

# Omni: Visualizing and Editing Large-Scale Volume Segmentations of Neuronal Tissue

by

Rachel Welles Shearer

Submitted to the Department of Electrical Engineering and Computer  
Science

in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

© Massachusetts Institute of Technology 2009. All rights reserved.

Author .....  
Department of Electrical Engineering and Computer Science  
May 22, 2009

Certified by .....  
Sebastian Seung  
Professor of Brain and Cognitive Sciences  
Thesis Supervisor

Accepted by .....  
Arthur C. Smith  
Chairman, Department Committee on Graduate Theses



# Omni: Visualizing and Editing Large-Scale Volume Segmentations of Neuronal Tissue

by

Rachel Welles Shearer

Submitted to the Department of Electrical Engineering and Computer Science  
on May 22, 2009, in partial fulfillment of the  
requirements for the degree of  
Masters of Engineering in Computer Science and Engineering

## Abstract

In this thesis, I worked on the OMNI software application for viewing and editing large connectomic image volumes. I designed and implemented the user interface and modules for viewing image volume slices and editing segmentation image volumes. The user interface incorporates the suggestions of connectomics researchers and features multiple viewing windows that can be rearranged without restriction. The user interface also features the OMNI Inspector, a special widget that provides the main interface for interaction with project image volumes and project preferences. The 2D navigation and editing modules use OpenGL textures to display images from large image volumes and feature a texture management system that includes a threaded texture cache. Editing is performed by painting voxels in a viewing window and allows the user to edit existing segment objects or create new ones.

Thesis Supervisor: Sebastian Seung  
Title: Professor of Brain and Cognitive Sciences



# Acknowledgments

- I would like to thank Brett Warne, my collaborator on the OMNI project, for not only tirelessly working on his part of the project but also for finding the time to help me with my work. His advice and guidance made OMNI and this thesis possible.
- I would also like to thank Srinivas Turaga and Daniel Berger for their help, suggestions, and encouragement.
- I also want to thank my supervisor Sebastian Seung for his leadership and infectious enthusiasm for the project.
- Finally, I would like to thank my family and friends for their love and support.



# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction: A new tool for connectomics</b>                        | <b>13</b> |
| 1.1      | An overview of previous applications. . . . .                           | 14        |
| 1.1.1    | SSECRET . . . . .   | 14        |
| 1.1.2    | Validate . . . . .  | 14        |
| 1.1.3    | Reconstruct . . . . .   | 15        |
| 1.1.4    | Knossos . . . . .   | 15        |
| 1.2      | OMNI is customized to meet the needs of researchers in the Seung Lab    | 15        |
| 1.2.1    | Flexible design . . . . .   | 15        |
| 1.2.2    | Large volumes: channels and segmentations . . . . .                     | 16        |
| 1.2.3    | 3d and 2d integration . . . . .   | 17        |
| 1.3      | Overview . . . . .  | 17        |
| <b>2</b> | <b>Handling large volumes</b>   | <b>19</b> |
| 2.1      | An overview of the OMNI pre-processing step . . . . .                   | 20        |
| 2.1.1    | Chunking the volume data . . . . .                                      | 20        |
| 2.1.2    | Handling mipmaps . . . . .  | 21        |
| 2.1.3    | Saving data to disk . . . . .   | 21        |
| 2.2      | The OMNI volume cache is the interface for access to processed data     | 22        |
| 2.2.1    | Pulling data from the cache . . . . .                                   | 22        |
| 2.2.2    | Loading data into the cache . . . . .                                   | 24        |
| 2.2.3    | Strategies for cache access . . . . .                                   | 24        |
| 2.3      | The OMNI texture cache is the interface for access to tile data . . . . | 25        |
| 2.3.1    | Pulling data from the cache . . . . .                                   | 25        |

|          |  |           |
|----------|--|-----------|
| 2.3.2    | Loading data into the cache . . . . .  | 26        |
| 2.3.3    | Strategies for cache access . . . . .  | 27        |
| <b>3</b> | <b>User Interface</b>  | <b>29</b> |
| 3.1      | OMNI design principles . . . . .   | 29        |
| 3.1.1    | Design for simplicity . . . . .  | 30        |
| 3.1.2    | Design for flexibility . . . . .   | 30        |
| 3.1.3    | Notes on the OMNI interface implementation . . . . .   | 31        |
| 3.2      | The OMNI main window . . . . .   | 32        |
| 3.2.1    | The main window guides project interaction . . . . .   | 32        |
| 3.3      | The OMNI Inspector . . . . .   | 34        |
| 3.4      | The OMNI viewing modules . . . . .   | 36        |
| <b>4</b> | <b>The Omni Inspector</b>  | <b>37</b> |
| 4.1      | The Omni Inspector is the primary interface for system interaction .   | 37        |
| 4.1.1    | Browsing the volume tree . . . . .   | 38        |
| 4.1.2    | Preference panes . . . . .   | 39        |
| 4.2      | Changes to data or preferences made in the OMNI Inspector are prop-<br>agated to the OMNI backend and vice versa . . . . . | 40        |
| 4.2.1    | Event management . . . . .   | 40        |
| 4.2.2    | Signals and slots . . . . .  | 41        |
| 4.3      | The Omni Inspector is implemented using the Model/View architecture  | 43        |
| 4.3.1    | Model/View in Qt . . . . .   | 44        |
| 4.3.2    | Sorting and filtering . . . . .  | 46        |
| <b>5</b> | <b>2D Navigation</b>   | <b>47</b> |
| 5.1      | OmView2d is the main viewing module class . . . . .  | 48        |
| 5.1.1    | OmView2d and the texture cache . . . . .   | 49        |
| 5.1.2    | User interaction . . . . .   | 50        |
| 5.2      | OMNI uses OpenGL for 2d display . . . . .  | 51        |
| 5.2.1    | OpenGL texturing . . . . .   | 51        |



|          |   |           |
|----------|---|-----------|
| 5.3      | OpenGL textures are tiled for interactive display . . . . .           | 51        |
| 5.4      | Generating textures . . . . .   | 52        |
| 5.4.1    | Texture colors . . . . .  | 53        |
| 5.4.2    | Texture compositing . . . . .   | 54        |
| <b>6</b> | <b>2D Editing</b>   | <b>57</b> |
| 6.1      | Changing modes in OMNI . . . . .                                      | 57        |
| 6.1.1    | Navigation Mode . . . . .   | 58        |
| 6.1.2    | Editing Mode . . . . .  | 58        |
| 6.2      | Selection . . . . .   | 59        |
| 6.3      | Voxel painting in a 2D viewing window . . . . .                       | 60        |
| 6.3.1    | Qt and painting . . . . .   | 60        |
| 6.3.2    | Bresenham's algorithm . . . . .                                       | 60        |
| 6.3.3    | Voxel edits are propagated through the system . . . . .               | 61        |
| 6.4      | Textures are updated in response to selection changes and edits . . . | 61        |
| 6.4.1    | Selection changes . . . . .   | 62        |
| 6.4.2    | Voxel edits . . . . .   | 62        |
| 6.4.3    | Updating texture content in <code>OmTile</code> . . . . .             | 62        |
| <b>7</b> | <b>Contributions</b>  | <b>65</b> |
| <b>A</b> | <b>System Diagrams</b>  | <b>67</b> |



# List of Figures

|     |   |    |
|-----|---|----|
| 1-1 | Two types of image volumes. . . . .   | 16 |
| 2-1 | Diagram of the chunking process at original level of detail and reduced level of detail. The chunks always have the same dimensions. . . . .  | 21 |
| 2-2 | Slice orientation for an image volume. . . . .  | 23 |
| 2-3 | The volume cache is referred to as <code>OmCachingMipVolume</code> in the OMNI source and contains references to <code>OmMipChunks</code> . The tile cache is referred to as the <code>OmThreadedCachingTile</code> and contains references to texture IDs. . . . . | 25 |
| 3-1 | The main window upon first opening OMNI (cropped for detail). . . .   | 32 |
| 3-2 | One possible layout for a collection of 2D views. . . . .   | 33 |
| 3-3 | The OMNI Inspector displaying the channel properties editor pane. . .   | 35 |
| 4-1 | The hierarchical structure of the OMNI Inspector volume tree. . . . .   | 38 |
| 4-2 | A portion of the OMNI segment object properties dialog. . . . .   | 38 |
| 4-3 | The event posting and retrieval process. . . . .  | 42 |
| 4-4 | General diagram of signal and slot interaction. Adapted from Qt signal and slot documentation. . . . .  | 43 |
| 4-5 | The OMNI volume tree module/view architecture. . . . .  | 44 |
| 4-6 | A diagram of the Qt tree model item structure and the representation of that structure in the OMNI volume tree. . . . .   | 45 |
| 5-1 | OMNI with 3 orthogonal 2D viewing windows. Cursor lines have been modified from original to aid contrast. . . . .   | 48 |

|     |  |    |
|-----|--|----|
| 5-2 | Structure of the OMNI 2D viewing module. Please see Chapter 2 for more information about the texture cache system. Diagram style adapted from Brett Warne. . . . .   | 49 |
| 5-3 | The same image before color mapping and after color mapping. . . .   | 54 |
| 5-4 | The same image viewed as a channel and after texture compositing. .  | 55 |
| 6-1 | The system mode toolbar. . . . .   | 58 |
| 6-2 | The tool selection parts of the OMNI Main Window toolbar. . . . .  | 58 |
| 6-3 | The two types of selection in OMNI. . . . .  | 59 |
| 6-4 | Voxel painting. . . . .  | 60 |
| 6-5 | The Bresenham line algorithm. Image adapted from Wikipedia. . . .  | 61 |
| A-1 | Structure of the OMNI backend, including OmMipChunkCoords (referred to as OmMipCoordinates in the diagram) OmMipChunks, and the OmCachingMipVolume. Attributed to: Brett Warne. A system for scalable 3d visualization and editing of connectomic data. Masters thesis, Massachusetts Institute of Technology, 2009. . . . . | 67 |

# Chapter 1

## Introduction: A new tool for connectomics

The mammalian nervous system contains tangled networks of neurons, which are cells that relay information in the form of electrical and chemical signals. A key research area in neuroscience involves efforts to map out the billions of neurons in brain tissue as a way to more fully understand the function of neurons and the development of the brain. Work in this field, known as connectomics, can be intensely time consuming – a complete wiring diagram of the neurons in the worm *C. elegans* took more than a decade to complete, and *C. elegans* has only 302 neurons. In contrast, the human brain is estimated to contain more than 100 billion neurons.

Researchers map out networks of neurons by inspecting thin cross-sections of brain tissue. The sections are acquired by repeatedly taking a picture of the top plane of a block of brain tissue, each time shearing away a thin slice until the entire block has been imaged. Views of the block of tissue from other angles can then be reconstructed from the images of the slices. By examining tissue from different angles, scientists tracing neural pathways can better understand the shape of an individual neuron.

Currently, researchers map neural pathways by using special editing software to view slices and trace lines – referred to as “countours” – around the boundaries of neurons. By working linearly through a stack of slices viewed from one particular angle, a researcher can determine the shape of a particular neuron as well as the path

it follows through the entire block of tissue. The set of traced contours can then be used to build a model of the neuron's structure.

The OMNI application provides an integrated way for researchers to navigate through large image volumes, trace neural boundaries, and view and edit a three-dimensional model of neuron segments. I worked with Brett Warne, another student in the M.Eng program, to build OMNI. While I designed the user interface and developed the 2D navigation and editing modules, Mr. Warne build much of the backend and the 3D model viewer and editor.

This section will review existing applications that perform similar tasks to OMNI, and present the motivation for building a brand new application.

## **1.1 An overview of previous applications.**

There are several existing software tools used by connectomics researchers to perform volume navigation and segmentation. Each offers different features, but none of them offer all the features that OMNI does.

### **1.1.1 SSECRET**

One software application used by connectomics researchers is called SSECRET, developed by Kitware, Inc. It allows slice viewing and navigation of large image volumes at high resolution. The design has a client-server architecture that allows the user to quickly scale slice images without lag or loading time because focusing speed depends on bandwidth and not image size. SSECRET also supports skeleton tracing. However, SSECRET does not support contour tracing or 3D segmentation visualization.

