which is usually 4 to 6 megapixels). These cameras, running at full speed, can easily produce data rates of $1GB/s$ [Reynaud et al., 2014], filling up a 500GiB SSD drive in less than 10 minutes, and amounting to nearly 90TiB of data *per day*, if running at full speed. For a visual comparison, have a look at Figure 1.4.

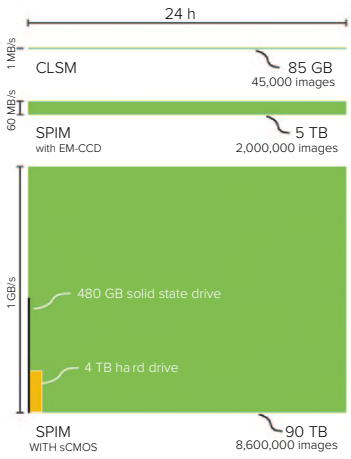

Figure 1.4: Comparison of the data produced by different microscope types within 24 hours. Adapted from [Reynaud et al., 2014].

This deluge of data now poses a large problem both for the scientists using the lightsheet microscopes and producing that data, and also for the support staff that has to take care of data storage, compute clusters, and so on. This has led to approaches where microscopy data is acquired and processed in an interleaved way, with e.g. 10 minutes of data acquisition followed by 10 minutes of processing, such as in the case of imaging the zebrafish heart [Mickoleit et al., 2014].

Furthermore, effective processing of long developmental timelapses, the parade discipline of lightsheet microscopes, is not possible without a cluster.

The high data rate combined with the high spatiotemporal quality of the data leads to interesting challenges regarding data storage and processing, and instrument interaction for current and future lightsheet microscopes:

1.3 Challenges and Opportunities

1.3.1 Taming the data

Data compression alone is not going to solve the data deluge issue posed by lightsheet microscopy: While the compression step requires time, but can nowadays, utilising efficient algorithms and multi-core processors or graphics cards, be made very quick, it also requires a decompression step, again taking time and restoring the data to it's original, unwieldy size. While successful efforts have already been made to democratise the use of lightsheet microscopes [Jahr et al., 2016, Pitrone et al., 2013, Gualda et al., 2013], the expensive data processing requiring the use of clusters hinders users from effectively deploying one or multiple lightsheet microscopes.

To really tame the data, one has to think of an alternative data representation, that could have a compression step, but without necessary decompression — instead, processing should happen on the alternative data representation.

Such a representation has been developed [Cheeseman et al., 2018], named the *Adaptive Particle Representation* (APR). The APR non-uniformly resamples images

and stores information only where it actually is, saving a lot of storage space, especially for sparsely populated images, such as from fluorescence microscopes.

In Chapter 13, *Rendering the Adaptive Particle Representation*, we are going to talk about how to use this representation for fast and efficient rendering of lightsheet microscopy data.

1.3.2 Smart microscopy requires smart interactions

Scherf and Huisken [Scherf and Huisken, 2015] have made the case in 2015 for smart and gentle microscopes, that not only know how to image a specimen, but also take great care in not disturbing the normal development of it, treating it as gently as possible, by means of adaptive laser power, imaging times and windows, and dynamic determination of regions of interest.

Royer has developed a microscope for long-term *Drosophila* imaging [Royer et al., 2016] that constantly measures image sharpness and embryo drift, and optimises the microscope's optical components in-between each stack of a timelapse for optimal image quality, fit for high-quality tracking and lineage tree creation for the whole time of embryo development.

Guignard, et al., have developed a microscope [Guignard et al., 2017] that combines adaptive lightsheet imaging with single-cell transcriptomics, yielding simultaneous insight into both gene expression and the spatiotemporal consequences of it.

All these microscopes, envisioned, and already existing, have in common that they require a very low to no level of human intervention during the imaging session, therefore allowing developmental imaging with unprecented precision, but no options to interfere with the specimen interactively, may it be via optogenetic manipulation, or laser microsurgery. Such tools are however invaluable for the biophysical investigation of tissue mechanics, and the combination of smart microscopy with smart, natural, and intuitive interaction techniques in 3D can open the door for new experiments leading to even deeper insight into both developmental and biophysical processes, such as *Drosophila* wingdisc formation, or retinal development [Matejčić et al., 2018]:

- In the case of *Drosophila* wingdisc formations, investigations of tissue tensions and mechanics have so far been focused only on flat pieces of tissue, which do not constitute the main part of development, and are actually hard to find in the developing embryo. 3D interaction in that scenario can provide the user with easy access to more complex geometries to perform ablation experiments in.
- In the case of retinal development, which takes place on highly curved surfaces and in complex volumes, additional 3D interactions for ablation and optogenetics can lead to more insight into defects in retinal development, of which human medicine might ultimately benefit.

For these use cases, we are going into deeper detail in Chapter 12, *Towards Interactive Virtual Reality Laser Ablation*, developing an interactive demo of how such interactions might take place in the future, on a microscope, equipped with 3D

virtual reality glasses, or from a room-scale virtual reality system. Additionally, in Chapter 11, *Bionic Tracking: Using Eye Tracking for Cell Tracking*, we discuss a new way to approach tracking and tracing problems on images resulting from fluorescence microscopy by utilising eye tracking.

Chapter 2:

Introduction to Visual Processing

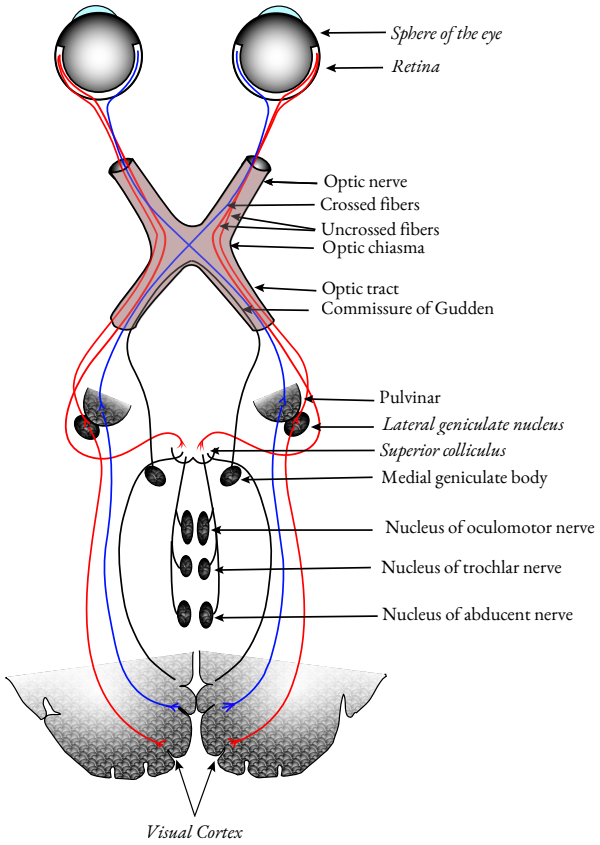

Figure 2.1: Schematic overview of the paths from the eye to the visual cortex, with the parts discussed in this chapter highlighted in italics. Adapted from *Anatomy of the Human Body* [Gray and Lewis, 1878], Public Domain.

In this chapter, we introduce the human visual system, the anatomy of the human eye and its physical capabilities and movements, as well as the processing happening to incoming photons in the retina and further downstream, in the central nervous system.

The goal of this chapter is to give the reader an understanding of the physiological processes that ultimately enable both eye-based natural user interfaces and cross reality applications, and introduce implications from physiology for such. In the end of the chapter, we will introduce some challenges and questions in the context of visual processing which will be addressed in later chapters of this thesis.

2.1 A Short Tour of the Human Visual System

The processing of visual stimuli happens in multiple stages: In Figure 2.1 we show an overview sketch of the nerve pathways involved in visual processing. We will discuss the following parts in deeper detail:

1. *The Optical Path* — Collection and accumulation of incoming photons by the optical system consisting of the *cornea*, *iris*, and *lens* onto the *retina*, and especially the most sensitive part of the retina, the *fovea*.
2. *The Retina* — Collection and translation of incoming photons into nerve pulses, and compression of the nerve signals for further processing,
3. *The Lateral Geniculate Nucleus* (LGN) — situated in the *thalamus* part of the forebrain, which serves as a relay for the information coming directly from the retina (interestingly, the left LGN processes information from the right eye, and vice versa — a pattern common in the human brain), and
4. *The Primary Visual Cortex* — Final processing of the signals in the *primary visual cortex* of the occipital lobe on the back of the brain.

2.2 Optical Path

Light enters the anterior chamber of the eye, travelling through the iris, then traversing the the vitreous humour (a gelatineous substance filling the interior of the eye) to the retina.

Evolution has optimised the refractive index of the human cornea to yield an optimal air/cornea boundary, rendering the final image sharp on the retina. The lens, held in place by the ciliary body, and the suspensory ligaments, focuses the incident light onto the retina, and especially on the most sensitive part of the retina, the fovea. The fovea is about 1.5 mm in diameter and contains the most photoreceptor-dense region — 300000/mm^2 compared to ≈ 100000/mm^2 in the periphery [Duchowski, 2017, Snowden et al., 2011].

Foveal, or central vision only makes up about 5º of the field of vision. In the most central part of the fovea, the foveola, about 133 cones per degree of visual angle lead to a resolvable frequency of 66 cycles/º, while at the fovea, the frequency already drop by about half, to 35 cycles/º [Duchowski, 2017]. In Figure 2.2, we show a scheme of the different ranges of vision in humans, with the region below 30º being the *field of useful vision*. The movements of the eye, described in Section 2.2.1, *Eye movements*, are able to make up for the small field of useful vision by constantly scanning a scene.

Apart from movements, the eye is also able to adapt itself internally to different viewing conditions. This adaption to visible objects happens in two ways:

- the iris size can be modulated, changing the amount of light reaching the retina by a up to a factor of 16. This contraction and expansion is not only due to light stimuli, but can also be triggered by drugs or hormonal changes, e.g. due to excitement, and

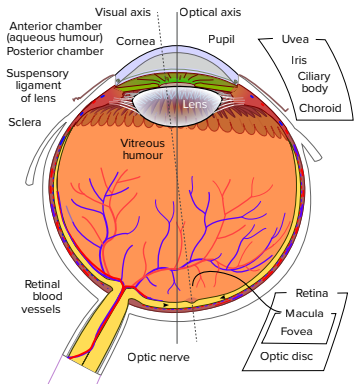

Figure 2.2: Anatomy of the human eye — Image (cc) by Rhcastilhos and Jmarchn, Wikimedia Commons.

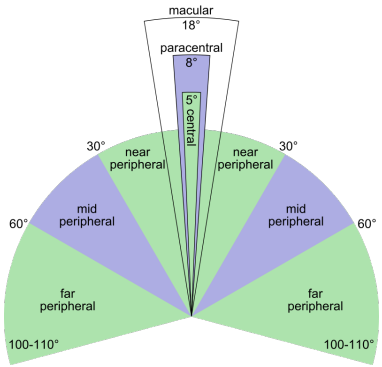

Figure 2.3: Ranges for peripheral and central vision in humans. Central or foveal vision offers the highest acuity. Image (cc) by Zyxwv99, Wikimedia Commons.

- ciliary muscles can modulate the lens thickness: when they relax, leading to tense zonules, the vision is adapted for distance, when they contract, the zonules get more slack, and the vision is adapted for closer objects — these are called vergence movements.

Vergence however is only one kind of the movements the eye can perform, so let's get into more detail about the other forms of movement.

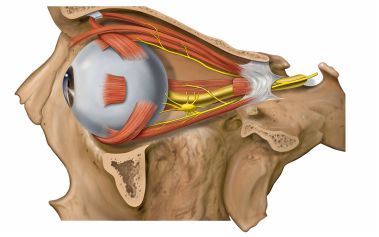

Figure 2.4: Muscles of the human eye. Image (cc) by Patrick Lynch, Wikimedia Commons.

2.2.1 Eye movements

Following the classifications from [Snowden et al., 2011] and [Duchowski, 2017], eye movements fall into one of five categories:

- *Saccades* are quick and jumpy movements of the eye to reposition the fovea to a new area of interest. As such, they can be voluntary or involuntary. In natural environments saccades occur with high speeds, and several times per second. Their peak angular velocity can exceed 900°/s, take approximately 200 ms to initiate, and then last for about 10 – 200 ms. Saccades are *stereotypical* and *ballistic*: Stereotypical movement means they always follow the same pattern of fast initial acceleration after an initial processing delay of about 200 ms, followed by movement with maximum velocity, and concluded by a rapid slowdown as the eye reaches the target area (see Figure 2.2.1 for example time series). Ballistic movement means they are planned and, once initiated, cannot be stopped. During execution of the movement there is no visual perception, rendering the subject temporarilly blind. This effect is called *saccadic suppression*[1] [Snowden et al., 2011].
- *Smooth pursuits* occur when a subject is tracking a moving stimulus, where the eye's angular velocity is matched to the movement of the image of the stimulus on the retina. These are the only smooth movements the eyes perform. While they are voluntary, they cannot be initiated without a moving stimulus [Cullen, 2016].
- *Fixations*, *tremors*, and *jitters* occurs when a subject focuses on a particular object of interest. Counterintuitively, these movements do not completely fix the image on the retina, but jitter around it within about 5° of visual angle. If they would not do that, the image would disappear within seconds, due to adaption of the receptors to overstimulation. This suggests that a different system is involved with fixations than with saccades or pursuits. Fixations last for about 150 – 600 ms and humans spend over 90% of viewing time with this kind of eye movement. It has also been found that miniature movements enhance the perception of high-frequency detail in a stimulus [Rucci et al., 2007].
- *Vergence* movements occur when a subject is moving its attention between near and far objects, with the eyes then moving in opposite directions.
- The *Optokinetic reflex* (OKR) is a compensatory reflex that stabilises moving objects on the retina, and moves the eyes back to the original position, in case the object moves outside the field of view (for example, when looking out of a train window).
- The *Vestibo-ocular reflex* (VOR) is another compensatory reflex that induces compensating eye movements during rotational and translational head movements, in order to stabilise the image on the retina. Often, the OKR and VOR work in

[1] The most visible effect of saccadic suppression in humans is the lack of motion blur during saccadic eye movements, opposed to e.g. fast head movements.

concert [Cullen, 2016].

In the context of 2-dimensional localisation of gaze directions, fixations, saccades, and smooth pursuits are the most important eye movements. Vergence movements in turn can be used for 3-dimensional gaze estimation, e.g. by detecting independent gaze directions for each eye, and finding the intersection [Mlot et al., 2016]. Future developments in display technology, where focus points can be modulated [Huang et al., 2015, Jang et al., 2017, Sun et al., 2017] will probably make that even more interesting.

All of the described movements are optimisations to provide the best image possible, using "the world's worst camera" [Duchowski, 2017]. We continue our discussion with the retina, the translator of photons to neural impulses.

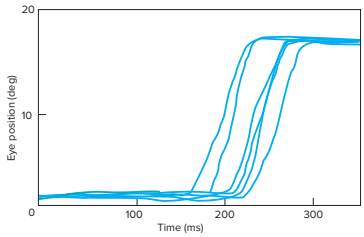

Figure 2.5: Example time series of saccadic eye movements: The movement starts after an initial processing delay of about 150 ms, followed by fast movement for about 50 – 100 ms. Image reproduced from [Snowden et al., 2011].

2.3 The Retina — Retinal Architecture and Processing

At the retina, the processing of incident photons starts in the true sense of the word, as so far we have only been concerned with transmission, modulation, and focussing.

The retina of mammals has a somewhat odd architecture, seen in Figure 2.3: The light is entering from the bottom of the image, so the light has to travel through a dense forest of neurons before reaching the photoactive rods and cones. This kind of architecture is called *inverted retina architecture*. What the true benefits of an inverted architecture are remains a matter of debate.

There are good reasons for the inverted architecture, such as easier supply of blood to the back side of the retina, rather than the front, which is very much needed by the (in terms of chemical energy) power-hungry photoreceptor cells. The neural tissue of the eyes has also been shown to act as waveguide for incoming photons, probably a mechanism to counter photon scattering through it [Franze et al., 2007]. One tradeoff is the existence of the blind spot where the optic nerve exits the eye, mended in most cases by the presence of two eyes, the Section 2.2.1, *Eye movements* described before, and the upstream neural processing.

After traversing this neuronal maze, photons reach the true actors of photon reception, the *rods* and *cones*.

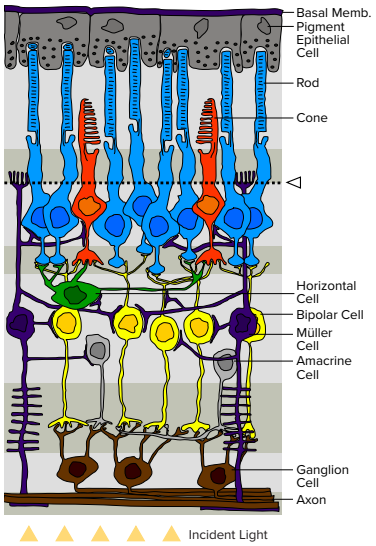

Figure 2.6: Inverted retinal architecture of mammals. Adapted from original illustration, (cc) by Marc Gabriel Schmid, Wikimedia Commons.

2.3.1 Rods and cones

These are the workhorses of the retina, responding in different lighting intensity conditions: While rods are highly sensitive in dim conditions, the *scotopic* regime, even responding to single-photon stimuli (*rhodopsin* is responsible for the actual reception in rods, and absorbs green light most strongly), cones respond more sensitively in high-intensity conditions, the *mesopic* regime. While cones exist in long-wavelength, middle-wavelength, and short-wavelength flavours, often called red, green, and blue, rods only exist in a single flavour.

Coming back to the distribution of photoreceptors among the retina, both types also follow different patterns: While most — 150000/mm^2 — of the rods exist around

12° – 15° of visual angle, the density of cones peaks at the fovea at 0° of visual angle, also with about 150000/mm^2. The cone density falls off sharply outside the fovea, reaching a density as low as ≈ 15000/mm^2 at 15°.

There are no rods at the fovea, and their falloff is not as sharp, slowly waning to about 50000/mm^2 in the periphery at 80°[Snowden et al., 2011]. See Figure 2.7 for a graph of the distribution.

The perceptual consequences of this distribution are interesting: While the sensitivity to color changes in the periphery sinks quite drastically due to the reduced number of cones, contrast sensitivity due to rods is still quite high.

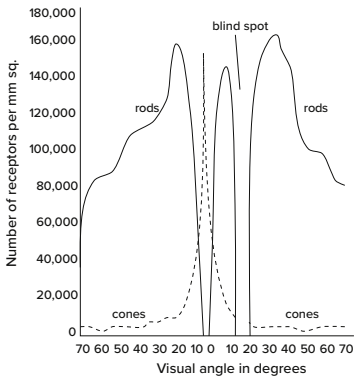

Figure 2.7: The distribution of rods and cones in dependence of the visual angle. While the distribution of cones sharply peaks around the fovea, the distribution of rods falls off slower in the periphery, and rods do not exist entirely at the fovea. Adapted from [Duchowski, 2017].

2.3.2 Retinal ganglion cells

Retinal ganglion cells are responsible for wiring the photoreceptors of the retina to the *lateral geniculate nucleus* (LGN) in the thalamus, and are actually dendrites of the optic nerve. What they are doing can already be described as image processing: They are wired to the photoreceptors in a layout that is essentially circular, with the area of responsibility called a *receptive field*.

A receptive field contains about 100 photoreceptors and consists of an inner and an outer ring that act in competition with each other: starting from their baseline neuronal activity, ON-center cells fire more when the center is stimulated, and the outside is not, while OFF-center cells fire more when the center remains unstimulated, but the outside is stimulated (*center-surround decorrelation*).

Instead of acting like the pixels of a camera sensor, this behaviour makes ganglion cells basically edge detectors, transmitting mostly the edge information downstream, which results in a large reduction in the amount of data that needs to be transmitted. Let's do an example calculation of the effect this has:

Retinal ganglion cells receive input from about about 128000000 cells — about 120000000 rods and 8000000 cones — assumed to carry, for simplicity, or 8bit of data, equivalent to 256 shades of gray. Assuming a "refresh rate" of 30 Hz, this amounts to ≈ 4 GiB/s(!). The ganglion cells however only have about 1000000 outputs connecting to the next processing area, the LGN, reducing the necessary data rate to the LGN to about 30 MiB/s [Brenner et al., 2000, Koch et al., 2006]. Would the optic nerve carry through all the neural connections from the rods and cones, it would not have an average 3.5 mm diameter, but about 20 mm, severely restricting the possible movements of the eye.

2.4 The Lateral Geniculate Nucleus

First, the axons from both eyes cross over at a point called the *optic chiasm*. There, the axons from the nasal side of each retina cross to the other side of the brain, while the axons from the temporal side do not cross. The part of the nerves between the optic chiasm and the LGN is called the *optic tract*.

The LGN itself has a 6-layer, staggered architecture: Projections from the same

side (*ipsilateral*) end up in layers 2, 3, and 5, while projections from the opposite side (*contralateral*) end up in 1, 4, and 6 — and projections from the same visual area connect to the same place in all the layers. In the LGN, three types of cells are found, that in some cases refine the receptive field structure of the retinal ganglion cells:

- *magnocellular* (M) cells, large, fast-responding cells connected to rods, to be found in the first two layers,
- *parvocellular* (P) cells, small, slow-responding cells, connected to the cones, and found in layers 3-6,
- *koniocellular* (K) cells, consisting of very small, heterogeneous cells, connected to only blue cones, to be found *between* the M and P layers.

The M and P cells have complimentary functional characteristics, with certain similarities to the photoreceptors they connect from (from [Kiley and Usrey, 2016] and [Duchowski, 2017]):

Characteristic	Magnocellular cells	Parvocellular cells	Koniocellular cells
Cell body size	Large	Small	Small
Receptive field size	Large	Small	*unknown*
Transmission time	Fast	Slow	*unknown*
Receptive fields	Large	Small	*unknown*
Sensitivity to small objects	Poor	Good	*unknown*
Sensitivity to change in light levels	Large	Small	*unknown*
Sensitivity to contrast	High	Low	*unknown*
Sensitivity to motion	High	Low	Low
Color discrimination	Broadband	Red/Green	Blue/Yellow

The function of the K cells remains a bit nebulous: They might play a role in motion detection and where-and-when processing [Eiber et al., 2018] and regulate other visual pathways [Martin and Solomon, 2019], and most likely are heterogeneous and form subpopulations [Casagrande, 1994]. Many details about the K cells remain unknown and are subject to current investigations [Kiley and Usrey, 2016].

Another striking fact about the LGN is that it does not receive most of its input from the retinal ganglion cells, but actually from the visual cortex itself. Through this feedback loop, the LGN is able to play a vital role in the direction of visual attention, focussing, and vergence of the eyes, as well as in stereoscopic mapping of the visual field.

Just as the retinal ganglion cells provide a spatial coding of their inputs by forming receptive fields, the LGN provides a temporal coding, resulting in an even more efficient transmission of information.

In terms of functional relevance to eye movements, the LGN plays an important role in the execution of saccades [Krebs et al., 2010], as well as indirectly controlling the ciliary muscles for vergence and focus described in Section 2.2, *Optical Path*.

2.5 The Superior Colliculus

The Superior Colliculus again has a layered structure, with 7 layers in total. The first three layers are called superficial layers and connect mainly to the retina and the LGN. The remaining intermediate and deep layers receive connections from a variety of sources, such as somatosensory inputs and the cerebral cortex in general. The superficial layers also have outputs to the LGN.

The Superior Colliculus is heavily involved in the control of the eye movements. Each of the colliculi, which are located on the left and right side of the brain, can be mapped to respective halves of the visual field. Experiments with electrical microstimulation in monkeys have shown that, depending on the site of the stimulus, either saccades or fixations can be evoked [Klier et al., 2001]. The coordinate system used by the Superior Colliculus is also not world coordinates, but retinal coordinates, where the area of the colliculus covered corresponds with the receptor counts in the visual field (see Figure 2.2).

2.6 The Visual Cortex

The axons leaving the LGN are called the *optic radiation*, and enter the *primary visual cortex* (also called or *V1* or *striate cortex*) in the occipital lobe of the brain.

In Figure 2.8, we can appreciate the — again — layered architecture of the primary visual cortex: A crucial difference this time is that the contralateral *and* ipsolateral projections arrive in the same layer. Additionally does layer 6 provide a feedback connection to the LGN, while layers 2, 3, and 5 also connect to areas outside the primary visual cortex. Large parts of the primary visual cortex are solely responsible for the fovea, and with increasing visual angle (compare again Figure 2.2), there are less and less cells associated.

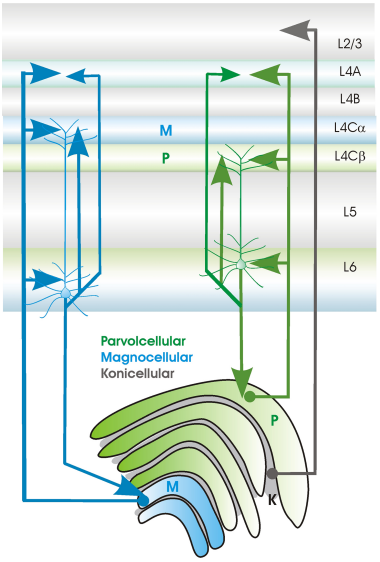

Figure 2.8: Correlation of the layered architecture in the LGN (lower half) with the cell layers in the Primary Visual Cortex (upper half): Neurons from the magnocellular, parvocellular and koniocellular LGN layers project into similar sublayers of the cortex. Observe that V1 layer L6 also projects back to the LGN. Reproduced from [Thomson, 2010].

But what does the primary visual cortex then do with the input? The cells in the primary visual cortex are tuned to orientation, and organised in *orientation columns*, meaning that cells responsible for a particular orientation are in the same column. A collection of orientation columns, called a *hypercolumn* then encodes all the possible orientations occurring in one visual area.

The hypercolumns can contain three different types of cells:

- *Simple Cells*: Not only does the layered architecture continue, but also the division in receptive fields. In contrast to the LGN's center-surround architecture, *simple cells* in V1 form bar structures, with either two or three areas that can be excitatory or inhibitory, and thereby form either bar or edge detectors.
- *Complex Cells:* They add together (e.g. with a mathematical *or* operation) the outputs of multiple simple cells, resulting in a receptive field that is not only sensitive to the orientation of the stimulus, but also to its relative position within the field.
- Finally, *Hypercomplex cells* or *End-stopped cells* wire together multiple complex cells, to additionally provide inhibition if a stimulus exceeds the receptive field —

complex cells would continue to fire there, and are thereby able to detect curved stimuli [Yazdanbakhsh and Livingstone, 2006, Snowden et al., 2011].

These types of cells can now be tuned to a multitude of properties: Some respond to high spatial frequencies, some to low, others are sensitive to size, or temporal frequency.

2.7 Beyond the Visual Cortex

Beyond the visual cortex, further processing of visual signals is done in areas called extrastriate cortical areas. Neurons in these areas are responsible for, e.g., motion tracking. Some of these areas are responsible for *visiomotor integration*, the derivation of motor signal from visual stimuli. We briefly want to discuss how the saccades, smooth pursuits, and the VOR/OKR are elicited by the visual system and the motor system [Prsa and Thier, 2016, Cullen, 2016]:

For *saccades*, both the amplitude and direction are controlled by the superior colliculus, which contains a neuronal map in terms of amplitude and direction. The neuronal inputs to this map stem from a wide population of neurons. The delay to saccade onset is about 100ms.

In the case of *smooth pursuits*, the eye movements are controlled by an interplay of neurons in the visual cortex, the brainstem, and the cerebellum. Cerebellar neurons in the floccular lobe and the vermis project to the motoneurons of the eye muscles. In addition, extra-visual signals, e.g., from anticipation of movement, are integrated into the process. While the cerebellum does not directly execute movements, it provides information to motoneurons and is constantly re-calibrated [Cullen, 2016]. The interplay of these systems allows accurate tracking of objects with speeds of more than 50º/s, starting only 100ms after motion onset. However, if the target's velocity changes direction frequently, this performance degrades.

Compared to the two eye movements just mentioned, the *Vestibuloocular Reflex* (VOR) has a remarkably fast response time of only about 5ms. It integrates information from the vestibular organs (the semicircular canals and otolith organs) in the inner ear, which respond best to relatively fast translational and rotational movements (>0.1 Hz). Slower movements are compensated by the *Optokinetic Reflex* (OKR), which is evoked by information of visual origin instead of vestibular origin. For the OKR, similar circuits as for smooth pursuits are involved. In addition, the OKR is controlled by the *Accessory Optic System*, which receives inputs from the Medial Superior Temporal Cortex (MST) and the Middle Temporal Cortex (MT).

2.8 Consequences for the Design of Eye-based Interfaces

In extension of [Duchowski, 2017], from the architecture and the physiological features of the visual system and its integration with the motor systems, we can draw a series of consequences for the design of gaze-based user interfaces:

Physiological Feature	Consequence
Cones fall off sharply outside the foveal region, high number of rods still exists in the periphery.	Color information (chrominance) can fall off as sharply outside the foveal region, while brightness (luminance) should not degrade as fast.
Magnocellular ganglion cells respond most strongly to stimuli in the periphery.	Contrast changes should only happen on purpose in the periphery, as the visual system can react strongly to objects appearing suddenly in the periphery, responding e.g. with a saccade to the new highly salient object. Interfaces where gaze is used as selection modality, or where gaze is used more passively, e.g. to indicate current attention (gaze-contingent interfaces, see Section 4.3) or which actively manage the users attention (attentive user interfaces) need to take special care about this.
Saccades lead to temporary blindness (saccadic suppression).	Eye tracking information correlated with e.g. saliency during a saccade might not be useful.
Smooth movements are not possible without smooth pursuit movements.	The limited possibility of voluntary, smooth movement needs to be taken into consideration e.g. when designing eye gesture-based interfaces.
Smooth pursuits are cannot track all movements equally well.	While smooth pursuits can easily track movements in excess of 50º/s, their reliability decreases with oscillatory movements faster than 1 Hz. However, if the motion can be anticipated, reliability is increased again.
Eye movements have characteristic velocities and durations.	Needs to be taken into consideration for systemic modelling of eye movements. As consequence, gaze-contingent or attentive user interfaces must not react too fast or require too swift user interaction.
Eye movements have processing delays.	While the VOR/OKR only incurs a processing delay of about 5ms, both saccades and smooth pursuits take 100ms, so fully-instant reaction from the user cannot be expected.
Eye tracking is not instant.	The processing delay of eye tracking hardware and software also needs to be taken into account.

2.9 Summary

In this chapter, we have summarised the machinery and neural circuitry behind the human visual system. We described the human eye and its movements, the structure of the retina, and the neural structures following downstream, such as the superior colliculus and the primary visual cortex. We have described the major eye movements and briefly discussed the interactions of the visual system with the motor control systems of the brain controlling the eye movements. Finally, we drawn conclusions for the design of eye-based user interfaces.

While this chapter aims to provide a concise introduction to the visual system and its integration with the motor systems, it is far from complete. For more detailed descriptions of the human visual system, we refer the interested reader to the book *Basic Vision* [Snowden et al., 2011], the series *Neuroscience in the 21st century*, especially the chapters about the *Retina: Neuroanatomy and Physiology* [Reichenbach and Bringmann, 2016], *Cortical Processing of Visual Signals* [Kiley and Usrey, 2016], *Cerebellum: Eye Movements* [Prsa and Thier, 2016], and *Visiomotor Integration* [Cullen,

2016]. In addition to these books, Duchowski's book, *Eye Tracking Methodology: Theory and Practise* [Duchowski, 2017], provides another introduction to the visual system, with special emphasis on the relevant parts and systems for eye tracking.

2.10 Challenges and Opportunities

2.10.1 Efficient representation of volumetric data

Adaptive sampling and by that, data reduction is done very efficiently already by the retinal ganglion cells. In Section 2.3.2, *Retinal ganglion cells*, we discussed that by the formation of receptive fields, these cells already reduce the data that has to be transmitted through the optic nerve from 4 GiB/s to about 30 MiB/s. Is it possible to use a similar approach for data reduction in the processing of volumetric data? In Chapter 13, *Rendering the Adaptive Particle Representation* we discuss a data reduction technique, the Adaptive Particle Representation [Cheeseman et al., 2018] inspired exactly by that.

2.10.2 Object tracking with support of the visual system

A task often encountered in image-based developmental and systems biology is the tracking of objects in volumetric data. One example is to identify cells in consecutive volumetric images that correspond to each other. Another example is the tracing of neurons from large still images, to ultimate generate a connectome — a representation of which neuron connects to which — in an effort to identify functional connections and correlations (see e.g. [Swanson and Lichtman, 2016] for a review). In the chapter Chapter 11, *Bionic Tracking: Using Eye Tracking for Cell Tracking* for a prototype of how to use smooth pursuit eye movements for cell tracking. Solving such tracking problems via eye tracking further requires robust eye tracking algorithms, a topic we also briefly touch in that chapter.

2.10.3 Optimal Viewpoint Determination by Modelling Visual Attention

Models for modelling visual attention based on image content have already been proposed [Itti and Koch, 2001, Gao et al., 2014]. In the context of image analysis and visualisation of large datasets it is becoming more important to find optimal viewpoints, e.g. for exploration, education or presentation purposes. A computational model of visual attention can help here to select the visually most interesting or salient images and viewer positions. One could imagine feeding the model with a random selection of viewpoints on the dataset, and evolving them in a manner they converge to the most salient points. Another option would be a combination of eye tracking and saliency modelling: The area the user is looking at could be analysed for the most salient neighbourhood, and the dataset translated or rotated accordingly. Both are however beyond the scope of this work, but might be pursued in the future.

Chapter 3:

A Short Introduction to Cross Reality

Cross reality, or *XR*, encompasses everything on the spectrum between fully virtual environments, and fully real environments. This includes especially virtual reality (VR), augmented reality (AR), and augmented virtuality.

In this chapter we give a brief overview of existing technology and current developments in the areas of virtual and augmented reality. We will explain the benefits of VR and AR, and outline associated challenges and opportunities offered, with an emphasis on biology and imaging. In the end we will sketch issues addressed in this thesis.

3.1 Virtual Reality, Augmented Reality, Mixed Reality

> "*Virtual Reality* is the computer-generated simulation of a three-dimensional image or environment that can be interacted with in a seemingly real or physical way by a person using special electronic equipment, such as a helmet with a screen inside or gloves fitted with sensors." — *Oxford Dictionary of English*

With the term *Virtual Reality* we describe environments that simulate parts of the real-world experience of human beings, such as the visual surroundings, auditory perception, and sometimes even proprioception[1] in an interactive, computer-generated three-dimensional environment. The world exterior to the simulated environment plays no role here, such that the user can become shut off from her real surroundings and fully immersed in the simulation, if it is convincing enough.

[1] Proprioception is the sense of relative motion and positioning of one's own body and/or its parts.

If the surroundings of the user are actually visible, e.g. via a set of glasses that are transparent and show the outside environment (or show them via cameras) and overlay information on top of it that extend or augment the capabilities or information content of the environment, we speak of *augmented reality*.

In the case of a mix of both, where there is a direct connection or overlap between the virtual, simulated world, and the real world, the setting is termed *mixed reality*. *Mixed reality* might take place anywhere in the *virtuality continuum*, except the extremal points of fully real environments, or fully virtual environments, while *cross reality* encompasses the full spectrum [Milgram et al., 1995].

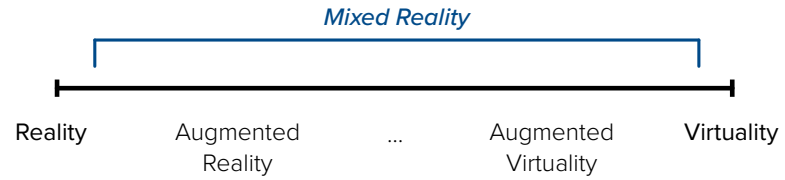

Figure 3.1: Virtuality continuum according to [Milgram et al., 1995], where mixed reality encompasses all settings that are not the extremal points, and cross reality encloses the extremal points as well.

3.2 Historic Perspective — 1800s to 1990s

The first virtual reality "glasses" have been introduced in the 1850's, as so-called *stereoscopes*, looking not unlike contemporary head-mounted displays. In the stereoscopes, the user would insert a postcard that is split in the middle in two parts, showing the subject of the postcard from two slightly different perspectives corresponding the capturing an image with two eyes, as in the human visual system.

In the early 1950, the *Sensorama* was introduced, an immersive movie theater, that not only included stereoscopic visuals, but also wind, sound, and even smell. The machine is shown in Figure 3.3.

With computer graphics still in it's infancy, the first steps towards a head-mounted display mainly for military purposes, were made in 1968 by Ivan Sutherland [Sutherland, 1968]. Sutherland developed a glasses-based virtual reality system (actually, augmented reality) that consisted of cathode-ray tubes mounted on the users head, with images being directed to the eyes by the means of mirrors. The tracking system for the contraption was suspended from the ceiling, looming over the user, hence the name of the system, *The Sword of Damocles*. *The Sword of Damocles* could display wireframe models of geometric objects overlaid onto the user's surroundings, and adapted to the viewpoint that had been calculated by the tracking system.

Big steps towards the current state of virtual and augmented reality were taken in the 1980s and 1990s by the University of Southern California's *Mixed Reality Lab*, and the company VPL, a spin-off of the lab. The lab developed not only head-mounted displays, but full-body virtual reality suits, providing the user with a force-feedback system, and gloves developed for NASA that would react to virtual objects and the user's grip [Zimmerman et al., 1987]. In addition to the personal systems based on head-mounted displays, room-scale systems such as the *CAVE* [Cruz-Neira et al., 1992] — a backronym for *CAVE Automatic Virtual Environment* — were developed in the mid-1990s. In contrast to the HMDs, these systems use the tracking of the user not to display a perspective-corrected image on a screen attached to the user's head, but on a (front or back-)projected wall or display at a distance to the user. Compared to HMDs, such CAVEs have the benefit that multiple people can use it simultaneously, with the constraint that only a single person will see the fully correct three-dimensional, immersive image. CAVE systems have found a large user base in the automative industry, and in architecture and design [DeFanti et al., 2010].

In the 1990s, interesting applications for various VR settings were explored in the research field. Especially UNC Chapel Hill's Virtual Reality Lab created a lot

Figure 3.2: A Holmes-type stereoscopes to view left/right-eye images as single image. Public Domain.

Figure 3.3: The sensorama. Image reproduced from Sensorama, Inc. Advertisement, 1962.

Figure 3.4: The *Sword of Damocles*. Note the cathode-ray tubes mounted to the sides of the user's head, and the mirrors directing the image to the eyes. Reproduced from [Sutherland, 1968].

of solutions for diverse areas such as pharmaceuticals [Brooks et al., 1990], electron microscope control [Taylor et al., 1993], or architecture [Airey et al., 1990], with the example of VR protein docking supported by haptics shown in 3.2.

Another fascinating idea from the 90s is the omnidirectional treadmill for exploring virtual worlds [Darken et al., 1997], where a moving 2D conveyor belt would compensate the user's movement in the virtual environment. These developments have led to the *First Virtual Reality Revolution*, aiming at ubiquity of virtual reality devices and their usage, sprouting movies and conferences focused on VR, and companies channeling R&D money into VR technology. Nicholas Negroponte conjectured in 1993 a widespread use of VR devices, and a company that "will soon introduce a VR display system with a parts cost of less than US$25"[2], while Fred Brooks estimated in 1994 "we will see high-resolution, low-lag systems doing serious applications within 3 years", although acknowledging that display technology back then was so bad it made the user "legally blind" [Bryson et al., 1994].

Unfortunately, the First Virtual Reality Revolution was not successful, at least from a commercial point of view — and most of the companies betting on its success went out of business until 1998 [Jerald, 2015]. Some reasons for the failure were:

- Due to the high cost of the systems, few people and labs were able to afford them, and often the systems remained only in research use,
- ergonomics issues arising both from the size and weight of the systems prevented usage for more than a short period of time, with Randy Pausch stating, "approximately 10% of the visitors adamantly decline the opportunity to wear a head-mounted display" [Bryson et al., 1994], and
- the visual fidelity then-contemporary computers could produce when rendering digital 3-dimensional imagery were simply neither good enough nor fast enough to provide a fully convincing, not sickness-inducing, experience.

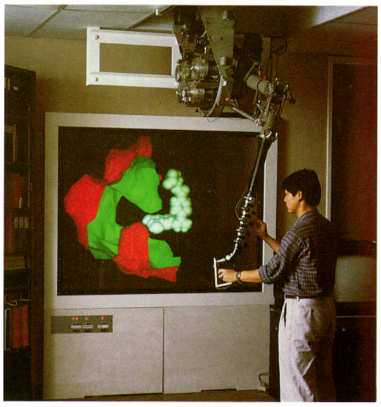

Figure 3.5: Protein docking example using the haptic *GROPE-III* system. Users reported a radically improved situational awareness from using the system. From [Brooks et al., 1990].

[2] See wired.com/1993/06/negroponte-11/.

3.3 Current Developments

The currently ongoing *Second Virtual Reality Revolution* has been enabled — at least in part — by the development of low-cost, high-resolution displays that are used in smart phones, and the gyroscopic sensors used alongside them.

The displays used in mobile phones form the ideal basis for ergonomic and lightweight head-mounted displays, as they feature both a low physical footprint, low energy use, and the right size and resolution to be put right in front of the eyes.

After showing several prototypes of head-mounted displays, Palmer Luckey, a former employee of the *Mixed Reality Lab*, produced the *Oculus Rift* in 2016 (see Figure 3.3 for the prototype, and Figure 3.3 for the final product), a translational and rotational tracking HMD complete with tracking system, based on full-HD smartphone displays. Soon after the *Rift*, other manufacturers presented similar devices, such as HTC's *Vive*, Sony's *Playstation VR*, Samsung's *GearVR*, or the set of Microsoft's *Windows Mixed Reality* glasses (a slight misnomer, being actually virtual reality glasses).

Figure 3.6: An early Oculus Rift prototype. Image reproduced from Engadget, https://www.engadget.com/2012/08/16/oculus-rift-hands-on/.

Figure 3.7: The Oculus Rift Virtual Reality HMD. Public domain.

On the side of augmented reality, Microsoft has been selling the developer kit of the *HoloLens* since 2016. The HoloLens is an untethered headset with its own CPU, GPU, and HPU (holographic processing unit, apparently used for tracking tasks, etc.). The HoloLens features *inside-out tracking*, where no external tracking hardware is needed, apart from the cameras inside the HMD itself. On the software side, the HoloLens supports rendering directly on the device via Direct3D11, or via low-latency remote rendering (named Holographic Remoting) on a separate computer, and streamed image transfer, with the images encoded as H264 video stream, and corrected by the HoloLens on-the-fly with affine transformations for rotations and translations, to compensate for network latency. In early 2019, Microsoft announced the *HoloLens 2*, with improved field of view, latency, and physical footprint (see Figure 3.8).

Figure 3.8: The HoloLens 2. Promotional picture, from microsoft.com/en-us/hololens.

Another available AR headset (as of 2019) is the *Magic Leap One*, featuring advanced optics, with three planes of focus, and a dedicated processing unit (dubbed *light pack*) featuring 8 GiB RAM and an ARM-architecture Nvidia Tegra X2 with an integrated Pascal-generation GPU with 256 CUDA cores, tentatively providing more compute power than the HoloLens. Also on contrast to the HoloLens that runs Windows 10, the Magic Leap One runs a custom Linux distribution named Lumin OS.

Figure 3.9: The Magic Leap AR Headset. Promotional picture, from magicleap.com.

Both the HoloLens and the Magic Leap One provide a glimpse of what will be possible, comfortable, and easy with mixed reality devices at some point in the near future. Both still suffer from a lack of resolution, field of view, and computational power, such that the impression these headsets leave in actual reality are still a bit of a stretch from their promotional materials.

In contrast, current-generation Virtual Reality HMDs are able to display convincing virtual reality environments to the user, with a high frame rate, and a large field of view, given a potent-enough CPU and GPU are used to produce the images. Compared to the *First VR Revolution*, performance of the rendering computers, and even more importantly, the tracking systems, has increased tremendously, the form factors and weights of the HMDs have gotten to usable and ergonomic dimensions, and their price has been reduced substantially, such that a VR-capable computer system, including the HMD, can be bought for as little as about €1500 in early 2019.

Current research topics in the usage of virtual reality include how people explore virtual environments [Sitzmann et al., 2018], the combination of multiuser virtual reality with physical systems on top of 5G low-latency networks [Bastug et al., 2017], or foveated rendering to gain significant speed-ups [Patney et al., 2016][3]. Clinicians have renewed interest in the usage of VR technologies in psychology and psychiatry, such as for treatment of anxiety disorders [Maples-Keller et al., 2017] or for the rehabilitation of stroke victims [Laver et al., 2017]. In the context of biology, VR has recently been used to sample molecular conformations in a multiuser environment [O'Connor et al., 2018], combining rendering on VR HMDs with cloud-based simulations of molecular structures, or for the visualisation of endocytosis datasets from electron microscopy [Johnston et al., 2018].

[3] In foveated rendering, only the part of the image seen by the user's fovea, the part of the retina with the highest spatial resolution, is rendered at full resolution.

For augmented reality, the visualisation of complex data, such as large graphs is an active area of research [Büschel et al., 2019], and it is also being explored as a new modality for neuronavigational systems in neurosurgery [Meola et al., 2017]. Utilising AR in combination with cyber-physical systems, such as human-robot collaborative assembly systems [Makris et al., 2016] or for debugging distributed systems [Reipschläger et al., 2018] is also investigated.

3.4 Issues

3.4.1 Motion Sickness or Simulator Sickness

Motion or Simulator sickness can occur when there is a disparity between the motion seen by the eyes, and the motion perceived by the vestibular system. There are several hypothesis why it arises [Jerald, 2015]:

- *sensory conflict theory* — motion sickness arises due to a conflict of the visual, vestibular, or proprioceptive systems that cannot be reconciled,
- *evolutionary theory* — the disparity of sensations from the different system is assumed by the body to originate in being poisoned, which is counteracted by the need to lie down, vomiting (to get rid of ingested poison), and nausea, to prevent the consumption of more poison,
- *postural instability theory* postulates that one of the main goals of animals is maintaining a stable posture, and sickness is a reaction to incomplete or inexistent learning — which also means people using VR systems for a while may experience a lessening in the intensity of their motion sickness, or
- *eye movement theory/nystagmus theory* — motion sickness arises from unnatural eye movements that would be required to stabilise the image on the retina. Both the vestibulo-ocular reflex (VOR) and the optokinetic reflex (OKR) are involved in stabilising images and the lack of saccadic suppression might also play a role (see the chapter [Visual Processing] for details on eye movements and processing).

Whatever the actual reason for motion sickness might be, it has to be kept in check for a user to be able to comfortably use a VR system. For that, several defences can be used:

- the system should run with a frame rate of 60fps, or better 90fps to maintain a fluid appearance, as everything below will be perceived as stuttering and can increase motion sickness — it is advisable to rather sacrifice realistic rendering instead of frame rate (see [Visual Processing] on more information about what movements are perceived as fluid),
- certain kinds of movement, such as lateral movements should be avoided, as they do not occur in the real world, and teleportation should be the preferred way of moving, with fading transitions at start and end[4], alternatively, dynamically reduce the field of view during fast movements [Fernandes and Feiner, 2016],
- the system should be set up to use the correct inter-pupillary distance (IPD) of the user, to provide the same image convergence as in the real world [Ukai and Howarth, 2008],

[4] See developer.oculus.com/design/latest/concepts/book-bp/.

- the tracking system used should be well-calibrated and work with low latency to avoid stuttering and jerky movements [Mania et al., 2004, Jerald, 2015].

Furthermore, there has also been research into other ways to prevent motion sickness, which has produced fascinating results: In [Whittinghill et al., 2015], the authors report on the addition of a "virtual nose" to the rendered scene, which reduces the occurrence of motion sickness and enabled their users to use their system for longer amounts of time without getting sick.

This list is of course far from exhaustive: In addition to the countermeasures just described, [Jerald, 2015, Chapter 19] offers a very comprehensive list of guidelines to counter adverse health effects when designing and using VR systems. In addition, [Clift, 2018] provides a review of both software and hardware solutions against motion sickness in VR.

3.4.2 Lack of Vergence

Current, commercially-available HMDs do not provide focus cues for the eye. This not only completely precludes the use of vergence for user evaluation or control, but also makes the issue of simulator sickness, described in the section before, worse.

Research-grade HMDs try to solve this now using light-field rendering [Wetzstein et al., 2013], to provide focus cues for the eye. In [Huang et al., 2015] for example, a three-layered HMDs is described, providing focus cues for foreground, background, and the area in-between.

Actually giving focus cues to the user would bring detection of the depth the user is looking at, and therefore 3D eye tracking, one step closer.

3.4.3 Hygienic Issues

In settings where a head-mounted display is used by many different people, such as for demo purposes, or at conferences, hygienic problems arise. To be comfortable, a HMD needs some cushioning, usually provided by a foamy insert on the HMD, which over time accumulate dirt and can also harbour germs. One solution to this is to provide washable inserts, as companies like VRCover[5] now offer. Museums using VR systems have also started to use disposable face masks to combat this problem[6].

[5] See vrcover.com

[6] See museumnext.com/2019/01/how-museums-are-using-virtual-reality/

3.5 Challenges

- *Can we use VR/AR for visualising microscopy data after acquisition, and provide a measurable benefit for the user from that?* — Visualisation might occur both at the time of acquisition (e.g. for checking correct imaging parameters), or later on, at the time of evaluation of the data. Both cases have in common that the user will most probably need to interact with the outside environment to adjust the microscope, or just to take notes. It would therefore be not beneficial to encumber the user inside a fully virtual environment, but rather augment the existing environment with the data that has been acquired. Intuitive interactions,

in which the users e.g. sifts through a set of time points of a microscopy dataset much alike to sifting through a pile of papers, would enhance the acceptance of such modalities.

- *Can we use VR/AR to control microscopes and do e.g. laser ablation experiments more efficiently?* — This use case can make use of both augmented and virtual reality settings: While augmented reality would benefit the user at design time of the instrument, e.g. by overlaying rulers, angles, and component descriptions on the optical table. Virtual reality on the other hand could be beneficial while performing e.g. laser ablation or optogenetic experiments undisturbed, in a fully immersive environment, fully concentrated on the specimen.

Chapter 4:

Eye Tracking and Gaze-based Interaction

Eye tracking is the process of following the direction of the user's eyes in order to determine what the user is looking at, and deriving information, context, and actions from that.

4.1 Eye Tracking Technologies

In this section, we're introducing common eye tracking technologies, and in the end compare them for user-friendliness and applicability for practical usage scenarios.

4.1.1 Search Coil Contact Lenses

Search Coil Contact Lenses constitute the earliest[Robinson, 1963] and probably most invasive form of eye tracking. For this form of eye tracking, a contact lens fitted with a coil is put directly on the user's sclera, who has to sit in a uniform magnetic field. Movements of the coil then cause an electric current, which is measured and correlated with the user's eye angle. This kind of eye tracking yields high spatiotemporal precision (< 1ms temporal resolution, <1º spatial resolution). See Figure 4.1 for a sketch of the principle.

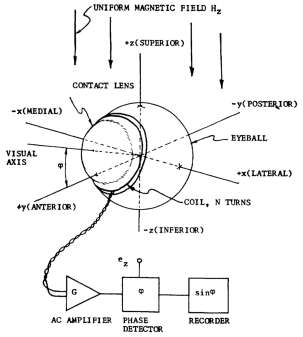

Figure 4.1: Scleral search coil contact lens eye tracking schematic, reproduced from [Robinson, 1963].

Figure 4.2: Electrooculography in use, still image reproduced from Biopac Student Lab, youtu.be/QXGiZBDkUw

4.1.2 Electrooculography

In electrooculography, the user's eyes are tracked by measuring potential changes on its skin. The technology employed is very similar to electroencephalography (EEG) or electromyography (EMG) — electrodes are placed around the eyes, and from the registered signal, the horizontal and vertical orientations of the eye ball are inferred. Additionally, a scalp electrode can be placed to also measure radial movements, yielding highly precise timings for the onset of saccadic movements [Keren et al., 2010]. Albeit EOG is plagued by the same issues as EEG/EMG — namely drift over time, and not entirely reproducible signal amplitudes, leading to inaccuracies for the generation of absolute eye positioning data — it does not rely on the user's eyes being visible, which can be a plus depending on the usage scenario. EOG can yield a horizontal resolution of 1-2º, while the vertical accuracy is usually less due to artifacts from lid movements [Heide et al., 1999]. Compared to contact lenses it is much less invasive, and compatible with wearers of both glasses and regular,

correcting contact lenses. See Figure 4.2 for an exemplary electrooculography setup.

4.1.3 Videooculography and Purkinje Imaging

In videooculography, the user's eyes are imaged by one camera per eye, focused on the pupil. Contemporary devices use infrared light to either track the pupil directly, or to image reflections created by the anterior and posterior surfaces of the cornea — the Purkinje images, see Figure 4.5 — which can then be used to calculate a vector between the pupil's center and the reflection to determine the point the user is looking at [Gneo et al., 2012]. This technique requires calibration in the beginning, and might also drift in longer tracking sessions. Yet, it is very unintrusive, and only requires a camera mounted to either an existing or new set of glasses (see Figure 4.3), or a set of two cameras mounted inside a HMD (see Figure 4.4). VOG yields a high spatial accuracy (~1º), while temporal accuracy suffers a bit due to camera and processing latencies (~3-5ms).

Figure 4.3: A videooculography setup, the Pupil Pro headset. Reproduced from [Kassner et al., 2014].

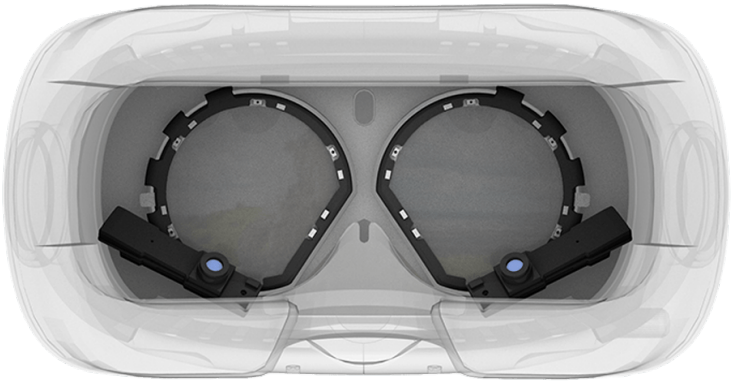

Figure 4.4: Videooculography using a HMD-based eye tracker from Pupil Labs, mounted on an HTC Vive. Image reproduced from pupil-labs.com/vr-ar.

