VAST *Lite* 1.5.0
Volume Annotation and Segmentation Tool

User Manual

Daniel R. Berger

May 8th, 2024

Contents

1 Introduction **1**

2 Getting Started **5**
2.1 System Requirements . . . 5
2.2 Program Setup . . . 5
2.2.1 Try It Out! . . . 6
2.2.2 Preferences . . . 6
2.3 Additional Files Included With VAST . . . 9

3 Working with VAST **11**
3.1 Viewing and Navigating an Image Stack . . . 11
3.1.1 Multi-panel view . . . 12
3.1.2 The side scrollbar . . . 12
3.1.3 The RAM (main memory) usage indicator . . . 12
3.1.4 Getting and setting coordinates . . . 13
3.1.5 Layers . . . 13
3.1.6 Control Buttons . . . 15
3.2 Image Stack Importing . . . 16
3.2.1 Importing image stacks: Pattern-based names . . . 17
3.2.2 Lossless and lossy compression . . . 18
3.2.3 Importing 3D volume files . . . 20
3.2.4 Image scale and description . . . 20
3.2.5 Preparing .VSVI files for large image stacks . . . 20
3.2.6 Remote image data (VSVR) . . . 27
3.3 Exporting Image Stacks . . . 28

4 Segmentation Layers **31**
4.1 Painting . . . 31
4.1.1 Multi-scale painting . . . 32
4.1.2 Automatic Z-filling . . . 33
4.1.3 Masking . . . 34
4.1.4 Semi-automatic segmentation from boundary maps . . . 34
4.1.5 Semi-automatic segmentation from an imperfect automatic segmentation . . 35
4.2 Filling . . . 35
4.2.1 Filling segments in the same layer . . . 36
4.2.2 Filling segments from a source layer to a target layer . . . 36
4.3 Splitting . . . 37
4.4 Segments . . . 38
4.4.1 Picking segments . . . 38
4.4.2 The segment hierarchy . . . 38

4.4.3 Re-ordering and moving segments in the tree . . . 39
4.4.4 Collapsing and expanding tree branches . . . 39
4.4.5 Adding new segments . . . 39
4.4.6 Generating segments from an automatic segmentation and skeleton . . . 40
4.4.7 Using anchor points and other coordinates . . . 40
4.4.8 Helper functions for arranging segments . . . 40
4.4.9 Select recently selected segments . . . 41
4.4.10 Global operations: Deleting and welding segment subtrees . . . 41
4.4.11 Global operations: Reevaluating bounding boxes . . . 41
4.4.12 Global operations: Segmentation cleaning . . . 42
4.4.13 Segment tags . . . 42
4.4.14 Editing the color of a segment . . . 42
4.4.15 Exporting volume measurements . . . 43
4.4.16 Exporting segmentation metadata . . . 43
4.4.17 Loading segmentation metadata . . . 43
4.4.18 Segment information . . . 43
4.4.19 Searching for a segment with a given name or ID . . . 44
4.4.20 The 'Collect' tool . . . 44
4.5 Saving Segmentations . . . 44
4.6 Segmentation Merging . . . 45
4.7 Importing Segmentations From Image Stacks . . . 45

5 Annotation Layers 49
5.1 Creating an Annotation Layer . . . 49
5.2 The 'Annotation Objects' Tool Window . . . 49
5.3 Annotation Object Flags . . . 52
5.4 Skeletons . . . 52
5.4.1 Editing skeletons . . . 52
5.4.2 The node context menu . . . 53
5.4.3 Quick navigation using skeletons . . . 54
5.4.4 Edge types . . . 55
5.5 The Node Tool . . . 55
5.6 The Annotation Schematics Window . . . 56
5.7 Exporting and Importing Annotation Data . . . 56
5.8 Importing Skeletons from Files . . . 56

6 Tool Layers 57
6.1 Chunked Region Collector . . . 57

7 The 3D Viewer 61
7.1 Using the 3D viewer to set the 2D view location . . . 62
7.2 Other 3D Viewer Modes . . . 63
7.2.1 Screenshot Rendering, Segmentations Only . . . 63
7.2.2 'Raymarcher 2' and 'Raymarcher 2 Tool' . . . 63

8 The VastTools Matlab Toolbox 65
8.1 Getting started with the VastTools Matlab Toolbox . . . 65
8.2 Exporting 3D Models . . . 66
8.3 Export Particle Clouds (3D Object Instancing) . . . 69
8.4 Export 3D Boxes, Planes and Scale Bars . . . 70
8.5 Exporting Skeleton 3D Models . . . 71
8.6 Exporting Projection Images . . . 71

8.7 Measuring . . . 74
8.7.1 Measure Segment Volumes . . . 74
8.7.2 Measure Masked Regions . . . 74
8.7.3 Measure Skeleton Lengths . . . 75
8.7.4 Measure Segment Surface Area . . . 75
8.7.5 Euclidian Distance Measurement Tool . . . 75
8.8 Simple Navigator Images . . . 75
8.9 Target Lists . . . 76
8.10 Extras . . . 76
8.10.1 Export Neuroglancer Link 76

A API Function Reference 77
A.1 General Functions . . . 77
A.1.1 res = connect(host, port, timeout) . . . 77
A.1.2 res = disconnect() . . . 78
A.1.3 errornumber = getlasterror() . . . 78
A.1.4 [apiversion, res] = getapiversion() . . . 78
A.1.5 [version, subversion, res] = getcontrolclassversion() . . . 78
A.1.6 [info, res] = getinfo() . . . 78
A.1.7 [info, res] = gethardwareinfo() . . . 79
A.2 Layer Functions . . . 79
A.2.1 [nroflayers, res] = getnroflayers() . . . 79
A.2.2 [linfo, res] = getlayerinfo(layernr) . . . 80
A.2.3 res = setlayerinfo(layernr, linfo) . . . 80
A.2.4 [mipscalematrix, res] = getmipmapscalefactors(layernr) . . . 80
A.2.5 res = setselectedlayernr(layernr) . . . 81
A.2.6 [selectedlayernr, selectedemlayernr, selectedsegmentlayernr, res] = getselectedlayernr() . . . 81
A.2.7 [selectedlayernr, selectedemlayernr, selectedannolayernr, selectedsegmentlayernr, selectedtoollayernr, res] = getselectedlayernr2() . . . 81
A.2.8 [id, res] = addnewlayer(laytype, name, refid[opt]) . . . 81
A.2.9 [id, res] = loadlayer(filename, refid[opt]) . . . 81
A.2.10 res = savelayer(layernr, targetfilename, forceit[opt], subformat[opt]) . . . 82
A.2.11 res = removelayer(layernr, forceit) . . . 82
A.2.12 [newlayernr, res] = movelayer(movedlayernr, afterlayernr) . . . 82
A.2.13 [isenabled, res] = getapilayersenabled() . . . 82
A.2.14 res = setapilayersenabled(enabledflag) . . . 82
A.2.15 [selectedlayernr, selectedemlayernr, selectedannolayernr, selectedsegmentlayernr, selectedtoollayernr, res] = getselectedapilayernr() . . . 83
A.2.16 res = setselectedapilayernr(layernr) . . . 83
A.3 Annotation Layer Functions . . . 83
A.3.1 [nrofobjects, firstannoobjectnr, res] = getannolayernrofobjects() . . . 83
A.3.2 [annolayerobjectdata, res] = getannolayerobjectdata() . . . 83
A.3.3 [aoname, res] = getannolayerobjectnames() . . . 84
A.3.4 [id, res] = addnewannoobject(refid, nextorchild, type, name) . . . 84
A.3.5 res = moveannoobject(id, refid, nextorchild) . . . 84
A.3.6 res = removeannoobject(id) . . . 84
A.3.7 res = setselectedannoobjectnr(id) . . . 84
A.3.8 [id, res] = getselectedannoobjectnr() . . . 84
A.3.9 [aonodedata, res] = getaonodedata() . . . 84
A.3.10 [nodenumbers, nodelabels, res] = getaonodelabels() . . . 85

A.3.11 res = setselectedaonodebydfsnr(nr) . . . 85
A.3.12 res = setselectedaonodebycoords(x, y, z) . . . 85
A.3.13 [nr, res] = getselectedaonodenr() . . . 85
A.3.14 res = addaonode(x, y, z) . . . 85
A.3.15 res = moveselectedaonode(x, y, z) . . . 85
A.3.16 res = removeselectedaonode() . . . 85
A.3.17 res = swapselectedaonodechildren() . . . 86
A.3.18 res = makeselectedaonoderoot() . . . 86
A.3.19 [branchid, res] = splitselectedskeleton(newrootdfsnr, newname) . . . 86
A.3.20 res = weldskeletons(annoobjectnr1, nodedfsnr1, annoobjectnr2, nodedfsnr2) . 86
A.3.21 [aodata, res] = getannoobject(id) . . . 86
A.3.22 [id, res] = setannoobject(id, aodata) . . . 87
A.3.23 [id, res] = addannoobject(refid, nextorchild, aodata) . . . 88
A.3.24 [annoobjectid, nodedfsnr, distance, res]=getclosestaonodebycoords(x, y, z, maxdistance) . . . 88
A.3.25 [nodedata, res] = getaonodeparams(annoobjectid, nodedfsnr) . . . 88
A.3.26 res = setaonodeparams(annoobjectid, nodedfsnr, nodedata) . . . 89
A.4 Segmentation Layer Functions . . . 89
A.4.1 [nr, res] = getnumberofsegments() . . . 89
A.4.2 [data, res] = getsegmentdata(id) . . . 90
A.4.3 [name, res] = getsegmentname(id) . . . 90
A.4.4 res = setanchorpoint(id, x, y, z) . . . 90
A.4.5 res = setsegmentname(id, name) . . . 90
A.4.6 res = setsegmentcolor8(id, r1, g1, b1, p1, r2, g2, b2, p2) . . . 90
A.4.7 res = setsegmentcolor32(id, col1, col2) . . . 91
A.4.8 [segdata, res] = getallsegmentdata() . . . 91
A.4.9 [segdatamatrix, res] = getallsegmentdatamatrix() . . . 91
A.4.10 [segname, res] = getallsegmentnames() . . . 91
A.4.11 res = setselectedsegmentnr(segmentnr) . . . 91
A.4.12 [selectedsegmentnr, res] = getselectedsegmentnr() . . . 91
A.4.13 res = setsegmentbbox(id, minx, maxx, miny, maxy, minz, maxz) . . . 91
A.4.14 [firstsegmentnr, res] = getfirstsegmentnr() . . . 92
A.4.15 [id, res] = addsegment(refid, nextorchild, name) . . . 92
A.4.16 res = movesegment(id, refid, nextorchild) . . . 92
A.5 Tool Layer Functions . . . 92
A.5.1 res = settoolparameters(toollayernr, toolnodenr, params) . . . 92
A.6 2D View Functions . . . 92
A.6.1 [x, y, z, res] = getviewcoordinates() . . . 92
A.6.2 [zoom, res] = getviewzoom() . . . 93
A.6.3 res = setviewcoordinates(x, y, z) . . . 93
A.6.4 res = setviewzoom(zoom) . . . 93
A.6.5 res = set2dvieworientation(orientation[opt], viewport[opt]) . . . 93
A.6.6 res = refreshlayerregion(layernr, minx,maxx,miny,maxy,minz,maxz) . . . 93
A.7 Voxel Data Transfer Functions . . . 93
A.7.1 [segimage, res] = getsegimageraw(miplevel, minx, maxx, miny, maxy, minz, maxz, flipflag, immediateflag, requestloadflag) . . . 93
A.7.2 [segimageRLE, res] = getsegimageRLE(miplevel, minx, maxx, miny, maxy, minz, maxz, surfonlyflag, immediateflag, requestloadflag) . . . 94
A.7.3 [segimage, res] = getsegimageRLEdecoded(miplevel, minx, maxx, miny, maxy, minz, maxz, surfonlyflag, flipflag, immediateflag, requestloadflag) . . . 94

A.7.4 [values, numbers, res] = getRLEcountunique(miplevel, minx, maxx, miny, maxy, minz, maxz, surfonlyflag, immediateflag, requestloadflag) 94
A.7.5 [segimage, values, numbers, res] = getsegimageRLEdecodedcountunique(miplevel, minx, maxx, miny, maxy, minz, maxz, surfonlyflag, flipflag, immediateflag, requestloadflag) 95
A.7.6 [segimage, values, numbers, bboxes, res] = getsegimageRLEdecodedbboxes(miplevel, minx, maxx, miny, maxy, minz, maxz, surfonlyflag, flipflag, immediateflag, requestloadflag) 95
A.7.7 res = setsegtranslation(sourcearray, targetarray) 95
A.7.8 [emimage, res] = getemimageraw(layernr, miplevel, minx, maxx, miny, maxy, minz, maxz, immediateflag, requestloadflag) 95
A.7.9 [emimage,res] = getemimage(layernr, miplevel, minx, maxx, miny, maxy, minz, maxz, immediateflag, requestloadflag) 96
A.7.10 [screenshotimage,res] = getscreenshotimageraw(miplevel, minx, maxx, miny, maxy, minz, maxz, collapseseg) 96
A.7.11 [screenshotimage,res] = getscreenshotimage(miplevel, minx, maxx, miny, maxy, minz, maxz, collapseseg) 96
A.7.12 res = orderscreenshotimage(miplevel, minx, maxx, miny, maxy, minz, maxz, collapseseg) 96
A.7.13 [screenshotimage,res] = pickupscreenshotimage(miplevel, minx, maxx, miny, maxy, minz, maxz, collapseseg) 96
A.7.14 res = setsegimageraw(miplevel, minx, maxx, miny, maxy, minz, maxz, segimage) 96
A.7.15 res = setsegimageRLE(miplevel, minx, maxx, miny, maxy, minz, maxz, segimage) 97
A.7.16 [pixelvalue, res]=getpixelvaluefromfullrescoords(layernr, miplevel, x, y, z) . . . 97
A.8 VAST UI Functions 97
A.8.1 [props, res] = getdrawingproperties() 97
A.8.2 res = setdrawingproperties(props) 98
A.8.3 [props, res] = getfillingproperties() 99
A.8.4 res = setfillingproperties(props) 99
A.8.5 [state, res] = getcurrentuistate() 100
A.8.6 res = setuimode(uimode) 101
A.8.7 res = showwindow(windownr, onoff[opt], xpos[opt], ypos[opt], width[opt], height[opt]) 101
A.8.8 [isenabled, res] = geterrorpopupsenabled(errornr) 102
A.8.9 res = seterrorpopupsenabled(errornr, enabledflag) 102
A.8.10 [isenabled, res] = getpopupsenabled(typenr) 102
A.8.11 res = setpopupsenabled(typenr, enabledflag) 102
A.9 Execute VAST Functions 103
A.9.1 res = executefill(sourcelayernr, targetlayernr, x, y, z, mip) 103
A.9.2 res = executecanvaspaintstroke(coords) 103
A.9.3 res = executestartautoskeletonization(toollayernr, mip, nodedistance_mu, regionpadding_mu) 103
A.9.4 res = executestopautoskeletonization() 103
A.9.5 [state, res] = executegetautoskeletonizationstate() 104
A.10 Helper Functions 104
A.10.1 ids=getidsfromexactname(name) 104
A.10.2 ids=getimmediatechildids(parentidlist) 104
A.10.3 ids=getchildtreeids(parentidlist) 104

B FAQ and Trouble Shooting 105
B.1 Frequently Asked Questions 105
B.2 Typical Use Cases 106
B.3 Some Performance Tips 108
B.4 Setting up VAST with a Wacom tablet 109
B.5 Keyboard Shortcuts in VAST 110
B.6 Terms of Usage and Privacy Statement 111

C Technical Information 113
C.1 Size limitations 113
C.2 Supported file formats for importing / exporting 113

Chapter 1

Introduction

VAST is a utility program for manual annotation and segmentation of large volumetric image (voxel) data sets. It enables users to work with image stacks in the petabyte range at interactive speeds, to explore them visually and to label structures of interest by voxel painting and skeletonization. VAST implements multi-scale painting to limit the amount of required memory while enabling interactive painting speed with any tooltip size. Manual annotations generated with VAST can then be used as training data for machine learning algorithms to perform automatic segmentation. Automatic segmentation algorithms typically operate on voxel data, which is VAST's native data format. VAST can also be used for proof-reading and correcting results of automatic segmentation algorithms. VAST supports loading several image stacks and segmentation stacks at once (in multiple layers) and provides innovative functions to combine data from several layers. VAST also supports skeleton annotation which can be used for abstracted object representation and morphological measurement.

VAST is also very portable and light-weight. It consists of a single Windows executable file which is independent of third-party libraries. It does not require to be installed and can be easily copied and for example run from a memory stick.

Using VAST, a very experienced user can produce a dense segmentation of well-stained and well-aligned cortex neuropil data (6x6x30 nm voxel size) at a speed of about 4 cubic microns per hour. If cross-sections are labeled with a color dot rather than accurately outlining the cross-sections, the labeling speed can be increased by a factor of 3 at the expense of accuracy (12 cubic microns per hour). If the filling function is used with a reasonable automatic segmentation, tracing processes can reach speeds exceeding one millimeter per hour and is essentially limited only by the speed of data streaming and human perception. Similar results can be obtained when using skeletons.

Many tools for labeling voxel data exist, but such tools usually have drawbacks when it comes to painting in large data sets. Most tools do not support voxel painting natively but are limited to skeletonizing or storing object outlines as vector graphics (for example splines). Also many of them require the data to be loaded in RAM completely. This is not possible for the data sets in the terabyte to petabyte range which are currently produced by serial-section electron microscopy of biological samples. Many of these tools are also developed as cross-platform applications, so that they can be run on Windows, Mac and Linux systems. This usually means that a non-native GUI system has to be used (e.g. Qt) which makes the program more bulky and more difficult to install and maintain, or alternatively, they have to run in a limited browser environment (often with extensive server-side support). As a stand-alone Windows program, VAST can take advantage of using native Windows GUI and graphics functions.

VAST is currently being used in several scientific projects to label cells in electron-microscopic and light microscopic image stacks. It was also used for the published studies [3–6, 8–12] among many others. VAST itself was recently also published [7]; please use this reference for citations.

The key concepts of VAST are:

- Image stacks can be imported into VASTs .vsv file format, where they are stored in a diced format. This, together with pre-computation of mip maps, allows for fast panning and zooming through the data when it is opened in VAST.
- .vsv files support lossless and lossy compression to reduce the resulting file size
- Imported image data, segmentations and annotations are stored in single files, which makes it easy to archive and share them
- VAST can also load image data directly from image tiles if mip-maps have been precomputed, using a .vsvi descriptor file, and from web servers over HTTP (openconnecto.me, neurodata.io, butterfly, Google Brainmaps, Neuroglancer precomputed and precomputed sharded formats, and simple image-tiles-on-open-web-server formats including Google cloud buckets and Amazon web services). This is particularly useful for very large volumes
- Dynamic multi-threaded caching in RAM with pre-loading for low-latency display update, with optional local hard drive caching
- Layering: Several image, segmentation, annotation and tool layers can be opened and displayed together with different blending and tinting options
- Multi-scale painting (in VAST, painting always happens at the currently displayed resolution (mip level))
- Automatic convex Z-filling during painting to speed up coarse labeling
- Automatic 2D-filling of closed contours
- 3D flood filling tool
- Painting and filling can be constrained by masks defined by a separate image or segmentation layer
- Label hierarchies allow for fast and reversible grouping of labeled segments in a tree structure
- Segments have anchor points to quickly find them in the volume
- Annotation layers hold editable skeletons with many features like node labels and edge types
- Tool layers support on-the-fly agglomerating automatically generated segmentations by using skeletons
- Exporting of segmentations, EM stacks and mixed image stacks ('screenshots') in multiple formats
- Importing of segmentations from image stacks
- File-modification free editing. Image files are not changed (except if you explicitly update information in them). Segmentation files are only changed it you save the segmentation back to the same file.
- TCP/IP-based API through which external programs can directly communicate with VAST, and supporting Matlab script 'VastTools' which provides supplementary functions like measuring labeled segments and exporting of 3D surface models.

Current limitations:

- Currently only 16-bit segmentations are supported
- There is no 'Undo' function
- Exporting of 3D models of segmented objects is currently only supported externally (using the VastTools Matlab script)
- Only rudimentary 3D viewer
- Not ideal for multi-user projects
- Not an open-source project, but the VAST *Lite* executable can be copied and used freely (see section B.6 for details)

Please note that VAST is under constant development, and is subject to change as new features are added and bugs are fixed. For bug reports or helpful suggestions, please contact me at: `danielberger@fas.harvard.edu`.

Chapter 2

Getting Started

2.1 System Requirements

VAST currently runs on 64-bit Windows computers that support DirectX 11. These are Windows 7 (SP1) - 11. Windows XP and older versions will not work. Currently only 64 bit versions of these operating systems are supported because 32 bit programs are limited in the amount of RAM they can handle (4 GB max. theoretically, less realistically). The computer also has to have a DirectX 11 compatible graphics card, which should however be available in all modern PCs.

Recommended system configuration:

- Windows PC with 64 bit Windows 7 (SP1) or later
- 16 GB of RAM (the more the better)
- DirectX 11 compatible graphics card
- 2 TB of disk space (depends on the size of the data you work with)
- Wacom Cintiq or other pen touch screen with configurable two-button pen, if workflow includes manual voxel segmentation

Minimal system configuration:

- Windows PC with 64 bit Windows 7 (SP1)
- 2 GB of RAM (at least 4 GB recommended)
- DirectX 11 compatible on-board graphics card
- Standard screen and mouse

2.2 Program Setup

To use VAST, simply copy the executable program into a folder where you have read/write access, and set up links on your desktop, start menu and/or taskbar if desired. It is important that VAST has read and write access to the folder where the executable is because it will write a configuration file (`vast_preferences.dat`) into the same folder to store your settings. Then start the executable.

2.2.1 Try It Out!

The quickest way to try out VAST is to use an online data set. Several online data sets are included in the .ZIP package of supplementary files as `.vsvi` files. You can save this package from the executable by clicking 'Yes' in the 'First Start' pop-up window when you start VAST for the first time, or by choosing 'Save Documentation .ZIP To Disk...' from the 'Info' menu.

Unzip the package (or drag&drop the contents to a folder outside the .ZIP using Windows Explorer for example).[1] Then, in VAST, go to 'Open EM...' and select one of the `.vsvi` files in the `VAST_package/Online Datasets/` folder, for example '`H01_EM_8nm.vsvi`'.[2] This will load images of a big EM stack stored at Google. Your computer has to have internet access for this to work.[3]

Click and drag the EM slice to pan. Use the mouse wheel or N and M keys to zoom. Use UP/DOWN arrow keys or A/Z to scroll through the stack.

Click on the little pencil icon in the toolbar to switch to 'Paint' mode. Choose 'Yes' in the popup window. Click and drag over the image to paint. You can select different paint colors in the 'Segment Colors' tool window. To erase, hold down the 'Delete' key while painting (or click and hold the right and left mouse buttons together). Select 'Keyboard Shortcuts' from the 'Window' menu for a list of available keyboard functions.

2.2.2 Preferences

VAST will set up the preferences for you when you run it for the first time on a computer (whenever it cannot find the preferences file `vast_preferences.dat`). To edit these preferences, click '`File / Preferences ...`' in the main menu of VAST which will open the Preferences dialog window. You should at least check once whether the folder in which VAST puts its temporary disk cache is on a hard drive with lots of free space. Depending on what you do, temporary disk cache files can get similarly large as the segmentation files you are working with, in particular if you use global segmentation editing functions like merging, segment deleting, or segment welding. Each VAST instance will write a cache file (`vastsegcache*.vss`) to this folder and delete it when VAST is closed. In case VAST crashed (which should not happen) or power was lost while VAST was running, a cache file may be left there which takes up disk space, so you may want to check for and delete old cache files once in a while if VAST was terminated abnormally.

Memory and Cache

On the left side of the Preferences dialog you can set how much cache memory VAST will use maximally for voxel images and for segmentations. VAST chooses initial values which are reasonable for your system. The rule of thumb is: If you can afford it, leave 1-2 GB for the system, and split the rest 1/3 each for image cache, segmentation cache, and general usage of VAST (don't assign). On a system with 8 GB RAM, this means to give 2000 MB to the image RAM cache and 2000 MB to the segmentation RAM cache. If you are only viewing images and not using segmentations, you can increase the size of the image cache and reduce the size of the segmentation cache accordingly. If you plan to use other programs at the same time or run two instances of VAST, please reduce these values as needed. VAST will not immediately use all of the allotted memory, but it will stop reserving new memory for cache blocks and re-use old blocks when it reaches the limit.

[1]Extracting the package contents to actual files is important! vasttools.m for example will not work properly when started from within the compressed .zip file.

[2]Alternatively, you can simply drag and drop the .vsvi file from the Windows Explorer window onto the main window of VAST to open it.

[3]Remote EM images are loaded from a server either from files or using a cutout service over HTTP. The .vsvi or .vsvr files are just text files defining the web address of the dataset and its properties.

In general, do not allow VAST to allocate more memory than the system has. This can result in severe performance issues. There is a memory usage indicator in the upper right corner of the VAST window which shows you how much memory is currently used. The blue frame indicates the maximum amount of RAM which VAST uses for image and segmentation caching. If the memory indicator becomes red and your system slows down, try to REDUCE the cache limits to allow Windows and VAST to use more RAM for other data.

Some of the segmentation cache is used for holding the currently displayed part of the segmentation in memory. When you exit the preferences dialog, VAST will tell you how much of the segmentation cache it needs for the current display settings and whether the cache size is sufficient.

'`Disk cache directory`': Here you can specify the folder where VAST stores its temporary disk cache. Click the '`[ ... ]`' button to browse. Set this to a folder where you have lots of free space (more than the size of the largest segmentation file you will be working with, since for certain functions VAST has to duplicate the segmentation data).

Painting and Annotating

'`Tablet mode (pen paints, finger moves)`': On some pen-enabled tablet computers VAST can distinguish finger and pen input. If this mode is enabled, the pen should paint and the finger should move the view when in Paint Mode. This may not work on some newer systems.

'`Show mouse pointer during painting`': Normally the mouse pointer is hidden during a paint stroke; enable this option if you want to keep it visible.

'`Choose new colors from`': Allows you to select which color space to use for assigning random colors to new segments and skeletons. Segment and skeleton colors can of course also be changed manually (see section 4.4.14).

'`Show nearby edges, and nodes up to distance`': If this value is 0, for skeletons in annotation layers only the nodes and edges in the current 2D section will be shown. If the value is > 0, nodes in adjacent sections will be shown as well, up to the specified distance (fading out and using the scale and opacity factors below). If the value is > 0, edges intersecting the current section will also be shown in full length, but with scale and opacity factors specified below. This makes edges look like sliders.

'`Scale factor (0..1)`': Scale factor for 2D display of skeleton nodes and edges in adjacent sections

'`Opacity factor (0..1)`': Opacity factor for 2D display of skeleton nodes and edges in adjacent sections

'`Label text backdrop opacity factor (0..1)`': Opacity factor for 2D display of skeleton node labels in adjacent sections

API Settings

'`Enable API remote connections at startup`': It can be useful to enable this option if you often use the VAST API to link to external programs. The API remote connection functionality can also be enabled and disabled in the '`Remote Control API Server`' dialog under '`Window`' in the VAST main menu.

'`Use port`': The standard port for API communication to other programs is 22081, but you can set a different port number here in case you want to run several instances of VAST and client programs on the same physical computer.

Display Properties

'`Maximal window width (pixels)`' specifies the width (or height, whichever is greater) of the largest window you plan to use, in pixels. This value is used to determine how many textured tiles are needed to fill the entire window at all zoom levels. Setting this to a smaller value reduces memory

consumption and increases cacheing speed, but if the value is too small, the image texture might not cover the complete 2D window at all zoom levels.

'Target resolution smaller than' lets you specify the effective resolution of the displayed textures on the screen in screen pixels per texture pixel. This affects at what zoom levels which mipmaps are used. '2' is a good general setting; '1' makes it more detailed but slower (and more memory-consuming), and '4' makes it faster but blurry/pixelated.

'Texture size (in pixels):' defines how large the texture tiles will be which are used for displaying image and segmentation textures. Depending on the graphics card some texture sizes might be faster than others. I recommend to leave this setting at 128.

'Texture smoothing': You can set here whether you want to use texture interpolation. This reduces aliasing effects but can result in a slightly blurred appearance of the textures. The most natural setting of this is, in my opinion, 'All except Mip 0', which will show pixels with sharp boundaries only if you zoom in more than the native resolution of the image data.

'Link tool windows to main window': When enabled, VAST's floating tool windows will automatically move to their default locations with respect of the main window when the main window is moved or rescaled.

The remaining options in this section set different UI opacity values between 0 (fully transparent) and 255 (fully opaque).

Behavior

'Use location trajectories': If this option is enabled, VAST will not instantly jump to a new location, for example when moving to the anchor point of a segment, but rather show a quick animation of moving through the image stack for the specified duration (0.25 seconds by default). This can convey a better understanding of where different locations are in the image stack.

'Z jump': This defines how far the 2D view will scroll in Z (assuming XY 2D view) when the right and left arrow keys are pressed (except when in Annotation mode). Start defines the start offset and Stepping the number of slices it will jump. The default parameters of Start 0 and Stepping 128 mean that the 2D view will scroll to slices 0, 128, 256, 384, ...

'Adjust scroll speed with mouse wheel while scrolling:': If this option is enabled the z-scroll speed (the rate at which you go through slices when holding down A, Z, S, X, Up, Down, PageUp, PageDown) is decoupled from the key repeat speed and can be controlled with the mouse wheel while one of those keys is held down. Roll the mouse wheel towards you to 'pull down' the scroll speed. Roll away from you to increase the scroll speed until you reach the maximum, which is the window update speed. The mouse wheel will still adjust the zoom level when used while you are not z-scrolling.

'Disable key navigation in tool window lists': If you use the arrow keys to scroll through the stack, key navigation in tool window lists can have unwanted side effects (change the selected segment, layer or annotation object). Therefore key navigation is disabled in VAST by default unless you enable it here.

'Enable safe saving': For speed and disk memory consumption reasons, the standard method to 'Save' a segmentation layer back to its file is to modify the file internally rather than to copy/modify everything to a new file (which is done when using 'Save As'). This reduces disk space consumption but can lead to file corruption if the saving process is interrupted. If 'Safe Saving' is enabled, the 'Save' function will internally use 'Save As' to generate a new file and delete the old file only after saving to the new file completed, and then rename the new file to the old file name. This prevents file corruption at the expense of disk space during saving.

Press OK after you're done configuring the preferences.

2.3 Additional Files Included With VAST

Under 'Info / Save Documentation .ZIP To Disk ...' in the main menu you can save out additional files which are packaged into the VAST executable, as a ZIP file. Select a target location and save, then unzip the ZIP file.

Currently this includes a set of .vsvi files to access some large EM data sets remotely, some Matlab scripts which can be useful for analyzing VAST data in Matlab, including the VastTools toolbox which can communicate directly with VAST through the API and provides additional functionality (see chapter 8), and this documentation as a PDF file.

Chapter 3

Working with VAST

VAST uses its own file format `.vsv`[1] to store image data. It can open `.vsv` files instantly and navigate in them quickly. If your data is a stack of for example `.png` images, you can either import it into a `.vsv` data file (recommended for image stacks of a few hundred Gigabytes or smaller), or prepare tiles, mipmaps and a `.vsvi` file to let VAST load the image data directly from image files which can be stored locally or on a web server (see section 3.2.5). After opening a `.vsv` or `.vsvi` image file in VAST, you can create a segmentation by painting on top of the images, or you can open an associated segmentation file (`.vss`)[2] and view image and segmentation together. `.vss` files tend to get big quickly, but can be packed efficiently, for example in a `ZIP` file. The new packed `.vss` files (introduced in VAST 1.4) also reduce the file size considerably compared to the unpacked format.

You can view segmentations, modify and save them. You can export segmentations as image stacks for using them in other analysis programs or to render the segmented objects in a 3D animation program like 3D Studio MAX or Blender. You can also import segmentation image stacks that were generated externally.

To generate skeletons (spatial binary trees of nodes and edges), use an Annotation layer (described in chapter 5). This function can be used for example to skeletonize neurites for morphology analysis or to proofread automatic segmentations (by supervoxel agglomeration) with the help of Tool layers (see chapter 6). Skeletons can also be exported and imported, and copied from one annotation layer to another.

3.1 Viewing and Navigating an Image Stack

You can open `.vsv`, `.vsvol`, `.vsvi`, or `.vsvr` files by using '`File / Open Image Volume ...`' in the main menu. Opened files will be added to a list under '`File / Open Recent Image Volume`', from where you can quickly access them again. The list contains the 16 most recent image stack files.

You can also open `.vsv` (and `.vsvol`, `.vsvi`, `.vsvr`, `.vss`, `.vsseg`) files by drag-and-drop from a file browser (Windows Explorer) onto the VAST window.

VAST comes with a set of `.vsvi` files to access some freely available large EM data sets remotely (see section 2.3 above). If you would like to import your own image data please refer to section 3.2 below.

[1] The file name extension .vsvol can be used instead, and may be preferable, because Microsoft thinks .vsv is a 'Microsoft Visio' file.

[2] Equivalently, you can use .vsseg instead of .vss as an alternative file name extension for VAST segmentation files.

VAST currently has 'Move', 'Annotation', 'Draw', 'Collect', 'Pick', 'Fill' and 'Split' modes, which you can set by clicking the tool buttons in the toolbar. The cross-of-arrows icon selects 'Move' mode and the little pencil selects 'Paint' mode. In this section we will explain how to use the 'Move' mode. For an explanation of the other modes please refer to sections 4.1 and 5.4.