### **1.1.2 Validate**

Validate is a tool implemented at the Seung Lab using MATLAB. Validate allows slice navigation and also displays previously traced contours and skeletons. The user can

look down three orthogonal axes of a block of tissue and see the path of a particular neuron as represented by contours displayed in a particular color. Generated 3D models can also be displayed in Validate and will be colored appropriately to match the 2D contours. However, Validate does not scale to large image volumes and does not support any editing.

### **1.1.3 Reconstruct**

Another tool is called Reconstruct. Reconstruct is a slice viewer and 2D editor which allows navigation between slice images and provides basic image editing tools for drawing contours or skeleton points on top of slice images. However, Reconstruct does not support large image volumes, does not allow the user to view orthogonal views of image data, and does not support 3D segmentation visualization.

### **1.1.4 Knossos**

Knossos is a scalable 2D viewing system, supporting large image volumes and basic skeleton point editing. Knossos does not support contour tracing or 3D segmentation visualization.

## **1.2 OMNI is customized to meet the needs of researchers in the Seung Lab**

Almost all of the previously discussed software applications are used by connectomics researchers in the Seung Lab, the group that supervised the OMNI project. However, lab researchers expressed dissatisfaction with the existing applications and suggested various ways that OMNI could improve the status quo.

### **1.2.1 Flexible design**

A lot of researchers stated that they did not like the user interfaces of the existing connectomics software applications, which they felt were awkward and difficult to

learn. They expressed a desire for a software application that they could customize for their individual working styles. This is especially critical when it comes to contour tracing, when a researcher might be performing the same repetitive operation over a long period of time. Researchers felt that a good user interface would make this process less tedious.

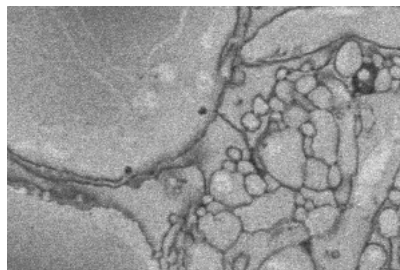
### 1.2.2 Large volumes: channels and segmentations

The Seung Lab produces a lot of very large image volumes, and most of them can't be viewed with existing connectomics software. There are two main types of image volumes:

**Channel image volumes** depict basic anatomical data. They are always grayscale images. A portion of a channel image volume is depicted in Figure 1-1(a).

**Segmentation image volumes** depict the results of contour tracing around the boundaries of neurons. A colored segmentation volume is depicted in Figure 1-1(b). Each color in that segmentation image represents one *segment object* – the boundaries of a neuron.

Segmentation image volumes are not actually colored by default. Instead, they are stored as bitmaps, each pixel containing an index value that maps to a particular segment object. All of the pixels within one segment object will have the same index value in the bitmap.



(a) Channel image



(b) Segmentation image

Figure 1-1: Two types of image volumes.



Connectomics researchers need an application that can display large volumes of channel and segmentation data. They also require an application that can edit segmentation data: creating segment objects, adding more pixels to a given segment object, and removing pixels from a segment object.

### 1.2.3 3d and 2d integration

It can be difficult to trace the path of a neuron through a large 3D volume of data. If researchers only look at 2D slices of an image volume at one particular angle, they can make mistakes while tracing contours because neurons may not run uniformly parallel or orthogonal to the image slice. Researchers in the Seung Lab requested a 2D viewing application that offers three orthogonal views of image data, allowing them to see a particular neuron from more viewpoints.

It is also important to researchers that the 3D view of segment objects be synchronized with the 2D view. This feature becomes especially important when performing error correction – an error in a tracing that is not obvious when looking at only three orthogonal views will become very apparent when viewed in 3D.

## 1.3 Overview

**Chapter 2** presents the preprocessing system developed to overcome the difficulty of reading large image volumes from memory

**Chapter 3** describes the OMNI user interface and the principles guiding its design.

**Chapter 4** introduces the OMNI Inspector, the primary interface for interacting with data volumes and an OMNI project.

**Chapter 5** presents the 2D navigation module.

**Chapter 6** introduces the 2D editing module.

**Chapter 7** contains a summary of my contributions to the OMNI project.



# Chapter 2

## Handling large volumes

A fundamental feature of the OMNI system is its ability to process large image volumes. As discussed earlier, researchers in connectomics will often work with volumes that are multiple gigabytes in size, and sizes are likely to increase dramatically in the future with new imaging techniques. Large volumes present a problem for any viewing application because images are unlikely to fit in video memory, and therefore the standard image viewing pipeline does not apply. OMNI addresses the large volume problem in two ways:

- Image volumes are pre-processed prior to viewing. The pre-processing step divides each large image volume into several smaller image volumes which are then saved to disk. The smaller image volumes can henceforth be accessed for viewing because they are small enough to fit in video memory.
- Once requested, image data is stored in a cache that allows faster and more efficient future access. There are actually two caches – one that manages disk access, and contains references to buffered raw image data, and another cache that manages OpenGL texture memory and contains references to textured image data in video memory. The caches are used in sequence when viewing images.

## 2.1 An overview of the OMNI pre-processing step

My collaborator Brett Warne designed and developed a large part of the OMNI pre-processing system, but I will present a brief overview here. Essentially the pre-processing system must be run on an image volume prior to viewing, and it saves data that it creates in a separate place without modifying the original data. The process of splitting each image into several smaller images is called "chunking" and may take several hours depending on the size of the original data. Running the pre-processing step is called "building" the volume data.

The build system handles image data slightly differently depending on the original image type. Channel images, which contain only anatomical data, are processed as pure image data. Segmentation images, which contain additional information, are processed as pure image data and also as specifications for building a three-dimensional mesh. The process of building mesh data is more complicated than the process of building image data, and will not be discussed here.

### 2.1.1 Chunking the volume data

After the user specifies the location of the volume data on disk, the build system runs through each file sequentially. Each image file is read into memory and copied back onto disk as a collection of square chunks of a size specified by the user. For example, if the user has specified that chunks should be of size 128 pixels by 128 pixels, a 512 pixel by 512 pixel original image would be divided into 16 smaller images. This scenario is depicted below. If the original image cannot be evenly divided into smaller chunks, blank space is added as appropriate. This means that there might be some chunks that only have a part of the original image on them.

The chunks are written back onto disk using a hierarchical directory format, and the VTK and HDF5 libraries are used here to facilitate reading and writing images.



(a) Image in 16 chunks



(b) Image in 4-chunk  
mipmap

Figure 2-1: Diagram of the chunking process at original level of detail and reduced level of detail. The chunks always have the same dimensions.

### 2.1.2 Handling mipmaps

The build system also creates multiple mipmaps – versions of the original image with reduced levels of detail. For example, after the 512 x 512 original image discussed earlier is divided into 16 smaller 128 x 128 images in the first step, it is downsampled to be a 256 x 256 image. This downsampled image is then divided into 4 chunks, still of size 128 x 128. Each of those 4 mipmapped chunks contain original data at a reduced level of detail, as depicted in Figure 2-1.

Creating mipmaps in the pre-processing phase makes it more efficient to view large images at multiple zoom levels. When the user zooms out far enough when viewing the original image, OMNI will replace that image with the appropriate mipmap image. So instead of drawing 16 textures, OMNI would only need to draw 4 textures.

### 2.1.3 Saving data to disk

Chunked data is saved during the build process into a special HDF5 file with a `.omni` extension. This special file contains a hierarchical directory structure wherein image chunks at different mip levels are saved inside different folders. The `.omni` file also serves as the project file for an OMNI project.

## 2.2 The OMNI volume cache is the interface for access to processed data

OMNI maintains a cache that allows access to chunked data created by the build system. This central cache is the interface through which all other modules of the OMNI application request image data from disk. The cache utilizes a shared pointer system that makes sure references to chunked data are not deleted while other parts of the application are still using them.

For a detailed diagram of part of the OMNI backend including the OMNI volume cache, please see Figure A-1.

### 2.2.1 Pulling data from the cache

When part of the OMNI application needs raw image data, it requests that data from the cache using a coordinate known as a `OmMipChunkCoord`. In OMNI, an `OmMipChunkCoord` represents the location of a given chunk in the data volume and is defined by four integers: mip resolution, x, y, and z coordinates. An `OmMipChunkCoord` is specified as shown below:

$$[\text{<mip level>}, (\text{<x-coordinate>}, \text{<y-coordinate>}, \text{<z-coordinate>})]$$

So the `OmMipChunkCoord` `[0 (1, 0, 0)]` represents the chunk at mip level 0 and x-coordinate 1. Because `OmMipChunkCoords` are zero-indexed, the example coordinate actually refers to the second chunk along the x-axis and the first chunk along the y-axis and z-axis. In this way, the `OmMipChunkCoord` format is a reflection of the chunking process that happens during a build. Through applying a series of transforms defined after the build process, any three-dimensional pixel position in the image volume can be converted to the particular `OmMipChunkCoord` that contains the data at that position.

After being provided a valid `OmMipChunkCoord`, the volume cache returns a shared pointer to a chunk. The chunks are represented internally as `OmMipChunks`. Once an

OMNI module has been provided a `OmMipChunk`, it can request a pointer to an allocated array of raw image data from that chunk. This process is known as “extracting a data slice.”

It is important to note that, up until this point, image volumes have been handled as three-dimensional structures. But for a module to request a data slice from a `OmMipChunk`, it must specify the alignment of that data within the entire image volume. Because OMNI can be used to view three orthogonal views of image data, there are three possible ways that images can align with the volume. These alignments are known as *planes*.

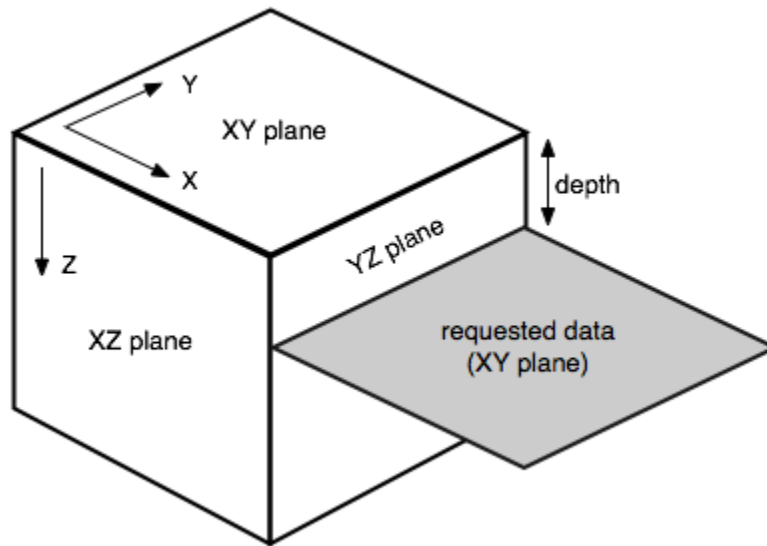


Figure 2-2: Slice orientation for an image volume.

The three possible planes are orthogonal to each other: **XY**, **XZ**, and **YZ**, as depicted in Figure 2-2. The module must also specify the depth at which the requested plane intersects the volume. After a module receives a pointer to a data slice, it can begin to interact with that slice as a two-dimensional array of raw image data that contains only the requested data at the requested orientation.

### 2.2.2 Loading data into the cache

When an OMNI module provides an `OmMipChunkCoord` and requests an `OmMipChunk`, the volume cache checks to see if it already contains a reference to it. If it does, it simply returns a shared pointer to that chunk. If not, then a “cache miss” has occurred. The volume cache handles the cache miss by using the given `OmMipChunkCoord` to construct a new `OmMipChunk` that points to the appropriate chunk location on disk, and adds a reference to the new `OmMipChunk` to the cache.

When the volume cache grows bigger than its allotted size, it deletes references to `OmMipChunks` that have not been recently requested. So references to `OmMipChunks` that are requested frequently are more likely to be in the volume cache than references to `OmMipChunks` that are requested rarely.