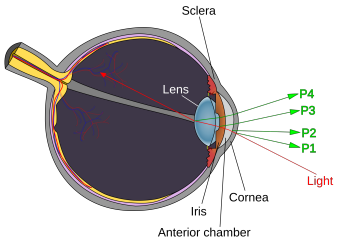

Figure 4.5: The physical origin of Purkinje images P1 to P4: *P1*, reflection on anterior corneal surface; *P2*, reflection on the posterior corneal surface; *P3*, reflection on the anterior surface of the lens; *P4*, reflection on posterior surface of the lens. Image (cc) by Z22, Wikimedia Commons.

In addition to the regular videooculography modalities, a recent study from Wang, et al. [Wang et al., 2016] makes use of thermal video imaging of the cornea, which is about 0.5ºC cooler than the limbus. They segment the thermal image, and locate the position of the cornea in the segmentation. They achieved a fair accuracy of ~2.3º, mostly limited by the resolution of the thermal sensor.

With *Pupil* [Kassner et al., 2014] and *Oculomatic* [Zimmermann et al., 2016], now open-source, open-hardware solutions exist for videooculagraphy-based eye tracking.

4.1.4 Comparison

Table 4.1: Comparison of eye tracking modalities.

Modality	Setup Effort	Intrusive-ness	Comfort	Spatial Accuracy	Temporal Accuracy
Scleral Contact Lenses	Very High	Very high	Low	<1º	<1ms
Electroocu-lography	High	Medium	High	1-2º	>2º
Videoocu-lography	Low	Low	High	1º	2-5ms

From the discussion of the various modalities, we conclude:

- *Search Coil Contact Lenses* remain the gold standard for eye tracking in clinical settings, where the highest spatiotemporal precision is required, and precision outweighs their low user comfort and high setup effort.
- *Electrooculography* is the most useful technique if relative eye coordinates are sufficient, or if the application might include the user's eyes not being visible, e.g. as it is the case in sleep analysis applications.
- *Videooculography* is the most useful technique when it comes to day-to-day use, due to its easy setup and low to inexistent user discomfort. Also, both software and hardware for such a setup is readily available [Kassner et al., 2014, Zimmermann et al., 2016].

4.2 Common issues

There exist some issues common to eye tracking applications: In this section, we are going to detail the *Midas Touch Problem* and the *Double Role of Gaze*, Accuracy and Reliability, Availability, and Privacy.

4.2.1 Midas Touch Problem and the Double Role of Gaze

The notion of the *Midas Touch Problem* was introduced by Jacob in 1990 [Jacob, 1990], and describes the issue that by looking at an object, the user inadvertently triggers an action. It is named after the Greek fable of King Midas, who, after wishing that everything he touches would turn to gold, ultimately starved to death.

The Midas Touch Problem is intimately linked with the *Double Role of Gaze*: while visual attention can be actively directed, it is also often influenced by visual distractions that carry a high saliency, such as flashing lights, fast moving objects, or even input from other senses, such as loud bangs. A shift towards passive attention may cause a disruption of the workflow of the user, or can, in the sense of the Midas Touch Problem, lead to wrong inputs. The issues are especially pronounced in the case that one wants to emulate mouse-based input with gaze input.

Both issues can be addressed e.g. by providing the user with an additional means to confirm that the selection action is actually the one she intends to perform. This can be achieved with a variety of means:

- *dwell time-/blink-based selection*, where an action is only triggered after the user has rested her gaze on the object [Jacob, 1990], or blinked once or multiple times as confirmation [Jacob, 1993, Ashtiani and MacKenzie, 2010]. These solutions lead to additional delays for the input, which, depending on the intent might or might not be a problem: If fast interaction is intended, e.g. for selecting highly salient objects in fast succession, dwell/blink-based confirmation is problematic, while when interacting e.g. with locked-in patients, it may provide an excellent way for communicating [Ashtiani and MacKenzie, 2010].
- *multimodal interaction*, where the user can utilise an additional device to confirm her intent, for example by pressing a button on a keyboard[Castellina and Corno, 2008], using an additional touchpad [Meena et al., 2017], a foot pedal or foot mat [Klamka et al., 2015, Hatscher et al. [2017]], or free-air pinch gestures [Pfeuffer et al., 2017].
- *computer vision techniques*, where visual attention and saliency [Itti and Koch, 2001] is modelled computationally, to determine which is the most probably object of attention at the moment [Wu and Wang, 2015, Theis et al., 2018].

4.2.2 Accuracy and Reliability

Due to individual differences between users, and also between different spike trains in EOG, measurement of gaze cannot be 100% correct. For cursor-based applications, filtering approaches can be used to weed out erratic recognitions [Zhang et al., 2008], and adjusting the interface shown to the user, e.g. via magnifying it or exaggerating details (see [Cockburn et al., 2009] for a review).

HMD-based eye tracking here has the benefit that the lighting situation can be controlled, and it'll mostly be dark inside the eye piece of the HMD, leading to more predictable and reliable gaze detection. For screen-mounted or mobile eye trackers however, the situation is a bit more difficult, as they might be used in many different lighting scenarios.

In terms of recognition reliability, advances have been made in recent years towards model-free determination of gaze, e.g. via artificial neural networks [Gneo et al., 2012]. One can expect this trend to continue, leading to more reliable algorithms. *Pupil* [Kassner et al., 2014] for example uses a combination of image segmentation combined with model-based gaze estimation.

4.2.3 Availability

Still back in 2014, eye tracking hardware was very expensive, with both screen-mounted and HMD-mounted trackers costing in excess of 10000 EUR.

Since then, new projects have emerged that provide either low-cost or open-source eye trackers, or even both. Examples of such projects are:

- *Pupil* [Kassner et al., 2014], where ready-to-use eye trackers for glasses, mobile or HMD use can be bought, but the bill of materials, construction manual, and software are open-sourced (see github.com/pupil-labs/pupil and docs.pupil-labs.com).
- *Oculomatic* [Zimmermann et al., 2016], which provides an open-source toolkit, as well as schematics for high-speed eye tracking, primarily for use in oculomotor research.
- *aGlass*, which has recently announced the availability of a low-cost, full-FOV eye tracking development kit for the HTC Vive, mainly to facilitate foveated rendering [Pohl et al., 2016].

These developments lead us to believe that widespread and cost-effective use of eye-tracking hardware will soon become a reality.

4.2.4 Privacy

With eye movements being an additional data point that can be used to fingerprint persons and track them through different media and situations, privacy is of course a concern when employing eye tracking.

This concern becomes even more important, as Hoppe, et al. have recently shown that tracking eye movements in daily activities is able to predict four of the Big Five personality traits [Hoppe et al., 2018] — namely neuroticism, extraversion, agreeableness, and conscientiousness —, and additionally, also perceptual curiousity [Collins et al., 2004]. The possibility of doing that makes it absolutely clear that acquired eye tracking data has to stay with the user, and must not leave the computer for outside processing, as the misuse potential is very high.

With e.g. Facebook also evaluating eye tracking for foveated rendering for their next-generation Oculus headsets[1], it remains to be seen whether eye tracking will turn into a privacy nightmare, or stay user-governed as a very promising and useful technology for future human-computer interaction.

[1] See e.g. uploadvr.com/oculus-is-working-on-eye-tracking-technology-for-next-generation-of-vr/ or uploadvr.com/oculus-patented-new-eye-tracking-device-days-acquiring-eye-tribe/.

4.3 Classification of Gaze-based User Interfaces

Several classifications of gaze-based user interfaces have been proposed:

- [Duchowski, 2017] proposes a distinction between *interactive* and *diagnostic* eye tracking systems, and further discerns the interactive systems into *selective* and *gaze-contingent* systems. Gaze-contingent systems, where the user's gaze is utilised to determine e.g. the current area of interest, are then divided into screen-based and model-based systems. See Figure 4.6L for a depiction.
- [Stellmach, 2013] extends on Duchowski's classification, keeping the distinction between interactive and diagnostic eye tracking, but discerning the interactive part into *gaze-directed pointing*, *eye-based clicking*, and *eye gestures*. Pointing is further distinguished into by precision (precise, coarse) or abstraction (point, target, or area-based). Gaze-contingent interaction is included in the gaze-directed pointing

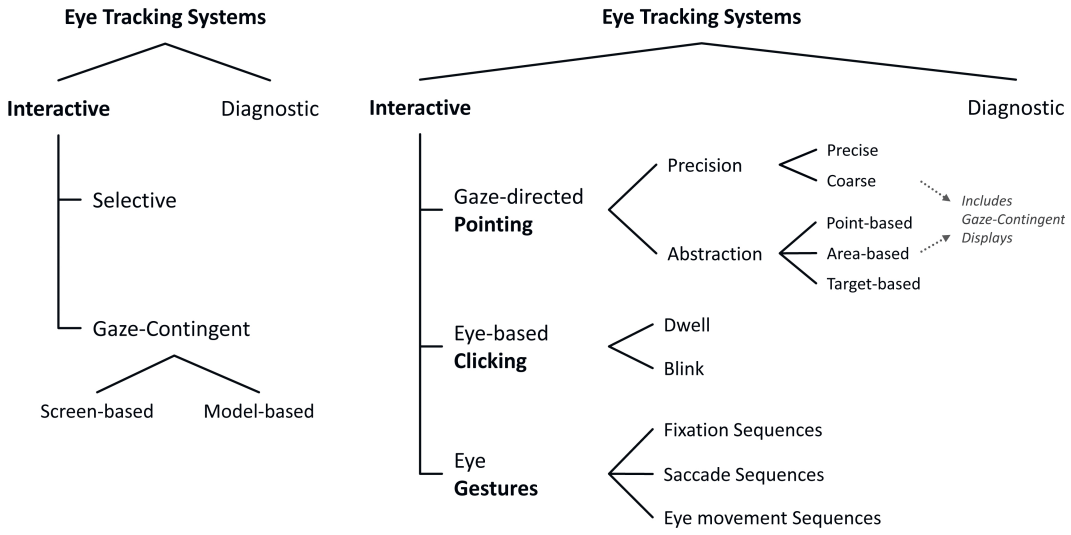

Figure 4.6: Classification of gaze-based interaction. Left: According to [Duchowski, 2017], Right: according to [Stellmach, 2013]. Figure reproduced from [Stellmach, 2013].

branch. The gaze-based input modalities are optionally combined with another input device (multimodal input). See Figure 4.6R for a depiction.

- [Hirzle et al., 2019] introduces another classification scheme, specialised for the use with head-mounted AR or VR displays. Their scheme aims at both at the classification of existing applications, and at an ideation tools to create new interactions. Hirzle's approach classifies based on the use of *device type* (VR or AR), *display type* (monoscopic or stereoscopic), *world knowledge* (based on [Milgram et al., 1995], full or none). World knowledge deserves more explanation: It describes how much the computer knows about the surrounding world of the user — none means no knowledge, and full means complete knowledge of geometry and also semantics. In the case of VR, the computer can only have full information about the user's surroundings, while in AR, only an overlay image could be presented to the user, or the device could have almost complete knowledge of the surroundings by scanning it, such as state-of-the-art devices such as the HoloLens or the MagicLeap do.

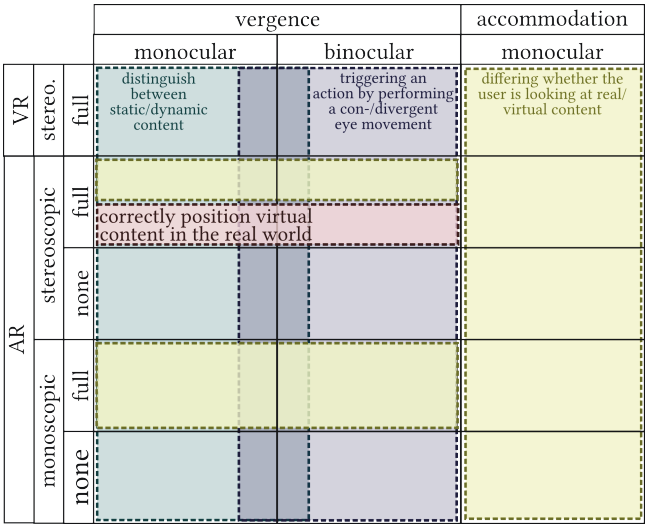

Figure 4.7: Gaze-based interaction classification according to [Hirzle et al., 2019]. In this figure, the interaction-centric view on the classification is shown, with specific interaction tasks classified into the respective fields. Figure reproduced from [Hirzle et al., 2019].

For a more in-depth discussion of gaze-based input and its categorisation, see [Stellmach, 2013]. For the remainder of this chapter, we want to discuss two types of gaze-based input specifically, namely *gaze-contingent user interfaces* and *attentive user interfaces*.

4.3.1 Gaze-contingent user interfaces

The paradigm of gaze-contingent user interfaces is based on adjusting the displayed information depending on where the user is gazing. It is not a selection modality, but one that rather uses the context of the gaze. As described above, Duchowski [Duchowski, 2017] categorises gaze-contingent interaction into the model-based approach and the image-based approach:

- *image-based*: an image or video is modified based on the user's gaze
- *model-based*: the information presented to the user is modified before it is actually rendered, e.g. by adjusting the level of detail of a 3D model, reducing the total polygon count.

An interesting example is foveated rendering, a development based on the non-uniform and fovea-focused resolution of the human visual system (discussed in the chapter [Visual Processing]). In foveated rendering, only the area where the user is gazing is rendered at full resolution, while the surrounding area is rendered with less resolution, or may even lack color information at all, because there are no cones in the periphery of the retina to detect such. An early example of foveated rendering is [Levoy and Whitaker, 1990], where the authors adapted a volume rendering software to only render the area that will be projected onto the fovea in full resolution, while the remainder is rendered at reduced resolution. A more recent example is [Bruder et al., 2019], where the authors propose expand on the original idea of foveated volume rendering by modelling visual acuity and optimise their sampling strategy according to that (although their approach comes at the cost of requiring heavy pre-processing). Images from both publications are shown in Figure 4.8.

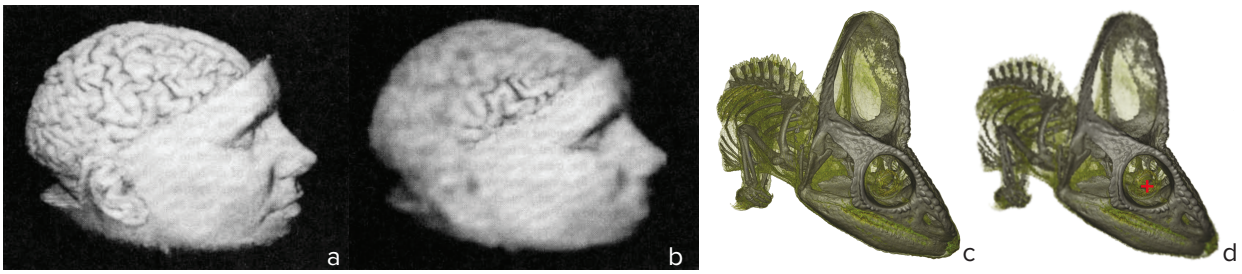

Figure 4.8: Foveated volume rendering evolution: a, b: Foveated and unfoveated volume renderings from [Levoy and Whitaker, 1990]. c, d: Foveated and unfoveated volume renderings from [Bruder et al., 2019].

Aside from volume rendering, Nvidia's recent Turing GPUs (RTX 20xx series) support *variable-rate shading*, which can be used to implement foveated rendering for regular rasterized or ray-traced graphics.

Considering that more and more, and also more complex data needs to be visualised in science and industry, gaze-contingent interaction is sure to stay very relevant

in the future, especially if combined with computational models of saliency like proposed in [Itti and Koch, 2001, Gao et al., 2014].

4.3.2 Attentive User Interfaces

Attentive User Interfaces aim to develop less intrusive and more sociable interface to computers: By detecting what the user is currently occupied with and how important that is, the algorithm will determine whether or not the user should be disturbed. For example in [Shell et al., 2003] the authors describe an attentive cell phone that can detect whether the user is currently engaged in a conversation (in the slightly intrusive manner of attaching a camera to the conversation partner). The cellphone will only proceed to interrupt the user when this is not the case. Another interesting example is the work presented in [Singh et al., 2018], where the authors describe a framework for estimating the users emotional state from eye tracking and physiological data, in order to present a user interface that is most suited in a given life-critical situation. In [Bulling, 2016], the author argues for the necessity of ubiquity of attentive user interface, in order to cope with the problem of continuous distraction in a multi-billion display world.

4.4 Summary

In this chapter, we have introduced the various modalities that can be used for eye tracking, with their positive and negative aspects, and came to the conclusion that for most applications that should come in widespread use, videooculography is most probably the best, due to being unobtrusive and fast. We have further briefly surveyed the landscape of gaze-based interaction and associated issues, and introduced gaze-contingent and attentive user interfaces in more detail. We will conclude the chapter by stating challenges and opportunities ahead, with one opportunity being addressed later in this work.

4.5 Challenges and Opportunities

4.5.1 Gaze-based object tracking

At the end of Chapter 2, *Introduction to Visual Processing*, in Section 2.10.2, *Object tracking with support of the visual system* we have already introduced the idea to utilise the power of the human visual system to follow objects for the purpose of tracking them. Such a system would fall into the category of gaze-contingent interaction. Furthermore, it would go beyond what is proposed in e.g. [Bruder et al., 2019], in the sense that it would need to handle multi-timepoint images on-the-fly, and combine it with an extension of the Radial Pursuit technique presented in [Piumsomboon et al., 2017]. See Chapter 11, *Bionic Tracking: Using Eye Tracking for Cell Tracking* for details.

4.5.2 Robust Pupil Segmentation

If eye tracking were to become a widely-used and reliable input modality, the first step, the detection of the pupil from a video stream, would need to be highly reliable.

While beyond the scope of this work, we have started preliminary work on automatic realtime segmentation of pupil images using genetic algorithms. This work was motivated by the algorithm for pupil segmentation proposed in [Kassner et al., 2014] and used in the Pupil software failing in the case of pupil images larger than 640x480 pixels, as it relies on Canny edge detection, which produces way too many false positives at high resolution.

Part II:

VR and AR for Systems Biology

> What the computer in virtual reality enables us to do is to recalibrate ourselves so that we can start seeing those pieces of information that are invisible to us but have become important for us to understand.
>
> —Douglas Adams

Chapter 5:

scenery — VR/AR for Systems Biology

The work presented in this part has been partially published in:

Günther, U., Pietzsch, T., Gupta, A., Harrington, K.I.S., Tomancak, P., Gumhold, S., and Sbalzarini, I.F.: scenery: Flexible Virtual Reality Visualization on the Java VM. *IEEE VIS*, Vancouver, 2019. arXiv preprint 1906.06726, DOI 10.1109/VISUAL.2019.8933605.

In the chapters before, we have highlighted the needs of systems biology for flexible ways of harnessing human-computer interaction, high-fidelity, customisable visualisations, and reproducibility.

In order to address these needs, we have chosen to develop our own visualisation framework: *scenery*, enabling prototyping and the delivery of multimodal, customisable, and interactive scientific visualisations, running on top of the Java Virtual Machine (JavaVM/JVM). scenery can be used on both desktop machines, and on distributed setups, such as the ones commonly used for CAVE systems or Powerwalls.

In this chapter, we are going to introduce the framework, starting with the development of ClearVolume [Royer et al., 2015], which later ignited the development of scenery. Subsequently, we outline the exact design goals and decisions made along the way, and compare scenery to existing frameworks and related works, followed by a high-level description of its components.

After this chapter, we will introduce scenery's subsystems in more detail.

5.1 ClearVolume

The work presented in this section has been developed in collaboration with Loïc Royer, Martin Weigert, Nicola Maghelli, and Florian Jug, Myers Lab, MPI-CBG, and has been published in:

Royer, L.A., Weigert, M., **Günther, U.**, Maghelli, N., Jug, F., Sbalzarini, I.F. and Myers, E.W.: ClearVolume: open-source live 3D visualization for light-sheet microscopy. *Nature Methods*, 2015.

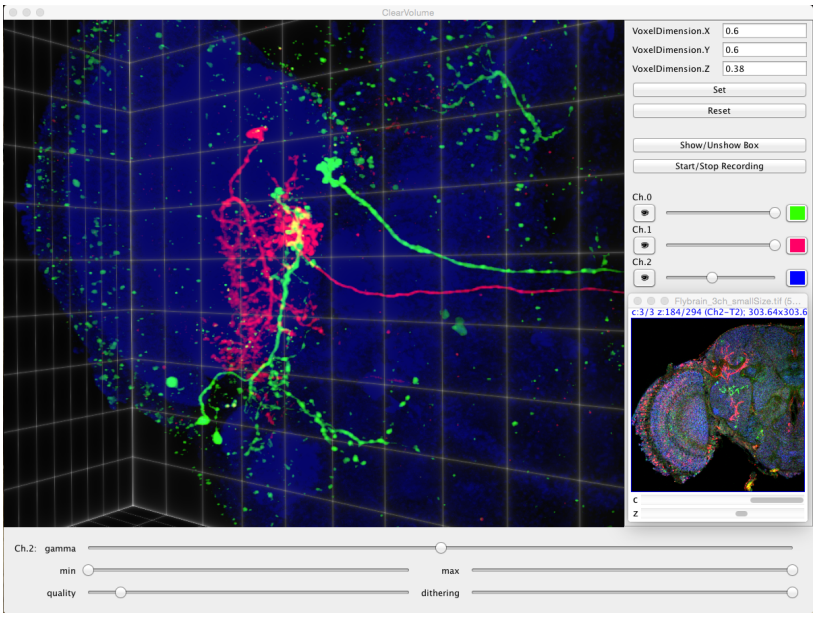

Figure 5.1: ClearVolume running inside Fiji, showing a multicolour *Drosophila melanogaster* brain dataset (courtesy of Tsumin Lee, Howard Hughes Medical Institute, Janelia Farm Research Campus), with a by-slice viewer inset. Reused from [Royer et al., 2015].

ClearVolume is a visualisation library enabling live, realtime visualisation of light-sheet microscopy data, with the capability of being integrated directly into an existing microscopy setup — ClearVolume can be used in conjunction with commonly-used microscopy control software, like MicroManager[Edelstein et al., 2010] or National Instruments LabVIEW.

The discerning features of ClearVolume are:

- *Local and remote visualisation* — the data acquired on the microscope can be visualised right on the instrument's control computer, or on a remote machine, with the data transferred over the network, albeit uncompressed. Especially when working with genetically modified organism, where the instrument has to be located in an access-controlled S1 or S2 area, remote viewing proved to be a practical tool to check on an experiment's progress, or on specimen health.
- *Source/sink architecture* — the data acquired in ClearVolume can be sent through a processing pipeline on the GPU, with the visualisation part at the end of the pipeline, and a number of processing steps before. These processing steps can include, e.g., image sharpness measurement, sample drift measurement, or Lucy-Richardson deconvolution.
- *Multipass maximum projection* (developed by Martin Weigert) — rendering large datasets in full resolution can be quite taxing on GPUs. To alleviate this problem, we have developed a new way of sampling along a traced ray, based on low-discrepancy sequences (such as the Fibonacci sequence). When multipass maximum projection rendering is active, the first samples along the ray are taken very coarsely, while subsequent samples are placed in a way to fill "holes" along the ray most efficiently, yielding a significant speedup (see Figure 5.2 for a sketch of the principle).

- *Sample tracking* (developed by the author) — utilising the source/sink architecture for data, we developed a simple center-of-mass tracking algorithm that stabilises the sample in the center of the user's field of view. The tracking can be enabled or disabled at any point in time (see Figure 5.3c).
- *Image quality measurement* (developed by Martin Weigert and Loïc Royer) — again utilising the source/sink architecture for data, we added a flexible evaluator of image sharpness to the system, where in the default settings the Tenengrad image sharpness measure is used (see Figure 5.3c), and
- *Fiji & KNIME integration* (developed by Florian Jug) — as ClearVolume supports visualising volumetric data from a microscope or any other source, we have developed plugins for Fiji[Schindelin et al., 2012] and KNIME, in addition to the integration with MicroManager[Edelstein et al., 2010] and National Instruments LabVIEW (see Figure 5.1).

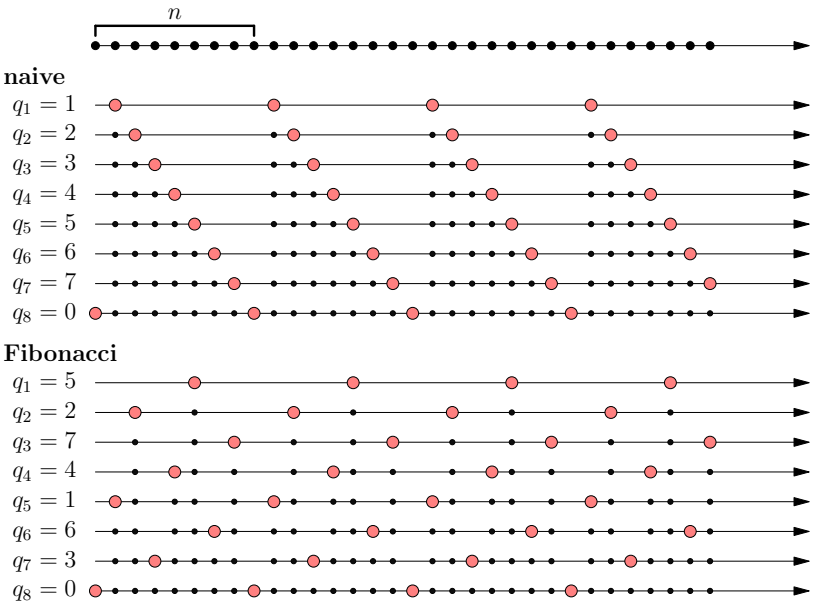

Figure 5.2: Multipass maximum projection — In the naive approach, consecutive samples along a ray are taken in single-step increments. With low-discrepancy sampling based on the Fibonacci sequence, not-yet sampled intervals along the ray are filled in most efficiently. In the figure, consecutive samples are shown top-to-bottom, with the current sample being highlighted in red. Reused from [Royer et al., 2015].

In the time since its publication, ClearVolume has proven to be a useful tool for a multitude of use cases, such as microscope development and usage [Royer et al., 2016, Kumar et al., 2018], method development for live imaging [Boothe et al., 2017], and the visualisation of correlative light microscopy/electron microscopy data [Russell et al., 2016].

However, ClearVolume only supports the visualisation of a single — although multicolour — registered volumetric dataset, which moreover needs to fit into GPU memory. With the ongoing foray of more complex computational methods into systems biology and imaging, however, volumetric data is not the only kind of data that needs to be visualised: segmentations, graph data, textual information, etc. need to be supported as well. In addition, infrastructure to support volumetric data that does not fit into the GPU memory fully needs to be supported. This in turn requires different rendering architecture, supporting complex scenes, filled with volumetric, geometric, and custom kinds of data. Furthermore, we also recognised the need for integration of various interaction hardware, such as head-mounted displays (HMDs)

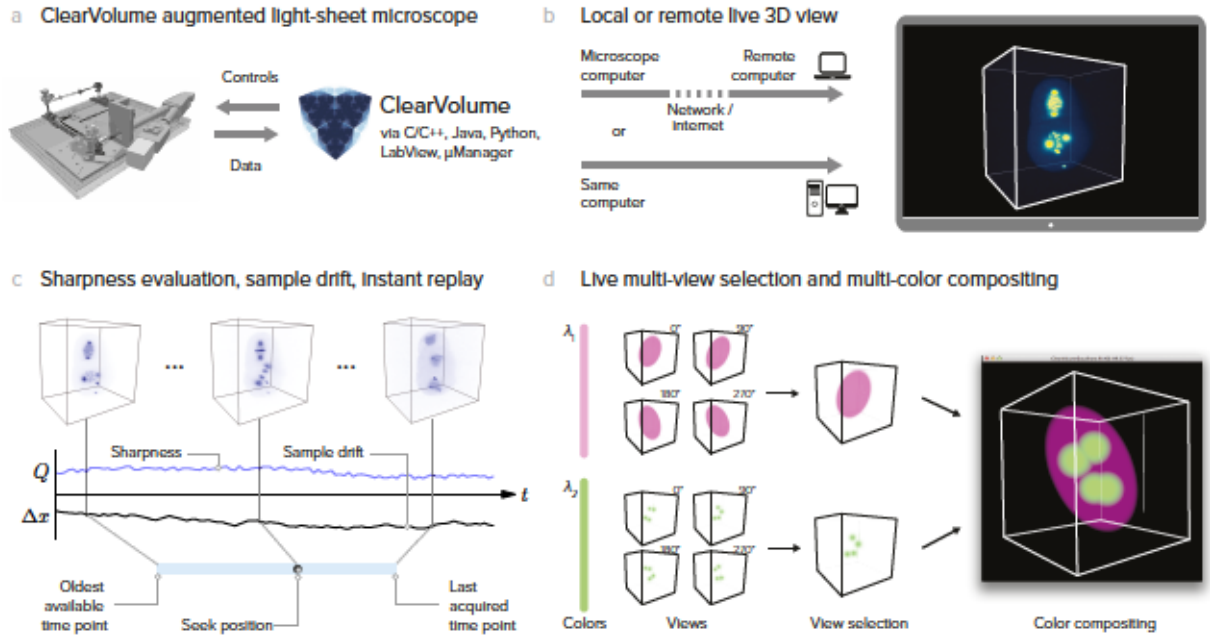

Figure 5.3: a Data flow in a ClearVolume-augmented microscopy application, b Local or remote visualisation using ClearVolume, c Evaluation of data fitness/sharpness and drift correction applied, d Multi-colour compositing. Reused from [Royer et al., 2015].

for Virtual Reality or gesture interfaces. The integration of such is possible with ClearVolume, but needs to happen on a lower level, with a framework designed around such input and output modalities.

The development of scenery started out of these particular needs.

5.2 Design Goals

From ClearVolume, we have kept the following goals for the development of scenery:

- **Free and open source software** — the resulting framework should be open-source such that the users, such as scientists, can dive into the code and see both what a particular algorithm does and/or modify it to their needs. Free and open-source software also assures the reproducibility of results over a longer period of time than closed-source software does.
- **Volume rendering** — fluorescence microscopy produces volumetric data, so the software package has to support both single-timepoint 3D volumetric images (as are also created in CT and MRI scans), and time series of 3D images, such as developmental time lapse images.
- **Extensible** — the software has to be easily extensible, such that it can be adapted to current and future needs easily, both from hardware and software side.
- **JavaVM/Fiji integration** — Fiji[Schindelin et al., 2012] is a free and open-source image analysis software package running on the JavaVM, which is used by thousands of biologists world-wide. To enable easy adoption and present the user with a familiar ecosystem, the software has to integrate with this particular ecosystem easily. For that, the software has either to be written to target the Java Runtime Environment, or be able to be called from that.

In light of the needs we have identified in the previous section, we have added the following design goals for scenery that go beyond the capabilities of ClearVolume:

- **Mesh rendering** — segmentation results or simulation results are also often produced in a mesh format, so the software also has to support these. Furthermore, localisation microscopy produces collocation points, which can also be interpreted as vertices and rendered as such.
- **Out of core volume rendering** — the size of volumetric data produced in imaging experiments or simulations is constantly growing, while e.g. bandwidth — be it network or memory bandwidth — and graphics memory do not grow at the same rate. As a result, the software has to support datasets that do not fit into the GPU memory anymore.
- **Distributed Rendering** — the software needs to be able to run on multiple machines in order to distribute the rendering workload, and synchronise scene content, e.g. for running on Powerwall systems[Woodward et al., 2001, Papadopoulos et al., 2015] or CAVE systems[Cruz-Neira et al., 1992].
- **Cross Reality** — the software has to support both virtual and augmented reality rendering modalities, such as head-mounted displays (HMDs).
- **Vulkan/OpenGL support** — To harness the power of current GPUs, the software should support the state-of-the-art rendering APIs Vulkan, and also provide a fallback to OpenGL 4.0+ in case Vulkan is not supported on the platform[1].

[1] A higher OpenGL version than 4.1 would also be desirable, but macOS only supports up to 4.1, unfortunately. OpenGL, together with OpenCL, is actually deprecated since macOS 10.14 *Mojave*. As Apple does not natively support the Vulkan API, we are going to use a wrapper, *MoltenVK*, that translates Vulkan calls to Apple's proprietary Metal API to support future development on the macOS platform.

5.3 State of the Art

Table 5.1 shows a comparison of scenery with other state-of-the-art software packages in terms of our design goals.

We find that none of the existing software packages satisfy our design goals fully, though *VTK* comes closest. *VTK* is widely used and stable, and actually provides wrapper code for use from the Java VM. The wrappers are, however, difficult to build and maintain. VTK also does not offer straightforward modifications of the rendering code. VTK's rendering routines have been updated recently to use OpenGL 2.1 [Hanwell et al., 2015], but that OpenGL version does still enable the use of modern technologies like compute shaders to fully exploit of current GPUs.

The CAVE development libraries *Vizard* and *CAVElib* offer out-of-the-box virtual reality support, but are not able to render volumetric data, nor are they extensible enough to add such functionality as a plugin. They also do not integrate with the Fiji ecosystem and the Java VM and are closed-source software, with significant license costs for all users.

The game engines *Unity* and *Unreal* in turn offer a wide variety of plugin-based extensions, albeit most of them are neither free nor open-source software. Both can be extended enough to facilitate volume rendering, but out-of-core volume rendering has not been shown yet. Both also support virtual reality and distributed rendering, although distributed rendering is an experimental feature at the time of writing. They also do not offer integration into the Fiji ecosystem or the Java VM.

Vaa3D and it's extension, *TeraFly*, does offer plugin-based extensibility, rendering of volumetric data — even out of core — but falls short on the virtual reality and distributed rendering support. Vaa3D also does not offer Fiji/Java VM integration, although there exists a Fiji plugin that can import Vaa3D data.

The commercial big data microscopy packages, *Amira*, *Arivis*, and *Imaris* all offer similar feature sets, with support for out-of-core volume rendering, mesh data support, even virtual reality. With the exception of Imaris, they use older graphics APIs than DirectX12/Vulkan or OpenGL 4.x and do not support distributed rendering setups. Imaris also offers Fiji integration, which the other two do not. However, non of them are open-source software and they do not support changes to their rendering routines.

Zeiss ZEN and is Zeiss' default microscopy acquisition software. It supports volumetric data and rendering of it, although not out-of-core. Since recently, ZEN also offers plugin-based extensibility, but is not open-source software and does not integrate with Fiji or the Java VM.

Finally, *BigDataViewer* has introduced an open-source architecture for access to large image datasets that do not fit into the main memory of a given machine, and are loaded on-the-fly from hard drives or even over the network. BigDataViewer supports by-slice rendering of volumetric data (thereby excluding VR rendering), but no meshes, and is tightly integrated with Fiji. We are going to use BigDataViewer's technology to implement out-of-core volume rendering in scenery.

Table 5.1: scenery compared to other software packages.

Software	Type	Free/open-source	Volumes	Out-of-core rendering	Meshes	Distr. Rendering	VR	Extensible	Cross-platform	OGL 4.1/D3D12/Vulkan	Fiji integration
Amira	Big data, Microscopy	-	•	•	•	-	•	-	-	-	-
Arivis	Big data, Microscopy	-	•	•	•	-	•	•	-	-	-
BigDataViewer [Pietzsch et al., 2015]	Big data, Microscopy	•	•	•	-	-	-	•	•	-	•
Imaris	Big data, Microscopy	-	•	•	•	-	•	•	-	•	•
Vaa3D [Bria et al., 2016]	Big data, Microscopy	•	•	•	•	-	-	•	•	•	-
ClearVolume [Royer et al., 2015]	Microscopy	•	•	-	-	-	-	•	•	•	•
Zeiss ZEN	Microscopy	-	•	-	-	-	-	•	-	-	-
Unity	Game engine	-	-	-	•	•	•	•	•	•	-
Unreal Engine	Game engine	-	-	-	•	•	•	•	•	•	-
Vizard	CAVE software	-	-	-	•	•	•	•	-	•	-
Mechdyne CAVELib	CAVE software	-	-	-	•	•	•	•	•	•	-
VTK [Hanwell et al., 2015]	Scivis engine	•	•	-	•	•	•	•	•	-	-
scenery	Scivis engine	•	•	•	•	•	•	•	•	•	•

Of the mentioned software packages, Amira, Arivis, Imaris, ZEN, Vizard, and CAVELib only run on a subset of our list of target platforms (Windows, macOS, Linux).

5.4 Implementation Challenges

5.4.1 Language

The first implementation challenge is finding the right programmings language. The obvious choice for graphics applications would be C/C++, or newer/safer alternatives, such as Rust[2].

[2] Rust is a new multi-paradigm programming language focused on memory safety, see rustlang.org.

One of our goals, however, is tight integration with the ImageJ ecosystem, which is targeting the JavaVM. While the JavaVM itself does not have the reputation of being a harbour for high-performance applications, which usually resort to C/C++, it offers a less steep learning curve for new users, and extremely well-designed abstractions, e.g. for multithreading between the different platforms we are targeting, making platform-specific code unnecessary in most cases.

We additionally view our framework as an interesting experiment on how to bring high-performance graphics to the JavaVM — a task that should now not be as difficult as before anymore, owing to pipelined graphics/compute APIs such as OpenCL, CUDA, and Vulkan.

We aim for excellent interoperability with the ImageJ ecosystem, yet want to use a functional language to make it easy to reason about the code, and avoid the large amounts of boilerplate code usually necessary in pure-Java projects. Within the realm of JavaVM-based languages, the contenders[3] therefore are

[3] As of early 2016, when the project was started. Since then, other languages, such as Ceylon, have matured a lot — and would probably now be considered as well.

- *Clojure*, a Lisp dialect,
- *Scala*, a functional and object-oriented language developed at *EPFL*,
- *Kotlin*, a language using both functional and object-oriented paradigms by the company JetBrains.

Clojure unfortunately has a very steep learning curve, resulting in a small pool of experts, and *Scala* makes usage from existing Java code difficult. *Kotlin* on the other hand provides a useful functional feature set, as well as low entry barriers, and a well developed, open-source and commercially maintained IDE [4]. Kotlin is additionally useable as a scripting language (see https://github.com/hbrandl/kscript). Since 2017, Kotlin is also a first-class citizen on the Android platform, which has since significantly boosted its popularity, making it one of the most popular newcomer languages, as determined by the PyPL index (*P*opularit*Y* of *P*rogramming *L*anguages, pypl.github.io).

[4] see kotlinlang.org.

Finally, returning to our previous remark about native languages vs. the JavaVM: In addition to targeting the JavaVM, *Kotlin* also offers to transpilation to JavaScript and to actual native code (via LLVM [Lattner and Adve, 2004]), making our design choice more future-proof, in the case that the JavaVM should at some day present

unsurmountable issues, or should we decide to include the web browser in our list of deployment targets.

5.4.2 Graphics APIs

The graphics API is the API that makes talks to the GPU. Historically — and before graphics cards were actually called GPUs — the developer would manipulate the geometry to be shown on screen on the CPU, and the graphics card executes hard-wired in-silicon graphics functions that could not be changed. OpenGL is an example for an API that originated in this era, back when graphics cards did not offer *actual* programmability, but would just execute a fixed set of instructions quicker than a regular processor could.

OpenGL has originally been developed by Silicon Graphics (SGI) in the early 1990's for their graphics workstations. Originally intended for CAD/CAM applications, OpenGL soon became the go-to API for developing cross-platform graphics applications and games.

Since the early days of OpenGL, and also after further development had been handed over to the Khronos Group, OpenGL's development model has been based around the definition of a standard, augmented by extensions approved by the Khronos Group's Architecture Review Board (ARB). New versions of the standard were then essentially made out of sets of mandatory ARB extensions a GPU would have to support to be declared compatible with OpenGL x.y.

Shader-based rendering, and with that, more programmability of the graphics pipeline, has been introduced into OpenGL fully with OpenGL 2.1, with further extensions made with 3.3 and 4.1. What has not changed since the beginning though, is the basic programming model, where the programmer modifies a global state, and render objects either directly (in the ancient days of OpenGL 1.x), or with the help of vertex buffer objects (VBOs). This programming model mapped very well to the early, fixed-function graphics cards.

Unfortunately, this programming model does not map well to current-generation GPUs, which have become massively parallel computing machines, with some sporting 4000 individual cores and more. These cores do not have the flexibility of regular CPUs, but eclipse them easily in specialty disciplines, such as the floating-point computations needed for matrix and vector math operations heavily used in both graphics and machine learning.

In 2016, the Khronos Group has published a new graphics API named *Vulkan* solving the issues with OpenGL, and essentially starting with a clean slate. Vulkan provides a much deeper, *close-to-metal* access to the GPU than OpenGL does, with the main differences being:

- More verbose code with less checks done by the driver, leading to more clarity about what is done, when, and how. Now, however, the developer needs to take greater care in adhering to the specification, as it clearly states that the driver is allowed to crash an application in case it is behaving out-of-specification.

- Better options for validation on the developer side. While the driver does not do many checks anymore, they can be activated on the application side by the *Validation Layers*, which can impact performance quite badly, but provide highly detailed information about where and when a problem occured, giving the developer quicker insight how to fix that problem.
- Possible higher performance by caching command buffers containing rendering commands, instead of scene iteration and draw call issuing with every frame. Command buffers can also be created by multiple threads in parallel, but need to be submitted serially.
- Resources, such as textures and buffers, and their descriptors, have to be allocated and managed on a much more fine-grained level than with OpenGL, but can be accessed directly, and do not need to be bound to texture units like in OpenGL.
- Vendor-independent (Homogeneous) Multi-GPU support (since Vulkan 1.1).
- A Conformance Test Suite (CTS) for the graphics drivers, ensuring that a Vulkan-based application behaves identically on all drivers.
- Shaders are not loaded from OpenGL Shading Language (GLSL) text files, but compiled SPIR-V byte code, with interesting new possibilities for introspection and reflection, e.g. via the tool/library *spirv-cross*.

From this list, it is clear that with great power comes great responsibility — with more effort required by the developer when writing Vulkan code, although that typically is more than compensated by better performance and much better debugging options.

During early development, we recognised that the OpenGL and Vulkan APIs do not map very well onto each other, as Vulkan operates on a much lower level. For this reason, scenery has been written with flexibility regarding the rendering backend in mind: It should be easy for the developer to replace one of the existing rendering backends with an entirely new one.

scenery now includes both an OpenGL 4.1 backend for use on the macOS operating system, and a Vulkan backend for use on Windows and Linux. The contract between the core library and the rendering backend is thin, making it very easy to create new rendering backends, with offline rendering via an external raytracing software, such as *Embree*[Wald et al., 2014], *OSPray*[Wald et al., 2017] or *OptiX*[Parker et al., 2013] being a future possibility.

5.4.3 Interfacing with Graphics API on the Java VM

Current versions (8.0+) of the Java Virtual Machine do not provide Java-native bindings to graphics APIs like Direct3D, OpenGL or Vulkan, but 3rd-party libraries exist to bridge this gap. We have chosen two projects for developing the OpenGL and Vulkan backends of scenery:

- *JOGL* (www.jogamp.org) provides an object-oriented Java interface to the OpenGL API in all of its current versions. In that, JOGL's interfaces are written in a very idiomatic way, which partially diverge quite far from the original OpenGL C API. They do, however, simplify the use for the developed software,

especially in situations where it has to be embedded into an existing GUI application. JOGL is used for the OpenGL backend in scenery.

- *LWJGL3* (www.lwjgl.org) provides a Java interface to the OpenGL and Vulkan APIs in all of their current versions. In contrast to JOGL, LWJGL3 keeps its API very close to the original, and does not wrap a normal C API in an object-oriented manner, but leaves that aspect to the developer, in case desired. This approach results in more flexibility, at the cost of higher effort for embedding into existing applications. LWJGL3 is used to develop the Vulkan backend in scenery. Memory management for Vulkan-related structures is mostly manual, but off-heap: Off-heap memory is memory that is not managed by the JavaVM. In the case of LWJGL3, on Windows `malloc()` is used for allocations, while on Linux and macOS, the high-performance memory allocator *jemalloc* (jemalloc.net](http://-jemalloc.net)) is used. Both allocators provide a better alternative to using Java's Direct Byte Buffers that incur a large performance penalty over raw `malloc()` calls, and are also a sparse resource, available only in a size determined at program startup. In LWJGL3, a thread-local memory stack is provided in addition, enabling high-speed temporary allocations. For details about memory allocation strategies in LWJGL3, see blog.lwjgl.org/memory-management-in-lwjgl-3/. In short, management of that memory is up to the developer.

Unfortunately, JOGL is not actively maintained anymore, and we aim to fully switch to the LWJGL3-powered Vulkan backend in the future.

5.5 Component overview

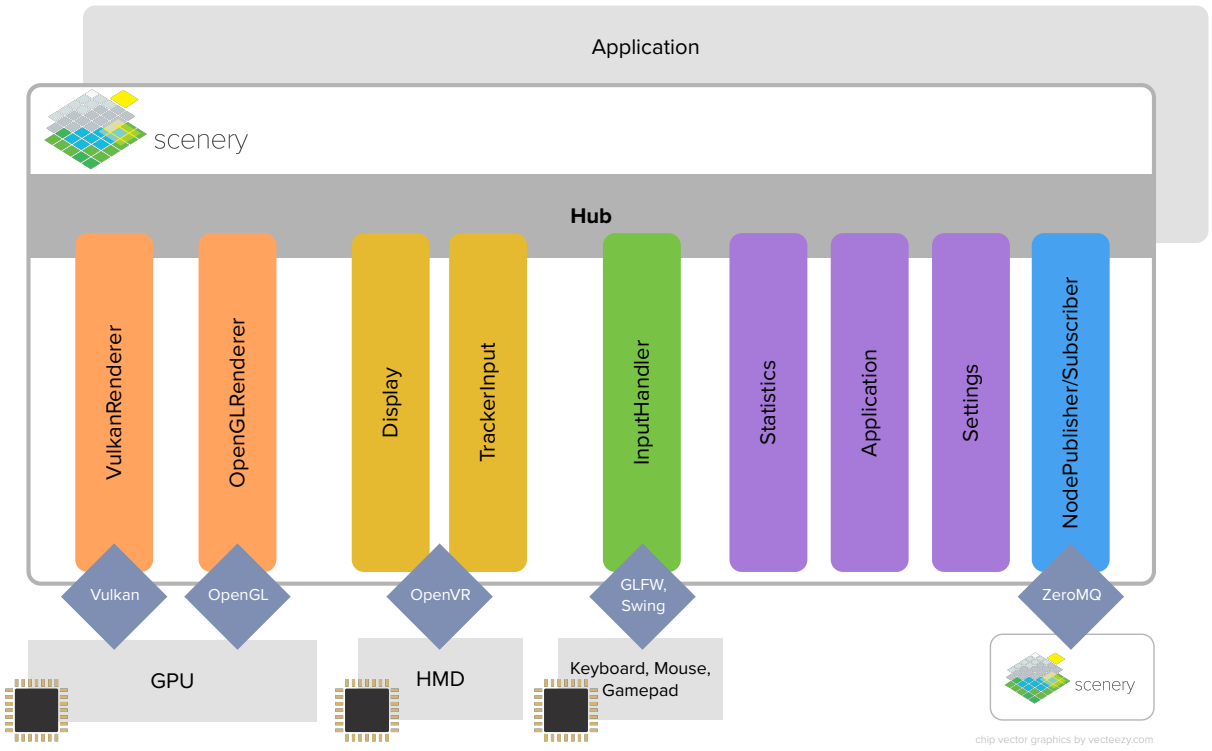

Figure 5.4: A high-level overview of scenery's components.

At the outermost architectural layer, the scenery framework consists of seven main software components:

- the *Scene*, represents the content that needs to be visualised as directed acyclic graph (scenegraph),
- the *Renderer*, taking care of the visual representation of the scene's contents,
- the *Input Handler* for responding to input events triggered by the user,
- the *External Hardware Handlers*, for handling, e.g., head-mounted displays or tracking systems,
- the *Publishers* and *Subscribers*, which track changes, e.g. to a Scene, and disseminate them to connected clients over the network in a distributed rendering setup,
- the *Hub*, tying all these systems together and allowing the system to query information about each components state, and
- the *Settings*, an instance-local database of default and user-defined settings that may also change during runtime.

The construction and interplay of all the components is handled by the class `SceneryBase`, which a user can directly subclass to create own applications.

In the next chapters, we are going to introduce the most important subsystems of scenery, namely those for rendering, input handling and integration of external hardware, and for distributed rendering. These chapters are followed by a brief description of the remaining subsystems in scenery. We conclude the description of scenery with Chapter 10, *Future Development Directions*.

5.6 Software Availability

scenery is available as free and open-source software under the LGPL 3.0 license at github.com/scenerygraphics/scenery.

Chapter 6:

Rendering

After having introduced a high-level overview of scenery in the previous chapter, in this chapter, we are diving deeper into the framework, describing each of its components. We start with scenegraph-based rendering and traversal:

6.1 Scenegraph-based rendering

A scenegraph is a data structure that organises objects in a hierarchical manner, in a tree or graph structure, where each node in the graph has its own transformation properties. In that way, it is very easy to describe spatial and organisational relations between objects, such as a car tyre belonging to a car, and the tyre moving with the parent object *car* (inheriting its transformations), when it moves.

In general, a scenegraph can contain connections between multiple nodes. In scenery, we use a scenegraph approach that is closer to a tree representation, because that structure more easily enables parallel scene element discovery [Boudier and Kubisch, 2015].

If bounding boxes are stored along with the nodes, the scenegraph can easily be extended to also include *bounding volume hierarchies* that can enable more efficient collision detection[1] or frustum culling[2].

[1] For general collision detection without additional acceleration data structures like bounding volume hierarchies, the whole scenegraph has to be traversed for each object that is checked for collisions, leading to a $O(n^2)$ complexity. In BVHs, objects are guaranteed not to overlap, if their parent bounding volumes to not overlap, which can greatly lower the candidate pool, speeding collision detection up tremendously.

[2] Frustum culling is the process of determining the objects that are currently in the camera's view frustum, and rendering only those. Acceleration data structures can help with determining the inside set in the same manner as in collusion detection.

6.1.1 Traversal

Scenegraphs can be traversed in a variety of ways, such as depth-first traversal. Depth-first traversal is also used by default in scenery. The renderer can make further optimisations, such as drawing it in front-to-back order, where spatial sorting is applied after gathering nodes, e.g. to draw transparent objects in the correct way.

The exception to normal scenegraph traversal is *instancing* (described in detail in Section 6.2.7, *Instancing*), where copies of a node are not added as children, but to the special `instances` property as performance optimisation. See the section on instancing for details.

6.1.2 The Nodes

Nodes are the elements of the scenegraph, and most of them are entities that can be rendered on-screen. These nodes are organised into an ordered, acyclic graph of parent-child relationships. Operations, such as mathematical transformations — such as translations, rotations, or scalings — are automatically propagated to a `Node`s `children`.

For actually organising nodes into a scene, a special node type exists, the `Scene`. `Node`s attached to a top-level `Scene` element as `children` become the top-level elements of the scene, and can in turn have their own `children`.

As a short example, a `Scene` might contain a `Node` *Cell*, which has children *Nucleus* and *ER*. The transformations of *Nucleus* and *ER* are then relative to *Cell*, and when the *Cell* moves or rotates, so will *Nucleus* and *ER*.

6.1.3 Transforms

A `Node` can have the following transforms:

- `position` – the position of the `Node` in 3D space
- `scale` – scaling along the X, Y, and Z axis
- `rotation` – a quaternion[3] describing the `Node`s orientation in space.
- `model` – local transforms how this `Node` is positioned with respect so its parent
- `world` – global transforms that include the `model` transform, as well as the `parent`s `world` transform

The transforms are calculated by a `Node`s `updateWorld(recursive: Boolean, force: Boolean)` routine, and stored to the `model` and `world` properties. If `recursive` is specified, the routine will descend to all children, calculate their transforms, and mark them as updated. If any of the transforms of a `Node` change during runtime, it will get its `needsUpdate` and `needsUpdateWorld` flag set, to be picked up on the next update run.

Some nodes might want to construct their own model and world matrices, overriding the behaviour of `updateWorld`: this can be achieved by setting the `wantsComposeModel` flag to `false`.

[3] Quaternions are a 4-dimensional extension of complex numbers, that can describe rotations in space. While rotations may as well be represented as matrices, such representations suffer from two problems: a) matrices cannot be smoothly interpolated and b) they may lead to *gimbal locking*, where the sine or cosine of an angle in a rotation matrix lead to a zero entry, making the transformation loose a degree of freedom — further multiplications will not be able to achieve a non-zero rotation around this axis. Quaternions do not suffer from both problems, they are however not as intuitive as other rotation representations such as Euler angles. In scenery, helper routines are provided to convert Euler angles or matrices to quaternions for user convenience.

6.2 The Rendering Procedure in scenery

In scenery, the contract with the renderer is thin. A renderer needs to:

- be able to render something (`render()` function),
- initialize a scene (`initializeScene()` function),
- take screenshots (`screenshot()` function),
- resize a (eventually existing) viewport (`reshape()` function),
- become embedded inside an existing window, e.g. from JavaFX or Swing (`embedIn` property),
- change the rendering quality (`setRenderingQuality()` function),

- toggle push mode (`pushMode` property, also see Section 6.2.3, *Push Mode*),
- hold settings (`settings` property),
- hold a window (`window` property), and
- close itself.

This design decision was made such that scenery can support a variety of different rendering backends, after discovering that OpenGL — which scenery started with as only renderer — and Vulkan do not map well to each other. With this architecture we are more easily able to extend rendering support in the future to e.g. software renderers, or external ray tracing frameworks.

The Renderer interface in scenery is defined as:

```
abstract class Renderer : Hubable {
    abstract fun initializeScene()
    abstract fun render()
    abstract var shouldClose: Boolean
    abstract var initialized: Boolean
        protected set
    abstract var settings: Settings
    abstract var window: SceneryWindow
    abstract var embedIn: SceneryPanel?
    abstract fun close()
    fun screenshot() {
        screenshot("")
    }

    abstract fun screenshot(filename: String = "")
    abstract fun reshape(newWidth: Int, newHeight: Int)
    abstract fun setRenderingQuality(quality: RenderConfigReader.↩
     RenderingQuality)
    abstract var pushMode: Boolean
    abstract val managesRenderLoop: Boolean

    abstract var lastFrameTime: Float
    abstract var renderConfigFile: String

    // more functions follow here
    ...
}
```

Listing 6.1: Renderer interface definition.

A renderer may also run in its own thread, but must indicate that properly by setting `managesRenderLoop`, as e.g. done by the OpenGL renderer. Otherwise, the renderer will run synchronous with scenery's main loop.

A renderer may store its own metadata related to a `Node` in its `metadata` field. This field is cleared upon the removal of a `Node` from the scene. The `metadata` must be uniquely named, such that renderers — which could be running in parallel — do not interfere with each other's `metadata`.