The easiest way to navigate in the image stack is by using the mouse in 'Move' mode. You can pan (move the image sideways) by left clicking and dragging it. You can use the mouse wheel to zoom in and out. Alternatively, you can zoom using the N and M keys, or the side scrollbar (see below). Use the UP and DOWN arrow keys or A and Z to scroll through the slices of the stack, or the side scrollbar to scroll more quickly. Alternatively, hold down the '5' key on the keypad and scroll by dragging up and down with the mouse anywhere in the main window (NumLock has to be off).

The main 2D window can be switched between showing XY, XZ and YZ slices through the image volume. The – key on the keypad cycles between the different modes. You can also select a 2D slice orientation in the main menu under 'View'. In the XY view, the data X axis points to the right on the screen, the Y axis down, and the Z axis into the screen. In the XZ view, the data X axis also points to the right on the screen, the data Z axis down on the screen and the data Y axis out of the screen. In the YZ view, the data X axis points into the screen, the data Y axis points right and the data Z axis down on the screen.

3.1.1 Multi-panel view

The main window can now be split into two, three or four panels, the size of which can be adjusted. You can select a panel layout either from the main menu, under 'View / Set Panel Layout', or from the context menu by right-clicking the main window (except when in 'Paint' mode). To change the size of the panels, click anywhere on the panel boundary and drag with the mouse.

The active panel is determined by the location of the mouse cursor and indicated by a yellow frame. The content of the active panel can be set in the main menu under 'View / Set Panel Content', or from the context menu by right-clicking the main window (except when in 'Paint' mode). Currently, panels can be set to XY, XZ and YZ slice view, or to show the Schematic Viewer for skeletons.

Overall view update speed may suffer if several 2D views are shown at the same time, since they require loading different parts of the voxel data. This also causes some overhead in RAM consumption.

3.1.2 The side scrollbar

VAST provides a side scrollbar for quickly zooming and moving through the stack. The side scrollbar is a region close to the left and the right edge of the main window. When you move the mouse cursor to the left or right edge of the window you will see it appear as a transparent white overlay strip.[3] Clicking into the side scrollbar and dragging the mouse up or down will scroll through the slices of the stack (left mouse button) or zoom (right mouse button). If you move the mouse cursor too far away from the side of the window, the view will 'jump back' to the previous view. If you move the mouse cursor very close to the top or bottom of the window while scrolling (not zooming), VAST will start to scroll continuously, with a speed depending on mouse cursor position. You can use this function to quickly scroll through a very large image stack.

3.1.3 The RAM (main memory) usage indicator

At the right side of the toolbar you can see a little field named 'RAM:' which shows the current RAM (random access memory) usage in your computer. The blue frame indicates how much RAM VAST will use maximally for segmentation- and image cache combined. This amount should not exceed

[3] You can set the opacity of the side scrollbar in the Preferences, under 'Side scrollbar opacity'

approximately 2/3 of your total RAM (you can adjust these settings in the Preferences, see section 2.2.2). The solid blue block shows how much RAM VAST has currently allocated for segmentation and image cache. The light blue area shows how much memory is allocated by VAST for other purposes. The green area shows how much RAM the Windows system and other programs are using. The colors will change to yellow if the total memory usage goes above 90%, and red if they go above 96%. Running out of available RAM can slow down your system significantly. However, in some cases Windows uses large amounts of the available RAM for disk caching and can free those instantly if more memory is needed by programs without affecting the system performance.

3.1.4 Getting and setting coordinates

VAST uses a coordinate system with a zero point in the upper left corner of the first slice, with positive X to the right and positive Y down in the slice, and Z marking the slice number (front-to-back).[4] Coordinates are given in pixels at full resolution (the coordinates are independent of the mip map displayed). The coordinates displayed in the upper left corner of the main window show the current location of the center of the main window. You can switch the displayed coordinates on and off by clicking '`View / Coordinates Overlay`' in the main menu. Zooming in or out will not move the center point of the window and therefore also not change its coordinates. Getting or setting coordinates will also use the coordinates of the center of the screen, as do the 'anchor points' of segments (see section 4.1). While you drag the slice with the mouse VAST displays a semi-transparent cross which indicates the location of the center.[5]

Once you load an image stack, a tool window labeled '`Coordinates`' will appear in the upper right corner of the main window. If the tool window is not displayed you can open it using '`Window / Coordinates`' from the main menu. It shows you the current center coordinates and allows you to read and set these values. The edit field in the Coordinates window is updated as you navigate through the stack. To save the current location, simply copy the coordinates from that text field (mark with the mouse and press `CTRL-C`, or press the '`C`' button), then paste it into the text editor of your choice. You can also set the coordinates by entering or pasting numbers here and pressing `Enter`. Pressing the '`P`' button also pastes the text from the clipboard and VAST will attempt to interpret it as coordinates. VAST will then jump to the new coordinates. The exact format of the string does not matter; VAST simply looks for the first three numbers in the string. VAST does not care whether there are commas or brackets or other non-numerical characters in the string. This function is quite useful if you want to store coordinates of interesting points in an external text file or spread sheet.

Please keep in mind that the coordinate denotes the center of the current view. The center is indicated by semi-transparent cross-hairs when you pan the view. You can also center any point by right-clicking it with the mouse in 'Move' mode and selecting '`Center`' from the context menu.

The dropdown-listbox in the `Coordinates` tool window lists the up to 64 most recent locations you visited. A new entry is added every time you pan the view (but currently not if you scroll through Z). You can go back to previous locations by selecting the coordinates from this list.

3.1.5 Layers

VAST can open several image, segmentation and annotation files as layers at the same time, provided that they have the same stack size. Each layer is listed in the 'Layers' tool window. The order in the list defines the order of the layers for display in the 2D window. Layers BELOW in the list are

[4] The logic behind the front-to-back order of the image stack is that the images represent sections which were cut from the front surface of a tissue block, and therefore the first section should be closest to the observer for correct geometry. This assumption is used for 3D viewing and model export.

[5] You can set the opacity of the center cross in the Preferences under 'Center cross opacity', and select whether it should be always displayed or only when dragging ('Autohide').

'in front'. However, segmentation layers are always rendered on top of all image stack layers, and annotation layers are rendered on top of image and segmentation layers. You can change the order of the layers by drag-and-drop in the list. If you can not see all layers in the list, increase the size of the tool window by dragging a corner.

One image layer, and, if available, one segmentation layer and/or one annotation layer is always the selected layer of its kind, and its name is displayed in bold text in the 'Layers' tool window. One of these layers, the one that was clicked last, is also highlighted, and is the layer for which the 'Layer Properties' are shown below the list of layers in the 'Layers' tool window:

- '`Solo`': If this function is enabled, only the currently selected layer will be displayed.
- '`Editable`': Uncheck this for segmentation layers you do not want to accidentally paint into. Disabled for image layers
- '`Visible`' (image layers only): Transparency value for this layer. Switch off to hide layer.
- '`Bright`' (image layers only): Image brightness; switch on to enable brightness control with the slider
- '`Contrast`' (image layers only): Image contrast; switch on to enable contrast control with the slider
- '`Alpha`' (annotation, segmentation, and tool layers): Opacity value for this layer. Switch off to hide layer. Duplicated in toolbar.
- '`SelAlpha`' (annotation, segmentation, and tool layers): Opacity value for selected segments (and children). Duplicated in toolbar.
- '`Pattern`' (segmentation layers only): Contrast of segment patterns. Duplicated in toolbar.
- '`Scale`' (annotation layers only): Scales the width of edges and diameter of nodes.
- '`Param`' (tool layers only): Unused.

Layer blending

Image layers can be blended with different transparency modes. Click on the button '`Menu`' in the 'Layers' tool window to access more options for the selected layer. Under '`Blend Mode`', you can select either '`Blend`' for alpha-blending, '`Additive`' for additive blending, or '`Subtractive`' for subtractive blending. The different settings for the transparency function are:

- '`Flat`': All pixels in the image share the same transparency [Default]
- '`Dark Transparent`': The darker a pixel ((R+G+B)/3), the more transparent it is
- '`Bright Transparent`': The brighter a pixel ((R+G+B)/3), the more transparent it is
- '`Max(RGB) Dark Transparent`': The darker a pixel (Max(R,G,B)), the more transparent it is
- '`Max(RGB) Bright Transparent`': The brighter a pixel (Max(R,G,B)), the more transparent it is
- '`Black Transparent`': All black pixels are fully transparent, and the opacity of all other pixels is controlled by the '`Visible`' slider

For RGB image stacks, 'RED Channel Target Color', 'GREEN Channel Target Color' and 'BLUE Channel Target Color' let you swap and disable different color channels individually, and even assign an arbitrary mixed color to each channel for more sophisticated color space transformations. This is particularly useful for mapping colors in fluorescence microscopy image stacks.

'Color Filter ...' will open a color selection dialog where you can choose a color by which the layer images should be (subtractively) filtered for display. To not filter the images, choose white (255,255,255) [Default]. Enable 'Invert' to invert image layers (black becomes white and white becomes black).

For image layers, if you enable 'Show Color Clamping' in the Layers window menu, VAST will show which pixels reach saturation while the 'Bright' and 'Contrast' sliders are moved, by inverting the brightness (by color channel) for saturated pixels. This can be helpful for example when part of the image should become fully transparent in the 3D viewer by setting those pixels to saturated black or white.

Segmentation layers can be used as masks to make part of the displayed image stack black or white. This can be useful for example to show only the interior organelles within a segmented neuron in the 3D viewer. To enable masking, select the segmentation layer in the Layers window, and select the desired masking mode in the Layers window menu under 'Blend Mode'. To get the full masking effect, set the 'Alpha' slider to 100.

3.1.6 Control Buttons

Under 'Window / Control Buttons' in the VAST main window you can open a tool window which shows ten configurable, clickable on-screen buttons that can also be triggered with number keys 1,2,..0. Resizing the window will automatically resize the buttons to make them convenient to use on a touchscreen.

To configure a button, right-click it and choose the new function from the context menu:

- 'Sticky Keys': This turns the button into a toggle button which can be switched on and off by individual clicks to mimic the function of one of the modifier keys SHIFT, CONTROL, DELETE or SIZE.

- 'Navigation / Up, Down, Page Up, Page Down': Use this to duplicate keyboard control of moving through the image stack

- 'Navigation / Z-Jump Up, Z-Jump Down': Duplicates the right and left arrow keys in segmentation painting mode. The Z-jump start offset and stepping size is set in the Preferences (see section 2.2.2).

- 'Navigation / Go to Next Segment or Annoobject in List': This duplicates the '+' key on the keypad. If a segmentation or annotation layer is selected, this will select the next segment or annotation object in the list (in depth-first-search order) and move the 2D view to its anchor point or selected node if available. This function can streamline workflows which involve going through many segments or annotation objects one-by-one.

- 'Navigation / Go to Previous Segment or Annoobject in List': This duplicates pressing the '+' key on the keypad while holding down SHIFT. If a segmentation or annotation layer is selected, this will select the previous segment or annotation object in the list (in depth-first-search order) and move the 2D view to its anchor point or selected node if available.

- 'Navigation / Cycle 2D view XY / XZ / YZ': This duplicates the '-' key on the keypad and will cycle through XY, XZ, YZ, and back to XY slices in the 2D view (of the active panel if multiple panels are shown).

- 'Mode Change': This duplicates the mode selector buttons in the VAST toolbar and can be used to switch modes with a key press.
- 'Set Active Layer': If you are switching between layers often you can map selecting a particular layer to a screen button and number key
- 'Show/Hide Specific Layer': Switches visibility of a particular layer on and off
- 'Paint Settings': Some settings of the 'Drawing Properties' tool window can be mapped here
- 'Fill Settings': Some settings of the 'Filling Properties' tool window can be mapped here
- 'Label Text / Set Segment Label Text': Replaces the whole label text of the selected segment with the specified string. Equivalent functions exist for annotation object names and skeleton node labels.
- 'Label Text / Append Text to Segment Label': Appends the specified string to the label text of the selected segment. This can for example be used to rapidly classify segments by adding specific text tags to their labels. For example, appending [P] to the segment name could indicate a pyramidal cell and [I] an interneuron. In a second step you could collect all segments with [P] in their name into a single folder. Equivalent functions exist for annotation object names and skeleton node labels.
- 'Label Text / Replace Part of Label Text': For this function you have to specify a text to be found in the label and the text that it should be replaced with. For example, if you use text tags and you want to append several tags, you could use ']' as 'Find' string and 'P]' as 'Replace With' string. Equivalent functions exist for annotation object names and skeleton node labels.
- 'Skeletons / Go to Parent Node, Go to Child Node 1/2, Go to Selected Node': Use these to navigate skeletons by using control buttons
- 'Skeletons / Toggle Exclude Node From Agglomeration': Toggles the agglomeration exclusion flag of the selected node of the selected skeleton in the selected annotation layer. Refer to section 6.1 for an explanation of this functionality.
- 'Skeletons / Set Parent Edge Type ...': Lets you change the type of the parent edge of the selected node of the selected skeleton in the selected annotation layer.
- 'Skeletons / Find Node Again (F3)': This is the 'search again' function for skeleton nodes; see section 5.4.3.

The control button mapping can be saved to, and loaded back from, .CSV files. To do this, right-click any control button and select 'Save Control Buttons Mapping ...' or 'Load Control Buttons Mapping ...' from the context menu.

3.2 Image Stack Importing

Volumetric image data stored as a series of 2D images, or as a serial 3D block of data, is not ideal for fast interactive viewing. When you import such a data set into VAST as a .vsv (or .vsvol) file, the images get converted to a diced data structure, mipmaps for the images are computed, and all

is stored within the `.vsv` file.[6] Using diced data does not only speed up loading of parts of images, but also enables fast loading of volumetric sub-regions or 2D sections at other orientations through the image data.

VAST does currently not include image alignment and stitching functions. If you are starting with an unaligned stack of images, you will first have to align the images with a different program (Fiji or Photoshop, for example) and then save a stack of aligned images which all have the same dimensions and are named and numbered in a consistent way (for example `img000.png, img001.png, ...`). Ideally all images should be in the same folder.

VAST can import single-tile image stacks, multi-tile image stacks, and 3D volume files. In a single-tile image stack, each slice of the stack consists of a single image file. In a multi-tile image stack, each slice is composed of several tiles in a XY grid, and each tile is stored in a separate image file. A 3D volume file stores all slice images in a single file (currently supported are multi-image .TIFF and NIfTI .nii files). VAST will convert image data to either 8-bit graylevel or 24-bit RGB when importing.

For importing and dicing, VAST will use the RAM cache which is normally used for cacheing EM image data during viewing and painting. Having lots of cache memory available will make importing somewhat faster, because images have to be re-loaded less often. You can set the size of the EM image cache in the Preferences (see section 2.2.2).

3.2.1 Importing image stacks: Pattern-based names

In the main menu of VAST, go to '`File / Import / Import EM Stack from Images to .VSV file ...`'. VAST will show a file browser dialog in which you can select one or several image files. For importing 3D NIfTI or multi-image TIFF files, please select only one file. If you import a single-tile stack and do not want to use pattern-based names, select all slice images in the correct order, because images will be stacked in the same order in which they appear in the system's list of selected files. The order is usually correct if you select the last image first, then shift-click (hold the `SHIFT` key down and click left with the mouse) the first image to select the whole range. You can also 'Select All' by pressing `CTRL-A` if the folder only contains the image files you want to import. If you want more precise control, you can use pattern-based names. If you make use of pattern-based names to import single- or multi-tile image stacks, it is sufficient to select one file, but even better to select the first and the last file in your set of images. Then click '`Open`'.

After selecting one or more image files (except for NIfTI and multi-image TIFF files), VAST will display the dialog shown in Figure 3.1. To import without pattern-based names, select 'Make Single-Tile Stack Using File Names and Order as Selected' and press `OK`.

If you select the second option, '`Use Pattern-Based Names`', the parameters in the lower part of the dialog window will be enabled. With pattern-based names, you specify a template string for the file names which contains placeholders for numbers, and ranges for these numbers. With this you can also import image stacks in which each slice is stored in several image files (multi-tile image stacks). This is useful for data sets in which a single slice is so large that it can not be stored in a single image file, but is stored in a set of tiles which form a regular grid. Please note that these tiles should *not* be unstitched image tiles as they come off a microscope, but they have to fit seamlessly. If you have a set of raw microscopic images which are not yet stitched and aligned, please use an external program to generate a stitched and aligned image stack first, and store each slice as a single image or a set of image tiles. You can then import those images into VAST.

VAST will use the file(s) you selected in the previous dialog to determine the source directory where the image files are and to generate a basic template for the file name. It assumes that all

[6] A *mipmap* is a downsampled version of an image. VAST uses power-of-two (2D, XY) mipmaps. For example, for an original image of 1024x1024 pixels, it will compute mipmaps of 512x512 and 256x256 pixels. It does this for every slice image in a stack.

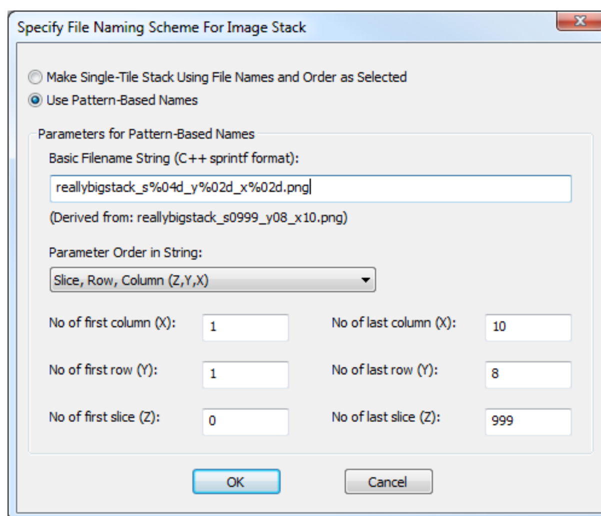

Figure 3.1: First dialog for importing EM image stacks: Specification of pattern-based names

images of the stack are in the same folder[7] (the one you picked an image from), and are named consistently with numbers for slices, rows and columns. In this dialog, you specify these ranges as well as a schema to derive the filename for a given slice/row/column coordinate. Not all image tiles have to exist, but VAST will warn you if image tiles can not be found.

Let's say, for example, you have a data set called '`bigstack`', which has 1000 slices, numbered from 0 to 999, and each slice has 10x8 tiles, numbered from 1 to 10 and 1 to 8. Assume the image in the upper left corner of the first slice is called '`bigstack_s0000_x01_y01.png`', the image tile right of it is called '`bigstack_s0000_x02_y01.png`', and so on. The last image in the lower right corner of the last slice would be called '`bigstack_s0999_x10_y08.png`'.

First, make sure that the file name in the edit box at the top contains the correct *C++ format string* (as it is used by `printf()`). In general, numbers which specify the slice, column and row coordinates have to be replaced by codes like '`%d`' (integer number) or '`%04d`' (integer number with zero-padding to 4 digits). VAST will then fill in those numbers for each image. The correct string for the example above would be: '`bigstack_s%04d_x%02d_y%02d.png`'. For more information about format strings, please refer to a C++ manual or ask the internet.

You can also use wildcards ('*') in the file name to allow for part of the name to change arbitrarily, for example if the wafer number or date or other information is included in the filename.

In the combo box below, select which coordinates are used in the file names, and in which order they appear. The edit boxes below let you specify the range of (integer) numbers for the three coordinates. Figure 3.1 shows how to set up these parameters for the example discussed above. After you entered all parameters, press `OK`.

3.2.2 Lossless and lossy compression

Next, VAST will let you specify the color mode and image compression (Figure 3.2).

Under 'Color Mode', please select if you want to import the images as 8-bit graylevel or 24-bit color images, and for graylevel which source color channel to use. When importing from graylevel images, please select the first option ('from RED channel'). For 16-bit source images, you can choose

[7] There is a workaround if images are stored in separate folders per section – You can specify a relative path within the filename template and use the slice parameter to index the folder. Remove an additional slice counter in the filename by using a wildcard (*).

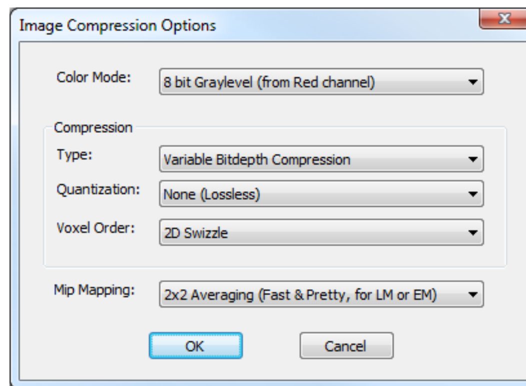

Figure 3.2: Second dialog for importing EM image stacks: Image compression options

whether VAST should keep the upper ('16bit MSB') or lower 8 bits ('16bit LSB') when converting to 8-bit images.

Under 'Compression' you can specify compression options. Under 'Type' you can select the compression method - Uncompressed, Variable Bitdepth Compression, zlib Compression, and Spectral Compression. The three different compression algorithms are by themselves lossless, but might produce slightly smaller or larger file size depending on your data. Variable Bitdepth Compression should be fastest when reading from the compressed files.

'Quantization' specifies whether the compression should be lossy or lossless. Lossyness is achieved by quantizing, meaning throwing away bits. For example, if you set Quantization to -2 bits, graylevel images will have only 6 bits resolution (64 different gray levels) rather than 8 bits (256 different gray levels). Throwing bits away reduces both file size and image quality.

'Voxel Order' defines in which order the pixels in the images will be stored. This can have an effect on compressed file size. '2D Swizzle' stores pixels in 2D Z-order. I usually get best results using '2D Swizzle', but '3D Swizzle' might be superior for very well aligned data (e.g. FIB-SEM).

'Mip Mapping' lets you select whether you would like to average or subsample pixels for the generation of mip maps. Averaging gives superior image quality for EM and LM image stacks, but changes the pixel values. Therefore, if pixel values should be preserved, for example if the image stack represents an automatic segmentation result (see section 4.1.5), subsampling should be used instead of averaging.

Next VAST will ask you to specify a target location and file name for the resulting `.vsv` image volume file. Use '`.vsv`' as extension for the file name. Choose a location where you have enough storage space for the file. The file will not only contain the original image data, but also the mip maps. For example, if you import 1024 images of 1024x1024 pixels each and store uncompressed, the `.vsv` file will be approximately $1024 \cdot 1024 \cdot 1024 + 512 \cdot 512 \cdot 1024 + 256 \cdot 256 \cdot 1024 = 1409286144$ Bytes (≈ 1.3 Gigabytes) large. Lossless compression will reduce the file size, and lossy compression even more, but by how much depends strongly on your data and the compression method used.

Then, the images will be read, diced, and put into the target file. After that, VAST will compute the mipmaps and put those in the target file too. This process can take several hours depending on the size of the data. For example, a big data set of 350 GB takes about 5 hours to import on a recent desktop machine. The limiting factor is usually the transfer speed of data from and to the hard drives.

Lastly, VAST will ask you to specify the XYZ voxel size of your image stack in nanometers (see below, section 3.2.4).

You can cancel the importing, but the target file will then be incomplete / corrupted and can not be used with VAST.

3.2.3 Importing 3D volume files

Importing a 3D volume file is easier than importing an image stack. VAST currently supports multi-image TIFF and NIfTI files. VAST will ask you to specify the name of the source file and the name of the target `.vsv` file. For NIfTI, files are loaded into memory completely for importing, so this only works for smaller volumes, and they are expected to store 8-bit data.

3.2.4 Image scale and description

After importing, VAST will show a dialog where you can set the voxel size in nanometers of your data in the file. Press 'Save to file' in the dialog to save the information you entered to the VSV file. The voxel size entered here is used for the scale bar which you can enable in the main menu under '`View / Scale Bar Overlay`', for the 3D viewer, and for scaling models and measurements in VastTools. You can access and change these numbers for the selected image layer under '`Info / Volume properties ...`' in the main menu. This dialog also displays how large your image stack is in voxels.

Under '`Info / EM File Information ...`' in the main menu you can enter and view text which will be stored in your VSV file as well. This can contain a description of the data, copyright information, or other.

3.2.5 Preparing .VSVI files for large image stacks

For image stacks which are larger than a few hundred gigabytes, importing to a `.vsv` file may become unfeasible. Importing might take weeks or months, and the resulting file might become too large for the file system or for storage on a single drive. For these cases VAST supports reading image stacks directly from sets of image tiles. However, the images have to be prepared in a particular way and a `.vsvi` file has to be created which tells VAST how to read different parts of the stack from the images. The following conditions have to be met:

- All image tiles should be stored in the same folder tree and named in a regular way so that VAST can generate file names to load all tiles needed for a particular region. The rules of tile naming are described in the `.vsvi` file for the data set (see below).[8]
- Powers-of-two mipmaps (2D, XY) have to be precomputed and stored with the original-resolution tiles.
- Image tiles should be small enough so that jumping to a new location will not result in a long delay before images show up, and the overhead for loading a region of data remains reasonable. Images tiles should also have a size in pixels which is a multiple of 16 in X and Y. 1024x1024 pixel tiles work well for us. Of course there is a tradeoff between the size of a single tile and the number of tiles which have to be stored. Split into tiles of 1024x1024 pixels, a 100x100x100 micrometer dataset at a resolution of 4x4x30 nanometers, which is roughly 2 Terabytes large, results in more than 2.8 million image files. To reduce the number of files the `.vsvi` format also supports image files in which image patches of 16 consecutive slices are concatenated vertically.

Storing image data as a set of files also has other advantages: Parts of the image stack can easily be read or updated by other programs, and the image stack can also be extended easily, without having to re-import all images.[9] Also, since different resolutions (mip maps) of the stack are stored

[8] One exception is that the full-resolution images can be located in a different path than all mipmaps.

[9] With limitations – Other dependent data sets, like segmentation layers, use a predetermined stack size and will not fit to the image stack any more if its size or alignment is changed.

separately, it is possible to only copy lower-resolution mip maps to a target drive and thus generate a much smaller lower-resolution version of the same data set.

.vsvi files are text files in a JSON-like format. Below are the contents of a .vsvi file which describes a locally stored, tiled image stack with mip maps:


```
{
  "Comment": "IARPA R0 100mu normalized",
  "ServerType": "imagetiles",

  "SourceFileNameTemplate": ".\mip0\%04d_*_montaged\%04d_*_tr%d-tc%d.png",
  "SourceParamSequence": "ssrc",
  "SourceMinS": 1,
  "SourceMaxS": 3394,
  "SourceMinR": 1,
  "SourceMaxR": 25,
  "SourceMinC": 1,
  "SourceMaxC": 25,

  "MipMapFileNameTemplate": ".\mip%d\slice%04d\%04d_tr%d-tc%d.png",
  "MipMapParamSequence": "mssrc",
  "SourceMinM": 1,
  "SourceMaxM": 7,

  "SourceTileSizeX": 1024,
  "SourceTileSizeY": 1024,
  "SourceBytesPerPixel": 1,
  "MissingImagePolicy": "nearest",
  "TargetDataSizeX": 25600,
  "TargetDataSizeY": 25600,
  "TargetDataSizeZ": 3394,
  "OffsetX": 0,
  "OffsetY": 0,
  "OffsetZ": 0,
  "OffsetMip": 0,
  "TargetVoxelSizeXnm": 4,
  "TargetVoxelSizeYnm": 4,
  "TargetVoxelSizeZnm": 30,
  "TargetLayerName": "IARPA R0 100mu normalized"
}
```


The different parameters in the file have the following meaning:

- "Comment": This text will be displayed in the user selects 'Info / EM File Information ...' from the main menu in VAST
- "ServerType": This has to be either "imagetiles", "http" or "googlecloudstorage". Use "imagetiles" for local image files. "http" or "googlecloudstorage" are described below (section 3.2.5).
- "SourceFileNameTemplate": The filename template (with relative path from the location of this file) for accessing the full-resolution image tiles, in printf() format. This may include wildcards (*).
- "SourceParamSequence": The parameter order of number parameters in the above template. "ssrc" for example means Slice, Slice, Row, Column. "f" is used together with "%s" for more complex name specifications, for example for the "neuroglancer_sharded" format (see below).
- "SourceMinS": The minimal value used for the Slice (Z) parameter in the stack. Slices count from the top to the bottom of the stack (front to back in 3D, assuming they are sections which are cut from a block).

- "SourceMaxS": The maximal value used for the Slice (Z) parameter in the stack
- "SourceMinR": The minimal value used for the Row (Y) parameter in the full-resolution stack. Rows count from the top to the bottom of the screen
- "SourceMaxR": The maximal value used for the Row (Y) parameter in the full-resolution stack
- "SourceMinC": The minimal value used for the Column (X) parameter in the full-resolution stack. Columns count from left to right on the screen
- "SourceMaxC": The maximal value used for the Column (X) parameter in the full-resolution stack
- "MipMapFileNameTemplate": The filename template (with relative path from the location of this file) for accessing the mip-map image tiles, in printf() format. This may include wildcards (*).
- "MipMapParamSequence": The parameter order of number parameters in the above template. "mssrc" for example means Miplevel, Slice, Slice, Row, Column.
- "SourceMinM": The minimal value used for the mip levels (M) parameter in the stack. This should always be 1. In VAST, mip level m has a reduction factor of $1/2^m$ in X and Y, meaning that mip level 0 is full resolution, 1 is half resolution (in both X and Y), 2 is quarter resolution and so on. The number of sections can also be halved if XY/XYZ mipmaps are used (see "MipMapZScaling" below).
- "SourceMaxM": The maximal value used for the mip levels (M) parameter in the stack. VAST expects a specific value here which depends on the maximal size of the dataset in pixels in X or Y; the lowest-resolution mip level should scale the data so that the maximal extent drops below 512 pixels. VAST will warn you if the value here differs from the expected value.
- "SourceTileSizeX": This is the horizontal size of a single image tile, in pixels
- "SourceTileSizeY": This is the vertical size of a single image tile, in pixels
- "SourceBytesPerPixel": This value is 1 for graylevel images and 3 for RGB images, 4 for RGBA images, and 4 or 8 for color-mapped segmentation data
- "MissingImagePolicy": This option describes what VAST will do if an image tile file can not be found. It can be set to "black" (show black tile), "white" (show white tile), or "nearest" (search up or down in the local 16-slice window for the nearest image tile which exists and use that instead; if none is found, leave black)
- "TargetDataSizeX": X extent of the whole stack, in pixels at full resolution
- "TargetDataSizeY": Y extent of the whole stack, in pixels at full resolution
- "TargetDataSizeZ": Z extent of the whole stack, in pixels at full resolution
- "OffsetX": X placement offset of image stack. Please leave this 0 at the moment.
- "OffsetY": Y placement offset of image stack. Please leave this 0 at the moment.
- "OffsetZ": Z placement offset of image stack. Please leave this 0 at the moment.
- "OffsetMip": Resolution shift in number of mipmaps
- "TargetVoxelSizeXnm": X size of one voxel at full resolution, in nanometers

- "TargetVoxelSizeYnm": Y size of one voxel at full resolution, in nanometers
- "TargetVoxelSizeZnm": Z size of one voxel at full resolution, in nanometers
- "TargetLayerName": Name of the layer when loaded into VAST. (Currently unused; instead, the file name is displayed as layer name)

In this example the main folder contains the `.vsvi` file and subfolders `mip0, mip1, ..., mip7`. `mip0` contains the full-resolution tiles, in a separate subfolder per slice. The subfolder for slice 0 files is named `0001_W01_Sec001_montaged` for example, and the subfolder for slice 3394 is named `3394_W13_Sec271_montaged`. As you can see the name components describing the wafer number W and section number Sec are changing but are not used for indexing the slice. They are therefore replaced by a wildcard (*) in the filename template in the `.vsvi` file.

Additional parameters for online-hosted image stacks:

In addition to image files stored locally, the VAST .VSVI format also supports images stored on http servers (including Amazon AWS) and on Google cloud storage.

`"ServerType": "http"` – This is used to load images which are openly available online using a URL. If you use this, you also have to specify the parameters "ServerName" (the name of the server) and "ServerPort" (default: 80). "SourceFileNameTemplate" then contains the file path on the server. – The VAST documentation package (save under "Info/Save Documentation .ZIP to Disk ..." in the VAST main menu) contains an example .VSVI which implements this, `Kasthuri15_AC4i_http.vsvi`.

`"ServerType": "googlecloudstorage"` – VAST supports loading images stored in Google cloud buckets, including OAuth2 authentication. To use this source, you'll also have to specify the following parameters in the VSVI file:

```
"ServerType": "googlecloudstorage",
"ServerName": "storage.googleapis.com",
"ServerPort": 443,
"LoopbackPort": 9004,
```

"SourceFileNameTemplate" and "MipMapFileNameTemplate" should include the file path on the server, starting with "/". While this works in a test, it may produce some security warning messages from Google.

- "ServerName": name of the fileserver, as in the URL
- "ServerPort": port to use; typically 80 (http) or 443 (https)
- "LoopbackPort": Necessary for Oauth2 authentication
- "SourceDataEncoding": either `raw` or `RAW`, `compressed_segmentation` (compressed segmentation as used by Jeremy Maitin-Shepard (Google) and Zetta.ai), or `neuroglancer_sharded`
- "neuroglancer_sharded": The format `neuroglancer_sharded` requires many more parameters, which can be defined under this key; see below.