### 2.2.3 Strategies for cache access

Constructing new `OmMipChunks` in response to volume cache misses is a more expensive process than simply returning references to existing `OmMipChunks`. For this reason, OMNI modules try to reduce requests for `OmMipChunks` that aren’t already referenced by the cache. This strategy becomes very important for image viewing, when repeated requests for data slices from `OmMipChunks` are being made in order to generate textures.

The OMNI 2d viewing module is designed to minimize the time it takes to access slice data by prioritizing requests for slice data within one `OmMipChunk` at a time. Effectively, this means that the module will make requests for data slices at all depth levels for one `OmMipChunk` before requesting access to another `OmMipChunk`. The 2d viewing module will also ask the volume cache if it contains a particular `OmMipChunk` before actually requesting access – if the `OmMipChunk` is not already referenced by the cache, the 2d viewing module might try to request data from an `OmMipChunk` already referenced by the cache first.



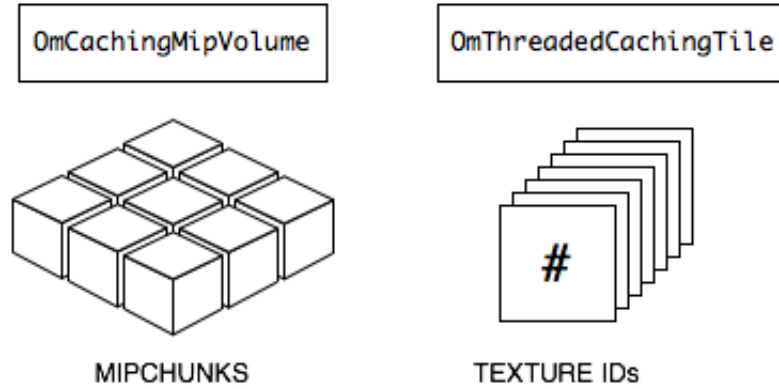


Figure 2-3: The volume cache is referred to as `OmCachingMipVolume` in the OMNI source and contains references to `OmMipChunks`. The tile cache is referred to as the `OmThreadedCachingTile` and contains references to texture IDs.

## 2.3 The OMNI texture cache is the interface for access to tile data

The OMNI 2d viewing module maintains another cache separate from the volume cache, known as the tile cache. The tile cache allows access to OpenGL textures that have been generated from raw image data.

Every texture in OpenGL is identified by a unique ID generated when the texture is created. References to these unique texture IDs are stored in the tile cache and returned upon request to the OMNI 2d viewing module, which can use them to access textures stored in video memory.

For a detailed diagram of the structure of the OMNI 2d viewing module, please see Figure 5-2 in Chapter 5.

### 2.3.1 Pulling data from the cache

When the OMNI 2d viewing module needs to draw a particular texture, it requests the texture ID from the tile cache using a coordinate known as a `OmTileCoord`. Like `OmMipChunkCoords`, `OmTileCoords` represent a location in the data volume and are defined by four numbers: mip resolution, `x`, `y`, and `z` coordinates. However, *unlike* `OmMipChunkCoords`, the `x`, `y`, and `z` coordinates in `OmTileCoords` refer to absolute

points in space, not the location of `OmMipChunks`. The three x, y, and z coordinates in an `OmTileCoord` are floating point numbers between 0 and 1.0, not integers. Given a pixel position in the image volume, an `OmTileCoord` is determined by applying a series of transforms defined after the build process.

After being provided a valid `OmTileCoord`, the tile cache returns a shared pointer to a texture ID. The texture IDs are represented internally as `OmTextureIDs`. Each `OmTextureID` object keeps track of a texture ID and the size of the corresponding texture in video memory. If the texture has been generated from a segmentation image, an `OmTextureID` also keeps track of the ID numbers of all of the segments contained in that image for use during segment editing. For more information on segment editing, refer to Chapter 6.

### 2.3.2 Loading data into the cache

When the OMNI 2d viewing module provides an `OmTileCoord` to the tile cache and requests an `OmTextureID`, the tile cache checks to see if it already contains a reference to it. If it does, it simply returns a shared pointer to the `OmTextureID`. If not, the tile cache handles the cache miss by using the given `OmTileCoord` to request raw image data from the volume cache.

The coordinates of the `OmTileCoord` are transformed to an `OmMipChunkCoord`, which is provided to the volume cache in return for a `OmMipChunk` from which raw image data can be requested (in the process described in Section 2.2.3). The raw image data is then supplied to OpenGL so that a texture can be generated, and the corresponding texture ID is used to construct a new `OmTextureID` object.

Unlike the volume cache, the tile cache features *threaded* access to the `OmTextureIDs` stored on disk. Once a request is made for a specific `OmTextureID`, the tile cache launches a thread to retrieve it. A threaded caching system is necessary because it allows OpenGL to continue drawing even when a texture ID is not immediately available. This is especially important when a texture needs to be generated from scratch, which involves requesting raw data from disk.

When the texture cache is full, it deletes references to `OmTextureIDs` that have not

been recently requested *and* deletes the corresponding textures from video memory. It is important to note that the size of the texture cache is specified by the user, and doesn't necessarily reflect the size of video memory. If the user under-specifies the size of the cache, textures might be deleted even though there is remaining room in video memory. Conversely, if the user over-specifies the size of the cache, OMNI might run out of video memory altogether. Control of the cache is left to the user – and the cache size can be redefined dynamically – because it gives the user more control over the performance of the OMNI 2d viewing module.

### **2.3.3 Strategies for cache access**

Generating new textures in response to tile cache misses is an expensive process, especially when the appropriate `OmMipChunk` is not currently contained in the volume cache. The threading feature of the tile cache compensates for this by making sure that the slow process of generating textures does not affect drawing or repaint speed. If a requested texture is not available immediately, the OMNI 2d viewing module will draw other textures instead of waiting for the texture to be generated. For a longer description of this process, see Chapter 5.



# Chapter 3

## User Interface

The OMNI user interface was designed to take advantage of user familiarity with current connectomics software tools while at the same time offering distinct usability and accessibility improvements for the expert users of the system. Because OMNI is a tool used to interact with large volumes of data, a responsive, efficient user interface is incredibly important.

### 3.1 OMNI design principles

The most significant challenge when designing the OMNI interface involved organizing the large number of requested features into a simple, elegant interface. There were a large number of desired features that didn't easily fall into established interface patterns. For example, users wished to be able to view different slices of an image volume sequentially – many suggested modeling the interface for such a task after the Photoshop interface for flipping back and forth between different layers of an image. This interface is very familiar and could be easy for users to adapt to, but how would that interface function when the data volume contains hundreds or even thousands of slices?

But developing a new interface entirely, without relying on established models and metaphors, would eventually require more user education and a longer learning period. A large part of the challenge of designing OMNI was the balancing act

between *simplicity* and *flexibility* – OMNI needed to be an application that is natural to learn and use, while at the same time allowing expert users the chance to totally customize their working environment.

### 3.1.1 Design for simplicity

The easiest approach to designing the OMNI user interface would have involved adapting the established interface patterns from SSECRET, Validate, Reconstruct, and Knossos because users were already familiar with them. However, a single application with roots in three different interfaces would have been inelegant and unwieldy.

In an effort to make the transition to OMNI easier, I decided to adapt the aspects of user interface that most previous applications shared – the three orthogonal 2D viewing windows and separate 2D and 3D viewing modules. This fundamental design decision makes OMNI immediately recognizable to users of the other systems, and reduces user memory load (in accordance with Jakob Nielsen’s usability heuristics [1]).

I further relied on the principle of simplicity when deciding on the placement of user options. The buttons for system mode changes are always visible and always display their current status to the user. Because OMNI can be run in two modes – navigation mode and editing mode – and because users can perform any number of actions within those modes, keeping the system mode status always visible is an efficient way to reduce user error.

### 3.1.2 Design for flexibility

Another challenge when designing the OMNI user interface involved making the program adaptable for expert users. While some users might only be using the application for a little while and require an interface that promotes simplicity, expert users need a system with shortcuts that will make their work more efficient.

A large amount of the negative feedback I received from researchers about current connectomics software tools involved time-consuming processes that they believed

could be improved by adding shortcuts. For example, expert users expressed a preference to use keyboard commands to complete frequent tasks, such as flipping between slices of an image volume or viewing segments belonging to different segmentation images. But there is a limited set of keys available and complicated keyboard shortcuts are difficult to remember. The more keyboard shortcuts the application has, the heavier the burden on both new users to learn them, and experienced users to remember them. This illustrates the challenge of designing a user interface for expert users.

In an effort to balance the different needs of new users and expert users, I decided to leave a lot of the decisions about layout and window size up to the user's personal preference. OMNI is designed to have a completely flexible, responsive window system that users can modify to suit their personal work style and methodology. Any component of the OMNI main window can be moved, resized, or popped out of the main window altogether. A new user might feel more comfortable leaving the windows in the default configuration, while an expert user has the ability to move the windows into exactly the way he or she desires.

Another OMNI feature useful for new and expert users alike is the undo/redone scheme. Any editing action, whether it be adding text to an annotation or painting a new segment, is added to the undo history and can be undone or redone at the click of a button. This feature allows all users to explore the system and make mistakes without consequences for their future work.

### **3.1.3 Notes on the OMNI interface implementation**

The OMNI interface is implemented using the Qt toolkit, a complete application and UI development framework. Qt was chosen because of its cross-platform support and wide range of available UI widgets (user interface objects), as well as its availability under the LGPL open-source license.

## 3.2 The OMNI main window

The OMNI main window is the home base for the user. It displays toolbars and menu options at the top and serves as the preliminary container for the viewing modules and the inspector module (discussed in Section 3.3). The toolbars are discussed in more detail in Chapter 6.

From the main window menu, a user can open an existing project, start a new project, and save a current project.



Figure 3-1: The main window upon first opening OMNI (cropped for detail).

### 3.2.1 The main window guides project interaction

All OMNI windows are initially docked inside the main window container when they are opened for the first time. As stated before, any OMNI window can be moved around within the main window container, placed inside other windows for tabbed viewing, and even popped out of the main window altogether. This allows the user to completely customize their window configuration for their needs. A sample configuration is depicted in Figure 3-2.

Brett Warne and I decided to follow the *project* interaction model for OMNI interaction because the actions of opening, creating, and saving a new project are familiar to many users. An OMNI project contains one data volume, which itself can contain multiple channel image volumes or segmentation image volumes. A data volume is understood to represent a single region of tissue that has been photographed