At the time of writing, scenery includes a high-performance Vulkan renderer, used by default on Windows and Linux, and an OpenGL renderer, used by default on macOS.

6.2.1 Windowing Systems

On the Java VM, the main options for drawing (to) windows and related elements are AWT, Swing, and JavaFX. In scenery, the currently supported mode for embedding into existing windows is Swing, which is supported by both the OpenGLRenderer and the VulkanRenderer.

In development, we had originally started out with using JavaFX, the most modern UI toolkit for the Java VM. JavaFX however has the problem that it works entirely GPU-based, but unfortunately does not allow the developer to directly access GPU resources, keeping them hidden and inaccessible. The consequence: Images already generated on the GPU have to be transferred to main memory, and are then again uploaded to the GPU as a texture by JavaFX's internal mechanisms. As bandwidth is usually a limited resource, this leads to severe performance issues, especially when rendering at higher resolutions beyond full HD (1920x1080). Recently, a new project named *DriftFX* [4] appeared, which aims to solve this issue by introducing operating system-native OpenGL rendering surfaces into JavaFX, circumventing the double-transfer issue just described. Our hope is we can harness this project in the future, and eventually extend it to support Vulkan as well.

[4] The DriftFX project (github.com/eclipse-efx/efxclipse-drift/) provides an extension for JavaFX to allow direct use of OpenGL. As this project appeared only very recently, we have not yet evaluated it.

6.2.2 Uniform Buffer Serialisation and Updates

Uniforms are properties that are communicated to the shader program, and are very similar to C-style structs. Before the advent of OpenGL 4.x and Vulkan, single uniform variables had to be stored and updated individually, leading to a large API overhead. Since OpenGL 4.x and Vulkan, Uniform Buffer Objects (UBOs) are available — instead of storing and updating single uniforms, UBOs are buffers of uniforms, which are updated together, and sent to the GPU. In contrast to single uniforms, this provides two main benefits:

- less API overhead by being able to serialise many uniforms into a single buffer, and uploading them to the GPU in one API call, and
- multiple-rate updates of different UBOs — before, uniforms had to be set every time they were used in a shader, now they can be updated only when needed, and for different UBOs, this can mean different update rates, further reducing the original overhead.

In scenery, UBOs belonging to a Node are tentatively updated with each frame before the main rendering loop, guaranteeing that a `Node` that has been added to the scene graph will have its transformations and properties ready at render time.

Serialisation of a `Node`'s transformations and properties are handled by the class `UBO`. In that class, member variables of the UBO are stored as a `LinkedHashMap` of a string (for the property name), and a lambda of type `() -> Any` for the value of the property. This mechanism enables determining values of properties that change during runtime, without rewriting the contents of the map. Order in the struct does matter, which is why a linked map is being used. The actual order of the properties is determined via shader introspection. For common data types (floats, doubles,

integers, shorts, booleans, vectors, and matrices), `UBO` will determine the necessary size and offset of a certain property, and write the contents of the property to a `ByteBuffer`, according to OpenGL's and Vulkan's `std140` buffer rules, storing data in the same memory layout as C-style structs.

After an UBO has been serialised for the first time, a hash is calculated from its current members. Upon revisiting this `UBO`, the previous member hash is compared with the current one, to determine whether re-serialisation is necessary or not. In case re-serialisation is not necessary, the buffer backing the UBO will remain untouched and continued to be used in that state. If it needs to be re-serialised, it will also be re-uploaded to the GPU.

6.2.3 Push Mode

Push Mode is a rendering optimisation especially for viewer-type applications, where continuous scene changes are not expected. In push mode, the renderer will keep track of updated UBOs (as described in Section 6.2.2, *Uniform Buffer Serialisation and Updates*) and modified scene contents, and only render a frame in the following cases:

- an UBO has been updated,
- an object has been added or removed from the currently visible objects, or
- an object that is visible has changed it's properties (e.g. uses a different material or texture now)

Buffer swaps may however still take place, so that special care is taken to update all swapchain images before stopping to actively render. This mechanism is implemented using a JDK-provided `CountDownLatch` that starts with the number of swapchain images, and is counted down by one for each render loop pass. When the latch reaches zero, rendering is discontinued until the next update happens.

The push mode mechanism also guarantees that all updates to the scene's content are obeyed, as it is not tied to e.g. input events, which might be caused by a myriad of devices, but the actual updates of scene contents or UBOs.

6.2.4 Configurable Rendering Pipelines

scenery provides configurable rendering pipelines which can contain multiple passes over the scene's geometry, or postprocessing (fullscreen) passes.

The renderpasses are read from a YAML file. A simple example for a forward shading pipeline with HDR postprocessing can be seen in Listing 6.2: In this simple pipeline, the scene contents are rendered in a single pass into the 16bit RGBA floating point rendertarget HDR (line 5) by the `Scene` rendering pass (line 11). The HDR postprocessing is done in the `PostprocessHDR` pass (line 18), which outputs to the viewport.

```
1 name: Forward Shading
2 description: Forward Shading Pipeline, with HDR postprocessing
3
```

```
rendertargets:
  HDR:
    size: 1.0, 1.0
    attachments:
      HDRBuffer: RGBA_Float16
      ZBuffer: Depth32

renderpasses:
  Scene:
    type: geometry
    shaders:
      - "Default.vert.spv"
           - "Default.frag.spv"
        output: HDR
    PostprocessHDR:
        type: quad
        shaders:
           - "FullscreenQuad.vert.spv"
                - "HDR.frag.spv"
           inputs:
                - HDR
               output: Viewport
               parameters:
                       Gamma: 1.7
                       Exposure: 1.5
```

Listing 6.2: Simple forward shading rendering pipeline definition.

In the render config file, both *rendertargets* and *renderpasses* are defined. A *rendertarget* consists of a framebuffer name, a framebuffer size, and a set of attachments of the framebuffer that can have different data types. A *renderpass* consists of a pass name, a type – geometry or quad (for postprocessing) –, a set of default shaders, and defined *inputs* and *outputs*. The renderpass may also define a set of *shader parameters*, which are handed over to the shader via the `UBO` mechanism, and supports all the data types supported by `UBO`.

The definition must contain one renderpass that outputs to Viewport, otherwise nothing will be rendered.

From the definition in the YAML file, `RenderConfigReader` will try to form a directed acyclic graph (DAG). The resulting graph for the forward shading example in Listing 6.2 is shown in Figure 6.1.

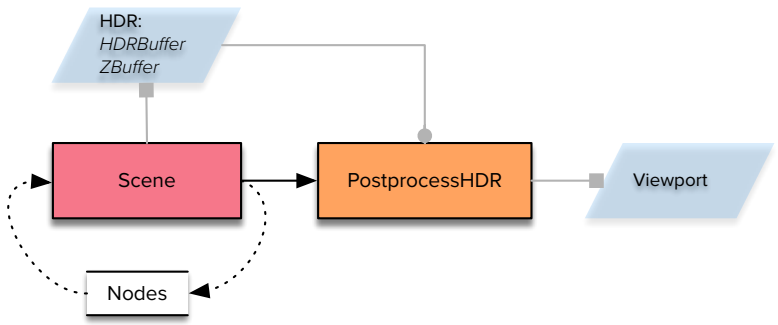

Figure 6.1: The graph representation of the ForwardShading rendering pipeline. Scene passes are shown with red background, postprocessing passes with orange background. Light blue parallelograms are framebuffers. Solid black arrows signify transition from one pass to the next, grey arrows show data dependencies, with squares standing for writes, and circles for reads. Dotted arrows show scenegraph accesses.

If a DAG cannot be formed from the given definition, `RenderConfigReader` will emit an exception.

Render configs are switchable during runtime and switching will cause the renderer to destroy and recreate its rendering framebuffers — while all other loaded textures and resources are preserved. This mechanism is e.g. used to toggle stereo rendering during runtime, and can facilitate rapid prototyping. In addition, the renderer can watch used shader files actively for changes, try to compile them, and, if compilation and linking is successful, replace them on-the-fly. To toggle this behaviour, `Renderer.watchShaders()` can be called.

```
name: Deferred Shading
description: Deferred Shading, with HDR postprocessing and FXAA
sRGB: true

rendertargets:
  GeometryBuffer:
    attachments:
      NormalsMaterial: RGBA_Float16
      DiffuseAlbedo: RGBA_UInt8
      ZBuffer: Depth32
  ForwardBuffer:
    attachments:
      Color: RGBA_Float16
  AOBuffer:
    size: 0.5, 0.5
    attachments:
      Occlusion: R_UInt8
  AOTemp1:
    size: 1.0, 1.0
    attachments:
      Occlusion: R_UInt8
  AOTemp2:
    size: 1.0, 1.0
    attachments:
      Occlusion: R_UInt8
  HDRBuffer:
    attachments:
      Color: RGBA_Float16
      Depth: Depth32
  FXAABuffer:
    attachments:
      Color: RGBA_UInt8

renderpasses:
  Scene:
    type: geometry
    renderTransparent: false
    renderOpaque: true
    shaders:
      - "DefaultDeferred.vert.spv"
      - "DefaultDeferred.frag.spv"
    output: GeometryBuffer
  AO:
    type: quad
    parameters:
      Pass.displayWidth: 0
      Pass.displayHeight: 0
      occlusionRadius: 1.0
      occlusionSamples: 4
      occlusionExponent: 2.0
      maxDistance: 1.0
      bias: 0.1
      algorithm: 0
    shaders:
      - "FullscreenQuadFrustum.vert.spv"
      - "HBAO.frag.spv"
    inputs:
      - GeometryBuffer
    output: AOTemp1
  AOBlurV:
    type: quad
    parameters:
      Pass.displayWidth: 0
      Pass.displayHeight: 0
      Direction: 1.0, 0.0
      Sharpness: 40.0
      KernelRadius: 8
    shaders:
      - "FullscreenQuad.vert.spv"
      - "HBAOBlur.frag.spv"
    inputs:
      - GeometryBuffer.ZBuffer
      - AOTemp1
    output: AOTemp2
  AOBlurH:
    type: quad
    parameters:
      Pass.displayWidth: 0
      Pass.displayHeight: 0
      Direction: 0.0, 1.0
      Sharpness: 40.0
      KernelRadius: 8
    shaders:
      - "FullscreenQuad.vert.spv"
      - "HBAOBlur.frag.spv"
    inputs:
      - GeometryBuffer.ZBuffer
      - AOTemp2
    output: AOBuffer
  DeferredLighting:
    type: lights
    renderTransparent: true
    renderOpaque: false
    depthWriteEnabled: false
    depthTestEnabled: false
    shaders:
      - "DeferredLighting.vert.spv"
      - "DeferredLighting.frag.spv"
    inputs:
      - GeometryBuffer
      - AOBuffer
    output: ForwardBuffer
    parameters:
      debugLights: 0
      reflectanceModel: 0
      Global.displayWidth: 0
      Global.displayHeight: 0
  ForwardShading:
    type: geometry
    renderTransparent: true
    renderOpaque: false
```

```
112     blitInputs: true
113     shaders:
114       - "DefaultForward.vert.spv"
115       - "DefaultForward.frag.spv"
116     inputs:
117       - ForwardBuffer.Color
118       - GeometryBuffer.ZBuffer
119     output: HDRBuffer
120   HDR:
121     type: quad
122     shaders:
123       - "FullscreenQuad.vert.spv"
124       - "HDR.frag.spv"
125     inputs:
126       - HDRBuffer.Color
127     output: FXAABuffer
128     parameters:
129       TonemappingOperator: 0
130       Gamma: 1.8
131       Exposure: 1.0
132       WhitePoint: 11.2
133   FXAA:
134     type: quad
135     shaders:
136       - "FullscreenQuad.vert.spv"
137       - "FXAA.frag.spv"
138     parameters:
139       activateFXAA: 1
140       showEdges: 0
141       lumaThreshold: 0.125
142       minLumaThreshold: 0.02
143       mulReduce: 0.125
144       minReduce: 0.0078125
145       maxSpan: 8.0
146       Global.displayWidth: 0
147       Global.displayHeight: 0
148     inputs:
149       - FXAABuffer
150     output: Viewport
151
152 qualitySettings:
153   Low:
154     AO.occlusionSamples: 0
155     FXAA.activateFXAA: 0
156     AO.shaders:
157       - "FullscreenQuadFrustum.vert.spv"
158       - "SSAO.frag.spv"
159   Medium:
160     AO.occlusionSamples: 8
161     FXAA.activateFXAA: 1
162     AO.shaders:
163       - "FullscreenQuadFrustum.vert.spv"
164       - "SSAO.frag.spv"
165   High:
166     AO.occlusionSamples: 4
167     FXAA.activateFXAA: 1
168     AO.shaders:
169       - "FullscreenQuadFrustum.vert.spv"
170       - "HBAO.frag.spv"
171   Ultra:
172     AO.occlusionSamples: 8
173     FXAA.activateFXAA: 1
174     AO.shaders:
175       - "FullscreenQuadFrustum.vert.spv"
176       - "HBAO.frag.spv"
```

Listing 6.3: Deferred Shading rendering pipeline definition, with forward shading for transparent geometry as separate step, HDR, and FXAA antialiasing as postprocessing steps.

A more complex rendering pipeline definition is shown in Listing 6.3. In this rendering pipeline configuration, we apply the following techniques:

- Deferred Shading [Deering et al., 1988], for being able to render a large number of lights by splitting geometry processing and lighting into two separate passes: for every pixel, first, surface normals (with an efficient normal storage, where 3D unit vectors are compressed into a 2D octogon [Cigolle et al., 2014]), surface material properties, and depth are stored into separate buffers in the `Scene` pass (line 35), second, the final shading of the pixel is determined from these buffers in the `DeferredLighting` pass (line 90).
- Ambient Occlusion via the HBAO algorithm [Bavoil et al., 2008] in the `AO` pass, with horizontal and vertical blurring in the `AOBlurV` and `AOBlurH` passes (lines 43, 60, and 75),
- tone-mapping of the 16bit HDR color output of the `DeferredLighting` pass in the `HDR` pass, using the ACES tone mapping operator[5] (line 120), and
- Anti-aliasing of the final image via the Fast approximate anti-aliasing algorithm [Lottes, 2009] in the `FXAA` pass (line 130).

This rendering pipeline configuration also showcases shader properties (see e.g. `Direction` or `Pass.displayWidth` in the `parameters` section of

[5] ACES, the Academy Color Encoding System, defines a particular curve for mapping from HDR to LDR color, see github.com/ampas/aces-dev/tree/v1.0.3 for details.

the `AOBlurH` pass). These are explained in more detail in the section [Shader Introspection and Shader Properties].

This more complex rendering pipeline can also be represented as a graph, as shown in Figure 6.2.

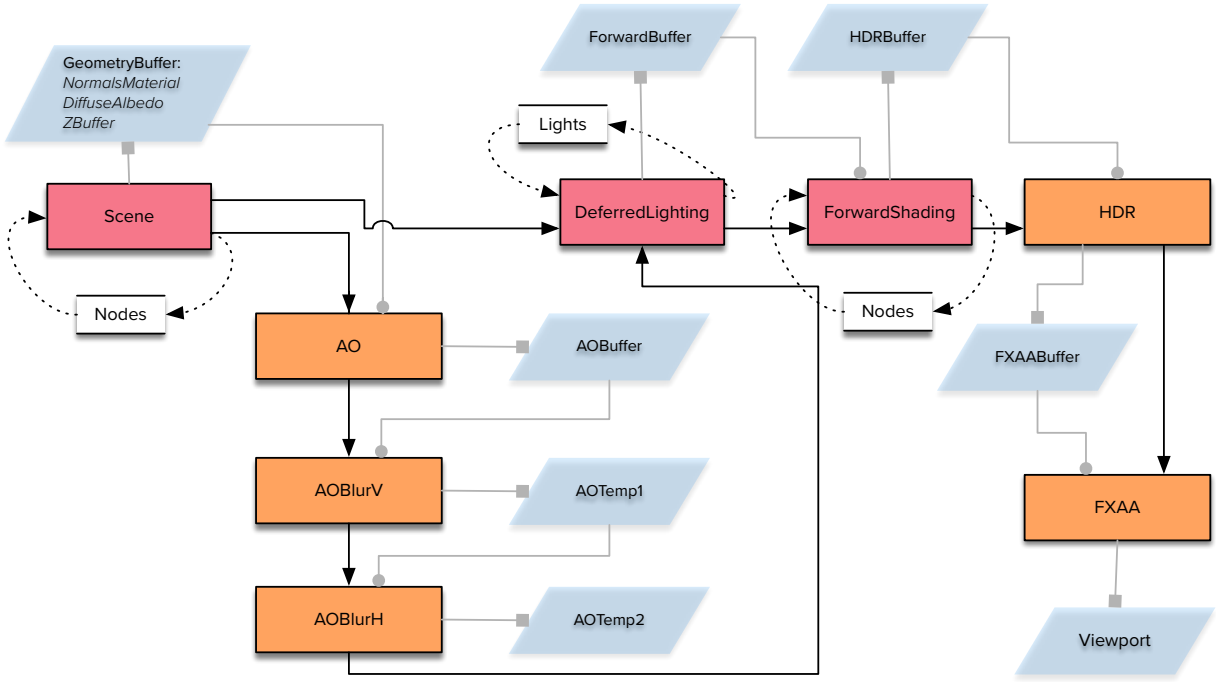

Figure 6.2: The graph representation of the DeferredShading rendering pipeline. Scene passes are shown with red background, post-processing passes with orange background. Light blue parallelograms are framebuffers. Solid black arrows signify transition from one pass to the next, grey arrows show data dependencies, with squares standing for writes, and circles for reads. Dotted arrows show scenegraph accesses.

6.2.5 General Rendering workflow

In scenery, scene object discovery (determining which objects are to be rendered), and updating the UBO's contents are done in parallel using Kotlin's coroutines.

The main rendering loop will proceed after the visible objects have been determined:

The main loop then proceeds as follows:

1. Loop through the flow of renderpasses, except Viewport pass:
   a. determine the kind of pass
   b. bind framebuffers for output
   c. blit contents of inputs into output framebuffer (if configured)
   d. set pass configuration and blending options
   e. iterate through scene objects (if scene pass or light pass) or draw fullscreen quad (if quad/postprocessing pass), and bind UBOs and buffers for each object as necessary
2. Run the viewport pass in the same way as (5), but siphon out data for third-party consumers (video recording, screenshots,...) if necessary
3. Swap buffers.

6.2.6 Shader Introspection, Shader Properties, and Shader Parameters

scenery's renderers by default perform introspection on the shader files they ingest, by using the *spirvcrossj*[6] library (github.com/scenerygraphics/spirvcrossj). Shader files are loaded via the `Shaders` class, which can provide both source code and SPIRV[7] bytecode, either from file sources, or from procedurally generated shaders, and potentially other sources. `Shaders` will try to provide both versions of a shader, but can be instructed to prioritise either the source code version or the SPIRV version of the shader.

[6] *spirvcrossj* is a wrapper of the Khronos Group's *spirv-cross* reflection library and the *glslang* reference compiler we have developed.
[7] SPIRV or SPIR-V is a binary bytecode format for shader files, specified by the Khronos Group. It serves as the primary provider of shader programs for Vulkan, but can also be used from OpenGL via vendor-specific extensions, and can be decompiled to plain GLSL or even HLSL.

By adding the `@ShaderProperty` annotation to a member variable of a `Node`-derived class, this variable can be made accessible from the shader via the same name. Supported data types are Java's default elementary types, as well as the `GLVector` vector type and the `GLMatrix` matrix type. Additionally, the `@ShaderProperty` annotation can be added to a hash map of type `HashMap<String, Any>` to provide even more flexibility (e.g. for procedurally-generated shaders). scenery discovers shader properties in the following way:

When loading the shader, construct a list of all the properties that are part of the `ShaderProperties` UBO in the shader file, e.g.

```
layout(set = 0, binding = 0) uniform struct ShaderProperties {
  float scale;
  mat4 model;
  vec3 color;
}
```

Metadata for each member in the form of an offset and a length are stored along the with the property, and follow GLSL's std140 rules for alignment.

When updating UBOs for a `Node`, scenery performs reflection (and caches that information) on all properties that carry the `@ShaderProperty` annotation: first, properties with the given name are checked, and if not found, the `shaderProperties` hash map is consulted as fallback. Not providing a shader property in the class that is required by the shader will result in an exception. The other way around, a shader property defined in the class, but not used in the shader will only trigger a warning. Shader properties defined in the UBO, but not used in the shader will not be skipped upon serialisation.

Shader parameters provide another way to hand parameters over to shader programs: These are defined in the rendering pipeline configuration, e.g. as (see Section 6.2.4, *Configurable Rendering Pipelines* for full examples of such configurations):

```
AO:
  type: quad
  parameters:
    Pass.displayWidth: 0
    Pass.displayHeight: 0
```

```
    occlusionRadius: 1.0
    occlusionSamples: 4
    occlusionExponent: 2.0
    maxDistance: 1.0
    bias: 0.1
    algorithm: 0
```

The shader parameters here are all the key-value pairs below `parameters`: The ones starting with `Pass` or `Global` are filled in automatically by the renderer, and are used e.g. for framebuffer sizes. All others are derived from scenery settings, and can be modified on-the-fly, enabling e.g. an easy way to switch between different algorithms in a shader, or turn functionality in the shader on and off. The corresponding declaration in the shader file looks the following (ordering does not matter and is resolved automatically):

```
layout(set = 2, binding = 0, std140) uniform ShaderParameters {
  int displayWidth;
  int displayHeight;
  float occlusionRadius;
  int occlusionSamples;
  float occlusionExponent;
  float maxDistance;
  float bias;
  int algorithm;
};
```

6.2.7 Instancing

Instancing can be done by defining a `Node`'s `instanceMaster` property as `true` and adding other `Nodes` to its `instances` property. Instances are not part of the regular scene graph for improved discovery performance, but their transformation matrices are updated in the same manner. Instanced properties can be added to the master node's `instancedProperties` hash map, and are serialised in the same manner as UBOs. As instances are not allowed to depend on each other, the serialisation is done in parallel in a number of worker threads using coroutines, such that hundreds of thousands of instances can be updated at interactive frame rates.

To use instancing, the user needs to provide a custom shader that declares the properties set in `instancedProperties`, e.g. as in

```
layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexNormal;
layout(location = 2) in vec2 vertexTexCoord;
layout(location = 3) in mat4 iModelMatrix;
```

where the location 3 defines the instanced model matrix. For both Vulkan and OpenGL, scenery will automatically derive a fitting vertex description.

It is important to note here that a matrix occupies more than a single vertex input, such that the next available location after the 4x4 matrix `iModelMatrix` would be location 7.

6.3 Rendering of Volumetric Data

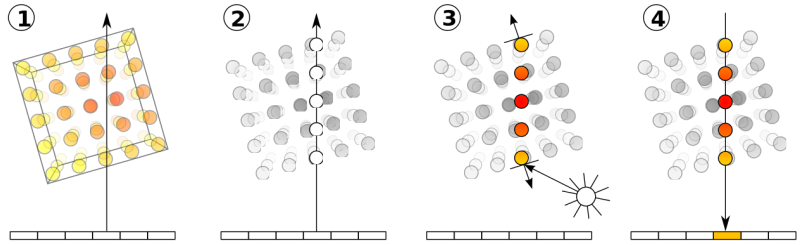

Figure 6.3: Volume raycasting schematic, 1. casting a ray through the volume, 2. defining sampling points, 3. calculation of lighting at the sampling points, 4. accumulation of the lit samples into a single pixel and alpha value

Volume rendering in scenery is done via volume raycasting, where a ray for each screen pixel, originating at the camera's near plane is shot perspectively correct through the piece of volumetric data, accumulating color and alpha information along the way. The accumulation function is customisable and can be used to realise e.g. the following compositing options:

- *maximum intensity projection* (MIP), where each voxel data point along the way is compared to the previous, and the maximum kept,
- *local maximum intensity projection* (LMIP), where each voxel data point along the way is compared to the previous, and the maximum kept, but only after reaching a user-defined threshold, and
- *alpha blending*, where the attenuation of light entering the volume is simulated in a physically plausible manner.

The first two of those, MIP and LMIP, are commutative in the sense that volumes superimposed on top of each other will lead to the same result, no matter in which order they are being rendered.

For alpha blending, the ordering of the volumes does matter, and accurate visualisation is only possible if all the volumes occupying the same space are rendering in the same moment. Alpha blending is based on a physical model, and in the next subsection we are going to derive the front-to-back compositing equations used in scenery, and then discuss options going beyond alpha compositing for added realism:

6.3.1 Alpha compositing

In general, the total spectral radiance $L_0(\vec{p}, \vec{v})$ — the quantity rendering determines for a pixel — at a point $\vec{p}$ in direction $\vec{v}$ is given by the rendering equation [Kajiya, 1986], where $\vec{l}$ is the light direction, and $\vec{n}$ the surface normal:

$$L_0(\vec{p},\vec{v}) = L_e(\vec{p},\vec{v}) \tag{6.1}$$
$$+ \int_{l\in\Omega} \mathrm{d}\vec{l} L_0(r(\vec{p},\vec{l},-\vec{l})(\vec{n}\cdot\vec{l})^+, \tag{6.2}$$

or,

Spectral radiance at $\vec{p}$ in direction $\vec{v}$ = Emission at $\vec{p}$ in direction $\vec{v}$
+ Reflected radiance from hemisphere Ω.

As this integral is recursive — the reflected radiance at a given point $\vec{l}$ depends on previous surface bounces subject to the same integral — it is in general very hard to solve accurately. For this reason, [Sabella, 1988] introduces an extension and simplification of the rendering equation for volumetric data: In this volumetric rendering equation, we compute $L(x)$, which is the spectral radiance at a point x along a ray, ignoring previous surface bounces:

$$L(x) = \int_x^y \mathrm{d}x' \epsilon(x') \exp\left(-\int_x^{x'} \mathrm{d}x'' \tau(x'')\right). \tag{6.3}$$

The spectral radiance here depends on $\epsilon(x')$, the emission, and $\tau(x')$, the absorption function, describing the probability of absorbing a photon along unit distance on the ray. Note here that this emission-absorption model completely ignores scattering.

Eq. 6.3 can be discretised as Riemann sum,

$$L(x) = \sum_i \epsilon_i \Delta x \cdot \exp\left(-\sum_j^{i-1} \tau_j \Delta x\right) \tag{6.4}$$
$$= \sum_i \epsilon_i \Delta x \cdot \prod_{j=0}^{i-1} \exp\left(-\tau_j \Delta x\right). \tag{6.5}$$

.

This discretisation gives natural rise to interpretations of color (pre-multiplied by the alpha value), and opacity:

$$\alpha_i C_i = \epsilon_i \Delta x \text{ (color)} \tag{6.6}$$
$$\alpha_i = 1 - \exp\left(\tau_i \Delta x\right) \text{ (alpha).} \tag{6.7}$$

With these equations, we can then derive the front-to-back compositing formula for alpha blending: Let $T_s^{s'}$ be the composite transparency of front-to-back samples $s, s+1, \ldots, s'$,

$$T_s^{s'} = \prod_{i=s}^{s'} T_i, \text{ then,} \tag{6.8}$$

$$C_s = \alpha_s C_s \tag{6.9}$$

$$T_s = 1 - \alpha_s, \text{ and} \tag{6.10}$$

$$C_s^{s'+1} = C_s^{s'} + \alpha_{s'+1} C_{s'+1} T_s^{s'} \tag{6.11}$$

$$T_s^{s'+1} = (1 - \alpha_{s'+1})\, T_s^{s'}. \tag{6.12}$$

Front-to-back compositing has the benefit that cast rays can be terminated early once $T_s^{s'}$ falls below a given threshold (or 0).

An example of an alpha-composited volume rendering is shown in Figure 6.4.

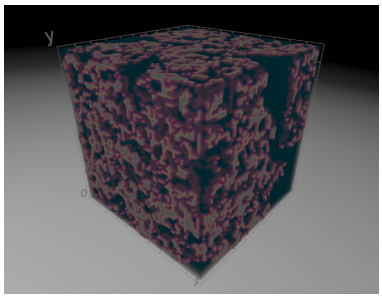

Figure 6.4: A volume rendered in scenery using alpha compositing, showing the Game of Life in 3D with volumetric ambient occlusion.

6.3.2 Out-of-core rendering

Out-of-core (OOC) rendering describes techniques for rendering volumetric data that does not fit into the GPU memory or main memory of a computer, and is therefore out-of-core.

BigDataViewer [Pietzsch et al., 2015] has introduced a HDF5 [Group, 1997]-based pyramid image file format that is now widely used. The program itself displays single slices that can be arbitrarily oriented to the user, and loads them on-the-fly from local or remote data sources (see Figure 6.3.2 for an example).

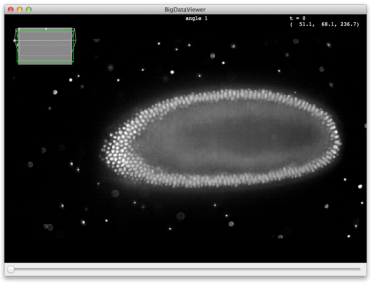

Figure 6.5: A *Drosophila* dataset rendered by-slice in BigDataViewer. Image courtesy of Tobias Pietzsch.

scenery also includes support for loading these data sets, and for that makes use of a BigDataViewer-provided library. An example of multiple volumes rendered using this technique is shown in Figure 6.6.

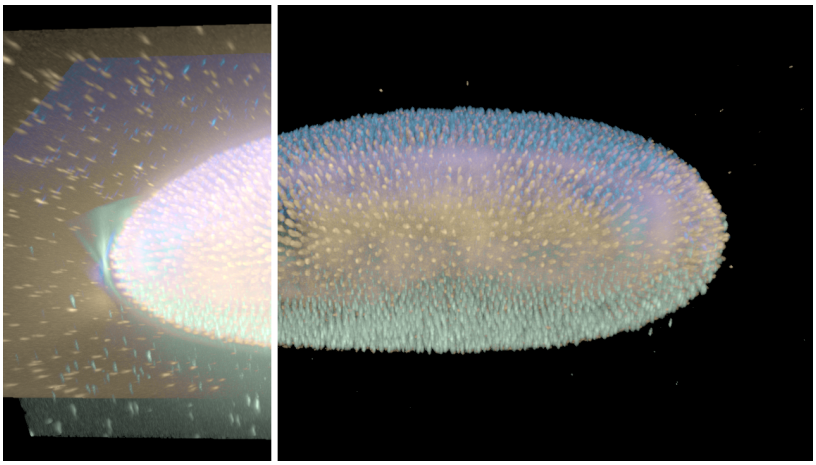

Figure 6.6: scenery rendering an out-of-core, multiview *Drosophila* dataset consisting of three different views (color-coded) using the BigDataViewer integration. volume rendering using maximum intensity projection. On the left-hand side, the transfer function has been adjusted to make boundaries between the different subvolumes visible more clearly.

Volumetric data for out-of-core rendering is stored in tiles of up to 2^{31} voxels. Tiles are addressed with 64bit, leading to a theoretical maximum data size of 2^{94} voxels, or about 20000 Yottabyte. Tiles are stored in a GPU cache. The cache is organised into small, uniformly-sized blocks storing a particular tile of the volume in the resolution pyramid. All tiles are padded by one voxel to avoid bleeding artifacts [Beyer et al., 2008]. To render a particular view of a volume, we determine the base resolution,

such that the screen resolution is matched for the voxel closest to the observer. We then prepare a 3D lookup texture (LUT) where each voxel corresponds to a volume block at base resolution. Each voxel in the LUT stores the coordinates of a tile in the in cache, as well as the resolution level relative to the base level, encoded as RGBA quadruple. For each visible volume tile, we determine the ideal resolution by the distance to the observer.

If a tile requested is already present in the cache, we encode its coordinates in the corresponding LUT voxel. Should a tile not yet be present in the cache, we enqueue the missing block for asynchronous loading through the cache layer of BigDataViewer. Recently-loaded blocks are inserted into the cache texture. The cache has least-recently used (LRU) behaviour, such that the oldest tiles are the first ones to be replaced. Missing blocks are substituted by lower-resolution data while not yet available. This technique is a combination of hierarchical blocking [Beyer et al., 2008, LaMar et al., 1999] and the missing data scheme introduced in [Pietzsch et al., 2015].

When the LUT is prepared, volume rendering proceeds by raycasting, while adapting the step size along the ray depending on distance to the observer. The correct coordinates for each voxel are determined by scaling its coordinate down to coincide with the voxel in the LUT. Nearest-neighbour sampling from the LUT then yields a block offset and scale in the cache texture. The final voxel value is then sampled from the cache texture, after reversing the scaling and translation operations.

This approach enables to render multiple volumes simultaneously, by adding additional LUTs for each volume. Non-out-of-core volumes can be rendered as well, such then do not require additional LUTs.

To produce the correct shader for multiple volumes, which can also change in number, we make use of a custom `ShaderFactory` that can ingest the shader code generated by BigVolumeViewer, and transform it to scenery's conventions. ShaderFactories are created for a particular number of volumes, e.g., 3 out-of-core (OOC) volumes, mixed with 2 regular ones, and cached up to a count of 8 factories. A sketch of the workflow is shown in Figure 6.7.

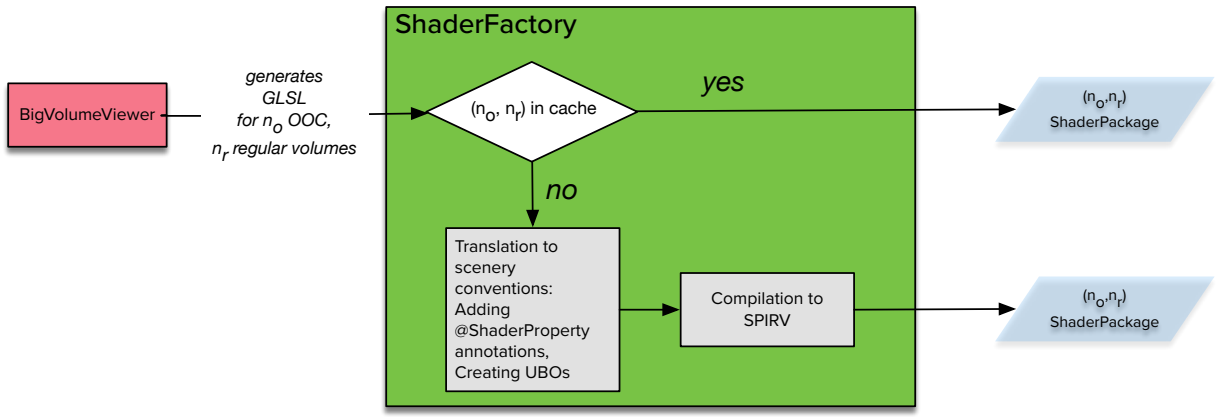

Figure 6.7: Workflow for translating between BigVolumeViewer and scenery.

Scan this QR code to go to a video demo of out-of-core volume rendering in scenery. For a list of supplementary videos see https://ulrik.is/writing/a-thesis.

6.4 Rendering with OpenGL

In this section, we will introduce how a scene is rendered with OpenGL, to compare it with Vulkan lateron.

6.4.1 Initialisation

When using OpenGL, the initialisation of the renderer proceeds in the following way:

1. the presence of a virtual reality HMD is checked.
2. the `RenderConfig` is parsed from a file.
3. it is checked whether the renderer will be embedded in a `SceneryPanel` for rendering with JavaFX, or if a drawable surface already exists. If both are not the case, a window owned by the renderer is created. After this, a separate animator thread takes over the initialisation.
4. an OpenGL 4.1 context is initialised.
5. a default set of buffers is initialised for:
   - Uniform Buffer Objects (UBOs)
   - light parameters
   - rendering parameters
   - Shader Properties
   - Shader Parameters
6. a default set of textures is initialised to provide textures in case a `Node`'s textures cannot be found.
7. the render passes defined by the `RenderConfig` are created, with the necessary framebuffers and attachments.
8. a heartbeat timer is initialised to record GPU usage (only on Nvidia cards running on Windows) and record performance data.

The renderer then proceeds with scene initialisation, initialising each `Node` in the scene in this way:

1. The `Node` is locked.
2. a new instance of `OpenGLObjectState` is attached to the `Node`'s `metadata` field
3. vertex buffers[8] and vertex array objects[9] are created.
4. custom shaders and material properties are initialised.
5. UBOs for transformations and material properties are initialised.
6. the renderer checks whether the `Node`'s definition includes any `@ShaderProperty` annotations, for which an additional UBO would be created (see Section 6.2.2, *Uniform Buffer Serialisation and Updates* for details).
7. The `Node` is marked as initialised and unlocked.

[8] Vertex Buffer Objects are the buffers on the GPU that actually contain the vertices of a Node's geometry. In scenery, they are stored in a strided format, meaning that vertices and normals are not stored as `V1V2V3N1N2N3...`, but rather as `V1N1V2N2V3N3`, for improved cache locality.

[9] Vertex Array Objects describe which buffers a rendered object in OpenGL uses, and what the data layouts of these buffers are.

6.4.2 Rendering loop with OpenGL

The rendering loop in the OpenGL-based renderer in turn works as follows. In the section about rendering with Vulkan, we will highlight the differences to OpenGL.

1. Early exit conditions are checked, e.g. for renderer shutdown, or for switching to fullscreen rendering.
2. UBOs are updated, and the existence of changes is recorded.
3. For all nodes that are rendered in a geometry pass:
   1. The Node is initialised if not seen before
4. If push mode is active and no changes have been recorded, the loop will return now. If push mode is not active, command buffer re-recording will commence.
5. For all render passes, draw commands are issued. Drawing proceeds in the following way:
   - if blitted images from the output of the previous pass are required, they are blitted into the current rendertargets, either by copying (if source and target formats are compatible), or actual blit.
   - *geometry pass*: viewport and blending parameters are set, and all nodes are iterated over, grouped by shader pipeline used in order to minimise state switches. Textures, custom shaders, and geometry are updated if necessary. With shader pipeline and vertex/instance buffers bound, the node is rendered.
   - *quad/postprocessing* pass: Viewport parameters are set, then the shader pipeline for the pass is bound, then a fullscreen quad is drawn using this pipeline.
6. Images created in the viewport pass are submitted to a possibly-connected HMD, stored as screenshots, or submitted to the video encoder.

6.5 Rendering with Vulkan

After having introduced the rendering steps in OpenGL, in this section we do the same for the newer, high-performance Vulkan API.

6.5.1 Initialisation

Compared to OpenGL, Vulkan provides much leaner, close-to-metal access to the GPU's resources. It is therefore more verbose, but does not do as much validation as OpenGL during runtime.

Instead in Vulkan, a *validation layer* provides guidance to adherence to the standard. While in OpenGL, all state and state changes are continuously checked for sanity, incurring a performance hit, Vulkan skips these checks, and outsources them to validation layers, that may be activated during startup, and usually only used during development or debugging. Compared to OpenGL error messages, Vulkan validation layers provide highly detailed error messages, thus enabling the developer to quickly pinpoint the source of a problem.

When using Vulkan, the initialisation of the renderer proceeds as follows:

1. the presence of a virtual reality HMD is checked, and window dimensions set accordingly.
2. The `RenderConfig` is parsed from a render config YAML file.

3. The renderer checks whether the user has requested the activation of validation layers, and activates them if indicated.
4. the renderer checks whether the output will be embedded in a `SceneryPanel`. If this is not the case, the renderer will request surface rendering extensions, and create a *Vulkan instance*[10], if not, the instance will be created without the extensions. Further extensions might be required by the presence of an HMD, e.g. for memory sharing.
5. A `VulkanDevice`[11] is created, the renderer defaults to the first device found, but this can be overridden by the user.
6. At least one device queue[12] is created. Multiple queues might be created for rendering, compute work, and data transfers, if supported by the selected `VulkanDevice`.
7. A `SwapchainRecreator` is created. This object is responsible for all resources that are resolution-dependent, such as framebuffers and rendering surfaces. It also takes care of initialising the render passes defined by the `RenderConfig`, with the necessary framebuffers and attachments. The swapchain is be one of:
   - `OpenGLSwapchain`: creates an OpenGL context and enables the Vulkan renderer to draw into that. Necessary for frame-locked rendering e.g. on CAVE systems.
   - `HeadlessSwapchain`: creates a swapchain without any rendering surfaces, e.g. for server use.
   - `SwingSwapchain`: create a swapchain for rendering into a Swing panel. Employs permanently mapped buffers and native surfaces for efficiency, and inherits from `VulkanSwapchain`.
   - `VulkanSwapchain`: creates a regular, window-based swapchain with GLFW. This is the standard case.
8. the default vertex descriptors[13] are created.
9. descriptor pools are created, from these, the default descriptor set layouts and descriptor sets[14] are prepared.
10. a default set of textures is initialised to provide textures in case a `Node`'s textures cannot be found.
11. a heartbeat timer is initialised to record GPU usage (only on Nvidia cards running on Windows) and record performance data.
12. a dynamically-managed memory pool for `Node` geometry is allocated.

Note here that the Vulkan renderer does not perform explicit scene initialisation on startup, but discovers the `Nodes` of a scene during the rendering loop.

[10] A *Vulkan instance* is the basic building block of a Vulkan application.

[11] A *Vulkan device* contains all the information about a device and its capabilities, and all allocations and executions are made with respect to a particular device, also enabling parallel runs on multiple devices.

[12] Work, may it be rendering or compute work, is submitted to a *queue* in Vulkan, and executed asynchronously by the GPU. A queue may be asked for work completion and can be waited on.

[13] *Vertex descriptors* describe the vertex layout for rendering and are somewhat comparable to OpenGL's Vertex Array Objects.

[14] *Descriptor set layouts* describe the memory layout of UBOs and textures in a shader, while *descriptor sets* contain their actual realisation, and link to a physical buffer, or texture.

6.5.2 Rendering loop with Vulkan

For comparison with the sketch of the OpenGL rendering loop above, we highlight differences to OpenGL with bold typeface here, and add an explanation of the particular difference right after that in italics.

1. Early exit conditions are checked, e.g. for renderer shutdown, or for switching to fullscreen rendering.
2. UBOs are updated, and the existence of changes is recorded.

3. **Node data is updated, for all nodes that are rendered in a geometry pass** (*in Vulkan, this is executed before iterating over all Nodes for rendering, such that the command buffers recorded there can be reused and do not contain geometry updates, etc., which are usually only executed once.*):
   1. The Node is initialised if not seen before,
   2. geometry is updated if necessary,
   3. textures are updated if necessary,
   4. custom shaders are reloaded if necessary.
4. If push mode is active and no changes have been recorded, the loop will return now. If push mode is not active, command buffer re-recording will commence.
5. For all render passes, **except the viewport pass, command buffers are recorded** (*in Vulkan, work for the GPU is not directly executed as in OpenGL, but enqueued in command buffers instead. These can be reused later on for increased performance, as one does not need to iterate over scene contents again, if nothing changed in between.*). For each pass, as many command buffers as there are swapchain images are prepared, such that multiple frames can be in-flight in parallel to ensure optimal GPU occupancy. Recording happens as follows:
   1. **If the currently-recording command buffer is still in-flight, it will be waited on using a fence** (*in Vulkan, synchronisation operations between CPU-GPU, or GPU-GPU, are more explicit than in OpenGL, giving the developer greater control over what happens when; here, we might need to wait for the command buffer to not overwrite its contents while still in use and thereby cause undefined state. The synchronisation here is done using a fence.*).
   2. The command buffer for the respective pass type is recorded:
      - if blitted images from the output of the previous pass are required, they are blitted into the current rendertargets, either **by copying , or actual blit** (*while OpenGL only knows actual blit operations, Vulkan can perform copies between images, which incur a lower overhead, if source and target formats are compatible*).
      - *geometry pass*: **A new Vulkan renderpass is started** (*in Vulkan, it is possible to describe state and dependencies between different passes using Vulkan renderpass objects, this information can be used by the driver for optimisations, e.g. if certain attachments of a framebuffer will not be reused, or if a renderpass has no execution dependencies, and can be executed in parallel to another one*), viewport parameters set, and all nodes are iterated over, grouped by shader pipeline used in order to minimise state switches. With the corresponding **descriptor sets** (*see* [15]) and vertex/instance buffers for the pipeline bound, the node is rendered.
      - *quad/postprocessing* pass: **A new Vulkan renderpass is started** (*see above*), viewport parameters set, then the shader pipeline for the pass is bound, together with the corresponding descriptor sets, then a fullscreen quad is drawn using this pipeline.
   3. The command buffer is submitted to the rendering queue.
6. **The viewport pass is recorded in the same way as above, submitted, and presented using the swapchain** (*in contrast to* `glSwapBuffers()` *in OpenGL to finish a frame, this is done in Vulkan by presenting to a Swapchain, which can be*

[15] *Descriptor set layouts* describe the memory layout of UBOs and textures in a shader, while *descriptor sets* contain their actual realisation, and link to a physical buffer, or texture.

tied to an window surface, or can be headless, Vulkan here is also less coupled to the windowing system than OpenGL, making integration with Swing, JavaFX, or headless rendering much easier).

7. Images created in the viewport pass are submitted to a possibly-connected HMD, stored as screenshots, or submitted to the video encoder.

6.6 Performance

The Java Virtual Machine is quite an unorthodox choice for realtime rendering. One reason might be that the JVM is still perceived as slow, especially when compared to close-to-metal languages like C or C++.

In this section, we compare performance of matrix multiplications on the Java VM with native code. Matrix multiplications in our context are particularly representative, as they occur in large amounts when doing scene graph traversals, in order to compute node positioning, scaling, and rotation in space.

6.6.1 Performance of Matrix multiplications on the Java Virtual Machine

The code for the performance comparison can be found at github.com/skalarproduktraum/java-autovectorisation-test

Recent studies in the *Computer Languages Benchmark Game*[16], which benchmarks different languages in scenarios such as computing Mandelbrot sets or n-body simulations, have shown that the performance of JVM-based software is on par with other VM-based languages (such as C# or Julia), or even Intel Fortran, and does not lag behind C/C++ much — usually a factor of two (see also Figure 6.8, benchmarking with "toy example" should however always be taken with a grain of salt, as real-world performance may be influenced by a variety of other factors).

[16] See benchmarksgame-team.pages.debian.net/benchmarksgame/.

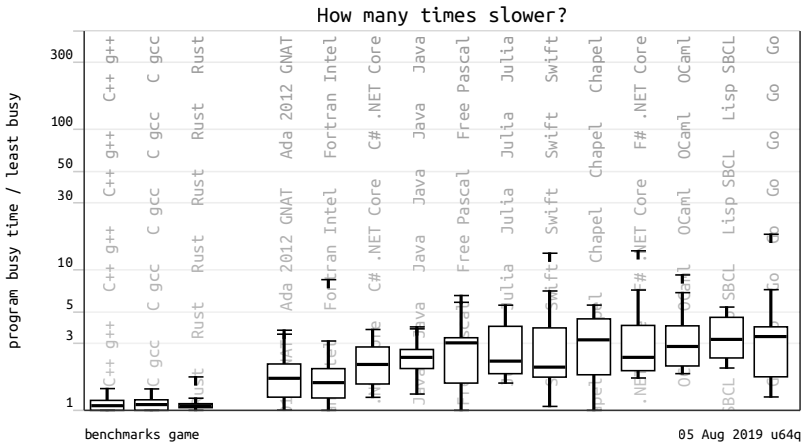

Figure 6.8: Comparison of different languages with the *Computer Languages Benchmark Game*, as of August 2019. Figure from benchmarksgame-team.pages.debian.net/benchmarksgame/which-programs-are-fastest.html.

High-performing code on the JVM is usually written by understanding what optimisations the JVM does in which case, and not trying to outsmart the JVM. For parts of the code the JVM determines as performance hotspots, the bytecode is taken

and compiled to native code *just-in-time*, usually leading to near-native performance. In this subsection, we will investigate a specific performance case, namely matrix multiplications. For any transformations in 3D, matrix multiplications play a crucial role, and are often executed thousands of times per second for a moderately complex scene.

For our comparison, we consider three different ways to do matrix multiplications, defined as $C_{ij} = \sum_k A_{ik}B_{kj}$:

- *loop-based*, where the two sums are implemented as loops over the floating-point numbers stored in float arrays,
- *loop-based with FMA*, where the two sums are implemented as loops over the floating-point numbers stored in float arrays, and the summation is done by the FMA (fused multiply-add) instruction, which executes a $a \cdot b + c$ operation in one CPU cycle (supported since Java 9), and
- *unrolled loop-based*, where the loops in the first two cases has been unrolled by hand.

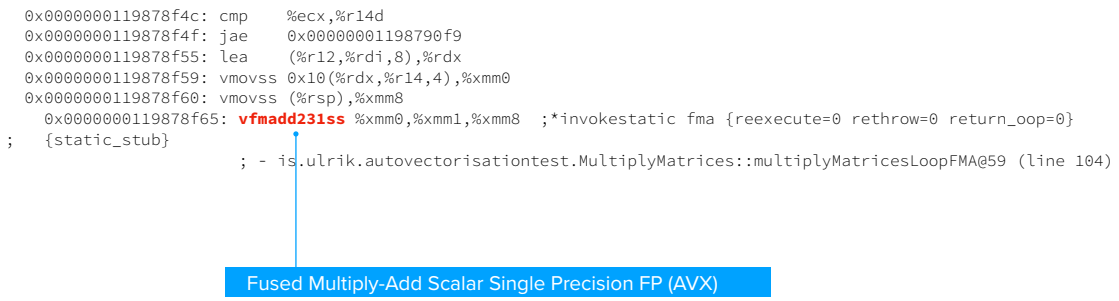

Figure 6.9: Generated assembly for the loop-based matrix multiplication with FMA, ran on JDK 9.0.4, macOS 10.12.6, Intel Core i7-4980HQ CPU @ 2.80GHz.

By using the *hsdis* utility of the Java Development Kit, it is possible to glimpse into the assembly code that gets generated by the JVM, after passing the flags `-XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly` to the JVM on startup.

In the loop case, we arrive at very inefficient assembly containing a lot of jumps, but also AVX512 SIMD (single instruction, multiple data) instructions mixed with SSE SIMD instructions — although they are only scalar operations, while ideally they'd be vectorised. In addition, the mixing of AVX and SSE registers incurs a performance penalty (the assembly is not shown here due to its length). In contrast, the FMA-based and unrolled loop version (see Figures 6.9 and 6.10) yield better assembly code with less jumps, and more AVX512 commands, leading to less CPU cycles needed for the matrix multiplication. The AVX512 commands however operate only on the 128bit SSE registers (`xmm0-15`), while for optimal performance they should use the 512bit AVX512 registers `zmm0-31`, or at least the 256bit AVX registers `ymm0-15`.

The following table shows the results for the different routines for matrix multiplications, with the benchmarks run on JDK 9.0.4, macOS 10.12.6, Intel Core

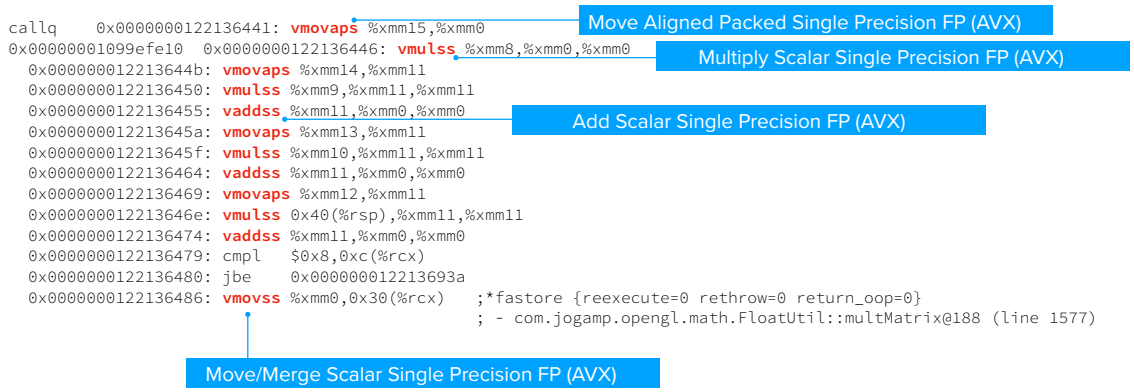

Figure 6.10: Generated assembly for the loop-unrolling matrix multiplication, ran on JDK 9.0.4, macOS 10.12.6, Intel Core i7-4980HQ CPU @ 2.80GHz.

```
#include <immintrin.h>
#include <intrin.h>

union Mat44 {
    float m[4][4];
    __m128 row[4];
};

void matmult_AVX_8(Mat44 &out, const Mat44 &A, const Mat44 &B)
{
    _mm256_zeroupper();
    __m256 A01 = _mm256_loadu_ps(&A.m[0][0]);
    __m256 A23 = _mm256_loadu_ps(&A.m[2][0]);

    __m256 out01x = twolincomb_AVX_8(A01, B);
    __m256 out23x = twolincomb_AVX_8(A23, B);

    _mm256_storeu_ps(&out.m[0][0], out01x);
    _mm256_storeu_ps(&out.m[2][0], out23x);
}
```

Listing 6.4: Example code for 4x4 matrix multiplication using AVX512 intrinsics. Source: https://gist.github.com/rygorous/4172889.

i7-4980HQ CPU @ 2.80GHz using the Java Microbenchmarking Harness (timings given are averaged over 10 iterations, with 5 iterations of warm-up before):

Table 6.1: 4x4 matrix multiplications routines on the JVM compared with native C++ routines.

Routine	Timing ± Std.Dev.
Loop-based	27.664±0.782ns
Loop-based, FMA	24.553±0.774ns
Unrolled loop	21,906±0.389ns
Unrolled loop	21,906±0.389ns
C++ loop-based (`gcc -O1`)	25.6ns
C++ AVX512 (`gcc -O3`)	3.2ns

Ideally, 4x4 matrix multiplication would utilise both AVX512 commands and registers. This can be achieved with hand-written C code, see Listing 6.4 for an example.

As can be seen from the table, the unoptimised C++ version of loop-based 4x4 matrix multiplication is in the same timing region as the loop-based Java version. The AVX512 is way faster, though. In the future, the JVM will gain native support for using SSE and AVX intrinsics manually in the language, almost certainly closing that gap. This initiative runs under the name *Project Panama*, and preview releases can be

6.7 Summary

In this chapter, we have introduced the rendering architecture of scenery, discussed the renderer interface, and special scenery features like semi-automatic instancing and custom rendering pipelines. We discussed the differences between the OpenGL and Vulkan renderers, and compared a performance-critical operation in the context of the JVM and native code.

For future work, we refer the reader to Chapter 10, *Future Development Directions*, which summarises the future targets for all components of scenery.

Chapter 7:

Input Handling and Integration of External Hardware

Here we describe the input handling subsystem of scenery, and how external hardware, such as head-mounted displays or natural user interfaces can be used. The major design goal for these subsystems was to enable the user to design interactions and tools cross-API — here, we are going to detail how we have achieved that.