Support for the Neuroglancer Precomputed Sharded format

VAST can also load image data from Google's 'Neuroglancer precomputed sharded' format.[10] To access such image stacks VAST needs many more data fields which can be derived from the dataset's

[10]https://github.com/google/neuroglancer/blob/master/src/neuroglancer/datasource/precomputed/sharded.md

`info` file. A detailed description is beyond the scope of this manual, but the dataset access files `H01_EM_8nm.vsvi` and `H01_SEG_C3_8nm.vsvi` provided in the supplementary package can serve as an example. Since .vsvi files are ASCII text files they can be viewed and edited in a simple text editor.

Additional parameters for specifying folder names, tile sizes and dataset sizes for each mip level separately:

Some datasets do not adhere to VAST's data format and naming standards but clients still want to load them into VAST. The following set of parameters may help in this case:

- "SourceMip0FolderName", "SourceMip1FolderName", ...: folder name for images, explicitly specified per mip level
- "SourceMip0TileSizeX", "SourceMip1TileSizeX", ...: X tile size in pixels, explicitly specified per mip level
- "SourceMip0TileSizeY", "SourceMip1TileSizeY", ...: Y tile size in pixels, explicitly specified per mip level
- "SourceMip0TileSizeZ", "SourceMip1TileSizeZ", ...: Z tile size in pixels, explicitly specified per mip level
- "SourceMip0DataSizeX", "SourceMip1DataSizeX", ...: X dataset size in pixels, explicitly specified per mip level
- "SourceMip0DataSizeY", "SourceMip1DataSizeY", ...: Y dataset size in pixels, explicitly specified per mip level
- "SourceMip0DataSizeZ", "SourceMip1DataSizeZ", ...: Z dataset size in pixels, explicitly specified per mip level

Mip levels 0 - 19 are supported for this.

Optional parameters are:

- "SourceSectionOrder": For example `"SourceSectionOrder": "36:68, 72, 70, 84:166"` – optional direct specification of the numbers to be used for the source sections, in Matlab notation. If this parameter is included, `SourceMinS`, `SourceMaxS`, and `OffsetZ` still have to be included but are ignored.
- "SourceMipTileSizeX", "SourceMipTileSizeY": If specified, these parameters can be used to set the tile size of the mip maps separately; otherwise, mip maps will be assumed to have the same tile size as the full-resolution images (as specified in "SourceTileSizeX" and "SourceTileSizeY").
- "SourceTileSizeZ", "SourceMipTileSizeZ": These are set to 1 by default, which means that each image file is expected to contain a 2D planar image tile of the dataset in XY direction. If you set this to values $n > 0$, VAST expects that each image file consists of n vertically concatenated image patches which contain n consecutive Z sections of the same XY region. This can speed up load times considerably on slow file systems (for example server storage mounted as a local drive via Samba) and at the same time reduce the number of files the server has to handle. The behavior can be set independently for full-resolution images ("SourceTileSizeZ") and mip map images ("SourceMipTileSizeZ"). It has been tested for `"SourceMipTileSizeZ": 16`. You can use `S` in "SourceParamSequence" and/or "MipMapParamSequence" to denote the

ending section of each composite image file, if you want to use that for folder of file names. Consider this example:

```
"MipMapFileNameTemplate": ".\mip%d\slice_%04d-%04d\r%02d\%04d-%04d_tr%d-tc%d.jpg",
"MipMapParamSequence": "msSrsSrc",
```

This would create file names like: `.\mip3\slice_1-16\r04\0001-0016_tr4-tc3.jpg`.

- "MipMapZScaling": VAST supports image stacks in which mip maps can have consecutively fewer sections than the full-resolution data. This can be used to keep the voxel dimensions of low-resolution mip maps approximately equal in X, Y and Z (isometric). Typically for data which has a higher resolution in X and Y than Z, mip map downsampling would first be done in X and Y only until an approximately isometric voxel size is reached, followed by downsampling in X, Y and Z for even lower-resolution mip maps. VAST has support of such image stacks for VSVI and VSVR sources as well as segmentation and annotation layers. To specify an image stack of this kind, include the "MipMapZScaling" option with a list of relative Z scales per mip layer, including full res (mip 0). The list should be 1s followed by 2s. For example:

  ```
  "SourceMinM": 1,
  "SourceMaxM": 10,
  "MipMapZScaling": "1 1 1 2 2 2 2 2 2 2 2",
  ```

  This would assume the same number of sections as the full-resolution data (mip 0) for mips 1 and 2, half the number of sections as mip 0 for mip 3, a quarter the number of sections as mip 0 for mip 4, and so on.

- "CachedMip0MainFolder": This optional parameter can be used to instruct VAST to read and cache the contents of the main folder for the full-resolution image data (mip 0). This can improve reading speed if that folder contains a lot of subfolders (for example one for each section with thousands of sections) and is hosted on a slow file system. The argument of the parameter is a string in "" specifying the path of the mip 0 main folder.

- "NumberOfImageLoaders": For image stacks hosted on a server where reading several files at once does not introduce a performance penalty this parameter can be used to improve the loading speed. The recommended value for this is 4, and it should not be set higher than 8 since VAST uses 16 rotating loading buffers internally and will stall if they are reassigned too often (but see "NumberOfImageTargets" below).

- "NumberOfImageTargets": Number of internal cache slots to use for loaded image tiles. Default is 16.

- "TargetColMapped": If this is set to 0 for a data set with "SourceBytesPerPixel" 4, the data will be interpreted as RGBA rather than color-mapped (which is the default).

Parameters for local disk caching:

For datasets which are stored on a file server it can make sense to save data which has been downloaded once on the local hard drive for faster loading when it is needed again. VAST supports this function if the following parameters are added to the `.vsvi` file:

- "FileCacheFolder": the folder to use for storing the cached images

- "ReadFileCache": set to 1 if VAST should read from the local file cache, 0 else

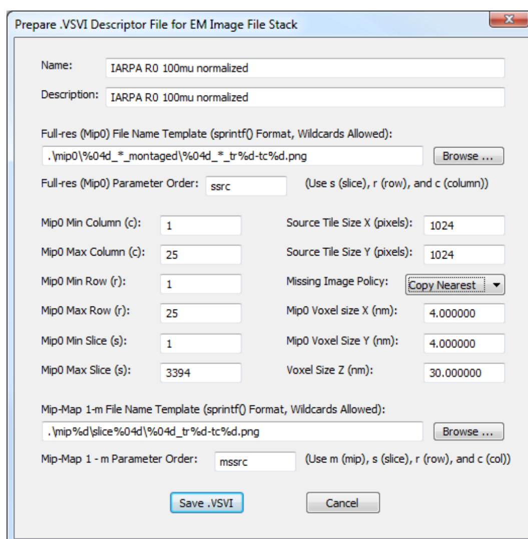

Figure 3.3: Dialog for the preparation of a .vsvi file, filled out for the example in this section

- "ReadFileCacheMipMaps": specifies which mip levels to read from the local cache (see example below)
- "WriteFileCache": set to 1 if VAST should write to the local file cache, 0 else
- "WriteFileCacheMipMaps": specifies which mip levels to write to the local cache (see example below)
- "WaitForWriteFileCache": set to 1 to have the main thread of VAST wait for cache writing. If 0, loaded image data will only be written to the local disk cache if time allows

Example for a local disk cache in a subfolder of where the `.vsvi` is located, which caches all mip levels except 0 and 1:

```
"FileCacheFolder": ".\H01_EM_8nm_cache\",
"ReadFileCache": 1,
"ReadFileCacheMipMaps": "0 0 1 1 1 1 1 1 1 1",
"WriteFileCache": 1,
"WriteFileCacheMipMaps": "0 0 1 1 1 1 1 1 1 1",
"WaitForWriteFileCache": 1,
```

Generating VSVI files in VAST

VAST has a function to help generating a standard `.vsvi` file for an image data set. To run this, select '`File / Import / Prepare .VSVI Image Stack ...`' in the main menu of VAST. First, VAST will ask you to specify a target file name for the `.vsvi` file and a location where to save it. Ideally the `.vsvi` file should be stored in the main folder for the data set and all the image tiles should be stored in subfolders, so that they can be referenced by a relative path. Next, VAST will ask you to select one or more example full-resolution image tiles. Then it will show you a dialog to specify the parameters of the `.vsvi` file. For this example the parameters should be set as shown in Figure 3.3.

Once you click the 'Save .VSVI' button, the information you entered will be saved to the `.vsvi` file you specified beforehand. VAST will not immediately open that file but you can load it in the same way as any other `.vsv` or `.vsvr` file. Since the `.vsvi` file is a text file, you can also view and edit it with normal text editors.

The `.vsvi` preparation tool will compute the total extent of your image stack from the tile size and number of tiles which you provided. It will also compute how many mip levels will be expected by VAST and store that value in the `SourceMaxM` parameter. The lowest-resolution mip map which VAST uses is the first one in which both width and height of the complete mip map slice image are less than 512 pixels. You can check which value VAST wrote to the `.vsvi` file to make sure you provide all mip maps needed.

This function does not contain all options discussed above, but you can easily modify the .VSVI file with a text editor to adjust/add parameters.

3.2.6 Remote image data (VSVR)

VSVR files let you open and access image stacks which are hosted by an online cutout service. VAST supports the *'Open Connectome Project Cutout Service'* from `http://www.openconnecto.me`[11], the newer `neurodata.io` data servers, Harvard's *Butterfly* cutout service, and Google's *'Brainmaps'* framework. To access a remote image stack you need a `.vsvr` file which specifies the parameters of the data set. `.vsvr` files are text files in a JSON-like format and share many parameters with the `.vsvi` files as described in section 3.2.5. Here is the content of an example file, `Kasthuri15_Harvard.vsvr`:[12]

```
{
  "Comment": "Kasthuri et al. 2015 Mouse Cortex Dataset, Lichtman Lab, Harvard",
  "ServerType": "butterfly",
  "ServerName": "jwlconnect.rc.fas.harvard.edu",
  "ServerPort": 80,
  "ServerFolder": "butterfly/data/",
  "DataPath": "/n/www_glichtman/public/kasthuri/mojo/mojo/",
  "SourceDataSizeX": 21504,
  "SourceDataSizeY": 26624,
  "SourceDataSizeZ": 1850,
  "TargetDataSizeX": 10747,
  "TargetDataSizeY": 12895,
  "TargetDataSizeZ": 1850,
  "OffsetX": 0,
  "OffsetY": 0,
  "OffsetZ": 0,
  "OffsetMip": 1,
  "TargetVoxelSizeXnm": 6,
  "TargetVoxelSizeYnm": 6,
  "TargetVoxelSizeZnm": 30,
  "TargetLayerName": "Kasthuri15 BF Harvard"
}
```

Optional parameters are "GoogleChangeStackID" and "LoopbackPort" which are used with Google Brainmaps data sets, as well as "MipMapZScaling" and "SourceBytesPerPixel" (see explanation above). Supported bytes per pixel are 1 (8-bit graylevel), 3 (24-bit RGB), 4 (32-bit) and 8 (64-bit). 32-bit and 64-bit images will be shown in VAST using semi-randomly assigned RGB values.

[11] See for example `https://arxiv.org/abs/1306.3543`. Essentially VAST requests zipped [128x128x16] pixel blocks of the data set from the data server with URLs which specify the requested region, like: *http://openconnecto.me/ocp/ca/kasthuri11/zip/6/1,129/1,129/1,17/*. The received file is then unzipped to extract the image data.

[12] This file will currently not work since the butterfly server on jwlconnect.rc.fas.harvard.edu has been retired.

Enabling local disk cache for remote datasets loaded via .VSVR files

You can also have the image data cached locally on disk, equivalent to similar functionality for `.vsvi` files described above. To enable, the following additional parameters have to be specified in the `.vsvr` file:

- ReadFileCache: Set to 1 if you want VAST to read images from the local file cache, 0 else. Only data not in the file cache will be requested from the network.
- WriteFileCache: Set to 1 if you want VAST to write newly downloaded image data to local files, 0 else. Set this to 0 for example if your hard drive is getting full and you don't want VAST to write any more image files.
- FileCacheFolder: This text string specifies the path to the main folder in which the images should be saved. A good way to organize your image data is to keep all the files in a subdirectory of where the `.vsvr` file is stored; for this, use a relative path starting with `".\"`.
- TileSizeXY: Specifies the X and Y size of the stored image tiles in pixels. VAST will concatenate 16 such tiles (of consecutive slices in Z) in each image file to reduce the overall number of files. I recommend keeping this value at 256.

Here is an example how this can look in a `.vsvr` file (please remember that only the last line should not have a comma in the end):

```
{
  "ReadFileCache": 1,
  "WriteFileCache": 1,
  "FileCacheFolder": ".\Kasthuri15_Harvard_cache\",
  "TileSizeXY": 256
}
```

3.3 Exporting Image Stacks

Exporting image volumes and/or segmentations as image stacks can be useful to generate slice animations, to transfer data to other programs like Fiji (ImageJ), or to process your results further externally in other ways.

To export a stack of image files from your volumetric data, select '`File / Export ...`' in the main menu. The dialog shown in Figure 3.4 will pop up. You can export screenshot image stacks, stacks of single-layer image data, and segmentation ID image stacks. You can specify a region of the data set to be exported, and a resolution (mip level) for the images. Data regions which are too large for storing in a single image per slice can be exported as a set of image tiles.

Export as:
Choose here if you want to export a single-tile image stack (one tile per slice) or a multi-tile stack (a grid of image tiles per slice). For the second option you can define the tile size to be used.

Region to export (Specify coordinates at full resolution):
This defines the region of the stack that should be exported, for all three targets (Screenshot, Image data, Segmentation). By default it is set to the full stack. You can restrict the export region here. X is the horizontal axis in the slice, pointing to the right; Y is the vertical axis in the slice, pointing downwards on the screen, and Slice (Z) defines the range of slices that should be exported. The first and second columns of the edit fields let you define start and end of the region for each axis, the third column defines the size of the exported region (edit fields change each other to stay consistent).

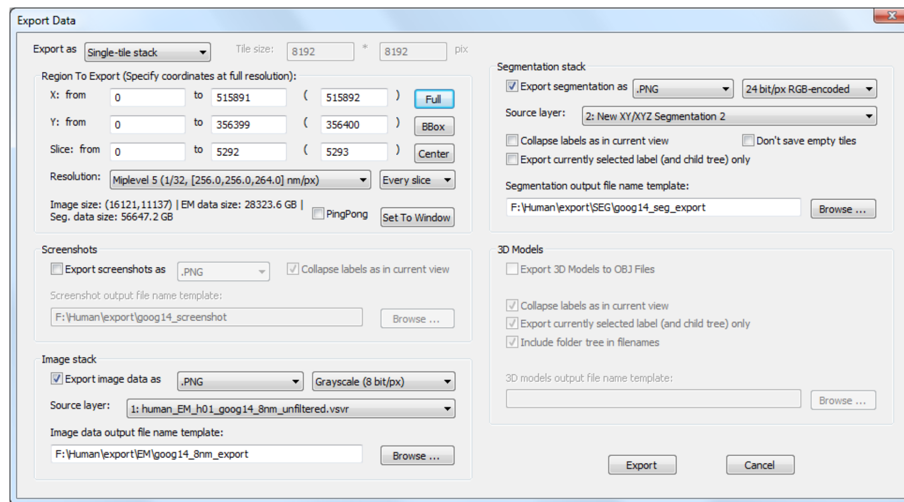

Figure 3.4: Export dialog

The 'Full' button will set the values back to the full extent of your stack.

The 'BBox' button sets the region to the bounding box of the currently selected segment and children. You can use this function to define a cut-out region from a painted segment[13], or use a new segment and just paint the upper left top and lower right bottom corners of the cutout region, select it and use the 'BBox' button to set the export region.

The 'Center' button will center the exported region to the current coordinates in the main window. You can for example use this in the following way: First move the image in the main window to the location you want to export. Then open the export dialog and specify the size of the image you want to export in pixels, in the third column of the 'Region to Export' section, for example to 1024, 1024, 1 for one image with size (1024,1024). Then click 'Center' to place the export region at the currently viewed location.

The 'Set To Window' button will set the export region to what is currently shown in the main 2D window of VAST.

Under 'Resolution:' you can select at which mip level you want to export. VAST does not support arbitrary scaling, but can export image stacks at its native mipmap scales (which are powers of two). You can also subsample the stack by slices (every nth slice). Below you can see the image size resulting from your settings and an estimate of the (raw) data size that will be exported. Compressed image formats like .PNG can however produce much smaller file sizes, depending on the image content.

PingPong can be used to generate slice animation videos which loop through a set of sections forward and backward. If this option is enabled, VAST will export a series of images in which each section appears twice, first in forward order and then in backward order.

Screenshots:

If you want to export a stack of 'screenshots' how the images look in the main window of VAST, enable the checkbox 'Export screenshots as'. VAST will generate an image stack which reproduces the pattern, blending and tinting settings as they are currently set in the main window, as if you would take actual screenshots. Select a target image format and filename prefix / location.

[13]Note that the bounding box is not always correct, in particular if you delete parts of what was painted before, the bounding box will not shrink.

Screenshots currently only contain image and segmentation layers (not skeletons in annotation layers).

Image stack:
This saves a stack of images from the selected 'image volume' layer. You can specify the target format, filename prefix and location.

Segmentation stack:
This saves the segment IDs of the segmentation layer selected in 'Source layer:' or part of it as an image stack. For RGB or ARGB target formats, the segment ID of each pixel (a 16-bit number) will be encoded in the color of the pixels in the exported image. Bits 0-7 will be stored in the blue channel and bits 8-15 will be stored in the green channel. The red channel will stay 0.

You can also choose whether to export individual segment IDs or to use the folder ID for segments which are in collapsed folders (enable '`Collapse labels as in current view`'), disable saving of empty tiles ('`Don't save empty tiles`'), and restrict exported segments to the selected segment and children ('`Export currently selected label (and child tree) only`').

3D models target:
This function is currently disabled because it is not fully implemented yet. For now please use VastTools to export 3D models of segmentations (see section 8.2).

When you are done setting up the parameters for the export, press '`Export`' and VAST will start exporting the image stack or stacks (VAST can export more than one target at the same time).

Chapter 4

Segmentation Layers

Segmentation layers hold volumetric (voxel) segmentation data, which assigns a segment or label to each image pixel. Each segment has meta-data associated with it, which includes its color, name, and location in the image volume. Different from image layers, these segmentation layers are editable within VAST by painting and filling. Typically, segmentation layers are used to segment (split) an image stack into regions (segments) by labeling each region with a separate segment. VAST supports up to 65535 segments per segmentation layer (16-bit IDs).

After opening an image stack in VAST, you can add an empty segmentation layer simply by clicking the pencil icon in the toolbar to enter 'Paint Mode'. To add a new, separate segmentation layer, click '`Menu`' in the '`Layers`' tool window and choose '`Add New Segmentation Layer`'. Each segmentation layer holds a separate set of segment colors and is (or will be, when saved) associated with its own segmentation file. Opening an existing segmentation from its .vss/.vsseg file will also add a segmentation layer.

4.1 Painting

In VAST, segmentations can be painted as a colored overlay of the image data. When a stack of EM images is loaded, you can enter 'Paint Mode' by clicking the little pencil icon in the toolbar. When you start a new segmentation like this, VAST will ask you if you want to add 16 segments (label colors) to your segment list. Also, two floating tool windows will appear at the right side. The upper one, '`Drawing Properties`' (Figure 4.1), provides options for drawing, whereas the lower, '`Segment Colors`', lets you select and organize the segment labels and their colors in the segmentation.

When in paint mode, you can paint on top of the currently displayed image stack. Select a color (label number) from the '`Segment Colors`' window at the right by clicking on it. Then click the left mouse button where you want to paint in the image.[1] You will see the outline of your current tooltip as a circle. By clicking and dragging the mouse you can paint larger regions. All painting happens in the selected segmentation layer, which is blended over the EM image (the EM image itself will not be changed). You can use the '`Alpha:`' checkbox in the toolbar to switch the selected segmentation layer on and off, and the slider right of it to set its opacity.[2] Segment colors do not have to be solid colors, but can have patterns. Use the '`Pattern:`' checkbox in the toolbar to switch patterns on and off, and use the slider right of it to manipulate the contrast of the patterns.[3] If you enable the '`SelAlpha:`' checkbox, the opacity of the selected segment and its children will be

[1] Even though you can use VAST with a mouse, it is designed to be used with a pen tablet.

[2] You can also temporarily hide all segmentation layers by holding down the H key.

[3] Patterns were meant to provide more visually distinguishable colors/segments; however, in many use cases this is not necessary.

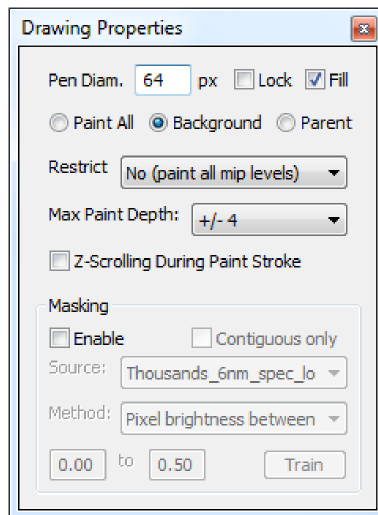

Figure 4.1: The Drawing Properties tool window

controlled separately by the SelAlpha slider. You can use this to highlight a particular segment or set of segments. You can also switch all image layers on and off, by clicking on the 'EM' checkbox in the toolbar. This is sometimes useful if you want to inspect just the segmentation. These controls are duplicated in the 'Layers' tool window for the selected layer.

You can change the size of the pen tooltip. The easiest way is, if you are using a pen tablet and VAST is properly configured, to hold down one of the pen buttons and to move the pen up or down on the screen. When using a mouse, you can hold down either the 'Tab' key or the '|\' key and click and drag up and down instead. You can also use the - and + keys on the keyboard. The current pen diameter (in pixels) is displayed in the Drawing Properties tool window. The third way of changing the tooltip size is to edit the 'Pen Diam.' text field in the tool window. You can also lock the current tooltip size if you don't want it to be changed accidentally, for example if the size of the tooltip is important for your data analysis, by switching on 'Lock'.

The checkbox 'Fill' next to it switches automatic filling of closed 2D contours on and off. If enabled, after each paint stroke VAST checks a rectangular area with the approximate extent of the stroke for empty closed contours of paint color and fills them with the paint color.

Below you can choose from 'Paint All', 'Background' and 'Parent'. This determines which voxels in the current segmentation are paintable. If you select 'Paint All', you will paint over or erase anything, no matter if it was painted before or not. If it is set to 'Background', previous paints will not change, but your paint will only be applied to voxels which have not yet been painted. If you erase, only the current paint color will be erased to empty (background). This is the most useful painting mode.

Instead of only affecting background pixels, 'Parent' mode will affect only pixels which have the color of the immediate parent of the current paint color (see below for a description of segment hierarchies). When erasing, voxels with the current paint color will be changed back to the parent color. This mode is only useful in special cases, in particular when re-labeling a previously painted area to a new color, for example when splitting a labeled object into a hierarchy of parts.

4.1.1 Multi-scale painting

A specialty of VAST is that it allows you to paint at different resolutions. In fact, VAST limits you to always paint at the currently displayed resolution. The advantage of this is that the amount of

data that has to be manipulated when you paint a stroke is limited by the window size and screen resolution. Otherwise, painting in very large volumes could easily lead to a situation in which the amount of data that has to be written for a paint stroke is much larger than what can be loaded in RAM at one time, which would cause all sorts of problems, including very slow painting. Also it does not make sense to paint at a resolution which is much higher than the screen resolution because mouse (or pen) precision is also limited. Finally, allowing low-resolution painting can save a lot of memory, since large objects can be painted coarsely.

In VAST, images are stored as a pyramid of mipmaps with reduced resolution using powers-of-two factors. Painting always happens at the resolution of the currently displayed mipmap. This means that you can change the resolution at which you are painting by zooming. A single segmentation can be composed of parts at different resolutions. For example it is possible to draw a rough outline of an object at a low resolution, and then to zoom in and correct the object's shape at a high resolution. VAST will automatically upscale and downscale the displayed segmentation as you zoom, but zooming will not change the painted segmentation. The segmentation is stored at the resolution at which it was painted. If you paint at a low resolution first and then correct at a high resolution, part of the low-resolution segmentation will be replaced by a high-resolution version. If you paint at a high resolution first and then correct at a low resolution, part of the high-resolution segmentation will be replaced by a low-resolution version, including pixels in the vicinity.[4]

Sometimes you might want to make sure that a painted segmentation has a certain resolution. You can enforce painting at only one resolution by restricting painting to a particular mipmap ('`Restrict`' in the `Drawing Properties` tool window). VAST will then enable painting only when the image stack is zoomed to display the selected mipmap.

4.1.2 Automatic Z-filling

The time that has to be spent to manually paint a segment in VAST depends largely on the number of 2D outlines that have to be drawn. Especially if you follow a process that runs vertically through the volume, you have to paint (almost) the same outline over and over again, for every slice. If you want to just get a rough outline of an object and you're not interested in a high precision of the boundary, you could increase the painting speed by a factor of n if you paint the outline only in every n-th slice, or paint n slices at a time. In the first case, you get gaps of $n-1$ slices between the painted outlines, in the second case it is hard to determine what you are actually painting because you can't see where your color goes in most of the slices.

VAST uses a third method. It supports automatic Z-filling of intermediate slices where the regions of the lower and the upper painted region overlap. It turns out that in most cases neuronal objects are *locally convex*. Exceptions are branches, for example when a spine neck runs very close to the dendritic shaft. Automatic z-filling will only fill in the volumes of the overlap between the specified painted regions, and in most cases (for convex objects) the filled regions will stay inside the segmented object.

Z-filling makes sense across a few slices only, because there will be no overlap if your object moves too much from slice to slice (runs at an oblique angle). Also, VAST has to load multiple slices in RAM to be able to fill in those slices. The maximal distance across which VAST lets you fill in depends on the size of the image cubes used. Currently the cubes are set to be 16^3 voxels large, and VAST allows you to fill in up to $+$-8 slices (because it makes sure that two layers of cubes are loaded at the time of painting). In the data sets we are using this is approximately as far as z-filling makes sense, and it speeds up painting by a factor of up to 8.

You can set how far the z-filling will reach by setting '`Max Paint Depth`' in the 'Drawing Properties' tool window. This value controls both the distance at which Z-filling occurs, and the stepping distance for navigating with S, X or PageUp, PageDown keys, to ensure gap-free painting.

[4] I have to do this because at the time of painting at a low resolution, not all higher-resolution images may be available in RAM (they may even be too large to be loaded in RAM).

Automatic Z-filling is only applied while painting, not when erasing. This makes it easier to correct what has been filled in in the case of non-convex neighborhoods. This also means that the best strategy to draw an object coarse-to-fine is to try to paint conservatively (try to stay within the object boundaries), and correct by adding paint rather than removing paint, because deleting has to be done in every slice individually.

'`Z-Scrolling During Paint Stroke`' is by default disabled to prevent painting errors when accidentally switching to the next slice before the paint stroke is finished. However, if you enable it, you can very quickly coarsely label a long neurite running through your stack vertically by scrolling through the stack while following the neurite with your pen – provided that loading of the image stack keeps up with the update rate of the screen. You can adjust the scroll speed with the mouse wheel if you enable '`Adjust Scroll Speed With Mouse Wheel While Z-Scrolling`' in the Preferences (see section 2.2.2).

4.1.3 Masking

The last section of the '`Drawing Properties`' tool window handles the settings for 'masking'. If you switch on masking by clicking the check box '`Enable`', only pixels will be painted for which the image in a selectable source layer fulfills certain criteria. You can choose from several methods which determine the paintable pixels depending on whether the brightness or color of the image pixel in the layer set under '`Source`' is in a certain range, which you can set. The value range for minimum and maximum brightness is [0..1]. For absolute brightness range modes and 'Trained Color' (see below), the source layer is restricted to be an image layer (not a segmentation layer).

If '`Contiguous only`' is checked, VAST will restrict painting to pixels which fulfill the 'masking' criteria and are topologically connected to the place where the pen is put down, by performing a masked flood filling operation. This is for example useful for semi-automatic segmentation with boundary maps (see below).

The method '`Use Trained Color`' lets you interactively set which color or brightness range to paint from examples. If you click the '`Train`' button, a small tool window will appear which lets you set and clear positive and negative examples from painted segments. This currently only uses the brightness / color statistics of single pixels and can not learn patterns or textures. To use this function, paint over some image area which you want to have labeled in one segment color, select that segment color and click '`Add Positive from Selected Segment`'. Then use a different segment color to label some image area which should be excluded from painting, make sure that segment color is selected, and click '`Add Negative from Selected Segment`'.[5] Then set the '`Method`' to '`Use trained color`'. Subsequently VAST will only paint over pixels which have brightness or color similar to the positive example pixels. If you add positive or negative examples several times, the new examples will be added to the set of already defined examples, until you clear the training data.[6]

4.1.4 Semi-automatic segmentation from boundary maps

Masked painting with a fixed brightness range as described above can be used to constrain painted areas by using a boundary map, so that the outline of painted objects is traced automatically.

A boundary map is a gray level image stack which, for each pixel, encodes the likelihood of the pixel being on a boundary as a brightness value. Such maps can be computed for example by using trained convolutional networks on the original EM stack and then loaded into VAST as a separate image layer. To use the boundary map for painting, enable masking with the '`Contiguous only`' option, and select the boundary map as source layer. Set the '`Method`' to '`EM pixel between`' and the brightness range so that boundary pixels are excluded from painting. Then increase the tooltip size so that it covers the cross-section of the to-be-segmented object. Then a single painting click

[5] VAST only scans the pixels in the selected segment color which are in the currently displayed slice and region.

[6] Internally VAST uses a 64^3 HSV cube for example storage and color space mip maps for generalization.

on the cross-section should label the object in that section with correct outline. If the boundary map is imperfect, spills can be easily corrected by manual deletion.

The boundary map layer does not have to be visible for this to work. You can hide it and just show the EM image layer, and still use the boundary map layer as source for masking.

4.1.5 Semi-automatic segmentation from an imperfect automatic segmentation

The 'Masking' section of the the Drawing Properties tool window also provides a mode in which an (imperfect) automatic segmentation can be used to help manual drawing. For this you have to open an automatic segmentation as a separate segmentation or image layer. In the 'Method:' drop-down list, select 'Autopick source color', and then select the automatic segmentation layer in the 'Source:' drop-down list. Then, every time you start a pen stroke, VAST will sample the color in the source layer (segment id or pixel color or gray level, depending on source layer type) where you put down the pen (the center of the pen tooltip) and subsequently restrict painting in the active target layer to pixels of exactly that color in the source layer. Enable 'Contiguous only' if the same source color occurs in several segments and you want to constrain painting to the clicked one.

So for example if the cross-section of the object which is painted is correctly segmented in the source layer, that cross-section will be copied from the source to the target layer perfectly if it is painted with a large enough tooltip which covers the whole cross-section, or the outline is followed roughly with the pen.

The option 'Autopick nonzero source' in the 'Method:' drop-down list does the same as above, but excludes the background (zero) from being picked as a masking color.

If 'Z-Scrolling During Paint Stroke' is enabled, you can very quickly draw vertically running processes with perfect outline if you follow the process with a large enough pen while scrolling through the slices. Splits in the source segmentation can be easily corrected by simply lifting the pen and putting it down again within the region of the second source color. If you set the 'Method' to 'Auto z-repick nonzero src', repicking will occur automatically for each section you scroll to. Mergers can be corrected by using different target colors for different parts of a source object and accurately drawing the boundary between the objects with the pen.

Using this mode for correcting an automatic segmentation or parts of it has the additional advantage that everything which is painted ends up in a separate layer, and one can be sure that everything in that layer has been looked at. Also the fact that this method works with source segmentations loaded as image layers allows for automatic segmentations to be used which have more than 64k segments. Arbitrary many segments can be converted to 24 bits by throwing away bits and loaded as an RGB image layer. 24 bits would make it very unlikely that two segments with originally different segment numbers, which were now converted to the same segment number, would be located very close to each other. In the rare case that this happens it could be treated in the same way as other mergers.

4.2 Filling

After a segmentation layer is loaded or generated in VAST, the Fill tool will be enabled. To use it, click the paint bucket icon in the tool bar ('Fill Mode'). The 'Filling Properties' tool window will appear.

The fill tool performs constrained three-dimensional filling operations on your data set from a seed point. This can for example be used for recoloring already segmented objects, for splitting merged objects, for supervoxel agglomeration from a source to a target layer, and for filling out complete objects on a reliable boundary map (see section 4.1.4 and below).

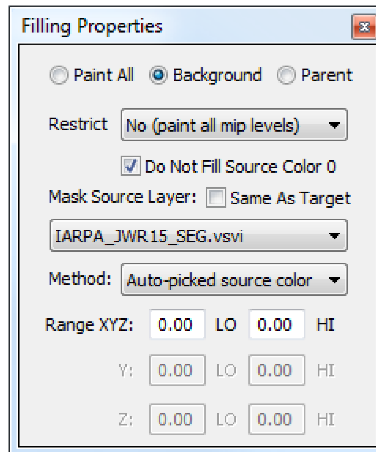

Figure 4.2: The Filling Properties tool window

VAST loads the parts of the data set which are reached by the filling operation block by block using the caching system, which means that this operation should also work on very large data sets with limited amounts of memory usage.[7]

4.2.1 Filling segments in the same layer

If the 'Mask Source Layer' option 'Same As Target' is checked, filling operations will be performed within the selected segmentation layer. The object which is clicked will be flood-filled with the paint color in 3D from the clicked seed point, *at the current mip level*. Just like painting, also filling operations are performed at the resolution of the currently viewed mip level. You can restrict filling to a particular mip level using the drop down menu next to 'Restrict'. This field is linked to the equivalent field in the 'Painting Properties' tool window. The 'Paint all', 'Background' and 'Parent' options are ignored in this mode, since 'Background' and 'Parent' modes would not work. Filling will always change the region of the color clicked to the paint color.