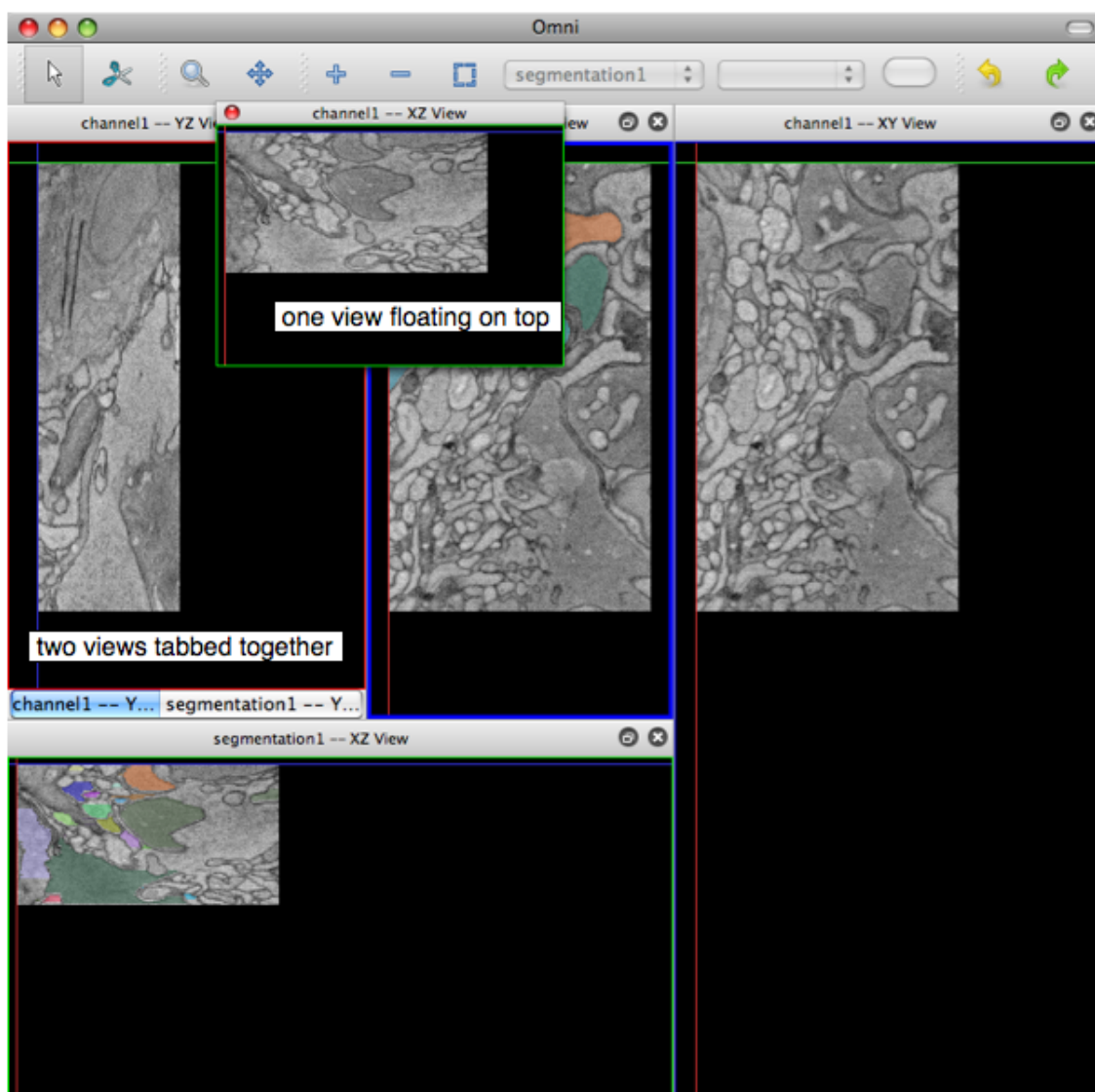


Figure 3-2: One possible layout for a collection of 2D views.

to produce the channel image volumes and segmented to produce the segmentation image volumes.

So, each channel or segmentation image volume depicts the same physical region of tissue – for example, a rabbit retina might have been imaged to produce one channel volume and two segmentation volumes. Because these rabbit retina image volumes all depict the same object, they are contained in a single data volume and thus a single OMNI project.

The volume/channel/segmentation hierarchy is fundamental to the OMNI system design and guides the user interface of the main window as well as the Omni Inspector module.

### **3.3 The OMNI Inspector**

The OMNI Inspector is essentially the OMNI project management dashboard. When a project is created or an existing project is opened, the OMNI Inspector opens automatically inside the main window.

Rather than directing project management actions through the main window menu bar and submenus like many other applications, OMNI features the OMNI Inspector as a separate window entirely – it can be moved around within the main window or popped outside completely, just like the viewing modules. Placing the OMNI Inspector inside its own window module means that the OMNI Inspector can be perpetually visible if the user desires – or disappear entirely. Thus OMNI does not force a particular project management method on the user, and allows them to interact with the OMNI Inspector according to their personal work preferences.

The project data volume contents are displayed in a tree structure, elements of which can be expanded or collapsed to display their contents. When an element of the volume tree is selected, the viewing pane on the right displays the properties of that particular selection. So, when a channel is selected in the tree, the channel properties editor displays in the right-hand pane as depicted in Figure 3-3. There are three possible property editors: channel editors, segmentation editors, and segment

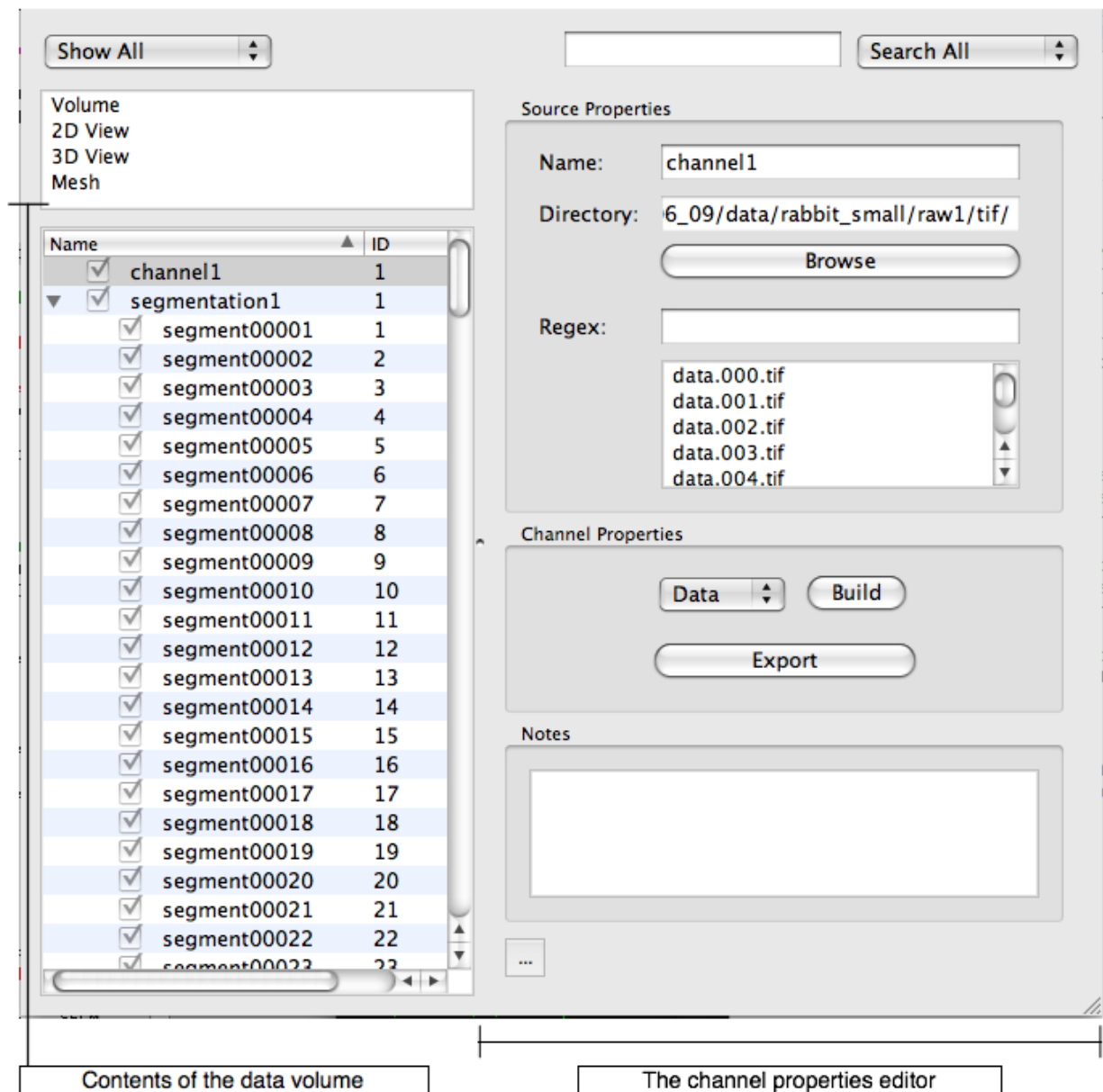


Figure 3-3: The OMNI Inspector displaying the channel properties editor pane.

object editors.

The right-hand pane also displays preference editors, selected by clicking on the preference names (“Volume,” “2D View,” etc) displayed in the list above the tree. The right-hand pane can be completely collapsed by the user if he or she would prefer to look at the volume tree by itself.

Right-clicking on a channel or segmentation listed in the volume tree opens a contextual menu that lists the various 2D viewing options for that particular channel or segmentation and allows the user to open a particular 2D view directly. Segmentations can be overlaid on top of channels by selecting both of them and right-clicking. For more information on the Omni Inspector, please see Chapter 4.

## 3.4 The OMNI viewing modules

As stated before, a key feature of the OMNI viewing modules is that they can be rearranged without restriction within the main window as well as popped out to form floating windows. They continue to seamlessly interact with each other regardless of their particular configuration.

There are three possible 2D views for each channel image volume or segmentation image volume, following the three orthogonal planes of a cube: XY view, XZ view, or YZ view. The contextual menu displayed upon a right-click in the Omni Inspector displays these three options, and the user can select one to open that particular view. The user may open as many instances of a particular view as he or she desires, and all instances of the same view update to reflect movement and zooming. For more information on interactions between 2D views, please see Chapter 5.

The user can also open a 3D view by clicking the “Open 3D View” menu option under the “Window” menu bar. 3D views can be rearranged and configured just like 2D views.

# Chapter 4

## The Omni Inspector

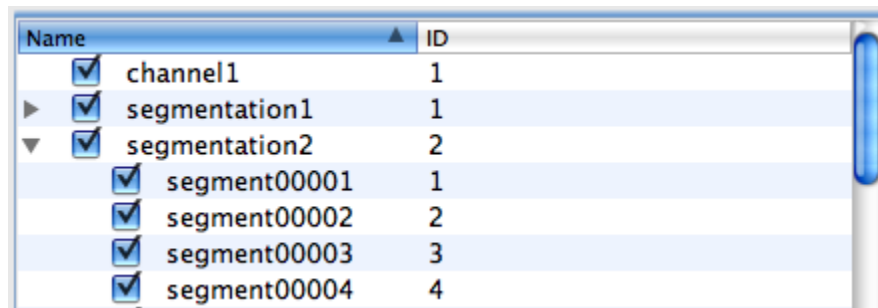
As previously stated, the OMNI Inspector is the OMNI project management dashboard. It displays the contents of the current project volume in a hierarchical tree structure and allows the user to add new data to the project or edit current project contents. Every time the OMNI Inspector is opened, it populates itself with data from the OMNI backend and continues to update itself to reflect system changes.

### 4.1 The Omni Inspector is the primary interface for system interaction

From the OMNI Inspector the user can:

- Browse the contents of the project data volume
- Add a new channel or segmentation image volume
- Edit the properties of channel and segmentation image volumes
- List all segment objects belonging to segmentation image volumes
- Open the OMNI 2d viewing modules
- Edit the project preferences
- Select a segment object for viewing or editing

### 4.1.1 Browsing the volume tree



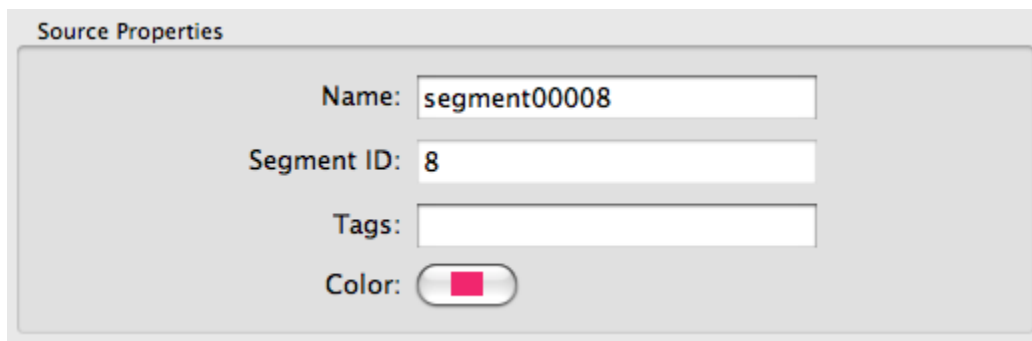
| Name  | ID |
|---|----|
| <input checked="" type="checkbox"/> channel1        | 1  |
| ▶ <input checked="" type="checkbox"/> segmentation1 | 1  |
| ▼ <input checked="" type="checkbox"/> segmentation2 | 2  |
| <input checked="" type="checkbox"/> segment00001    | 1  |
| <input checked="" type="checkbox"/> segment00002    | 2  |
| <input checked="" type="checkbox"/> segment00003    | 3  |
| <input checked="" type="checkbox"/> segment00004    | 4  |

Figure 4-1: The hierarchical structure of the OMNI Inspector volume tree.