7.1 Input Handling

Input handling is done using using Tobias Pietzsch's open-source *ui-behaviour* library[1]. The ui-behaviour library is heavily used in the ImageJ/Fiji ecosystem, and provides a handles input events via `InputTriggers` and `Behaviours`. An `InputTrigger` is the related to the physical event, such as a key press, or a mouse movement, scroll, or click. A `Behaviour` is the triggered action. *ui-behaviour* is able to handle AWT input events. For scenery, we have extended the library to also be able to handle events originating from JOGL, GLFW, JavaFX, or headless windows. Furthermore, we extended the library to enable custom mappings for buttons of hand-held controller devices, or VRPN[2] devices.

Spatial input, such as HMD positioning and rotations, are handled by the specific implementation of a `TrackerInput`, such as an `OpenVRHMD`[3], or a `VRPNTracker`[4]. Multiple of these inputs can coexist peacefully, and will not interfere with each other, but rather augment.

[1] See github.com/scijava/ui-behaviour/ for details.

[2] VRPN (Virtual Reality Periphery Network) is an abstraction layer that enables access to a variety of Virtual Reality-associated input devices, such as tracked stereo glasses, Wiimotes or Flysticks. See [Taylor et al., 2001].

[3] OpenVR or SteamVR is a runtime and abstraction layer that provides a vendor-independent API to various VR headsets, and access to the associated input devices and rendering surfaces. Usable VR headsets include e.g. the Oculus Rift, HTC Vive, or the various Windows Mixed Reality headsets. OpenVR has been developed by Valve Software, more information about the library can be found at github.com/ValveSoftware/OpenVR.

[4] VRPN (Virtual Reality Periphery Network) is an abstraction layer that enables access to a variety of Virtual Reality-associated input devices, such as tracked stereo glasses, Wiimotes or Flysticks. See [Taylor et al., 2001].

7.2 Head-mounted displays and natural/gestural user interface devices

Support for head-mounted displays and control devices is at the moment provided by two means:

1. utilising the lwjgl bindings for SteamVR/OpenVR to interface with off-the-shelf HMDs like the Oculus Rift or HTC Vive (see class `OpenVRHMD`).
2. utilising the custom-built wrappers for VRPN[Taylor et al., 2001], called jVRPN (github.com/scenerygraphics/jvrpn). jVRPN is used in scenery to e.g. provide support for DTrack or OptiTrak tracking systems used in CAVE systems or Powerwalls (for more details, see the class `TrackedStereoGlasses` and `VRPNTracker`).

When a new HMD is to be added to scenery, a new class has to be created for the HMD, and that class needs to implement two interfaces:

- `Display`, for querying information about the HMD's rendering surfaces, projection matrices, and state, and
- `TrackerInput`, for providing spatial information to scenery, about the HMD's position, rotation, and associated input devices.

The `Display` interface looks as follows:

```
interface Display {
    fun getEyeProjection(eye: Int, nearPlane: Float = 1.0f, farPlane: Float = 1000.0f): GLMatrix
    fun getIPD(): Float
    fun hasCompositor(): Boolean
    fun submitToCompositor(textureId: Int)
    fun submitToCompositorVulkan(width: Int, height: Int, format: Int,
                                 instance: VkInstance, device: VulkanDevice,
                                 queue: VkQueue,
                                 image: Long)
    fun getRenderTargetSize(): GLVector
    fun initializedAndWorking(): Boolean
    fun update()
    fun getVulkanInstanceExtensions(): List<String>
    fun getVulkanDeviceExtensions(physicalDevice: VkPhysicalDevice): List<String>
    fun getWorkingDisplay(): Display?
    fun getHeadToEyeTransform(eye: Int): GLMatrix
}
```

This interface provides submission to both OpenGL and Vulkan renderers, which is necessary as they work quite differently[5]. The `update` function is used to update HMD state once per frame, while transformations are cached as much as possible. The renderer will only render to the device if `initializedAndWorking()` returns `true`, and `getWorkingDisplay()` returns a `Display` instance. HMDs can choose to have a compositor to which the resulting rendered image is submitted. If this is not the case, e.g. as it is with tracked stereo glasses for CAVEs, the image is rendered regularly.

[5] In OpenGL, there is e.g. no explicit device selection, as this is done implicitly by the driver. In Vulkan, this is done by the application explicitly, and so information about on which device a rendered texture resides has to be passed to the compositing application, such as the one of SteamVR/OpenVR.

Equally simple is the `TrackerInput` interface:

```
interface TrackerInput {

    fun getOrientation(): Quaternion
    fun getOrientation(id: String): Quaternion
    fun getPosition(): GLVector
    fun getPose(): GLMatrix
    fun getPoseForEye(eye: Int): GLMatrix
    fun initializedAndWorking(): Boolean
    fun update()
```

```
    fun getWorkingTracker(): TrackerInput?
    fun loadModelForMesh(device: TrackedDevice, mesh: Mesh): Mesh
    fun loadModelForMesh(type: TrackedDeviceType = TrackedDeviceType.Controller, mesh: Mesh): Mesh
    fun attachToNode(device: TrackedDevice, node: Node, camera: Camera? = null)
    fun getTrackedDevices(ofType: TrackedDeviceType): Map<String, TrackedDevice>
}
```

In this interface, the noteworthy functions are `getPoseForEye()`, which returns a transformation matrix relative to the origin containing translational and rotational information separately for each eye. Furthermore, an HMD may provide multiple tracked devices, such as nunchucks, which can be queried via `getTrackedDevices()`. Via `attachToNode()` a controller can be attached any `Node` in the scene graph (most likely one representing a 3D model of the controller), which subsequently inherits the transformations of the tracked device. A suitable model for such a device may be provided by the HMD via `loadModelForMesh()`. Our implementation of SteamVR HMDs provides the correct mesh for a given HMD via this function.

7.3 Augmented Reality and the Hololens

scenery also includes support for the Microsoft Hololens, a stand-alone, untethered augmented reality headset, based on the Universal Windows Platform (see class `Hololens`). The Hololens includes its own CPU and GPU, due to size constraints they are, however, not very powerful, and especially when it comes to rendering of volumetric datasets, underpowered.

To get around this issue, we have developed a thin, Direct3D-based client application for the Hololens that makes use of Hololens Remoting, a proprietary streaming protocol developed by Microsoft[6]. This client receives pose data from the Hololens, as well as all other parameters required to generate correct images, such as the projection matrices for each eye. This data is then forwarded to a Hololens interface within scenery, based on the regular HMD interface. Initial communication to acquire rendering parameters is done via a ZeroMQ request-reply socket, while receiving of per-frame pose data is handled with an additional, publish-subscribe socket due to better latency.

[6] The exact details of how this works are not published, but apparently work by streaming the image data for both eyes over the network, compressed with H264.

The Hololens remoting applications are usually fed by data rendered with Direct3D, which lets us immediately recognise the problem that *scenery* can only render via OpenGL and Vulkan at the present moment.

Fortunately, a shared memory extension for Vulkan, `NV_external_memory`, exists in the standard that enables *zero-copy* sharing of buffer data between different graphics APIs, by using a keyed mutex. Programmatically, this is done as:

1. On the host (*scenery*) side, verify that the device supports this type of Direct3D texture for importing.
2. For each swapchain image, allocate a Direct3D shared handle texture with flag

`D3D11_RESOURCE_MISC_SHARED_KEYEDMUTEX` on the side of the client application. This image will serve as the final render target to be sent to the Hololens[7].
3. For each shared handle, create a Vulkan image, with memory bound to the shared handle.
4. Request access to the image via the keyed mutex, and store image data in it, e.g. via `vkCmdBlitImage`. Keyed mutex handling is done by the extension itself, via extra information attached to the appropriate `vkQueueSubmit` call. The command buffer for the blit operation needs to be recorded only once and can be reused as long the resolution does not change.

[7] We allocate multiple image buffers and use them in a double/triple-buffering manner for read/write access to prevent the GPU stalling.

The inclusion of the keyed mutex information into the `vkQueueSubmit` call in the last step has the benefit that no additional network communication via ZeroMQ is necessary to indicate by which part of the software the shared texture is used at the moment, leading to increased performance.

7.4 Eye Tracking

scenery includes support for the Pupil Labs eye tracking solution [Kassner et al., 2014](www.pupil-labs.com), implemented in the class `PupilEyeTracker`. This class communicates with the Pupil Labs software *Pupil Capture* or *Pupil Service* via ZeroMQ, with msgpack data serialisation. Our implementation provides HMD-based screen-space and world-space calibration, reporting of the gaze positions, normals, timestamps, and confidences.

For calibration, the user is presented with a series of points to look at, which serve to establish a connection between the eye tracker's captured direction of gaze with a screen-space or world-space position in the scene. For each calibration point, 80 different samples are taken to account for microsaccades, and the first 20 are discarded to exclude those samples that might still include eye movement, while the user is moving from one point to the next. After finishing the calibration, the user is informed of successful or unsuccessful calibration, and can repeat the calibration, if necessary.

After successful calibration, scenery enables the developer to connect the outputs of the eye tracker (in the form of gaze normals, gaze positions, and a confidence rating) to the properties of any object. By default, gaze points are only output in case the confidence reaches more than 80%, leaving the decision to the developer which of the higher-confidence samples to use.

More details about eye tracking can be found in Chapter 4, *Eye Tracking and Gaze-based Interaction*, with a use case implemented in Chapter 11, *Bionic Tracking: Using Eye Tracking for Cell Tracking*.

Chapter 8:

Distributed Rendering

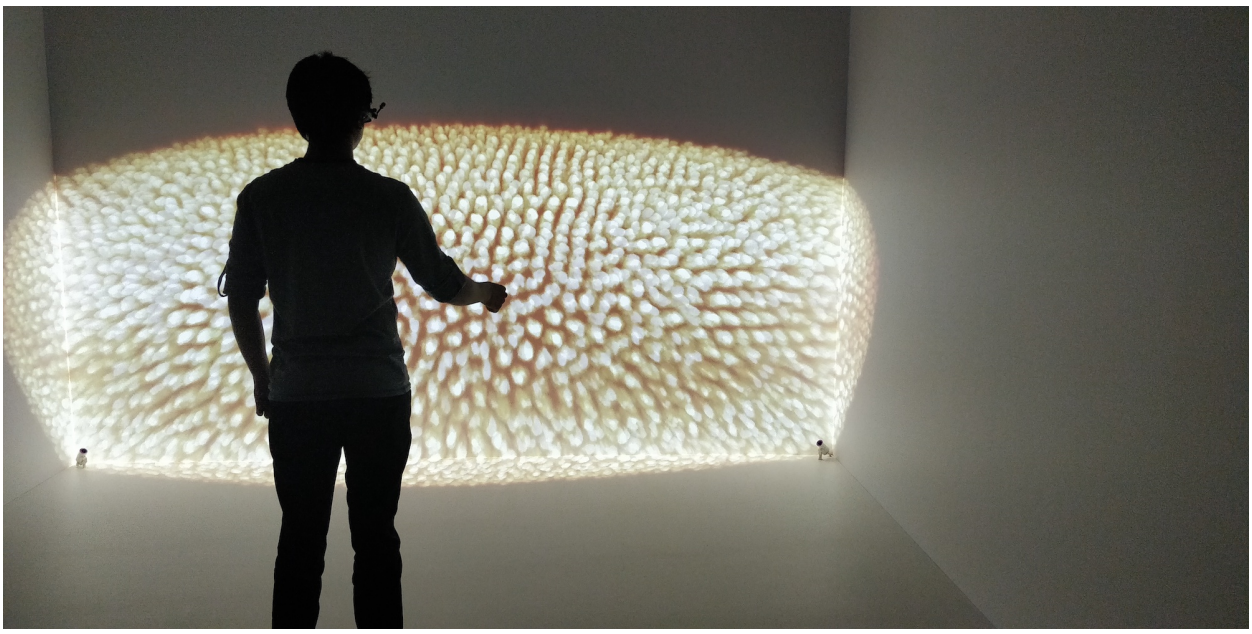

Figure 8.1: A user interacting with a *Drosophila* dataset rendered on a clustered 4-sided CAVE setup with 5 machines. Photo courtesy of Aryaman Gupta, MPI-CBG, Dresden.

Apart from rendering to VR/AR headsets, scenery includes support for parallel and distributed rendering on multiple machines. While all scenery-based applications are in principle ready to run on multiple machines in concert, a bit of configuration is required. This chapter details the necessary steps, starting with the geometry definition for multi-projector/multi-screen environments, followed by an explanation what steps are necessary on the software side. We end the chapter with information about how scenery keeps the information on all machines synchronised.

8.1 Screen Geometry Definition

For setting up a multi-projector, multi-machine environment with scenery, the physical geometry of the projection space has to be defined. This is done using a simple YAML file, an example of which can be seen in Listing 8.1.

```
1 name: CAVE example configuration
2 description: Multi-screen configuration, demoing a 4-sided CAVE environment
```

```
3  screenWidth: 2560
4  screenHeight: 1600
5
6  screens:
7    front:
8      match:
9        type: Property
10       value: front
11     lowerLeft: -1.92, 0.00, 1.92
12     lowerRight: 1.92, 0.00, 1.92
13     upperLeft: -1.92, 2.40, 1.92
14   left:
15       match:
16         type: Property
17         value: left
18       lowerLeft: -1.92, 0.00, -1.92
19       lowerRight: -1.92, 0.00, 1.92
20       upperLeft: -1.92, 2.40, -1.92
21   right:
22       match:
23         type: Property
24         value: right
25       lowerLeft: 1.92, 0.00, 1.92
26       lowerRight: 1.92, 0.00, -1.92
27       upperLeft: 1.92, 2.40, 1.92
28   floor:
29       match:
30         type: Property
31         value: floor
32       lowerLeft: -1.92, 0.00, -0.48
33       lowerRight: 1.92, 0.00, -0.48
34       upperLeft: -1.92, 0.00, 1.92
```

Listing 8.1: Simple configuration file for a four-sided CAVE environment with each wall having a resolution of 2560x1600 pixels.

First, the name and description of the configuration are set (lines 1 and 2), along with the screen dimensions in pixels (lines 3 and 4, all screens have to have the same pixel width and height). Following that, a set of screens is defined (line 6 and following). These screens can have arbitrary names, and should ideally reflect a physical property, e.g. relating to their positioning in the room. The `match` element (e.g. line 8) defines how scenery determines which machine in the cluster is associated with which screen. Two ways are possible for this:

- `type: Property`, where the screen is determined by the system property `scenery.screenName` set to the string given in `value`, or
- `type: Hostname`, where the screen is determined by matching the hostname of the machine to the name given in `value`.

The following three vectors, `lowerLeft`, `lowerRight`, and `upperLeft` determine the corners of the screen surface in physical space, which is assumed to be rectangular (e.g. lines 11, 12, and 13). Note here that the tracking system also needs to be calibrated to the same coordinate space. In the example above, the coordinate origin is on the floor, 0.48m in front of the `front` screen, and each screen has a width of 3.84m, and a height of 2.40m.

8.2 Synchronisation of Scene Data

Scene data is synchronised using a custom ZeroMQ-based protocol. ZeroMQ is a low-latency message-passing library with bindings for a variety of languages, and support of all major operating systems. It is very resilient to network issues, e.g., lost connections are reestablished automatically. Furthermore, ZeroMQ supports multiple connection topologies, such as publish-subscribe (non-blocking), request-reply (blocking), or push-pull. We use the publish-subscribe topology, which is non-blocking, to synchronise scene contents. The actual synchronisation is deliberately kept extremely simple, in order to be able to execute the synchronisation step as fast as possible.

Two classes in scenery are responsible for scene synchronisation, *NodePublisher* and *NodeSubscriber*. The NodePublisher creates a ZeroMQ Publisher socket, to which an arbitrary number of *NodeSubscribers* can connect. Node changes are detected using the mechanism described in Section 6.2.3, *Push Mode*, it is the same the renderer uses to decide whether a node needs re-rendering. If a node changes, it is serialised into a stream of bytes using the open-source library Kryo[1] library. Kryo was chosen on the basis of performance and usability: It outperforms most other Java serialisation libraries, while not requiring code changes or large additions, and can be augmented with custom (de)serialisation routines.

[1] See github.com/EsotericSoftware/Kryo for code and details.

A NodePublisher instance is created by default when scenery's base class `SceneryBase` initialises. It then listens on the local network interface on port 6666 for connections. NodeSubscribers are not created by default. They are only created if the system property `scenery.MasterNode` is set to the address of a master node.

A schematic of the scene synchronisation is shown in Figure 8.2.

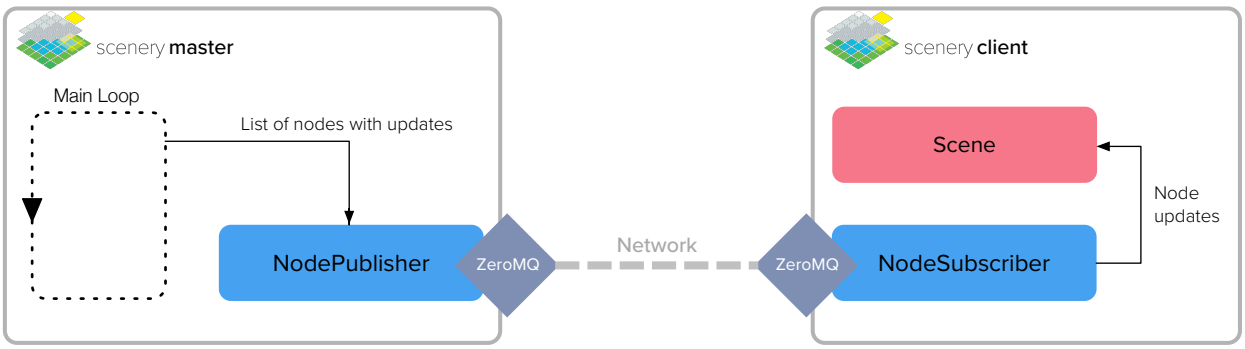

Figure 8.2: Schematic of the scene synchronisation in scenery, where one or more clients connect to a master in order to synchronise scene contents over the network via ZeroMQ. See text for details.

8.3 Software Setup for Clustering

At the time of writing, scenery requires a script to launch the remote clients that connect to the NodePublishers on the master node. On Windows, the utility `psexec` can be used, while on Linux `ssh` is the utility of choice. In Listing 8.2, a script is shown to launch scenery instances on a number of nodes with `psexec`: Lines 3 to 6 launch scenery instances on the machines wall1 to wall4, with user credentials given by `username` and `password`.

```
@echo off
echo "Running test %1"
call psexec -f -d -i -u username -p password \\wall1 -c run.bat %1 left
call psexec -f -d -i -u username -p password \\wall2 -c run.bat %1 front
call psexec -f -d -i -u username -p password \\wall4 -c run.bat %1 floor
call psexec -f -d -i -u username -p password \\wall3 -c run.bat %1 right

exit /b 0
```

Listing 8.2: `run-cluster.bat` for launching multiple scenery instances on different Windows machines via `psexec`.

Listing 8.3 shows the script that launches the individual instance on a machine with the required parameters: Line 2 creates a network share from a path on the master node containing the scenery application, and Line 2 runs scenery, activating fullscreen mode (line 3), activating framelock (line 4), declaring the master node address (line 6), screen name (line 7), and activating VR rendering from the start (lines 8 and 9). In this example, the screen configuration for each wall is determine by the system property `scenery.ScreenName`, as defined in Listing 8.1 on line 8.

```
net use S: \\master\scenery-base
java -cp "S:/scenery/target/*;S:/scenery/target/dependency/*" -Xmx16g^
 -Dscenery.RunFullscreen=true^
 -Dscenery.VulkanRenderer.UseOpenGLSwapchain=true^
 -Dscenery.Renderer.Framelock=true^
 -Dscenery.MasterNode=tcp://10.1.2.201:6666^
 -Dscenery.ScreenName=%2^
 -Dscenery.Renderer.Config=DeferredShadingStereo.yml^
 -Dscenery.vr.Active=true^
 org.junit.runner.JUnitCore %1 > S:\%2.log 2>&1
net use S: /delete /yes
```

Listing 8.3: `run.bat` for running a scenery instance on a node.

The remote clients can be launched directly from within the IDE, or individually, independently of the master process (e.g., manually from the command line). Initial or resumed connections are possible at any point in time. The program can be designed in two ways:

1. The local instance on master and clients handle scene construction themselves, or
2. scene construction is handled by the master node, and all changes and additions are communicated over the network to the clients.

Which of the strategies to choose depends on the problem at hand: Fully local initialisation can be beneficial if the data can be loaded from a local source for improved performance, while non-local construction is useful in the case that data can be loaded quickly from a common data source, such as a NAS (network-attached storage), SAN (storage-area network), or cluster file system.

In the future, we would like the extend the network capabilities of scenery in such a way that the remote-launch scripts are not necessary anymore, but a general-purpose client is run on the slave machines which will automatically connect to an active master node. Additionally, the synchronisation modes presented here can be

used for networking in general, meaning that remote multi-user environments are also possible with scenery, but have not been explored yet.

Chapter 9:

Miscellaneous Subsystems

We will briefly describe the remaining subsystems of scenery: Hub, Settings, Statistics, and OpenCL contexts. The Hub is the communications backbone of scenery, while the Settings store gathers user- and system-defined settings from various places and makes them available to scenery in a type-safe manner. The Statistics subsystem collects running averages of timings or frame rates, and an OpenCL context can be used to perform general-purpose computation on the CPU or GPU.

9.1 The Hub

All of scenery's subsystems — including the renderers, input handlers, VR devices, etc. — register with a hub, which is unique to an application.

A hub can be queried for the presence of a subsystem via `Hub.get(e: SceneryElement)`. This routine will either return the `SceneryElement` is has been asked for, or `null` if that subsystem has not been registered. A `SceneryElement` can be a renderer, OpenCL compute context, statistics collector, node publishers or subscribers, settings storage, and regular or natural input devices. A new `SceneryElement` may be added to a hub via the generic function `<T: Hubable> Hub.add(e: SceneryElement, obj: T)`, which will return the object that has been added to the hub.

With this design, it is possible to realise full applications that handle the application logic, rendering, input, and clustering, but also headless applications that run only the application logic, but do not produce any visual output, nor take any input. In this way, we can easily realise minimal unit tests or integration tests that only include the necessary subsystems for the test case at hand, while still enabling communication between the required subsystems.

Furthermore, by implementing the `Hubable` interface, the developer is able to add own subsystems.

9.2 Settings store

scenery uses a simple key-value store for settings. In the key-value store, values are fixed to their initial type once they have been set for the first time during an application run.

The settings system allows a bit of leeway in typing: for example, up- and downcasts, e.g. from `float` to `double` and vice-versa are possible, but a warning will be emitted to make it easier for the developer to track down errors, should they occur. This constrained flexibility allows to interoperate with scripting languages that do not exhibit different floating point types, such as JavaScript.

During startup, settings are gathered in the following order, with later ones taking precedence:

- scenery's default settings, usually defined in the interfaces of the various subsystems, such as `Renderer`, which, e.g., defines the supersampling factor[1] `Renderer.SupersamplingFactor` to be `1.0`.
- Configuration files, where the settings are stored in YAML format (by default, `.scenery/[applicationName].conf.yml` in the user's home folder),
- System properties from the command line. E.g., handing the optional flag `-Dscenery.Renderer=OpenGLRenderer` to the JVM on startup will override both scenery's default renderer setting, and the configuration file. System properties starting with `scenery.` are automatically translated to scenery settings, with the scenery setting name being the part after the dot.

[1] Supersampling refers to a rendering technique where images are rendered larger than necessary and downscaled later on for display in order to avoid aliasing artifacts.

New settings can be added via `Settings.set(name: String, contents: Any)`, or `Settings.setIfUnset(name: String, contents: Any)` if overwriting an existing setting — which might come from system properties or settings files — is not desired.

Settings can be queried at runtime via `<T> Settings.get(name: String, defaultValue: T? = null)`. The default value, which is optional, can provide a fallback if necessary. If a setting is not found or cannot be cast to the type requested, an exception is emitted.

For inspection all existing settings can be queried as a string via `Settings.list()`, or their names returned as a `List<String>`, via `Settings.getAllSettings()`.

9.3 Statistics

The Statistics subsystem can be used to collect information on timings or framerates. By default, the renderers use this subsystem to collect the timings of each rendering pass, and of the framerate. For all statistics collected, a running average over the last 100 values is kept.

A new statistic can be added by calling `Statistics.add(name: String, value: Float, isTime: Boolean)`, where `value` is expected to be in nanoseconds, if it is a time value. When `isTime` is false, the values are not converted to milliseconds for displaying. In addition to adding already calculated values, `Statistics.addTimed(name: String, lambda: () -> Any)` can be used to measure the runtime of any `lambda` function invocation, and add that as a statistic.

All statistics can be printed on standard output or REPL [2], by calling `Statistics.log()`. See Listing 9.1 for an example.

[2] scenery provides a REPL, or read-evaluate-print loop, to interactively issue commands during runtime.

```
Statistics - avg/min/max/stddev/last
OpenGLRenderer.updateInstanceBuffers   - 0.18/0.11/0.41/0.05/0.19
OpenGLRenderer.updateUBOs              - 0.38/0.24/0.68/0.09/0.33
Renderer.AO.renderTiming               - 0.14/0.08/0.30/0.04/0.15
Renderer.AOBlurH.renderTiming          - 0.11/0.06/1.63/0.16/0.11
Renderer.AOBlurV.renderTiming          - 0.14/0.06/1.83/0.24/0.12
Renderer.DeferredLighting.renderTiming - 0.28/0.15/1.43/0.15/0.40
Renderer.FXAA.renderTiming             - 0.07/0.04/0.17/0.02/0.07
Renderer.ForwardShading.renderTiming   - 0.53/0.30/2.68/0.32/0.55
Renderer.HDR.renderTiming              - 0.10/0.06/0.25/0.03/0.11
Renderer.Scene.renderTiming            - 0.28/0.18/1.81/0.16/0.29
Renderer.fps                           - 25.50/0.00/57.00/25.84/56.00
loop                                   - 0.00/0.00/0.00/0.00/0.00
ticks                                  - 515.50/466.00/565.00/28.87/565.00
```

Listing 9.1: Sample statistics output from scenery

In this example, all default statistics are shown: the timings of the individual renderpasses (e.g. `Renderer.HDR.renderTiming`), the frame rate `Renderer.fps`, and the timings for UBO and instance buffer updates (all in milliseconds).

9.4 OpenCL contexts

For general-purpose computations on the GPU, scenery offers access to OpenCL, if supported by either GPU or CPU. OpenCL can be used perform any kinds of computations that are not related to the regular graphics pipeline. In scenery, we use an OpenCL context (in the class `OpenCLContext`) for example to generate signed distance fields[3] for high-quality font rendering [Green, 2007, Chlumsky, 2015]. This technique requires high-resolution distance fields for each glyph of the font, which are subsequently downsampled and stored in an atlas for later access. This distance field can then be very efficiently sampled in a shader to find the (anti-aliased) outline of the font without storing the font as a texture at different resolutions. For an example rendering, see Figure 9.1.

[3] A distance field of a binary image stores the distance to the next border between foreground (white) and background (black) for each pixel. A signed distance field stores the signed distance, where (depending on convention) positive means inside, and negative means outside.

The creation of the distance fields of a glyph is quite simple and serves as an example how to use an OpenCLContext. The abridged code showing the most important parts for this process is given in Listing 9.2. For full code, see the `SDFFontAtlas` class in scenery (in `src/main/kotlin/graphics/scenery/fonts`), and `DistanceTransform.cl` (in `src/main/resources/graphics/scenery/fonts`) for the OpenCL kernel code.

```
// resolution of high-res distance field in pixels
val distanceFieldSize = 512
// create OpenCL context
val context = OpenCLContext()
// generate image of P character in given font, returns pair of
// character width and generated image byte buffer
val character = genCharImage("P", font, distanceFieldSize)

// wrap the byte buffer returned from genCharImage
// such that OpenCL can read it
val input = context.wrapInput(character.second)
// create an output buffer to store the distance field
val outputBuffer = ByteBuffer.allocate(distanceFieldSize * distanceFieldSize)
// wrap the output byte buffer such that OpenCL can
// write to it
```

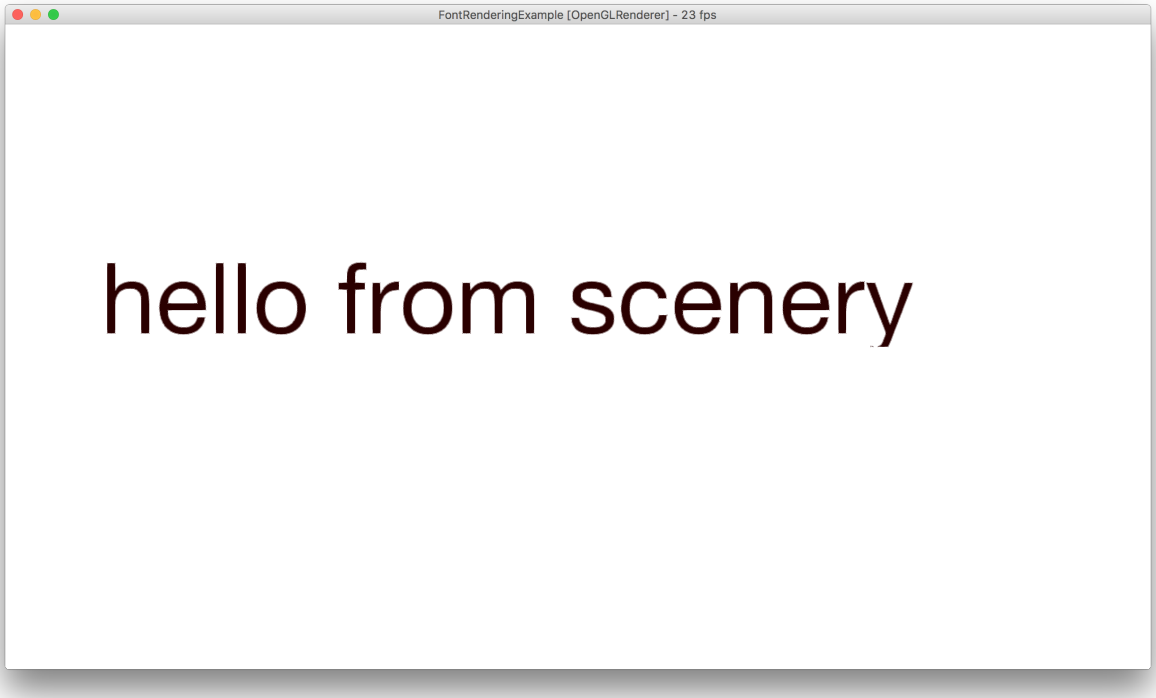

Figure 9.1: Font in scenery rendered from a signed distance field font atlas, generated via an OpenCL kernel. Note that the per-glyph distance field only has a size of at most 64x64 pixels, whereas the rendered resolution in this case is already 160 pixels in glyph height. For details, see text.

```
16 val output = context.wrapOutput(outputBuffer)
17
18 // load the signed distance field kernel
19 // and run it
20 context.loadKernel(kernelResource, "SignedDistanceTransformUnsignedByte")
21     .runKernel("SignedDistanceTransformUnsignedByte",
22     distanceFieldSize * distanceFieldSize,
23     input,
24     output,
25     distanceFieldSize,
26     distanceFieldSize,
27     maxDistance)
28
29 // read the result back into outputBuffer
30 context.readBuffer(output, outputBuffer)
```

Listing 9.2: OpenCL usage example.

Chapter 10:

Future Development Directions

10.1 Improved rendering

While scenery already supports a set of different rendering methods, and exchangeable and customisable rendering pipelines, we will extend the default rendering pipeline further with more state-of-the-art algorithms in the future:

- We would like to use more efficient ways of deferred shading, like forward+ shading [Harada et al., 2012] or clustered deferred shading [Olsson et al., 2012] — these variations of deferred shading support a large number of light sources, while integrating with regular forward shading effects and material pipelines more easily. Ideally, options for machines in several performance categories could be offered, as scenery will probably be used on notebooks as well as high-powered workstations.
- Volume rendering in scenery is still rather basic, supporting only alpha blending, and (local) maximum intensity projection. We would like to explore options like Monte Carlo light transport (e.g. as used in [Kroes et al., 2012]), or path tracing [Novák et al., 2018]. In addition to the added visual fidelity, such methods enable the precise simulation of light emitted by fluorescent proteins possible [Abdellah et al., 2017], opening the way to create better simulated data, e.g. for testing image analysis algorithms.
- Shadowing has not been implemented yet, but provides higher visual fidelity as well. We have started exploring various ideas in screen-space shadowing, with a preliminary rendering of the *Sponza* demo scene shown in Figure 10.1.
- The advent of ray tracing cores in consumer hardware, as in Nvidia's RTX series GPUs, offers new possibilities for global illumination (and shadowing). We would like to explore how these new possibilities can improve the rendering of experimental and simulated scientific data.

10.2 Improved networking

As described in Chapter 8, *Distributed Rendering*, scenery already provides facilities to synchronise scene content over a network. We would like to extend our networking implementation such that scientists can work collaboratively *and* remotely and jointly on datasets, without requiring large changes to the code of an application. Another

Figure 10.1: The *Sponza* demo scene rendered with screen-space shadowing in scenery with an experimental rendering pipeline. Sponza model from Morgan McGuire, Computer Graphics Archive, casual-effects.com/data

possibility is the collaboration between scientists in the same place via AR headsets, e.g., for discussing datasets and results while the dataset is hovering over the desk around which the scientists are sitting.

10.3 In situ visualisation in Virtual Reality

We have already started to explore the possibility of coupling scenery with simulation frameworks such as OpenFPM [Incardona et al., 2019] and ISAAC [Matthes et al., 2016]. We would like to explore VR-augmented visualisation of simulation data, while the simulation is running (*in situ* visualisation), and steering of the running simulation. With support for both virtual reality headsets and distributed rendering setups, as well as various devices for interaction, scenery is an ideal framework for this. We have already developed a prototype for a small molecular dynamics simulation, with the simulation code running in OpenFPM, and the visualisation on four rendering nodes, and one compositing node in scenery. This example is shown in the supplementary video under the QR code on the right.

Scan this QR code to go to a video demo of *in situ* visualisation in scenery. For a list of supplementary videos see https://ulrik.is/writing/a-thesis.

In situ visualisation provides further interesting research avenues:

- How can we efficiently decouple updates of the simulation data from the visualisation, such that the low latency and high frame rate requirements of VR can be met? Here, we have already prototyped a client-server architecture, where renderings are composited on the head node of a cluster, and then forwarded to a visualisation client that does the final compositing and VR rendering.
- How can the framework be designed to provide support for a wide range of simulation applications, yet require only minimal code changes? At the moment, we require only minimal code changes to include *in situ* visualisation in OpenFPM, but does that scale to all use cases? Could the simulation code provide metadata such that scenery can automatically discover visualisable or tuneable parameters of the simulation?
- How can we represent volumetric data efficiently, such that compositing of non-convex regions becomes possible, as domain decompositions from the simulation may be non-convex? In this case, an extension of *Volumetric Depth Images* (VDI)

[Frey et al., 2013] might provide a way out — VDIs are intermediate representation of volumetric data, where changes in the data below a certain threshold are compressed into a block representation, similar to run-length encoding. VDIs are then re-used for novel-view synthesis, where the rendering of a volumetric dataset is calculated from a new viewpoint, without traversing the entire dataset again.

Part III:

Case studies

> At its very core, virtual reality is about being freed from the limitations of actual reality. Carrying your virtual reality with you, and being able to jump into it whenever and wherever you want, qualitatively changes the experience for the better.
>
> —John Carmack

Chapter 11:

Bionic Tracking: Using Eye Tracking for Cell Tracking

The work presented in this chapter has been done in collaboration with Kyle I.S. Harrington (University of Idaho, Moscow) and Raimund Dachselt (TU Dresden), and has been published as:

Günther, U., Harrington, K.I.S., Dachselt, R., Sbalzarini, I.F.: *Bionic Tracking*: Using Eye Tracking to Track Biological Cells in Virtual Reality. *BioImageComputing at ECCV 2020*. arXiv preprint 2005.00387.

We are going to detail the *Bionic Tracking* strategy for augmenting biological tracking tasks for 3D data over time with eye gaze data in order to make them easier and faster to do. Bionic Tracking utilises a combination of virtual reality and eye tracking in order to do so.

We will first discuss the tracking problems usually encountered in biology and then detail the design process that went into Bionic Tracking, and finally show a proof-of-concept that the strategy works in the case of tracking cells during the early development of *Platynereis* embryos.

11.1 Tracking Problems in Biology and Challenges

In cell and developmental biology, the image data generated by fluorescence microscopy is often only a means to an end: Many tasks require exact information about the position of cells during development, or even their entire history, the so-called cell lineage tree.

Images from fluorescence microscopy don't have any well-defined intensity scale — in contrast to CT images, for example, where a specific intensity indicates a particular tissue type — and intensity might even vary across a single cell. It can therefore be very hard to follow a moving, dividing, or dying cell around in a developing tissue or organism. Often, the task of tracking cells is done manually over a series of time points, which can be a very tedious process. In the past, tracking software was often developed for a specific model organism, e.g., for *C. elegans*, and relied on their stereotypical development to succeed in tracking their cells. Such approaches however either fail entirely, or produce unreliable results for larger organism. For that

reason, (semi)automated approaches have been developed which are independent of the model organism and can track large amounts of cells:

- *TGMM*, Tracking by Gaussian Mixture Models [Amat et al., 2014, 2015], is an offline tracking solution that works by generating oversegmented supervoxels from the original image data, then fit all cell nuclei with a Gaussian Mixture Model and evolve that through time, and finally use the temporal context of the various lineages to create the final lineage.
- *TrackMate* [Tinevez et al., 2017] is a plugin for Fiji [Schindelin et al., 2012] that provides automatic, semi-automatic, or manual tracking of single particles in imaging datasets. TrackMate is highly customisable and allows the user to even extend it with self-developed spot detection or tracking algorithms.
- *MaMuT*, the Massive MultiView Tracker [Wolff et al., 2018], is in some sense the successor to TrackMate, allowing the user to track cells in large datasets, often originating from lightsheet microscopes, and containing multiple different views. MaMuT's viewer is based on BigDataViewer [Pietzsch et al., 2015], and is able to handle very large amounts of data.

All of these automated approaches however have in common that they need manual curation as a final step, as all of them do make assumptions about cell shapes, modelling them as Gaussian blobs, as in the case of TGMM. The main problem we are going to address in this chapter is the manual curation step, which might be needed either for verification, or to handle especially difficult developmental stages, where cell shapes may vary wildly.

The challenges of this step are twofold:

- Image data from fluorescence microscopy can be very inhomogeneous, with inhomogeneity stemming from both the distribution of fluorescent proteins, and from the diversity of cell or nuclear shapes. Deriving general algorithms that can capture both regular and very deformed cells or nuclei is a highly challenging task.
- Manually curation of cell lineages is very tedious at the moment, as the users have to go through each timepoint and connect cells between the timepoints. This is often done on a per-slice basis and by mouse, leading to a time-consuming process. Manually tracking a single cell through 100 timepoints with this process can take up to 30 minutes, and tracking a single dataset whole can take many months.

11.2 Design Space and Related Work

Bionic Tracking is powered by a combination of eye tracking and virtual reality:

A user is tasked to follow a cell with her eyes, the gaze direction recorded, and the targeted cell then determined, turning the 3-dimensional localisation problem into a 1-dimensional one — from the whole volume of image data, to a set of rays through it. As described in Chapter 2, *Introduction to Visual Processing*, the human visual system excels in following moving objects smoothly, and in datasets used for cell tracking, the cells are also assumed to be moving smoothly.

Gaze in general has been used in human-computer interaction for various kinds of interactions. Briefly, it has been used as an additional input modality in conjunction with touch interaction [Stellmach and Dachselt, 2012] or pedaling [Klamka et al., 2015], or for building user interfaces on top of it, e.g. for text entry [Lutz et al., 2015]. However, the particular kind of eye movements we are exploiting here mainly – *smooth pursuits* — are under-explored in human-computer, especially for 3D interaction: In [Kosch et al., 2018], the authors use deviations from smoothness in smooth pursuits to evaluate cognitive load, while in [Vidal et al., 2013], smooth pursuits are used for item selection in 2D user interfaces. To the author's knowledge, only [Piumsomboon et al., 2017] use smooth pursuits for 3D interaction in their *Radial Pursuit* technique, where the user can select an object in a 3D scene by tracking it with her eyes, and it will become more "lensed-out" the longer she focuses on a particular object.

With the addition of virtual reality, we give the user a tool for enhanced navigation and cognition when exploring a complex 3D dataset [Slater and Sanchez-Vives, 2016], and second, utilise the tracking data from the head-mounted display, consisting of head position and head orientation, for constraining the eye tracking data to remove outliers from the gaze data, e.g., by calculating the quaternion distance between eyeball rotation and head rotation. In addition, head tracking data from the HMD can be used to foveate the rendering of the volumetric dataset, dimming areas the user is not looking at [Levoy and Whitaker, 1990, Bruder et al., 2019]. We have not yet explored foveation (e.g., to boost tracking rates) yet. In the context of biological or biomedical image analysis, VR has been applied e.g. for virtual colonscopy [Mirhosseini et al., 2019] or for tracing of neurons from connectome data [Usher et al., 2017].

Let us take a critical look at whether only one of the two technologies could be sufficient in achieving our goal of making manual cell tracking faster and more comfortable:

- When *removing eye tracking*, the head orientation could still be used as a cursor. However, following small and smooth movements with your head is not something humans are used to, the eyes will always lead the way, and the head will follow via the vestibulo-ocular reflex.
- When *removing virtual reality*, the effective "canvas" the user can use to follow cells around become restricted to the small part of the visual field a regular screen occupies. Alternatively, large screens, such as Powerwalls, could be used, but these also do not offer the freedom of movement that virtual reality headsets offer, especially when the user needs to move inside the dataset, or even evade objects while tracking a cell.

In terms of the design space for gaze interaction on head-mounted displays introduced by [Hirzle et al., 2019], we utilise (stereoscopic) VR with full world information, combined with binocular eye tracking.

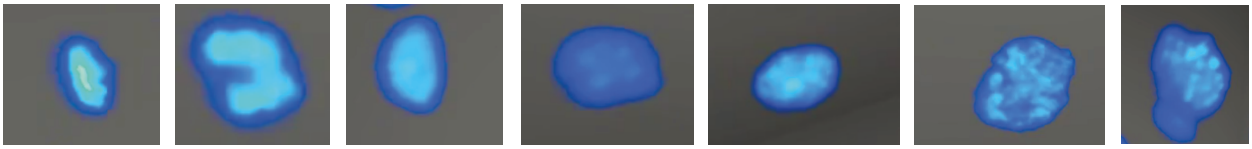

Figure 11.1: Diversity of nuclear shapes in early *Platynereis* development, taken from the first 100 timesteps of a developmental timelapse. Dataset courtesy of Mette Handberg-Thorsager, Tomancak Lab, MPI-CBG.

11.3 Tracking cells in early Platynereis development

Platynereis dumerilii is an annelid, a segmented worm. Its embryonic development has a very characteristic feature, *spiral cleavage* where dividing cells turn in spiral form during their division. Arising from this geometric peculiarity, a wide variety of cell shapes can be found in developing *Platynereis*. Their membranes are inherently hard to segment, also due to the stochastic distribution of fluorescent markers. Alternatively, nuclei can be tracked, but their shapes vary as well, with some examples shown in Figure 11.1.

Scan this QR code to go to a video showing the early development of a *Platynereis* embryo. For a list of supplementary videos see https://ulrik.is/writing/a-thesis.

11.4 Design Process

11.4.1 Initial Prototype

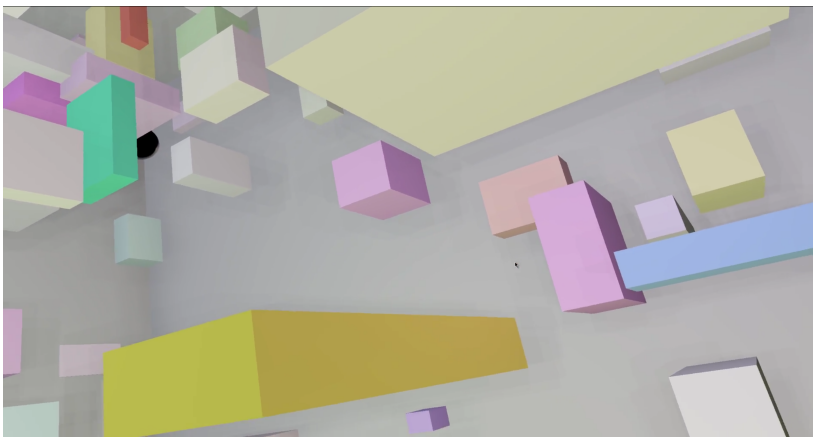

Figure 11.2: 2D Screenshot of the attentive tracking prototype. The sphere to be tracked can be seen in the upper left corner of the image. See the text for details.

For the initial prototype, within *scenery* we created a virtual reality-based crowded environment consisting of many differently-sized and differently-colored boxes.

This prototype was tested with an HTC Vive on a set of 5 people familiar with tracking problems, either from the biological or algorithmic side. The participants were not told that they are going to perform a tracking problem in order to prevent priming them.

In the presented scene, a black sphere is performing random motions in the a crowded space, and the participant was instructed to follow this sphere, and not lose sight of it. A screenshot of the prototype can be see in Figure 11.2. While participants stated that manual tracking is usually perceived as a tedious, boring and annoying task, they described following the sphere in VR as interesting, easy and fun.

Encouraged by the positive reactions to the first simple prototype, a next, more serious prototype was planned. Just tracking head orientation and position would not be enough for the precision required, so an eye-tracking solution was integrated into the HTC Vive to gain access to more detailed information on where the user is looking at any given point in time.

11.4.2 Selecting the eye tracking hardware

For the this project, we have chosen the *Pupil* eye trackers produced by *Pupil Labs*[Kassner et al., 2014][1], as they provide a solution that provides both open-source software and very competitively-priced hardware that is simple to integrate into HMDs. The software offered is available as LGPL-licensed open-source software on Github (https://github.com/pupil-labs) and can be extended with custom plugins.

[1] The *Pupil* HMD-based eye tracker from Pupil Labs, see https://www.pupil-labs.com.

In addition to being open, data gathered by the software is available to external applications via a ZeroMQ-based protocol — in contrast to closed-source proprietary libraries required by other products — which even enables the use of the eye tracking data over the network.

At the time of writing, HTC's extended version of their *Vive Pro* HMD, the *Vive Pro Eye*, with integrated eye tracking hardware, is also becoming available. It will be interesting to compare the two solutions in the future, especially as the *Vive Pro Eye* will be more competitively priced (around EUR1400) than a regular *Vive* combined with the *Pupil* eye trackers (EUR600 for the HMD, plus EUR1100 for the eye trackers).

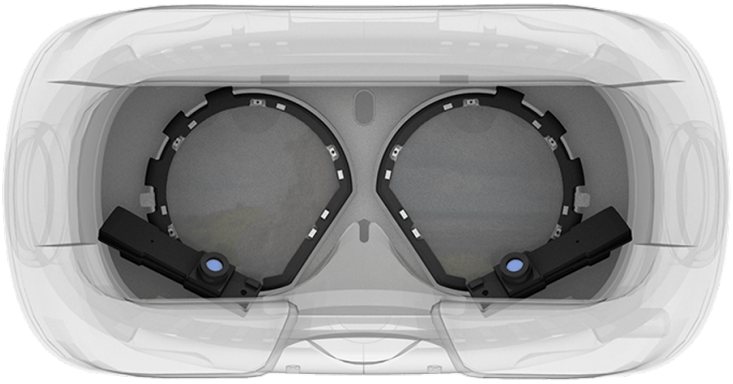

Figure 11.3: The Pupil eye tracking cameras integrated into a HTC Vive HMD. The cameras view the eyes of the user from below, while the eyes are illuminated by a set of IR LEDs position around the lenses of the HMD. Image reproduced from www.pupil-labs.com.

An image of how the eye tracking solution looks integrated into a HTC Vive HMD can be seen in Figure 11.3.

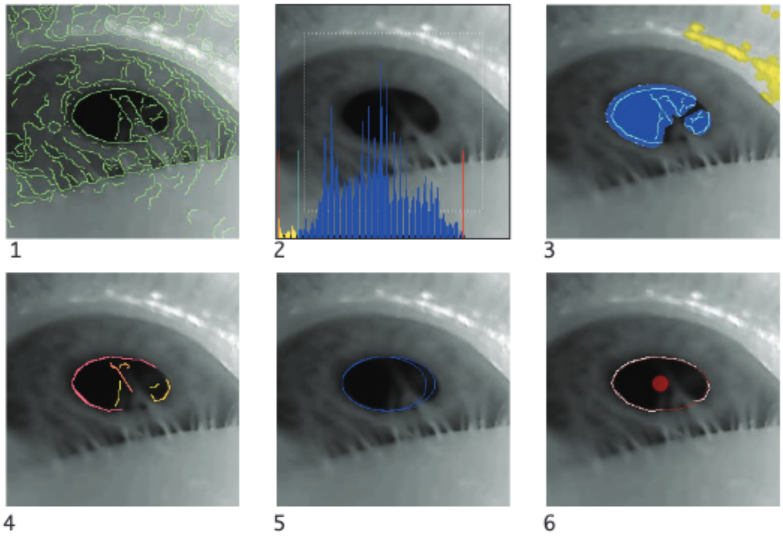

Figure 11.4: Pupil detection in the *Pupil* software. See text for a description of the steps. Image reproduced from [Kassner et al., 2014].

.

11.5 Pupil detection and calibration

11.5.1 Pupil 2D and 3D detection

Using three user-defined parameters, pupil intensity range, and pupil min/max diameter, *Pupil* extracts the pupil from the camera images as follows (from [Kassner et al., 2014]):

1. A Canny edge detector is applied to the camera image (Figure 11.4-1),
2. Darker regions of the image are selected, based on the pupil intensity range parameter, the offset of the first histogram maximum, edges outside this area are discarded (Figure 11.4-2),
3. the remaining edges are filtered to exclude specular reflections, such as from the infrared LEDs, then the remaining edges are extracted into contours using connected components (Figure 11.4-3, spectral reflections in yellow),
4. the remaining contours are filtered and split into sub-contours based on the continuity of their curvature (Figure 11.4-4),
5. candidate pupil ellipses are formed using least squares fitting, with the major axis within the bounds of the pupil min/max parameter, and a combinatorial search is done on the remaining contours to see which might be added to the ellipse for additional support. Resulting ellipses are evaluated based on the ratio of supporting edge length and ellipse circumference, called *confidence* in *Pupil*, and finally
6. if the best result's confidence is above a defined threshold, the candidate ellipse is reported as result, otherwise the algorithm returns that no pupil has been found.

If 3D detection is selected in *Pupil*, the result ellipse is passed on to the algorithm described in [Swirski and Dodgson, 2013]. As we are only going to use 2D detection for *Attentive Tracking*, we are not going to detail this algorithm, but refer the interested reader to the paper instead.

We have found that the the combination of high-resolution camera images (resolution over 640x480 pixels) in combination with the Canny edge detector used in the first step of the algorithm leads to contour overdetection, and therefore useless pupil detections. We therefore used a camera resolution of 640x480 pixels, which provides the best tradeoff between speed, processing cost, and accuracy. The used cameras provide a framerate of 120fps at this resolution, which in turn leads to a high-enough temporal resolution for tracking eye movements, as the framerate of the HMD is 90 fps maximum. We are currently exploring alternatives to the algorithm described, based on genetic algorithms.

Although by using vergence as binocular depth cues, 3D detection could yield additional constraints on at what depth the user was looking, in the following, we use the 2D detection algorithm, which we found to be more reliable.

11.5.2 Calibration procedure

Eye positions, size, etc. are subject to large individual differences. It is therefore required to calibrate the eye trackers before each use, to be able to get reliable gaze data out.

In case of regular, glasses-mounted eye trackers, *Pupil* offers an integrated calibration procedure, while for HMD-based settings, we need to create our own calibration routine. Our custom calibration routine works as follows and is based on the one used in the `HMDeyes` example from Pupil Labs:

1. We show the user the images recorded by the two eye tracking cameras in the VR HMD, such that the position of the headset can be adjusted so the eyes can be detected well by *Pupil*,
2. when the user starts the calibration procedure, we instruct her to follow the points shown on-screen. First, we show a single highlighted point in the center of the screen,
3. after acquiring enough samples for calibration (scenery defaults to 120 samples per point, discarding the first 15 to remove samples where the eyes where potentially still moving), the screen space position of the calibration point is sent to Pupil,
4. the next counter-clockwise point out of a set of 6 equidistant points on the circle is shown to the user,
5. after all 7 points have been gazed at by the user, Pupil's calibration routine will try to construct a correspondence between the gaze vectors of both eyes and the screen-space coordinates submitted,
6. if the calibration routine is successful, the calibration interface will be hidden, and the regular application can continue. If the calibration routine is not successful, we restart from 1. until successful or until the user cancels.

The user can now continue with tracking cells in the loaded dataset.

Scan this QR code to go to a video showing the calibration procedure for *Bionic Tracking*. For a list of supplementary videos see https://ulrik.is/writing/a-thesis.

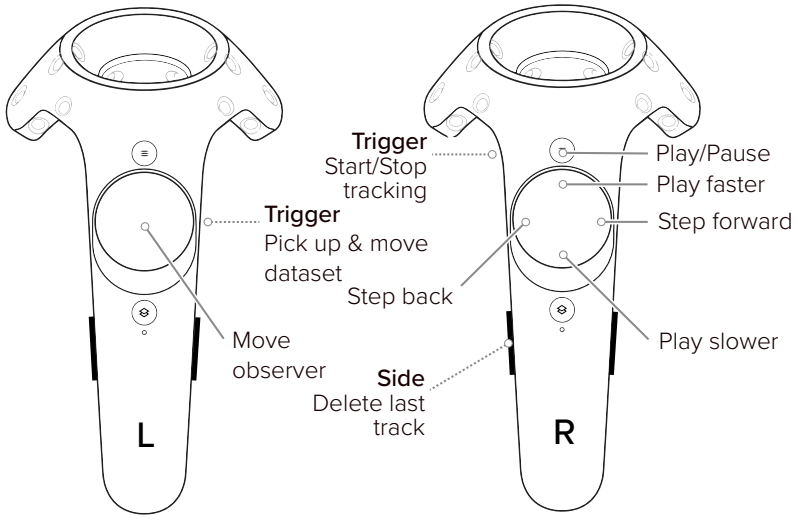

Figure 11.5: Controller bindings for using *Bionic Tracking*. See text for details. Vive controller drawing from VIVEPORT Developer Documentation, developer.vive-port.com.