The filling operation uses a 6-neighborhood (fillable voxels at the six sides of a filled voxel will also be filled). If a source object consists of disconnected regions, only the region connected to the seed point will be filled. This can be used to split off parts of an object, for example if two dendrites are accidentally merged. To do this the two parts have to be disconnected topologically (for example by erasing the boundary between them in Paint mode, or by over-painting the to-be-split-off object with the target color, for example using Parent mode), and then filling one side with the target color.

The 'Background' color can also be used as a paint color, to erase objects.

'Do Not Fill Source Color 0' prevents from accidentally filling empty areas.

4.2.2 Filling segments from a source layer to a target layer

Similar to the semi-automatic segmentation functions described in sections 4.1.4 and 4.1.5, the fill tool can also be used to semi-automatically generate a proof-read segmentation in a target layer, by constraining filling by a separate source layer. This is a powerful function which can considerably speed up segmentation.

To use a separate source layer, uncheck the 'Mask Source Layer' option 'Same As Target' in the 'Filling Properties' tool window. This will enable additional filling options. First of all, the

[7]There is some overhead since the sides of active blocks are stored in memory for continuing the fill operation into adjacent blocks, but at no time does the whole dataset or object have to be stored in memory at once.

'Paint all', 'Background' and 'Parent' options are now accessible, which constrain painting to the target layer and can protect previously painted segments (see section 4.1). These are linked to the equivalent options in the 'Painting Properties' tool window.

Below 'Mask Source Layer' you can choose which layer should be used as source for a mask to constrain filling. 'Method' defines how the mask is derived from the source layer. The following methods are available:

- 'Pixel brightness between': Only pixels which have a brightness in the range defined in 'Range XYZ' in the source layer are fillable. The range values have to be between 0 and 1. This method can be used to perform filling which is constrained by a boundary map (see section 4.1.4).
- 'Pixel brightness XYZ': Same as 'Pixel brightness between', but different value ranges can be defined depending on fill direction (see below).
- 'Autopick source color': Every time the fill tool is applied, the color in the source layer at the seed point will be sampled. If the HI and LO values are 0, filling will be constrained to pixels which have the exact picked color in the source layer. Nonzero values of HI and LO allow filling of pixels with a source layer color in a range around the picked color. This method can for example be used to 'copy' segments from an (imperfect) segmentation layer to a proofread target segmentation (see section 4.1.5), or to copy segments from one segmentation layer to another.
- 'Autopick source XYZ': Same as 'Autopick source color', but different value ranges can be defined depending on fill direction (see below).
- 'Hybrid recolor / autopick': In this mode, if an existing segment in the target layer is clicked, it will be recolored as explained in section 4.2.1, and if an empty area is clicked it will be filled as in mode 'Autopick source color' above.

The XYZ options above can help in cases where a boundary map is used with an image stack which is imperfectly aligned or the Z resolution is low. If consecutive Z sections are shifted more than the width of the boundary, the fill color can 'leak out' into a neighboring segment from one section to the next. To prevent this, boundary maps can be blurred in each XY slice, and the pixel brightness range can be set much more conservatively for filling in Z than for filling in XY. This would let the fill color spread to the next section only where the pixels are far enough away from boundaries, so that leaks are prevented, but in each layer that is reached the fill color could still spread in XY much further to the actual boundary (credits go to Yaron Meirovitch for this idea).

4.3 Splitting

The split function in VAST allows to add planar split boundaries to existing segmented objects. For example, spines on a dendrite can be made separate objects by using this function to introduce a straight, minimal-cross-section split boundary in a child color between the dendrite and the spine, and then use the child color to fill the spine.

The split tool can be enabled by clicking the corresponding icon in the tool bar (box with diagonal line and arrow pointing to the right). Enabling it will open the 'Splitting Properties' tool window, in which some options can be set. The split function can currently work in two modes, selectable in the drop-down menu. Either the selected segment is used to paint on its immediate parent ('Use Selected Segment On Parent'), or each click adds a new child segment to the parent clicked, and uses it immediately for the split boundary ('Append numbered child'). In the second case a child segment name and appended number counter can be specified.

To use the splitting tool, click-and-drag at the location where you want to introduce a split within the parent segment, in the 2D view. Dragging will let you define the orientation of the split boundary in the 2D cross-section. When you release the mouse or pen, the split will be applied. At this point VAST will search through planes which intersect the line you specified in 2D, to find the angle at which the plane has the minimal (fully-contained) intersection area with the parent object, and then paste the child color into the parent segment along this plane to separate the two sides. One side can then be filled with the child color to make it a separate sub-object.[8]

While you are dragging the split line in 2D, you can hold down the 'Delete' key (typically mapped to a pen button) and release the mouse button or pen to cancel the split. The split line indicator will disappear when you do that. You can also hold down the 'Tab' or '\|' key to translate rather than rotate the split boundary (typically mapped to the other pen button). When in translation mode, the split indicator will appear in green.

A few caveats:

- As with other functions in VAST, the split boundary is drawn at the currently displayed mip resolution. Zoom to the desired resolution first, then introduce the split.
- The search area for the split boundary has a limited size, so this function will fail for very large cross-sections.

4.4 Segments

Segments (segment colors) in VAST are the components of a segmentation. Each segment has a unique ID number which is used to mark voxels as part of this segment in the segmentation images. It also has a number of additional parameters, in particular its color and pattern (see section 4.4.14), its name or label, its position in the segment hierarchy (see section 4.4.2), its anchor point (see section 4.4.7) and its bounding box. The segments which are defined in the currently selected segment layer are listed in the 'Segment Colors' tool window. Segments which have been used are marked with a * at the beginning of their name.

4.4.1 Picking segments

You can select the segment color to paint or fill with in the 'Segment Colors' tool window by clicking on it in the tree view. You can also pick any color you see in the segmentation layer by using the pipette tool. To do this hold down the SHIFT key and click on the segment you wish to select. This makes is very easy to switch between segment colors while painting. Alternatively you can use the 'Pick Segment Mode' which you can select in the main toolbar. If you have several segmentation layers, you can pick colors in a different segmentation layer as well, and VAST will automatically make that segmentation layer active. If segment colors in several layers overlap, VAST will cycle through the layers if you pick several times.

If you hover over segment colors in the main window when in picking mode, VAST will display the name of the segment as a tooltip. If the segment is in a different segmentation layer, the name will be shown in brackets together with the layer name.

4.4.2 The segment hierarchy

VAST can arrange segments in a tree-like hierarchy. This means that each segment can have other segments as children, which can themselves have children, and so on. VAST also allows you to

[8] Be careful when doing the filling – For example, if you fill a spine that you separated with a split plane, but the spine touches another spine, then the fill can fill one spine up, jump to the neighbor spine, fill that spine back down and end up filling the whole dendrite!

collapse and expand parts of the tree dynamically, which switches between a visualization which shows a whole branch of the tree in the same color or individual sub-branches in individual colors (see section 4.4.4). For example, if all spines of a spiny dendrite are labeled as sub-objects (children) of the dendritic shaft, one can instantly flip between a display in which the whole dendrite has the same color, or each spine has a different color, by opening and closing the dendritic shaft folder. Segments can also be used as folders to group segments, for example to classify labeled objects. You can use tags to mark certain segments as folders, to help external analysis (see section 4.4.13). The grouping can also be applied when segmentations are exported.

Segment hierarchies are visualized and edited in the '`Segment Colors`' tool window. This window uses a 'tree view control', similar to the navigation pane of a Windows Explorer window, which makes usage very intuitive. More functions related to the segments can be found in the tool window's menu, which opens either by clicking the 'Menu' button or right-clicking a segment in the tree.

4.4.3 Re-ordering and moving segments in the tree

To re-order the segments, simply drag and drop them with the mouse. You can only select and drag one segment at a time,[9] but if the segment has children the whole branch will be moved (including all children). Please note that to make a segment the first child of another segment, you have to drag it to the right side of the tool window, right of the new parent segment. The new parent segment will then be highlighted in blue, instead of the black line indicating the target space between two segments. You can move any segment, with exception of the 'Background' segment. The Background segment can also not have any children.

Because it can be cumbersome to move hundreds of items from one folder to another one by one, VAST has functions '`Make all siblings children`', '`Make all children siblings`', '`Collect by Name into Selected Folder`' and others which can help in certain situations (see section 4.4.8).

4.4.4 Collapsing and expanding tree branches

You can collapse and expand tree branches, which are displayed in the same way as folders and subfolders are in the Windows explorer, by clicking on the little '+' or '-' sign left of parent segments. When you collapse a folder, all its children will be displayed in the same color as the parent. If you pick a segment color from the segmentation layer by shift-clicking, and the selected segment is in a collapsed folder, the folder will be automatically expanded to show the native color of the segment you selected. In addition the context menu has two functions to set the collapse state of all subfolders of the selected segment at once, under '`Expand / Collapse Child Folders`'. If the 'Background' segment is selected, this will be applied to the complete segmentation layer. Also see section 4.4.13.

4.4.5 Adding new segments

The context menu of the '`Segment Colors`' tool window provides several functions to add more segments to the selected segment layer. You can add a segment as next sibling or as a child of the selected segment. '`Add 10 Segments`' will add 10 segments immediately after the selected segment. The most sophisticated way to add segments is '`Add Named Segments ...`', which lets you specify a naming scheme and add multiple named segments at the desired target location in the segment tree. VAST will attempt to guess a naming scheme from the name of the currently selected segment.

You can change the name (label) of any segment in the same way as file names are changed in the Windows Explorer – click a selected segment name a second time, then rename it. Segment names can be maximally 255 characters long.

[9]This is a limitation of the Windows Tree-view control which is used here.

4.4.6 Generating segments from an automatic segmentation and skeleton

You can generate segments automatically if you have an automatic segmentation and agglomerated objects with a skeleton, using a tool layer. This function allows you to 'render' the result of the agglomeration to editable segments, which you can then refine manually by voxel painting. You can find the function for this under 'Add Segments From Other Layers...' in the context menu of the 'Segment Colors' tool window. The transfer to segments is done by applying the 'Fill' function at each skeleton node, while using trans-layer masking by the automatic segmentation. You will have to specify at what mip level to perform the filling, whether to exclude fills if nodes are on background (source color 0), exclude fills at nodes where the 'exclude' flag is set, and whether or not to add a new segment color at each node which is filled rather than using a single segment per processed skeleton. Please refer to section 6.1 to learn more about how to use Tool Layers to agglomerate automatic segmentations.

4.4.7 Using anchor points and other coordinates

Each segment has an 'Anchor Point' stored with it. This is an XYZ coordinate vector which indicates the location of the segment in the image stack. Initially the anchor point is set to the location at which the segment is painted first. You can jump to the anchor point by right-clicking on a used segment in the 'Segment Colors' tool window and selecting 'Go to Anchor Point' from the context menu. You can quickly jump to the anchor point of the selected segment by pressing the 'Home' key or 'G' (for 'go to').[10] To set the anchor point of the selected segment to the current view location (as indicated by the center cross), select 'Set Anchor Point' from the context menu. You will have to confirm this action in a pop-up window to prevent accidental setting of anchor points.

Additional coordinate triplets can be stored in the label of each segment. To store the current view coordinates in the label of the selected segment, select 'Add Current Coordinates to Label' in the 'Segment Colors' tool window context menu. You can also copy-paste coordinates as text from the 'Coordinates' tool window or other sources. If those coordinate triplets are framed with round brackets ( ) in the segment label, you can jump to them easily using the context menu. Right-click the segment and choose 'Go to Coordinates in Label', then select the coordinates you want to jump to. VAST supports up to 16 coordinate triplets per segment label, as long as the segment label does not exceed the maximal length of 255 characters.

Newer .vss/.vsseg files also store the most recently painted location for each segment. Select 'Go to Last Painted Location' to go there.

4.4.8 Helper functions for arranging segments

Under 'Arrange' in the context menu you can find functions to move many segments at once. 'Move to Parent Level' moves the selected segment up in the tree and makes it a sibling of the folder it is in. 'Make All Siblings Children' will move all siblings of the selected segment into its folder (make them children of the selected segment). 'Make All Children Siblings' moves all children of the selected segment out of its folder and makes them siblings.

'Collect by Name into Selected Folder' can be used to collect segments into the (selected) target folder which have a name (label) which contains the specified substring. For example, if you tagged segments with certain properties (say, cell type; [P] for a pyramidal cell and [I] for an interneuron) you can use this function to quickly collect all cells of a certain type into a folder to display or analyze them together. Be careful with these functions because currently there is no 'Undo'.

[10] If you pressed the 'Home' key accidentally and want to go back to where you were, you can select the previous location from the drop-down menu in the 'Coordinates' tool window.

'`Sort Segments in Folder by ID (low to high)`' will sort the immediate children of the selected segment along their ID numbers. Any child trees of sorted segments will move with their parent but not get sorted internally.

4.4.9 Select recently selected segments

Under '`Select Recently Selected`' in the '`Segment Colors`' tool window's context menu you can find a list of the segments you had recently selected. You can click on one of the listed segments to select it again. You can use the keys '<' and '>' to jump backward and forwards through the list of recently selected segments.

4.4.10 Global operations: Deleting and welding segment subtrees

The context menu of the '`Segment Colors`' tool window provides two functions which change segment numbers in the whole segmentation layer, one to delete segments from the segmentation layer ('`Delete Segment + Subtree`') and one to weld all collapsed folders in a subtree to a single segment each (making them all the folder color, while removing the child segment numbers), '`Weld Collapsed Segments in Subtree`'. If the Background segment is selected, *all* collapsed folders in the segmentation layer will be welded to single segments.

Deleting and welding segments is actually more difficult than it seems because it involves traversing the whole segmentation data set and inspecting every single painted voxel. When you choose to delete the selected segment and its children, VAST will actually have to not only set all voxels with those segment numbers to 0, but also renumber all the other voxels so that the used segment numbers will have no 'gaps' (all segment numbers between 0 and n are used).

'Welding' will merge a set of collapsed folders in the subtree of the selected segment into single segments. For this VAST needs to renumber all voxels with segment numbers of children of collapsed folders in the selected segment subtree to the segment number of their collapsed folder, and also, similar to when deleting, renumber the other segment voxels so that the resulting segmentation is free of segment number gaps.

Because these functions change more or less the complete segmentation and VAST is designed so that the opened segmentation file is not changed, it basically has to copy almost all segmentation data to the temporary cache file. Depending on what segmentation you are working on this can take a lot of time, memory and disk space. Also *it will change the internal ID numbers of the segments*, so please think twice before using these functions in case you are relying on absolute segment IDs in your analysis. These functions are also not well tested; please report any bugs you might encounter.

4.4.11 Global operations: Reevaluating bounding boxes

For each segment VAST keeps a bounding box, which stores the minimal and maximal coordinates of pixels painted with this segment. This can be used to define a cutout region which contains the complete segment. VAST extends this bounding box automatically as a segment grows. However, if a segment shrinks, for example by manual erasing parts of it, the bounding box will not shrink because VAST does not know where else the segment color may be used in the segmentation. This means that bounding boxes can be significantly larger than the object itself, for example if the segment was painted/erased accidentally in the wrong place. For this VAST provides a function to reevaluate the bounding boxes. To use this, zoom the 2D view to the mip level you want to do the evaluation at and select the segment or segment subtree you want to reevaluate (segment and all its children). Then select the function '`Reevaluate Bounding Boxes in Subtree`' in the '`Segment Colors`' tool window context menu. The higher the resolution which you are using, the longer this process will take, since VAST has to go through the complete segmentation data at that resolution and find all voxels which have any of the selected segment colors.

4.4.12 Global operations: Segmentation cleaning

When painting segmentations manually, there can be left-over imperfections like holes and stray pixels (for example if part of a segment was incompletely erased). The same can occur in automatic segmentations. You can try to find these for example by using the 3D viewer, but correcting everything manually can be a lot of work. For this purpose VAST provides a function to perform segmentation cleaning. To use it, select `Clean Segments in Subtree ...`' in the '`Segment Colors`' tool window context menu. A parameter dialog window will open in which you can select the mip level at which to perform the cleaning, which segments to clean, and the cleaning region. Below you can specify the parameters for the cleaning operation itself. Cleaning is performed by loading overlapping parts of the segmentation at a time (a 'search window') and filling holes / removing debris within each windowed region.

'`Search Window Diameter (pixels)`' specifies the size of the search window. If the window is too small, large pieces of debris and large holes will be missed if they never completely fall within a window. Making the window large will increase its memory consumption.

'`Remove Debris up to Volume (pixels):`' lets you constrain the size of separate segment regions to be removed, to separate debris from objects which should be kept. If you uncheck this option, debris will not be removed.

'`Fill Holes`' can also be enabled/disabled, and a volume threshold for hole filling can be set as well.

Please note that with the current implementation of this function there may still remain some pixels and holes in higher-resolution mip levels, which are too small to show up in the mip level used for cleaning. However, if you export images or models at the cleaned resolution, debris and holes should have been properly removed.

4.4.13 Segment tags

Each segment can have a 'Tag' which you can select in the '`Segment Colors`' context menu under '`Tags`'. A tag is a number between 0 and 15 which can indicate the type of the segment. The tag value is exported with the segmentation metadata text file and can be used to help external analysis (bits 24-27 in the 'flags' field). By default the tag of all segments is 0. VAST uses tag 1 to indicate that the segment is a 'Folder Segment', which is not a segmented object by itself but rather a folder which contains other folders and objects. This information can be used to collapse all objects, but expand all folders – select '`Expand Only Segments Tagged as Folder`' under '`Expand / Collapse Child Folders`' in the '`Segment Colors`' tool window context menu.

Segments which have a tag that is not 0 will have an icon with a colored frame in the Segment Colors tree window. Folder segments have a gray frame.

4.4.14 Editing the color of a segment

The color and pattern of any segment can be changed. Right-click on a segment in the list and go to the sub-menu '`Colors`' of the context menu, then choose the desired option. Each segment has a primary color, a secondary color, and a pattern that is used to blend between them. You can also randomize the colors of all segments of the selected subtree (the selected segment and all its children) or the immediate children, or set the primary or secondary color (or pattern) of all segments of the selected subtree or the immediate children to the same color or to a color range.[11] VAST lets you specify two colors to define a color range from which the individual colors will be chosen randomly. The range is interpolated in the HSV color space (hue, saturation, value).

[11] If the Background segment is selected, these functions will apply to all segments in the segment layer (except the Background segment).

Please remember that by collapsing segment folders you can quickly and reversibly switch the displayed color of a segment to the color of the collapsed parent, without the need to change the color of all children individually.

4.4.15 Exporting volume measurements

Exporting volume measurements of segments, classically a function of the Matlab helper script `vasttools.m`, can now be done directly in VAST. Select '`Export Volume Measurements...`' in the menu of the '`Segment Colors`' tool window to do this. You'll have to specify at which resolution (mip level) and in which region to analyze the data, which segments to include, and where to store the results. The measurements can be exported as a plain text file or a comma-separated-values text file (select in drop-down menu in the file dialog under '`Browse...`').

4.4.16 Exporting segmentation metadata

The entry '`Save Segmentation Metadata ...`' in the '`Segment Colors`' tool window context menu lets you export the metadata of the segments to a text file, which you can then for example parse with MATLAB to extract colors, hierarchies, names, anchor points etc. of the segments for analysis. A Matlab script which can parse these files is included in the supplementary package (`scanvastcolorfile.m`). The format of the text file is described at the beginning of the text file itself:

`1`	ID number of the segment
`2`	Flags field of the segment as 32-bit value
`3-6`	Primary color as red, green, blue, pattern1
`7-10`	Secondary color as red, green, blue, pattern2 (unused)
`11-13`	XYZ coordinates of the segment's anchor point (in voxels)
`14-17`	IDs of parent, child, previous and next segment (0 if none)
`18`	If the segment is collapsed into a folder, this is the folder ID
`19-24`	Segment bounding box (may be incorrect if voxels were deleted)
`25`	Segment name as text in " "

Table 4.1: Columns of the segment metadata text file

The 'Flags' field uses the following bits: 0: isused; 8: isexpanded; 16,17: isselected (1: selected, 2: in child group of selected); 24-27: tag (0..15)

Alternatively, metadata can be saved to a comma-separated-value text file (.CSV); you can select the target file format in the drop-down list in the file dialog.

4.4.17 Loading segmentation metadata

Metadata saved to a `.txt` text file with the function described above can be loaded back to a segmentation layer by using '`Load Segmentation Metadata ...`' in the '`Segment Colors`' tool window context menu. This function can be used to restore a previous coloring or sorting of segments in a segmentation layer. However, loading will not work if there is a mismatch of the number of segments, and may cause problems/crashes in VAST if there are errors in the file (in case you edit it manually), though the parser checks for a few common problems.

4.4.18 Segment information

'`Segment Information ...`' in the context menu opens a window which shows you the internal information (metadata) associated with the selected segment. You can use this information to

count children of the segment, get its internal ID or other parameters. Getting the segment info of the 'Background' segment returns statistics for the whole segmentation layer. The text can be copy-pasted if needed.

4.4.19 Searching for a segment with a given name or ID

At the top of the 'Segment Colors' tool window there is an edit field which can be used to find segments. As you type or paste a text string into this field, VAST will select the next segment (after the selected segment in depth-first search order) the name of which contains the typed sub-string. The edit field is case-sensitive. If there is more than one segment which contains this sub-string, you can click on the magnifying glass right of the text edit field to go to the next segment that matches your text. The F3 key has the same function, provided that the segment tree sub-window of the 'Segment Colors' tool window is active.

You can also search for a segment with a particular ID. To do this, type an opening bracket [into the find edit field, followed by the ID number (internal segment number) you are looking for.

4.4.20 The 'Collect' tool

The 'Collect' tool ('Collect Segment Mode') can be selected in the toolbar by clicking the icon with an arrow pointing at a folder. When in Collect mode, segments you click will be moved in the segment tree to become children of the currently selected segment. This can be useful to quickly classify different objects into different types. Simply make a folder segment for each type, select it, and click on the objects in the image that are of that type with the 'Collect' tool. You can also use this to collect parts of objects in an automatic segmentation into one folder each and weld them (see section 4.4.10) to correct splits.

Using this tool is a bit dangerous because there is no Undo. If you click the wrong object it might be difficult to remember where it came from and there is currently no easy way to 'move it back'. Secondly, when an segment is moved, all its child segments are moved with it, but not its parent(s). So if you are dealing with objects which consist of several parts, make sure that you move the parent segment of the object and not only a sub-branch of its tree.

4.5 Saving Segmentations

Important: VAST DOES NOT SAVE AUTOMATICALLY while you paint. Your tracings will be held in RAM and/or a cache file on disk until you explicitly save them. If you open a segmentation from a .vss/.vsseg file and work on it, the file will not be changed unless you explicitly tell VAST to save the changes you made to the opened file by selecting 'Save Segmentation' from the main menu. If you want to keep the previous version and save to a new file, use 'Save Segmentation As ...' instead. VAST will then take all data from the opened file, the RAM cache, and the segmentation cache file, and combine them into a new file on disk.

We have had cases in which people had VAST open for several days without saving and lost a lot of work when the computer crashed. Please save your work once in a while.

In the newest versions of VAST the .vss/.vsseg file format was amended to accommodate new functions, which makes it not backwards-compatible with older versions of VAST (1.3 and earlier). However, you can open a new segmentation file in VAST 1.4 or later and save it in 'legacy format' which you can then open in older versions of VAST. To do this, use 'Save As' and select the 'VAST 1.2.1 legacy segmentation file' format in the drop-down list at the bottom of the file dialog.

VAST 1.4 and later also supports .vss/.vsseg files which are internally packed to reduce file size (possibly at the expense of read/write speed). In the 'Save As' dialog you can choose whether you want to save to a packed or unpacked .vss/.vsseg file in the drop-down list at the bottom of the window.

The 'Save Segmentation' and 'Save Segmentation As ...' functions in the main menu always save the segmentation layer which is selected in the 'Layers' tool window. Alternatively, to prevent confusion, you can right-click the layer you want to save in the 'Layers' tool window and choose 'Save Selected Segmentation Layer' or 'Save Selected Segmentation Layer As ...' there.

4.6 Segmentation Merging

If you have two or more (for example partial) segmentations (.VSS files) of the same image stack you can merge them into a single segmentation. For this, first open one of the segmentations you want to merge in VAST, and then select 'File / Import / Merge .VSS Segmentation Files in ...' from the main menu, and choose the .VSS file(s) you would like to merge in with the opened segmentation. During merging, VAST will add the selected .VSS files to the current segmentation. VAST lets you choose whether you want to change segment IDs of the merged-in data so that they do not overlap with existing segments, or keep segment IDs of the imported file (then only segments with new IDs will be added to the segment list, and segments with existing IDs will extend pre-existing regions). You can also choose whether to put new segments into a separate folder, and whether or not to overwrite pre-existing voxels or only write into empty voxels. You can select several .VSS files at once for merging, which will then be added one-by-one.

VAST does not save the merged segmentation automatically nor does it change any of the segmentation files. It will generate the merged segmentation in the segmentation cache, and if it does not fit in RAM it will end up in the temporary segmentation file. Please save the resulting merged segmentation if you want to keep it, for example via 'File / Save Segmentation As ...' from the main menu. Please make sure that there is enough space on the drive used for the disk cache.

4.7 Importing Segmentations From Image Stacks

VAST can import segmentations from image stacks, either generating a new segmentation or merging with a previously loaded segmentation. The imported segmentation will be added to the selected segmentation layer, or a new layer will be generated if no segmentation layer exists.

Segmentation source images can be PNG or TIFF files in 8-bit graylevel, 16-bit graylevel or 24-bit RGB format. Multi-image TIFF files are also supported.

If importing from 24-bit RGB images, just like in the image files generated during segmentation exporting (see section 3.3 below), the RGB values of the image pixels will be interpreted as segment numbers with the least significant 8 bits (0..7) in the blue channel, bits 8..15 in the green channel, and bits 16..23 in the red channel. Since VAST can currently only handle 16-bit segmentations, the red channel should always be 0. Because of the requirement to have a gap-free list of segment numbers in segmentations however, VAST will scan all images for existing segment numbers and renumber them as necessary if it finds gaps or if color channels are used differently than expected. This means that you can use any color mapping for the segments (as long as you don't exceed the 64k limit), and VAST will sort it out for you during importing, but the resulting segment IDs within VAST might differ from what you intended.

To import, select 'File / Import / Import Segmentation from Images ...' from the main menu. VAST will ask you to select one or several image files of the image stack you want to import. Then it will open a dialog where you can specify the parameters for segmentation importing.

File name(s):
You have to specify a name template for all image files in the stack. This has to be a *C++ format string* (as it is used by printf()). This means that numbers which specify the slice, column and

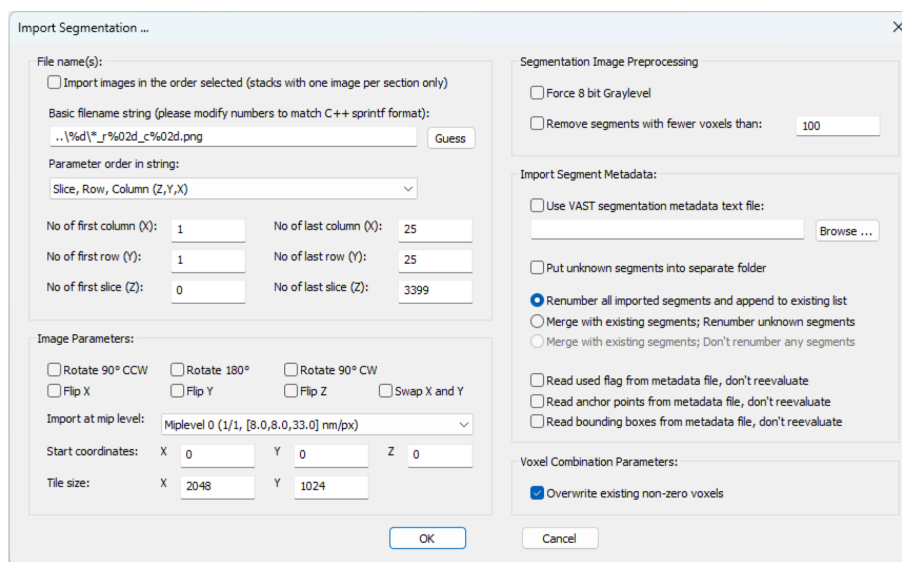

Figure 4.3: Import segmentation options

row coordinates in the file name have to be replaced by codes like '%d' (integer number) or '%04d' (integer number with zero-padding to 4 digits). VAST will then fill in those numbers for each image. For more information about format strings, please refer to a C++ manual or ask the internet. You also have to specify the order and the value limits for the parameters of your file name template, to define the range of names of all the images or image tiles in your stack.

For example, assume you have a single-tile image stack of 100 images in a single folder, and the images are named 'slice_s0000.png' to 'slice_s0099.png'. The file name template should then be set to 'slice_s%04d.png', because the slice counter is an integer number which is zero-padded to four digits. Also, in this case, you would set the 'Parameter order in string:' to 'Slice (Z)', since the only parameter in the template is the slice counter. To get the correct values for the slice counter set 'No of first slice (Z):' to 0 and 'No of last slice (Z):' to 99.

Figure 4.3 shows a more complicated example. Here, segmentation images are tiled and each section is stored in a separate subfolder. The dataset consists of 3400 sections, and each section is stored as 25x25 image tiles of 1024x1024 pixels each. The first tile of the first section is named as and located in .\0\slice1_w01_r01_c01.png, the last tile of the last section is .\3399\slice3400_w27_r25_c25.png. If you select a single image as reference, VAST will assume that directory as start directory, but you can use a relative path in the file name template to go to different directories. '..\%d\' in the file name template will go to the correct directory for each slice, and we use the slice counter to fill in this parameter. Next, the beginning of the file name repeats the section counter with an offset of 1, and also has a wafer parameter which we don't need for indexing. We can omit this part of the file name by using a wildcard (*). Lastly, the row and column parameters are replaced by '%02d' since they are zero-padded to two digits.

In simpler cases you can use the 'Guess' button to let VAST automatically construct the basic filename string.

In case you have a stack of images with only a single tile per section, check 'Import images in the order selected (stacks with one image per section only)' to skip this step completely.

Image Parameters:
Here you can rotate and flip the images if necessary, and tell VAST where to put them into the currently opened volume. The 'Tile Size' is extracted from the image file you selected, but you can

adjust it here too. The 'Start coordinates' are currently not allowed to be negative. Please crop your images prior to importing if necessary.

Segmentation Image Preprocessing:
The option '`Force 8 bit Graylevel`' tells VAST to interpret the images as single-channel 8 bit images, by only using the least-significant 8 bits of the image data. This can help in cases where the loaded images are supposed to be 8 bit graylevel images but are encoded or interpreted as RGB or RGBA color images.

If you enable '`Remove segments with fewer voxels than:`', you can clean up messy segmentations upon import by removing small debris segments. All segments with fewer voxels than the given limit will not be imported (and other segments will be renumbered to fill the gaps). If you use this function you currently cannot use a segmentation metadata text file since the size threshold may remove segments which would be needed for the segment hierarchy in the metadata file.

Import Segment Metadata:
If you have a segment metadata text file (.txt) for the imported segmentation stack (same format as the file written in section 4.4.16) you can provide it here. The options below specify how VAST should deal with the segments in the imported stack. If you are importing into a previously loaded segmentation, you can either keep the imported segments separate by renumbering and appending them to the existing segments, or merge the imported segments which have already existing segment IDs with the existing ones. New segments will be appended. If you use the option 'Merge with existing segments; Renumber unknown segments' together with a metadata file, the hierarchy in the metadata file will be ignored. The three options below let you choose whether you want to import the usage flags, anchor points, and bounding boxes as specified in the metadata file, or reevaluate them from the imported image data.

Voxel Combination Parameters:
If 'Overwrite existing non-zero voxels' is checked, imported non-zero voxels will overwrite existing non-zero voxels. If it is not checked, VAST will keep existing non-zero voxels and ignore imported non-zero voxels at the same location.

Similar to segmentation merging, VAST will not save the segmentation automatically after importing. Please use `File / Save Segmentation As ...`' from the main menu to save the result.

Chapter 5

Annotation Layers

In contrast to segmentation layers, annotation layers contain geometric objects rather than voxel data. Right now they can be used to create and modify *skeletons*. Skeletons in VAST are binary trees[1] with nodes and edges, which can be used to quickly annotate and navigate through complex objects, and to measure geometric properties like lengths, distances and branching angles.

5.1 Creating an Annotation Layer

Annotation layers can store annotation objects (skeletons) and object folders. Each annotation layer, except if it was generated new, is associated with a VAST annotation file (equivalent to segmentation layers and segmentation files). Annotation layers are also associated with a particular image volume and can only be used if an image volume of the expected size is loaded. To add a new annotation layer to the layers list, you can either click the 'Annotation Mode' button in the toolbar, or select 'Add New Annotation Layer' from the menu of the 'Layers' window.