The OMNI Inspector volume tree contains a node for each channel image volume and a node for each segmentation image volume. Segmentation nodes contain child nodes representing segment *objects*, which are displayed when the segmentation node is expanded, as depicted in Figure 4-1.

Clicking on a node in the tree causes a change in the pane on the right-hand side of the OMNI Inspector. The right-hand pane contains a properties dialog for the active node in the tree, and refreshes automatically with data from the backend upon activation.

Within the properties dialogs for channel nodes and segmentation nodes, the user can change the name of the node, specify the directory where the original image volume is located, specify a regular expression for the image files, and build chunked data (as discussed in Chapter 2). There is also a notes field where the user can add annotations.



**Source Properties**

Name:

Segment ID:

Tags:

Color:

Figure 4-2: A portion of the OMNI segment object properties dialog.

Within the properties dialog for segment nodes, depicted in Figure 4-2, the user can change the name of the segment object, view the segment object ID number (which is a read-only field), add tags, and pick a new segment object color.

Any change to a text field in a properties dialog is added to a local undo queue and can be undone or redone when that particular text field is in focus.

### **4.1.2 Preference panes**

The OMNI Inspector also allows the user to change global OMNI preferences. When an item from the list above the volume tree is selected (refer to Figure 3-3), a preference dialog opens on the right-hand side of the OMNI Inspector. The preference dialog contains the current user preferences loaded from the preferences list stored in the OMNI backend.

From the volume preferences dialog, the user can add a channel or segmentation image volume to the project, set the chunk size, and add annotations. Properties of the image volume such as pixel resolution and data extent are also displayed in the volume preferences dialog.

The 2D and 3D viewing module and Mesh preference dialogs allow the user to set preferences specific to each view. For example, the 2D preference dialog gives the user the opportunity to specify the size of the volume cache and the tile cache, and to specify the number of textures he or she would like the cache to request ahead of time. In the 3D preference dialog, the user can adjust the field of view and the clipping plane distance, and use the Mesh preference dialog to adjust how the segment object meshes display in the 3D view.

## 4.2 Changes to data or preferences made in the OMNI Inspector are propagated to the OMNI backend and vice versa

When a change is made in the Omni Inspector, that change is propagated to the OMNI backend and other modules are notified so that they can update. Similarly, when another module makes a change to OMNI data, the change is again propagated and the OMNI Inspector updates. Most changes, however, will be made through the OMNI Inspector because it is the primary interface for interacting with OMNI data.

News of a change in system data or preferences becomes available to OMNI modules through use of the OMNI events framework and the Qt signals and slots framework.

### 4.2.1 Event management

The OMNI event framework consists of a collection of event objects that derive from the abstract Qt `QEvent` class. These events can be posted by any OMNI module, but are only received by OMNI modules that are explicitly listening for them. When a listener module receives an event, it can identify from the event type the kind of change that was made to the OMNI backend – and, thus, the module can determine the actions it should take to update itself. An event might also contain specific functions that give listener modules additional information about how the system was changed.

Events are managed by the `OmEventManager` class, which dispatches events to OMNI modules that have registered as listeners with the `OmEventManager`. The `OmEventManager` can dispatch events in two ways:

**Sending the event.** When an event is *sent*, it is processed by the Qt event loop immediately and returns to the originating module only after any listener modules have processed the event. It is useful to *send* events that require an immediate



response from listener modules, such as events spawning dialog boxes or error messages.

**Posting the event** A *posted* event is added to a queue for dispatch and processing at a later time. The next time the Qt event loop runs, it dispatches all queued events, compressing some together in the interest of avoiding multiple identical events. Events can be *posted* that don't depend on an immediate response, such as paint events or preference update events.

### A brief example

A scenario describing an event being posted and received follows and is depicted in Figure 4-3.

1. The user presses "W" - the keyboard shortcut for viewing the next image down in the image stack - while inside a 2D viewing window.
2. The 2D viewing module makes a call to the OMNI state manager indicating that a change in the current slice depth must occur.
3. The state manager requests that the OMNI event manager post a view event that indicates the view center (the intersection of all of the orthogonal views) has changed.
4. The event manager posts the event to the Qt event dispatch queue.
5. All 2D viewing modules listening for view events receive a view event indicating that the view center has changed.
6. The 2D viewing modules update their views.

### 4.2.2 Signals and slots

The two OMNI user interface modules – the OMNI Inspector and the Main Window – communicate with each other using a *signals and slots* system unique to Qt. Signals are emitted by user interface modules in response to changes in state. Slots are normal

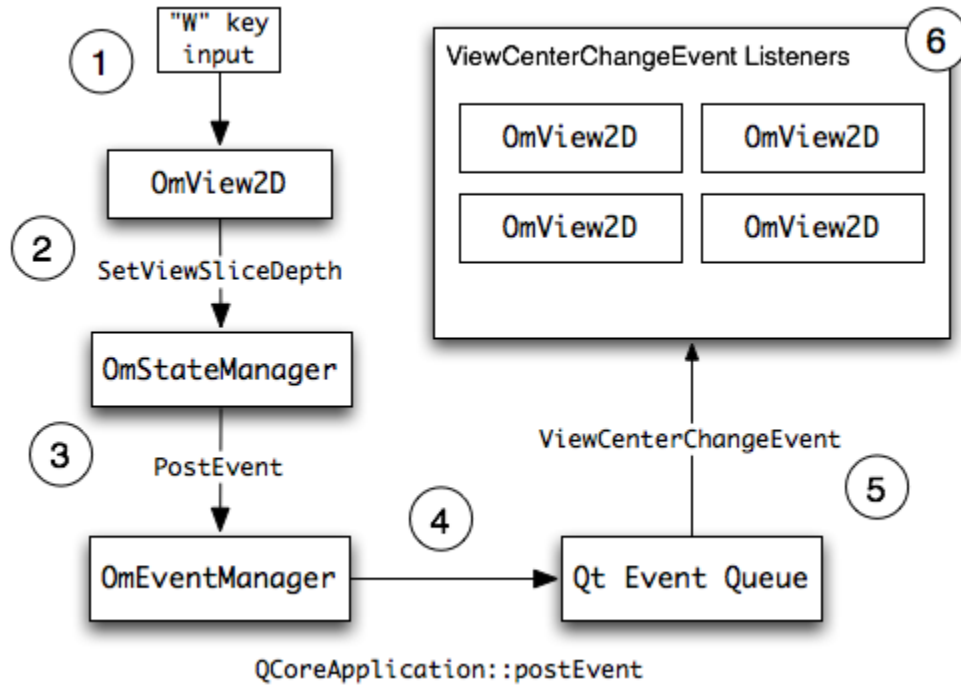


Figure 4-3: The event posting and retrieval process.

member functions that may be called in response to a particular signal or directly called outside of an emitted signal.

As depicted in Figure 4-4, signals and slots are explicitly connected to one another. After a signal has been emitted, any slots connected to that signal are executed immediately, independent of the Qt event loop, and the emitting module waits to execute further instructions until all of the slots have returned. Multiple signals can be connected to a single slot, and one signal can be connected to multiple slots. OMNI modules that emit signals do not know which other modules have slots connected, and slots do not know which signals they have been connected to. This loose coupling between modules that emit signals and modules that execute slots ensure that OMNI interface modules remain largely independent of one another.

In the OMNI application, most of the signal and slot connections are within the Qt widgets that compose the Omni Inspector and between the Omni Inspector and the Main Window. This ensures that the OMNI user interface modules respond quickly to each other and the viewer has a seamless experience.

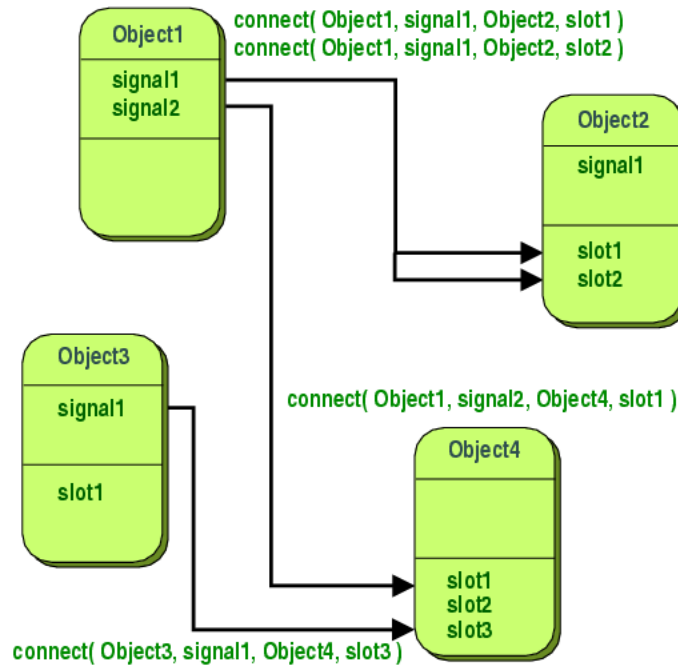


Figure 4-4: General diagram of signal and slot interaction. Adapted from Qt signal and slot documentation.

### 4.3 The Omni Inspector is implemented using the Model/View architecture

In an effort to keep the underlying data representation in the OMNI backend separate from the way it is presented in the OMNI Inspector, I used the Qt Model/View classes to build the Omni Inspector, the architecture of which is depicted in Figure 4-5. This ensured that I could make changes to the way the volume tree displays nodes without having to change the way the data is structured.

The model/view design pattern is commonly used when building user interfaces, as it provides a framework that separates data from presentation. In the model/view architecture, the *model* provides an interface to the data source so that the *view* never has to access data directly.

### 4.3.1 Model/View in Qt

In Qt, the interface between model and view is accomplished by the model providing *model indexes* – references to individual data elements – to the view. In this way the view can retrieve data without directly interacting with the underlying data structure.

Qt provides a series of standard abstract classes for the model and the view that communicate with each other using model indexes and the signal and slot mechanism. By inheriting from the abstract model class, one can create a custom model that interacts with any data structure. The Qt abstract view classes implement list, tables, and hierarchical tree views that use model indexes to determine how data should be displayed.

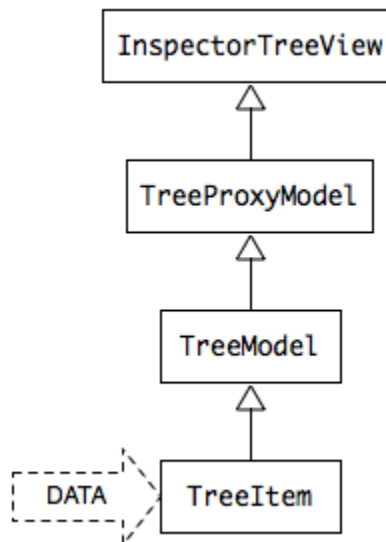
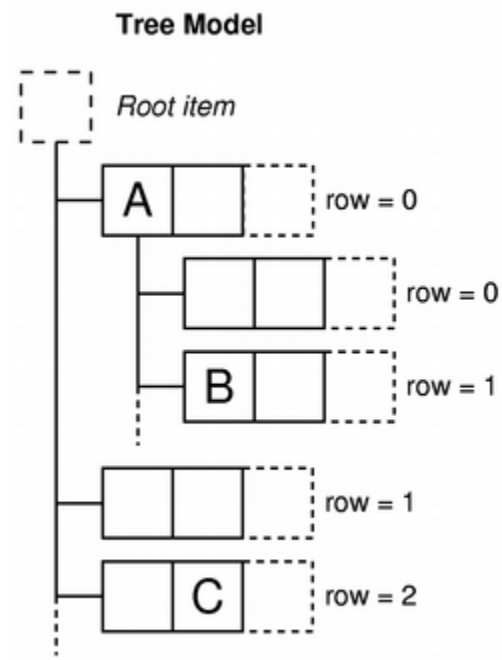


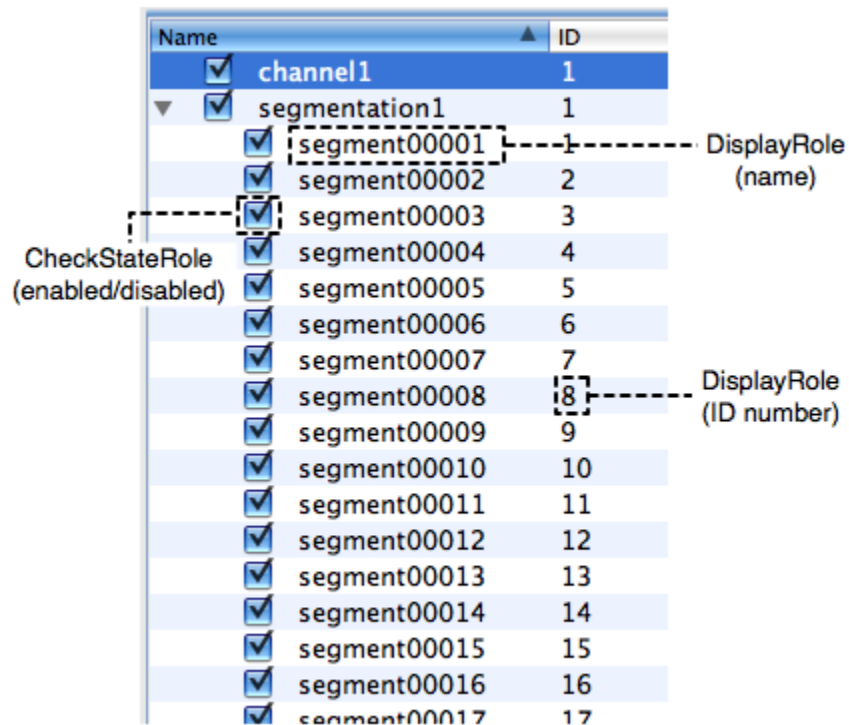
Figure 4-5: The OMNI volume tree module/view architecture.