11.6 Tracking Procedure

After calibration and before starting the tracking procedure for a single cell, the user can position himself freely in space, and also move to the right position in time for the dataset. All of these functions can be performed using the HTC Vive handheld controllers. The controller bindings are shown in Figure 11.5, with them the user can perform the following:

- move the dataset by holding the left-hand trigger and moving the controller around,
- use the directional pad on the left-hand controller to move forward, back, left, right, with respect to the direction the user is looking into,
- start and stop tracking by pressing the right-hand trigger,
- play and pause the dataset by pressing the right-hand menu button,
- playing the dataset faster or slower by pressing the right-hand directional pad up or down, and
- stepping through the timepoints of the dataset by pressing the right-hand directional pad left or right.

To adjust for handedness of the user, the controller mappings can be swapped.

The user can start the tracking process as soon as she is ready and has found the cell she wants to track. Starting a tracking step will start playing the dataset if it is currently paused. When tracking is active, gaze directions and other metadata are collected, and can be analysed automatically in the next step. The limitation at the moment is that the user *has* to look at a trackable object when the tracking step is started, in order to seed the analysis algorithm.

11.7 Analysis of Eye Tracking Data

After all rays have been collected for a tracking step, all further track data is derived from the set of rays, which we call the *hedgehog*. An individual ray we dub *spine*, as it contains more information than just the ray's orientation and direction. In particular, it contains:

- the confidence of the gaze data point it was derived from,
- the entry and exit points through the volume in volume-local coordinates (meaning for each coordinate axis $\in [0.0, 1.0]$),
- the head orientation at recording time,
- the head position at recording time,
- the timepoint of the volume it belongs to, and
- a list of samples taken in uniform spacing along the ray, along with the spacing.

How this collection looks visually in 3D is depicted in Figure 11.6 (when visualising spines, we usually only show the spatial extent of them, and show their associated confidence as color code — all data associated with the spine is however used in the analysis). In the depicted image, the tracking user's position is the intersection of all spines, which are shown here in an elongated way for visualisation purposes. In this case, the spines are color-coded by the timepoint of the volumetric dataset.

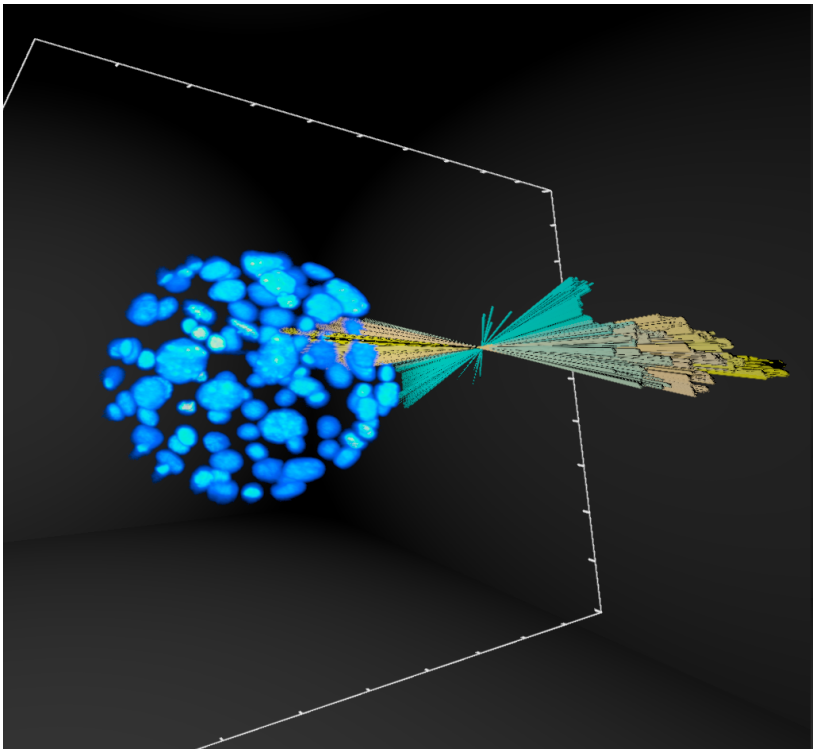

Figure 11.6: The hedgehog of a tracking step of a single cell through 100 timepoints in a *Platynereis* dataset. In the dataset, a Histone marker is used for fluorescence. Each spine is color-coded by timepoint, with early timepoints shown in green, and later ones in yellow. Dataset courtesy of Mette Handberg-Thorsager, Tomancak Lab, MPI-CBG.

Additionally, the hedgehog can be represented in two dimensions, with time on the Y axis, and depth along a given ray along the X axis. An example of that is shown in Figure 11.7. In this figure it is clearly visible that the rays do have different lengths, which is due to the angle they intersect the dataset. Also note that each line on the Y

axis represents one gaze sample, of which we collect up to 60 per second, leading to 1614 samples in the plot (16 samples per timepoint on average).

Figure 11.7: The raw plot of the hedgehog rays. On the Y axis, volume intensity along a single ray is shown, on the X axis, time runs from top to bottom. See text for details.

The data can also be smoothed with a moving window average over time. An example of that with the same dataset as before is shown in Figure 11.10. In this plot we additionally show the local maxima along each ray in red. The track of the cell we were following is clearly visible. As there are movements of the user, and other cells or objects might appear in front of the cell the user is tracking, the challenge is now how to reliably use the temporal information contained in the hedgehog to find a track for the cell.

11.7.1 Graph-based temporal tracking

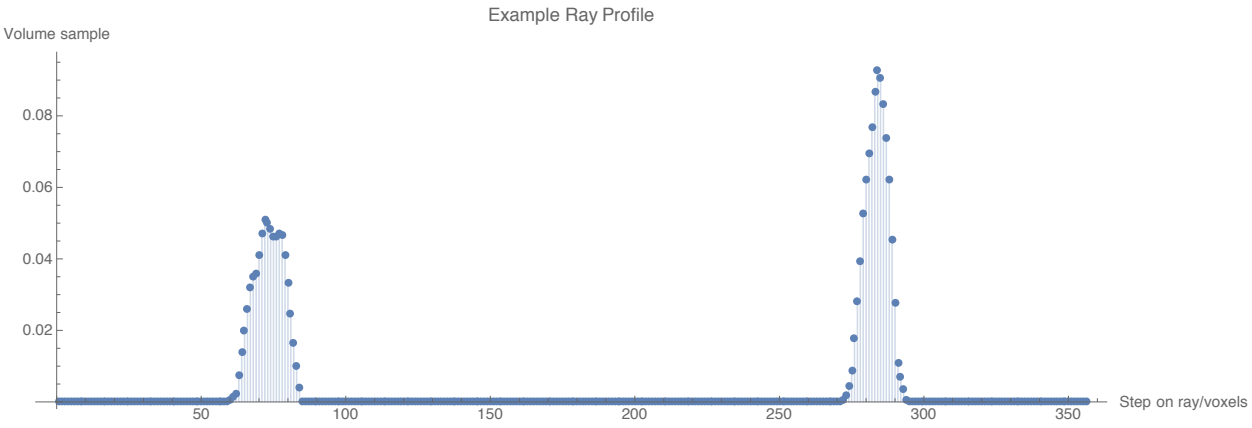

Figure 11.8: An example profile of an entire ray through a volumetric dataset. X axis is step along the ray in voxels, Y axis volume sample value. In this case, there are two local maxima along the ray, one close to the observer, at index 70, and another one further away at 284.

To reliably connect cells together over several timepoints, we use an incremental graph-based approach utilising all spines that have local maxima in their sample value. An example ray through a volume is shown in Figure 11.8. In the figure, the position in voxels along the ray is shown on the X axis, while the Y axis shows the value of the volume at that point of the ray. We assume that when starting a tracking step, the user is looking at an unoccluded object that will be visible as a local maximum along the ray to seed the algorithm.

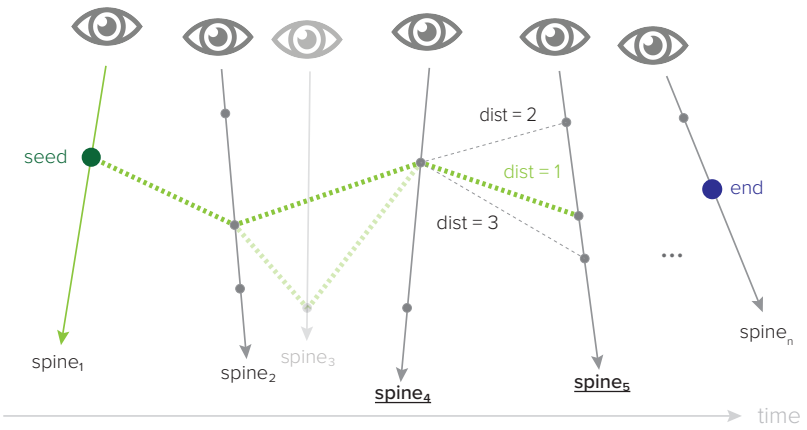

Figure 11.9: A graphical illustration of the incremental graph-search algorithm used to extract tracks from a hedgehog. Time runs along the X axis. spine_1 is the initial seed point where to start tracking. The algorithm is currently at spine_4, determining how to proceed to spine_5, which has multiple possible cell detections. In this case, the middle track with $d = 1$ wins, as it is the shortest world-space distance away from the current point.

For each timepoint, we have collected a variable number of spines, whose count varies between 0 and 60 (with an average of 16) — zero spines might be obtained in case that the user blinks and no detection was possible. To connect the initial seed point with the other correct spines correctly, we step through the list of spines one-by-one, performing the following steps:

1. Advance to the next spine,
2. connect the currently active point from the previous spine with the local maximum on current next spine that has the lowest world-space distance — with this weighting we can exclude cases where another object was briefly moving between the user and the actually tracked object. The process of connecting one local maximum to the next closest one is a version of *dynamic fringe-saving A*-search* [Sun et al., 2009] on a grid, where all rays get extended the the maximum length in the whole hedgehog along the X axis, and time flows along the Y axis.

A graphical representation of the algorithm is given in Figure 11.9 and the algorithm itself is summarised in Algorithm 1.

Data: Hedgehog $\mathcal{H}$ with spines $\mathcal{S}$
Result: Track $\mathcal{T}$, consisting of points p_i

$v_{\text{current}} \leftarrow \mathcal{S}$ first;
forall *Spine* $s \in \mathcal{H} \setminus v_{\text{current}}$ **do**
  `/* Find index of the closest local maximum */`
  $i_{\text{closest}} \leftarrow$ `FindIndexOfClosestMax` $(v_{\text{current}}, \|\cdot\|^2)$;
  `/* If no maximum found, skip to next spine */`
  if $i_{\text{closest}} == \textit{null}$ **then**
    skip to next s;
  end
  v.position $\leftarrow s$.origin $+ s$.direction $\cdot i_{\text{closest}}$;
  $v_{\text{current}} \leftarrow v$;
  $\mathcal{T} + v$
end

Algorithm 1: Algorithm for evaluation of the hedgehog in *Bionic Tracking*. See text for a detailed explanation of the steps.

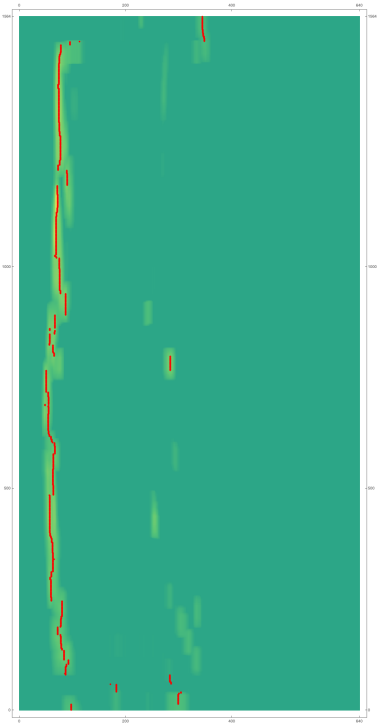

Figure 11.10: The same hedgehog with local maxima marked. On the Y axis, volume intensity along a single ray is shown, on the X axis, time runs from top to bottom. Local maxima are shown in red. See text for details.

11.8 User Study

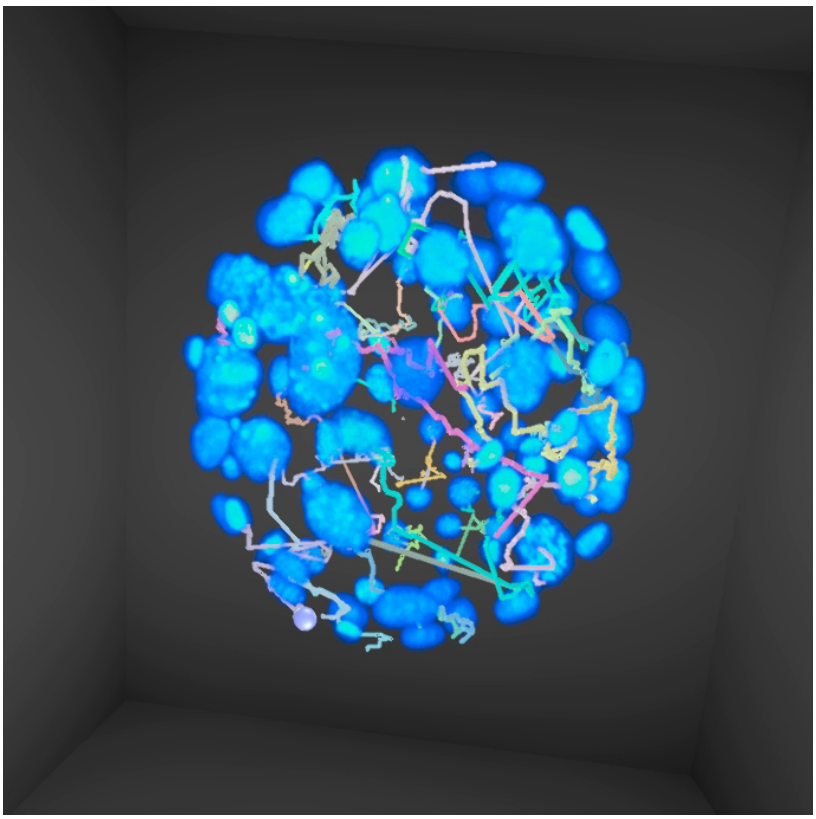

Figure 11.11: 52 cell tracks created by the author for a 101 timepoint time-series dataset of a*Platynereis* embryo. The tracks were created in about 40 minutes. See the supplementary video for the creation of a single track, and a debug visualisation showing intersections with the nucleus.

In order to evaluate the performance and usability of the Bionic Tracking method, we have conducted a user study with seven experts in either manual or algorithmic cell tracking, or both (median age 36, s.d. 7.23, 1 female, 6 male). In the study, the users were given the task to track cells in the *Platynereis* dataset also featured in Figure 11.11. One of the participants was already familiar with the dataset. The user study was conducted on a Dell Precision Tower 7910 workstation (Intel Xeon E5-2630v3 CPU, 8 cores, 64 GB RAM, GeForce GTX 1080Ti GPU) running Windows 10, build 1909, with a HTC Vive VR headset equipped with eye trackers by Pupil Labs.

The users who participated in the study had no or very limited experience with using VR interfaces up to this point (5-point scale, 0 means no experience, and 4 daily use: mean 0.43, s.d. 0.53), only one of them had previously used an eye-tracking-based user interface. (same 5-point scale: mean 0.14, s.d. 0.37).

11.8.1 Procedure

Before starting the experiment, the users were informed of goals and potential risks of the study (e.g. simulator sickness). In a questionnaire that was split into a pre-experiment and a post-experiment part, the users were asked about the presence of any motor or visual impairments, previous VR experience, and their current wellbeing. The full questionnaire is available in Appendix C.

The users then got a quick introduction into the software and into VR environ-

ments in general, if necessary. After the fit of the headset was ensured, the eye trackers were calibrated. The users were then asked to create as many tracks as they liked and are comfortable with. If any of the created tracks did not satisfy them, the offending track could be deleted.

After the experiment was done, the post-experiment part was filled out, in this part users had to judge the usability and suitability of the software, were asked again for their wellbeing, and in addition had to rate their experience with both the NASA TLX questionnaire [Hart and Staveland, 1988] and the Simulator Sickness Questionnaire (SSQ, [Kennedy et al., 1993]). The questions about the usability and suitability of the software were based on both the System Usability Score [Brooke, 1996] and the User Experience Questionnaire [Laugwitz and Held, 2008].

As final element of the study, a free-form interview was conducted in which the users could comment about the software, and suggest improvements.

11.8.2 Results

In the experiment, users created up to 32 cell tracks in 10 to 29 minutes.

The average SSQ score was 25.6 ± 29.8 s.d. (median 14.9), approximately on par with other VR applications that have been evaluated using SSQ [Singla et al., 2017]. For the NASA TLX score, we used all categories (mental demand, physical demand, temporal demand, success, effort, insecurity) on a 7-point scale where 0=Very Low and 6=Very High for the *demand metrics*, and 0=Perfect, 6=Failure for the *performance metrics*. Users reported medium scores for mental demand (2.71 ± 1.70) and for effort (2.86 ± 1.68), while reporting low scores for physical demand (1.86 ± 1.95), temporal demand (1.57 ± 0.98), and insecurity (1.14 ± 1.68). Most importantly, the participants did judge themselves to have been rather successful with the cell tracking tasks (1.71 ± 0.75).

The users explicly expressed interest in using Bionic Tracking for their own tracking tasks (3.43 ± 0.53; 5-point scale here and for the following questions: 0=No agreement, 4=Full agreement). The tracks created were judged to look reasonable (2.57 ± 0.98), and Bionic Tracking was deemed to provide an improvement over their current manual tracking methods (3.14 ± 0.90). Furthermore, the users stated that they could create new cell tracks not only with reasonable confidence (2.86 ± 0.69), but much faster (3.29 ± 0.76). Users also found the software to be relatively intuitive (2.43 ± 0.98) and did not need long to learn how to use it (0.59 ± 0.79). Especially the ergonomics of the method were remarked about in the follow-up interviews:

> ”It was so relaxing, actually, looking at this [cell] and just looking.” (P2, the user remarked further after the interview that the technique might prevent carpal tunnel issues often encountered when tracking using mouse and keyboard.)

> ”I figured this could be like a super quick way to generate the [cell] tracks.” (P7)

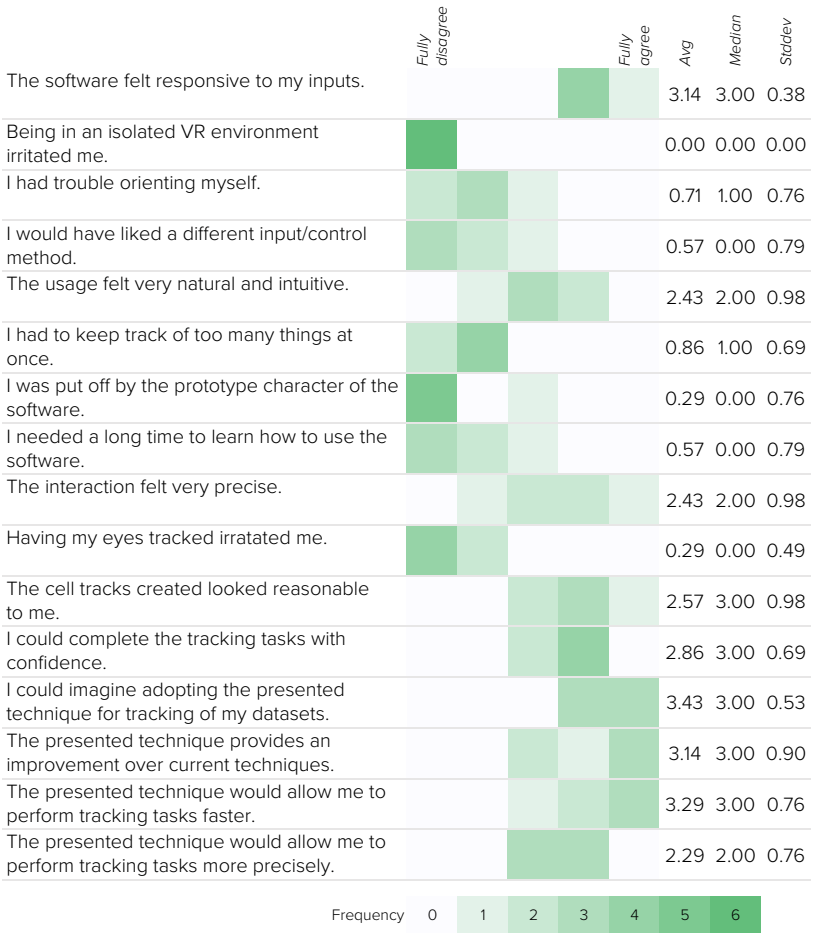

Figure 11.12: Results of usability and acceptance question from the user study. Note that the questions are formulated both positively and negatively.

The results from all questions related to software usability and acceptance are summarized in Figure 11.12.

We made two more interesting observations in the user study:

First, we saw that users adjust playback speed more often than image size in VR. After exploring different settings – users could choose speeds from 1-20 timepoints/second – all users independently settled on a playback speed of 4-5 timepoints/second for tracking, corresponding to 200-250 ms of viewing time per timepoint, which coincides with the onset delay of smooth-pursuit eye movements (see Section 2.2.1, *Eye movements*, and [Duchowski, 2017]). The chosen visual size of the dataset was also usually chosen to be approximately human-scale (which was also the default setting, but experimented with by the users).

Second, despite having no or limited previous VR or eye tracking experience, the users did not at all feel irritated by the environment (0.00 ± 0.00), nor by the use of eye tracking (0.29 ± 0.49).

Our preliminary results and user study show that cell tracks can be reliably reconstructed by "just looking at them", using eye, head and body movements that are used in everyday life. Importantly, the users estimated that the Bionic Tracking

method would yield a speedup of a factor 2 to 10 (3.33 ± 6.25) compared to tracking cells with a 2D interface.

11.9 Discussion and Future Work

In this chapter we have introduced the *Bionic Tracking* strategy for tracking cells in 3D microscopy images in an effort to speed up manual tracking and proofreading and developed a proof of concept. Preliminary results show that we might be able to achieve approximately an order of magnitude speedup compared to manually tracking cells. Before we can bring this strategy into actual use for biologists, we need to do two more things:

- First, implement interactions that allow to track or proofread lineage trees. Such an interaction could for example include the user pressing a certain button whenever a cell division occurs, and then track until the next cell division, and
- Second, Bionic Tracking has to benchmarked against other automatic solutions, e.g. on cell tracking challenge datasets (see e.g. CellTrackingChallenge, [Ulman et al., 2017]).

We foresee the limitation that for tracking large lineages entirely, Bionic Tracking will not work, simply for combinatorial reasons. It can however be used to track early-stage embryos where cells may have less-defined shapes, or it may provide constraints to training data to machine learning algorithms. Furthermore, Bionic Tracking could be used in a divide-and-conquer manner in conjunction with an automatic tracking algorithm that provides uncertainty scores, and only be applied in regions where the algorithm cannot cross a given uncertainty threshold. We could further increase the usefulness of Bionic Tracking by not just searching for local maxima along rays, but actually extract the centroids of cells.

Ultimately, we would like to integrate Bionic Tracking into existing tracking software, such that it can be helpful for a more general audience. Current developments in eye tracking hardware indicate falling prices in the near future, such that those devices might become way more common soon. Alternatively, one could imagine just having one or two eye tracking-enabled HMDs, and make them available to users in a bookable item-facility-like manner.

Scan this QR code to go to a video showing tracking of a cell via *Bionic Tracking* in early *Platynereis* development. For a list of supplementary videos see https://ulrik.is/writing/a-thesis.

Chapter 12:

Towards Interactive Virtual Reality Laser Ablation

The investigation of biological phenomena not only rests on observation of such, but also on the ability to interfere with them. Especially where biomechanical and biophysical questions need to be answered, *laser ablation* or *microsurgery* plays an important role. In laser ablation, a high-powered, and usually pulsed, UV or IR laser is used to destroy cells or parts of tissues precisely, while not interfering with their neighbours, in a manner much more precise than purely mechanical manipulation could achieve.

Nowadays, experiments in this realm are carried out with simple slice-based 2D user interfaces, while the specimen and processes investigated get spatiotemporally more and more complex.

In this chapter, we introduce the use of virtual reality to microsurgery to overcome this problem. We present two prototypes that we developed to investigate user satisfaction and compatibility, and propose a microscope design that will incorporate an ablation system that can be steered in virtual reality. The prototypes presented require flexible visualisation of both geometric data and large, time-series volumetric data, as well as integration of additional hardware such as VR HMDs and controllers, we show how our visualisation framework, introduced in [scenery — Democratising VR/AR Visualisation for Systems Biology], enabled those developments. First, we start with an introduction to microsurgery and the underlying biophysical principles.

12.1 Introduction to Microsurgery

For the interaction of light with biological tissue, five different regimes exist, depending on the applied power density. These are shown in table 12.1[Niemz, 2019].

Table 12.1: Regimes for light interacting with biological tissue.

Regime	Power density (W/m^2)	Exposure time (s)
Photochemical interaction	$10^{-3} - 10$	10+
Thermal interaction	$10 - 10^{6}$	$10^{-5} - 10$
Photoablation	$10^{6} - 10^{10}$	$10^{-9} - 10^{-6}$
Plasma-induced ablation	$10^{10} - 10^{14}$	$10^{-13} - 10^{-10}$

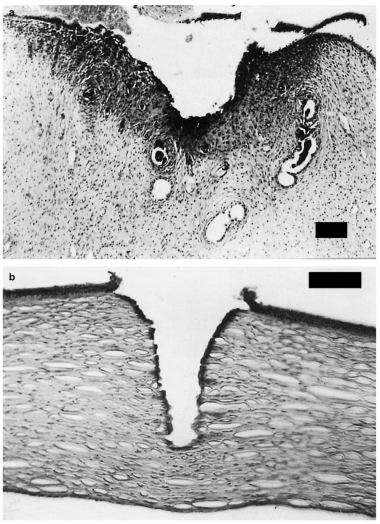

Figure 12.1: Coagulated tissue samples, a: Uterine tissue of a Wistar Rat, using a 10W continious wave laser, b: Human cornea coagulated with 120 pulses of 5 mJ from an Er:YAG laser. Reproduced from [Niemz, 2019].

Regime	Power density (W/m^2)	Exposure time (s)
Photodisruption	$10^{11} - 10^{16}$	$10^{-12} - 10^{-8}$

In case of *photochemical interaction*, chemical reactions are triggered by the application of laser light, with most of the effects originate from decay products of these chemicals, and not the laser itself [Niemz, 2019].

Thermal interaction in turn is characterized by extended tissue damage due to vaporisation, coagulation, carbonisation or melting. This is not desirable for precise manipulation on the cellular or tissue level, as clearly visible in Figure 12.1.

In case of very high plasma energies, *photodisruption* occurs and is marked by both shock-wave generation and/or cavitation, which also leaves surrounding tissue damaged and is therefore not desirable for microsurgery.

Tuning down the energy density by one to two orders of magnitude, we come to the most useful regime for microsurgery on the cellular level: the *plasma-induced ablation* regime, highlighted in the table. This regime provides ablation of the target area by *optical breakdown*, confined to the focal point of the laser, with no damage around that area. Two examples are shown in 12.2.

Optical breakdown of tissue occurs when the applied electric field $\boldsymbol{E}$ exceeds the ionisation energy $\boldsymbol{E_I}$ of the molecules and atoms present. Ionisation then occurs within a few hundred picoseconds, and the radiation is absorbed by the created plasma. The plasma is created by an effect called *inverse Bremsstrahlung*[1], where a free electron is accelerated by an inbound photon, which in turn collides with an atom, ionising it, and resulting in two new free electrons, with less kinetic energy, leading to an avalanche effect. Even if the original material was transparent, the plasma will be opaque to the incident radiation. This effect makes it possible to ablate areas that are otherwise transparent.

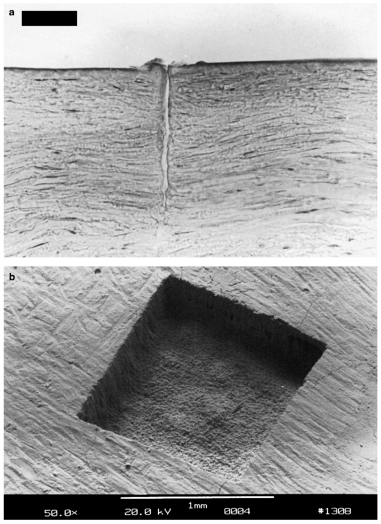

Figure 12.2: a: Cut in a human cornea sample achieved with an picosecond Nd:YAG laser, b: $1x1mm^2$ cut in a human tooth sample with 16000 1 mJ pulses, with cracking only due to EM sample preparation. Reproduced from [Niemz, 2019].

[1] *Inverse Bremsstrahlung* is the opposite of the regular *Bremsstrahlung* effect, where a high-velocity electron gets rapidly decelerated in an atom's electric field, emitting high-energy photons during the process.

12.2 Example Use Cases

Laser ablation has found wide application in cellular biology, here we show a few examples from this wide variety:

- In [Brugués et al., 2012] the authors use femtosecond infrared laser ablation repeatedly to induce synchronous depolymerisation in *Xenopus* metaphase spindles and by that are able to infer information about the length distribution of microtubule segments in the spindle.
- In [Saha et al., 2016], the authors describe the use of a picosecond Nd:YAG laser for disruption of the cellular cortex of *C. elegans* embryos and gastrulating *D. rerio* embryos in order to determine its properties, which are modeled as a 2D film of a viscoelastic active gel.
- In [Li et al., 2014], the authors use laser ablation and optogenetics to disrupt the *C. elegans* AIY interneuron and by that are able to show it is important for locomotion and direction reversal of motion.

- In [Li et al., 2019], the authors use laser ablation to investigate the migration of Trunk Neural Crest cells in chick embryos. In the paper they partially eliminate the lamellipodium to investigate its role in cell-cell contact attraction in conjunction with cell-cell adhesion and find both play counteracting roles.

12.3 Related work

While the use of virtual reality and associated interaction techniques for arbitrary laser ablation has not been demonstrated yet, various authors have made contributions in that direction:

In [Engelbrecht et al., 2007], the authors demonstrate a SPIM-based microsurgery/laser ablation setup, powered by a 355 nm UV laser, which is able to perform multiple cuts with difficult geometries, and high precision. Most importantly, this is to our knowledge the first paper where cuts in all three spatial dimensions were demonstrated, ranging from sub-micron precision for the ablation of microtubules to the cutoff of entire *D. rerio* fins.

On the interaction side, [Peng et al., 2014] demonstrated the *Virtual Finger* system to boost the precision of selection and tracing tasks in 3D environments, such as for neuron tracing, or in microsurgery settings. The authors employ a combination of raycasting together with region growing and shortest path determination for the precision enhancement of their methods.

[Oswald, 2010] demonstrated a versatile laser ablation setup on top of a confocal microscope that is able to perform cuts with rates in the 1 kHz range over an area of $100 \times 100\,\mu m^2$.

In terms of volumetric cuts, the authors of [Brugués et al., 2012] used a femtosecond infrared laser to perform plane-like cuts composed of many, micrometer-spaced lines inside the spindle apparatus of a *Xenopus* nucleus.

12.4 Observations

From our survey of related works and use cases in the previous section, we observe the following:

Current interfaces for laser ablation often do not feature a 3D view of the specimen, but usually utilise a 2D window with different controls for moving around, as shown in Figure 12.3. Such systems provide support for planar cuts, in the form of lines, circles, and rectangles.

Complex, three-dimensional cuts are hard to perform on a regular screen from an interaction perspective — it is hard to translate from an image projected on a 2D screen to a — potentially moving — 3D volumetric specimen, and even with sophisticated techniques as in [Peng et al., 2014], it can become overwhelming for the user. Nevertheless, complex geometries need to be investigated in order to better understand the biophysics of, e.g., the cellular cortex.

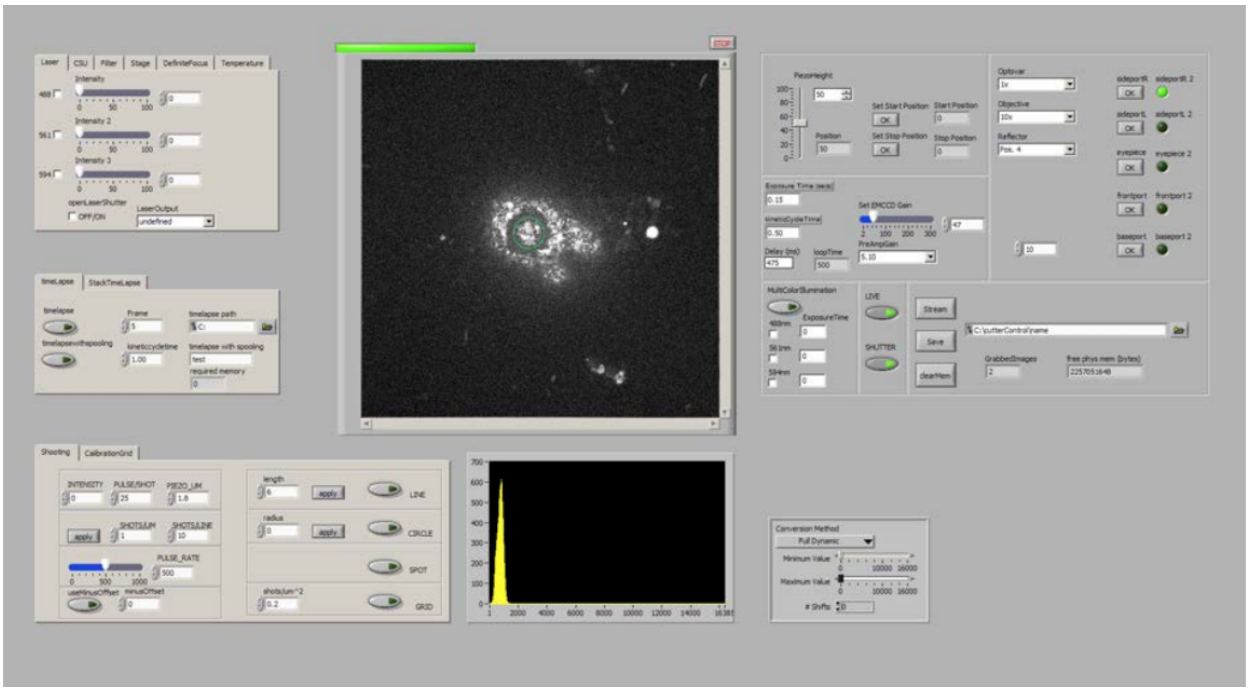

Figure 12.3: A window-based 2D interface for laser ablation. Laser and stage controls for movement are shown in the tabs on the upper left, power controls for the ablation unit on the lower left. The view of the specimen is shown at the center, with the current cut overlaid as a circle. Reproduced from [Oswald, 2010].

Most of the systems surveyed use a confocal microscope as a basis. In many use cases, lightsheet microscopes could be used for their superior speed and gentle imaging as instruments for 3D ablation purposes. Their particular way of mounting samples can beneficial for quickly moving the sample in three directional axis plus one rotational axis, something that is not possible with e.g. confocal microscopes. Even if the sample requires to be mounted on a microscopy slide, variations of the original lightsheet microscope design exist that have similar geometries as confocal microscopes.

12.5 First Prototype

As a first prototype, we developed a browser-based (threejs, https://threejs.org) prototype that makes use of the *LeapMotion* gesture controller. In the prototype, the user can perform tubular cuts in a simulated geometry of a *C. elegans* adult worm using the gesture controller. The process is visualised on the user's computer screen and does not use any VR visualisation techniques. A screenshot of the prototype can be seen in Figure 12.4. The workflow of the prototype is

1. orient the specimen of *C. elegans* in the desired way by using keyboard and mouse,
2. form a circular structure with thumb and index finger, and draw the desired tubular structure into the aligned worm, and finally
3. a cylindrical tube is calculated from the defined circular samples via Centripetal Catmull-Rom spline interpolation [Catmull and Rom, 1974]. Catmull-Rom splines have been chosen here as they always go through their control points, and do not form cusps, which both are desirable properties for surfaces later to be used in laser ablation.

We identified two major issues with this approach:

1. Orientation of the specimen using keyboard and mouse is error-prone and was noted to be not very comfortable and intuitive, especially when combined with subsequent gestural interaction,
2. the gestural interaction was found to be imprecise, as a feeling of 3-dimensionality or immersion did not come up when being restricted to a regular, flat screen without any VR functionality.

We abandoned browser-based prototyping after this first iteration, as loading times were already too long when using the geometry model of 26MiB, and would be even longer if any volumetric data would be used — such data can easily reach many GiB. We want to note here that this initial experience also contributed to the decision to start the development of scenery, such that we can efficiently prototype software that enables interaction with geometry data, and large volumetric data.

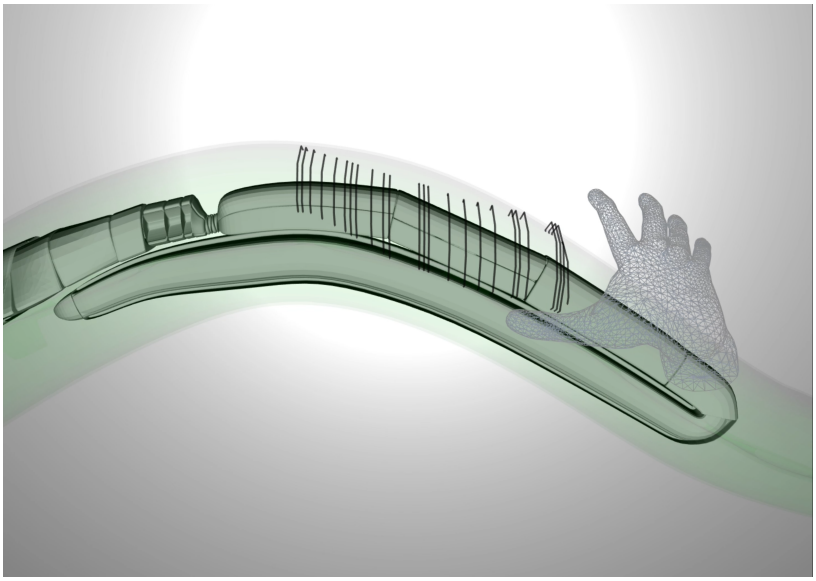

Figure 12.4: Screenshot of the *LeapMotion*-based interaction prototype, where the user has delineated a tubular structure along the *C. elegans*' gonad system. *C. elegans* model courtesy of openworm.org.

12.6 Second Prototype

The software for the second prototype was developed with our visualisation framework *scenery*, described in detail in [scenery — Democratising VR/AR Visualisation for Systems Biology]. We switched away from browser-based prototyping, as the amounts of volumetric data required to be handled in the demo are too large for browser-based software, and because *scenery* is an ideal toolkit for such prototype, due to its support for large volumetric data and VR devices. We choose a VR setup using an HTC Vive HMD with two controllers. The HTC Vive VR package is state-of-the-art at the time of writing, provides high-resolution displays for both eyes and low-latency, hand-held controllers. In addition, the controllers can be augmented with additional devices, tracked by a small puck that can be attached to arbitrary objects, or even body parts for full-body tracking. We did however only use the hand-held controllers for this prototype.

12.6.1 Description of study

In the prototype, the user is shown a pre-recorded, multi-timepoint dataset of a *C. elegans* embryo in three-cell-stage. The embryo had been genetically engineered to express a fluorescent protein in the histones of its DNA, such that the chromosomes (and, to a lesser extent, the associated spindle apparatus orchestrating DNA condensation, duplication, and division) are visible. The time series dataset was played faster than realtime to evaluate quick decision making and the ability to perform cuts under time constraints. A screenshot of the prototype is shown in Figure 12.6.

The users can control movement with the touchpad of the left-hand controller, and also move around physically, as they are being tracked by the VR system in an area of about 2m by 3m. The right-hand controller can then be used to activate a wand-like tool to designate areas for ablation. See 12.5 for a visual representation of the controls.

For simplicity, the prototype was designed such that there is no undo function, but a cut drawn, once finished, would be performed instantly. In real use, this is most probably not a universal solution, but may have benefits in certain situations, where interaction speed has a higher priority than precision.

We conducted a study with 8 experts in laser ablation (average age of 31, 4 female, 4 male, all right-handed, and recruited from different labs of the Max Planck Institute of Molecular Cell Biology and Genetics). The study subjects were informed about contents and goal of the study, and eventual risks and adverse health effects arising from the use of VR glasses. The study subjects were not compensated for taking part in the study.

Before the start of the study, users were asked about their familiarity with smartphone-based VR, computer-based VR, and standalone VR, as well as for their current wellbeing.

After an introduction to the software and familiarisation with the dataset, the users were asked to perform the following tasks:

- perform several cuts in the chromosomes of the uppermost cell
- perform one triangular cut in centrosomes of the uppermost cell, and one in the centrosome of the lower cell
- perform several cuts in the metaphase plates that form in the lower cell after playing half the dataset.

Performing all of these tasks took 5 to 10 minutes per user.

After the study, the users were asked about the following aspects:

- again, for their wellbeing,
- for different aspects of the prototype, the likelihood of adoption of VR-steered laser ablation,
- for physical and mental demands, assessed by the NASA-TLX (*Task Load Index*) [Hart and Staveland, 1988] scoring system, and

Scan this QR code to go to a video demo of the VR ablation prototype. For a list of supplementary videos see https://ulrik.is/writing/a-thesis.

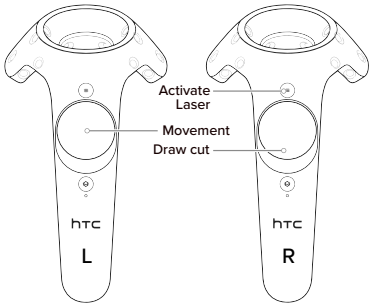

Figure 12.5: Controls for second prototype. Vive controller drawing from VIVEPORT Developer Documentation, developer.viveport.com.

- for symptoms of simulator sickness using the SSQ scoring system [Kennedy et al., 1993] (SSQ takes 16 different symptoms of discomfort — ranging over nausea, oculomotor, and disorientation symptoms — into consideration to calculate a final, weighted score).

For standardised evaluation, we choose NASA-TLX and SSQ, as they perfectly match our application setting, do not interfere with the study process itself, and have been widely used and validated. Newer methods to assess motion sickness in real or virtual environments, such as [Keshavarz and Hecht, 2011] have not been used, as they have been designed to assess motion sickness during the course of the study, which would have caused interference with the tasks the user were asked to perform.

After filling out our questionnaire, the users were asked to participate in an additional, voluntary interview, to ask detailed questions about their experience with the prototype. All of the users agreed to participate in the follow-up interview. The questionnaire used is available in Appendix A.

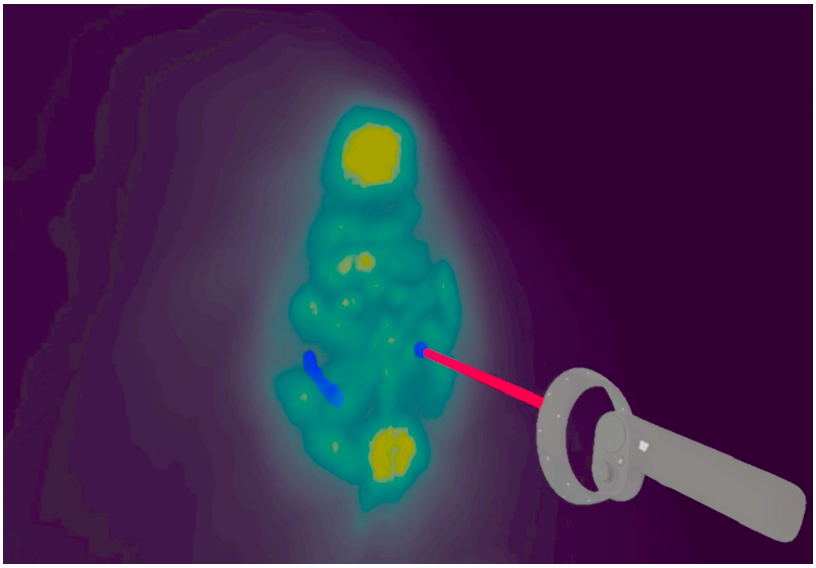

Figure 12.6: Screenshot of the second virtual reality-powered laser ablation prototype. In the prototype, we show the mitotic spindle apparatus in a pre-recorded dataset showing a *C. elegans* embryo undergoing mitosis. The tube-like objects in the center of the image are the condensing chromosomes in the cell nucleus, in the process of being separated by the mitotic spindle. The task of the user is to draw in cuts using VR controllers. See text for a full description. Dataset courtesy of Loïc Royer (MPI-CBG/CZI).

12.6.2 Results — General Questions

Results for the general questions section of the study questionnaire are shown in Figure 12.7. Users were universally satisfied with the quality and usability of the prototype, and were not irritated by having to perform tasks in VR they were previously only used to with 2D interfaces. They generally reported that showing the dataset in human size scales and positions — in the study, the dataset is shown with a height of 1.5 m, hovering 1.5 m above (virtual) ground — was well chosen. They did not have trouble learning the interface, and were in general ready to use the interface within a few minutes. While the users felt that the way of visualising the dataset supported the performed tasks well, a number of users criticised the fidelity of the visualisation and indicated in the follow-up interview they would like e.g. adjustable transfer functions, as the opacity of the dataset sometimes interfered with the tasks. In terms of input modality — the users were given state-of-the-art handheld VR controllers — users

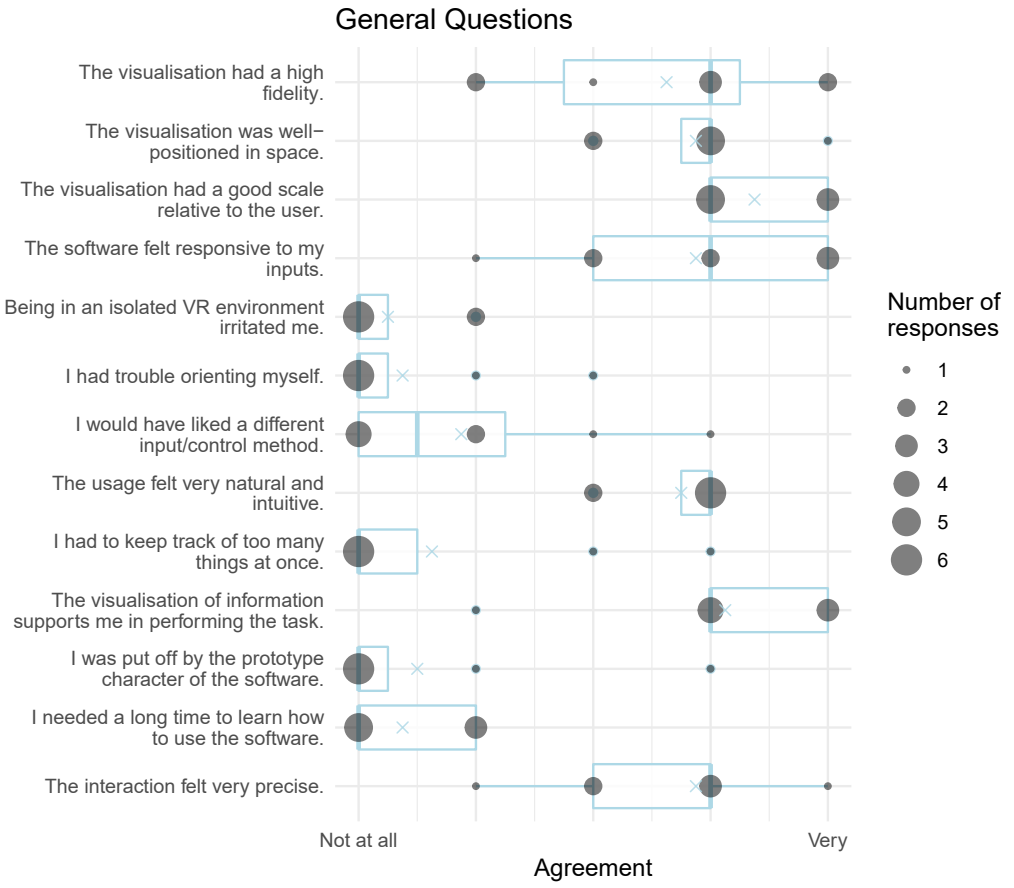

Figure 12.7: Results of the general questions section of the user study.

were mostly satisfied, only a few users would have liked different input modalities more. In the follow-up interviews these users indicated that a pencil-like interface would feel more precise than the HTC Vive controllers used.

In general, the very positive user response shows that most of the design decisions in the prototype have proven correct, so it can be refined further, and then deployed to control an actual physical system.

12.6.3 Results — Wellbeing, Workload, and Simulator Sickness

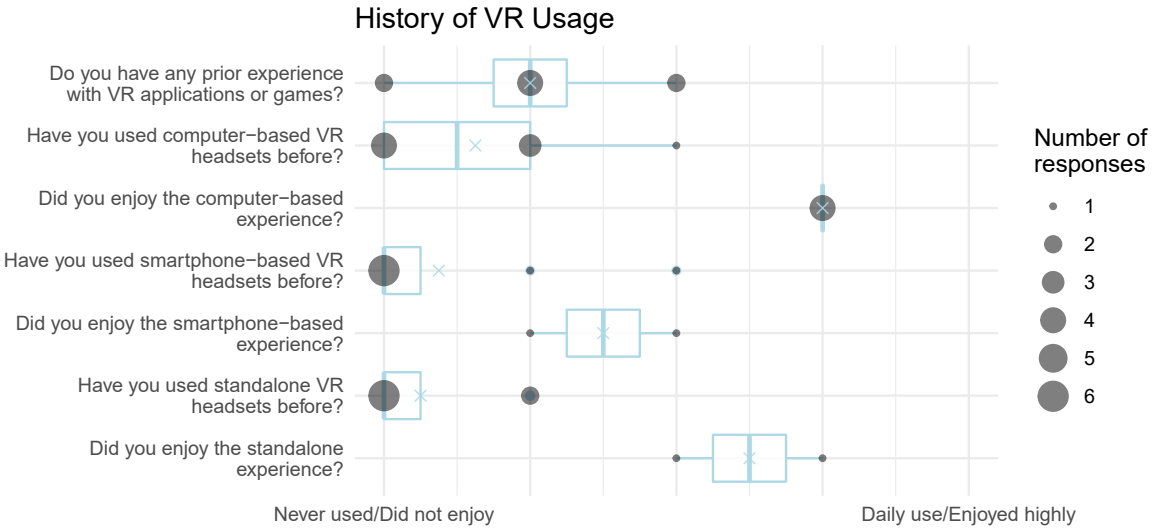

Figure 12.8: History of previous VR usage and satisfaction in our study group.

All users tolerated the usage of the prototype very well. In Figure 12.11 it can also be seen that wellbeing — indicate by the *Concentrated*, *Motivated*, *Headache*, *Tired*, *Dry/Aching Eyes*, and *Nausea* data points taken before and after the study — was not affected by the test in a significant manner. This finding is confirmed by the low SSQ scores we obtained:

The average total SSQ score was 6.2 ± 6.7. Compared to the calibration sample in [Kennedy et al., 1993], this is a very low score, as only the 60th percentile was in that realm, and the mean of the calibration sample was 9.8 ± 15.0, nearly 1.5-times the score in our study.

The users in the test had mostly a low degree of previous exposure to VR systems before (see Figure 12.8). Those who had previous exposure to VR games or applications were mostly happy with it.

We found a correlation between a history of VR usage and low SSQ scores (see Figure 12.11), indicating there might be a training effect. Furthermore, a history of VR usage correlated well with low TLX scores, and a general appreciation of the visualisation, size, and positioning of the dataset in the study.

Workload evaluation results are shown in Figure 12.9. Users generally reported

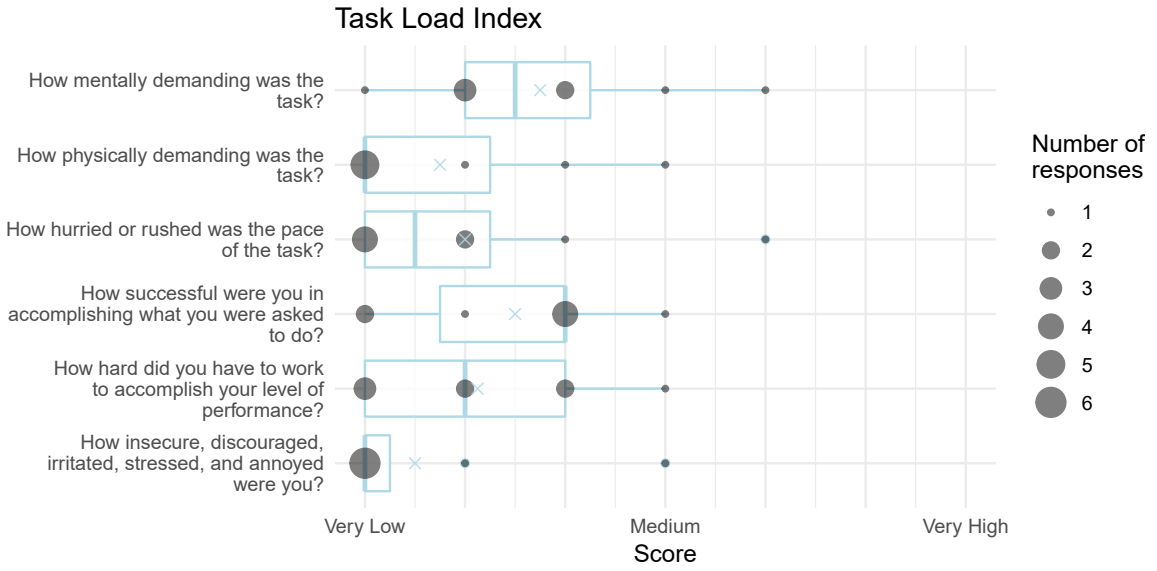

Figure 12.9: Task Load Index (TLX) results in the user study.

very low mental and physical demands in the study, and mostly did not have to work hard to achieve the desired results. A single user felt that the task was a bit hurried or rushed, and two felt that the mental demand was average or above-average. Users in general did not feel insecure or annoyed performing the task using the proposed interface.

Both wellbeing and workload results indicate that using the proposed VR interface is very comfortable for the users and allows them to perform 3-dimensional ablation/selection tasks with ease, and without experiencing motion sickness.