When the annotation layer is selected in the 'Layers' window, the sliders allow to adjust the opacity of the annotation objects in that layer in the same way as for segmentation layers (allowing separate control for the selected annotation object if 'Sel Alpha' is enabled). The 'Scale' slider controls the width/diameter of the edges and nodes in the 2D window. Annotation layers are always rendered on top of image layers and segmentation layers, independent of the ordering in the 'Layers' window. However different annotation layers are rendered back-to-front in the order in the 'Layers' window.

Equivalent to segmentation layers, annotation layers can also be saved to and loaded from data files. They use the file extensions `.vsanno` and `.vsa`. Load and save functions can be found in the 'File' menu of the main menu in VAST. You can also save or save-as annotation layers in the context menu of the 'Layers' window.

5.2 The 'Annotation Objects' Tool Window

When you generate or open an annotation layer and go to 'Annotation Mode', the 'Annotation Objects' tool window will appear, which lists all annotation objects stored in that layer. You can click any of the annotation objects in the list to select them. You can change the name of annotation objects by clicking it twice (like in a Windows explorer window). You can also drag-and-drop annotation objects there, like segments in the 'Segment Colors' tool window.

[1] A binary tree is a directed, acyclic graph with a defined root node, where each node except the root node has a 'parent node', and each node can have up to two 'child nodes'. All nodes are connected by edges, and cycles are not allowed.

If you switch between different annotation layers, this window will switch its content, equivalent to the 'Segment Colors' tool window.

At the top of the 'Annotation Objects' window, there is a search field which you can use to find annotation objects with particular names in the list. To search for an annotation object ID, type '[' followed by the ID number.

The menu of the 'Annotation Objects' tool window provides the following functions related to annotation objects:

- 'Go to Root Node': Centers the 2D view on the root node of the selected skeleton object. Same as hold down 'Shift' and pressing 'Home' or 'G'
- 'Go to Selected Node': Centers the 2D view on the selected node of the selected skeleton object. Same as pressing 'Home' or 'G'
- 'Add New Folder': Adds a new folder object to the list of annotation objects
- 'Add New Skeleton': Adds a new skeleton object to the list of annotation objects
- 'Add Named Skeletons ...': Adds skeleton objects with defined names to the list of annotation objects
- 'Add Skeletons From Other Layers ...': Currently this allows you to copy anchor points of segments in a segmentation layer to skeleton nodes (one skeleton per segment, each with a single node at the location of the anchor point), and to copy annotation objects from other annotation layers, or just their root nodes.
- 'Delete Object': Deletes the selected annotation object
- 'Arrange / Move to Parent Level': Same as the equivalent function for Segment Colors; see section 4.4.8
- 'Arrange / Make All Siblings Children': Same as the equivalent function for Segment Colors; see section 4.4.8
- 'Arrange / Make All Children Siblings': Same as the equivalent function for Segment Colors; see section 4.4.8
- 'Arrange / Collect by Name Into Selected Folder': Same as the equivalent function for Segment Colors; see section 4.4.8
- 'Arrange / Sort Siblings by ID (low to high)': Same as the equivalent function for Segment Colors; see section 4.4.8
- 'Expand/Collapse Child Folders / Collapse All': Same as the equivalent function for Segment Colors; see section 4.4.4
- 'Expand/Collapse Child Folders / Collapse All in Subtree' Same as the equivalent function for Segment Colors; see section 4.4.4
- 'Colors / Random Color': Changes the color of the selected annotation object to a random color
- 'Colors / Change Color ...': Brings up the color selector window to change the color of the selected annotation object to a specific color
- 'Colors / Change Color of Borders ...': Sets the border color (typically white) of all annotation objects in the annotation layer to the specified color

- 'Colors / Subtree / Randomize all Colors of Subtree': Changes the colors of the annotation objects in the selected subtree to a color you choose
- 'Colors / Subtree / Make Node Colors of Subtree Unique': This function can be useful when using node colors of several skeletons to identify the skeleton. This function will make sure that the node colors of all selected skeletons (the skeletons of the selected subtree in the 'Annotation Objects' tool window) will be different (unique for each skeleton). If a color is duplicated, VAST will attempt to change node colors as little as possible to keep the general appearance.
- 'Colors / Subtree / Set all Colors of Subtree': Changes the colors of the annotation objects in the selected subtree to random colors
- 'Flags': Lets you switch flags for the selected annotation object on and off (see below)
- 'Set Object UserID ...': Allows to define one unsigned 64-bit integer value for each object
- 'Define Flag Types ...': See below, section 5.3.
- 'Define Edge Types ...': See below, section 5.4.4.
- 'Save Annotation Layer Data ...': Saves metadata of the objects in the current annotation layer to a .CSV file. This includes total lengths for skeletons. Use the drop-down in the file save dialog to select whether a header line explaining the table columns should be included.
- 'Load Annotation Layer Data ...': Loads metadata of the objects in the current annotation layer from a .CSV file in the same format as saved by the function above. You can select in a pop-up dialog which metadata to import (object hierarchy, names, colors, user flags and user IDs). Together with the function above this can be used to modify annotation layer metadata externally, without using the API.
- 'Save Skeleton Data ...': Saves node and edge information of all skeletons in the selected layer to a .CSV file. Use the drop-down in the file save dialog to select whether a header line explaining the table columns should be included.
- 'Copy Selected EXT List to Clipboard (non-excluded)': Used in conjunction with tool layers. The 'Chunked Region Collector' stores the source segment IDs in the EXT value of the respective skeleton nodes, and this function lets you export these values for the selected skeleton or all skeletons in the selected folder by copying them to the Windows clipboard. You can then paste (CTRL-V) the clipboard contents to another program like Neuroglancer or a text editor. The 'non-excluded' version only exports EXT values of nodes for which the 'exclude from agglomeration' flag is not set. The 'all non-zero' version exports all EXT values except 0.
- 'Annotation Layer Information ...': Displays statistics of the annotation layer.
- 'Annotation Object Information ...': Displays statistics of the selected annotation object. For folders, this includes statistics of the objects within the folder.
- 'Count Node Labels ...': Counts the number node labels in the selected skeleton which are equal to or contain a text string you can specify
- 'Node Tool': Allows for selection of nodes with specific properties and performing actions on them. Please see below, section 5.5.

5.3 Annotation Object Flags

For each annotation layer, VAST lets you specify up to 64 named flags which can be switched on and off for each annotation object. This can be for example useful to make individual annotation objects part of different, overlapping classes (for example 'Layer 4', 'Glia Cell' etc.).

To specify flag names, choose 'Define Flag Types ...' in the menu of the 'Annotation Objects' tool window. In the pop-up dialog, select one of 64 flags in the drop-down menu and then edit its name. If the name is not empty, it will be added to the active list of flags. To remove a flag from the active list, simply remove its name.

To switch flags for an annotation object on or off, select the object in the 'Annotation Objects' list and choose the flags in the context menu of the 'Annotation Objects' tool window under 'Flags'.

5.4 Skeletons

Skeletons in VAST are binary trees. Each skeleton has one root node. All nodes of a skeleton are connected by directed edges. Each node can have at most one parent (the edge leading towards the root) and two child edges (leading away from the root). If you hover the mouse over a node of the selected skeleton, its parent edge will be indicated with two thin white lines at the side, while the child1 edge will be indicated by a single thin white line in the middle.

5.4.1 Editing skeletons

In 'Annotation Mode' (second button in the toolbar) you can edit skeletons in the following ways:

- To add a new node to the selected skeleton, simply left-click anywhere in the dataset outside the nodes and edges of the selected skeleton. Click and drag for exact placement. The node will be connected with an edge to the previously selected node, except if it is the root node.
- Click any node of the selected skeleton to select it. Click and drag nodes to change their location. You can also drag nodes to different sections by z-scrolling while dragging them.
- Click and drag an edge of the selected skeleton to insert a node in that edge.
- Hold down the 'Delete' or the '‘~' key and click a node of the selected skeleton to delete it. Branch nodes (nodes with three edges on them) cannot be deleted. To delete a branch node, first delete one of its edges.
- Hold down the 'Delete' or the '‘~' key and click an edge of the selected skeleton to delete it. This will split the selected skeleton in two separate skeletons, and a new skeleton object for the child side will be generated in the annotation object list.
- To merge two skeletons into one, place a node of the target skeleton very close to a node of the skeleton you want to merge with it. Then right-click that node and choose 'Weld Nodes (Merge Trees)'. The unselected skeleton will become part of the selected skeleton by removing the overlapping node of the unselected skeleton and linking its children to the selected node of the selected skeleton. For this to work the selected node has to be a leaf node (no children) which is not the skeleton root node.
- Right-click a node of the selected skeleton for additional options of that node; see section 5.4.2 below.
- Right-click an edge of the selected skeleton if you want to set the edge type or weight, delete the edge or show edge information including the edge length.

- If you hold down 'Shift' you can hover over nodes and edges to see the name of the Skeleton they belong to. Clicking any node while holding down 'Shift' will select that node and the skeleton it belongs to. Clicking any edge while holding down 'Shift' will select the parent node of the edge and the skeleton it belongs to.
- Hold down the 'TAB' or '\|' key and then left-click a node and move the pen or mouse up/down to change its node diameter interactively. Alternatively, hold down the middle mouse button, then click left and drag up and down.

5.4.2 The node context menu

When you right-click a node of the selected skeleton either in the 2D view or in the schematic viewer, a context menu with additional options of that node opens, with the following functions:

- 'Set Node Label ...': This sets a text label for the node which will be displayed in the 2D view and Schematic Viewer. Node labels are also searchable (see below).
- 'Remove Node Label': Removes the node label from a node. This is not the same as adding a node label with an empty string.
- 'Set Node Diameter (..)': Each node has a diameter assigned to it, which can be set interactively (as described above) or by typing in a value here. Nonzero diameters are shown in the 2D view as a circle around the node, if it is selected.
- 'Find Next Node With Exact Label ...': Use this function to search for a node in the selected skeleton which has a specific node label. The key F3 can be used to 'search again' afterwards.
- 'Find Next Node With Partial Label ...': Use this function to search for a node in the selected skeleton where the node label contains a specific substring. The key F3 can be used to 'search again' afterwards.
- 'Find Next Node With EXT Value ...': Use this function to search for a node with the specified EXT value. The key F3 can be used to 'search again' afterwards. - EXT values are used in conjunction with Tool Layers; see section 6.1.
- 'Find Next Node Selected in Node Tool ...': The Node Tool (section 5.5) can be used to select nodes with specific features. Use this function to search for, and go to, nodes selected in the Node Tool.
- 'Find Again (F3)': For any of the searches above, this function or hitting F3 will 'search again'.
- 'Go to Root Node': Selects and centers the root node of the selected skeleton
- 'Select Node But Undo Move':
- 'Jump To Closest Neighbor': This option os enabled if a node of a different skeleton is close-by to the selected node. Use this to select that other skeleton and node
- 'Delete Node': Deletes the selected node, if it is deletable
- 'Swap Children': The two children of a node can be swapped with this function, so that child1 becomes child2 and vice versa.
- 'Make Root Node': The selected node becomes the root node of the tree, and all other nodes and edges are adjusted to form a normal binary tree

- 'Weld Nodes (Merge Trees)': Weld the selected node to a nearby node of a different skeleton. If welding succeeds, the other skeleton becomes part of the selected one, linked at the selected node. The nearby node is deleted in the process.
- 'Delete Parent Edge (Split Trees)': Splits the selected skeleton by deleting the parent edge of the selected node. The subtree containing the parent node and root remains the previous skeleton, now missing a branch; the cut branch is added as a separate skeleton to the annotation objects of the layer.
- 'Exclude from Agglomeration': Sets or clears the 'exclude' flag of the selected node. Used with tool layers; see section 6.1 below.
- 'Exclude all Nodes With This Ext': Same as above, but also excludes all other nodes on the selected skeleton which have the same EXT value stored. The context menu when right-clicking the 2D view in annotation mode, but not clicking a node, also provides this function; the EXT value used is then the source value as defined by the associated tool layer at the clicked location, if one is specified (see section 6.1 below).
- 'Include all Nodes With This Ext': Clears the 'exclude' flag for this node and all other nodes on the selected skeleton which have the same EXT value stored.
- 'Show Node Information ...': Shows information for the selected node like stored values, coordinates, distance to root, etc.
- 'Count Node Labels ...': Enter a text string and VAST will show you how many nodes in the selected skeleton have this node label, or have a label containing this text.

5.4.3 Quick navigation using skeletons

When in 'Annotation Mode' (second button in the toolbar), you can navigate around the data set using the selected skeleton as follows:

- Press 'Home' or 'G' to center the currently selected node of the selected skeleton
- Hold down 'Shift' and press 'Home' or 'G' to center and select the root node of the selected skeleton
- Press the left arrow key or 'Q' to select and slide the view to the parent node of the selected node
- Press the right arrow key or 'E' to select and slide the view to the first child node of the selected node
- Hold down 'Shift' and press the right arrow key to select and slide the view to the second child node of the selected node
- Hold down 'Control' and press the right or left arrow keys to slide to the next node (parent or child side) which either is a branch node or has a text label. Hold down both 'Control' and 'Shift' and press the right arrow key to start towards child2 instead of child1 (if it exists).
- You can jump to a close-by node of a different skeleton (select that skeleton and node) by right-clicking the selected node and choosing 'Jump To Closest Neighbor' from the context menu. This can for example be used to jump across synapses from one neuron to another, if the dendrites and axons are annotated with skeletons.

- You can traverse the skeleton tree in depth-first order, stopping at nodes with a specific label or EXT value. To do this, right-click the node you want to start from and select 'Find Next Node With' with the option you want to use. Pressing `F3` will search with the same settings again. More advanced ways to select nodes for traversal are provided by the Node Tool (section 5.5).

5.4.4 Edge types

Edges of skeletons can have one of 16 edge types, which can be selected in the context menu when right-clicking the edge. VAST pre-specifies names and edge patterns, but you can edit these in the menu of the 'Annotation Objects' tool window under 'Define Edge Types ...'. The dialog window lets you choose edge types in the drop-down list and edit their names and pattern. These settings are stored together with the annotation data in the `.vsanno` / `.vsa` files.

5.5 The Node Tool

You can open the 'node tool' from the context menu of the 'Annotation Objects' tool window.

The node tool allows you to select a specific subset of nodes on one or several skeletons of the selected annotation layer. You can then perform different operations on just the selected nodes, and/or use the 'Find next node selected in node tool' function to traverse the skeleton by jumping to just the selected nodes.

To add nodes to the selection, first set the 'From:' and 'Add/Remove Nodes:' options, then click the 'Add' button. An additional pop-up dialog may appear depending on the options chosen.

To remove nodes from the selection, first set the 'From:' and 'Add/Remove Nodes:' options, then click the 'Remove' button. An additional pop-up dialog may appear depending on the options chosen.

The 'Clear' button unselects all nodes.

To perform a function on the selected nodes, set what you want to do in the 'Function:' field first, then click the 'Execute ...' button.

Here are a few use cases for the node tool:

- Assume you want to make sure you complete the skeletonization of a neuron, which means that all branches should either end or exit the volume. Assume that you use node labels 'end' and 'exit' to indicate the end nodes of completed branches. You can now use the node tool to find all branch end points which still need to be completed. To do this, first add all leaf nodes in the selected skeleton (nodes without children; so ends of your skeleton) to the selected nodes. Then you remove all nodes which have a label containing 'end' or a label containing 'exit' from the selection. Afterwards only nodes where the skeleton needs to be extended are selected. You can then jump directly to these nodes, one by one, using the 'find' function in the node context menu as described above.

- Let's say you marked the location of thousands of synapses on the skeleton of a neuron with the node label 'syn'. Now you decide you would like to have all synapses on the dendritic subtree labeled 'insyn' and all synapses on the axonal tree labeled 'outsyn'. You can do this with the node tool. First select the base node of the axon. Then, in the node tool, set 'Add/Remove Nodes' to 'from nodes in selected subtree' and 'Nodes with exact name ...', click 'Add' and add all nodes named 'syn'. This should only select nodes named 'syn' on the axon. Then set 'Function' to 'Replace text in node labels' and click the 'Execute ...' button. In the 'Replace Text' dialog set 'Find:' to 'syn' and 'Replace With:' to 'outsyn' and click the 'OK' button. Use a similar procedure to rename just the 'syn' nodes on the dendritic subtree.

- Or say you would like to export a list of coordinates of all nodes which you tagged 'Synapse', to run an external analysis of the shape of the synapse cloud. Use the 'Add' function to select

all nodes with label containing 'Synapse', then choose the function 'Save node coordinates to file' and press 'Execute ...'. In the file dialog drop-down selector at the bottom, choose 'Save as type:' to 'Comma-separated values file without header' (or with header, if you prefer), and save.

- You can also visualize the location of all nodes selected in the node tool in a 3D rendering, by placing an instance of a 3D object (say, a sphere) at each location, by using the 'Export Particle Clouds (3D Object Instancing)' function of VastTools (see section 8.3).

5.6 The Annotation Schematics Window

Under 'Window / Annotation Schematics' you can enable the 'Annotation Schematics' window. Alternatively you can set a main window panel to show the annotation schematics ('Schematics Viewer'). This window shows the tree of the selected skeleton in a schematic, zoom- and pannable 2D format. To pan, simply click and drag, and to zoom use the mouse wheel. You can click nodes in the Annotation Schematics window to select the node and to move the VAST 2D window to the location of that node. You can also right-click nodes to access the node context menu (see section 5.4.2) and use key presses to navigate through the tree as described above.

Under 'Options' in the menu of the Annotation Schematics window you can switch between collapsed and expanded view ('Collapse'). In collapsed view, only nodes are shown which are end nodes, branch nodes, or nodes with text labels. You can also reset the view or move to the selected node or root node, and enable or disable node following ('Follow Active Node'), which moves the field of view of the Annotation Schematics window so that the selected node is always visible.

The schematic viewer can be extremely helpful when navigating large and complicated skeletons, especially when node labels are used to mark nodes of interest.

5.7 Exporting and Importing Annotation Data

Annotation data can be exported to `.CSV` text files. In the menu of the 'Annotation Objects' tool window, choose 'Save Annotation Layer Data ...' to save the metadata of the objects in the current annotation layer to a .CSV file. Use the drop-down in the file save dialog to select whether a header line explaining the table columns should be included. To import annotation layer metadata, choose 'Save Annotation Layer Data ...'. You can select in a pop-up dialog which metadata to import (object hierarchy, names, colors, user flags and user IDs). Together with the save function this can be used to modify annotation layer metadata externally, without using the API.

Choose 'Save Skeleton Data ...' to save node and edge information of all skeletons in the layer to a .CSV file. Use the drop-down in the file save dialog to select whether a header line explaining the table columns should be included. To import skeleton data, use the import function (see section 5.8 below).

The included Matlab script `vasttools.m` can also export skeletons to model files (`OBJ/MTL`, `PLY` and `SWC`), which can then be used for visualization; see section 8.5.

Annotation data can also be exported (and imported) via the VAST API, see section A.3.

5.8 Importing Skeletons from Files

Under 'Import / Import Skeletons from Files' you can import skeleton data from `CSV` and `SWC` text files to the currently selected annotation layer.

`CSV` files are required to be in the same format as exported from VAST (see section 5.2).

`SWC` files are expected to be at micrometer scale with a positive Z axis, as exported by `vasttools.m` without swapping X/Y axes and not inverting the Z axis (see section 8.5).

Chapter 6

Tool Layers

Tool layers are a new layer type for functions which take information from other layers and generate a derived result on the fly, which is also voxel data. Currently, tool layers only host a single function called 'Chunked Region Collector', which can be used to aggregate supervoxels of a typically automatically generated segmentation into larger objects by means of a skeleton (proofreading, correcting splits).

6.1 Chunked Region Collector

The 'chunked region collector' is a tool layer function which allows to use skeletons from an annotation layer to collect supervoxels in an oversplit source segmentation (loaded as an image layer) to generate voxelized objects which merge all selected parts. Whenever a skeleton node is put down, the source segmentation at that location is sampled (at the selected mip level) and added to the list of source IDs stored with the skeleton. The tool layer uses the list of source IDs for each skeleton to build a translation table from source segment IDs to target colors, defined by the skeleton colors. This table is then used to generate the tool layer output voxel data layer from the source segmentation layer on the fly.

The information for the agglomeration translation is stored in the EXT values of the skeleton nodes, and saved with the annotation layer file. If you want to apply the same skeletons to a new auto-segmentation dataset, you can force VAST to re-sample the source values at all nodes by checking 'Refresh Source' in the chunked region collector options dialog.

To use the 'chunked region collector' functionality, first open an image layer in VAST which contains a voxelized segmentation. Then add an annotation layer, either a new one or load one from a file. Next, in the Menu of the `Layers` tool window, select '`Add New Tool Layer`'. This will add a tool layer to the layers list (with a cogwheel icon) and show a dialog to configure the 'chunked region collector'.

- 'Image Volume Source Layer': Select the layer which contains the automatic segmentation you loaded in the drop-down menu. This layer will serve as source for the parts of the object you collect with the skeleton.
- 'Segmentation Source Layer': This is currently not fully supported, please leave as 'None'.
- 'Collector Source Layer': Select the annotation layer which shall contain the skeletons which specify which parts to collect for each object in the drop-down menu.
- 'Collect at Mip Level': Set this to the mip level at which you will be working, or leave as 'Auto' to always sample at the currently displayed mip level.

- 'Chunk Size': The chunked region collector allows for different translation tables in different parts of the volume. If you want to use this feature, set the desired chunk size here, otherwise leave 'unlimited'. Using chunking can help with very large datasets because segmentation IDs can be reused in different parts of the dataset, and it may also reduce the memory and computational overhead of the translation table(s). However it has the disadvantage that you'll have to collect all parts of large objects if they span several chunks.

- 'Output Format': This is the format of the output voxel data of the tool layer, which can be either RGB or RGBA. RGB is slightly faster and uses less memory than RGBA but does not allow for Alpha/SelAlpha functionality of the agglomerated objects. Choose RGBA if you need that functionality.

- 'Refresh Source': Source segment IDs are stored in the nodes of the skeletons in the annotation layer and saved and loaded with the annotation layer file. These IDs should be correct if the same source segmentation (image volume source layer) is used. However, if you want to change to a different source segmentation, you can ask VAST to re-sample all source values here.

You can also access this dialog at a later time from the Layers tool window under 'Menu / Edit Tool Layer ...' when the tool layer is selected.

Finding the responsible skeleton for an agglomerated region

If the tool layer is selected and you are in Annotation Mode (where you can edit skeletons), you can shift-click an agglomerated object in the 2D slice view to select the skeleton which is responsible for the agglomeration.

Excluding and including specific supervoxels of an agglomerated object

In case the auto-segmentation has large mergers and you want to exclude a particular supervoxel that you have included with the agglomerated object, first make sure that you are in Annotation Mode with the skeleton responsible selected. Then right-click the object in the 2D view and select '`Exclude ID from Agglomeration in Selected Skeleton`' from the context menu to exclude that supervoxel from the agglomerated object. This will set the exclude-flag in all nodes of the skeleton which overlap this source segment ID.
Select '`Include ID for Agglomeration in Selected Skeleton`' to undo that operation.

Generating an editable segmentation from a tool layer agglomeration

You can copy the result of a tool layer agglomeration to a segmentation layer, where it can then be edited by voxel painting. To do this, open a segmentation layer or add a new one. If needed click the pen icon in the toolbar to open the 'Segment Colors' tool window. Then, choose 'Add Segments From Other Layers ...' from the menu of the 'Segment Colors' tool window. In the dialog set the 'Source Layer' to the tool layer you want to copy from. Choose which agglomerated objects you want to copy to the segmentation layer. 'Generate What:' has to be 'Translayer filling at skeleton nodes ('Source Layer' has to be a tool layer)'. Click the OK button. A second dialog will appear where you can set additional options. The copy procedure involves executing a segmentation fill operation at each skeleton node, masked by the agglomerated object in the tool layer. Importantly, 'Fill at mip level' will determine at which mip level the fill operations are performed. A higher resolution (lower number of the chosen mip level) will result in a more detailed segmentation, at higher memory and filling time cost. Click the 'OK' button to start the filling. Save the segmentation layer to a file if you are happy with the result.

Exporting agglomerated objects as voxel data (image stacks)

The tool layer outputs an RGB or RGBA image stack which uses the colors of the skeletons for each agglomerated object. The output colors can be used as object IDs if they are unique (no two objects have exactly the same color). You can enforce this in the annotation layer if you put all skeletons involved into one folder, select the folder, and select 'Colors / Subtree / Make Node Colors of Subtree Unique' in the 'Annotation Objects' tool window menu. VAST will adjust the colors slightly so that all skeletons in that folder have different colors. Set the 'Alpha' value of the tool layer to 100%, disable 'Sel Alpha' and hide all other layers (for example by enabling 'Solo'). Then use the 'File / Export ...' function to export the agglomerated segmentation as 'Screenshots' image stack.

Exporting agglomerated objects as 3D models

3D models can be exported via the 'vasttools.m' script in Matlab (see section 8.2). To export 3D models of tool-layer-agglomerated objects, set up the 2D view as described above, making sure that all output colors are unique and only the tool layer is shown at 100% Alpha. Then in the 3D object export dialog in VastTools, select 'Isosurfaces from screenshots, one object per color' under 'Export what:' and export as usual.

Chapter 7

The 3D Viewer

VAST contains a simple 3D viewer which lets you view a region of your image stack in 3D. It uses a volumetric texture with transparency, rather than surface meshes of objects. This allows for volumetric rendering of any image data, not just segmented objects.

To open the 3D viewer, select 'Window / 3D Viewer' in the main menu of VAST.

The 3D viewer uses the image content of the 2D view. Once the image in the 2D window is set up, select 'View / Update' in the menu of the 3D Viewer window to load the data into graphics card memory. The block shown will be more or less centered around where the current 2D view is located (the center cross location), and use the current 2D resolution (mip level). In addition, if available, the currently selected skeleton(s) in the selected annotation layer are also shown, as lines.

Assume for example you have an EM stack and a segmentation loaded in VAST, and you want to see just one segmented object with its children in 3D. For this, adjust the 2D view so that only that segmented object is shown and everything else is black: Switch off the EM layer and 'Alpha', enable and maximize 'SelAlpha', and select the parent segment of the segments you want to show up in 3D. Center the 2D view to the segment of interest and zoom to the desired resolution. Then, in the 3D Viewer window, make sure that 'Nonzero Opaque' is selected under 'Transparency' in the window menu, and select 'View / Update' to load just your segmented object(s) into the 3D viewer. An easier way to achieve this is by using the 'Screenshot Rendering, Segmentations Only' mode (see below).

The transparency in the 3D view is derived from the pixel brightness. You can choose the mode under 'Transparency' in the window menu:

- 'Opaque': All voxels will be rendered fully opaque, so you will only see the surface of the volume block.
- 'Nonzero Opaque': Only fully black voxels will be transparent, all other voxels will be fully opaque. Best for rendering segmentations.
- 'Show Dark': The darker a voxel is, the more opaque it is rendered: (alpha=255-max(r,g,b))
- 'Show Bright': The brighter a voxel is, the more opaque it is rendered: (alpha=max(r,g,b))
- 'Max Projection': Finds and displays the maximal value for R, G and B (separately) along each view ray. Voxel transparency values are ignored.

- 'Min Projection': Finds and displays the minimal value for R, G and B (separately) along each view ray. Voxel transparency values are ignored.

Except for 'Max Projection' and 'Min Projection', to see the effect of the selection, update the data by selecting 'View / Update' in the menu of the 3D Viewer window.

If you enable the option Transparency / Surface Only and update, all interior voxels of segmented objects will show up black in the 3D viewer. In combination with the transparency setting Show Bright this can be used to show segmented objects as approximated surface bubbles rather than solid objects, which can be useful for example if one wants to show where organelles are located inside cells.

Under 'Size' in the menu you can choose from six different sizes for the volumetric texture or set it to a custom size. The larger the texture is, the more detailed will the 3D model be, but it will also require more memory on the graphics card and update and render slower.

You can zoom the 3D view in and out with the mouse wheel and rotate it in all three axes by dragging with the mouse (the mouse cursor shows which rotation axes will be affected, depending on where the mouse cursor is in the window). Choose 'View / Reset Orientation' from the 3D viewer window menu to set the view back to the orientation of the 2D view. You can translate the camera by using keys Up, Down, Left, Right, Page Up and Page Down. To reset the camera, click on 'View / Reset Camera' in the 3D viewer window menu.

You can have the 3D viewer remember the zoom level and color/transparency settings and decouple them from the 2D window view. The stored settings will be used whenever you update the 3D viewer data to a new location or selection of segments. To hold the current settings, check 'Lock Blend Params', 'Lock Center Coords' and/or 'Lock Mip Level' under View in the menu of the 3D viewer window.

To save the current 3D scene to an image, select 'View / Save Screenshot ...' from the 3D viewer menu. You can save the screenshot as PNG, (uncompressed) TIFF, BMP, or JPEG image file. Under 'View / Save Animation ...' you can save simple rotation and rocking animations as image series. Use 'View / Set Window Size' before saving to generate images with a particular XY resolution.

Further options available in the menu of the 3D viewer include changing the background color, displaying a line cube around the data block, and showing the current location of the VAST 2D view as cross-hairs. The latter allows you to use the 3D viewer as an 'overview map' when working in the 2D stack.

7.1 Using the 3D viewer to set the 2D view location

The 3D viewer can be used to set the 2D view to the same location that was clicked in the 3D scene. To do this, hold down the SHIFT key and click on an object in the 3D scene. VAST will compute the coordinates of the closest point of the 3D scene at the clicked location which is more opaque than a threshold, and set the view in the 2D window to those coordinates. The 2D view will update as soon as it is selected as active window. The threshold can be set to an opacity of 0.1, 0.5 or 0.9 (0 meaning transparent and 1 opaque) under 'Picking' in the menu of the 3D window.

7.2 Other 3D Viewer Modes

7.2.1 Screenshot Rendering, Segmentations Only

Under 'View / Mode' in the menu of the 3D viewer window you can choose different render modes. The default 'Screenshot Rendering' is described above. 'Screenshot Rendering, Segmentations Only' uses the same rendering engine, but only loads segmentation layers (no image layers). Selected skeletons can still be displayed as well.

7.2.2 'Raymarcher 2' and 'Raymarcher 2 Tool'

These modes use a different render engine which is still under development, but is usable. The size of the displayed volume is currently fixed, but this viewer allows for interactive zooming and panning. The data will update at changes of mip level when zooming, or after panning.

To rotate the cube of data, simply click and drag with the mouse (same as in 'Screenshot Rendering' mode). Mouse click location defines whether the rotation occurs as orbiting around the center or rotating around the center view axis.

To zoom, use the mouse wheel. The 3D viewer will switch to new mip levels automatically.

To pan, hold down control key and click and drag with the mouse. Dragging moves in the plane defined by the side of the cube you click on. When holding down the control key, a grid will show the plane in which you will move the data when clicking and dragging with the mouse.

To center a specific location, hold down the control key and click the location. The depth is determined by the first solid object encountered behind the mouse location. This is different from 'Screenshot Rendering' mode, where `SHIFT` is used instead of `CONTROL`. To see the center location in the 3D viewer, switch on crosshairs ('View / Show Crosshairs' in the menu).

Zooming, panning and location change are linked to the 2D window; if you move and zoom in the 3D viewer, the 2D view will move and zoom accordingly, and vice versa.

This mode shows the selected skeleton(s) of the selected annotation layer as nodes and edges. Nodes and/or edges can be switched off in the menu under 'View / Annotations'; skeletons can also be rendered white to make them stand out better (perform a small pan to trigger the update).

Please remember that this mode is still under development. Please use with caution; some functions may not always work as expected.

Chapter 8

The VastTools Matlab Toolbox

VAST includes an API through which it can communicate directly with client programs, using a TCP/IP connection. The Matlab script VastTools, which is included with VAST Lite, uses this API to provide users with a number of additional tools for VAST, including target lists, 3D surface exporting and measurement functions. It can be found in the `vast_package.zip` file. Since VastTools is a Matlab script, users can extend the interface with their own functions. Documentation of the VAST API is provided in Appendix A.

8.1 Getting started with the VastTools Matlab Toolbox

To start the toolbox, open `vasttools.m` in Matlab and run it. If Matlab asks, change the current directory. A small window should pop up with a menu, message field and cancel button.

Before you can use any of the functions of the toolbox you have to connect VastTools to an instance of VAST which is currently running. First, in VAST, you must enable the Remote Control API Server. Select 'Window / Remote Control API Server' in the main menu of VAST. In the tool window which then opens, enable a TCP/IP port for communication by clicking the 'Enable' check box in the upper left corner. If you are running VAST and VastTools on the same computer, you can use the standard settings (IP `127.0.0.1` and port 22081 on both the VAST and Matlab side). If you are using VastTools a lot and do not want to enable the API every time manually, you can set a flag in the Preferences to have it enabled automatically when VAST starts (see section 2.2.2, 'API Settings').

Then, in VastTools, connect to VAST by selecting 'Connect / Connect to VAST' in the main menu of the VastTools window. If the menu item 'Connect / Connect to VAST' changes to 'Disconnect / Disconnect from VAST' you are connected. The message log in the VAST Remote Control API Server window will also show when a remote connection has been accepted. In case you are using a different IP or port, for example because VAST and VastTools run on separate computers, you can set the IP address and port to use under 'Connect / Connection Options' in the VastTools main menu.

Once connected, you can use the different functions of the VastTools toolbox by selecting from the main menu of the VastTools window in Matlab.

Currently VastTools is not always locking the selected layer, view, appearance etc. in VAST during exporting, but many exporting functions use these settings. This means that if you change settings in VAST while exporting through VastTools you may get unintended results. To prevent that it is better not to continue working in the source instance of VAST while VastTools is exporting.