In the OMNI Inspector, the volume tree is composed of a tree model and a tree view. Tree models are distinguished from other types of models in that they contain model indexes that can parent one another, imitating the hierarchical parent-child structure of a tree view. Data elements in the model are referred to by *row* and *column* number and, if necessary, the model index of the parent element. Model data elements are depicted in Figure 4-6(a).

Any individual element in the model can be associated with many different kinds



(a) Tree model. Diagram adapted from the Qt model class documentation.



(b) OMNI tree model

Figure 4-6: A diagram of the Qt tree model item structure and the representation of that structure in the OMNI volume tree.

of data. Qt uses the concept of an *item role* to enable a model element to provide different kinds of data depending on the needs of the view. In the OMNI inspector tree model, a data element (known as a *tree item*) can be associated with a channel image volume, segmentation image volume, or segment object – as such, a tree item needs to contain information to enable the model and view to distinguish different tree items from each other. Each tree item (depicted in Figure 4-6(b)) contains a name, the item type of the node (channel/segmentation/segment), whether the item is enabled or disabled, an ID number, and item annotations.

### 4.3.2 Sorting and filtering

Because the Model/View architecture of the OMNI Inspector volume tree view separates display from the underlying data, sorting or filtering the tree becomes an easy task. But instead of altering the tree view class to sort data, Qt uses the idea of a *proxy model* to arrange the data within the model itself. This enables multiple views to share a single model without requiring each view to implement its own sorting or filtering operations. From the perspective of a view, a proxy model acts exactly like a regular model, except that it maps data items from their original locations to new locations depending on whether a sort or a filter is activated.

The OMNI volume tree proxy model implements sorting and filtering operations. A user can sort the volume tree alphabetically, in which case the proxy model reverses the order of the model index rows associated with each tree item. When a user performs a filter (using the search box shown at the top of Figure 3-3), certain tree items are removed from the tree view entirely depending on whether their item roles match the search criteria.

# Chapter 5

## 2D Navigation

The OMNI 2D navigation module controls the 2D viewing windows that allow users to examine slices of channel and segmentation image volumes. The navigation module is designed to support fast access to data from large volumes and supports dynamic panning and zooming.

Each 2D viewing window is associated with a particular orthogonal view and source image volume or *combination* of source image volumes, specified with a contextual menu in the OMNI Inspector (see Chapter 4 for more information). One 2D viewing window can display images from:

- One channel image volume
- One segmentation image volume
- Multiple channel image volumes overlaid on top of one another
- One segmentation image volume overlaid on top of one channel image volume

All 2D viewing windows are synchronized with regard to zooming. For example, if a user zooms in on the image in one 2D viewing window, all other 2D viewing windows will also zoom in. 2D viewing windows displaying the same orthogonal view are synchronized with regard to panning as well as zooming.

A feature of the OMNI 2D viewing windows is the presence of a cursor that indicates the position of orthogonal image slices relative to one another. This allows

the user to look at one 2D viewing window and determine the positions of images displayed in other, orthogonal viewing windows. This feature is depicted in Figure 5-1.

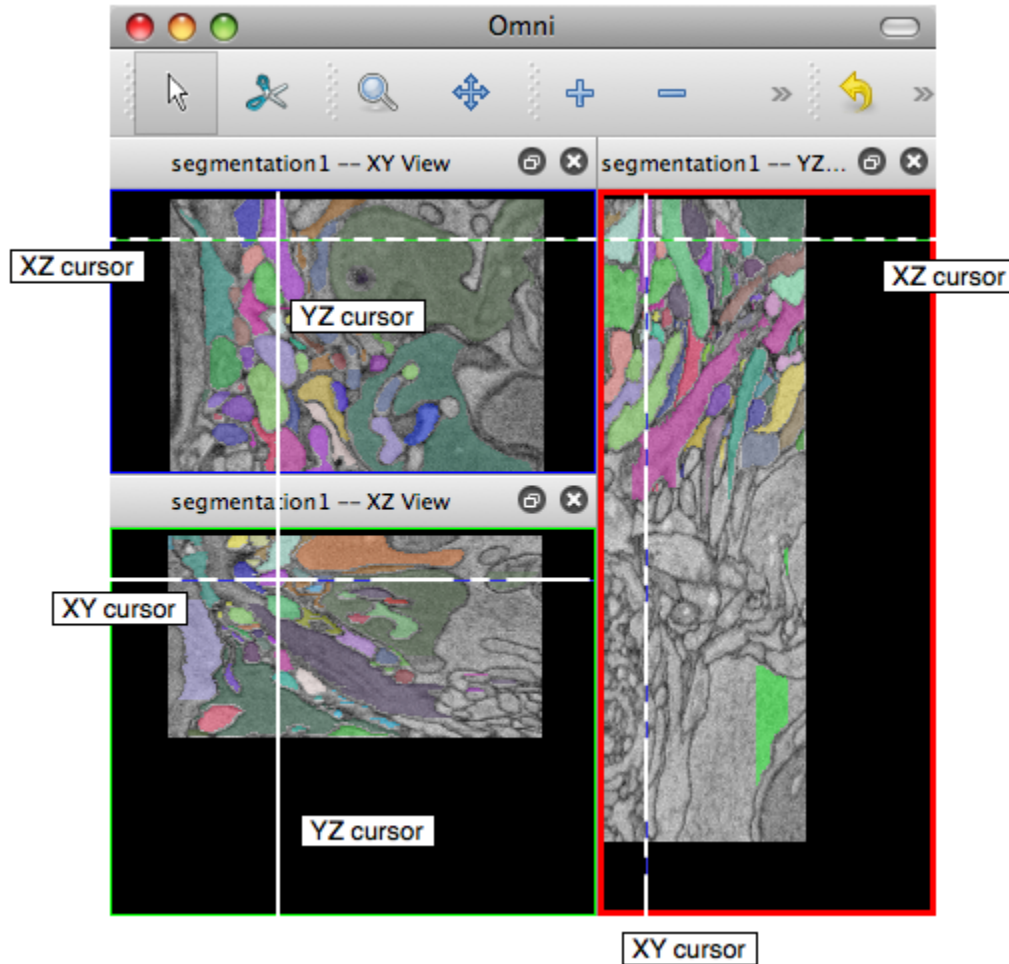


Figure 5-1: OMNI with 3 orthogonal 2D viewing windows. Cursor lines have been modified from original to aid contrast.

## 5.1 OmView2d is the main viewing module class

The `OmView2d` class manages all 2D navigation and editing, including texture cache management, window painting, and user interaction.

Each time the user opens up a 2D viewing window from the OMNI Inspector, they specify the source image volume as described previously. The OMNI Inspector



then signals the Main Window to populate a child window with a new instance of the `OmView2d` class, sending it the ID numbers of the associated image volumes.

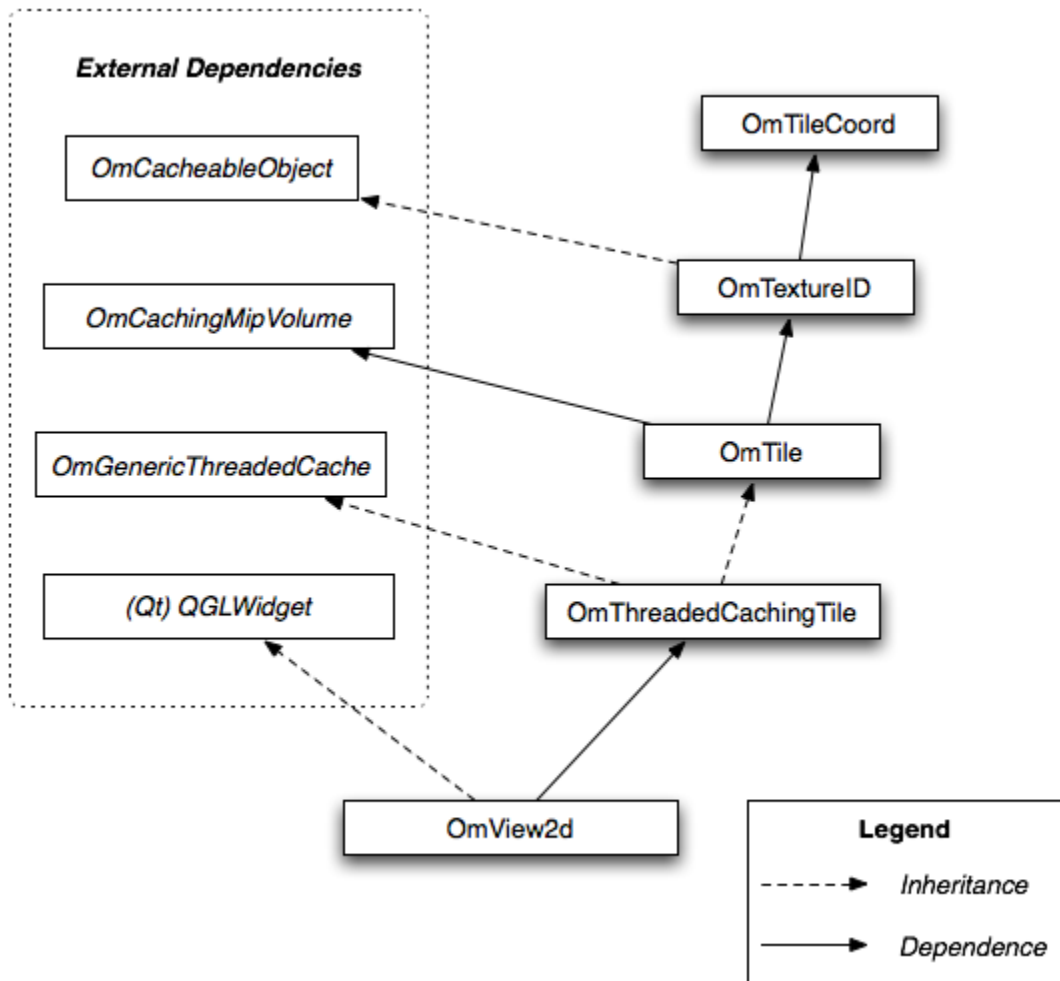


Figure 5-2: Structure of the OMNI 2D viewing module. Please see Chapter 2 for more information about the texture cache system. Diagram style adapted from Brett Warne.