12.6.4 Results — Acceptance and Potential Adoption

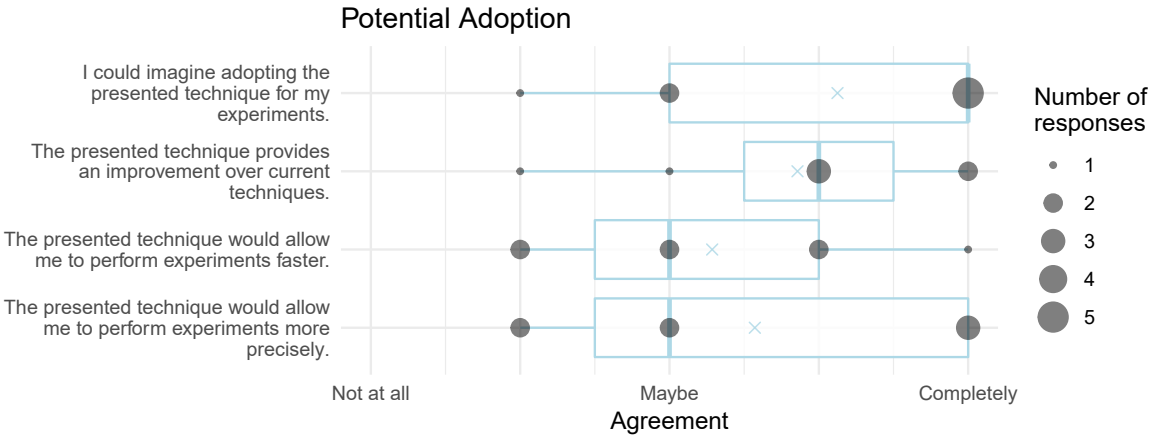

Figure 12.10: Results of the adoption questions section of the study.

The majority of the users stated they could imagine adopting the presented technique for their experiments (see Figure 12.10 for all results), and stated that the

technique provides an improvement over the current way laser ablation experiments are performed.

Users seemed unsure whether the technique presents an improvement in terms of speed or precision: While they tended towards improvement in both cases, in both cases, four users answered only *maybe* or less. In the follow-up interview we found out the reasons for the uncertainty here lies in the different models systems users investigate: users which had to produce a large number of reproducible cuts in their day-to-day experiments tended to be more skeptical about free-form drawing. We are going to address this problem in the next prototype, see Section 12.6.6, *Requested Changes and Additions* for more details.

12.6.5 Results — Further correlations

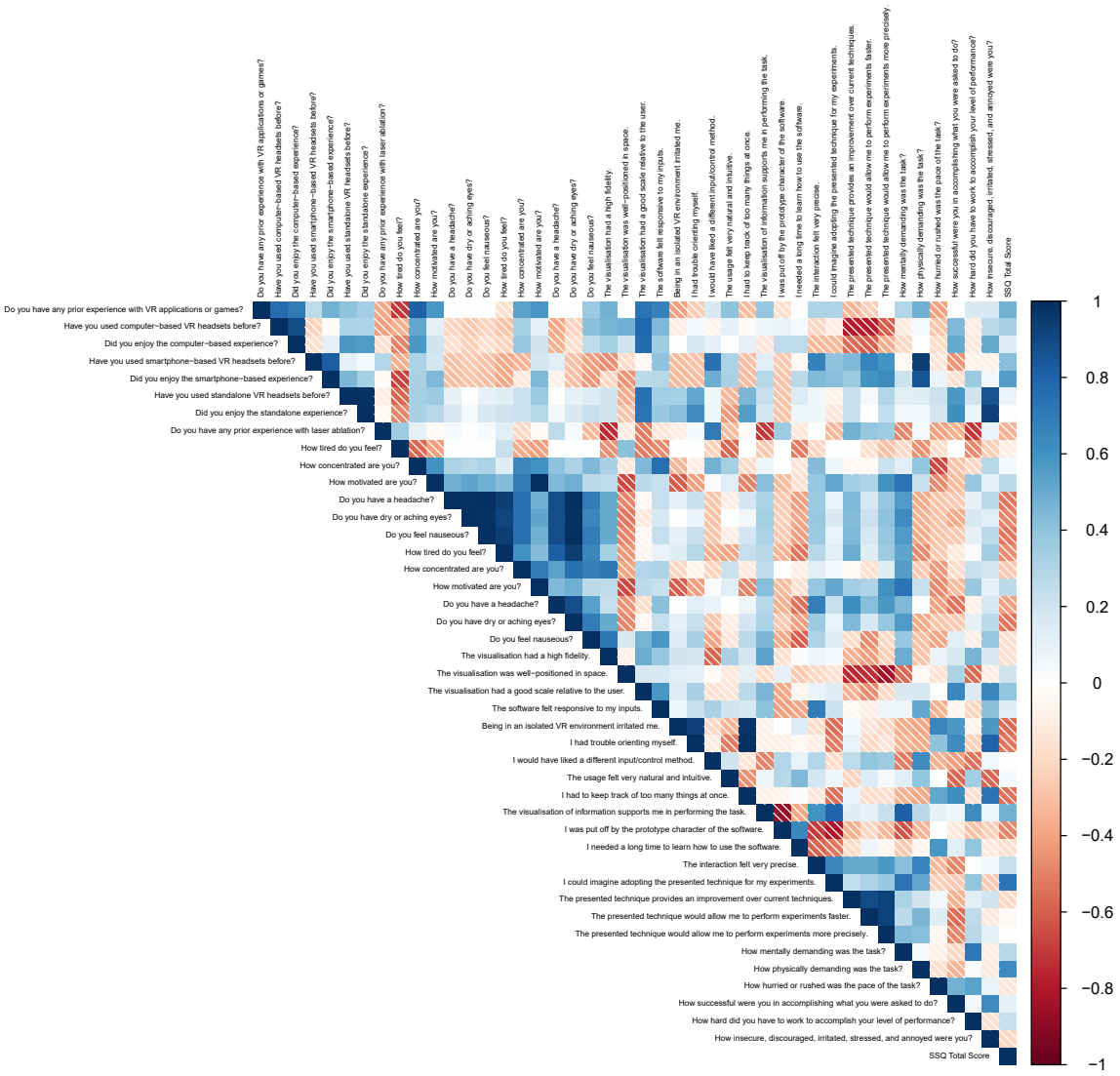

Figure 12.11: Correlations between questions in the questionnaire, only including SSQ summary score. For full correlation plot, please see Appendix B.1.

In the correlation matrix in Figure 12.11 a few more interesting correlations can

be observed. Correlations were computed as Pearson correlation coefficients. The observed correlations are:

- Positive answers to the Adoption questions (see Figure 12.10) correlate well with affirmative answers to in the General section about fidelity and naturalness of the prototype, indicating the the subjects answered these questions truthfully.
- Individual TLX and SSQ scores anticorrelate with wellbeing scores, indicating that the less well a user felt before or after the study, the more demanding the tasks were scored, and the more sick the subject felt afterwards.
- Questions in the General section (see Figure 12.7) did not correlate well with each other, indicating that each of them provides a valuable and independent data point.

12.6.6 Requested Changes and Additions

In the interviews conducted after the study, users were asked to comment further on the presented prototype and suggest improvements and additions. From this feedback, we decided to implement the following features and changes:

- **Confirm and Undo**: In addition to the regular freeform mode with immediate ablation after completing the drawing, another mode, where the target shape is drawn first, and confirmed after additional inspection was requested. In this mode, undo or erase will be possible as well.
- **Brush size**: The ablation laser by default has a specific cut size. By combination of multiple shots, larger cuts can be created. In the interface, this can then be handled in a similar way as brush size adjustments in applications like *Adobe Photoshop*.
- **Template mode**: Many cuts a user has to perform are to be reproducible over a set of different specimen. For that reason, a template mode will be added, where a shape defined at one point can be reused later, optionally after translating, scaling or rotating it.
- **Semi-automatic Guides**: In 2D/3D or presentation applications, such as *Autodesk Maya*, *Adobe Photoshop* or *Apple Keynote*, interactive guides exist to help the user with element alignment. Users often perform ablations relative to one or more specific landmarks, such as centrosomes in the mitotic spindle, or these dots in the Drosophila pupal wing (see Figure 12.12 for an example what such a landmark might be). Semi-automatic guides will be added such that they can indicate to the user which are the optimal points or contours for ablation. The semi-automatic guides will also be scriptable so they can be adjusted for a specific experiment.
- **Toolbelt**: The already existing freeform mode will be combined into a toolbelt, e.g., attached to a VR controller, with the user being able to seamlessly switch between different tools. The toolbelt will also offer the possibility to create custom tools by scripting.

These changes will be implemented in a future version of the software. The proposed changes can be easily integrated in our scenery-based prototype.

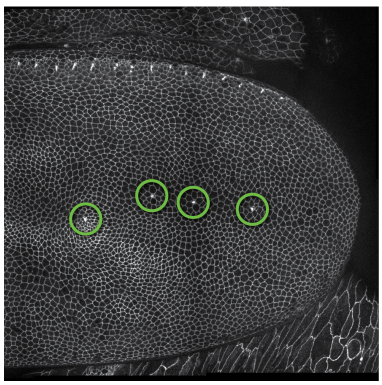

Figure 12.12: Sensory organs in the *Drosophila melanogaster* pupal wing circled in green. These can be used as landmarks for laser ablation. Image courtesy of Romina Piscitello, Eaton Lab, MPI-CBG.

12.7 Proposed Hardware Realisation

In our proposed setup, we are going to use a lightsheet microscope of the SPIM variety as default to overcome the speed and inflexible mounting issues of confocal microscopes in order to provide the user with instant feedback, and to treat the sample more gently, potentially for repeated application of cuts.

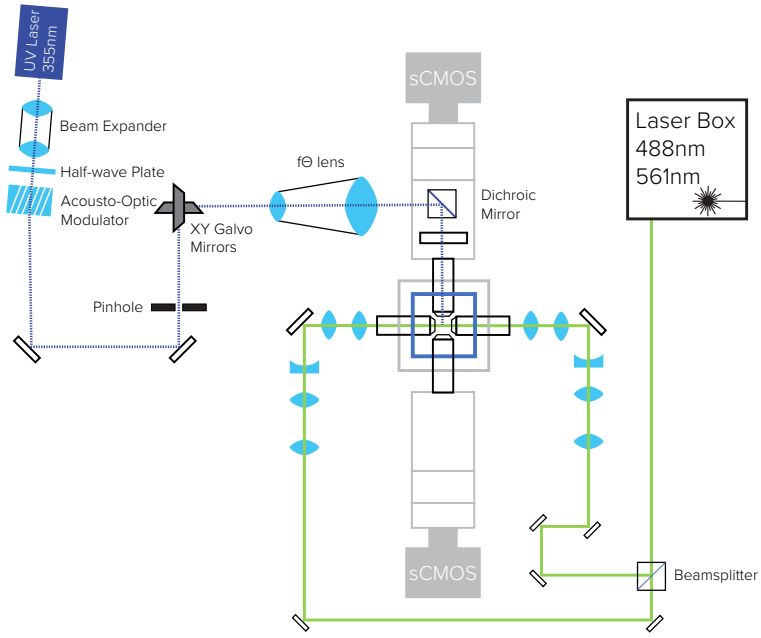

Figure 12.13: Beam paths of our proposed hardware solution, based on a X-SPIM version of the OpenSPIM, with two illumination and two detection arms, and the UV ablation unit coupled into one of the detection arms. See text for details. Figure extended from X-OpenSPIM design by Johannes Girstmair.

The ablation unit design we propose is based on the design of [Oswald, 2010]. In Oswald's original design, the ablation unit was coupled to a spinning disk confocal microscope, and the design is already quite modular. We want to keep the modularity of the unit, such that it can also be used with other microscopes, e.g. spinning disk confocals as in the original, because some samples might need mounting on a glass slide. *C. elegans* adults for example do not enjoy embedding into agarose as it is common in lightsheet microscopy, while *Danio rerio* or *Drosophila* specimen tolerate it excellently.

A sketch of the setup is shown in Figure 12.13. The ablation unit is connected to an extension of the OpenSPIM microscope [Pitrone et al., 2013] for double-sided illumination and double-sided detection developed by Johannes Girstmair at UCL London and MPI-CBG, Dresden dubbed X-SPIM [Girstmair et al., 2016]. The X-SPIM design has the benefit that the sample can be more evenly illuminated from two sides, limiting the need for multi-angle acquisitions where the sample needs to be rotated, as light can only penetrate biological tissue to a limited extent. While more complex than the original OpenSPIM design, even an X-SPIM is less complex than a spinning disk confocal microscope and can be built by an experience microscopist within a day.

The ablation unit itself consists of the following parts:

- A 355nm Nd:YAG picosecond-pulsed UV laser, providing the necessary power output to reach the plasma-induced ablation regime described in Section 12.1, *Introduction to Microsurgery*. The laser provides high-energy pulses of 10 μJ with a rate of up to 1 kHz, with a pulse width of 500 ps, yielding power densities of up to 10 $\mathrm{TW} \cdot \mathrm{cm}^{-3}$.
- An Acousto-optical modulator (AOM) is used to be able to quickly change the laser power to reach the optimal regime for ablation. The AOM works by diffracting the beam by phonon waves in a silicium dioxide crystal. The first diffraction order of the beam then is adjustable between $0 - 80\%$ of the original input power.
- Two galvanometric mirrors for steering in the X and Y axis are used to steer the UV laser beam in an fΘ lens. An fΘ lens translates a beam of incidence angle Θ by $f \cdot \Theta$, where f is the focal length of the lens. It is used in our proposed setup instead of the scanning telescope in the original setup.

The total magnification of the system has to be designed to overfill the entrance aperture of the objective, in our case the upper detection objective. The exact specifications of the objective to use are still under consideration. The setup will then contain an adjustable beam expander such that the ablation unit can be adapted to multiple systems.

For computer control of the microscope, the ClearControl interactive/automatic microscope control software (github.com/clearcontrol/clearcontrol) has been ported by Robert Haase and Johannes Girstmair to support the OpenSPIM hardware components. We have further coupled ClearControl with scenery and sciview to facilitate live visualisation and control.

12.8 Future Work

The next prototype of the software will incorporate the changes proposed in the previous section.

While we have focussed on the task of laser ablation, we believe that the interactions we have proposed are also applicable to other tasks in microscopy that require an surface or volume selection, such as optogenetics and photoconversion [Boyden et al., 2005], where photoactivatable proteins are used to steer cellular functions, or focused light-induced cytoplasmic streaming (FLUCS) [Mittasch et al., 2018], where intracellular flows can be induced by scanning a focused laser over the specimen.

To improve the fidelity of the visualisation during the ablation procedure, we want to to extend our software framework to support better rendering algorithms [Kroes et al., 2012, Igouchkine et al., 2017] to provide better visual assistance and guidance to the user (also see Chapter 10, *Future Development Directions* for more details).

Finally, we aim to provide an open-source/open-hardware solution to perform both laser ablation and optogenetics tasks with the assistance of VR interfaces, based on a customised OpenSPIM microscope.

Chapter 13:

Rendering the Adaptive Particle Representation

The work presented in this chapter has been done in collaboration with Bevan Cheeseman, Sbalzarini Lab, MPI-CBG, and is partially published in:

Cheeseman, B.L., **Günther, U.**, Susik, M., Gonciarz, K., and Sbalzarini, I.F.: Adaptive Particle Representation of Fluorescence Microscopy Images. *Nature Communications*, 2018. bioRxiv preprint 263061.

13.1 Introduction

The Adaptive Particle Representation (APR) [Cheeseman et al., 2018] is a representation of image data that does not rely on regular sampling as found in pixel images, but instead uses computational particles to represent point intensities and further properties in space-filling data structure similar to an octree. Especially in the context of fluorescence microscopy, where images are mostly sparse, this alternative representation allows for highly efficient data storage and processing, resulting in space savings of a factor of 10 to 100 compared to the original image size.

13.2 Theory

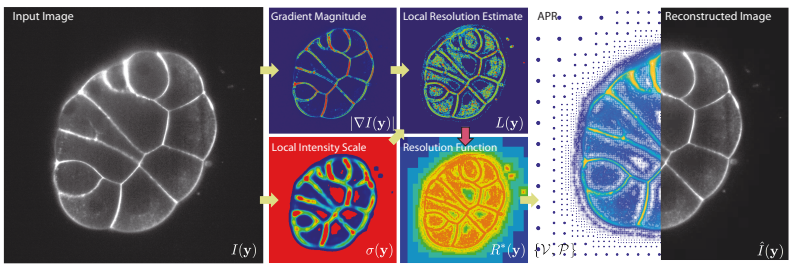

Figure 13.1: High-level overview of the APR construction pipeline: *1.* Input image *2.* Determination of the gradient magnitude and local intensity scale, allowing to adjust for local intensity variations across the image *3.* Estimation of the Local Resolution *4.* Construction of the Resolution Function from the Optimal Valid Particle Cell set *5.* The final APR as combination of the Optimal Valid Particle Cell set $\mathcal{U}$ and the Particle Set $\mathcal{P}$. Image reproduced from [Cheeseman et al., 2018].

As bottlenecks in fluorescence microscopy not only exist with storage, but also with processing of the generated imagery, underlying the APR are four representation criteria:

- *RC1* — The APR must guarantee a user-definable representation of noise-free images and must not degrade the signal-to-noise-ratio of noisy images.
- *RC2* — Memory cost and computational cost of the APR must be proportional to the information content of the image, and not its pixel size.
- *RC3* — It must be possible to rapidly convert from image to APR, and back.
- *RC4* — The APR must reduce both memory cost and computational cost, and allow existing algorithms to consume it with minimal changes, and without resorting back to a pixel representation during processing.

An overview of the APR construction pipeline is given in Figure 13.1. Before continuing, let us introduce and explain a few terms that we are going to need:

- *Particles* — particles in the APR are a generalisation of pixels that can carry properties, such as, but not limited to, intensity or size. Particles, in contrast to pixels, to not need to reside on a Cartesian grid.
- *Implied Resolution Function* — as the APR resamples an image using particles the implied resolution function governs the required or desired resolution everywhere in the image.
- *Local Intensity Scale* — or $\sigma(y)$ for a point y in the image is a estimation of the dynamic range locally present around y, introduced to not over-value bright parts of an image, and under-value dim parts.
- *Pulling Scheme* — the pulling scheme is an algorithm that efficiently solves the particle positioning with regard to the implied resolution function, turning a problem that would otherwise scale as $O(N^2)$ (N being the number of particles) to $O(N)$.

13.2.1 Reconstruction condition

At each point y of an image $I(y)$, and an image $\hat{I}(y)$ reconstructed from an APR, *RC1* can be reformulated as finding the resolution function $R(y)$ that maximises

$$|I(y) - \hat{I}(y)| \leq E\sigma(y) \tag{13.1}$$

where E is the user-specified maximum error, and $\sigma(y)$ is the local intensity scale. As $\hat{I}(y)$ is reconstructed by interpolating over all particles in the APR, finding an optimal $R(y)$ is an $O(N^2)$ operation in the number of particles.

By introducing two restrictions on the formulation of $R(y)$, we can solve this problem though:

First, we restrict $R(y)$ to satisfy

$$R(y) \leq L(y^*), \forall y : |y - y^*| \leq R(y), \tag{13.2}$$

with $L(y) = E\sigma(y)/|\nabla I|$, with ∇I being the gradient of the image. This inequality is called the *Resolution Bound* and L the *Local Resolution Estimate*. If the underlying image is assumed to be differentiable everywhere, and $\sigma(y)$ assumed to be sufficiently smooth, the Resolution Bound is stricter than the Reconstruction Condition, and an $R(y)$ subject to it will yield an equal or better representation accuracy.

Second, the Resolution Function $R(y)$ is further restricted to consist only of square blocks, whose sizes are powers of two. Then, the optimal Resolution Function can be found in $O(N)$.

13.2.2 Particle Cells

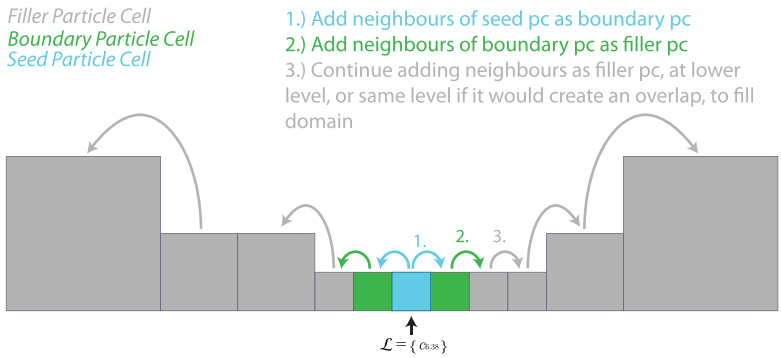

Figure 13.2: Formation of the Optimal Valid Particle Set in the case that the local particle cell set $\mathcal{L}$ only has one cell. Image reproduced from [Cheeseman et al., 2018].

The blocks constituting the Resolution Function must be powers of $1/2$ the image edge length in pixels $|\Omega|$[1]. The piecewise constant Resolution Function which is then defined by the upper edges of these blocks is called the *Implied Resolution Function* $R^*(y)$, and it's blocks are called the *Particle Cells*, which all have a side length of $|\Omega|/2^l$. l is called the *Particle Cell Level* and ranges from $l_{\text{min}} = 1$, where the corresponding block has half the size of the original image, to l_{max}, corresponding to blocks of pixel size.

[1] If an image edge length is not a power of two, $|\Omega|$ is rounded up, and the image not padded.

With the two restrictions introduced, the determination of the optimal Resolution Function can be reduced to finding the smallest set $\mathcal{U}$ of particle cells defining a Resolution Function $R^*(y)$ that satisfies the Resolution Bound 13.2. This smallest set is called the *Optimal Valid Particle Cell set* (OVPC).

For now finding this set, we reformulate the Resolution Bound 13.2 in terms of Particle Cells:

- particle cells become arranged in a tree structure, with an individual particle cell labeled $c_{i,l}$ by level l and location i in the tree. The tree itself is a binary tree in 1D, a quadtree in 2D, and an octree in 3D.
- within this tree structure, the descendents of a particle cell can be naturally defined as all the child particle cells in the tree up to l_{max}.
- $L(y)$ can then be represented as a set of particle cells $\mathcal{L}$ generated by iterating over all pixels y^*, and adding the particle cell with $l = \lceil \log_2 \frac{|\Omega|}{L(y^*)} \rceil$ and $i = \lfloor \frac{2^l y^*}{|\Omega|} \rfloor$, if it is not there already. $\mathcal{L}$ is called the *Local Particle Cell set* (LPC).

For the formation of the OVPC, particles are given an additional *type* property, which is `seed` if the cell is in both $\mathcal{U}$ and $\mathcal{L}$; `neighbor` in case their neighbor is of type `seed`; and `filler` in all other cases. See Figure 13.2 for a schematic how this set if formed in the case of $\mathcal{L}$ only containing one cell.

Then, the Resolution Bound can be reformulated as:

A set of Particle Cells $\mathcal{V}$ will define an Implied Resolution Function $R^*(y)$ satisfying the Resolution Bound 13.2 for $L(y)$, iff $\forall p \in \mathcal{V}$ none of its descendents, or neighbor's descendents are in the LPC set $\mathcal{L}$.

13.2.3 Pulling Scheme

With this definition, we can go on to describe the Pulling Scheme for finding the OVPC set in $O(N)$ time. The name arises from the behaviour of a cell in $\mathcal{L}$ that pulls the resolution function down, leading to smaller particle cells across the image.

The algorithm for the pulling scheme is summarised in Algorithm 2.

Data: Local Particle Cell set $\mathcal{L}$
Result: Optimal Valid Particle Cell set $\mathcal{V}(\mathcal{L})$

Function *pulling_scheme($\mathcal{L}$)*
  Represent all possible Particle Cells $\mathcal{C}$ from l_{max} to l_{min} in a multi-resolution pyramid and set all Particle Cells type to EMPTY;
  forall *Particle Cells $c \in \mathcal{C}$ where $c \in \mathcal{L}$* **do**
    c.type = SEED
  end
  for $l_c = l_{max} : l_{min}$ **do**
    `/* Fill neighbors (Step 1) */`
    forall *neighbors n of $c \in \mathcal{C}(l_c)$ where c.type is (SEED or PROPAGATE)* **do**
      if *n.type is EMPTY* **then**
        n.type = BOUNDARY
    end
    `/* Set Parents (Step 2) */`
    forall *parents p of $c \in \mathcal{C}(l_c)$ where c.type is (SEED, PROPAGATE, or ASCENDANT)* **do**
      p.type = ASCENDANT
    end
    if $l_c > l_{min}$ **then**
      `/* Set Ascendant Neighbors (Step 3) */`
      forall *neighbors n of $c \in \mathcal{C}(l_c - 1)$ where c.type is ASCENDANT* **do**
        if *n.type is EMPTY* **then**
          n.type = ASCENDANT_NEIGHBOR
        if *n.type is SEED* **then**
          n.type = PROPAGATE
      end
      `/* Set Fillers (Step 4) */`
      forall *children d of $c \in \mathcal{C}(l_c - 1)$ where c.type is (ASCENDANT_NEIGH or PROPAGATE)* **do**
        if *(d.type is EMPTY* **then**
          d.type = FILLER
      end
  end
  return all type SEED, BOUNDARY and FILLER Particle Cells in $\mathcal{C}$ as $\mathcal{V}$;

Algorithm 2: The Pulling Scheme algorithm. The Pulling Scheme efficiently computes the OVPC set $\mathcal{V}$ from the Local Particle Cell set $\mathcal{L}$ using a temporary pyramid mesh data structure. $\mathcal{C}(l)$ denotes all Particle Cells on level l.

The Pulling Scheme has the following properties:

- *Predictability and self-similar structure* — neighbouring particle cells never differ by more than one level from each other, and are arranged in a fixed pattern around the smallest particle cells in the set. This structure is independent of the level itself and results in self-similarity between the levels. From this property, the OVPC set $\mathcal{V}$ can easily be constructed from any LPC set $\mathcal{L}$ with a single particle cell $c_{i,l}$.
- *Separability* — The OVPC set can be found by considering each particle cell in $\mathcal{L}$ on its own, and afterwards combining them into one set covering the whole image, using a minimum operation on the particle cells. See Figure 13.3 for a visualisation.
- *Redundancy* — When constructing $\mathcal{V}$, all particle cells in $\mathcal{L}$ that have descendants can be ignored, as descendants imply either the same or a tighter constraint on the Resolution Function.

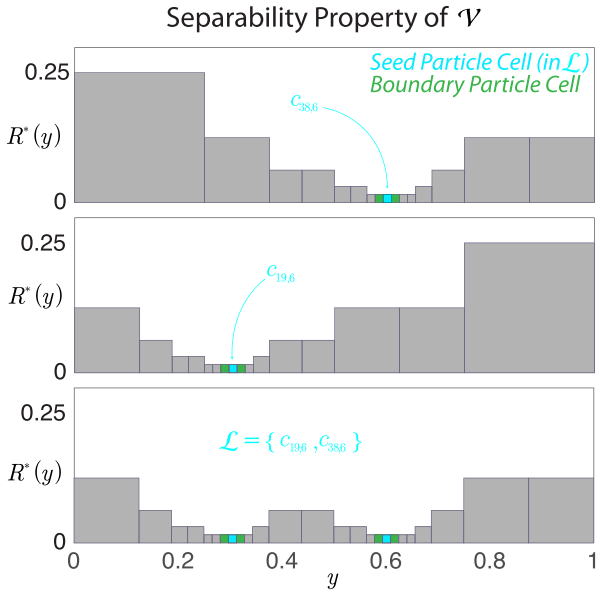

Figure 13.3: Separability property of the Pulling Scheme: In the first two parts, the construction of $R^*(y)$ is shown for two separate particle cells, $c_{19,6}$ and $c_{38,6}$. In the third part of the figure, their combination into the Local Particle set $\mathcal{L}$ is shown. Image reproduced from [Cheeseman et al., 2018].

13.2.4 Creating the APR from the Optimal Valid Particle Cell set

After determining $\mathcal{V}$ via the pulling scheme, the particles $\mathcal{P}$ have to be placed. The Resolution Bound implies that within radius $R^*(y)$ of a pixel at y, at least one particle has to be placed. This means that for each $c_{i,l} \in \mathcal{V}$, a particle p is added to $\mathcal{P}$ with location

$$y_p = \frac{|\Omega|}{2^l}(i + 0.5). \tag{13.3}$$

As particle positions are already governed by the particle cell, they do not need to be stored explicitly. The only data then stored explicitly are particle properties, such as interpolated intensities I_p.

Finally, the APR is formed from both the Optimal Valid Particle Cell set $\mathcal{V}$, and the Particle Set $\mathcal{P}$.

13.3 Related Work

Adaptive sampling and multiresolution approaches have quite a history in image processing: Ranging from pyramid image representations [Adelson et al., 1984], over super-pixels [Achanta et al., 2012, Amat et al., 2012], wavelet decompositions, level-set methods [Monasse and Guichard, 2000], dictionary-based sparse representations [Davis et al., 1997], to adaptive mesh representations [Demaret and Iske, 2002, Wang et al., 1996, Yang et al., 2003], and dimensionality reduction [Schmid et al., 2013, Heemskerk and Streichan, 2015]. None of these methods however are able to guarantee all the Reconstruction Criteria we have outlined earlier.

If we venture outside of just image processing and turn to (realtime) rendering, there are two additional techniques that bear a similarity to the APR:

- *Sparse Voxel Octrees* (SVOs) [Laine and Karras, 2010, Crassin, 2011] work by voxelising a given geometry, with the actual voxels being stored in an octree data structure as final leaf nodes. SVOs are great for storing very large mesh data, but cannot efficiently represent volumetric data as we try to achieve.
- *VDB* [Museth, 2013] uses B+trees [Bayer and McCreight, 1972] to hierarchically represent volumetric data. From the spatial organisation, VDB is closest to our approach, although the leaf nodes of their tree do still contain voxels instead of particles. Figure 13.4 shows a 2D representation of a VDB dataset.

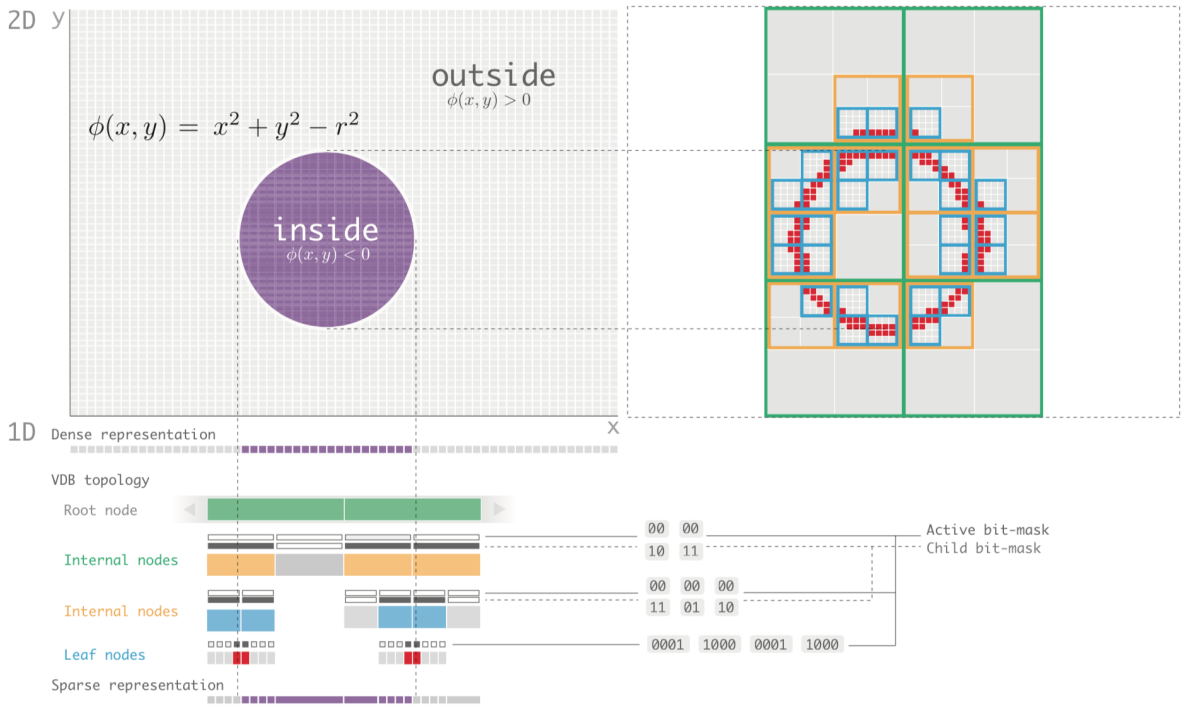

Figure 13.4: Representation of a narrow-band level set stored in the VDB data structure. The lower left part shows the tree structure of a 1D VDB representation of the circle above, with the sparse representation displayed at the bottom left. On the right, the 2D structure of the circle represented as VDB is shown. Branching factors here are chosen for visualisation purposes, and are chosen larger in practise (Reproduced from [Museth, 2013]).

13.4 Integration into scenery

Our APR software library, *libapr* is written in C++ and available at github.com/cheesema/libapr. For interfacing with *scenery*, and the JavaVM ecosystem in general, we have developed a SWIG [Beazley, 1996] (swig.org) wrapper that exposes nearly all of the *libapr* functionality to Java. The wrapper functionality is part of the main repository of *libapr*.

SWIG works by creating an *interface definition file* that specifies all header files that need to be wrapped. In our library this file can be found in the root directory as `libapr.i`, and is quite short. The interface definition also includes additional code that is needed to make the wrapping work, e.g., for renaming functions in the case that naming rules clash between wrapper and wrappee, or for specialising templated code. It also includes custom allocator/deallocator code for the `APR` class and `ExtraParticleData`[2] class, as memory management between garbage-collected languages and non-garbage-collected ones is not straightforward. In our case, both those classes will retain references to a loaded APR, such that it does not get garbage collected by the VM.

[2] The `ExtraParticleData` class contains functionality for attaching additional properties to particles, such as intensities.

13.4.1 Limitations

SWIG does not have very good support for templated code, and the APR library is heavily templated. Therefore, the wrapped library contains only support for 16bit APRs, although this is not too limiting, as that is the most common case, at least in our main use case of fluorescence microscopy.

13.4.2 Future Directions

We are exploring alternatives to SWIG, such as JavaCPP (github.com/bytedeco/javacpp), which has better support for state-of-the-art C++ features, and also includes an automatic wrapper generator, as well as the possibility for manual adjustments, which SWIG provides with the interface definition file.

A prototype of this effort has been developed by Krzysztof Gonciarz and can be found at github.com/krzysg/LibAPR-java-wrapper.

13.5 Goals

For visualising the Adaptive Particle Representation, we set the following goals:

- the software has to handle the APR datasets in realtime
- the software needs to be capable of integration with the existing Fiji/ImageJ ecosystem, i.e. it needs to be usable from Java
- the software needs to support virtual reality visualisation
- the software needs to make use of the attributes provided by the APR particles, such as position, normals, etc., such that they can be used for visualisation or selection purposes.

13.6 Initial prototype

The first prototype developed for APR visualisation, named *dive*, came to be before scenery development even had started.

It was able to primitively visualise APR datasets as point clouds, with no postprocessing applied (see Figure 13.5), but included limited support for the Oculus Rift DK2 HMD. It was also written from scratch using OpenGL 3.3 and SDL, where the need for a much quicker prototyping solution became apparent, as development efforts using this approach were not sustainable going forward.

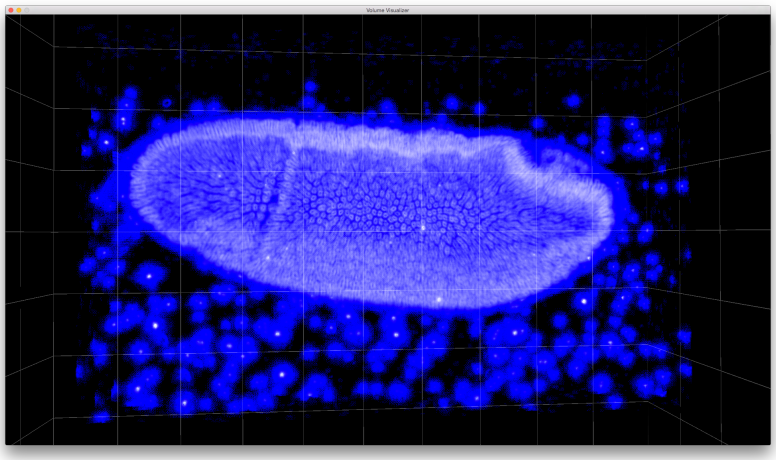

Figure 13.5: Visualisation of a *Drosophila* dataset using *dive* as a point cloud. The original dataset size is 960 MiB, while the APR only consumes 70 MiB. Dataset courtesy of Tomancak Lab, MPI-CBG Dresden.

13.6.1 User feedback

A poster and demo of *dive* was shown at the 2014 BioImageInformatics conference in Leuven, Belgium. In addition, it was tested by 2 more lab members internally. The main points collected were:

- flat colouring leads to a false impression of the dataset
- dataset loading times were considered good, especially as the original dataset is about 960MiB, and the APR dataset only 70MiB.
- filtering based on particle properties was desired to be possible.

13.7 Second prototype

The second prototype built was based on *scenery*, and included a port of the code used in *dive* for importing the APR files. This port resulted in the creation of the Java wrappers discussed in Section 13.4, *Integration into* scenery.

This new implementation now had support for particle properties, as well as HDR and Ambient Occlusion as postprocessing options, as provided by default by scenery. A visualisation with Ambient Occlusion (AO) on/off is shown in Figure 13.6. The second prototype did not include support for VR headsets, as this was still work in progress in scenery back then.

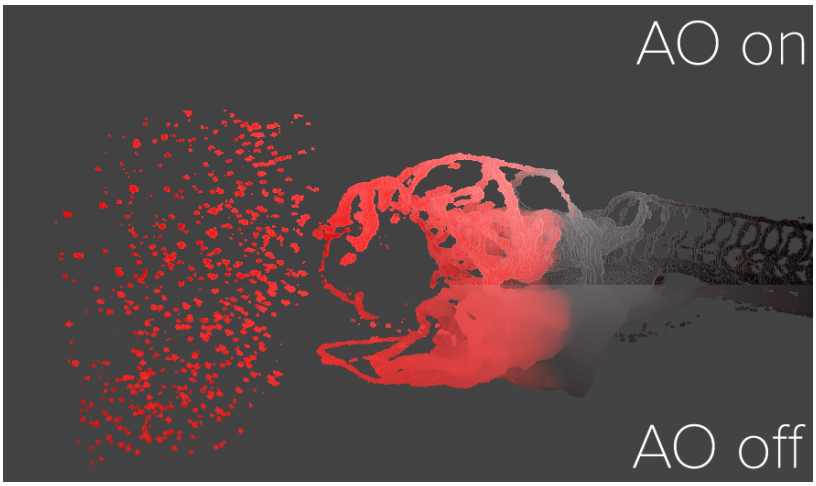

Figure 13.6: Visualisation of a *Danio rerio* vasculature dataset using scenery. Top: Ambient occlusion on, revealing the details of the vasculature. Bottom: Ambient occlusion off. Dataset courtesy of Stephan Daetwyler, Huisken Lab, MPI-CBG Dresden & Morgridge Institute for Research, Madison, USA.

13.7.1 User feedback

Compared to the first prototype, user feedback now was better:

- Ambient Occlusion was well liked, as it gives the dataset a more plastic, more detailed appearance and highlights small details.
- Custom colormaps make the visualisation of APR datasets more flexible.
- Filtering can help to create segmentation-like visualisations easily.

The following additions were requested:

- 3D model-like appearance is unusual for microscopy datasets, biologists are more used to maximum intensity projections or alpha blending-based renderings.
- While filtering based on particle properties was now possible upon loading the dataset, it would be better if filtering could be controlled interactively.

13.8 Particle-based rendering in scenery

For interactive rendering of the APR, we have integrated the Java wrapper with *scenery*[3]. In scenery, we render the APR as point-based graphics, and subject it to the same postprocessing steps as all other renderings (such as screen-space ambient occlusion, and HDR exposure correction).

[3] See github.com/skalarproduktraum/aprrenderer for demo code.

For point-based rendering, positions and intensities of particles on all levels of the APR are reconstructed according to Eq. 13.3, and stored in a `Mesh`, with the mapping shown in Table 13.1.

Particle normals can be stored with the regular APR data as additional property, but they might also be computed on-the-fly. The vertex data is then rendered with a custom shader that provides multiple options for colouring the particles, such as colouring by level, by distance to observer, or intensity. The shader also enables thresholding of the particles, resulting in a very simple way to render visualisations of an APR similar to isosurfaces and segmentations by just selecting particles with a given intensity. An example segmentation of *D. rerio* head vasculature using graph

cuts is shown in Figure 13.7, and an example direct particle rendering of a *Drosophila melanogaster* embryo is shown in Figure 13.8.

Table 13.1: APR-to-scenery mapping for particle properties.

scenery Node property	APR contents
`Mesh.vertices`	Particle position (x, y, z)
`Mesh.normals`	Particle intensity, particle cell level, Particle normal x (optional)
`Mesh.texcoords`	Particle normal y, Particle normal z

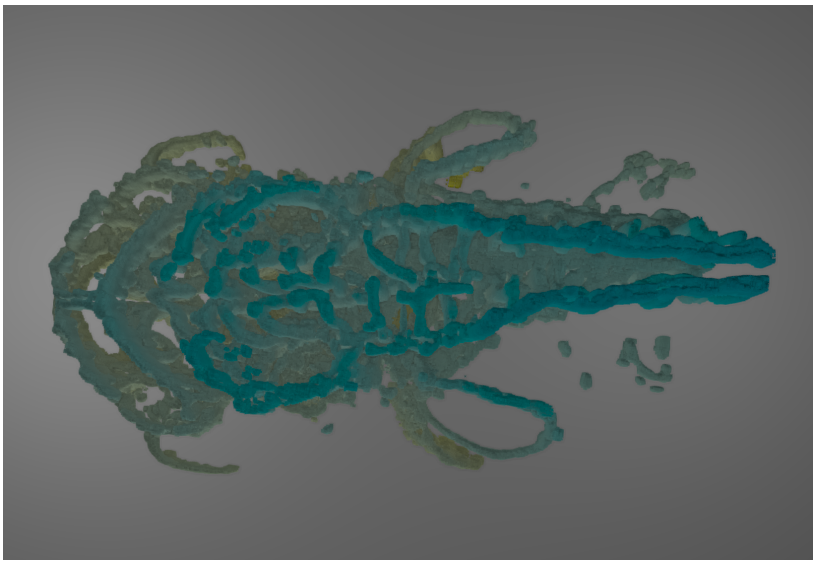

Figure 13.7: Image of an APR-based segmentation of *Danio rerio* head vasculature visualised as point-based graphics. Particles are coloured by distance to the camera. Dataset courtesy of Stephan Daetwyler, Huisken Lab, MPI-CBG Dresden & Morgridge Institute for Research, Madison, USA

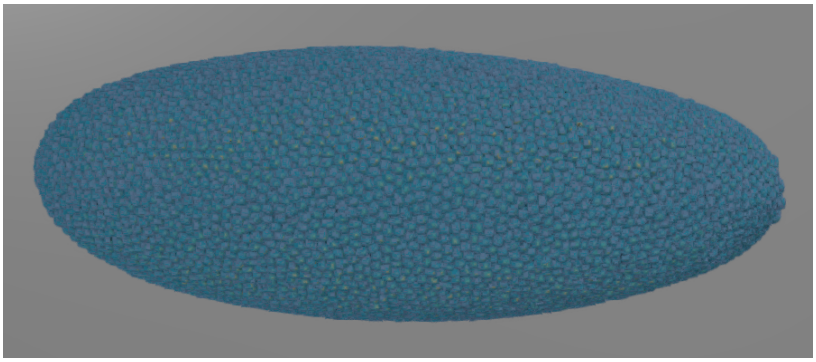

Figure 13.8: Image of a APR-based direct particle rendering of a *Drosophila melanogaster* embryo after cellularisation. Particles are here colored by level, with blue signifying the highest-resolution level, and red the lowest-resolution level. Dataset courtesy of Loïc Royer, MPI-CBG Dresden & Chan-Zuckerberg Biohub, San Francisco, USA, obtained using a custom-built automatic lightsheet microscope.

13.9 Particle-based maximum intensity projection

Maximum intensity projections of datasets are among the most common visualisations used in fluorescence microscopy, therefore they need to be supported on the APR as well.

The algorithm for achieving this is relatively simple:

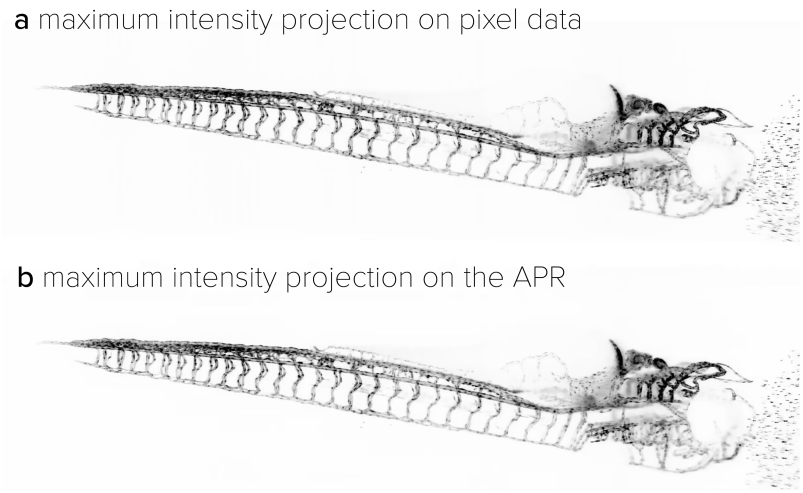

Figure 13.9: Comparison of maximum intensity projections of a *Danio rerio* (zebrafish) vasculature dataset with **a** the maximum intensity projection based on the original pixel data and **b** the maximum intensity projection based on the APR. Visually, there is no perceivable difference, only when the contrast is exaggerated, blocking artifacts from the lower particle cells become visible. Dataset courtesy of Stephan Daetwyler, Huisken Lab, MPI-CBG Dresden & Morgridge Institute for Research, Madison, USA, obtained using a custom-built lightsheet microscope.

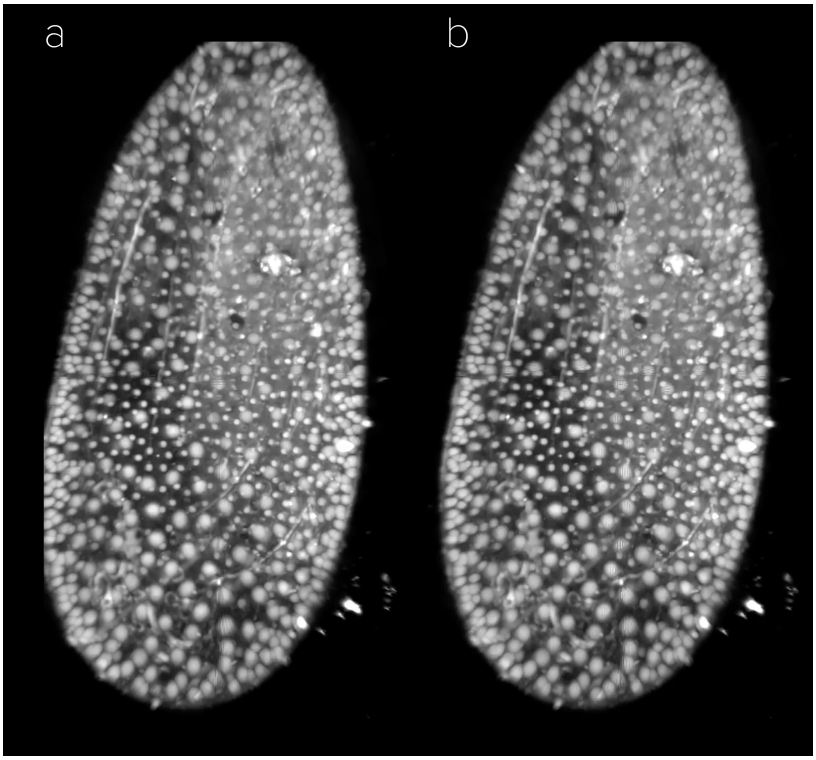

Figure 13.10: Comparison of maximum intensity projections of a *Tribolium castaneum* (red flour beetle) dataset with **a** the maximum intensity projection based on the original pixel data and **b** the maximum intensity projection based on the APR. Visually, there is no perceivable difference, only when the contrast is exaggerated, blocking artifacts from the larger, lower resolution particle cells become visible. Dataset courtesy of Akanksha Jain, Tomancak Lab, MPI-CBG Dresden, obtained using a Zeiss Lightsheet Z1 microscope.

1. create a multi-resolution representation of the original image dimensions down to the lowest resolution (largest) particle cell level, which each level having a factor 2 lower resolution than the one before,
2. iterate over all particles in the APR dataset, and adding the interpolated pixel intensity to the determined pixel position,
3. the resulting per-level images are blended together to yield the final maximum intensity projection. More formally, this algorithm is stated in Algorithm 3.

Two example renderings resulting from this algorithm are shown in Figures 13.9 and 13.10, where they are compared with a maximum intensity projections from the same pixel-based dataset. Visually, there is no difference, although blocking artifacts on the largest, lowest resolution particle cell levels will appear when the contrast is exaggerated.

Data: APR consisting of OVPC $\mathcal{U}$ and particle set $\mathcal{P}$
Result: Maximum projection of the APR, M

Function *maxProjectAPR($\mathcal{U}$, $\mathcal{P}$)*
  for $l_c = l_{max} : l_{min}$ **do**
    `image` ← initialize as empty with dimensions Ω_{l_c} **for** $c_{i,l_c} \in \mathcal{U}$ **do**
      $y_p \leftarrow \frac{|\Omega|}{2^{l_c}}(i + 0.5)$ `/* InterpolateIntensity can either directly use the particle's intensity, or interpolate it */`
      `image` (y_p) ← `InterpolateIntensity(`c_{i,l_c}, $\mathcal{P}$`)`
    end
    `/* Blend levels together, e.g. by max operation */`
    M ←`blend(`M, `image)`
  end

Algorithm 3: Maximum Intensity Projection on the APR.

13.10 Particle-based volume rendering of the APR on the GPU

Unfortunately, the algorithm presented in the previous section only works well on the CPU, as it requires a lot of random accesses to change pixel values, and therefore does not map well to the massively parallel architecture of GPUs. By just gathering particles on a per-level basis, it does not use of the space decomposition inherent to the APR. In this section, we present an alternative algorithm that solves these problems and makes the APR suitable as a basis for interactive volume rendering of large datasets.

In addition, the methodology we proposed in Section 13.8, *Particle-based rendering in scenery*, while simple, does not deal well APRs containing more then 1 or 2 million particles, especially not when a lot of small particles occupy very little screen space, need to be depth-sorted, and blended together. In such cases, the performance can degrade very quickly. Furthermore, the typical user of the APR might not be used to particle-based renderings, but rather to volume renderings of microscopy data.

Our approach bears similarity to the algorithm proposed in [Knoll et al., 2019],

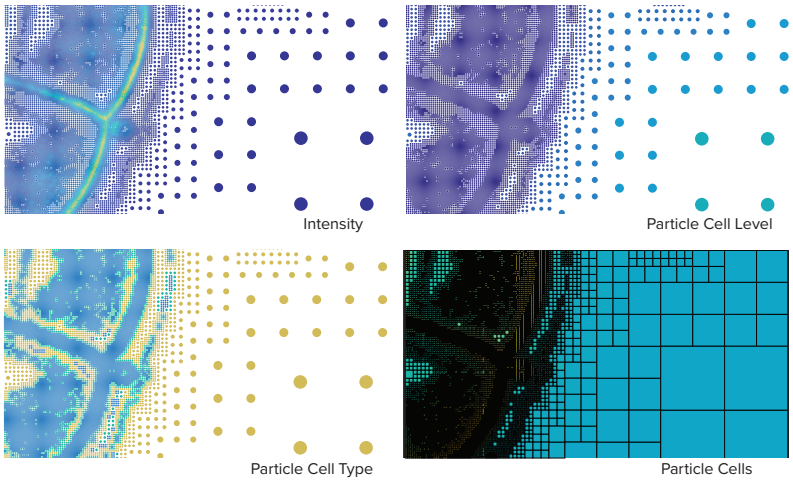

Figure 13.11: A slice of an APR rendered as particle intensities, particle cell level (larger circle equals lower level), cell type, and cell decomposition. Image reproduced from [Cheeseman et al., 2018].

Data: APR consisting of OVPC $\mathcal{O}$ and particle set $\mathcal{P}$
Data: Ray with origin $\vec{o}$ and normalised direction $\vec{d}$
Result: Intensity per pixel, $\hat{I}$, computed along the ray

Function *volumerender_apr_for_ray*($\mathcal{O}$, $\mathcal{P}$, $\vec{o}$, $\vec{d}$)
  while $c \neq$ null **do**
    /* Query next cell from APR structure */
    $c \leftarrow$ NextCell($\mathcal{O}$, $\mathcal{P}$, $\vec{o}$, $\vec{d}$);
    $\vec{o}' \leftarrow$ ExitPointForCell(c, $\vec{o}$, $\vec{d}$);
    forall *Hit particles* $i \in c$ **do**
      $\vec{h} \leftarrow \vec{o} + |\vec{p}_i - \vec{o}| \cdot \vec{d}$;
      $I_i \leftarrow$ ParticleIntensity($\vec{h}$);
      /* sample intensity according to 13.5 */
      $I(I_i, \vec{h}) \leftarrow$ SampledIntensity(I_i, $\vec{h}$);
      /* Blend particles together, e.g. by
         front-to-back blending */
      $\hat{I} \leftarrow$ blend($\hat{I}$, $I(I_i, \vec{h})$);
    end
    $\vec{o} \leftarrow \vec{o}'$;
  end

Algorithm 4: Volume rendering of the APR. See text for a detailed explanation of the steps.

where radial basis functions are used as a basic primitive to emulate rasterisation-based billboard splatting via raytracing. Compared to [Knoll et al., 2019], we however do not need to build an acceleration data structure, as the APR already provides a decomposition of space from which the particles can efficiently be accessed. See Figure 13.11 for an example how the APR decomposes space into cells in an example dataset. Using this decomposition, the dataset can be traversed efficiently for raycasting. The particles in the APR then also have a naturally-defined bounding box, and can be interpolated using Gaussians/Radial Basis Functions, or a piecewice constant function (or for that matter, any custom interpolation method). For calculating the intersection of a particle i at position $\vec{p}_i$ with radius r_i with a ray of normalised direction $\vec{d}$ and origin $\vec{o}$, the hit point $\vec{h}$ is given as

$$\vec{h} = \vec{o} + |\vec{p}_i - \vec{o}| \cdot \vec{d}. \tag{13.4}$$

Depending on whether to reconstruct using a Gaussian or a piecewise constant function, the sampled intensity $I(I_i, \vec{h})$ then is one of

$$\begin{aligned} I(I_i, \vec{h})_{\text{Gaussian}} &= I_i \cdot \exp\left(-\frac{\left(\vec{h} - \vec{p}_i\right)^2}{r_i^2} \right) \\ I(I_i, \vec{h})_{\text{Piecewise}} &= I_i, \end{aligned} \tag{13.5}$$

where I_i is the intensity of the particle i. The algorithm for APR traversal for a single ray of origin $\vec{o}$, direction $\vec{d}$, and maximum length $d_{\max}$ is given in Algorithm 4.