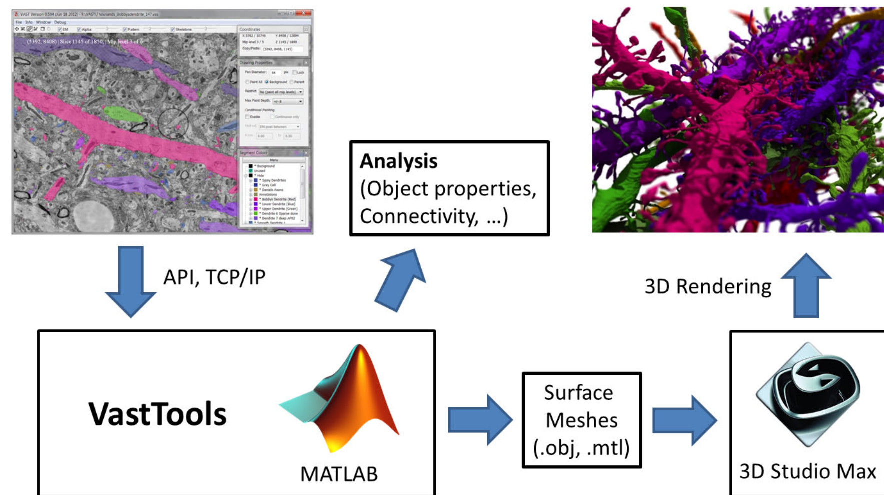

Figure 8.1: Exporting from VAST using VastTools

8.2 Exporting 3D Models

3D model exporting generates surface meshes of the painted voxelized segmentation in VAST (or from image or tool layers), and saves the resulting meshes in Wavefront OBJ files which are widely supported by 3D rendering and animation programs. VastTools uses Matlab's `isosurface` function to generate the meshes. For large export regions, it will generate surface meshes in parts of the volume to limit memory usage, and then stitch the mesh pieces together to generate the final objects. The exporter also saves a .MTL file for each OBJ file which defines the material properties (color) as defined in VAST. You can also measure the surface area of all exported objects and save the results to a text file.

To use this function you have to first run `vasttools.m` in Matlab and connect to VAST, as described above. Then select 'Export / Export 3D Objects as Mesh Files ...' from the main menu in VastTools. A dialog window will pop up in which you can specify all the parameters for the export. Once you click 'OK' the exporting will start and run until finished or until you press 'Cancel' in the main VastTools window. This will export objects of the *selected segmentation layer* (or use screenshots).

The following parameters for the export can be set:

Render at resolution:
Here you can select the mip level at which the surfaces should be generated. Lower mip levels will result in more detailed models, but also generate larger model files with more triangles and take longer to process. Models and surface area computations will be automatically scaled to compensate.

Use every nth slice:
If you want to reduce the resolution in Z you can do so by skipping slices. If you set this value to 2, every second slice will be used, if you set it to 3, every third, and so on. Models and surface area computations will be automatically scaled to compensate.

Render from area:
This specifies the region of the current volume in which you want to generate surface models. By default the region is set to cover the whole volume, but you can change the values to cover a smaller region if needed. These values are always specified at full resolution, no matter which mip level is set in the 'Render at resolution' selector. -
One Caveat here: VastTools remembers these settings while it is running. However if you close VAST and open it again and load a different data set and reconnect to VastTools, these values are not adjusted; please make sure you set the correct export region.

'Set to full':
Push this button to set the area boundaries to the full extent of the data set opened in VAST.

'Set to selected bbox':
This sets the render area to the combined bounding box of the segment selected in VAST and all its children. CAUTION: The segment bounding boxes are currently not always correct. For example, if you erase voxels you previously painted in VAST, the bounding box will not shrink.

'Set to current voxel':
This button sets the render area to a single voxel, the current center voxel set in VAST. Use together with 'Extend to current voxel' to define a target area by its corner points.

'Extend to current voxel':
This button extends the boundaries of the render area to include the current center voxel set in VAST. Use together with 'Set to current voxel' to define a target area by its corner points.

'Mip region constraint':
For objects which sparsely span a large volume, like the dendritic and axonal trees of a neuron, exporting based on a bounding box can take a long time. Most volumetric regions within the bounding box are empty but nonetheless have to be loaded into Matlab and analyzed because it is not known where parts of the object are located. The 'mip region constraint' can significantly speed up the exporting in such cases. The general idea is that VastTools checks at a much lower resolution where parts of the object are located, and then only processes those regions at full export resolution. This can of course go wrong if your exported object is so thin or small that it does not show up in lower-resolution mip maps (because of voxel subsampling), so please use this function with caution. If parts of your object are missed, increase the resolution (lower the mip level used) or add padding ('with padding (pixels):') to also load and process regions nearby to where the exported object was found at the lower resolution. Search regions are expanded by the value in the padding field, the unit of which is number of pixels at the resolution set in the 'Mip region constraint:' field.

Voxel size (full res):
These fields will be filled in automatically by the values provided by VAST. The voxel size is used to scale your models correctly. You can override the voxel size manually by entering different values into the edit fields.

Scale models by:
Additional scaling factors for your models. By default these factors are set to 0.001 (1/1000) in all directions to convert the units from nanometers to micrometers. It may be useful to scale down models even more for some 3D software (I found in particular Blender to struggle with large coordinate values).

Model output offset:
By default the models will be placed so that the upper left corner of the data set loaded in VAST is at the origin (0,0,0) of the 3D model space. This means that even if you change the export resolution or restrict the export to a smaller region under 'Render from area', the models will still be placed correctly within the global coordinate system of the data set, and models exported from different regions of the same data set will be placed correctly with respect to each other. You can however provide a constant offset here if you want to move the models somewhere else.

Processing block size:
As mentioned above, models are exported in smaller blocks and then merged together. The values here specify the block size. A smaller block size will reduce memory consumption and may increase the speed of isosurface computation, but will increase the processing time needed for glueing model parts together. Changing the block size should only have an effect on memory consumption and processing speed, not on the exported models. I recommend using the default values.

Erode/Dilate voxel data by 1 before exporting (segmentation only):
This can help with removing small dust-like object parts of the exported mesh if the segmentation painting is messy. If enabled, segmentation data will be shrunk by one pixel, followed by an expansion by one pixel, before computing the mesh. This will remove solitary voxels completely.

Export what:
Here you can select whether you want to export based on segmentations, or on screenshots.

For segmentations, you can choose to export models of all segments, or of a selected branch, and you can choose whether to export all segments individually or glue them together as they are currently displayed in VAST (by selecting a parent segment, and collapsing and expanding folders in VAST). This information will be read from VAST when you press OK, so you can make adjustments in VAST while the VastTools export dialog is open.

Exporting from screenshots allows you to generate surface meshes from image stacks directly, without having to segment objects first. This also works for exporting surface meshes from tool layers. You can use the layer mixing and brightness/contrast settings in the VAST 'Layers' tool window to adjust the result. These functions can also be used to export onion-like shells of fluorescence signals from confocal image stacks, to render realistic-looking 3D fluorescence clouds by using surface-normal-dependent transparency and additive blending, as was done in some of the figures in [8].

Different options are available for converting the pixel brightness to one or several surface meshes. Use 'Isosurfaces from screenshots, one object per color' to export 3D models of objects which were agglomerated with a tool layer; see section 6.1.

File name prefix:
All OBJ file names, and also the names of the exported objects, will start with this prefix string.

Object Colors:
Here you can select between 'Object colors from VAST' (the default) and 'Object volumes as JET colormap'. The latter only works if you previously computed object volumes (under 'Measure / Measure Segment Volumes ...').

Target folder:
All generated files (OBJ/MTL or PLY) will be stored to this target folder. Use the 'Browse...' button to select a different folder.

File Format:
You can choose here whether to export models as OBJ and MTL files, or as PLY files (another standard mesh file format).

Include Vast folder names in file names:
If enabled, the VAST folder names of the segment hierarchy will be added to all OBJ file names, and also the names of the exported objects. This makes it easy to select and process subsets of the objects based on their names. However, please keep in mind that the length of file names can be limited, depending on your file system, and names which are too long can lead to problems.

Invert Z axis:
In enabled, the models will be mirrored so that they lie below the $z = 0$ plane. This reflects the actual shape of the objects in the tissue, if slices are counted up as the are cut off the surface of a block.

Write 3dsMax bulk loader script to folder:
If enabled, a small file called '`loadallobj_here.ms`' will be saved to the target directory. This is a 3dsMax script which, if executed in 3dsMax, will batch-load all objects in the same directory. This is very useful if you are working with many object files.

Close surface sides:
If enabled, meshes will be closed at the sides where they exit the extraction region.

Skip model file generation:
Enable this option if you just want to measure model surface area and not generate OBJ files.

Disable network warnings:
For some datasets, like data stored on a Google Brainmaps server, the server sometimes returns an error message rather than an image even though the request was correct. In VAST this leads to an error pop-up which can be disabled. If this option is checked, VastTools will disable these error messages so that the export will not be interrupted should they occur. This will not affect the exported model quality since VAST will repeat the data request in case it receives error messages.

Save surface statistics to file:
If this is enabled, the export script will also compute the surface area of each exported object (by summing up the triangle surface areas) and save the result to the text file with the provided name. This file will always be stored in the same folder as the OBJ files (the target folder), so please give only a file name in the text field, not a file name with target path. Computing the surface area will take additional time, so only enable it if you need it.

8.3 Export Particle Clouds (3D Object Instancing)

This function generates compound 3D mesh models from a reference 3D object (for example, a model of a vesicle) which is placed at different locations, as defined by a segmentation or annotation in VAST. For segmentations, source locations can be either anchor points of objects or the center points of separated painted regions (either handled in each XY section separately, or volumetrically in XYZ). For annotations they can be skeleton nodes, or nodes selected in the Node Tool (see section 5.5).

With this function it is for example possible to make vesicle clouds with spherical vesicle meshes, based on manually annotated vesicles where each vesicle is labeled by a separate dot of paint. The export function places a copy (instance) of the vesicle object in the center of each painted dot. This function can also be used to indicate 3D locations of particular importance with different 3D object glyphs.

Many of the parameters in the 'Export Particle Clouds (3D Object Instancing) ...' dialog are equivalent to 3D object exporting (see above). The following parameters are specific to this export function:

Coords from:
This option defines which segment colors in the selected segmentation layer, or which objects in the selected annotation layer, should be used to define target coordinates.

Coords mode:
Select here whether for each segment, you want to extract the centroid locations of all painted regions in each XY section separately (2D, 4-connectivity), use volumetrically connected region centroids (3D, 6-connectivity), or simply use the anchor point locations of the segments. In the latter case the source resolution, nth slice, extraction area, processing block size, and max region diameter parameters are ignored. Alternatively, you can use the coordinates of skeleton nodes from the selected annotation layer or nodes selected in the Node Tool.

Max region diameter at selected mip level:
If connected regions are extracted from the segmentation voxel data, these values tell VastTools how many additional rows and columns of voxels have to be loaded at the sides of the processing blocks to make sure that complete regions are found. Ideally, the painted markers for the target locations should be small so that no large overhead is needed. These parameters only apply if the 'Coords mode' uses segmented regions.

Source model OBJ:
Click 'Browse' to open a file browser where you can select a source 3D model (.obj file) to be placed at the target locations. A 40nm vesicle model (at micron scale) is included with VastTools.

If 'Recenter Model' is checked, the model will be translated so that its center, defined by the average of its vertex coordinates, is at (0,0,0). 'Rescale source model' lets you specify an isometric scaling factor for the source model.

File Format:
Here you can select whether to save the results into many OBJ files or whether to concatenate the output into a single OBJ file.

Save coordinates:
If this option is enabled, the locations of the placed 3D objects and the volumes of their regions will be saved to the specified text file. This information can be used to count the objects and to analyze their locations.

8.4 Export 3D Boxes, Planes and Scale Bars

VastTools can export 3D box models in dataset coordinates, which you can use to show regions of the data set in 3D renderings. This means that boxes exported with this function should show up in the correct location and scale with respect to exported 3D models when loaded into a 3D render

program. To use this function select 'Export / Export 3D Box ...' from the VastTools menu. A dialog window will open with many of the same settings as in the object exporting dialog (see above for a detailed description).

Three box styles are supported:

- `'Wireframe'`: This will output a 3D box frame mesh with box color and wireframe width as defined. Click the 'Set' button to change the wireframe color.
- `'Solid-color surfaces'`: This is used to generate a six-sided box, colored in a single color (set under 'Box Color')
- `'Textured surfaces (VAST screenshot images)'`: Use this to generate a six-sided box with correct textures on its sides, taken as screenshots from VAST's 2D window. This function will output three files, an .OBJ file specifying the box coordinates, a .MTL file to specify its material with reference to the surface texture, and the surface texture itself as a .PNG file. The surface texture image contains the box surface arranged in a way so that you can print it, cut it out, and make a paper box model of your dataset if you like; however, you'll have to make sure yourself that the texture resolution is isometric. Please use a mip level appropriate for the size of your box to avoid generating an unnecessarily large texture image file.

If the thickness of the box in one dimension is zero (same min and max value under 'Box coordinates'), VAST will export an axis-parallel plane rather than a box. This works for solid-color and textured surface modes.

Scale bar 3D models can be exported under 'Export / Export 3D Scale Bar ...'. You can specify a scale bar length, width, orientation, and color. This function (as well as all other model export functions) uses the data set scale as it is set under 'Info / Volume properties ...' in the VAST main menu. Please make sure that the scaling is correct before exporting scale bars and other models.

If you export 3D mesh models with standard scaling, and the voxel size is set correctly in VAST, the models should be exported at micrometer scale (a distance of 1 in the model file should be exactly 1 micrometer). This means you can also simply add a box to your 3D render scene to use as a scale bar. Simply set its length to the length of the scale bar you want to add, in micrometers.

8.5 Exporting Skeleton 3D Models

This function exports skeletons of the selected annotation layer as nodes and edges (not meshes) to model files. The parameters in the export dialog are very similar to other export functions (see descriptions above).

Currently supported file formats are `OBJ/MTL` (Wavefront OBJ) files, either as individual files per skeleton or concatenated to a single file, as `PLY` (Stanford Triangle) files, and `SWC` files (ASCII, Standardized neuron morphology format). SWC files do not store object colors.

`SWC` files can be imported back into VAST (see section 5.8). For them to appear in the right location, make sure that 'Invert Z axis' and 'Swap X and Y axes' are unchecked when exporting. For use in 3D Studio Max, please check both options.

8.6 Exporting Projection Images

The 'Export Projection Image' tool allows you to generate 2D projection images in which your microscopic image stacks and/or segmentations are projected along a cardinal axis. This is for example useful to generate 'renderings' of your segmentation or a Z-projection of part of a confocal light microscopic image stack. You can also use these projection images as 'Simple Navigator Images', as described in section 8.8. The following parameters can be set in the export dialog:

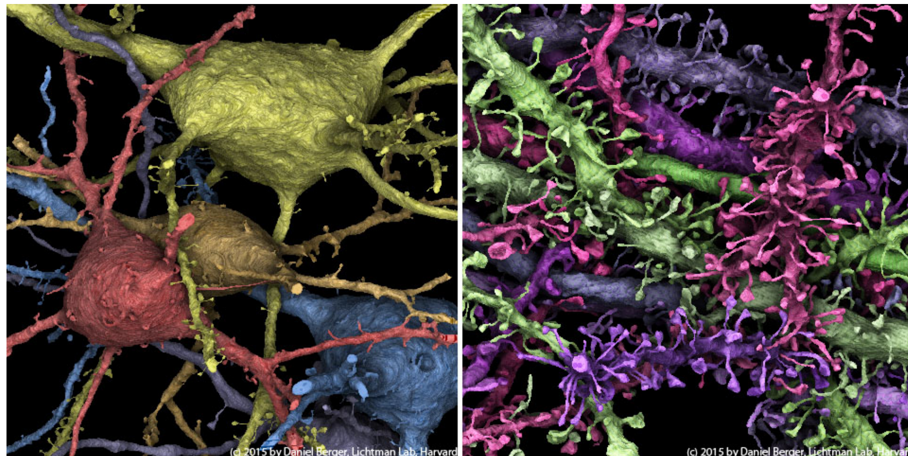

Figure 8.2: Example stack projection images with simulated shadows, generated with VastTools in Matlab

Render at resolution:
Sets the mip level at which the projection image is rendered. Higher mip levels generate smaller images at lower resolution.

Use every nth slice:
To reduce the resolution in Z (and speed up rendering of very large stacks) you can skip slices if you set this to an integer value larger than 1.

Render from area:
This determines the XYZ block in the image volume from which the image will be generated. You can use the buttons 'Set to current voxel' and 'Extend to current voxel' together with moving the center cross in the VAST 2D window to different locations to help you set up a source area. 'Set to full' sets the area to the complete volume, and 'Set to selected bbox' sets it to the combined bounding box of the selected segment and its children. CAUTION: The bounding boxes associated with each segment in VAST are currently not always correct. For example, if you erase voxels you previously painted in VAST, the bounding box will not shrink.

Projection axis, stretching:
Select here along which cardinal axis you would like to project and from what side (which side should be in front in the image), and whether the resulting image should be stretched in Z (for projections along the X or Y axis) if the slice thickness is different from the pixel size in XY. The stretching will be determined by the voxel size of your image stack as set in VAST (see section 3.2.4).

File name, Target folder:
Enter the target file name and location here. The projection image will be saved there after generation if 'Save to file' is enabled.

Segmentation preprocessing:
Select here how you want to use the segmentation for the projection image. This determines which parts of the segmentation are used, and for the 'Screenshots' image source whether the segments should be colored as displayed in VAST (collapsed) or in their individual native colors (uncollapsed).

Expand segments by n pixels:
Makes your segmented regions larger. Useful when using the segmentation as a mask.

Blur edges by n pixels:
Blurs the edges of the segmentation to smooth out the projection image.

Image source:
Select here what you want to use as source images for the projection image. Select 'Screenshots' if you want to use images as they are shown in VAST (using all brightness and contrast settings, layer blending and tinting, segmentation transparency, patterns, etc). A faster option is to just use the primary colors of either the selected or all segmentation layers.

Background color:
Color shown behind the image stack where the stack is transparent

Opacity source:
Determines which parts of each image plane should be rendered opaque for the projection image. Choose to render only segmented areas, only unsegmented areas, or all (Constant). You can also use the pixel brightness of screenshots to define opacity, and either render bright or dark pixels opaque.

Object opacity [0..1]:
Multiplier for the opacity of areas selected in 'Opacity source'. Use a value between 0 (transparent) to 1 (fully opaque).

Blending mode:
Select here whether you want to alpha-blend images with the selected opacity map, add all images, or do a maximum or minimum value projection.

Use shadows, Shadow cone angle:
Enable the check box if you want to include (fake) shadows in the rendering. The shadow cone angle determines how much the shadow spreads out (by means of an image blur filter) from slice to slice.

Depth attenuation (far brightness) [0..1]:
If you set this to values below 1, slices which are further away in the stack will be darkened. This can support the impression of depth in the resulting projection image.

Normalize projection image:
If enabled, the brightness of the final projection image will be adapted so that it uses the whole brightness range (without changing color hue).

Disable network warnings:
For some datasets, like data stored on a Google Brainmaps server, the server sometimes returns an error message rather than an image even though the request was correct. In VAST this leads to an error pop-up which can be disabled. If this option is checked, VastTools will disable these error messages so that the export will not be interrupted should they occur. This will not affect the exported image quality since VAST will repeat the data request in case it receives error messages.

Save individual slices:
If this option is checked, the output is saved as individual slices rather than a single rendered images. This option is useful for rendering animations with a parallax effect (using external software).

8.7 Measuring

VastTools provides several functions to measure objects labeled in VAST.

8.7.1 Measure Segment Volumes

This function can count the number of voxels of different segmented objects in a source area, and save the results to a text file. Parameters in the setup dialog of this function are similar to those of exporting 3D models and measuring surface sizes (see section 8.2 above). Please make sure that the voxel size you use is correct and that you cover the complete area you want to measure to get correct results.

8.7.2 Measure Masked Regions

This function can be used to measure the mean brightness, histograms of brightness, or statistics of segmentation IDs of all pixels in an image or segmentation layer which are underneath regions painted in a separate segmentation layer and to export the results to a .CSV or text file. Similar to other export functions, you can set a volumetric area and a resolution (mip level) at which to perform the analysis.

Masking Layer:
Select here which segmentation layer should be used as a mask.

Source Layer:
Select here which layer should be analyzed in the regions defined by the masking layer.

Mask preprocessing:
Here you can select which segments in the masking layer to use as mask regions for each measurement.

Source preprocessing (seg):
Here you can select which segments in the source layer should be analyzed. This is only used if the source layer is a segmentation layer.

Measure what:
If the source layer is an image layer, select here whether you want to export mean brightness or brightness histograms for each segmented region.

Currently image layers which are either 8-bit (gray level) or 24-bit (RGB) are supported. For 24-bit RGB image data, you can either export means for all three color channels (per segment region), or extract histograms for one of the color channels.

If the source layer is a segmentation layer, you can choose here whether the result should be exported as a full matrix or as a sparse matrix.

8.7.3 Measure Skeleton Lengths

This function saves a list of the total lengths of all skeletons in the selected annotation layer to a .CSV or text file. Measuring lengths in VAST works by using skeletons. You can place a skeleton down the main axis of a painted voxel object, and then measure and export its length by using this function.
You can also view the total length of any skeleton by selecting 'Annotation Object Information...' in the 'Annotation Objects' tool window menu in VAST. Also, total lengths of all skeletons are exported with other information by the 'Save Annotation Layer Data ...' function in the 'Annotation Objects' tool window menu in VAST.

8.7.4 Measure Segment Surface Area

Since this function relies on a surface mesh to estimate the surface area, it is part of the 'Export 3D Surfaces as OBJ Files' function (see section 8.2). You do not have to generate model files if you just want to measure surface area (check 'Skip model file generation' in the parameter setup dialog). Please make sure that the voxel size you use is correct and that you cover the complete area you want to measure to get correct results.

8.7.5 Euclidian Distance Measurement Tool

This will open a dialog in which you can make simple 3D distance measurements between points in VAST.

First, go to a location in VAST (use the center cross). Then, in VastTools, click the 'Get' button next to the first coordinate to read the current location from VAST. Then go to the other location and click 'Get' next to the second coordinate. The distance between the two locations will show up as a distance in voxels, and in nanometers.

You can easily jump back to the first and second coordinate in VAST by pressing the 'GO!' button.

Make sure you are using the right voxel size to get accurate nanometer measurements. Click 'Update' to read the current voxel size from VAST. To change the voxel size, please set the voxel size in the .vsv file used in VAST. To do this go to 'Info / Volume Properties ...' in VAST and change the voxel size (see also section 3.2.4). Then click 'Update' in the VastTools Euclidian Distance Measurement Tool.

You can copy the values in the edit fields in the Euclidian Distance Measurement Tool to paste them into different programs, but editing these values directly will not work.

Alternatively, you can use a skeleton edge directly in VAST to measure Euclidian distances. Place two nodes at the desired end points and then right-click the edge between them. The edge length (which is the Euclidian distance between the end points) will be shown at the bottom of the context menu.

8.8 Simple Navigator Images

Simple Navigator Images provide a means to navigate in an image stack using a projection image. After you rendered a projection image in VastTools using 'Export / Export Projection Image ...' (see section 8.6), you can generate a clickable Simple Navigator Image from it. Select 'Navigate / New Simple Navigator Image From Last Projection Image ...'. A new window should pop up which shows the last projection image together with a menu and toolbar. Use the arrow tool to click on a segment anywhere in the projection image to have VAST move to that location in your data set. The magnifying glass tool and the hand tool can be used to zoom and pan the image respectively.

Use 'File / Save Simple Navigator Image ...' to save this Simple Navigator Image to a file which you can open again later. You can have several Simple Navigator Images open at the same time, for example projection images along several axes, and navigate using all of them.

Please note that this functionality has been superseded by the 3D Viewer, which can also be used to navigate in the image stack (see section 7.1).

8.9 Target Lists

Target lists can store the current view coordinates, zoom level and selected segment together with comments. Click the '`Add current VAST location`' button to add the current view in VAST to the target list. To move the view in VAST back to a stored location, click on the 'GO!' field of the row in the table. You'll see the change when you make VAST the active window. If you want to update the coordinates, zoom level and/or selected segment number of one of the rows in the target list to the current state in VAST, select it and click the '`Update First Selected`' button.

The '`Update Zoom`' and '`Update Segnr`' checkboxes specify whether zoom level and selected segment number should be transmitted to VAST when you press 'GO!', and also whether they will be updated in the list if you press the '`Update First Selected`' button.

Functions to delete, rearrange and add separator lines to the list are provided in the Edit menu. You can save target lists to a file and load them back later. You can have several target lists open at once and cut/copy/paste between them. You can also select the list, or part of the list, and copy (Ctrl-C) and paste (Ctrl-V) the contents into other programs, like a text editor or Excel.

There is now also a function to import a series of coordinates (and zoom levels) into a target list from a Matlab matrix, under '`Edit / Import Coordinates From Matlab Matrix`'.

Target lists are stored as simple .mat Matlab files, so you can generate them yourself if you fill a .mat file with the appropriate variables. Simply load a target list file into Matlab to see which variables it contains.

8.10 Extras

8.10.1 Export Neuroglancer Link ...

This function currently only supports the human H01 dataset, and specifically the C3 automatic segmentation. If segments of the C3 segmentation are agglomerated with a skeleton using a tool layer, this function can be used to generate a Neuroglancer URL which is saved to a text file and can be opened in a browser window (by copy-pasting it into the address bar) to show the same set of segments. The selected object or folder of objects on the selected annotation layer is used as a source for the C3 segmentation IDs.

VSVI files for the H01 EM stack and C3 segmentation are part of the supplementary package which is included inside the VAST executable; see section 2.3.

Appendix A

API Function Reference

VAST includes an API through which external programs can communicate with VAST. The communication is done through a TCP/IP connection. This makes it possible to write client programs in any programming language which supports the TCP/IP protocol. It also allows client programs to run on separate computers and communicate through the network.

VAST comes with a client-side implementation in Matlab, which is used by the VastTools toolbox and included in the `vast_package.zip` file (see section 2.3).

In Matlab the VAST API is implemented as a class, `VASTControlClass.m`. Internally it uses the `jtcp` library for TCP/IP communication in Matlab. The different API functions are implemented as class methods.

Using the VAST API is very simple. With `VASTControlClass.m` and its helper functions available (in the path), simply make an instance of the class and use its functions. Here is a short example which connects to VAST, reads out some basic information, and disconnects:

```
vast=VASTControlClass();
res=vast.connect('127.0.0.1',22081,1000);
if (res==0)
  warndlg('Connecting to VAST at 127.0.0.1, port 22081 failed.','Error');
else
  vinfo=vast.getinfo();
  disp(vinfo);
  vast.disconnect();
end;
```

You can also use VastTools to handle the connection. Simply run VastTools and connect, then run your own script. VastTools uses a global variable '`vdata`'. To access the API functions, simply enable access to the global variable, then use the VAST API like so:

```
  global vdata;
  vinfo=vdata.vast.getinfo();
  disp(vinfo);
```

The following API functions are currently available (Version 4, as returned by `getapiversion`):

A.1 General Functions

A.1.1 res = connect(host, port, timeout)

Tries to connect to VAST via TCP/IP. 'host' is given as a text string, 'port' as a number and 'timeout' as a number (in milliseconds). Returns 1 if connected, 0 if connection failed.

A.1.2 res = disconnect()

Disconnects the TCP/IP connection to VAST. Always returns 1. If the request fails, a Java Error will stop the script, so a return of 0 is currently not possible.

A.1.3 errornumber = getlasterror()

Retrieves the error code of the last call of an API function. Error codes have the following meaning:

0	No error
1	Unknown error
2	Unexpected data received from VAST - API mismatch?
3	VAST received invalid data. Command ignored.
4	VAST internal data read failure
5	Internal VAST error
6	Could not complete command because modifying the view in VAST is disabled
7	Could not complete command because modifying the segmentation in VAST is disabled
10	Coordinates out of bounds
11	Mip level out of bounds
12	Data size overflow
13	Data size mismatch - Specified coordinates and data block size do not match
14	Parameter out of bounds
15	Could not enable diskcache (please set correct folder in VAST Preferences)
20	Annotation object or segment number out of bounds
21	No annotation or segmentation available
22	RLE overflow - RLE-encoding makes the data larger than raw; please request as raw
23	Invalid annotation object type
24	Annotation operation failed
30	Invalid layer number
31	Invalid layer type
32	Layer operation failed
50	`sourcearray` and `targetarray` must have the same length

Table A.1: VAST API Error Codes as returned by getlasterror()

A.1.4 [apiversion, res] = getapiversion()

Returns the version of the API provided by the currently connected VAST executable. The API version described here is version 4.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.1.5 [version, subversion, res] = getcontrolclassversion()

Returns the version of the `VASTControlClass.m` script (not part of the API). The current version is 5.7, introduced in VAST 1.4.

A.1.6 [info, res] = getinfo()

Reads out general information from VAST.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

Returns a struct with the following fields if successful, or an empty struct [] if failed:

info.datasizex	X (horizontal) size of the data volume in voxels at full resolution
info.datasizey	Y (vertical) size of the data volume in voxels at full resolution
info.datasizez	Z (number of slices) size of the data volume in voxels
info.voxelsizex	X size of one voxel (in nm)
info.voxelsizey	Y size of one voxel (in nm)
info.voxelsizez	Z size of one voxel (in nm)
info.cubesizex	X size of the internal cubes used in VAST in voxels; always 16
info.cubesizey	Y size of the internal cubes used in VAST in voxels; always 16
info.cubesizez	Z size of the internal cubes used in VAST in voxels; always 16
info.currentviewx	Current view X coord in VAST in voxels at full res (window center)
info.currentviewy	Current view Y coord in VAST in voxels at full res (window center)
info.currentviewz	Current view Z coord in VAST in voxels (slice number)
info.nrofmiplevels	Number of mip levels of the current data set

Table A.2: getinfo() struct

A.1.7 [info, res] = gethardwareinfo()

Returns the hardware properties of the computer on which VAST is running in the struct `info`. The fields of the struct are as follows:

info.computername	text	Name of the computer
info.processorname	text	Name of the processor
info.processorspeed_ghz	double	Clock speed of the processor in GHz
info.nrofprocessorcores	integer	Number of processor cores
info.tickspeedmhz	double	Tick speed of the VAST computer in MHz
info.mmxssecapabilities	text	Text describing the processor capabilities
info.totalmemorygb	double	Memory installed, in GB
info.freememorygb	double	Free memory, in GB
info.graphicscardname	text	Graphics card name
info.graphicsdedicatedvideomemgb	double	Dedicated video memory in GB
info.graphicsdedicatedsysmemgb	double	System memory dedicated for graphics, in GB
info.graphicssharedsysmemgb	double	System memory shared for graphics, in GB
info.graphicsrasterizerused	text	'Undefined', 'Hardware', 'Warp' or 'Reference'

Table A.3: gethardwareinfo() struct

If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 2.

A.2 Layer Functions

A.2.1 [nroflayers, res] = getnroflayers()

Returns the number of layers currently loaded in VAST.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.2 [linfo, res] = getlayerinfo(layernr)

Returns information about the layer with the given number. layernr can be a number between 0 and getnroflayers-1. If the function succeeds, it will return a struct `layerinfo` with fields as described below and res will be 1. If it fails it will return an empty struct and res will be 0. This returns the values which are also visible in the VAST 'Layers' tool window.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

`linfo.type`	0: Image layer (VSV), 1: Segmentation layer, 2: Annotation layer, 3: VSVR image layer, 6: VSVI image layer, 7: Tool layer
`linfo.editable`	1 if the layer is editable, 0 else
`linfo.visible`	1 if the layer is visible, 0 else ('Visible' checkbox)
`linfo.brightness`	'Bright' (brightness) or 'Sel Alpha' checkbox
`linfo.contrast`	'Contrast' or 'Pattern' checkbox
`linfo.opacitylevel`	Slider next to 'Visible', which defines layer opacity
`linfo.brightnesslevel`	Slider next to 'Bright' or 'Sel Alpha'
`linfo.contrastlevel`	Slider next to 'Contrast' or 'Pattern'
`linfo.blendmode`	The current blend mode (see below)
`linfo.blendoradd`	0: alpha blending, 1: additive blending, 2: subtractive blending*
`linfo.tintcolor`	Tint color as 32-bit RGBA value. Default: white (`0xffffffff`)*
`linfo.name`	Name of the layer
`linfo.redtargetcolor`	Red channel target color (RGBA). Default: red (`0xff0000ff`)**
`linfo.greentargetcolor`	Green channel target color (RGBA). Default: green (`0x00ff00ff`)**
`linfo.bluetargetcolor`	Blue channel target color (RGBA). Default: blue (`0x0000ffff`)**
`linfo.bytesperpixel`	Number of bytes per pixel in the layer
`linfo.ischanged`	Whether the contents of the layer have been changed (1) or not (0)
`linfo.inverted`	Whether the layer is shown inverted (1) or not (0)*.
`linfo.solomode`	Whether 'Solo' mode in the 'Layers' tool window is on (1) or off (0)

Table A.4: getlayerinfo() struct. *: Available only for image layers. **: Available only for RGB image layers. The 'inverted' flag does not work for 32-bit and 64-bit RGB-translated image layers.