### 5.1.1 `OmView2d` and the texture cache

After instantiation, the `OmView2d` class initializes a new `OmThreadedCachingTile` texture cache, defining the size of the cache according to the user's stated preference and passing the cache the ID numbers of the the associated image volumes. Then, any time a window repaint is indicated, `OmView2d` uses the current window size,

chunk dimension, and pan and zoom state to determine a series of `OmTileCoords` that indicate – in absolute spatial coordinates along with a mip level – the location in the image volume of the images to be drawn. It then requests the `OmTextureID` object associated with each `OmTileCoord` from the texture cache.

If the texture cache contains a reference to the appropriate `OmTextureID` – meaning that a texture has already been created – then it returns that reference. `OmView2d` then determines the OpenGL texture ID, requests the texture from OpenGL and paints it in the window.

If the texture hasn’t been created yet, and the texture cache does not contain a reference to the `OmTextureID`, then a texture must be allocated with the appropriate image data. This process is described in Section 5.4.

### 5.1.2 User interaction

Any user interaction – key presses, mouse clicks, or mouse movement – inside of a Qt widget that has focus sends key events and mouse events. The `OmView2d` class is a Qt widget and thus can listen for these interaction events.

By pressing the arrow keys and clicking or dragging the mouse, the user can pan across an image displayed in an OMNI viewing window. Other keys<sup>1</sup> have other effects:

**The “W” key** will display an image one slice deeper in the source image volume

**The “S” key** will display an image one slice back in the source image volume.

**The “+” key** zooms in on the image.

**The “-” key** zooms out from the image.

Once `OmView2d` receives an interaction event, it posts a view event to the event manager, following the process described in Section 4.2.1. Other 2D viewing windows are listening for these events and will update their current display if necessary. In this

---

<sup>1</sup>Of course, all of these key bindings can be changed according to user preference.

way, the synchronized navigation described at the beginning of this chapter occurs across all open viewing windows. Information about current images displayed is also written to the OMNI backend, so that new viewing windows will display images at the correct location in the image volumes.

## 5.2 OMNI uses OpenGL for 2d display

Both the OMNI 2D and 3D modules use the OpenGL API to display images and meshes. OpenGL is widely known for its use in 3D graphics applications, but it also offers high performance when displaying 2D images.

Qt allows developers to easily integrate OpenGL rendering into a Qt application. Any widget that needs to display OpenGL graphics can inherit from a special Qt widget called `QGLWidget` that allows the use of standard OpenGL rendering commands. In the OMNI 2D navigation module, `OmView2d` inherits from the `QGLWidget`.

### 5.2.1 OpenGL texturing

When using OpenGL to render images, any image data that will be painted on a viewing window is stored in video memory as an OpenGL *texture* – that is, an array of pixel data in video memory. The basic OpenGL texture pipeline involves passing OpenGL a reference to raw image data. OpenGL then allocates space in video memory and uploads the data as a texture, returning a texture ID number. From then on, that data can be painted onto the screen directly from video memory – and accessing video memory is much faster than reading image data from disk.

## 5.3 OpenGL textures are tiled for interactive display

Because OMNI image volumes are often too large to fit entirely in video memory, they are divided into chunks during the pre-processing step described in Chapter 2. The

`OmMipChunks` produced by the build phase are then the basic volumes from which the 2D navigation module produces 2D OpenGL textures. These textures are seamlessly tiled across the screen in order to recreate an orthogonal slice of the original image volume.

Using tiled images allows the 2D navigation module to reduce the number and size of textures it has to create or access in video memory at any one time, which enables the viewing windows to be more responsive to user input. As the user pans around an image, `OmView2d` creates and display textures as required and deletes textures that have not been recently displayed when the texture cache becomes full. Textures can also be created ahead of time (to a degree specified by the user in the 2D preferences) and referenced in the cache for future use.

When a viewing window is painting, occasionally the texture corresponding to part of the image is not available. This can occur when a viewing window has just been opened and textures have yet to be created, or when the texture cache becomes full and the least-recently accessed textures have to be deleted. `OmView2d` will paint as many textures as it can during a window paint, skipping over regions that are missing textures. After the correct texture has been created, the texture cache will raise an event that signals a redraw of the `OmView2d` viewing window.

## 5.4 Generating textures

The `OmThreadedCachingTile` texture cache inherits from an `OmTile` class. This `OmTile` class is responsible for responding to the the “cache misses” that occur when a texture corresponding to a given `OmTileCoord` has not yet been created. When handling a cache miss, the `OmTile` class must request raw image data from the volume cache and create an OpenGL texture from that image data.

It may seem contradictory that the `OmTile` class can call OpenGL functions to create textures and generate texture IDs without inheriting from `QGLWidget`. However, this behavior is allowed because of OpenGL context sharing.

An OpenGL context is basically a wrapper for OpenGL state information – the

current texture ID, the current color, and so on. Normally, textures created in one OpenGL context can't be accessed by a different context, and each viewing window has a different OpenGL context so textures aren't inadvertently shared between different views. But, the texture caches associated with each viewing window need to be able to create textures that can later be accessed by the viewing window during painting.

The solution is to create a new OpenGL context that *shares* with the original OpenGL context. Two shared contexts can access the same resources. So when a viewing window is instantiating its `OmThreadedCachingTile` texture cache, it passes its current OpenGL context to the texture cache, and the texture cache creates a second context that shares with the `OmView2d` context. Before the texture cache asks the `OmTile` class to create or retrieve a texture, it makes its context the current OpenGL rendering context – thus ensuring any future OpenGL calls inside `OmTile` operate within that shared context. Therefore, `OmTile` can create, retrieve, or delete the same textures that the `OmView2d` viewing window has access to.

The `OmTile` class requests image data from the volume cache by supplying a `OmMipChunkCoord` as described in Chapter 2. If the viewing window is set to only display images from a single channel image volume, then `OmTile` can directly provide this raw image data to OpenGL for texture creation. Otherwise, `OmTile` must figure out the appropriate image coloring or combine that image data with other raw image data in a process called *texture compositing* before it can create a new texture.

### 5.4.1 Texture colors

As stated in Chapter 1, segmentation image volumes are bitmaps that contain traced segment objects differentiated from each other by their indexes. In the raw image data, this means that pixels belonging to one segment object might all be designed by the index value 100, while pixels belonging to a different segment object would be designed by the index value 39. A pixel that doesn't belong to any segment object is designated by the value 0. A raw segmentation image bitmap is provided in Figure 5-3(a).

During the pre-processing build step, OMNI gathers information about the segment objects described within the segmentation image volume, and creates a mapping between newly generated segment object ID numbers and the original segmentation image volume index values. So a pixel that has the index value 39 might be mapped to a segment object with the ID number 3. All other pixels in the segmentation image volume with the index number 39 would be mapped to the same segment object 3.

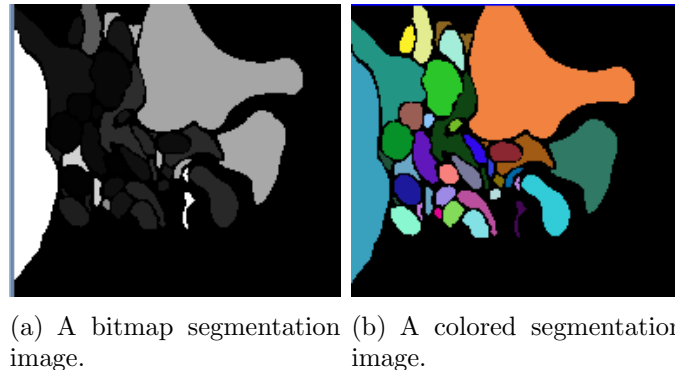


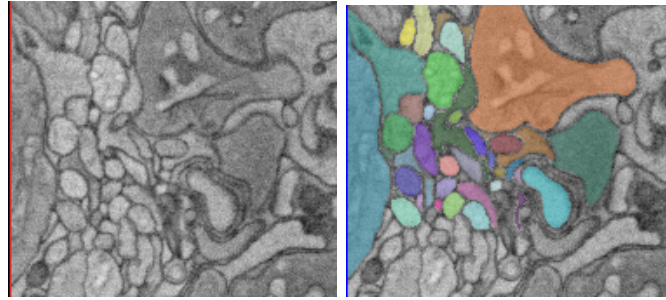
Figure 5-3: The same image before color mapping and after color mapping.

During the build step, a random color is also assigned to each segment object. The ID numbers and colors assigned to a segment object can be viewed in the segment object properties pane of the OMNI Inspector, shown in Figure 4-2.

When the `OmTile` class is creating a texture from segmentation image data, it needs to create a full-color RGBA buffer containing the appropriate color for each pixel of each segment object, as illustrated in Figure 5-3(b). This is accomplished by looping through each of the pixels of segmentation image data and querying the OMNI backend for the segment ID number mapped to each raw image index value. The segment ID number can then be used to query the OMNI backend again, this time for the appropriate segment object color. The color data is added to the buffer, and the loop continues until the end of the original segmentation image data.

### 5.4.2 Texture compositing

If the user has specified that a viewing window should display a combination of image volumes, then the `OmTile` class has additional work to do before it can create a



(a) A channel image.

(b) A channel image composited with a segmentation.

Figure 5-4: The same image viewed as a channel and after texture compositing.

texture. If two or more images need to be combined, then `OmTile` must loop through each of the pixels of the raw image data and combine the data appropriately.

For a segmentation image volume overlaid on top of a channel image volume, the `OmTile` class can combine segmentation image data with channel data *at the same time* as it colors the segmentation image data as described in the last section. At a given pixel, the appropriate segment object color is blended with the channel image data according to an alpha blending function. The value of *alpha* determines how transparent the color will be, and is supplied by the user in the 2D preferences in the OMNI Inspector. If the segment object ID of a particular pixel is determined to be 0, there is no segment object data for that pixel and the channel image data is shown without being blended. A composite texture is provided in Figure 5-4(b).

For multiple channel images overlaid together, corresponding pixels from each raw channel image are averaged together to create a full-color RGB buffer. The hue of each channel image volume can be specified by the user in the OMNI Inspector.





# Chapter 6

## 2D Editing

The OMNI 2D viewing windows support editing: painting new voxels on the screen to correct previously-created segmentations or to create a new segment object entirely. If the user is correcting a segment object, then the color of the paint matches the color of that segment object. If the user is creating a new segment object, then they can pick the color themselves using the OMNI Inspector.

### 6.1 Changing modes in OMNI

In an effort to support panning, zooming, and painting, OMNI was designed to operate in two different modes: navigation mode and editing mode. The two modes allow OMNI to reuse mouse events and key events, which have different effects depending on the current mode. Reusing these events improves usability – the user does not have to keep track of a lot of different key combinations in order to differentiate whether an action is for navigation or editing.

The toolbars can also be rearranged or popped out of the main window entirely, allowing the user to customize their environment for navigation or editing.

The user changes modes in OMNI by clicking on one of the mode icons in the Main Menu system mode toolbar, depicted in Figure 6-1. The cursor icon represents navigation mode, and the scissor icon represents editing mode.



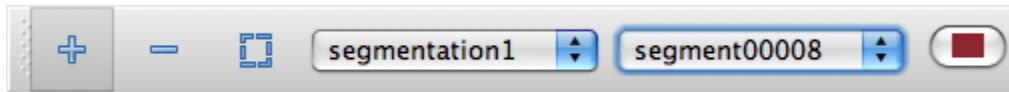
Figure 6-1: The system mode toolbar.

### 6.1.1 Navigation Mode

If the user selects navigation mode, symbolized by the cursor icon on the left, then they can select from one of the two tool icons depicted in Figure 6-2(a). The magnifying glass icon enables zoom mode, in which the user can use their mouse instead of key combinations to zoom in and out of an image. The icon on the right represents the default pan mode, in which the user can use their mouse to pan around an image.



(a) Navigation mode toolbar.



(b) Edit mode toolbar.

Figure 6-2: The tool selection parts of the OMNI Main Window toolbar.

### 6.1.2 Editing Mode

If the user selects edit mode, symbolized by the scissor icon on the right of the system mode toolbar in Figure 6-1, voxel painting is enabled. The user can then select from the buttons and combo boxes depicted in Figure 6-2(b) to determine the type of painting they wish to do. The “+” icon changes to **Add Voxel Mode**, where painting actions add voxels to a segment object. The “minus” icon changes to **Subtract Voxel Mode**, where painting actions will subtract voxels from a segment object. The next icon on the right changes to **Selection Mode**, where the user can make edit selections using the mouse.