13.11 Discussion and Future Work

We have introduced different visualisation and rendering techniques for the Adaptive Particle Representation (APR), and provided a brief summary of how it is constructed from a given image. We have shown the evolution of rendering the APR, going from an initial, unshaded prototype, over particle-based rendering with interpolated surface normals and ambient occlusion, to maximum intensity projection and a proposal for volume rendering on the APR data structures.

At the moment, we are finalising the interfacing of the `libapr` C++ code with Java code, such that scenery can fully benefit from the APR data structures and transfer these directly to the GPU without any conversions necessary such that the algorithm described in Section 13.10, *Particle-based volume rendering of the APR on the GPU*, can be implemented efficiently and without resorting to additional data structures. When the integration is done and verified, we want to perform benchmarking, comparing regular volume rendering to volume rendering on the APR. We expect a large speedup compared to pixel-based images, especially for large, but sparse volumetric datasets of >100 TB, where the APR can show its full potential.

Additionally, the spatial sampling of the APR is currently being extended to the

time domain by Bevan Cheeseman (APR+t). For multi-timepoint datasets, this is going to provide additional data reduction, such that volume rendering of multi-timepoint datasets with high timepoint sizes could also become much faster.

On the applications side, remote 3D collaboration on volumetric datasets is currently hampered by the need to either possess or transfer the dataset to all participants — for gigabyte-sized datasets a nuisance, for terabyte-sized datasets nearly impossible without time-consuming preparation, and sharing of data via sneakernet. Alternatively, browser-based approaches like CATMAID [Saalfeld et al., 2009] can be used, but suffer from high latency — especially in the case of VR/AR rendering — and the need for centralised, fast hardware.

The APR, and especially the APR+t, provides a solution here, by reaching 20 to 100-fold data reduction, moving gigabytes to megabytes, and terabytes to gigabytes. As scenery already includes capabilities for synchronisation over the network, remote 3D collaboration on large volumetric datasets is an interesting research avenue for the future.

Chapter 14:

sciview — Integrating scenery into ImageJ2 & Fiji

The work presented in this chapter has been done in collaboration with Tobias Pietzsch (MPI-CBG), Curtis Rueden (University of Wisconsin, Madison), Stephan Daetwyler (MPI-CBG and UT Southwestern, Texas), and Kyle I.S. Harrington (University of Idaho, Moscow, and HHMI Janelia Farm), and is currently being prepared for publication as:

Günther, U., Pietzsch, T., Rueden, C., Daetwyler, S., Huisken, J., Elicieri, K., Tomancak, P., Sbalzarini, I.F., Harrington, K.I.S.: sciview — Next-generation 3D visualisation for ImageJ & Fiji.

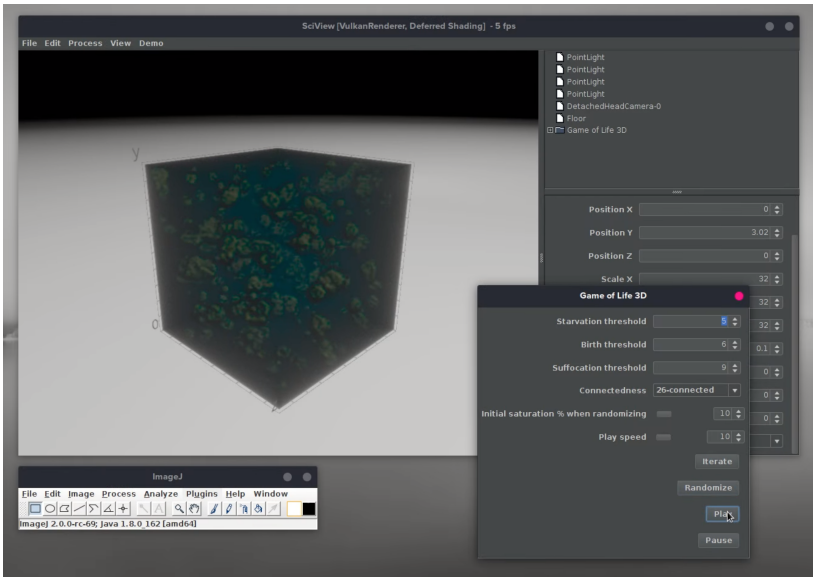

Figure 14.1: Screenshot of the sciview main window, showing the Game of Life 3D demo.

Fiji [Schindelin et al., 2012] is a widely-used — as of April 2019, it has been cited over 10000 times — open-source distribution of ImageJ for biological image analysis (Fiji stands for "Fiji is just ImageJ"). It is now the predominant ImageJ [Schneider et al., 2012] distribution. It recently had its its underlying infrastructure modernised tremendously, e.g. by replacing its basic image processing library with imglib2 [Pietzsch et al., 2012], and replacing the original ImageJ 1.x with ImageJ2,

which has brought better interoperability and better overall design [Rueden et al., 2017].

The original 3D visualisation tool in Fiji is the 3D Viewer [Schmid et al., 2010], which has now become dated and is not actively developed anymore, necessitating a replacement. After the start of the development of scenery in early 2016, we started the development of a replacement for 3D Viewer, named *sciview*. sciview builds on the infrastructure provided by the Fiji, SciJava[1], and ImageJ2 projects [Rueden et al., 2017].

[1] See scijava.org and imagej.net/SciJava for more information about the projects.

In this chapter, we introduce sciview and explain how intertwining scenery and the ImageJ2/Fiji ecosystem creates a powerful new tool for interactive visualisations in the life sciences. We start by introducing the ImageJ2/Fiji ecosystem in more detail, focussing on the developer side and explaining how the various parts are used in our project. After that, we introduce examplary use cases that have not been possible before, and projects that are enabled by scenery and sciview.

14.1 Integration into the ImageJ2 & Fiji ecosystem

With its basis, SciJava common, ImageJ2 focuses heavily on modularisation, extensibility, and interoperability. At the core of ImageJ2 is the support for N-dimensional image data, going beyond the "traditional" stack of 2D images from microscopy, and extending towards multispectral and hyperspectral images, which might even include information about wavelengths, sample counts, polarisation states, etc. On the interoperability side, the SciJava infrastructure enables simple integration of plugins that only need to be written once into several different software suites, such as ImageJ2/Fiji, KNIME [Berthold et al., 2008], and Icy [de Chaumont et al., 2012].

We make use of three main components from the ImageJ2 effort:

- SciJava common, for providing core abstractions for extensible applications, such as for plugins and commands (more on that in a moment),
- SCIFIO, the *SCientific Image Format Input and Output* library, facilitating image input and output in various formats, as well as interoperability between formats, and
- ImgLib2 [Pietzsch et al., 2012], which decouples image representation from processing and storage.

SciJava common provides several services that enable integration into ImageJ2/Fiji installations:

- the `PluginService` for dynamic plugin discovery at runtime,
- the `EventService`, for publishing and subscribing to scenery-related events, such as Node additions, removals, and changes,
- the `IOService` for providing access to file input/output, e.g., via SCIFIO.

sciview itself is implemented as a SciJava `Service`, the `SciViewService`. A user can create a new sciview instance by running `SciViewService.createSciView()`,

```
@Plugin(type = Command.class, menuRoot = "SciView", //
        menu = { @Menu(label = "Edit", weight = EDIT), //
                 @Menu(label = "Add Volume", weight = EDIT_ADD_VOLUME) })
public class AddVolume implements Command {

    @Parameter
    private SciView sciView;

    @Parameter
    private Dataset image;

    @Parameter(label = "Voxel Size X")
    private float voxelWidth = 1.0f;

    @Parameter(label = "Voxel Size Y")
    private float voxelHeight = 1.0f;

    @Parameter(label = "Voxel Size Z")
    private float voxelDepth = 1.0f;

    @Parameter(label = "Global rendering scale")
    private float renderScale = 1.0f;

    @Override
    public void run() {
        Node n = sciView.addVolume( image, new float[] { voxelWidth, ↩
     voxelHeight, voxelDepth } );
        n.setRenderScale(renderScale);
    }

}
```

Listing 14.1: Example SciJava Command for adding a volume to a sciview scene.

or get an already active instance by `SciViewService.getActiveSciView()`. sciview instances can also be named, and later accessed independently via `SciViewService.getSciView(name: String)`.

All commands the user can execute from the sciview main window are implemented as SciJava `Command` `Plugins`. As a simple and instructive example, the `Command` that adds the `Edit > Add Volume` menu item is shown in Listing 14.1.

This example code illustrates one of the prime features of ImageJ2: the separation of data model and view (or GUI). All of the class members annotated as `@Parameter` are going to be either populated automatically or exposed in the UI by SciJava's plugin infrastructure: all the members referring to `Dataset`, or any kind of `Service` will be automatically populated with the currently open dataset or services at hand from the ImageJ instance. E.g, the parameter `sciView` will point to the currently active sciview instance. Members labelled `@Parameter` with no relation to a service will be user-editable parameters shown in the GUI dialog. How this `Command` is rendered in the default ImageJ2 Swing GUI is shown in Figure 14.1. Such parameters can be named (`label`), given minimum and maximum boundaries, or styled as a particular widget.

At the moment, all automatically-generated GUIs are using Swing. The generation system however is abstract enough such that Swing can be replaced by JavaFX or another UI toolkit in the future.

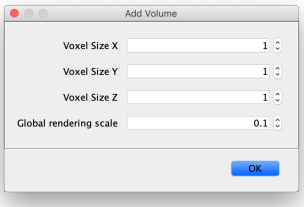

Figure 14.2: A simple example for an automatically generated UI: sciview's *Add Volume* dialog, shown in the default ImageJ2 Swing GUI.

14.2 Example Use Cases

The remainder of this chapter is intended to showcase applications of sciview. We introduce three examples and discuss how sciview has helped in creating them.

14.2.1 Zebrafish development

The visualisations shown in this section have been published as part of:

Daetwyler S., **Günther, U.**, Modes, Carl D., Harrington, K.I.S., and Huisken, J.: Multi-sample SPIM image acquisition, processing and analysis of vascular growth in zebrafish. *Development* 146 (6), 2019. bioRxiv preprint 478149.

Scan this QR code to go to a video demo of zebrafish vasculature development visualised in sciview. For a list of supplementary videos see https://ulrik.is/writing/a-thesis.

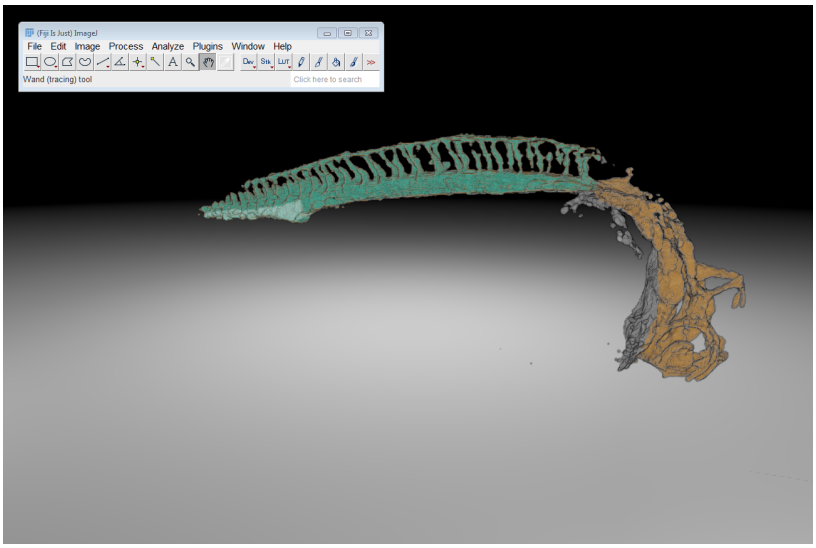

Figure 14.3: Screenshot of sciview, showing a multicolour segmentation of *Danio rerio* vasculature. Dataset courtesy of Stephan Daetwyler, Huisken Lab, MPI-CBG Dresden and Morgridge Institute for Research, Madison, USA.

For the publication [Daetwyler et al., 2018] we developed a custom visualisation pipeline to cope with the terabytes of image data generated in experiments which simultaneously imaged multiple *Danio rerio* embryos over the course of several days in order to investigate vascular development. This was developed before scenery gained support for out-of-core volume rendering, so alternative techniques had to be employed to create timelapse videos of vascular development. The resulting script for controlling sciview to create the animation shown in Figure 14.4 and the supplementary video is shown in Listing 14.2 — it reads all TIFF files from a given location (line 19 onward), and iterate through them one-by-one (line 23), saving a screenshot on each iteration (line 35).

```
1 // import necessary packages
2 importPackage(java.nio.file);
3 importPackage(java.io);
4 importPackage(java.lang);
5 importPackage(java.util);
6
7 // get sciview object
8 var sc = sciView.getActiveSciView();
9
```

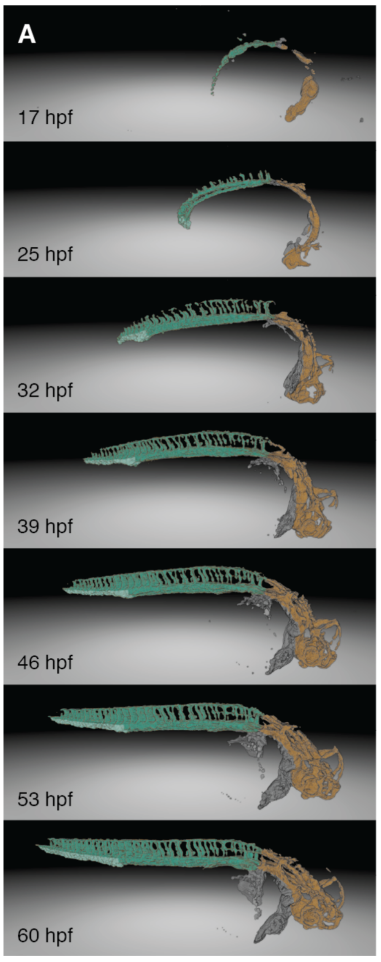

Figure 14.4: Frames from a developmental timelapse of *D. rerio* rendered in sciview, from [Daetwyler et al., 2018].

```
// get scene object
var scene = sc.getAllSceneNodes()[0].parent;

// get imported volume, number 6 in the array
var fish = scene.children[6];
// scale to half the original size
fish.renderScale = 0.5;

// get all TIFF files and sort by name
var files = new File("C:/my/fish/data/tiff/combined/angle000/").listFiles();
Arrays.sort(files, 0, files.length-1);

// iterate over all files
for(i = 0; i < files.length-1; i++) {
    var f = Paths.get(files[i]);
    fish.readFrom(f, true);
    Thread.sleep(1500);

    // adjust transfer function range and LUT
    fish.trangemin = 0.0;
    fish.trangemax = 255.0;
    sc.setColormap(fish, lut.loadLUT(lut.findLUTs().get("VirtualFishAssignment.↩
     lut")));
    Thread.sleep(500);

    // take a screenshot (which is saved to disk)
    sc.takeScreenshot();
    Thread.sleep(1500);
}
```

Listing 14.2: sciview example script in JavaScript to display a zebrafish vasculature developmental timelapse, with each frame loaded individually. See text for details.

14.2.2 Constrained Segmentation of the Zebrafish heart

In this use case, sciview is leveraged to simplify the segmentation procedure for parts of the zebrafish heart by introducing interactive mesh-based cropping as a pre-segmentation step.

Segmentation of intricate structures in 3D microscopy images, such as the zebrafish heart, using simple, off-the-shelf algorithms and suites such as *Weka* [Hall et al., 2009], is often hampered by oversegmentation or noise. If the user can interactively constrain the region to be segmented, the performance of such algorithms can be increased tremendously.

The approach is the following:

1. The user uses ImageJ's point selection tool to roughly select points that define a convex shape around the region of interest for segmentation,
2. the image is visualised as a 3D volume in sciview, and the points from the point selection are shown as spheres within the volume,
3. the user moves the points around, either using a mouse, or VR controllers, until they constrain the dataset in a satisfactory manner (see Figure 14.5a),
4. a convex hull is calculated from such points using sciview's `InteractiveConvexMesh` command,

Scan this QR code to go to a video demo of the constrained segmentation. For a list of supplementary videos see https://ulrik.is/writing/a-thesis.

5. the convex hull is used to determine the inside of the relevant region of the volumetric dataset, and outside parts are removed or set to zero (see Figure 14.5b).
6. the resulting dataset is used with a trainable *Weka* segmenter in 3D.

For the last step, less than 10 annotations had to be manually performed, in order to reach a good segmentation quality, as shown in Figure 14.6. This approach involves a tractable effort, and is able to effectively reduce formerly impossible segmentation problem to feasible ones.

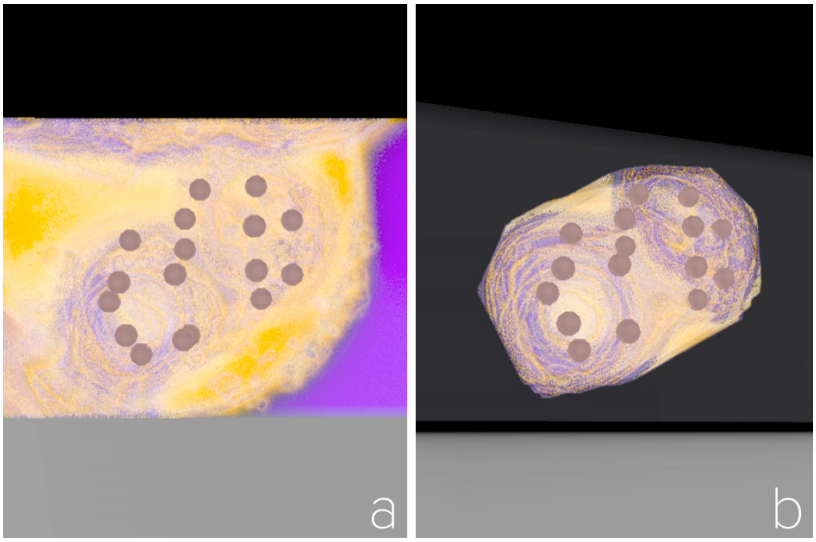

Figure 14.5: Constraining a volumetric image for segmentation using the *thingy* command in SciView. a: Constrain points drawn into dataset (step 1), b: Outside of convex hull removed (step 5). See text for details. Dataset courtesy of Anjalie Schleppi, Huisken Lab, MPI-CBG and Morgridge Institute for Research.

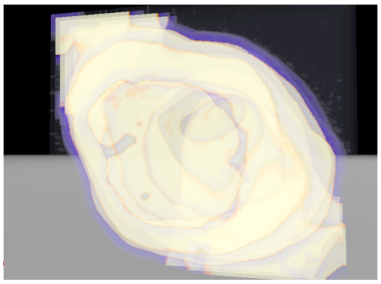

Figure 14.6: Constrained segmentation result using a trainable Weka segmenter on the volume cropped before (step 6), as shown in Figure 14.5. See text for details. Dataset courtesy of Anjalie Schleppi, Huisken Lab, MPI-CBG and Morgridge Institute for Research.

14.2.3 EmbryoGen – Generating test data for algorithmic analysis of lightsheet imaging data

Vladimir Ulman (Tomancak Lab, MPI-CBG) has developed a software, *EmbryoGen*, to create artificial, but realistic-looking images of developing *Drosophila* embryos. EmbryoGen was developed in order to be able to compare segmentation and tracking algorithms with actual ground truth data of cell positions, sizes, and velocities, which is normally not available for microscope-acquired fluorescence microscopy datasets.

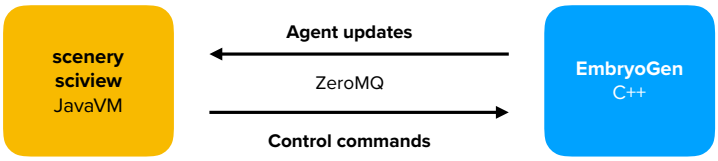

Figure 14.7: EmbryoGen architecture, with the actual simulation written in C++ talking to scenery and sciview via a ZeroMQ-based protocol. See text for details.

EmbryoGen's visualisation is based on scenery and sciview, in order to harness the simultaneous visualisation of mesh data, coming from simulated cell shapes and positions, and volumetric data, calculated from the simulated cells. Employing a client-server architecture, EmbryoGen has the actual simulation code written in C++, and communicates with scenery and sciview via a simple, ZeroMQ-based protocol. A sketch of the architecture is shown in Figure 14.7. The support of instancing in

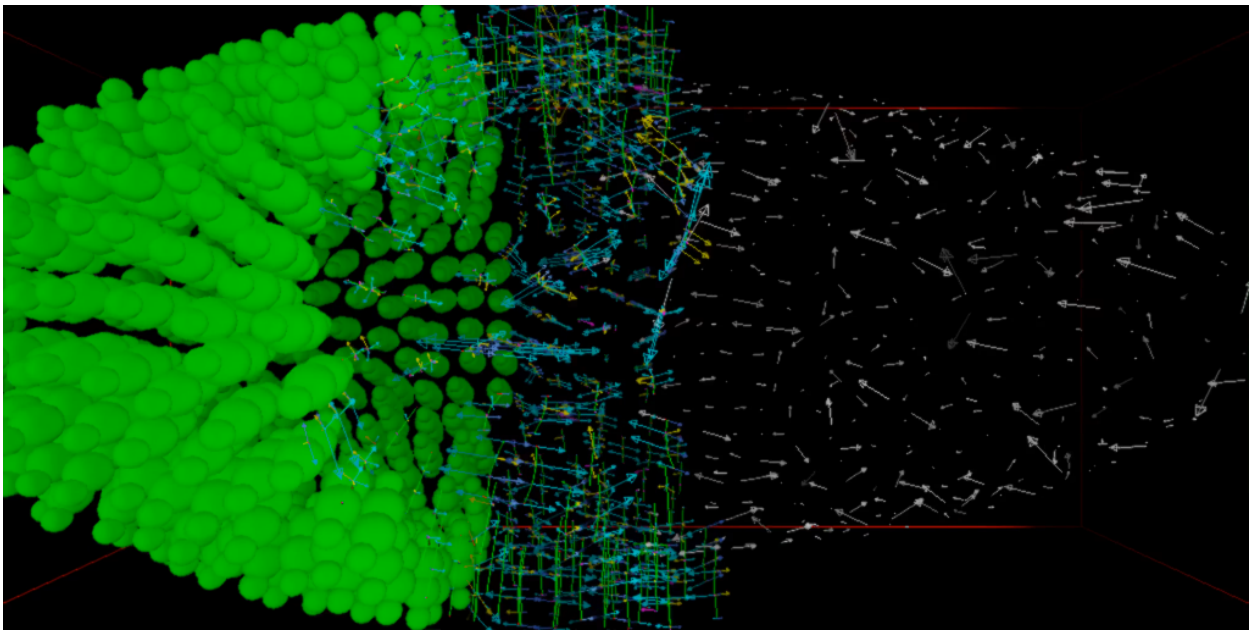

Figure 14.8: Visualisation of a simulated *Drosophila* embryo using *EmbryoGen* in sciview. The cells are shown as green spheres, while the equilibrating forces acting on them are shown as arrows on the right side where the cells are hidden. Courtesy of Vladimir Ulman, Tomancak and Jug Labs, MPI-CBG and Center for Systems Biology Dresden.

scenery (see Section 6.2.7, *Instancing*) is helpful in this particular use case, as the number of simulated cells can easily reach many tens of thousands.

The visualisation of the simulation is used for debugging and demonstration purposes: The cells can be visualised individually (see Figure 14.8, with cells shown in green, and the forces acting on them as white arrows. Furthermore, the optical flow induced by the cells moving can be visualised (see Figure 14.9.

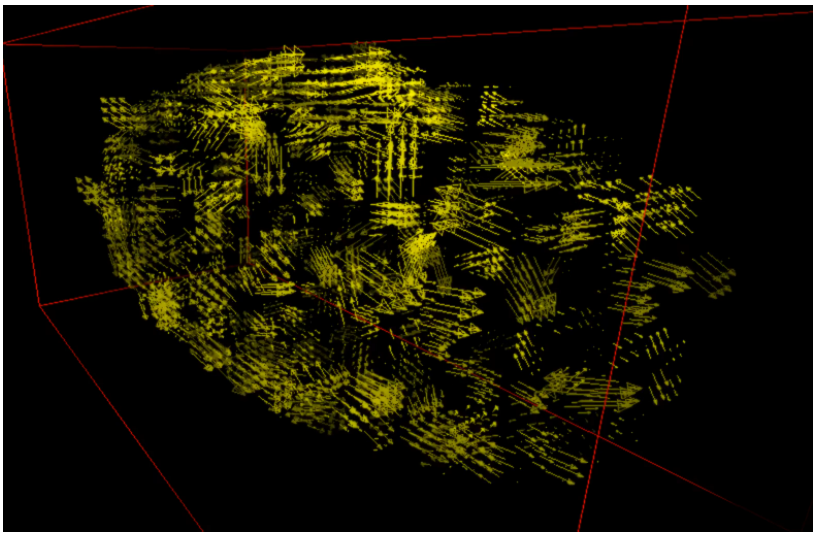

Figure 14.9: Visualisation of the optical flow generated by a simulated *Drosophila* embryo using *EmbryoGen* in sciview. Courtesy of Vladimir Ulman, Tomancak and Jug Labs, MPI-CBG and Center for Systems Biology Dresden.

14.2.4 Agent-based Simulations

We have used sciview to visualise agent-based simulations with large numbers of agents. By adapting the existing agent- and physics-based simulation toolkit *brevis* [Harrington and Stiles, 2017], we were able to increase the number of agents that

can be visualised at interactive frame rates by a factor of 10, from originally 1000, on a notebook equipped with a Nvidia Quadro P4000 GPU with 8 GB of graphics memory.

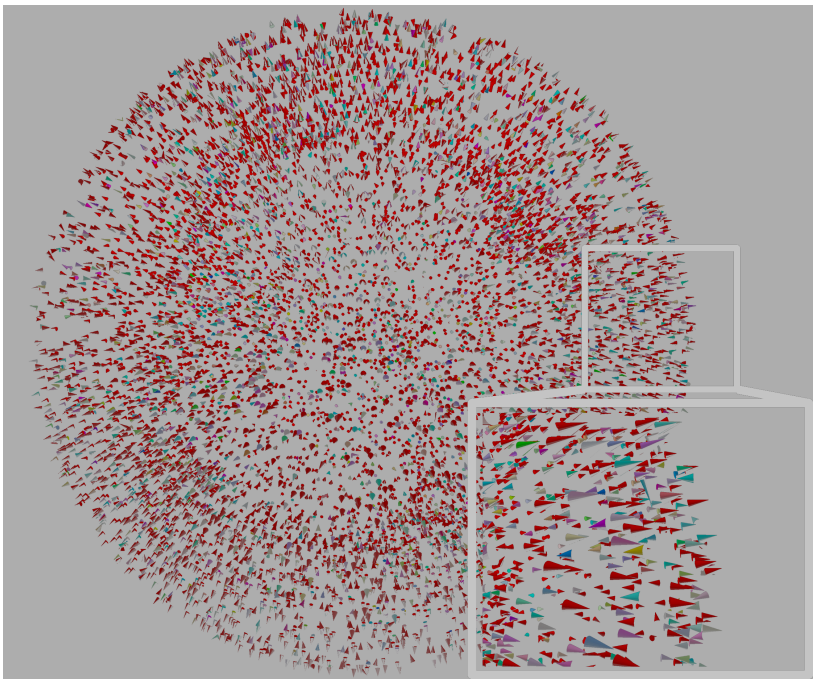

Figure 14.10: An agent-based simulation of 10.000 independent agents simulated using *brevis* and visualised using *sciview*. See text for details.

This performance improvement enables previous studies of swarms with evolving behaviours to be revisited under conditions that may enable new levels of emergent behaviour [Harrington and Magbunduku, 2017, Gold et al., 2014]. In Figure 14.10, we show 10.000 agents that use flocking rules inspired by [Reynolds, 1987] to collectively form a sphere.

Subsequently, we were able to further increase the number of agents, pushing the limits of scenery. The highest number of agents we could visualise was 2.500.000, albeit at a framerate of only about 2-5 fps, as the GPU becomes overwhelmed with vertex and geometry processing.

14.3 Conclusions and Future Work

In the future, we want to integrate sciview even more tightly with the existing ImageJ ecosystem, for example by making image processing commands available in a context-based menu that can also be used in VR/AR, such that image processing tasks can be executed without having to take off the headset.

In order to make sciview more interoperable with other ecosystems, we will provide wrapper libraries. Such a library has already been developed to use sciview from Python, enabling use of, e.g., SciPy, Numpy, TensorFlow, or PyTorch[2], is currently undergoing testing, and will be published as open-source software in the near future.

Furthermore, we would like to provide better interoperability with simulation frameworks such as Morpheus [Starruß et al., 2014] or OpenFPM [Incardona et al.,

[2] SciPy and Numpy (see scipy.org) are Python frameworks for scientific computing, while TensorFlow (tensorflow.org) and PyTorch (pytorch.org) are widely-used frameworks for GPU-accelerated deep learning and neural network prototyping.

2019], to better facilitate the visualisation and analysis of experimental data in conjunction with data from models and simulations. In this context, we have already done preliminary work for simulating and visualising the growth of artificial neurons in conjunction with Simple Neurite Tracer [Longair et al., 2011] and a custom-built growth simulation.

While the current GUI of sciview is based on Swing, we would like to provide support for JavaFX in the future as well. If Fiji/ImageJ2 also move to JavaFX, this would provide a very powerful combination, as it opens up the possibility to also run on touch-based devices, such as smartphones and tablets.

14.4 Software Availability

sciview is available as free and open-source software under the 2-clause BSD license at github.com/scenerygraphics/sciview. It is also available via a ImageJ/Fiji update site (see https://imagej.net/Update_Sites for details) under https://sites.imagej.net/SciView. An automatically-built unstable version also available in addition at https://sites.imagej.net/SciView-Unstable.

Part IV:

Conclusion

I expect that within the next five years more
than one in ten people will wear
head-mounted computer displays while
traveling in buses, trains, and planes.

—Nicolas Negroponte (1993)

Chapter 15:

Conclusions and Outlook

In this thesis, we have introduced scenery, a flexible and extensible rendering framework for the Java Virtual Machine, along with several use cases where the functionality of scenery is either used to improve workflows in systems biology, or integrated into existing software ecosystems for wider dissemination.

Now, we would like to revisit the individual contributions, and provide an outlook towards future research challenges.

15.1 Framework

With *scenery*, we have introduced a framework using the state-of-the-art rendering API Vulkan in order to provide VR/AR rendering of geometric and volumetric datasets of large size. The initial development of scenery was motivated by the need to make 3D data actually 3D, and harness the perceptual improvements brought by VR/AR for biological problems, such as tracking or instrument control. scenery was developed with five goals in mind:

1. VR/AR support, both for headsets and room-scale systems like CAVEs,
2. Out-of-core volume rendering of large datasets,
3. User- and developer-friendly API
4. Cross platform, support for at least Windows, macOS, and Linux, and
5. Run natively on the Java VM, and be embeddable in existing applications.

No existing software packages fulfilled these goals when we started the development of scenery, and at the time of writing, this still holds true. scenery is also the first framework for scientific visualisation that uses the modern Vulkan API.

In addition to the already mentioned VR/AR support and the out-of-core volume rendering of large datasets, we succeeded in creating a cross-platform framework that can help produce prototypes and applications on the three major operating systems. The framework can be embedded in existing applications on the Java VM, like Fiji, and — according to user feedback — is both user and developer friendly.

In the future, we would like to extend the capabilities of scenery: This includes improved rendering algorithms as outlined in Chapter 10, *Future Development Di-*

rections, improved support for AR applications, and venturing into new areas, such as *in situ* visualisation.

15.2 Case studies and applications

With the case studies in this work we demonstrated two things:

1. VR/AR can provide a tangible benefit for various visualisation and interaction applications in systems biology, and
2. scenery is a powerful and practical tool to develop such.

In Chapter 11, *Bionic Tracking: Using Eye Tracking for Cell Tracking*, we combined virtual reality exploration of 4D biological datasets with eye tracking technology to enable the user to track cells by just looking at them, and following them around in time and space. We chose to develop this in virtual reality, as the tracking enables the user to look and move around, and even perform evasive movements, without the necessity for additional input devices, mimicking the exploration of objects in the real world. The combination of the tracking information from the virtual reality headset with the eye tracking data provided a powerful way to discern real and spurious cell detections, and an easy way to reduce the problem of finding a cell from 3D to 1D — from the whole 3D dataset, to just along a ray through the dataset. We have demonstrated a graph search-based approach, similar to fringe-preserving A* search, to connect detections of cells between multiple timepoints. The user study we have conducted shows that users are able to create cell tracks of good quality with Bionic Tracking, and have indicated that, compared to other manual annotation methods, Bionic Tracking might speed up the process by at about an order of magnitude — not only significantly lowering the workload on the person performing the tracking, but also improving the ergonomics of the tracking process.

In the future, we want to continue the validation of our approach, e.g. with pre-annotated datasets or simulated datasets, where ground truth data is available, and make the technique available in tracking software packages, such as TrackMate, MaMuT, or Mastodon.

In Chapter 12, *Towards Interactive Virtual Reality Laser Ablation*, we have developed a prototype for the virtual reality control of laser ablation devices on volumetric microscopes, in order to make spatiotemporally complex processes accessible for experimental interference. We tested our prototypes with pre-recorded data on a set of people familiar with laser ablation, and collected their feedback. The feedback of the users has been mostly positive, with most of the users stating the presented technique provides an improvement over the state-of-the-art and could enable them to perform experiments not doable before, or perform experiments more quickly or precisely. From the user feedback, we have also collected ideas for future developments, namely the inclusion of a *virtual toolbelt* that can provide multiple cutting geometries, as well as a macro mode to easily produce highly reproducible laser cuts even in difficult geometries. We have furthermore sketched ideas for an OpenSPIM-based lightsheet microscope with an ablation unit.

In the future, we are going to implement the suggestions from the user tests, and build a microscope to actually perform the actual laser ablation, and conduct another user test with the actual microscope, with data acquired in realtime.

In Chapter 13, *Rendering the Adaptive Particle Representation*, we have given a short introduction to the Adaptive Particle Representation (APR) as a new and highly efficient way of representing fluorescences images. In this case study, we have shown that scenery can also adapt to new, unconventional representations of image data, and can be a useful tool both during development and use of such. We introduced the prototyping stages for visualising the APR, presented particle-based rendering, and CPU-based maximum intensity projections of the APR, and finally outlined a new algorithm to render the APR in a similar way to volume renderings on the GPU via raytracing.

In the future, we would like to finish the implementation of the raycasting algorithm, and evaluate the performance of the raycasting algorithm, especially on very large datasets exceeding multiple terabytes when stored as traditional pixel images. Additionally, we want to integrate the APR rendering methods into sciview.

In Chapter 14, *sciview — Integrating scenery into ImageJ2 & Fiji*, we have introduced sciview, a visualisation plugin for ImageJ2 & Fiji, based on the scenery framework. With sciview, we combine the advanced rendering and customisation capabilities of scenery, with a user-friendly interface in a software package that is widely used within the biomedical imaging community. We have shown how sciview harnesses the SciJava infrastructure for efficient integration into the ecosystem, and demonstrated example use cases, including the visualisation large data originating from the imaging of zebrafish vascular development, manually-constrained image segmentation, embryo simulation for ground truth generation, and visualisation of agent-based simulations.

In the future, we would like to integrate sciview even tighter into the ecosystem and associated applications, and provide an actual VR interface for performing image analysis tasks, as such tasks at the moment still have to be performed with the regular menu structure in Fiji.

With these four case studies, we believe that we have demonstrated both theses stated above: In interactive ablation, VR is invaluable for exploration and understanding of the dataset, and scenery helped to rapidly prototype interactions. For Bionic Tracking, scenery has provided the crucial underpinning, providing visualisation of large volumetric data, VR, and eye tracking support. With the demonstration of APR rendering, we have shown that scenery is a powerful toolkit even for unusual kinds of volumetric data, and finally, with sciview, we demonstrated scenery can be used to build applications and make VR available to a wider audience in the biomedical imaging community.

With this, we conclude this thesis. Thank you for reading.

Appendix A:

Questionnaire for VR Ablation User Study

This appendix shows the full questionnaire test subjects had to fill out for the user study in the chapter 12, *Towards Interactive Virtual Reality Laser Ablation* — for more details about the study please see this chapter. The questionnaire is a multi-page PDF and starts on the next page.

Freeform Interactive Laser Ablation using Virtual Reality

Please make sure you have understood and signed both the *Declaration of Consent* and the *Declaration of Consent for the Storage of Personal Data for Research Purposes* before filling out this form.

About you

1. Age _____
2. Gender _____
3. Dominant Hand
   - □ Left
   - □ Right
4. Do you require any optical aid?
   - □ No
   - □ Yes, _____
5. Are you wearing any right now?
   - □ No
   - □ Yes, _____
6. Do you have any impairment in color vision?
   - □ No
   - □ Yes, _____
7. Do you have any impairment in spatial perception?
   - □ No
   - □ Yes, _____
8. Do you have any motor impairment?
   - □ No
   - □ Yes, _____

Pre-study questions

9a. Do you have any prior experience with VR applications or games?

None □—□—□—□—□ Daily use

9b. Have you used computer-based VR headsets before? *Examples for such devices are Oculus Rift, HTC Vive, etc.*

Never □—□—□—□—□ Daily use

9c. Did you enjoy the experience?

Not at all □—□—□—□—□ Very much

9d. Have you used smartphone-based VR headsets before? *Examples for such devices are Samsung Gear VR, Zeiss VR ONE Plus, etc.*

Never □—□—□—□—□ Daily use

9e. Did you enjoy the experience?

Not at all □—□—□—□—□ Very much

9f. Have you used standalone VR headsets before? *Examples for such devices are Oculus Go, or the Lenovo Mirage Solo.*

Never □—□—□—□—□ Daily use

9g. Did you enjoy the experience?

Not at all □—□—□—□—□ Very much

9h. Do you have any prior experience with laser ablation?

None □—□—□—□—□ Daily use

Condition before the study

10a. How tired do you feel? Not at all □—□—□—□—□ Very
10b. How concentrated are you? Not at all □—□—□—□—□ Very
10c. How motivated are you? Not at all □—□—□—□—□ Very
10d. Do you have a headache? Not at all □—□—□—□—□ Very
10e. Do you have dry or aching eyes? Not at all □—□—□—□—□ Very
10f. Do you feel nauseous? Not at all □—□—□—□—□ Very

Post-study questions

Please read the questions carefully, some are formulated in a affirmative way, some are not.

Condition after the study

11a. How tired do you feel? Not at all □—□—□—□—□ Very
11b. How concentrated are you? Not at all □—□—□—□—□ Very
11c. How motivated are you? Not at all □—□—□—□—□ Very
11d. Do you have a headache? Not at all □—□—□—□—□ Very
11e. Do you have dry or aching eyes? Not at all □—□—□—□—□ Very
11f. Do you feel nauseous? Not at all □—□—□—□—□ Very

General Questions

12a. The visualisation had a high fidelity. Not at all □—□—□—□—□ Very
12b. The visualisation was well-positioned in space. Not at all □—□—□—□—□ Very
12c. The visualisation had a good scale relative to the user. Not at all □—□—□—□—□ Very
12c. The software felt responsive to my inputs. Not at all □—□—□—□—□ Very
12d. Being in an isolated VR environment irritated me. Not at all □—□—□—□—□ Very
12e. I had trouble orienting myself. Not at all □—□—□—□—□ Very
12f. I would have liked a different input/control method. Not at all □—□—□—□—□ Very
12g. The usage felt very natural and intuitive. Not at all □—□—□—□—□ Very
12h. I had to keep track of too many things at once. Not at all □—□—□—□—□ Very
12i. The visualisation of information supports me in performing the task. Not at all □—□—□—□—□ Very
12j. I was put off by the prototype character of the software. Not at all □—□—□—□—□ Very
12k. I needed a long time to learn how to use the software. Not at all □—□—□—□—□ Very
12l. The interaction felt very precise. Not at all □—□—□—□—□ Very

Adoption Questions

13a. I could imagine adopting the presented technique for my experiments.
Not at all □—□—□—□—□ Completely
13b. The presented technique provides an improvement over current techniques.
Not at all □—□—□—□—□ Completely
13c. The presented technique would allow me to perform experiments faster.
Not at all □—□—□—□—□ Completely
13d. The presented technique would allow me to perform experiments more precisely.
Not at all □—□—□—□—□ Completely

Task Load Questions

14a. How mentally demanding was the task?

Very Low □—□—□—□—□—□—□ Very High

14b. How physically demanding was the task?

Very Low □—□—□—□—□—□—□ Very High

14c. How hurried or rushed was the pace of the task?

Very Low □—□—□—□—□—□—□ Very High

14d. How successful were you in accomplishing what you were asked to do?

Perfect □—□—□—□—□—□—□ Failure

14e. How hard did you have to work to accomplish your level of performance?

Very Low □—□—□—□—□—□—□ Very High

14f. How insecure, discouraged, irritated, stressed, and annoyed were you?

Very Low □—□—□—□—□—□—□ Very High

Simulator Sickness

Do you experience any of the following symptoms? How badly?

15a. General discomfort — None □—□—□—□ Very High

15b. Fatigue — None □—□—□—□ Very High

15c. Headache — None □—□—□—□ Very High

15d. Eyestrain — None □—□—□—□ Very High

15e. Difficulty focussing — None □—□—□—□ Very High

15f. Increased salivation — None □—□—□—□ Very High

15g. Sweating — None □—□—□—□ Very High

15h. Nausea — None □—□—□—□ Very High

15i. Difficulty concentrating — None □—□—□—□ Very High

15j. Fullness of head — None □—□—□—□ Very High

15k. Blurred vision — None □—□—□—□ Very High

15l. Dizzy with open eyes — None □—□—□—□ Very High

15m. Dizzy with closed eyes — None □—□—□—□ Very High

15n. Vertigo — None □—□—□—□ Very High

15o. Stomach awareness — None □—□—□—□ Very High

15p. Burping — None □—□—□—□ Very High

Appendix B:

Full Correlations in VR Ablation Questionnaire

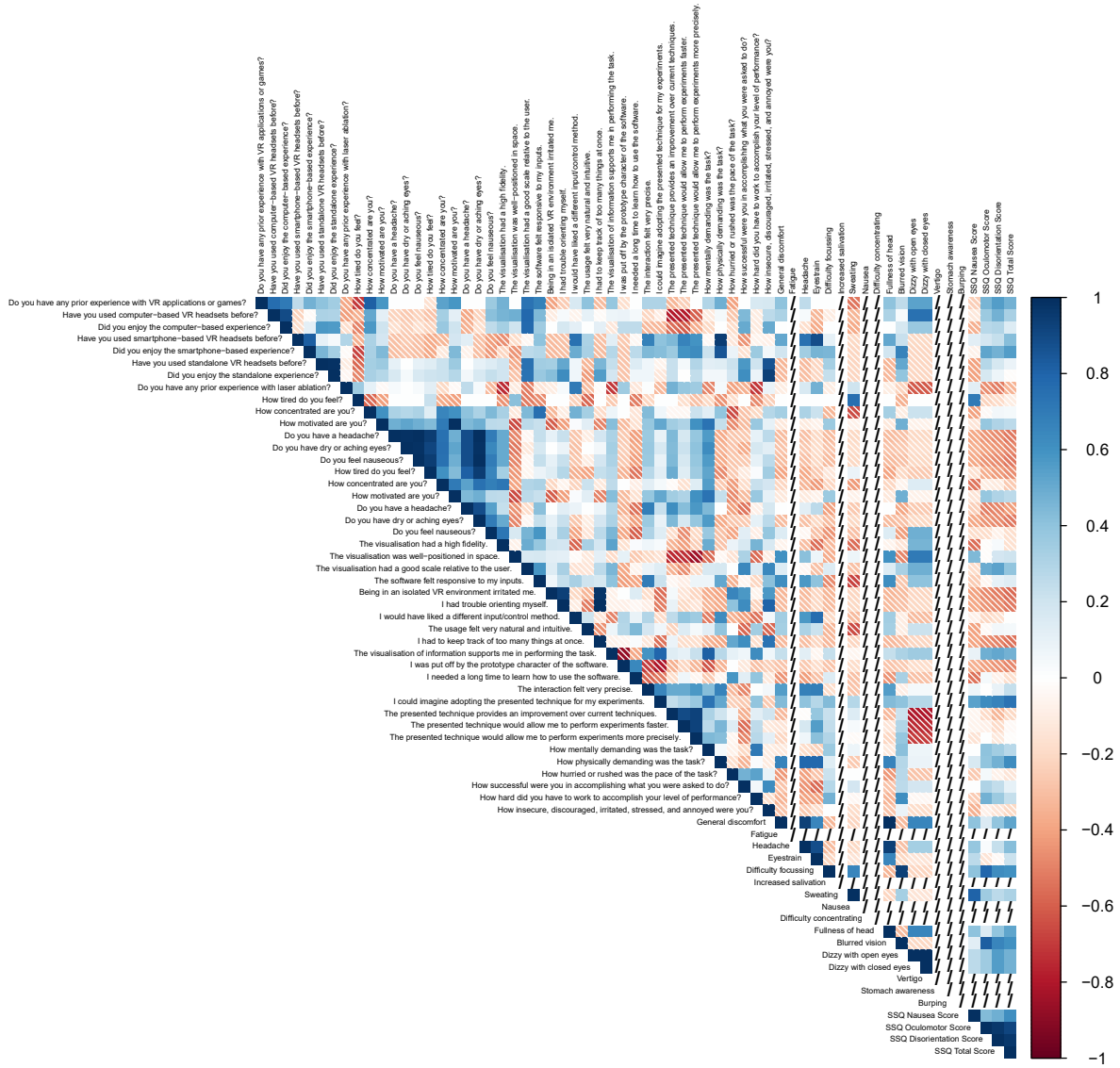

Figure B.1: Full correlation plot for questionnaire answers, including all SSQ symptoms. A black X indicates data where no correlation could be determine due to the standard deviation being zero. This is only the case for some SSQ symptoms that were not reported by any subject.

Appendix C:

Questionnaire for Bionic Tracking User Study

This appendix shows the full questionnaire test subjects had to fill out for the user study in chapter 11, *Bionic Tracking: Using Eye Tracking for Cell Tracking* — for more details about the study please see this chapter. The questionnaire is a multi-page PDF and starts on the next page.

bionic tracking: eye tracking for cell tracking in VR

Please make sure you have understood and signed both the *Declaration of Consent* and the *Declaration of Consent for the Storage of Personal Data for Research Purposes* before filling out this form.

About you

1. Age _____
2. Gender _____
3. Dominant Hand
   - □ Left
   - □ Right
4. Do you require any optical aid?
   - □ No
   - □ Yes, _____
5. Are you wearing any right now?
   - □ No
   - □ Yes, _____
6. Do you have any impairment in color vision?
   - □ No
   - □ Yes, _____
7. Do you have any impairment in spatial perception?
   - □ No
   - □ Yes, _____
8. Do you have any motor impairment?
   - □ No
   - □ Yes, _____

Pre-study questions

9a. Do you have any prior experience with VR applications or games?

None □—□—□—□—□ Daily use

9b. Have you used computer-based VR headsets before? *Examples for such devices are Oculus Rift, HTC Vive, etc.*

Never □—□—□—□—□ Daily use

9c. Did you enjoy the experience?

Not at all □—□—□—□—□ Very much

9d. Have you used smartphone-based VR headsets before? *Examples for such devices are Samsung Gear VR, Zeiss VR ONE Plus, etc.*

Never □—□—□—□—□ Daily use

9e. Did you enjoy the experience?

Not at all □—□—□—□—□ Very much

9f. Have you used standalone VR headsets before? *Examples for such devices are Oculus Go, or the Lenovo Mirage Solo.*

Never □—□—□—□—□ Daily use

9g. Did you enjoy the experience?

Not at all □—□—□—□—□ Very much

9h. Did you ever use an eye tracking-based user interface?

Never □—□—□—□—□ Daily use

9i. Do you have any prior experience with cell tracking?

None □—□—□—□—□ Daily use

Condition before the study

10a. How tired do you feel? Not at all □—□—□—□—□ Very

10b. How concentrated are you? Not at all □—□—□—□—□ Very

10c. How motivated are you? Not at all □—□—□—□—□ Very

10d. Do you have a headache? Not at all □—□—□—□—□ Very

10e. Do you have dry or aching eyes? Not at all □—□—□—□—□ Very

10f. Do you feel nauseous? Not at all □—□—□—□—□ Very

Post-study questions

Please read the questions carefully, some are formulated in a affirmative way, some are not.

Condition after the study

11a. How tired do you feel? Not at all □—□—□—□—□ Very

11b. How concentrated are you? Not at all □—□—□—□—□ Very

11c. How motivated are you? Not at all □—□—□—□—□ Very

11d. Do you have a headache? Not at all □—□—□—□—□ Very

11e. Do you have dry or aching eyes? Not at all □—□—□—□—□ Very

11f. Do you feel nauseous? Not at all □—□—□—□—□ Very

General Questions

12a. The software felt responsive to my inputs. Not at all □—□—□—□—□ Very

12b. Being in an isolated VR environment irritated me. Not at all □—□—□—□—□ Very

12c. I had trouble orienting myself. Not at all □—□—□—□—□ Very

12d. I would have liked a different input/control method. Not at all □—□—□—□—□ Very

12e. The usage felt very natural and intuitive. Not at all □—□—□—□—□ Very

12f. I had to keep track of too many things at once. Not at all □—□—□—□—□ Very

12g. I was put off by the prototype character of the software. Not at all □—□—□—□—□ Very

12h. I needed a long time to learn how to use the software. Not at all □—□—□—□—□ Very

12i. The interaction felt very precise. Not at all □—□—□—□—□ Very

12j. Having my eyes tracked irratated me. Not at all □—□—□—□—□ Very

12k. The cell tracks created looked reasonable to me. Not at all □—□—□—□—□ Very

12l. I could complete the tracking tasks with confidence. Not at all □—□—□—□—□ Very

12m. Relative to regular manual tracking, tracking with the presented technique would take _____ times the time.

Adoption Questions

13a. I could imagine adopting the presented technique for tracking of my datasets.

Not at all □—□—□—□—□ Completely

13b. The presented technique provides an improvement over current techniques.

Not at all □—□—□—□—□ Completely

13c. The presented technique would allow me to perform tracking tasks faster.

Not at all □—□—□—□—□ Completely

13d. The presented technique would allow me to perform tracking tasks more precisely.

Not at all □—□—□—□—□ Completely

Task Load Questions

14a. How mentally demanding was the task?

Very Low □—□—□—□—□—□—□ Very High

14b. How physically demanding was the task?

Very Low □—□—□—□—□—□—□ Very High

14c. How hurried or rushed was the pace of the task?

Very Low □—□—□—□—□—□—□ Very High

14d. How successful were you in accomplishing what you were asked to do?

Perfect □—□—□—□—□—□—□ Failure

14e. How hard did you have to work to accomplish your level of performance?

Very Low □—□—□—□—□—□—□ Very High

14f. How insecure, discouraged, irritated, stressed, and annoyed were you?

Very Low □—□—□—□—□—□—□ Very High

Simulator Sickness

Do you experience any of the following symptoms? How badly?

15a. General discomfort — None □—□—□—□ Very High

15b. Fatigue — None □—□—□—□ Very High

15c. Headache — None □—□—□—□ Very High

15d. Eyestrain — None □—□—□—□ Very High

15e. Difficulty focussing — None □—□—□—□ Very High

15f. Increased salivation — None □—□—□—□ Very High

15g. Sweating — None □—□—□—□ Very High

15h. Nausea — None □—□—□—□ Very High

15i. Difficulty concentrating — None □—□—□—□ Very High

15j. Fullness of head — None □—□—□—□ Very High

15k. Blurred vision — None □—□—□—□ Very High

15l. Dizzy with open eyes — None □—□—□—□ Very High

15m. Dizzy with closed eyes — None □—□—□—□ Very High

15n. Vertigo — None □—□—□—□ Very High

15o. Stomach awareness — None □—□—□—□ Very High

15p. Burping — None □—□—□—□ Very High

List of Tables

List of Figures

Bibliography

Marwan Abdellah, Ahmet Bilgili, Stefan Eilemann, Julian Shillcock, Henry Markram, and Felix Schürmann. Bio-physically plausible visualization of highly scattering fluorescent neocortical models for in silico experimentation. *BMC Bioinformatics*, 18 (S2), 2017. DOI: 10.1186/s12859-016-1444-4.

Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk. SLIC superpixels compared to state-of-the-art superpixel methods. *IEEE transactions on pattern analysis and machine intelligence*, 34(11), 2012. DOI: 10.1109/tpami.2012.120.

Edward H Adelson, Charles H Anderson, James R Bergen, Peter J Burt, and Joan M Ogden. Pyramid methods in image processing. *RCA engineer*, 29(6), 1984.

Marc Adrian, Jacques Dubochet, Jean Lepault, and Alasdair W. McDowall. Cryo-electron microscopy of viruses. *Nature*, 308 (5954), 1984. DOI: 10.1038/308032a0.

John M Airey, John H Rohlf, and Frederick P Brooks. Towards image realism with interactive update rates in complex virtual building environments. *ACM SIGGRAPH Computer Graphics*, 24(2), 1990. DOI: 10.1145/91385.91416.

Fernando Amat, Eugene W Myers, and Philipp J Keller. Fast and robust optical flow for time-lapse microscopy using super-voxels. *Bioinformatics*, 29(3), 2012. DOI: 10.1093/bioinformatics/bts706.

Fernando Amat, William Lemon, Daniel P Mossing, Katie McDole, Yinan Wan, Kristin Branson, Eugene W Myers, and Philipp J Keller. Fast, accurate reconstruction of cell lineages from large-scale fluorescence microscopy data. *Nature Methods*, 11(9), 2014. DOI: 10.1038/nmeth.3036.

Fernando Amat, Burkhard Höckendorf, Yinan Wan, William C Lemon, Katie McDole, and Philipp J Keller. Efficient processing and analysis of large-scale light-sheet microscopy data. *Nature Protocols*, 10(11), 2015. DOI: 10.1038/nprot.2015.111.