Blend modes for image layers are: 0: no transparency ('Flat'), 1: the darker `mean(r,g,b)` the more transparent, 2: the brighter `mean(r,g,b)` the more transparent, 3: the darker `max(r,g,b)`, the more transparent, 4: the brighter `max(r,g,b)`, the more transparent, 5: black transparent / nonzero opaque.
Blend modes for segmentation layers are: 0: normal, 1: mask to black, 2: mask to white, 3: inverse mask to black, 4: inverse mask to white.

A.2.3 res = setlayerinfo(layernr, linfo)

Sets the parameters of the layer with the given number (counting from 0 in the order as they are displayed in the Layers window in VAST). `linfo` is a struct that can have a subset or all of the entries as in `getlayerinfo()` above, except `type` and `name`.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.4 [mipscalematrix, res] = getmipmapscalefactors(layernr)

Returns a matrix containing the texture scaling factors of all mip levels. Each row contains the scaling factors for one mip level (row 1 is for mip level 1, for example). The three columns are the

scaling factors for the X, Y and Z directions. The scaling factors for mip level 0 are always (1,1,1) and are not included. The scaling factors can be used to compute the voxel coordinates at different mip levels: To convert from full-resolution coordinates to pixel coordinates at a mip level, divide each coordinate by its scaling factor in the row corresponding to the desired mip level.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.5 res = setselectedlayernr(layernr)

Sets the selected layer in VAST to the layer with the given number (counting from 0 as the first layer in the list). If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.6 [selectedlayernr, selectedemlayernr, selectedsegmentlayernr, res] = getselectedlayernr()

Returns the numbers of the currently selected layer, EM layer, and segment layer, or -1 if there is no such layer. The 'selected layer' is the layer which is actually highlighted in the 'Layers' tool window in VAST (which was selected last).
This function is included for backwards compatibility. If you need the selected annotation and tool layers as well, please use `getselectedlayernr2` below.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.7 [selectedlayernr, selectedemlayernr, selectedannolayernr, selectedsegmentlayernr, selectedtoollayernr, res] = getselectedlayernr2()

Returns the numbers of the currently selected layer, EM layer, annotation layer, segmentation layer, and tool layer, or -1 if there is no such layer. The 'selected layer' is the layer which is actually highlighted in the 'Layers' tool window in VAST (which was selected last).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.8 [id, res] = addnewlayer(laytype, name, refid[opt])

Adds a new layer with the given `name` to the list. `laytype` is: 1: annotation layer, 2: segmentation layer (XY mipmapping), 3: tool layer, 4: segmentation layer with XY/XYZ mipmapping (if available).
`refid` is the layer number after which the new layer should be inserted (counting from 0 as they are listed in the Layers window). If `refid` is not specified, the new layer will be inserted after the selected layer. An image volume layer has to be present in VAST for this function to succeed.
If the call succeeded, id will be the number of the new layer and `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.9 [id, res] = loadlayer(filename, refid[opt])

Opens a layer from a file with the specified `filename`. `refid` is the layer number after which the loaded layer should be inserted (counting from 0 as they are listed in the Layers window). If `refid` is not specified, the loaded layer will be inserted after the selected layer or as the first and only layer

in the layers list.
If the call succeeded, id will be the number of the new layer and `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.10 res = savelayer(layernr, targetfilename, forceit[opt], subformat[opt])

Saves the layer with the selected `layernr` (counting from 0 as they are listed in the Layers window) to a file with name `targetfilename`. `forceit` and `subformat` are optional parameters. If `forceit` is 1, the file will be written even if the target file exists (the previous file will be overwritten). If `forceit` is not specified or set to 0, VAST will not overwrite existing files. `subformat` can be used for specifying the file format when saving segmentation layers. `subformat` 0 (default) saves to an unpacked .VSS file, 1 to a packed .VSS file (slower but much smaller), and 2 to a legacy .VSS format compatible with VAST 1.2.1 and earlier.
Only annotation layers and segmentation layers can be saved.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.11 res = removelayer(layernr, forceit)

Removes the layer with the selected `layernr` (counting from 0 as they are listed in the Layers window) from VAST. If `forceit` is 1, the layer will be removed even if it has been changed.
Please note that currently the last image volume layer cannot be removed since it defines general data properties. Please also be aware that removing layers will change the layer numbering.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.12 [newlayernr, res] = movelayer(movedlayernr, afterlayernr)

Moves the selected layer to a different position in the Layers window list. `afterlayernr` specifies after which other layer the moved layer (idetified by `movedlayernr`) should be placed. To move a layer to the top of the stack, first place it underneath the first layer, and then move the first layer down. Please be aware that moving layers will change the layer numbering.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.13 [isenabled, res] = getapilayersenabled()

Returns whether separate API layer selection is enabled (`isenabled` is 1) or not (`isenabled` is 0). When enabled, API functions which work with a 'selected layer' will use the layer specified by `setselectedapilayernr` below. When disabled, they will use the layer selected in the VAST user interface (which can be controlled by the API through the function `setselectedlayer`).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.14 res = setapilayersenabled(enabledflag)

Enables or disables separate API layer selection by passing 1 as `enabledflag` to switch it on and 0 to switch it off. When enabled, API functions which work with a 'selected layer' will use the layer specified by `setselectedapilayernr` below. When disabled, they will use the layer selected in the VAST user interface (which can be controlled by the API through the function `setselectedlayernr`

above).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.15 [selectedlayernr, selectedemlayernr, selectedannolayernr, selectedsegmentlayernr, selectedtoollayernr, res] = getselectedapilayernr()

This function returns the numbers of the layers of different types which are currently selected for API access. `selectedlayernr` is the most recent layer selected, of any type.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.2.16 res = setselectedapilayernr(layernr)

Selects the layer with the given `layernr` for API access (API layer control has to be enabled via `setapilayersenabled` above for the API layer selection to be used by API functions).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3 Annotation Layer Functions

A.3.1 [nrofobjects, firstannoobjectnr, res] = getannolayernrofobjects()

Returns how many objects exist in the selected annotation layer in `nrofobjects` and the number (id) of the first annotation object in the hierarchy in the 'Annotation Objects' tool window.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.2 [annolayerobjectdata, res] = getannolayerobjectdata()

Returns information about all annotation objects in the currently selected annotation layer, as a matrix with one row per object and the following columns:

1	ID number of the annoobject
2	Flags field of the segment as 16-bit value: currently bit 0 is 'isused', bit 8 is 'isexpanded'
3	Type of the annoobject: 0: folder, 1: skeleton
4-7	Primary color as red, green, blue, pattern1
8-11	Secondary color as red, green, blue, pattern2 (unused)
12-14	XYZ coordinates of the annoobject's anchor point (in voxels)
15-18	IDs of parent, child, previous and next annoobject (0 if none)
19	If the annoobject is in a collapsed folder this is the folder ID, else a copy of column 1
20-25	Annoobject bounding box (x1,y1,z1, x2,y2,z2)
26	Lower 32 bits of the user flags
27	Higher 32 bits of the user flags
28	Lower 32 bits of the user ID
29	Higher 32 bits of the user ID

Table A.5: annolayerobjectdata matrix

If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.3 [aoname, res] = getannolayerobjectnames()

This function returns the names of all objects in the selected annotation layer in `aoname`, in the same order as in the returned matrix in `getannolayerobjectdata` above.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.4 [id, res] = addnewannoobject(refid, nextorchild, type, name)

Adds a single annotation object with the given `name` to the selected annotation layer. The new annotation object will be added either immediately after (`nextorchild`=0) or as a child of (`nextorchild`=1) the annotation object with the number (id) given in `refid`. If `type` is 0, a folder will be added, if `type` is 1, a skeleton will be added.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.5 res = moveannoobject(id, refid, nextorchild)

Relocates the annotation object with number `id` in the tree of objects in the 'Annotation Objects' tool window. It is moved to be either the immediately following sibling of the object with number `refid` (if `nextorchild` is 0) or as child of the object with number `refid` (if `nextorchild` is 1).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.6 res = removeannoobject(id)

Removes the annotation object with the specified ID from the selected annotation layer. Please note that annotation objects with children in the hierarchy cannot be removed/deleted (but you can remove the object after you move or remove its children).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.7 res = setselectedannoobjectnr(id)

Selects the object with the given `id` number in the selected annotation layer.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.8 [id, res] = getselectedannoobjectnr()

Returns the id number of the selected object in the selected annotation layer in `id`.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.9 [aonodedata, res] = getaonodedata()

Returns a matrix containing information about all nodes of the selected annotation object in the selected layer, if it is a skeleton. This function is outdated, please consider using the function **??**getannoobject(id)— instead.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.10 [nodenumbers, nodelabels, res] = getaonodelabels()

Returns the node labels of the selected skeleton in the selected annotation layer. The function returns a list of node numbers (as counted in depth-first-search order) of nodes which have labels in `nodenumbers`, and the corresponding label strings in the cell array `nodelabels`.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.11 res = setselectedaonodebydfsnr(nr)

Selects the node given by `nr` in the selected skeleton of the selected annotation layer. `nr` is the number of the node in the skeleton, counting from the root node (0) in depth-first-search order. The depth-first-search numbers of the nodes change if nodes are added or deleted or edges swapped.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.12 res = setselectedaonodebycoords(x, y, z)

Selects a node of the selected skeleton of the selected annotation layer by coordinates. (x,y,z) define the location of the to-be-selected node in pixels at full resolution.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.13 [nr, res] = getselectedaonodenr()

Returns the depth-first-search number of the selected node of the selected annoobject, if it is a skeleton.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.14 res = addaonode(x, y, z)

If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.15 res = moveselectedaonode(x, y, z)

Moves the selected node of the selected skeleton of the selected annotation layer to new coordinates. (x,y,z) define the new location in pixels at full resolution.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.16 res = removeselectedaonode()

This function deletes the selected node of the selected skeleton of the selected annotation layer, if possible. If the deleted node has two attached edges, the neighbor nodes will be directly connected. Nodes with three attached edges cannot be deleted.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.17 res = swapselectedaonodechildren()

This function swaps the child 1 and child 2 branches of the selected node of the selected skeleton of the selected annotation layer.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.18 res = makeselectedaonoderoot()

This function makes the selected node of the selected skeleton of the selected annotation layer the root node of that skeleton, flipping edges in the skeleton as necessary.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.19 [branchid, res] = splitselectedskeleton(newrootdfsnr, newname)

This function splits the selected skeleton of the selected annotation layer into two parts. This is done by removing the parent edge of the node with the depth-first-search number specified by `newrootdfsnr`. This node will become the new root node of a new skeleton which will be added to the annotation objects list, using the name given by `newname`. The ID of the generated annotation object is returned in `branchid` (0 in case of error).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.20 res = weldskeletons(annoobjectnr1, nodedfsnr1, annoobjectnr2, nodedfsnr2)

This function can be used to merge two skeletons into one. The annoobjectnr2 skeleton will become part of the annoobjectnr1 skeleton by removing the nodedfsnr2 node of annoobjectnr2 and linking its children to the nodedfsnr1 node of the annoobjectnr1 skeleton. For this to work the nodedfsnr1 node has to be a leaf node (no children) which is not the skeleton root node.

If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.21 [aodata, res] = getannoobject(id)

Retrieves all information of the annoobject with number `id` in the struct `aodata`, counting from 1. If `id` is 0, the data of the selected annoobject will be returned. The struct has the following fields:

type	0 for folders, 1 for skeletons
label	Name of the annoobject
nodecolor	32-bit value specifying the colors to be used for the skeleton nodes
edgecolor	32-bit value specifying the colors to be used for the skeleton edges
userflags	64-bit value, user-defined
userid	64-bit value, user-defined
nodematrix	32-bit unsigned int matrix specifying the nodes and edges of the skeleton, see below
nodediameters	Array of double values defining the diameters set for all nodes
nodelabels	Array array of strings containing the node labels used in the skeleton
parentedgeweights	List of double values for the weights of the parent edge of each node, 0 if none
ext	Array of 64-bit integer values used to store tool layer related data; one per node
ext2	Array of 32-bit integer values used to store tool layer related data; one per node
selectflags	Array of 32-bit integer values. Bit 0: set if node in range selection. Bit 1: set if node selected in node tool.

Table A.6: aodata struct

`nodematrix` contains one row per node, starting with the root node in depth-first-search order, with the following columns:

1	Flags: see below
2	X coordinate of node
3	Y coordinate of node
4	Z coordinate of node
5	Row number of parent node. Has to be smaller than this row (depth-first order)
6	Row number of child1 node
7	Row number of child2 node
8	Type of node: 32-bit value stored with each node
9	Type (0..15) of parent edge if it exists, 0 else
10	Index of label in `nodelabels` if this node has a label, 0 else

Table A.7: aodata.nodematrix columns

The following bits in the `flags` field are used:

bit 0	Set if this is the selected node
bit 1	Set if this is the root node
bit 8	Set if this node has a parent edge and node
bit 9	Set if this node has a child1 edge and node
bit 10	Set if this node has a child2 edge and node
bit 16	Set if ext is valid (tool layer functionality)
bit 17	Exclude flag (tool layer functionality)

Table A.8: aodata.nodematrix flags

If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.3.22 [id, res] = setannoobject(id, aodata)

Sets specific parameters of the annoobject with number `id`, counting from 1. If `id` is 0, the data of the selected annoobject will be edited. `aodata` specifies the parameters to be edited equivalent to

getannoobject above (table A.6). Values in arrays have to be provided in depth-first search order of nodes in the skeleton tree, one value per node. Setting one or several of the following parameters is currently supported:

label	Name of the annoobject
nodecolor	32-bit value specifying the colors to be used for the skeleton nodes
edgecolor	32-bit value specifying the colors to be used for the skeleton edges
userflags	64-bit value, user-defined
userid	64-bit value, user-defined
nodediameters	Array of double values defining the diameters of all nodes
parentedgeweights	Array of double values for the weights of the parent edge of each node, 0 if none
ext	Array of 64-bit integer values used to store tool layer related data; one per node
ext2	Array of 16-bit integer values used to store tool layer related data; one per node

Table A.9: aodata fields supported by setannoobject

If the call succeeded, res will be 1, otherwise 0. Use getlasterror() to retrieve the error code.

A.3.23 [id, res] = addannoobject(refid, nextorchild, aodata)

Adds a new annoobject immediately after the annoobject with id number refid, if nextorchild is 0, or as a child of the annoobject with id number refid, if nextorchild is 1. aodata is a struct with the same format as returned by getannoobject above. The fields type and label are mandatory. if type is 1, indicating a skeleton annoobject, the field nodematrix is also mandatory.

VAST does a number of consistency and error checks on the aodata struct, but not everything is checked. Please note that uploading faulty data may lead to unexpected behavior, including crashes.

If the call succeeded, res will be 1, otherwise 0. Use getlasterror() to retrieve the error code.

A.3.24 [annoobjectid, nodedfsnr, distance, res]=getclosestaonodebycoords(x, y, z, maxdistance)

This function looks for nodes which are closer than maxdistance pixels away from (x,y,z), given in full-resolution coordinates. If any exist, it will return the closest node and the id of its object as well as the distance. Distances are computed as 3D Euclidian distances in voxel coordinates (not normalized for anisotropic voxel sizes).

If the call succeeded, res will be 1, otherwise 0. Use getlasterror() to retrieve the error code.

A.3.25 [nodedata, res] = getaonodeparams(annoobjectid, nodedfsnr)

Returns a struct nodedata containing parameters of the node with depth-first-search number nodedfsnr from the annotation object annoobjectid (which has to be a skeleton) in the selected annotation layer. If annoobjectid is 0, the selected annotation object will be used. Please note that nodedfsnr is 0-based, meaning that the root node has nodedfsnr 0.

The struct nodedata has the following fields:

coords	X,Y,Z Coordinates of the queried node, in full-resolution pixels
parentcoords	X,Y,Z Coordinates of the parent node of the queried node
child1coords	X,Y,Z Coordinates of the first child node of the queried node
child2coords	X,Y,Z Coordinates of the second child node of the queried node
diameter	Diameter value of the queried node (double)
xflags	Extended flags for the node (8-bit unsigned integer)
ext	EXT value of the queried node (64-bit unsigned integer; from image layer)
ext2	EXT2 value of the queried node (16-bit uint as 32-bit; from segmentation)
haslabel	This is 1 if the node has a label and 0 else
label	The node label, if the node has a label, or an empty string else
hasparentedge	This is 1 if the node has a parent edge and 0 else
parentedgetype	If the parent edge exists, this is its type (32-bit unsigned integer)
parentedgeweight	If the parent edge exists, this is its weight (double)

Table A.10: nodedata struct

The bits in `xflags` are: bit 0: 1 if ext is valid; bit 1: exclude flag; bit 2: 1 if ext2 is valid.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Added in API version 5.9.

A.3.26 res = setaonodeparams(annoobjectid, nodedfsnr, nodedata)

Sets the parameters specified in struct `nodedata` in the node with depth-first-search number `nodedfsnr` in the annotation object `annoobjectid` (which has to be a skeleton) in the selected annotation layer. If `annoobjectid` is 0, the selected annotation object will be used. Please note that `nodedfsnr` is 0-based, meaning that the root node has `nodedfsnr` 0.

The struct `nodedata` can have any of the following fields:

coords	X,Y,Z Coordinates of the queried node, in full-resolution pixels (move node)
diameter	Diameter value of the queried node (double)
xflags	Extended flags for the node (8-bit unsigned integer)
ext	EXT value of the queried node (64-bit unsigned integer)
ext2	EXT2 value of the queried node (16-bit unsigned integer)
label	The node text label
parentedgetype	Type of the parent edge (if the parent edge exists) (32-bit unsigned integer)
parentedgeweight	Weight value of the parent edge (if the parent edge exists) (double)

Table A.11: Settable parameters in nodedata struct

The bits in `xflags` are: bit 0: 1 if ext is valid; bit 1: exclude flag; bit 2: 1 if ext2 is valid.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Added in API version 5.9.

A.4 Segmentation Layer Functions

A.4.1 [nr, res] = getnumberofsegments()

Returns the number of segments of the active segmentation layer or `[]` if failed.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.4.2 [data, res] = getsegmentdata(id)

Reads out information of the segment with number `id` in the active segmentation layer. This is the same information that gets written to a text file by 'Save Segmentation Metadata ...' in VAST (see section 4.4.16).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.
Returns a struct with the following fields if successful, or an empty struct `[]` if failed:

`data.id`	ID of the requested segment
`data.flags`	Flags field of the segment as 32-bit value
`data.col1`	Primary color as 32-bit value
`data.col2`	Secondary color as 32-bit value
`data.anchorpoint`	XYZ coordinates of the segment's anchor point (in voxels)
`data.hierarchy`	IDs of parent, child, previous and next segment (0 if none)
`data.collapsednr`	If the segment is collapsed into a folder, this is the folder ID
`data.boundingbox`	Segment bounding box (may be incorrect if voxels were deleted)

Table A.12: getsegmentdata() struct

A.4.3 [name, res] = getsegmentname(id)

Returns the name of the segment with the given ID in the active segmentation layer. Returns `[]` if the ID is out of range.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.4.4 res = setanchorpoint(id, x, y, z)

Sets the anchor point of the segment with the given ID in the active segmentation layer to the given coordinates (in voxels at full resolution). Only non-negative values are allowed for x, y and z.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.4.5 res = setsegmentname(id, name)

Sets the name of the segment with the given ID in the active segmentation layer.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.4.6 res = setsegmentcolor8(id, r1, g1, b1, p1, r2, g2, b2, p2)

Sets the primary and secondary colors (and pattern) of the segment with the given ID in the active segmentation layer. Values for r1, g1, b1, r2, g2 and b2 have to be between 0 and 255. p1 defines the pattern and has to be between 0 and 15. p2 is currently unused.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.4.7 res = setsegmentcolor32(id, col1, col2)

Same as setsegmentcolor8, but the two colors for the segment are given as two 32-bit values.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.4.8 [segdata, res] = getallsegmentdata()

This function is similar to `data = getsegmentdata(id)` above, but retrieves the data for all segments in the active segmentation layer at once and returns a cell array of structs instead of a single struct.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.4.9 [segdatamatrix, res] = getallsegmentdatamatrix()

Same as `data = getallsegmentdata(id)` above, but returns the data as a matrix with one row per segment and one column per data value. The columns are the same as when exporting segment metadata to a file (see section 4.4.16), with the exception of the segment names.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.4.10 [segname, res] = getallsegmentnames()

This function is similar to `name = getsegmentname(id)` above, but retrieves the names of all segments in the active segmentation layer at once and returns them in a cell array. If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.4.11 res = setselectedsegmentnr(segmentnr)

Sets the selected segment in the active segmentation layer in VAST to the segment with the given ID number. If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.4.12 [selectedsegmentnr, res] = getselectedsegmentnr()

Returns the ID number of the currently selected segment in the active segmentation layer, or -1 if an error occurred. If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.4.13 res = setsegmentbbox(id, minx, maxx, miny, maxy, minz, maxz)

Sets the bounding box of the segment with number `id` in the active segmentation layer.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.
Introduced in API version 2.

A.4.14 [firstsegmentnr, res] = getfirstsegmentnr()

Returns the segment number (id) of the first segment (after Background) in the active segmentation layer.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.
Introduced in API version 2.

A.4.15 [id, res] = addsegment(refid, nextorchild, name)

Adds one segment with the given `name` to the selected segmentation layer. If `nextorchild` is 0, the new segment will be added directly behind the segment with ID `refid`. If `nextorchild` is 1, the new segment will be added as last child of the segment with ID `refid`.

If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.
Introduced in API version 3.

A.4.16 res = movesegment(id, refid, nextorchild)

Moves the segment with the specified `id` in the selected segmentation layer to the location indicated by `refid` and `nextorchild`. If `nextorchild` is 0, the segment will be removed from its current location and added directly behind the segment with ID `refid`. If `nextorchild` is 1, the segment will be removed from its current location and added as last child of the segment with ID `refid`.

If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.
Introduced in API version 3.

A.5 Tool Layer Functions

A.5.1 res = settoolparameters(toollayernr, toolnodenr, params)

This function sets parameters of a tool layer. `toollayernr` specifies the tool layer to edit and toolnodenr must currently be 0.

`params` is a struct that can contain the following fields:

`params.nodetype`	Reserved for future use
`params.imagesourcelayer`	Specifies the Image Source Layer number
`params.annosourcelayer`	Specifies the Collector Source Layer number
`params.sourcemiplevel`	Source mip level, see below

Table A.13: settoolparameters() struct

`params.sourcemiplevel` specifies the mip level at which segment IDs from the image source layer should be sampled. Set to -1 to use the mip level of the 2D window.

A.6 2D View Functions

A.6.1 [x, y, z, res] = getviewcoordinates()

Returns the current view coordinates in VAST in voxels at full resolution.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.6.2 [zoom, res] = getviewzoom()

Returns the current zoom value in VAST.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.6.3 res = setviewcoordinates(x, y, z)

Sets the current x, y, z coordinates (in voxels at full resolution) of the view (center of window) in VAST.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.6.4 res = setviewzoom(zoom)

Sets the current zoom value. Zoom values are currently integer values and can be negative. The higher the value, the more magnified the view. A zoom value of 0 sets the pixel size of the full resolution image to exactly the 'Target Resolution Smaller Than' value specified in the VAST Preferences. A value of -8 zooms out by a factor 2 (and shows the next lower mip level texture).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.6.5 res = set2dvieworientation(orientation[opt], viewport[opt])

Sets the slicing orientation of the 2D view. `orientation` 0: XY slicing (default), 1: XZ slicing, 2: YZ slicing. `viewport` has to be 0.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.6.6 res = refreshlayerregion(layernr, minx,maxx,miny,maxy,minz,maxz)

Removes all cached data in the defined region of the selected layer from the cache and primes it for reload. This function is currently supported only for external image layers (.VSVR, .VSVI) and can be used to re-load data in VAST which was changed externally, for example, if image files in a .VSVI image stack were modified by a different program.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.7 Voxel Data Transfer Functions

A.7.1 [segimage, res] = getsegimageraw(miplevel, minx, maxx, miny, maxy, minz, maxz, flipflag, immediateflag, requestloadflag)

Reads out the segmentation as a voxel image for a given mip level (resolution) and area, as defined by minimum and maximum values for the ranges in x, y and z. This function transmits the segmentation image as raw data (a one-dimensional array of bytes); consider using `getsegimageRLEdecoded()` below for a version with faster transmission. If `flipflag` is 1, X and Y axis will be swapped to make the image appear the same as in VAST. If `flipflag` is unspecified or 0, segimage will appear with X and Y axis swapped.

If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

minx, maxx, miny, maxy, minz and maxz are given in voxels *at the resolution defined by* `miplevel`. You can retrieve the size of the volume at full resolution (mip level 0) using the function getinfo(). To compute the size of the volume in X and Y at a different mip level `miplevel`, use the following code:

```
xmin=bitshift(xmin,-miplevel);
xmax=bitshift(xmax,-miplevel)-1;
ymin=bitshift(ymin,-miplevel);
ymax=bitshift(ymax,-miplevel)-1;
```

Note that the size of the volume in Z does not change with mip level (this is the number of slices).

A.7.2 [segimageRLE, res] = getsegimageRLE(miplevel, minx, maxx, miny, maxy, minz, maxz, surfonlyflag, immediateflag, requestloadflag)

Reads the requested volume region and returns it as an runlength-encoded (RLE) string. This function is useful if you want to directly process the RLE string; otherwise use one of the functions below.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.
See `getsegimage()` above for information on volume coordinates.

The runlength-encoded image is an array of pairs of unsigned 16 bit values. The first value in a pair is the segment ID which follows, and the second value is the number of voxels with that ID to follow. The encoding order of voxels in the volume is row by row (X from left to right) concatenated for one plane (Y from top to bottom), and planes are concatenated from the first to the last slice of the volume. Please note that sequences of the same value defined by one pair can loop around at the end of a row and/or extend into the next plane. Run lengths longer than $2^{16}-1$ (65535) are encoded in several pairs.

A.7.3 [segimage, res] = getsegimageRLEdecoded(miplevel, minx, maxx, miny, maxy, minz, maxz, surfonlyflag, flipflag, immediateflag, requestloadflag)

Same as `getsegimage()` above, except that the data is transmitted run-length encoded (RLE) from VAST to Matlab, and then decoded on the Matlab side. This can improve speed because it reduces the size of the transmitted data. If `surfonlyflag` is 1, all interior voxels (any nonzero voxels for which all six direct neighbors have the same ID) are set to 0. If `flipflag` is 1, X and Y axis will be swapped to make the image appear the same as in VAST. If `flipflag` is unspecified or 0, segimage will appear with X and Y axis swapped. See `getsegimageraw()` above for information on volume coordinates.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.7.4 [values, numbers, res] = getRLEcountunique(miplevel, minx, maxx, miny, maxy, minz, maxz, surfonlyflag, immediateflag, requestloadflag)

This function returns the segment IDs (values) and number of voxels of each ID (numbers) in the requested subvolume. See `getsegimageraw()` above for information on volume coordinates.

If the call succeeded, res will be 1, otherwise 0. Use getlasterror() to retrieve the error code.

A.7.5 [segimage, values, numbers, res] = getsegimageRLEdecodedcountunique(miplevel, minx, maxx, miny, maxy, minz, maxz, surfonlyflag, flipflag, immediateflag, requestloadflag)

Combines the functions getsegimageRLEdecoded() and getRLEcountunique() above, and returns both the decoded subvolume and a list of segment IDs in the volume and number of voxels for each ID. If flipflag is 1, X and Y axis will be swapped to make the image appear the same as in VAST. If flipflag is unspecified or 0, segimage will appear with X and Y axis swapped. See getsegimageraw() above for information on volume coordinates.
If the call succeeded, res will be 1, otherwise 0. Use getlasterror() to retrieve the error code.

A.7.6 [segimage, values, numbers, bboxes, res] = getsegimageRLEdecodedbboxes(miplevel, minx, maxx, miny, maxy, minz, maxz, surfonlyflag, flipflag, immediateflag, requestloadflag)

Same as getsegimageRLEdecodedcountunique() above, but also returns the bounding boxes for all IDs. The coordinate order for the bounding boxes in bboxes is: [minx,miny,minz,maxx,maxy,maxz], one row per segment.[1] bboxes is not changed by flipflag. See getsegimageraw() above for information on volume coordinates.
If the call succeeded, res will be 1, otherwise 0. Use getlasterror() to retrieve the error code.

A.7.7 res = setsegtranslation(sourcearray, targetarray)

Tells VAST how to translate the segmentation volume before transmitting it through getsegimage* and getRLE* functions. sourcearray and targetarray are arrays of unsigned (positive) 16-bit values and have to have the same length. VAST will convert all voxels with IDs in sourcearray to the corresponding values in targetarray. Voxels with other IDs will be removed (set to 0). To disable the segmentation translation, call setsegtranslation([],[]).
If the call succeeded, res will be 1, otherwise 0. Use getlasterror() to retrieve the error code.

A.7.8 [emimage, res] = getemimageraw(layernr, miplevel, minx, maxx, miny, maxy, minz, maxz, immediateflag, requestloadflag)

Reads out the EM image of the specified layer as a voxel image for a given mip level (resolution) and area. The area is defined by minimum and maximum values for the ranges in x, y and z. This function transmits the segmentation image as raw data (a one-dimensional array of bytes) with either one or three bytes per voxel, depending on whether the layer has one or three color channels. Use getemimage to request a correctly reshaped two-, three- or four-dimensional image.
If the call succeeded, res will be 1, otherwise 0. Use getlasterror() to retrieve the error code.

[1] Sorry that the order is different than in most other places.

A.7.9 [emimage,res] = getemimage(layernr, miplevel, minx, maxx, miny, maxy, minz, maxz, immediateflag, requestloadflag)

Same as `getemimageraw`, but reshapes the one-dimensional array to a matrix of the requested dimensions. The order of dimensions in the matrix is Y,X,Z,C (C for color, R G B, if the layer has three color channels).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.7.10 [screenshotimage,res] = getscreenshotimageraw(miplevel, minx, maxx, miny, maxy, minz, maxz, collapseseg)

Reads out the EM and segmentation stack as currently displayed in VAST as a voxel image for a given mip level (resolution) and area. The area is defined by minimum and maximum values for the ranges in x, y and z. If `collapseseg` is 1, the color of the segmentation will appear as displayed in VAST, if it is 0, each segment will be colored in its native (uncollapsed) color. This function transmits the segmentation image as raw data (a one-dimensional array of bytes). Use `getscreenshotimage` to request a correctly reshaped two-, three- or four-dimensional image. Screenshot images are always returned in RGB format (three bytes per voxel).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.7.11 [screenshotimage,res] = getscreenshotimage(miplevel, minx, maxx, miny, maxy, minz, maxz, collapseseg)

Same as `getscreenshotimageraw`, but reshapes the one-dimensional array to a matrix of the requested dimensions. The order of dimensions in the matrix is Y,X,Z,C (C for color, R G B).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

A.7.12 res = orderscreenshotimage(miplevel, minx, maxx, miny, maxy, minz, maxz, collapseseg)

Orders VAST to send data and returns immediately. Must call `pickupscreenshotimage` as next API function!

$$Not well tested, use with caution$$

A.7.13 [screenshotimage,res] = pickupscreenshotimage(miplevel, minx, maxx, miny, maxy, minz, maxz, collapseseg)

Call after orderscreenshotimage to receive data. Parameters sent during order have to match parameters of pickup function. Do not call other VASTControlClass functions between order and pickup!

$$Not well tested, use with caution$$

A.7.14 res = setsegimageraw(miplevel, minx, maxx, miny, maxy, minz, maxz, segimage)

Writes the (2D or 3D) image in `segimage` to the active segmentation layer in VAST, at the mip level `miplevel` and target coordinates given in minx, maxx, miny, maxy, minz, maxz. These coordinates

have to be specified at the resolution of `miplevel` (see explanation in section `getsegimageraw` above).
VAST will not accept segmentation images which contain segment numbers which exceed the segment list. For existing segments, the segment metadata will be updated to include the imported segmentation image.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.
Introduced in API version 2.

A.7.15 res = setsegimageRLE(miplevel, minx, maxx, miny, maxy, minz, maxz, segimage)

Same as `setsegimageraw`, but this version encodes the segmentation image using RLE and transmits the RLE-encoded data to VAST. This reduces the amount of data that has to be transmitted, but RLE encoding in Matlab is slow, so `setsegimageraw` may be the faster alternative for Matlab.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.
Introduced in API version 2.

A.7.16 [pixelvalue, res]=getpixelvaluefromfullrescoords(layernr, miplevel, x, y, z)

This function retrieves the value of a single pixel at the defined full-resolution coordinates (`x,y,z`) from the specified image or segmentation layer with the given layer number (`layernr`), using the data at the specified mip level (`miplevel`). The value range of `pixelvalue` depends on the data format in the selected layer (integer between 8-bit to 64-bit). If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 4.