The combo boxes in the edit toolbar allow the user to select the segment object they are editing by name, and the color button displays the color of that segment

object. Only one segment object can be edited at a time.

## 6.2 Selection

The OMNI editing module introduces the idea of *selection* – the process by which a user can click inside a viewing window and see the selected segment object highlighted, as depicted in Figure 6-3(a). The user can click on any segment object listed in the OMNI Inspector or shown in a viewing window and see the selection propagate through the OMNI system. For example, a user who selects multiple segment objects inside the OMNI Inspector volume tree will see those segment objects highlighted in the 2D viewing windows and the 3D viewing window. Conversely, a user who selects an object inside a 2D viewing window will see the selection propagate to the OMNI Inspector volume tree and the 3D viewing window.

Selection can be used simply to identify the locations of segment objects relative to each other, but it also is used for voxel editing. A segment object that has been selected becomes a candidate for editing. Even though multiple segment objects can be selected, only one segment object can be edited at a time.

A selection manifests differently in a 2D viewing window depending on the current system mode. In navigation mode, a selected object is colored opaque yellow by default (though this is a preference that can be set by the user). In edit mode, the selected object loses transparency and becomes opaque, remaining its original color. The change in color and opacity means that the user can look at a viewing window and immediately identify the selected segments and the current system mode.

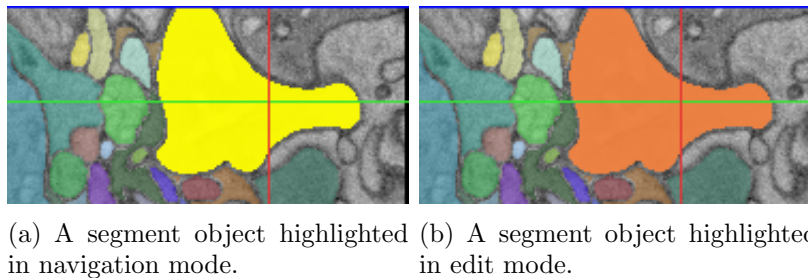


Figure 6-3: The two types of selection in OMNI.

## 6.3 Voxel painting in a 2D viewing window

The painting interface is very simple, and any user who has ever used a drawing program will be familiar with it. To paint, the user simply clicks their mouse and drags it across the viewing window, coloring voxels as the mouse passes over them. The result of voxel painting is depicted in Figure 6-4. The segment object that is being edited is colored dark red, and matches the paint color. Note that the user has zoomed in on the image, so the individual pixels available for painting are clearly visible. However, painting can be performed at any zoom level.



Figure 6-4: Voxel painting.

### 6.3.1 Qt and painting

The OMNI 2D editing module follows the position of the user's mouse by listening for Qt's mouse events. Qt sends `mouseMoveEvents` while the mouse is being moved that indicate the position of the cursor at regular intervals. Cursor position is calculated relative to the viewing window, so it is a simple operation for the editing module to compute whether the voxels to be colored are actually inside an image.

### 6.3.2 Bresenham's algorithm

Once the editing module has a pair of cursor coordinates, it can calculate the voxels on the line between them that need to be colored. This is accomplished using the Bresenham line algorithm, which determines all of the points that compose an

approximation of a straight line between a starting point and an ending point. The Bresenham algorithm is illustrated in Figure 6-5.

After using Bresenham’s algorithm to calculate a series of voxels, the editing module posts a voxel event to the event manager, passing the coordinates of the voxels to be edited and following a similar process to the one described in Section 4.2.1.

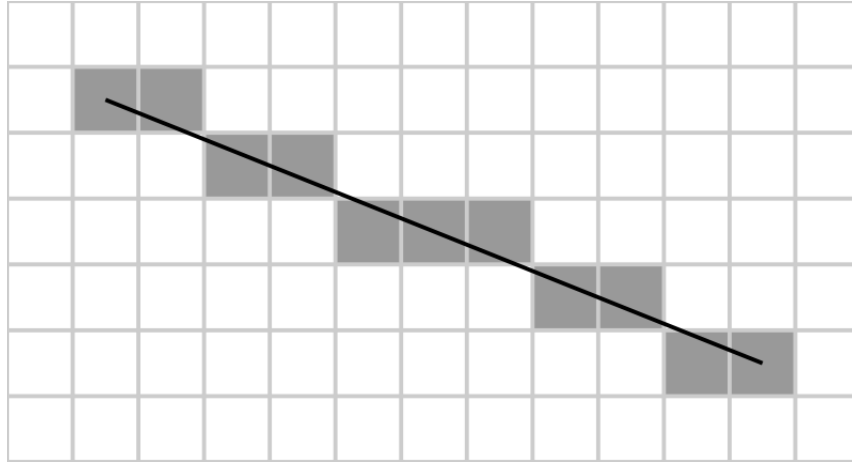


Figure 6-5: The Bresenham line algorithm. Image adapted from Wikipedia.

### 6.3.3 Voxel edits are propagated through the system

Other modules listening for voxel events – knowing the current segment object being edited and receiving voxel coordinates with the event – change their display to reflect the results of an edit. In an `OmView2d` viewing window, this requires updating textures to reflect new segment object colors.

## 6.4 Textures are updated in response to selection changes and edits

When the user changes the value of alpha in the 2D preference pane, selects a new segment object, or makes a voxel edit, those changes propagate to the 2D viewing window. Because the viewing window is displaying a collection of textures stored

in video memory, the navigation module must alter the image data of one or more textures in order to change the display.

All of the possible display changes are propagated as events to `OmView2d`, which can determine from the event the kind of change that was made.

### 6.4.1 Selection changes

Selection changes affect entire segment objects, and so the segment object ID numbers of newly selected and deselected segments are included with every selection change event.

Each `OmTextureID` object includes a list of the segment object ID numbers of the segment objects contained in the texture it references (which might be 0, if the texture contains image data from a channel image volume). So, when `OmView2d` is requesting texture IDs from the texture cache during a window painting operation, it can check which segment objects are contained in each texture. If a texture contains segment objects that have been edited, then `OmView2d` sends a request to the texture cache to update the content of that texture.

### 6.4.2 Voxel edits

Voxel edits affect only particular voxels, not whole segment objects. So, only the coordinates of the edited voxels within a segmentation image volume are sent with a voxel event.

While `OmView2d` is requesting texture IDs from the texture cache during a window painting operation, it can calculate whether the coordinates of edited voxels are contained within the texture it is requesting. If they are, then `OmView2d` sends a request to the texture cache to update the content of that texture.

### 6.4.3 Updating texture content in `OmTile`

When the texture cache receives a request to update the content of a particular texture, it can use a special OpenGL function to copy new data to the texture in video

memory. This function, `glTexSubImage2D`, is much faster than allocating memory for a new texture.





# Chapter 7

## Contributions

The goal of the OMNI project was to design and develop a system that enables 2D and 3D viewing, navigation, and editing of large connectomic image volumes. In this section, I review my contributions to the OMNI project.

**I designed a user interface for OMNI** that incorporates the suggestions of expert users in a design that is simple enough for new users to learn, but flexible enough for expert users to use efficiently.

**I developed the OMNI Inspector module**, the interface through which users interact with the image volumes in an OMNI project. The OMNI Inspector interacts with other modules through events, signals, and slots so that it always displays a current representation of the system state.

**I designed and implemented the 2D viewing module**, which provides efficient display of large image volumes and incorporates multiple synchronized viewing windows.

**I designed and implemented the 2D editing module**, which allows users to edit segmentation image volumes through painting voxels in 2D windows.

**I implemented the OMNI texture cache**, which manages OpenGL textures for the 2D viewing module.



# Appendix A

## System Diagrams

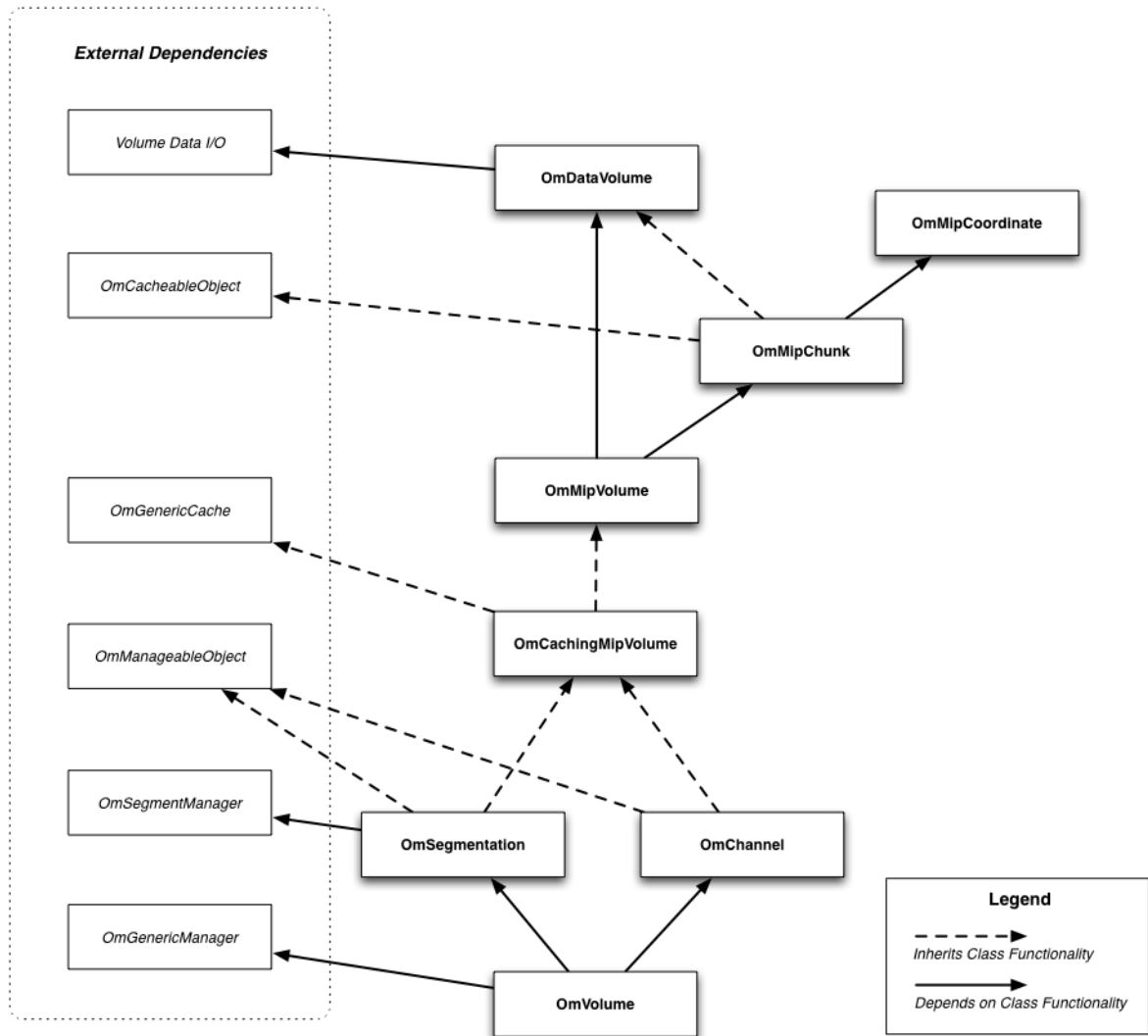


Figure A-1: Structure of the OMNI backend, including OmMipChunkCoords (referred to as OmMipCoordinates in the diagram) OmMipChunks, and the OmCachingMipVolume. Attributed to: Brett Warne. A system for scalable 3d visualization and editing of connectomic data. Masters thesis, Massachusetts Institute of Technology, 2009.

# Bibliography

- [1] Jakob Nielsen. “Ten Usability Heuristics”  
[http://www.useit.com/papers/heuristic/heuristic\\_list.html](http://www.useit.com/papers/heuristic/heuristic_list.html)