Behrooz Ashtiani and I. Scott MacKenzie. BlinkWrite2: An improved text entry method using eye blinks. In *Proceedings of the 2010 Symposium on Eye-Tracking Research & Applications - ETRA '10*, page 339, Austin, Texas, 2010. ACM Press. DOI: 10.1145/1743666.1743742.

Ejder Bastug, Mehdi Bennis, Muriel Medard, and Merouane Debbah. Toward Interconnected Virtual Reality: Opportunities, Challenges, and Enablers. *IEEE Communications Magazine*, 55(6), 2017. DOI: 10.1109/mcom.2017.1601089.

Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. Image-space horizon-based ambient occlusion. *ACM SIGGRAPH 2008 talks*, 2008.

Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3): 173–189, 1972. ISSN 0001-5903, 1432-0525. DOI: 10.1007/BF00288683.

David M. Beazley. SWIG : An Easy to Use Tool For Integrating Scripting Languages with C and C++. *usenix.org*, 1996.

Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Christoph Sieb, Kilian Thiel, and Bernd Wiswedel. KNIME: The Konstanz Information Miner. In Christine Preisach, Hans Burkhardt, Lars Schmidt-Thieme, and Reinhold Decker, editors, *Data Analysis, Machine Learning and Applications*, pages 319–326. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-78239-1 978-3-540-78246-9. DOI: 10.1007/978-3-540-78246-9_38.

Johanna Beyer, Markus Hadwiger, Torsten Möller, and Laura Fritz. Smooth Mixed-Resolution GPU Volume Rendering. 2008.

Tobias Boothe, Lennart Hilbert, Michael Heide, Lea Berninger, Wieland B Huttner, Vasily Zaburdaev, Nadine L Vastenhouw, Eugene W Myers, David N Drechsel, and Jochen C Rink. A tunable refractive index matching medium for live imaging cells, tissues and model organisms. *eLife*, 6, 2017. DOI: 10.7554/elife.27240.

Pierre Boudier and Cristoph Kubisch. GPU Driven Large Scene Rendering. In *GPU Technology Conference*, San Jose, 2015.

Edward S Boyden, Feng Zhang, Ernst Bamberg, Georg Nagel, and Karl Deisseroth. Millisecond-timescale, genetically targeted optical control of neural activity. *Nature Neuroscience*, 8(9), 2005. DOI: 10.1038/nn1525.

Naama Brenner, William Bialek, and Rob de Ruyter Van Steveninck. Adaptive rescaling maximizes information transmission. *Neuron*, 26(3), 2000.

Alessandro Bria, Giulio Iannello, Leonardo Onofri, and Hanchuan Peng. TeraFly: Real-time three-dimensional visualization and annotation of terabytes of multidimensional volumetric images. *Nature Methods*, 13(3), 2016. DOI: 10.1038/nmeth.3767.

John Brooke. SUS - A quick and dirty usability scale. In *Usability Evaluation In Industry*, page 7. CRC Press, June 1996. ISBN 978-0-7484-0460-5.

Frederick P. Brooks, Ming Ouh-Young, James J. Batter, and P. Jerome Kilpatrick. Project GROPEHaptic displays for scientific visualization. *ACM SIGGRAPH Computer Graphics*, 24(4), 1990. DOI: 10.1145/97879.97899.

Valentin Bruder, Christoph Schulz, Ruben Bauer, Steffen Frey, Daniel Weiskopf, and Thomas Ertl. Voronoi-based Foveated Volume Rendering. In *EUROVIS 2019*, Porto, Portugal, 2019.

Jan Brugués, Valeria Nuzzo, Eric Mazur, and Daniel J Needleman. Nucleation and Transport Organize Microtubules in Metaphase Spindles. *Cell*, 149(3), 2012. DOI: 10.1016/j.cell.2012.03.027.

Steve Bryson, Steven K. Feiner, Frederick P. Brooks, Philip Hubbard, Randy Pausch, and Andries van Dam. Research frontiers in virtual reality. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '94*, pages 473–474. ACM Press, 1994. ISBN 978-0-89791-667-7. DOI: 10.1145/192161.192287.

Andreas Bulling. Pervasive Attentive User Interfaces. *Computer*, 49(1), 2016. DOI: 10.1109/mc.2016.32.

Wolfgang Büschel, Stefan Vogt, and Raimund Dachselt. Augmented Reality Graph Visualizations: Investigation of Visual Styles in 3D Node-Link Diagrams. *IEEE Computer Graphics and Applications*, PP(99), 2019. DOI: 10.1109/mcg.2019.2897927.

Vivien A. Casagrande. A third parallel visual pathway to primate area V1. *Trends in Neurosciences*, 17(7):305–310, 1994. ISSN 01662236. DOI: 10.1016/0166-2236(94)90065-5.

Emiliano Castellina and Fulvio Corno. Multimodal gaze interaction in 3D virtual environments. In *Proceedings of the 4th Conference on Communication by Gaze Interaction – COGAIN 2008: Communication, Environment and Mobility Control by Gaze*, pages 33–37, 2008.

Edwin Catmull and Raphael Rom. A class of local interpolating splines. In *Computer Aided Geometric Design*, pages 317–326. Elsevier, 1974. ISBN 978-0-12-079050-0. DOI: 10.1016/B978-0-12-079050-0.50020-5.

Bevan L. Cheeseman, Ulrik Günther, Krzysztof Gonciarz, Mateusz Susik, and Ivo F. Sbalzarini. Adaptive particle representation of fluorescence microscopy images. *Nature Communications*, 9(1), 2018. DOI: 10.1038/s41467-018-07390-9.

Viktor Chlumsky. *Shape Decomposition for Multi-Channel Distance Fields*. PhD thesis, Czech Technical University, Prague, Czech Republic, 2015.

Zina Cigolle, Sam Donow, Daniel Evangelakos, Michael Mara, Morgan McGuire, and Quirin Meyer. A Survey of Efficient Representations for Independent Unit Vectors. *Journal of Computer Graphics Techniques*, 3, 2014.

Lee Clift. Novel Control Systems for Virtual Reality, and their Effects on Cyber Sickness. 2018.

Andy Cockburn, Amy Karlson, and Benjamin B Bederson. A review of overview+detail, zooming, and focus+context interfaces. *ACM Computing Surveys (CSUR)*, 41(1), 2009. DOI: 10.1145/1456650.1456652.

Robert P Collins, Jordan A Litman, and Charles D Spielberger. The measurement of perceptual curiosity. *Personality and Individual Differences*, 36(5), 2004. DOI: 10.1016/s0191-8869(03)00205-8.

Cyril Crassin. *GigaVoxels: A Voxel-Based Rendering Pipeline for Efficient Exploration of Large and Detailed Scenes*. PhD thesis, Universite de Grenoble, 2011.

Carolina Cruz-Neira, Daniel J Sandin, Thomas A DeFanti, Robert V Kenyon, and John C Hart. The CAVE: Audio visual experience automatic virtual environment. *Communications of the ACM*, 35(6), 1992. DOI: 10.1145/129888.129892.

Kathleen E. Cullen. Visuomotor Integration. In Donald W. Pfaff and Nora D. Volkow, editors, *Neuroscience in the 21st Century*, pages 961–1005. Springer New York, New York, NY, 2016. ISBN 978-1-4939-3473-7 978-1-4939-3474-4. DOI: 10.1007/978-1-4939-3474-4_25.

Stephan Daetwyler, Ulrik Günther, Carl D. Modes, Kyle I.S. Harrington, and Jan Huisken. Multi-sample SPIM image acquisition, processing and analysis of vascular growth in zebrafish. *bioRxiv*, 2018. DOI: 10.1101/478149.

Rudolph P. Darken, William R. Cockayne, and David Carmein. The omni-directional treadmill: A locomotion device for virtual worlds. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology - UIST '97*, pages 213–221, Banff, Alberta, Canada, 1997. ACM Press. ISBN 978-0-89791-881-7. DOI: 10.1145/263407.263550.

Geoff Davis, Stéphane Mallat, and Marco Avellaneda. Adaptive greedy approximations. *Constructive approximation*, 13(1), 1997.

Fabrice de Chaumont, Stéphane Dallongeville, Nicolas Chenouard, Nicolas Hervé, Sorin Pop, Thomas Provoost, Vannary Meas-Yedid, Praveen Pankajakshan, Timothée Lecomte, Yoann Le Montagner, Thibault Lagache, Alexandre Dufour, and Jean-Christophe Olivo-Marin. Icy: An open bioimage informatics platform for extended reproducible research. *Nature Methods*, 9(7), 2012. DOI: 10.1038/nmeth.2075.

Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, Neil Hunt, Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: A VLSI system for high performance graphics. *ACM SIGGRAPH Computer Graphics*, 22(4), 1988. DOI: 10.1145/378456.378468.

Thomas DeFanti, Daniel Acevedo, Richard Ainsworth, Maxine Brown, Steven Cutchin, Gregory Dawe, Kai-Uwe Doerr, Andrew Johnson, Chris Knox, Robert Kooima, Falko Kuester, Jason Leigh, Lance Long, Peter Otto, Vid Petrovic, Kevin Ponto, Andrew Prudhomme, Ramesh Rao, Luc Renambot, Daniel Sandin, Jurgen Schulze, Larry Smarr, Madhu Srinivasan, Philip Weber, and Gregory Wickham. The future of the CAVE. *Open Engineering*, 1(1), 2010. DOI: 10.2478/s13531-010-0002-5.

Laurent Demaret and Armin Iske. Scattered data coding in digital image compression. *Curve and Surface Fitting: Saint-Malo*, 2003, 2002.

Andrew Ted Duchowski. *Eye Tracking Methodology: Theory and Practice*. Springer, Cham, third edition edition, 2017. ISBN 978-3-319-57883-5 978-3-319-57881-1.

Arthur Edelstein, Nenad Amodaj, Karl Hoover, Ron Vale, and Nico Stuurman. Computer control of microscopes using μManager. *Current protocols in molecular biology*, 14, 2010. DOI: 10.1002/0471142727.mb1420s92.

Calvin D Eiber, Abrar S Rahman, Alexander N J Pietersen, Natalie Zeater, Bogdan Dreher, Samuel G Solomon, and Paul R Martin. Receptive Field Properties of Koniocellular On/Off Neurons in the Lateral Geniculate Nucleus of Marmoset Monkeys. *The Journal of Neuroscience*, 38(48), 2018. DOI: 10.1523/jneurosci.1679-18.2018.

Christoph J. Engelbrecht, Klaus Greger, Emmanuel G. Reynaud, Uroš Kržic, Julien Colombelli, and Ernst H. K. Stelzer. Three-dimensional laser microsurgery in light-sheet based microscopy (SPIM). *Optics Express*, 15(10):6420, May 2007. ISSN 1094-4087. DOI: 10.1364/OE.15.006420.

Ajoy S Fernandes and Steven K. Feiner. Combating VR sickness through subtle dynamic field-of-view modification. In *2016 IEEE Symposium on 3D User Interfaces (3DUI)*, pages 201–210, Greenville, SC, USA, March 2016. IEEE. ISBN 978-1-5090-0842-1. DOI: 10.1109/3DUI.2016.7460053.

Kristian Franze, Jens Grosche, Serguei N. Skatchkov, Stefan Schinkinger, Christian Foja, Detlev Schild, Ortrud Uckermann, Kort Travis, Andreas Reichenbach, and Jochen Guck. Müller cells are living optical fibers in the vertebrate retina. *Proceedings of the National Academy of Sciences*, 104(20), 2007. DOI: 10.1073/pnas.0611180104.

Steffen Frey, Filip Sadlo, and Thomas Ertl. Explorable Volumetric Depth Images from Raycasting. *2013 XXVI Conference on Graphics, Patterns and Images*, 2013. DOI: 10.1109/sibgrapi.2013.26.

Renwu Gao, Seiichi Uchida, Asif Shahab, Faisal Shafait, and Volkmar Frinken. Visual Saliency Models for Text Detection in Real World. *PLoS ONE*, 9(12), 2014. DOI: 10.1371/journal.pone.0114539.

Johannes Girstmair, Anne Zakrzewski, François Lapraz, Mette Handberg-Thorsager, Pavel Tomancak, Peter Gabriel Pitrone, Fraser Simpson, and Maximilian J. Telford. Light-sheet microscopy for everyone? Experience of building an OpenSPIM to study flatworm development. *BMC Developmental Biology*, 16(1), 2016. DOI: 10.1186/s12861-016-0122-0.

Massimo Gneo, Maurizio Schmid, Silvia Conforto, and Tommaso D'Alessio. A free geometry model-independent neural eye-gaze tracking system. *Journal of neuroengineering and rehabilitation*, 9(1), 2012. DOI: 10.1186/1743-0003-9-82.

Jacob Gold, Adam Wang, and Kyle I.S. Harrington. Feedback Control of Evolving Swarms. In *Artificial Life 14: Proceedings of the Fourteenth International Conference on the Synthesis and Simulation of Living Systems*, pages 884–891. The MIT Press, July 2014. ISBN 978-0-262-32621-6. DOI: 10.7551/978-0-262-32621-6-ch145.

Henry Gray and Warren H Lewis. *Anatomy of the Human Body*. 1878.

Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 Courses - SIGGRAPH '07*, page 9, San Diego, California, 2007. ACM Press. ISBN 978-1-4503-1823-5. DOI: 10.1145/1281500.1281665.

The HDF Group. Hierarchical Data Format, version 5, 1997.

Emilio J Gualda, Tiago Vale, Pedro Almada, José A Feijó, Gabriel G Martins, and Nuno Moreno. OpenSpinMicroscopy: An open-source integrated microscopy platform. *Nature Methods*, 10(7), 2013. DOI: 10.1038/nmeth.2508.

Leo Guignard, Ulla-Maj Fiuza, Bruno Leggio, Emmanuel Faure, Julien Laussu, Lars Hufnagel, Gregoire Malandain, Christophe Godin, and Patrick Lemaire. Contact-dependent cell communications drive morphological invariance during ascidian embryogenesis. *bioRxiv*, 2017. DOI: 10.1101/238741.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10, November 2009. ISSN 19310145. DOI: 10.1145/1656274.1656278.

Marcus D. Hanwell, Kenneth M. Martin, Aashish Chaudhary, and Lisa S. Avila. The Visualization Toolkit (VTK): Rewriting the rendering code for modern graphics cards. *SoftwareX*, 1, 2015. DOI: 10.1016/j.softx.2015.04.001.

Takahiro Harada, Jay McKee, and Jason C. Yang. Forward+: Bringing Deferred Lighting to the Next Level. *Eurographics 2012*, page 4 pages, 2012. ISSN 1017-4656. DOI: 10.2312/conf/eg2012/short/005-008.

Kyle I.S. Harrington and Louise Magbunduku. Competitive dynamics in eco-evolutionary genetically-regulated swarms. In *Proceedings of the 14th European Conference on Artificial Life ECAL 2017*, pages 190–197, Lyon, France, September 2017. MIT Press. ISBN 978-0-262-34633-7. DOI: 10.7551/ecal_a_034.

Kyle I.S. Harrington and Tim Stiles. Kephale/brevis 0.10.4. *Github*, 2017. DOI: 10.5281/zenodo.822902.

Sandra G. Hart and Lowell E. Staveland. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. *Advances in Psychology*, 52, 1988. DOI: 10.1016/s0166-4115(08)62386-9.

Benjamin Hatscher, Maria Luz, Lennart E Nacke, Norbert Elkmann, Veit Müller, and Christian Hansen. GazeTap: Towards hands-free interaction in the operating room. *the 19th ACM International Conference*, 2017. DOI: 10.1145/3136755.3136759.

Idse Heemskerk and Sebastian J Streichan. Tissue cartography: Compressing bio-image data by dimensional reduction. *Nature Methods*, 12(12), 2015.

Wolfgang Heide, Eberhard Koenig, Peter Trillenberg, Detlef Kömpf, and David S. Zee. Electrooculography: Technical standards and applications. The International Federation of Clinical Neurophysiology. *Electroencephalography and Clinical Neurophysiology. Supplement*, 52:223–240, 1999. ISSN 0424-8155.

Roger Heim, Andrew B. Cubitt, and Roger Y. Tsien. Improved green fluorescence. *Nature*, 373(6516), 1995. DOI: 10.1038/373663b0.

Teresa Hirzle, Jan Gugenheimer, Florian Geiselhart, Andreas Bulling, and Enrico Rukzio. A Design Space for Gaze Interaction on Head-mounted Displays. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*, pages 1–12, Glasgow, Scotland Uk, 2019. ACM Press. ISBN 978-1-4503-5970-2. DOI: 10.1145/3290605.3300855.

Sabrina Hoppe, Tobias Loetscher, Stephanie A Morey, and Andreas Bulling. Eye Movements During Everyday Behavior Predict Personality Traits. *Frontiers in Human Neuroscience*, 12, 2018. DOI: 10.3389/fnhum.2018.00105.

Fu-Chung Huang, Kevin Chen, and Gordon Wetzstein. The light field stereoscope: Immersive computer graphics via factored near-eye light field displays with focus cues. *ACM Transactions on Graphics (TOG)*, 34(4), 2015. DOI: 10.1145/2766922.

Jan Huisken. Optical Sectioning Deep Inside Live Embryos by Selective Plane Illumination Microscopy. *Science*, 305(5686), 2004a. DOI: 10.1126/science.1100035.

Jan Huisken. *Multi-View Microscopy and Multi-Beam Manipulation for High-Resolution Optical Imaging*. PhD thesis, Albert-Ludwigs-Universität Freiburg im Breisgau, 2004b.

Oleg Igouchkine, Yubo Zhang, and Kwan-Liu Ma. Multi-Material Volume Rendering with a Physically-Based Surface Reflection Model. *IEEE Transactions on Visualization and Computer Graphics*, 24(12), 2017. DOI: 10.1109/tvcg.2017.2784830.

Pietro Incardona, Antonio Leo, Yaroslav Zaluzhnyi, Rajesh Ramaswamy, and Ivo F. Sbalzarini. OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers. *Computer Physics Communications*, 241:155–177, August 2019. ISSN 00104655. DOI: 10.1016/j.cpc.2019.03.007.

Laurent Itti and Christof Koch. Computational modelling of visual attention. *Nature Reviews Neuroscience*, 2(3), 2001. DOI: 10.1038/35058500.

Robert J. K. Jacob. What you look at is what you get: Eye movement-based interaction techniques. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems Empowering People - CHI '90*, pages 11–18, Seattle, Washington, United States, 1990. ACM Press. ISBN 978-0-201-50932-8. DOI: 10.1145/97243.97246.

Robert J. K. Jacob. Eye movement-based human-computer interaction techniques: Toward non-command interfaces. *Advances in Human-Computer Interaction*, pages 151–190, 1993.

Wiebke Jahr, Benjamin Schmid, Michael Weber, and Jan Huisken. eduSPIM: Light Sheet Microscopy in the Museum. *PLoS ONE*, 11(8), 2016. DOI: 10.1371/journal.pone.0161402.

Changwon Jang, Kiseung Bang, Seokil Moon, Jonghyun Kim, Seungjae Lee, and Byoungho Lee. Retinal 3D. *ACM Transactions on Graphics*, 36(6), 2017. DOI: 10.1145/3130800.3130889.

Jason Jerald. *The VR Book: Human-Centered Design for Virtual Reality*. Association for Computing Machinery, October 2015. ISBN 978-1-970001-12-9. DOI: 10.1145/2792790.

Martin Jinek, Krzysztof Chylinski, Ines Fonfara, Michael Hauer, Jennifer A Doudna, and Emmanuelle Charpentier. A programmable dual-RNA-guided DNA endonuclease in adaptive bacterial immunity. *Science*, 337(6096), 2012. DOI: 10.1126/science.1225829.

Angus P.R. Johnston, James Rae, Nicholas Ariotti, Benjamin Bailey, Andrew Lilja, Robyn Webb, Charles Ferguson, Sheryl Maher, Thomas P. Davis, Richard I. Webb, John McGhee, and Robert G. Parton. Journey to the centre of the cell: Virtual reality immersion into scientific data. *Traffic*, 19(2), 2018. DOI: 10.1111/tra.12538.

James T. Kajiya. The rendering equation. *ACM SIGGRAPH Computer Graphics*, 20(4), 1986. DOI: 10.1145/15922.15902.

Moritz Kassner, William Patera, and Andreas Bulling. Pupil: An open source platform for pervasive eye tracking and mobile gaze-based interaction. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 1151–1160, Seattle, Washington, 2014. ACM Press. ISBN 978-1-4503-3047-3. DOI: 10.1145/2638728.2641695.

Robert S. Kennedy, Norman E. Lane, Kevin S. Berbaum, and Michael G. Lilienthal. Simulator Sickness Questionnaire: An Enhanced Method for Quantifying Simulator Sickness. *The International Journal of Aviation Psychology*, 3(3), 1993. DOI: 10.1207/s15327108ijap0303_3.

Alon S Keren, Shlomit Yuval-Greenberg, and Leon Y Deouell. Saccadic spike potentials in gamma-band EEG: Characterization, detection and suppression. *NeuroImage*, 49(3), 2010. DOI: 10.1016/j.neuroimage.2009.10.057.

Behrang Keshavarz and Heiko Hecht. Validating an Efficient Method to Quantify Motion Sickness. *Human Factors: The Journal of Human Factors and Ergonomics Society*, 53(4), 2011. DOI: 10.1177/0018720811403736.

Caitlin Williams Kiley and W. Martin Usrey. Cortical Processing of Visual Signals. In Donald W. Pfaff and Nora D. Volkow, editors, *Neuroscience in the 21st Century*, pages 773–792. Springer New York, New York, NY, 2016. ISBN 978-1-4939-3473-7 978-1-4939-3474-4. DOI: 10.1007/978-1-4939-3474-4_24.

Konstantin Klamka, Andreas Siegel, Stefan Vogt, Fabian Göbel, Sophie Stellmach, and Raimund Dachselt. Look & Pedal: Hands-free Navigation in Zoomable Information Spaces through Gaze-supported Foot Input. In *Proceedings of the 2015 ACM on International Conference on Multimodal Interaction - ICMI '15*, pages 123–130, Seattle, Washington, USA, 2015. ACM Press. ISBN 978-1-4503-3912-4. DOI: 10.1145/2818346.2820751.

Eliana M. Klier, Hongying Wang, and J. Douglas Crawford. The superior colliculus encodes gaze commands in retinal coordinates. *Nature Neuroscience*, 4(6), 2001. DOI: 10.1038/88450.

Aaron Knoll, R. Keith Morley, Ingo Wald, Nick Leaf, and Peter Messmer. Efficient Particle Volume Splatting in a Ray Tracer. In Eric Haines and Tomas Akenine-Möller, editors, *Ray Tracing Gems*, pages 533–541. Apress, Berkeley, CA, 2019. ISBN 978-1-4842-4426-5 978-1-4842-4427-2. DOI: 10.1007/978-1-4842-4427-2_29.

Kristin Koch, Judith McLean, Ronen Segev, Michael A Freed, Michael J Berry, Vijay Balasubramanian, and Peter Sterling. How much the eye tells the brain. *Current Biology*, 16(14), 2006.

Thomas Kosch, Mariam Hassib, Paweł W. Woźniak, Daniel Buschek, and Florian Alt. Your Eyes Tell: Leveraging Smooth Pursuit for Assessing Cognitive Workload. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*, pages 1–13, Montreal QC, Canada, 2018. ACM Press. ISBN 978-1-4503-5620-6. DOI: 10.1145/3173574.3174010.

Ruth M. Krebs, Marty G. Woldorff, Claus Tempelmann, Nils Bodammer, Toemme Noesselt, Carsten N. Boehler, Henning Scheich, Jens-Max Hopf, Emrah Duzel, Hans-Jochen Heinze, and Mircea A. Schoenfeld. High-Field fMRI Reveals Brain Activation Patterns Underlying Saccade Execution in the Human Superior Colliculus. *PLoS ONE*, 5(1), 2010. DOI: 10.1371/journal.pone.0008691.

Thomas Kroes, Frits H Post, and Charl P Botha. Exposure Render: An Interactive Photo-Realistic Volume Rendering Framework. *PLoS ONE*, 7(7), 2012. DOI: 10.1371/journal.pone.0038586.

Manish Kumar, Sandeep Kishore, Jordan Nasenbeny, David McLean, and Yevgenia Kozorovitskiy. Integrated one- and two-photon scanned oblique plane illumination (SOPi) microscopy for rapid volumetric imaging. *Optics Express*, 26(10), 2018. DOI: 10.1364/oe.26.013027.

Samuel Laine and Tero Karras. Effective Sparse Voxel Octrees - Analysis, Extensions and Implementation. *Nvidia Technical Reports*, 2010.

E. LaMar, B. Hamann, and K.I. Joy. Multiresolution techniques for interactive texture-based volume visualization. *Proceedings Visualization '99 (Cat. No.99CB37067)*, 1999. DOI: 10.1109/visual.1999.809908.

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, San Jose, CA, USA, 2004. IEEE. ISBN 978-0-7695-2102-2. DOI: 10.1109/CGO.2004.1281665.

Bettina Laugwitz and Theo Held. Construction and Evaluation of a User Experience Questionnaire. In *Proc. USAB 2008*, pages 63–76. Springer, 2008.

Kate E Laver, Belinda Lange, Stacey George, Judith E Deutsch, Gustavo Saposnik, and Maria Crotty. Virtual reality for stroke rehabilitation. *Cochrane Database of Systematic Reviews*, 11(11), 2017. DOI: 10.1002/14651858.cd008349.pub4.

Marc Levoy and Ross Whitaker. Gaze-directed volume rendering. *ACM SIGGRAPH Computer Graphics*, 24(2), 1990. DOI: 10.1145/91385.91449.

Xiang Li, Davina V Gutierrez, M Gartz Hanson, Jing Han, Melanie D Mark, Hillel Chiel, Peter Hegemann, Lynn T Landmesser, and Stefan Herlitze. Fast noninvasive activation and inhibition of neural and network activity by vertebrate rhodopsin and green algae channelrhodopsin. *Proceedings of the National Academy of Sciences*, 102(49), 2005. DOI: 10.1073/pnas.0509030102.

Yuwei Li, Felipe M. Vieceli, Walter G. Gonzalez, Ang Li, Weiyi Tang, Carlos Lois, and Marianne E. Bronner. In Vivo Quantitative Imaging Provides Insights into Trunk Neural Crest Migration. *Cell Reports*, 26(6), 2019. DOI: 10.1016/j.celrep.2019.01.039.

Zhaoyu Li, Jie Liu, Maohua Zheng, and X.Z. Shawn Xu. Encoding of Both Analog- and Digital-like Behavioral Outputs by One C. elegans Interneuron. *Cell*, 159(4), 2014. DOI: 10.1016/j.cell.2014.09.056.

Mark H. Longair, Dean A. Baker, and J. Douglas Armstrong. Simple Neurite Tracer: Open source software for reconstruction, visualization and analysis of neuronal processes. *Bioinformatics*, 27(17), 2011. DOI: 10.1093/bioinformatics/btr390.

Timothy Lottes. Fast approximate anti-aliasing (FXAA). *NVIDIA Corporation, Santa Clara, CA, USA, Feb*, 2009.

Otto Hans-Martin Lutz, Antje Christine Venjakob, and Stefan Ruff. SMOOVS: Towards calibration-free text entry by gaze using smooth pursuit movements. *Journal of Eye Movement Research*, 8(1), 2015. DOI: 10.16910/jemr.8.1.2.

Sotiris Makris, Panagiotis Karagiannis, Spyridon Koukas, and Aleksandros-Stereos Matthaiakis. Augmented reality system for operator support in human–robot collaborative assembly. *CIRP Annals - Manufacturing Technology*, 65(1), 2016. DOI: 10.1016/j.cirp.2016.04.038.

Katerina Mania, Bernard D. Adelstein, Stephen R. Ellis, and Michael I. Hill. Perceptual sensitivity to head tracking latency in virtual environments with varying degrees of scene complexity. In *Proceedings of the 1st Symposium on Applied Perception in Graphics and Visualization - APGV '04*, page 39, Los Angeles, California, 2004. ACM Press. ISBN 978-1-58113-914-3. DOI: 10.1145/1012551.1012559.

Jessica L. Maples-Keller, Brian E. Bunnell, Sae-Jin Kim, and Barbara O. Rothbaum. The Use of Virtual Reality Technology in the Treatment of Anxiety and Other Psychiatric Disorders. *Harvard Review of Psychiatry*, 25(3), 2017. DOI: 10.1097/hrp.0000000000000138.

Paul R. Martin and Samuel G. Solomon. The koniocellular whiteboard. *Journal of Comparative Neurology*, 527(3), 2019. DOI: 10.1002/cne.24426.

Marija Matejčić, Guillaume Salbreux, and Caren Norden. A non-cell-autonomous actin redistribution enables isotropic retinal growth. *PLoS Biology*, 16(8), 2018. DOI: 10.1371/journal.pbio.2006018.

Alexander Matthes, Axel Huebl, René Widera, Sebastian Grottel, Stefan Gumhold, and Michael Bussmann. In situ, steerable, hardware-independent and data-structure agnostic visualization with ISAAC. *Supercomputing Frontiers and Innovations*, 3(4), December 2016. ISSN 23138734. DOI: 10.14529/jsfi160403.

Yogesh Kumar Meena, Hubert Cecotti, KongFatt Wong-Lin, and Girijesh Prasad. A multimodal interface to resolve the Midas-Touch problem in gaze controlled wheelchair. *Conference proceedings : ... Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Annual Conference*, 2017, 2017. DOI: 10.1109/embc.2017.8036971.

Antonio Meola, Fabrizio Cutolo, Marina Carbone, Federico Cagnazzo, Mauro Ferrari, and Vincenzo Ferrari. Augmented reality in neurosurgery: A systematic review. *Neurosurgical Review*, 40(4), 2017. DOI: 10.1007/s10143-016-0732-9.

Michaela Mickoleit, Benjamin Schmid, Michael Weber, Florian O Fahrbach, Sonja Hombach, Sven Reischauer, and Jan Huisken. High-resolution reconstruction of the beating zebrafish heart. *Nature Methods*, 11(9), 2014. DOI: 10.1038/nmeth.3037.

Alexander Mietke, Frank Jülicher, and Ivo F. Sbalzarini. Self-organized shape dynamics of active surfaces. *Proceedings of the National Academy of Sciences*, 116(1), 2018. DOI: 10.1073/pnas.1810896115.

Paul Milgram, Haruo Takemura, Akira Utsumi, and Fumio Kishino. Augmented reality: A class of displays on the reality-virtuality continuum. In Hari Das, editor, *Photonics for Industrial Applications*, pages 282–292, Boston, MA, December 1995. DOI: 10.1117/12.197321.

Seyedkoosha Mirhosseini, Ievgeniia Gutenko, Sushant Ojal, Joseph Marino, and Arie Kaufman. Immersive Virtual Colonoscopy. *IEEE Transactions on Visualization and Computer Graphics*, 25(5), 2019. DOI: 10.1109/tvcg.2019.2898763.

Matthäus Mittasch, Peter Gross, Michael Nestler, Anatol W. Fritsch, Christiane Iserman, Mrityunjoy Kar, Matthias Munder, Axel Voigt, Simon Alberti, Stephan W. Grill, and Moritz Kreysing. Non-invasive perturbations of intracellular flow reveal physical principles of cell organization. *Nature Cell Biology*, 20(3), 2018. DOI: 10.1038/s41556-017-0032-9.

Esteban Gutierrez Mlot, Hamed Bahmani, Siegfried Wahl, and Enkelejda Kasneci. 3D Gaze Estimation using Eye Vergence:. In *Proceedings of the 9th International Joint Conference on Biomedical Engineering Systems and Technologies*, pages 125–131, Rome, Italy, 2016. SCITEPRESS - Science and and Technology Publications. ISBN 978-989-758-170-0. DOI: 10.5220/0005821201250131.

Pascal Monasse and Frederic Guichard. Fast computation of a contrast-invariant image representation. *IEEE Transactions on Image Processing*, 9(5), 2000.

Ken Museth. VDB. *ACM Transactions on Graphics*, 32(3), 2013. DOI: 10.1145/2487228.2487235.

Markolf H. Niemz. *Laser-Tissue Interactions: Fundamentals and Applications*. Springer International Publishing, Cham, 2019. ISBN 978-3-030-11916-4 978-3-030-11917-1. DOI: 10.1007/978-3-030-11917-1.

Jan Novák, Iliyan Georgiev, Johannes Hanika, and Wojciech Jarosz. Monte Carlo Methods for Volumetric Light Transport Simulation. *Computer Graphics Forum*, 37(2), 2018. DOI: 10.1111/cgf.13383.

Michael O'Connor, Helen M. Deeks, Edward Dawn, Oussama Metatla, Anne Roudaut, Matthew Sutton, Lisa May Thomas, Becca Rose Glowacki, Rebecca Sage, Philip Tew, Mark Wonnacott, Phil Bates, Adrian J. Mulholland, and David R. Glowacki. Sampling molecular conformations and dynamics in a multiuser virtual reality framework. *Science Advances*, 4(6), 2018. DOI: 10.1126/sciadv.aat2731.

O Olsson, M Billeter, and U Assarsson. Clustered deferred and forward shading. *Proceedings of the Fourth ACM ...*, 2012.

Felix Oswald. *A Precise and Rapid UV Laser Ablation System for Cell Biology*. PhD thesis, Technische Universität Dresden, 2010.

Charilaos Papadopoulos, Kaloian Petkov, Arie E Kaufman, and Klaus Mueller. The Reality Deck–an immersive gigapixel display. *IEEE computer graphics and applications*, 35(1), 2015. DOI: 10.1109/mcg.2014.80.

Steven G Parker, Greg Humphreys, Morgan McGuire, Martin Stich, Heiko Friedrich, David Luebke, Keith Morley, James Bigler, Jared Hoberock, David McAllister, Austin Robison, and Andreas Dietrich. GPU ray tracing. *Communications of the ACM*, 56 (5), 2013. DOI: 10.1145/2447976.2447997.

Anjul Patney, Marco Salvi, Joohwan Kim, Anton Kaplanyan, Chris Wyman, Nir Benty, David Luebke, and Aaron Lefohn. Towards foveated rendering for gaze-tracked virtual reality. *ACM Transactions on Graphics (TOG)*, 35(6), 2016. DOI: 10.1145/2980179.2980246.

Hanchuan Peng, Jianyong Tang, Hang Xiao, Alessandro Bria, Jianlong Zhou, Victoria Butler, Zhi Zhou, Paloma T Gonzalez-Bellido, Seung W Oh, Jichao Chen, Ananya Mitra, Richard W Tsien, Hongkui Zeng, Giorgio A Ascoli, Giulio Iannello, Michael Hawrylycz, Eugene Myers, and Fuhui Long. Virtual finger boosts three-dimensional imaging and microsurgery as well as terabyte volume image visualization and analysis. *Nature Communications*, 5, 2014. DOI: 10.1038/ncomms5342.

Ken Pfeuffer, Benedikt Mayer, Diako Mardanbegi, and Hans Gellersen. Gaze + pinch interaction in virtual reality. In *Proceedings of the 5th Symposium on Spatial User Interaction - SUI '17*, pages 99–108, Brighton, United Kingdom, 2017. ACM Press. ISBN 978-1-4503-5486-8. DOI: 10.1145/3131277.3132180.

Tobias Pietzsch, Stephan Preibisch, Pavel Tomančák, and Stephan Saalfeld. ImgLib2—generic image processing in Java. *Bioinformatics*, 28(22), 2012. DOI: 10.1093/bioinformatics/bts543.

Tobias Pietzsch, Stephan Saalfeld, Stephan Preibisch, and Pavel Tomancak. BigDataViewer: Visualization and processing for large image data sets. *Nature Publishing Group*, 12(6), 2015. DOI: 10.1038/nmeth.3392.

Peter G Pitrone, Johannes Schindelin, Luke Stuyvenberg, Stephan Preibisch, Michael Weber, Kevin W Eliceiri, Jan Huisken, and Pavel Tomancak. OpenSPIM: An open-access light-sheet microscopy platform. *Nature Methods*, 10(7), 2013. DOI: 10.1038/nmeth.2507.

Thammathip Piumsomboon, Gun Lee, Robert W. Lindeman, and Mark Billinghurst. Exploring natural eye-gaze-based interaction for immersive virtual reality. In *2017 IEEE Symposium on 3D User Interfaces (3DUI)*, pages 36–39, Los Angeles, CA, USA, 2017. IEEE. ISBN 978-1-5090-6716-9. DOI: 10.1109/3DUI.2017.7893315.

Daniel Pohl, Xucong Zhang, Andreas Bulling, and Oliver Grau. Concept for using eye tracking in a head-mounted display to adapt rendering to the user's current visual field. *the 22nd ACM Conference*, 2016. DOI: 10.1145/2993369.2996300.

Stephan Preibisch, Fernando Amat, Evangelia Stamataki, Mihail Sarov, Robert H Singer, Eugene Myers, and Pavel Tomancak. Efficient Bayesian-based multiview deconvolution. *Nature Methods*, 11(6), 2014. DOI: 10.1038/nmeth.2929.

Mario Prsa and Peter Thier. Cerebellum: Eye Movements. In Donald W. Pfaff and Nora D. Volkow, editors, *Neuroscience in the 21st Century*, pages 1297–1314. Springer New York, New York, NY, 2016. ISBN 978-1-4939-3473-7 978-1-4939-3474-4. DOI: 10.1007/978-1-4939-3474-4_39.

Andreas Reichenbach and Andreas Bringmann. Retina: Neuroanatomy and Physiology. In Donald W. Pfaff and Nora D. Volkow, editors, *Neuroscience in the 21st Century*, pages 673–745. Springer New York, New York, NY, 2016. ISBN 978-1-4939-3473-7 978-1-4939-3474-4. DOI: 10.1007/978-1-4939-3474-4_22.

Patrick Reipschläger, Burcu Kulahcioglu Ozkan, Aman Shankar Mathur, Stefan Gumhold, Rupak Majumdar, and Raimund Dachselt. DebugAR: Mixed Dimensional Displays for Immersive Debugging of Distributed Systems. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*, pages 1–6, Montreal QC, Canada, 2018. ACM Press. ISBN 978-1-4503-5621-3. DOI: 10.1145/3170427.3188679.

Emmanuel G Reynaud, Jan Peychl, Jan Huisken, and Pavel Tomancak. Guide to light-sheet microscopy for adventurous biologists. *Nature Methods*, 12(1), 2014. DOI: 10.1038/nmeth.3222.

Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, 21(4): 25–34, August 1987. ISSN 00978930. DOI: 10.1145/37402.37406.

David A. Robinson. A Method of Measuring Eye Movemnent Using a Scieral Search Coil in a Magnetic Field. *IEEE Transactions on Bio-medical Electronics*, 10(4), 1963. DOI: 10.1109/tbmel.1963.4322822.

Loïc A Royer, Martin Weigert, Ulrik Günther, Nicola Maghelli, Florian Jug, Ivo F Sbalzarini, and Eugene W Myers. ClearVolume: Open-source live 3D visualization for light-sheet microscopy. *Nature Methods*, 12(6), 2015. DOI: 10.1038/nmeth.3372.

Loïc A Royer, William C Lemon, Raghav K Chhetri, Yinan Wan, Michael Coleman, Eugene W Myers, and Philipp J Keller. Adaptive light-sheet microscopy for long-term, high-resolution imaging in living organisms. *Nature Biotechnology*, 34(12), 2016. DOI: 10.1038/nbt.3708.

Michele Rucci, Ramon Iovin, Martina Poletti, and Fabrizio Santini. Miniature eye movements enhance fine spatial detail. *Nature*, 447(7146), 2007. DOI: 10.1038/nature05866.

Curtis T. Rueden, Johannes Schindelin, Mark C. Hiner, Barry E. DeZonia, Alison E. Walter, Ellen T. Arena, and Kevin W. Eliceiri. ImageJ2: ImageJ for the next generation of scientific image data. *BMC Bioinformatics*, 18(1), 2017. DOI: 10.1186/s12859-017-1934-z.

Matthew R. G. Russell, Thomas R. Lerner, Jemima J. Burden, David O. Nkwe, Annegret Pelchen-Matthews, Marie-Charlotte Domart, Joanne Durgan, Anne Weston, Martin L. Jones, Christopher J. Peddie, Raffaella Carzaniga, Oliver Florey, Mark Marsh, Maximiliano G. Gutierrez, and Lucy M. Collinson. 3D correlative light and electron microscopy of cultured cells using serial blockface scanning electron microscopy. *J Cell Sci*, 130(1), 2016. DOI: 10.1242/jcs.188433.

Stephan Saalfeld, Albert Cardona, Volker Hartenstein, and Pavel Tomancak. CATMAID: Collaborative annotation toolkit for massive amounts of image data. *Bioinformatics*, 25(15), 2009. DOI: 10.1093/bioinformatics/btp266.

Paolo Sabella. A rendering algorithm for visualizing 3D scalar fields. *ACM SIGGRAPH Computer Graphics*, 22(4), 1988. DOI: 10.1145/378456.378476.

Arnab Saha, Masatoshi Nishikawa, Martin Behrndt, Carl-Philipp Heisenberg, Frank Jülicher, and Stephan W. Grill. Determining Physical Properties of the Cell Cortex. *Biophysical Journal*, 110(6), 2016. DOI: 10.1016/j.bpj.2016.02.013.

Nico Scherf and Jan Huisken. The smart and gentle microscope. *Nature Biotechnology*, 33(8), 2015. DOI: 10.1038/nbt.3310.

Johannes Schindelin, Ignacio Arganda-Carreras, Erwin Frise, Verena Kaynig, Mark Longair, Tobias Pietzsch, Stephan Preibisch, Curtis Rueden, Stephan Saalfeld, Benjamin Schmid, Jean-Yves Tinevez, Daniel James White, Volker Hartenstein, Kevin Eliceiri, Pavel Tomancak, and Albert Cardona. Fiji: An open-source platform for biological-image analysis. *Nature Methods*, 9(7), 2012. DOI: 10.1038/nmeth.2019.

Benjamin Schmid, Johannes Schindelin, Albert Cardona, Mark Longair, and Martin Heisenberg. A high-level 3D visualization API for Java and ImageJ. *BMC Bioinformatics*, 11(1), 2010. DOI: 10.1186/1471-2105-11-274.

Benjamin Schmid, Gopi Shah, Nico Scherf, Michael Weber, Konstantin Thierbach, Citlali Pérez Campos, Ingo Roeder, Pia Aanstad, and Jan Huisken. High-speed panoramic light-sheet microscopy reveals global endodermal cell dynamics. *Nature Communications*, 4(1):2207, October 2013. ISSN 2041-1723. DOI: 10.1038/ncomms3207.

Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. NIH Image to ImageJ: 25 years of image analysis. *Nature Methods*, 9(7), 2012. DOI: 10.1038/nmeth.2089.

Jeffrey S. Shell, Ted Selker, and Roel Vertegaal. Interacting with groups of computers. *Communications of the ACM*, 46(3), 2003. DOI: 10.1145/636772.636796.

Gaganpreet Singh, Sergi Bermúdez i Badia, Rodrigo Ventura, and José Luís Silva. Physiologically Attentive User Interface for Robot Teleoperation - Real Time Emotional State Estimation and Interface Modification using Physiology, Facial Expressions and Eye Movements:. In *Proceedings of the 11th International Joint Conference on Biomedical Engineering Systems and Technologies*, pages 294–302, Funchal, Madeira, Portugal, 2018. SCITEPRESS - Science and Technology Publications. ISBN 978-989-758-279-0. DOI: 10.5220/0006733002940302.

Ashutosh Singla, Stephan Fremerey, Werner Robitza, and Alexander Raake. Measuring and comparing QoE and simulator sickness of omnidirectional videos in different head mounted displays. In *2017 Ninth International Conference on Quality of Multimedia Experience (QoMEX)*, pages 1–6, May 2017. DOI: 10.1109/QoMEX.2017.7965658.

Vincent Sitzmann, Ana Serrano, Amy Pavel, Maneesh Agrawala, Diego Gutierrez, Belen Masia, and Gordon Wetzstein. Saliency in VR: How Do People Explore Virtual Environments? *IEEE Transactions on Visualization and Computer Graphics*, 24(4): 1633–1642, April 2018. ISSN 1077-2626. DOI: 10.1109/TVCG.2018.2793599.

Mel Slater and Maria V. Sanchez-Vives. Enhancing Our Lives with Immersive Virtual Reality. *Frontiers in Robotics and AI*, 3, 2016. DOI: 10.3389/frobt.2016.00074.

Robert J. Snowden, Peter Thompson, and Tom Troscianko. *Basic Vision: An Introduction to Visual Perception*. Oxford University Press, Oxford, rev. ed edition, 2011. ISBN 978-0-19-957202-1.

Jörn Starruß, Walter de Back, Lutz Brusch, and Andreas Deutsch. Morpheus: A user-friendly modeling environment for multiscale and multicellular systems biology. *Bioinformatics*, 30(9):1331–1332, May 2014. ISSN 1460-2059, 1367-4803. DOI: 10.1093/bioinformatics/btt772.

Sophie Stellmach. *Gaze-Supported Multimodal Interaction*. PhD thesis, Technische Universität Dresden, 2013.

Sophie Stellmach and Raimund Dachselt. Look & touch: Gaze-supported target acquisition. In *Proceedings of the 2012 ACM Annual Conference on Human Factors in Computing Systems - CHI '12*, page 2981, Austin, Texas, USA, 2012. ACM Press. ISBN 978-1-4503-1015-4. DOI: 10.1145/2207676.2208709.

Qi Sun, Fu-Chung Huang, Joohwan Kim, Li-Yi Wei, David Luebke, and Arie Kaufman. Perceptually-guided foveation for light field displays. *ACM Transactions on Graphics*, 36(6), 2017. DOI: 10.1145/3130800.3130807.

Xiaoxun Sun, William Yeoh, and Sven Koenig. Dynamic fringe-saving A*. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, volume 2, pages 891–898, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the May 21-23, 1963, Spring Joint Computer Conference on - AFIPS '63 (Spring)*, page 329, Detroit, Michigan, 1963. ACM Press. DOI: 10.1145/1461551.1461591.

Ivan E. Sutherland. A head-mounted three dimensional display. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I on - AFIPS '68 (Fall, Part I)*, page 757, San Francisco, California, 1968. ACM Press. DOI: 10.1145/1476589.1476686.

Larry W Swanson and Jeff W Lichtman. From Cajal to Connectome and Beyond. *Annual Review of Neuroscience*, 39(1), 2016. DOI: 10.1146/annurev-neuro-071714-033954.

Lech Swirski and Neil A. Dodgson. A fully-automatic, temporal approach to single camera, glint-free 3d eye model fitting. In *Proceedings of ECEM 2013*, Lund, Sweden, 2013.

Russell M. Taylor, Warren Robinett, Vernon L. Chi, Frederick P. Brooks, William V. Wright, R. Stanley Williams, and Erik J. Snyder. The nanomanipulator: A virtual-reality interface for a scanning tunneling microscope. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '93*, pages 127–134. ACM Press, 1993. ISBN 978-0-89791-601-1. DOI: 10.1145/166117.166133.

Russell M. Taylor, Thomas C. Hudson, Adam Seeger, Hans Weber, Jeffrey Juliano, and Aron T. Helser. VRPN: A device-independent, network-transparent VR peripheral system. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology - VRST '01*, page 55, Baniff, Alberta, Canada, 2001. ACM Press. ISBN 978-1-58113-427-8. DOI: 10.1145/505008.505019.

Lucas Theis, Iryna Korshunova, Alykhan Tejani, and Ferenc Huszár. Faster gaze prediction with dense networks and Fisher pruning. *arXiv:1801.05787 [cs, stat]*, January 2018.

Alex M. Thomson. Neocortical layer 6, a review. *Frontiers in Neuroanatomy*, 2010. ISSN 16625129. DOI: 10.3389/fnana.2010.00013.

Jean-Yves Tinevez, Nick Perry, Johannes Schindelin, Genevieve M. Hoopes, Gregory D. Reynolds, Emmanuel Laplantine, Sebastian Y. Bednarek, Spencer L. Shorte, and Kevin W. Eliceiri. TrackMate: An open and extensible platform for single-particle tracking. *Methods*, 115(IEEE Signal Proc. Mag. 23 3 2006), 2017. DOI: 10.1016/j.ymeth.2016.09.016.

Kazuhiko Ukai and Peter A. Howarth. Visual fatigue caused by viewing stereoscopic motion images: Background, theories, and observations. *Displays*, 29(2), 2008. DOI: 10.1016/j.displa.2007.09.004.

Vladimír Ulman, Martin Maška, Klas E G Magnusson, Olaf Ronneberger, Carsten Haubold, Nathalie Harder, Pavel Matula, Petr Matula, David Svoboda, Miroslav Radojevic, Ihor Smal, Karl Rohr, Joakim Jaldén, Helen M Blau, Oleh Dzyubachyk, Boudewijn Lelieveldt, Pengdong Xiao, Yuexiang Li, Siu-Yeung Cho, Alexandre C Dufour, Jean-Christophe Olivo-Marin, Constantino C Reyes-Aldasoro, Jose A Solis-Lemus, Robert Bensch, Thomas Brox, Johannes Stegmaier, Ralf Mikut, Steffen Wolf, Fred A Hamprecht, Tiago Esteves, Pedro Quelhas, Ömer Demirel, Lars Malmström, Florian Jug, Pavel Tomancak, Erik Meijering, Arrate Muñoz-Barrutia, Michal Kozubek, and Carlos Ortiz-de-Solorzano. An objective comparison of cell-tracking algorithms. *Nature Methods*, 14(12):1141–1152, December 2017. ISSN 1548-7091, 1548-7105. DOI: 10.1038/nmeth.4473.

Will Usher, Pavol Klacansky, Frederick Federer, Peer-Timo Bremer, Aaron Knoll, Jeff Yarch, Alessandra Angelucci, and Valerio Pascucci. A Virtual Reality Visualization Tool for Neuron Tracing. *IEEE Transactions on Visualization and Computer Graphics*, 24(1), 2017. DOI: 10.1109/tvcg.2017.2744079.

Mélodie Vidal, Andreas Bulling, and Hans Gellersen. Pursuits: Spontaneous interaction with displays based on smooth pursuit eye movement and moving targets. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing - UbiComp '13*, page 439, Zurich, Switzerland, 2013. ACM Press. ISBN 978-1-4503-1770-2. DOI: 10.1145/2493432.2493477.

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. Embree. *ACM Transactions on Graphics*, 33(4), 2014. DOI: 10.1145/2601097.2601199.

Ingo Wald, GP Johnson, Jefferson Amstutz, Carson Brownlee, Aaron M. Knoll, Jim Jeffers, Johannes Günther, and Paul Arthur Navratil. OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):931–940, January 2017. ISSN 1077-2626. DOI: 10.1109/TVCG.2016.2599041.

Quan Wang, Laura Boccanfuso, Beibin Li, Amy Yeo-jin Ahn, Claire E Foster, Margaret P Orr, Brian Scassellati, and Frederick Shic. Thermographic eye tracking. *the Ninth Biennial ACM Symposium*, 2016. DOI: 10.1145/2857491.2857543.

Yao Wang, Ouseb Lee, and Anthony Vetro. Use of two-dimensional deformable mesh structures for video coding. II. The analysis problem and a region-based coder employing an active mesh representation. *IEEE Transactions on circuits and systems for video technology*, 6(6), 1996.

Michael Weber and Jan Huisken. Omnidirectional microscopy. *Nature Methods*, 9(7), 2012. DOI: 10.1038/nmeth.2022.

Gordon Wetzstein, Ivo Ihrke, and Wolfgang Heidrich. On Plenoptic Multiplexing and Reconstruction. *International Journal of Computer Vision*, 101(2), 2013. DOI: 10.1007/s11263-012-0585-9.

David Matthew Whittinghill, Bradley Ziegler, Tristan Case, and James Moore. Nasum virtualis: A simple technique for reducing simulator sickness. *Games Developers Conference (GDC)*, 2015.

Carsten Wolff, Jean-Yves Tinevez, Tobias Pietzsch, Evangelia Stamataki, Benjamin Harich, Léo Guignard, Stephan Preibisch, Spencer Shorte, Philipp J Keller, Pavel Tomancak, and Anastasios Pavlopoulos. Multi-view light-sheet imaging and tracking with the MaMuT software reveals the cell lineage of a direct developing arthropod limb. *eLife*, 7, 2018. DOI: 10.7554/elife.34410.

Paul Woodward, T Ruwart, D Porter, K Edgar, S Anderson, M Palmer, R Cattelan, T Jacobson, J Stromberg, and T Varghese. Powerwall. *University of Minnesota*, 2001.

Huiyue Wu and Jianmin Wang. A visual attention-based method to address the midas touch problem existing in gesture-based interaction. *The Visual Computer*, 32(1), 2015. DOI: 10.1007/s00371-014-1060-0.

Yongyi Yang, Miles N. Wernick, and Jovan G. Brankov. A fast approach for accurate content-adaptive mesh generation. *IEEE Transactions on Image Processing*, 12(8):866–881, August 2003. ISSN 1057-7149. DOI: 10.1109/TIP.2003.812757.

Arash Yazdanbakhsh and Margaret S Livingstone. End stopping in V1 is sensitive to contrast. *Nature Neuroscience*, 9(5), 2006. DOI: 10.1038/nn1693.

Xinyong Zhang, Xiangshi Ren, and Hongbin Zha. Improving eye cursor's stability for eye pointing tasks. In *Proceeding of the Twenty-Sixth Annual CHI Conference on Human Factors in Computing Systems - CHI '08*, page 525, Florence, Italy, 2008. ACM Press. ISBN 978-1-60558-011-1. DOI: 10.1145/1357054.1357139.

Thomas G. Zimmerman, Jaron Lanier, Chuck Blanchard, Steve Bryson, and Young Harvill. A hand gesture interface device. *ACM SIGCHI Bulletin*, 18(4), 1987. DOI: 10.1145/29933.275628.

Jan Zimmermann, Yuriria Vazquez, Paul W Glimcher, Bijan Pesaran, and Kenway Louie. Oculomatic: High speed, reliable, and accurate open-source eye tracking for humans and non-human primates. *Journal of Neuroscience Methods*, 270, 2016. DOI: 10.1016/j.jneumeth.2016.06.016.

Selbstständigkeitserklärung

1. Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.
2. Bei der Auswahl und Auswertung des Materials sowie bei der Herstellung des Manuskripts habe ich Unterstützungsleistungen von folgenden Personen erhalten: (keine)
3. Weitere Personen waren an der geistigen Herstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.
4. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und ist auch noch nicht veröffentlicht worden.
5. Ich bestätige, dass ich die geltende Promotionsordnung der Fakultät Informatik der Technischen Universität Dresden anerkenne.

Ort, Datum

Unterschrift des Doktoranden