A.8 VAST UI Functions

A.8.1 [props, res] = getdrawingproperties()

Returns a structure `prop` containing the current values of the 'Drawing Properties' tool window:

props.paintcursordiameter	Integer value between 1 and 511
props.paintcursorlocked	0 (off) or 1 (on)
props.autofill	0 (off) or 1 (on)
props.zscrollenabled	0 (off) or 1 (on)
props.overwritemode	0: paint background, 1: paint all, 2: paint parent
props.mippaintrestriction	Integer value between -1 and number of mip levels. -1 means 'no restriction', otherwise restriction to painting indicated mip level only
props.paintdepth	0, 2, 4 or 8
props.useconditionalpainting	0 (off) or 1 (on)
props.cp_contiguousonly	0 (off) or 1 (on)
props.cp_method	Value between 0 and 6, see below
props.cp_sourcelayernr	Value between -1 and number of layers. -1 means 'no source layer', otherwise the layer with the indicated number (counting from the top in the layers toolwindow, starting with 0)
props.cp_lowvalue	Lower value of range or threshold for selected masking method
props.cp_highvalue	Upper value of range or threshold for selected masking method

Table A.14: getdrawingproperties() struct

Please note that `overwritemode` and `mippaintrestriction` are handled globally and are linked for the 'Drawing Properties' and 'Filling Properties' windows. The values of the `props.cp_method` parameter have the following meaning:

0	Pixel brightness between `cp_lowvalue` and `cp_highvalue`
1	Pixel brightness between `cp_lowvalue` and `cp_highvalue`, using 100% value range of EM background to normalize
2	Pixel brightness between `cp_lowvalue` and `cp_highvalue`, using 95% value range of EM background to normalize
3	Use trained color
4	Autopick source color
5	Autopick nonzero source color
6	Auto-Z-repick nonzero source color. This is the same as 5, except that the source color for masking will be re-picked during a paint stroke every time the user goes to a different section

Table A.15: Values of props.cp_method

If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.

A.8.2 res = setdrawingproperties(props)

Sets the parameters of the 'Drawing Properties' tool window. `props` is a struct that can have a subset or all of the entries as in `getdrawingproperties()` above.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.

A.8.3 [props, res] = getfillingproperties()

Returns a structure `prop` containing the current values of the 'Filling Properties' tool window:

`props.overwritemode`	0: paint background, 1: paint all, 2: paint parent
`props.mippaintrestriction`	Integer value between -1 and number of mip levels. -1 means 'no restriction', otherwise restriction to painting indicated mip level only
`props.donotfillsourcecolorzero`	0 (off) or 1 (on)
`props.sourcelayersameastarget`	0 (off) or 1 (on)
`props.sourcelayernr`	Value between -1 and number of layers. -1 means 'no source layer', otherwise the layer with the indicated number (counting from the top in the layers toolwindow, starting with 0)
`props.method`	Value between 0 and 4, see below
`props.lowvalue_x`	Lower value of range or threshold for selected method, X direction
`props.highvalue_x`	Upper value of range or threshold for selected method, X direction
`props.lowvalue_y`	Lower value of range or threshold for selected method, Y direction
`props.highvalue_y`	Upper value of range or threshold for selected method, Y direction
`props.lowvalue_z`	Lower value of range or threshold for selected method, Z direction
`props.highvalue_z`	Upper value of range or threshold for selected method, Z direction

Table A.16: getfillingproperties() struct

Please note that `overwritemode` and `mippaintrestriction` are handled globally and are linked for the 'Drawing Properties' and 'Filling Properties' windows.

The values of the `props.method` parameter have the following meaning:

0	'Pixel brightness between': Fill only into adjacent pixels where the source layer pixel brightness is between `lowvalue_x` and `highvalue_x`
1	'Pixel brightness XYZ': Fill only into adjacent pixels where the source layer pixel brightness is between `lowvalue_x/y/z` and `highvalue_x/y/z`, depending on filling direction
2	'Autopick source color': Pick source color and fill only into adjacent pixels where the source layer pixel value is in a range using `lowvalue_x` and `highvalue_x` distance from the picked color
3	'Autopick source color XYZ': Pick source color and fill only into adjacent pixels where the source layer pixel value is in a range using `lowvalue_x/y/z` and `highvalue_x/y/z` distance from the picked color, depending on filling direction
4	'Hybrid recolor / autopick': If the clicked pixel in the target layer is nonzero, recolor that color by filling; otherwise execute mode 2 (Autopick source color)

Table A.17: Values of props.method

If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.

A.8.4 res = setfillingproperties(props)

Sets the parameters of the 'Filling Properties' tool window. `props` is a struct that can have a subset or all of the entries as in `getfillingproperties()` above.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.

Introduced in API version 5.

A.8.5 [state, res] = getcurrentuistate()

Returns a structure `state` containing the current values of the state of the VAST user interface:

`state.mousecoordsx`	Current X position of the mouse pointer in data coordinates (mip0, full resolution)
`state.mousecoordsy`	Current Y position of the mouse pointer in data coordinates (mip0, full resolution)
`state.lastleftclickx`	X position of the mouse pointer in data coordinates (mip0, full resolution) when the left mouse button was last clicked
`state.lastleftclicky`	Y position of the mouse pointer in data coordinates (mip0, full resolution) when the left mouse button was last clicked
`state.lastleftreleasex`	X position of the mouse pointer in data coordinates (mip0, full resolution) when the left mouse button was last released
`state.lastleftreleasey`	Y position of the mouse pointer in data coordinates (mip0, full resolution) when the left mouse button was last released
`state.mousecoordsz`	current slice number (Z location, full resolution)
`state.clientwindowwidth`	Width of the client area of the 2D window on screen, in pixels
`state.clientwindowheight`	Height of the client area of the 2D window on screen, in pixels
`state.reservedflag`	Unused
`state.lbuttondown`	Whether the left mouse button is held down. 0 (no) or 1 (yes)
`state.rbuttondown`	Whether the right mouse button is held down. 0 (no) or 1 (yes)
`state.mbuttondown`	Whether the middle mouse button is held down. 0 (no) or 1 (yes)
`state.ctrlpressed`	Whether the CTRL button is held down. 0 (no) or 1 (yes)
`state.shiftpressed`	Whether the SHIFT button is held down. 0 (no) or 1 (yes)
`state.deletepressed`	Whether the DELETE button is held down. 0 (no) or 1 (yes)
`state.spacepressed`	Whether the SPACE button is held down. 0 (no) or 1 (yes)
`state.spacewaspressed`	Whether SPACE was pressed since last call. 0 (no) or 1 (yes)
`state.uimode`	Current UI mode; see below
`state.hoversegmentnr`	Segment number of the pixel of the active segmentation layer under the mouse cursor, if any; otherwise 0
`state.miplevel`	Currently displayed mip level in the 2D view
`state.paintcursordiameter`	Current paint cursor diameter in pixels

Table A.18: getcurrentuistate() struct

The values of the `state.uimode` parameter have the following meaning:

0	'Translate' mode
1	'Rotate' mode (currently unused)
2	'Paint' mode
3	'Annotation' mode
4	'Paint/Translate1' mode - In paint mode, but switched to 'Translate' mode by holding down the CTRL key
5	'Paint/Translate2' mode - In paint mode, but still switched to 'Translate' mode because left mouse button is down
6	'Collect' mode
7	'Pick' mode
8	'Fill' mode
9	'Split' mode

Table A.19: Values of state.uimode

If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.

A.8.6 res = setuimode(uimode)

Sets the user interface (UI) mode in VAST. `uimode` has to be a value between 0 and 6 (please note that these values are different from `state.uimode` above because some modes cannot be set!):

0	'Translate' mode
1	'Annotation' mode
2	'Paint' mode
3	'Collect' mode
4	'Pick' mode
5	'Fill' mode
6	'Split' mode

Table A.20: Values of uimode

If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.

A.8.7 res = showwindow(windownr, onoff[opt], xpos[opt], ypos[opt], width[opt], height[opt])

Shows or hides windows in VAST and moves them to specific locations with respect to the main VAST window. `windownr` indicates which window is affected. `onoff`, `xpos`, `ypos`, `width` and `height` are optional parameters. If `onoff` is 1, the window will be shown (default), if it is 0 it will be hidden. If defined, `xpos` and `ypos` will define the target location of the upper left corner of the window with respect to the upper left corner of the main window, in pixels to the right and down. `width` and `height`, if specified, define the target size of the window.

`windownr` is:

1	'Layers' tool window
2	'Coordinates' tool window
3	reserved
4	'Annotation Objects' tool window
5	'Drawing Properties' tool window
6	'Filling Properties' tool window
7	'Splitting Properties' tool window
8	'Segment Colors' tool window
9	'Control Buttons' tool window
10	'Keyboard Shortcuts' tool window
11	'Remote Control API Server' tool window
12	'3D Viewer' window
13	'Autoskeletonizer' window

Table A.21: Values of windownr

If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.

A.8.8 [isenabled, res] = geterrorpopupsenabled(errornr)

Returns whether the error pop-ups for the error with the given errornr in VAST are enabled (1) or disabled (0).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.

A.8.9 res = seterrorpopupsenabled(errornr, enabledflag)

Enables or disables the error pop-ups for the error with the given errornr in VAST. To enable, set enabledflag to 1, to disable set enabledflag to 0.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.

A.8.10 [isenabled, res] = getpopupsenabled(typenr)

Returns whether the pop-ups for the given type in VAST are enabled (1) or disabled (0). Currently the only supported typenr is 0, which denotes the display of progress pop-up windows.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.

A.8.11 res = setpopupsenabled(typenr, enabledflag)

Enables or disables the pop-ups for the given typenr in VAST. To enable, set enabledflag to 1, to disable set enabledflag to 0. Currently the only supported typenr is 0, which controls the display of progress pop-up windows.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.

A.9 Execute VAST Functions

A.9.1 res = executefill(sourcelayernr, targetlayernr, x, y, z, mip)

Performs (masked) filling to the segmentation layer specified by targetlayernr, with filling seed at (x,y,z) in full-resolution (mip 0) coordinates, at the specified mip level `mip`. Other than that, the parameters specified in the 'Filling Properties' tool window are used. They can be changed using the setfillingproperties() API function. Filling may take a long time and this function will only return when the filling function has completed.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.

A.9.2 res = executecanvaspaintstroke(coords)

Performs a pen stroke on the selected segmentation layer through the list of X,Y *window* coordinates specified in `coords`. `coords` has to be a matrix with two columns, in which each row represents one consecutive point of the paint stroke with the value in the first column defining the X coordinate and the value in the second column defining the Y coordinate.
Since this function uses VASTs paint canvas system, window coordinates are used, and painting will only work if the region of the segmentation layer used has loaded on the layer's 2D canvas (is displayed in the 2D window without red frames).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.

A.9.3 res = executestartautoskeletonization(toollayernr, mip, nodedistance_mu, regionpadding_mu)

Starts the autoskeletonizer using the specified tool layer number, mip level, node distance, and region padding. This function will fail if the autoskeletonizer is already running or invalid parameters are used. Skeletonization is performed by filling of voxel regions. It will start from the selected node in the annotation layer specified by the given tool layer. The region size used is defined by the coordinates of the currently selected node and its parent node (which has to exist). The voxel data block loaded and processed extends by `nodedistance_mu+regionpadding_mu` around the bounding box defined by those two nodes. Nodes are added sequentially at the distance `nodedistance_mu`.
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.11.

A.9.4 res = executestopautoskeletonization()

If the autoskeletonizer is running, this will ask VAST to stop it at the next opportunity (when the current step finishes). `res` will be 0 if the autoskeletonizer was not running and can therefore not be stopped (error code 24).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code. Introduced in API version 5.11.

A.9.5 [state, res] = executegetautoskeletonizationstate()

Returns the current status of the autoskeletonizer in `state`. `state.isrunning` is 1 if the autoskeletonizer is currently running and 0 if not. `state.iswaiting` is 1 if the autoskeletonizer is currently waiting for VAST to load the required voxel data, and 0 if not. `state.nrnodesadded` contains the number of nodes added so far (or, if `state.isrunning` is 0, the number of nodes added in the previous autoskeletonization run).
If the call succeeded, `res` will be 1, otherwise 0. Use `getlasterror()` to retrieve the error code.
Introduced in API version 5.11.

A.10 Helper Functions

A.10.1 ids=getidsfromexactname(name)

Returns the ID(s) of all segments in the selected segmentation layer which match the given name exactly. In case of errors, `ids` will be empty.

A.10.2 ids=getimmediatechildids(parentidlist)

Returns the ID(s) of all segments which are immediate children of any of the segments with ids in `parentidlist`, in the selected segmentation layer. In case of errors, `ids` will be empty.

A.10.3 ids=getchildtreeids(parentidlist)

Returns the ID(s) of all segments which are children of any of the segments with ids in `parentidlist`, in the selected segmentation layer. In case of errors, `ids` will be empty.

Appendix B

FAQ and Trouble Shooting

B.1 Frequently Asked Questions

My image stack is not aligned. How do I get it aligned into VAST?
VAST does not have any stack alignment (nor stitching) functionality. You'll have to use other programs to render an aligned image stack (for example Adobe Photoshop or plugins in Fiji [1], [2]), and then import that aligned stack into VAST; or you'll have to work on an unaligned image stack.

Can I analyze multi-channel optical image stacks in VAST?
Yes. You can load several image stacks at once, provided they are the aligned and the same size, and each one can be RGB or graylevel. You can tint different image stacks in different colors to distinguish different channels and blend them together in different ways (see section 3.1.5).

Does VAST support 4-dimensional data sets (for example time-lapse data of a 3D structure)?

No.

How do I open a .VSS file without a matching .VSV file in VAST?

To open a .vss/.vsseg file without matching image stack in VAST, simply drag-and-drop it from a file browser onto an empty instance of VAST. VAST will generate an empty image stack with the correct size automatically. However, since .vss/.vsseg files do not store voxel dimensions, VAST will ask you to enter the voxel size in nanometers for correct scaling of the data.

How do I deal with self-touching objects?
If you need to be able to recover the true shape of an object, for example for correct skeletonization or computation of the surface area, places where there are self-touches (for example, a dendritic spine touching the dendritic shaft) can be problematic. The easiest solution is to leave a gap between the two sides, but that is not always feasible, in particular in the Z direction. One way to get around this is by using sub-objects and glue. Just like a plastic model which is constructed from parts, you would make the spine a child of the parent and add 'glue' – a different segment which you treat specially in the analysis – to the interface where parent and child object are actually connected.

How do I make shiny 3D pictures and animations from the segmentations in VAST?
VAST does currently not have 3D rendering capabilities apart from the 3D viewer. You can use

VastTools to extract .OBJ (Wavefront OBJ) model files of objects segmented in VAST (section 8.2). These models can then be loaded into 3D rendering programs (I use Autodesk 3dsMax).

I accidentally pressed 'Home'. How do I get back to where I was just painting?
The 'Coordinates' tool window keeps a temporally ordered list of recently visited places. Select the second entry from the bottom in the drop-down menu list to jump back.

I exported my segmentation as an image stack, but the images are all black. Why?
When exporting segmentations (not screen shots!) as image stacks, the RGB or graylevel values of the pixels encode the internal segment IDs of your segmentation, starting from 1, as explained in section 3.3. Small ID numbers will show up as very dark gray or blue. Load an image into Photoshop or Fiji and normalize the contrast to check the segments.

Suddenly all internal segment IDs changed – What happened?
VAST currently keeps a contiguous list of segment IDs between 1 and n for n segments at all times. This means that if segments are removed from a segmentation, the other segments will 'move up' to keep the list contiguous. This happens when you use the functions '`Delete Segment + Subtree`' or '`Weld Segment Subtree`'. If you are using the internal IDs to identify particular segments and do not want them to change, avoid those functions. For example you can put deleted segments into a 'Deleted' folder and/or re-use them instead of actually deleting them. If you use '`Merge Segmentations in...`' or '`Import Segmentation ...`' with certain settings, the internal segment IDs of the imported segmentation will also be changed so that they don't overlap with existing IDs. You can specify that in the import dialog.

Why is it called 'VAST Lite' and not just 'VAST'?
The 'Lite' in the name emphasizes that this is not supposed to be the final version of the software. It is currently a usable tool with a limited set of capabilities, which is provided to the scientific community without restrictions. Development of VAST continues, and there may at some point be a released version with more features. Also, 'VAST *Lite*' is a better name for Google searches than 'VAST'.

B.2 Typical Use Cases

Analyzing synaptic connectivity in a cortex EM stack
It is possible to define the location of synapses and their synaptic partners by painting the synaptic membrane. We do this in two steps. First, we paint the axons and dendrites with individual colors. Then we generate a second data set (a second segmentation layer) in which just the synapse membranes are labeled, using a pen with fixed size (for example 16 pixels diameter) which is wide enough to overlap with pre- and postsynaptic labels. Make sure that the 3D region painted for each synapse is a single connected component and that different synapses are separate connected components. We then export both data sets as image stacks and use a Matlab script to find each synapse by doing connected-component analysis. For each synapse we then find the axon and dendrite which occupy the same voxels as the synapse in the other data set, which gives us the connectivity information. If axons and dendrites are classified as such either via name or segment hierarchy, we can also extract which side is presynaptic and which side is postsynaptic.

For more advanced approaches, axons, dendrites and synapses may be extracted from the EM images with machine learning algorithms and represented as image or segmentation data, which is then imported into VAST or loaded from external sources for proof-reading. Synaptic connectivity can also be investigated with skeletons, using specific edge types and node proximity to represent synapses.

Counting and classifying objects by painting
Just as for the synapses in the previous example, you can use connected-component analysis in Matlab to count other objects in the stack, for example neuron cell bodies. If you paint all neurons of each type in the same color and use different colors for different types, connected-component analysis can be used to separate the different cell bodies for each type, given that they are separated in space. The connected-component analysis will also give you the number of objects of each type, and their volume, if you count the painted voxels for each connected component.

For small objects like vesicles you can use the Particle Cloud exporter in VastTools to do the connected-component analysis and export a text file with the center locations and volumes of all connected components; see section 8.3.

If you paint objects, for example cell bodies, using a seperate segment color for each, you can then classify them quickly by sorting them into different folders (say, 'Neuron' or 'Glia') by using the 'Collect' tool (see section 4.4.20), or you can use the Control Buttons function to add different text tags to the name of each segment depending on its class ('Append Text to Label', see section 3.1.6). You can quickly and systematically go through large lists of labeled cell bodies using the `keypad +` button.

Masking out a single cell from a confocal light microscopy image stack
VAST can also load image stacks acquired in light microscopy. If a subset of cells with overlapping branches is fluorescently labeled in such an image stack, VAST can be used to generate a Z-projection image which shows only one cell. For this, first generate a 3D mask by painting over all parts of the cell you want to show. Switch off 'Alpha' for the segmentation layer in VAST so the segmentation is invisible. Then use the VastTools function 'Export / Export Projection Image' to generate a Z-projection image of only the segmented regions of the image stack. For this use 'Screenshots' as image source, 'Segmented areas' as opacity source, and 'Additive' as blending mode. You can optimize brightness/contrast and blending of several layers in VAST to tune the Z-projection image.

Tracking objects in a video
If you translate a video into a sequence of images, you can of course import this image sequence as a stack into VAST (even in color). In the same way as you can label three-dimensional structures in VAST, you can label objects or regions or fiducial points as they move through the video. You can then export the labelings as an image stack and analyze locations in the image and movement, or use the Particle Cloud exporter in VastTools to export a text file with fiducial coordinates; see section 8.3.

Defining fiducial points in an unaligned image stack for manually aided alignment
Some EM image stacks are difficult to align with automatic methods, for example if the image quality is low, there is high-contrast background, or the tissue slices have folds. Manually defined fiducial points which should end up in the same place from slice to slice can help improve the alignment. You can load an unaligned stack into VAST and use manual painting to define fiducial points. Use a different color for each feature you are tracking through the stack, so that in your alignment script you can tell which points belong together. Again you can use the Particle Cloud exporter in VastTools to export a text file with fiducial coordinates to help the analysis; see section 8.3.

Proof-reading an automatic segmentation
VAST can load automatic segmentations in two ways, either as image stack or as editable segmentation stack. The second option currently requires that the segmentation has less than 2^{16} (65535 or less) segments.

If the segmentation is loaded as an image stack, a proof-read segmentation can be generated in a separate segmentation layer, using masked painting (see section 4.1.5) and filling operations (section

4.2.2). Boundary maps can also be loaded as image stacks and used to generate a proof-read segmentation (see sections 4.1.4 and 4.2.2).

If the segmentation is loaded as a segmentation layer, it can be edited directly. Split segments can be merged by using the Collect tool (section 4.4.20) and subsequent welding (section 4.4.10). Mergers can be corrected by erasing or recoloring the boundary (for example by adding a color for the to-be-split-off object as a child, and using parent mode (section 4.1) to recolor it where it touches the other object), then use filling to completely split it off (section 4.2.1).[1] Of course the user has full freedom to edit the segmentation simply by painting as well.

Automate workflows with the VAST API and Matlab scripts

The VAST API (section A), which is for example used by `vasttools.m`, allows you to remote-control many functions in VAST and can be very useful to automate otherwise time-consuming tasks. For example, Matlab can read specific subregions of image or segmentation stacks from VAST, process them, and write them back to VAST. In this way VAST can be a data server for Matlab, and the actual data source can be external, so that VAST serves as a cacheing bridge between the external source and Matlab. The API even allows to execute some processing steps in VAST like painting and filling.

It is also possible to have a VAST user initiate a Matlab processing step by pressing a button. To do this, run a Matlab script which polls `getcurrentuistate()` in a loop and checks whether the SPACE bar is pressed. When a space bar press is detected, Matlab can for example construct a URL based on other information it reads from VAST and open a web browser window with Neuroglancer showing the 3D object which is at the mouse pointer location in VAST.

B.3 Some Performance Tips

- If file access is very slow (when you move through the stack and it takes time until the images appear) consider storing the data locally and/or on SSD drives. In particular, especially when using non-SSD drives, put files which you use together on physically separate drives. The problem is often that two files (for example an EM layer and the segmentation layer) are loaded at the same time from the same non-SSD hard drive, which causes the read/write head to jump forth and back between two locations at high speed. This slows down file access a lot.

  If images are loaded from image files (`.VSVI`), loading can be slow if the image tiles are very large, since a stack of whole image tiles has to be loaded even if only a small region is displayed. Consider making the image tiles smaller (1024x1024 or less) to improve performance.

- If you experience a low frame rate (the mouse cursor is jumping rather than moving smoothly), try to reduce the size of the 'Maximal Window Width' in the Preferences to something like 1280. On very large screens you can set the 'Target resolution smaller than' to 4 (Default: 2) to help.

- If VAST slows down considerably after using it for a while, check if the RAM of your computer is full (check the RAM usage indicator in the upper right corner of the VAST main window, see section 3.1.3). Once memory is full, Windows might swap parts of the data to the hard drive, which can slow down processing a lot. To fix that problem tell VAST to use less memory by REDUCING the maximal RAM cache memory sizes in the Preferences dialog. This should not cause problems even if you are working with large image stacks since VAST does not need to load all data in RAM at once. The only effect which reducing the cache memory size should

[1] This requires however that the split-off branch is a single connected component.

have is that VAST may have to load parts of your data set from disk more often, which is slower than reading from RAM.

- For diagnosing low performance it can sometimes be useful to know which function within VAST uses most time. You can enable the profiling graph overlay in the VAST main menu under 'View / Profiling Overlay' to see time consumption for different VAST subsystems over time.

B.4 Setting up VAST with a Wacom tablet

When it comes to fast and accurate digital painting, tablets and in particular tablet screens can improve productivity significantly. We use various Wacom Cintiq screen tablets. While the larger Cintiqs have a better screen, they can be expensive and bulky. I personally use a Cintiq 13HD, which can be laid flat onto a desk and close to the user. A similar newer model would be the Wacom Cintiq 16. There are also many cheaper alternatives from different manufacturers.

An alternative could be tablet laptops, if they fulfill the system requirements for VAST. Not all tablets are ideal, for example if pen input is unreliable or if the pen has no buttons or the buttons cannot be customized.

Wacom tablets come with driver software which lets you configure the pen buttons for each program. For optimal workflow in VAST I find it most useful to have 'erase' on one pen button and 'change tooltip size' on the other. For this it is easiest to set one pen button to 'right click' and the other one to 'middle click'. If that does not work for your system (sometimes Windows uses the Right Click event in a special way, for example), you can configure the pen buttons to simulate equivalent key presses (see Appendix B.5).

If you see brief circular animations when you use the pen and drawing is delayed, you should go to 'Pen and Touch' in the Windows Control Panel and switch off 'Flicks'. On the tab 'Flicks', uncheck 'Use flicks to perform common actions quickly and easily' and click 'Apply'.

In Windows 10, find the checkbox 'Use Windows Ink' in the Wacom driver ('Wacom Tablet Properties' in the Control Panel) and uncheck it. This solves most of the lag problems.

B.5 Keyboard Shortcuts in VAST

You can open a window which lists all the keyboard shortcuts in VAST from the main menu under `Window / Keyboard Shortcuts`. Here is a summary.

<table>
<tr><td>SHIFT or ENTER</td><td>Pick segment color by mouse click</td></tr>
<tr><td>CTRL or INSERT</td><td>Temporarily go to 'move' mode</td></tr>
<tr><td>TAB or \|</td><td>Move mode: Click left and move pen up/down to zoom;
Paint mode: Click left and move pen up/down to change pen diameter</td></tr>
<tr><td>`~ or DELETE</td><td>Paint mode: Erase mode</td></tr>
<tr><td>H or L</td><td>Hides segmentation while held down</td></tr>
<tr><td>U</td><td>Shows only the selected layer while held down (solo)</td></tr>
</table>

Table B.1: Mode Modifiers (hold down)

UP or A	One slice up
DOWN or Z	One slice down
PAGEUP or S	MAXPAINTDEPTH slices up
PAGEDOWN or X	MAXPAINTDEPTH slices down
LEFT or Q	Move or Paint mode: Z-jump up; Annotation mode: Traverse skeleton left (to parent)
RIGHT or E	Move or Paint mode: Z-jump down; Annotation mode: Traverse skeleton right (to child). Hold SHIFT to go to child 2

Table B.2: Slice Navigation

`MAXPAINTDEPTH` is the value set under 'Max Paint Depth' in the 'Drawing Properties' dialog.

-_	Decrease tooltip diameter
=+	Increase tooltip diameter
N or Keypad /	Zoom out
M or Keypad *	Zoom in
F	Flash selected segment
I, O, P	Set paint mode to Paint all, Background, Parent
HOME or G	Go to anchor point of selected segment
,< / .>	Select previous / next segment in recently selected list
Keypad +	Select next segment or annotation object and go to anchor point if available
Keypad -	Cycle 2D view between XY, YZ, and XZ slices
1,2,..0	Trigger 'Control Button' function (see section 3.1.6)
SPACE	Can be used to signal remote programs via the API (see section A.8.5)

Table B.3: Other Controls

B.6 Terms of Usage and Privacy Statement

This version of VAST ('the software') is free of charge and may be distributed freely, but not sold. Commercial usage is allowed.

You are using this software at your own risk. Even though it has been tested extensively, it is not free of bugs. Please keep backup copies of your data.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

VAST does not collect usage statistics or other data. In particular, it does not transmit any of your image or segmentation data anywhere.[2]

This version of VAST is 'AI-free'; its code base was written without use of LLMs like ChatGPT or Copilot.

This software uses `easyzlib.c`, which is based on the `zlib` library by Jean-loup Gailly and Mark Adler, and `libtiff` with Copyright (c) 1988-1997 Sam Leffler and Copyright (c) 1991-1997 Silicon Graphics, Inc. VastTools uses `jtcp.m` by Kevin Bartlett and `loadawobj.m` by W.S. Harwin, University Reading (Matlab BSD license).

[2]Except if you set that up explicitly using the Remote Control API Server of course. You can make it transmit segmentation data through the API to VastTools for example, but that is under your control.

Appendix C

Technical Information

C.1 Size limitations

Maximal file size for image stacks and segmentations: Limited by maximal file size on disk; theoretical maximum 2^{64} bytes.
Largest EM stack that has been imported into VAST so far: $\sim$ 5 Terabytes
Largest EM stack opened remotely (via .vsvr and .vsvi): $\sim$ 1.3 Petabytes

Pixel formats for image stacks: 8-bit graylevel, 24-bit RGB, 32-bit (RGB color-mapped in display), 64-bit (RGB color-mapped in display)

Maximal number of labels supported: currently $2^{16} - 1$ (needs 600 bytes of RAM per label).

Largest image supported (at full resolution): $(2^{31} - 1) \cdot (2^{31} - 1)$

C.2 Supported file formats for importing / exporting

Importing of image stacks

Currently 8 bit graylevel and 24-bit RGB image stacks are supported.

Stacks and tiled stacks:	.JPG, .PNG, .TIF, .BMP
3D Volume files:	.NII (NIfTI); will be converted to 8 bits when imported

Versions 1.1 and later of VAST Lite use `libtiff` for importing from .TIF images, which solves the previous problems caused by the Windows GDI+ TIFF routines.

Exporting of image stacks

Stacks and tiled stacks:	.PNG, .TIF (uncompressed, ZIP, LZW), .RAW, as 8-bit indexed, 24-bit RGB, 32-bit ARGB and 64-bit ARGB

Importing of segmentations

Segmentations can be imported from RGB .TIF and .PNG image stacks. The segment number for each pixel is encoded in the RGB value of the image as follows: Bits 0-7 of the label number are expected in the blue channel, bits 8-15 in the green channel, and bits 16-23 in the red channel. This is the same format used for exporting segmentations to image stacks (see below). Please be aware that VAST can currently only handle 16-bit segmentations.

Exporting of segmentations

When exporting segmentations, the available file formats depend on the range of segment numbers used. For example, if the highest segment number is greater than 255, 8 bit indexed file formats will not be available. In that case the label numbers will be encoded into the color channels (for example for RGB, bits 0-7 of the label number will be put into the blue channel, bits 8-15 into the green channel, and bits 16-23 into the red channel).

Stacks and tiled stacks:	.PNG, .TIF (uncompressed, ZIP, LZW), .RAW
3D Volume files:	.NII (NIfTI) 8 bit only (currently unsupported)

Exporting of screen shots

Stacks and tiled stacks:	.PNG, .TIF (uncompressed, ZIP, LZW), .JPG, .RAW (all 24 bit RGB)
3D Volume files:	.NII (Nifti) 8 bit only (currently unsupported)

Bibliography

[1] Cardona A., Saalfeld S., Schindelin J., Arganda-Carreras I., Preibisch S., et al.: *TrakEM2 Software for Neural Circuit Reconstruction*, PLoS ONE, **7**(6):e38011, (2012), doi:10.1371/journal.pone.0038011.

[2] Saalfeld, S., Fetter, R., Cardona, A., and Tomancak, P.: *Elastic volume reconstruction from series of ultra-thin microscopy sections*, Nature Methods, **9**(7), (2012), 717-720.

[3] Srubek Tomassy, G., Berger, D., Chen, H., Kasthuri, N., Hayworth, K., Vercelli, A., Seung, S., Lichtman, J., and Arlotta, P.: *Distinct Profiles of Myelin Distribution Along Single Axons of Pyramidal Neurons in the Neocortex*, Science, **344**(6181), (2014) 319-324, doi:10.1126/science.1249766.

[4] Kasthuri, N., Hayworth, K., Berger, D., Schalek, R., Conchello, J., Knowles-Barley, S., Lee, D., Vazquez-Reina, A., Kaynig, V., Jones, T., Roberts, M., Morgan, J., Tapia, J., Seung, H.S., Gray Roncal, W., Vogelstein, J., Burns, R., Sussman, D., Priebe, C., Pfister, H., and Lichtman J.: *Saturated Reconstruction Of A Volume Of Neocortex*, Cell, **162**(3), (2015) 648-661.

[5] Morgan, J., Berger, D., and Lichtman J.: *The Fuzzy Logic of Network Connectivity in Mouse Visual Thalamus*, Cell, **165**, (2016) 192-206.

[6] Quadrato, G., Nguyen, T., Macosko, E., Sherwood, J., Yang, S., Berger, D., Maria, N., Scholvin, J., Goldman, M., Kinney, J., Boyden, E., Lichtman, J., Williams, Z., McCarroll, S., and Arlotta, P.: *Cell diversity and network dynamics in photosensitive human brain organoids*, Nature, **545**, (2017), 48-53, doi:10.1038/nature22047.

[7] Berger, D., Seung, S., and Lichtman J.: *VAST (Volume Annotation and Segmentation Tool): Efficient Manual and Semi-Automatic Labeling of Large 3D Image Stacks*, Frontiers in Neural Circuits, **12**, (2018), Article 88, doi:10.3389/fncir.2018.00088.

[8] Fang, T., Lu, X., Berger, D., Gmeiner, C., Cho, J., Schalek, R., Ploegh, H., and Lichtman, J.: *Nanobody immunostaining for correlated light and electron microscopy with preservation of ultrastructure*, Nature Methods, (2018), doi:10.1038/s41592-018-0177-x.

[9] Morgan, J. and Lichtman J.: *An individual interneuron participates in many kinds of inhibition and innervates much of the mouse visual thalamus*, Neuron, **106**(3), (2020), 468-481.e2, doi:10.1016/j.neuron.2020.02.001

[10] Alon, S., Goodwin, D., Sinha, A., Wassie, A., Chen, F., et al.: *Expansion sequencing: Spatially precise in situ transcriptomics in intact biological systems*, Science, **371**(6528), (2020), doi:10.1126/science.aax2656

[11] Witvliet, D., Mulcahy, B., Mitchell, J.,Meirovitch, Y., Berger, D., Wu, Y., Liu, Y., Koh, W., Parvathala, R., Holmyard, D., Schalek, R., Shavit, N.,Chisholm, A., Lichtman, J., Samuel, A., Zhen, M.: *Connectomes across development reveal principles of brain maturation*, Nature, **596**, (2021), 257-261, doi:10.1038/s41586-021-03778-8

[12] Shapson-Coe, A., Januszewski, M., Berger, D.R., et al.: *A petavoxel fragment of human cerebral cortex reconstructed at nanoscale resolution*, Science, in press (2024)